STORAGE METHODS FOR FAST ACCESS TO LARGE CARTOGRAPHIC DATA COLLECTIONS - AN EMPIRICAL STUDY

Andreas Kleiner
Department of Geography, University of Zurich (Irchel)
Winterthurerstrasse 190, CH-8057, Zurich (Switzerland)
e-mail: K247720@CZHRZU1A.bitnet

ABSTRACT

Various cell methods to organize large cartographic data sets under static conditions are implemented and compared. A grid scheme is tested in comparison to methods using a bintree subdivision of space for the organization of background data for drawing base maps. A regular grid index shows best performance and storage efficiency. A pyramid scheme using overlapping grids suited for geographical object storage is compared to the single-level schemes and should be preferred, as it avoids the effort of composing fragmented objects.

1. INTRODUCTION

During the last years, vector-based cartographic data bases were implemented for applications of small and medium size. For practical reasons, large digital cartographic systems have been realized mostly in raster format. At present, efforts on national and international levels are undertaken to collect base map information of whole map series in vector format. Interactive systems require short answer times for spatial queries. Typical systems will display thematic information on top of base maps. While the former will be dynamic information, the latter are large static data collections without frequent change. An important task is to organize these map data for fast queries. The motives for the present work are twofold. First, there exist several individual approaches. It is still a difficult question, which of all proposed methods to use in a real-world application. Comparative tests can hardly be found. Second, all work has dealt with methods for dynamic applications, where data are frequently updated. Static conditions allow other solutions and optimizations of storage arrangement. While an earlier paper (Kleiner and Brassel 1986) presented a theoretical discussion, the subsequent task was to realize an empirical comparative study of different methods. This paper presents the first results.

A detailed discussion of the basic concepts and a survey of the relevant publications can be found in Kleiner and Brassel (1986). Kleiner (1986) is a more complete collection of publications. As the mechanical process of reading from disk is relatively slow, theory tells to concentrate on minimizing this operations for queries. **Cell methods** preserve local proximity of data by attributing one or more storage buckets to a region in geographical space. In dynamic systems, buckets are spread all over the storage by continuous updates. Under static conditions, however, neighbourhood relations should not only be preserved locally by the definition of cells, but also in the arrangement of the buckets on

disk. Sequential reading should be maximized to avoid index accesses and, if physically realized, long mechanical seek time of the reading head. This seek time is critical for sequentially organized optical disks. The one-dimensional order that maximizes preservation of two-dimensional relationships is the **Peano path**, resulting from **bit-interlacing** or **bit-shuffling** of coordinate pairs. The order of the resulting codes corresponds to the traversal of space in a quadtree or bintree.

On the level of individual data, each object should be provided with its enclosing rectangle to speed up clipping at the query window. A basic distinction is made between **background data** - unorganized graphical data for base maps - and **geographical objects** - map information needed for thematic maps, e.g. topological structures of polygon networks for choropleth mapping. In contrast to methods using the flexible bintree scheme also applicable for dynamic data, much simpler static concepts like a regular grid with row and column computation should be considered as well. A **first major question** arises: Is it worthwhile to use a complex, flexible cell method in comparison to the spatially rigid, simple grid scheme? Frank (1983) proposed a scheme with a hierarchy of overlapping cells that allows to always find a cell which an object of arbitrary form and size can fit without the need to cut it at cell borders. The **second major question** is: Is it worthwhile to use a more complex cell scheme for geographic object storage to avoid composing cut parts for each query?

To answer these questions the relevant methods were implemented. In the next sections, the implemented alternatives for storing background data (2.) and those for storing geographical objects (3.) are presented. Each structure and query algorithm is shortly introduced and then the test results are discussed.

2. METHODS FOR STORING BACKGROUND INFORMATION

2.2 Grid Partition

The simplest cell method is to subdivide space into a regular grid. The key of each cell can be computed from its row and column numbers. The spatial query algorithm consists in scanning the cells in the relevant window. The following key organizations are used.

2.2.1 Direct Hashing

The cells are stored row by row and column by column. The addresses of the corresponding buckets can be directly computed, if the buckets are of fixed size. The problem of uneven data distribution leads to empty and overflowing cells. For the latter, overflow chains are formed and appended at the end of the file. Obviously, empty cells have to be stored with one unoccupied bucket.

The size of the cells has to be fixed in advance. If the cells are too large, they are often filled only sparsely; if their size is too small, long overflow chains will appear. A reasonable solution is to strive for cells that fill just one bucket on the average. With inhomogeneous data distributions and a low probability to search in very sparsely occupied regions, it is recommended to use cells that are smaller

than determined by the above given "school-book" rule. This means that the scheme is used as if the average data density was higher.

The advantage of the method is its simplicity, resulting in short central processing time (as well as easy software development and maintenance). No index accesses are needed, but for some cells, whole chains of buckets have to be read, while empty cells are read without retrieving any data. There is no storage overhead for an index structure, but the size of the data themselves is very large in case of inhomogeneous spatial distribution with many empty cells.

2.2.2 Index Hashing

The problem of empty cells can be eliminated by introducing an index corresponding to the spatial grid. Instead of computing a data address, the position in the index is obtained. It contains either a pointer to a data bucket or a nil value. The arrangement is optimized by collecting all buckets belonging to a cell and storing them sequentially. The trade-off between index accesses and accesses to empty cells (and between index storage and storage of empty cells) depends on data characteristics, and the tests cannot lead to a general decision. However, the additional space used for the index should be more than compensated by the saving in data storage.

2.3 Bintree Partition

The bintree is the most flexible regular tree division of space of "trie" type. In this application field, it has to be preferred to the less economical quadtree, as with a quadtree split always two bintree splits are performed at once. This scheme is combined with index methods using coordinate shuffling.

The question at hand is how to organize the spatial query. An elegant algorithm was published by Orenstein and Merrett (1984) and is the basis for the present implementation. The idea is to utilize the hierarchy of the space partition for a recursive algorithm. A bintree representation of the search window is constructed to be filled with the largest possible cells (getting smaller towards the borders). These cells then are subject to searches. The algorithm is limited by the depth of the real tree. For each search cell involved, beginning at the root of the tree, one of three cases occurs:

1. The cell lies completely outside the region of interest. It is neither searched nor further split.
2. The cell lies completely inside the region of interest. All data cells inside the search cell are read and processed.
3. The cell overlaps the region of interest. It is split, and the two sub-cells are handled recursively.

The search cells are processed in Peano order and within each search cell, all potential sub-cells form an uninterrupted segment of the Peano path. The search of a cell always begins with an index access to get the first data address. The rest can be read sequentially without the use of the index. The formation of the largest possible search cells guarantees maximum sequential reading.

A data cell can be smaller than, equal to or larger than the search cell. In the latter case, it must be ensured that the data cell is not read more than once.

Without going into details, it shall be mentioned that further improvements are combined with the elimination of recursion. Splitting of a cell or forming a parent cell involves only one bit of its Peano key.

2.3.1 EXCELL

This method by Tamminen (1983) is based on Extendible Hashing, which uses a directory of data addresses for each potential hash value. Adaptation to the data distribution is achieved by assigning of one or more pointers to the same data bucket based on the trie principle. If coordinate shuffling is used as hash function into the index, the assignment of the index to the data is an implicit bintree. In this static implementation, empty directory positions are filled by the address of the next non-empty cell.

The index of very large data sets cannot be held internally and must be kept on disk, distributed to pages. The effort to find the first data address involves the computing of the index position and at most a single access to the page containing the relevant part of the index. The size of the index may be a problem. As it exists in the maximum resolution of the key space, large cells occupy large parts of the index with the same address.

2.3.2 Two-level EXCELL

Large Extendible Hashing indices must be stored externally, particularly if there are many data layers with their own indices. However, such a structure should take advantage of the growing central storage capacity of modern hardware. For this purpose, an extension of EXCELL was developed, where a two-level index is used: an on-line index of fixed depth loaded into memory (from now on called internal index) and external indices for regions of higher resolution. In the practical implementation the external indices are kept in one file. As these local refinements to the first-level resolution may have any depth, a second internal structure is needed parallel to the first, i.e. a directory that indicates the corresponding "sub-depths". Searching is performed by computing the position in the internal index. This index item either points to a data bucket or to an external subdirectory. In the latter case, the pointer marks the beginning position of the relevant index in the external file. The external index now contains the required data address.

As the size of the index grows by the power of two, the overall index size decreases strongly with the introduction of an internal index. The effort of a search is reduced, when a data bucket is directly addressed by the internal index, as there is no need for a disk access; it is slightly worsened, when an external index position has to be accessed by two subsequent hash function computations. A similar approach was proposed by Davis (1986) for dynamic applications with a combination of EXCELL and interpolation-based index maintenance.

2.3.3 Bintree with B-tree

B-trees as the "classical" index structure in conventional data bases, can also be used to index bintree cells. The B+-tree is the variant used for range search, where the leaves form a sequential list of all existing keys (Comer 1979, Abel

1984). As the static conditions allow sequential searching without the use of an index, the original B-tree is better suited. The worst case effort to find the first data address with a B-tree is a logarithmical function of the size of the data. Average behaviour is influenced by buffering.

2.4 General Remarks on the Test Environment

The results of the specific environment will be discussed in respect to their general application. The present implementations are written in Pascal and run on an Apple Macintosh II with a Winchester disk of 30 ms average access time. The size of index pages was chosen as one disk block (512 bytes), data buckets consist of two blocks. Thus all figures for data file accesses have to be divided by two to get the actual number of data buckets. The measurements are comparable among the methods, but specific to hardware parameters, i.e. type of disk and the power of the CPU. An important distinction has to be made between **disks for direct access** and **sequentially organized disks**. The average access time of the optical CD-ROMS, which belong to the latter kind, is more than ten times slower than that of magnetic disks.

2.5 Test Results

The data set used for the present tests is part of the World Data Bank II and contains line data. The sizes of the different index and data files are indicated in table 1. WDB stands for the original World Data Bank files. With grid storage (G) that includes empty cells, the required file space is more than doubled. Using the grid structure with an index (G/I) to avoid storing of empty cells, the size of data is reasonable, while the index is very small. Attention should be paid to the large size of the bintree structure. In fact, the bintree with its spatial flexibility has a low bucket occupancy. **"Vertical" adaptation to variable data density by allocation of buckets assigned to one cell in a chain obviously is more economical than the "horizontal" adaptation of splitting cells in geographical space.** In order to explain the higher storage efficiency of the grid scheme, the consequences of a bucket overflow have to be analyzed. If an additional bucket is appended to a grid cell, only one line is affected that will continue in the new bucket. If a bintree cell is split, all contained lines can happen to be geometrically cut by the new borderline; even the same line may multiply across the border. Each intersection forces the introduction of new boundary points and the creation of additional enclosing rectangles.

The EXCELL directory (E/I) is almost as large as the data themselves. The use of the two-level EXCELL (E/2) leads to a significant reduction of the file of external directories, although the present test implementation used an internal directory of only depth 7. The B-tree (B/B) has a depth of 4.

Queries were carried out with variations of spatial range and in regions of different data density. Table 2 shows typical results for different sizes of the query window in areas of reasonable data density. The examples are chosen to be representative within a certain variance. The answer to the basic question is clear with respect to the present environment: **The grid solution performs significantly better than the bintree methods.** In the relation of internal processing to external reading the latter can even be neglected. Looking at

external processing, the results differ according to the size of the required map window. In general, the bintree partition is thriftier with accesses only for small windows, whereas large regions need fewer accesses to data buckets with the grid organization. How is this phenomenon to be explained?

In view of the **number of data accesses,** two factors have to be considered: the **adaptability to data distribution in space** and the **bucket occupancy**. The first factor is of significance only at the borders of the query window. If an area of high data density lies at a border of a "horizontal" partition in the geographical domain, only small cells hardly overlapping the boundary are retrieved. With large grid cells and "vertical" overflow chaining, a whole chain is read whose contents only partially overlap the window. In the internal regions, however, the second factor is decisive. The fact that the grid is more economical with respect to storage utilization results in a lower number of buckets. As a result, the border conditions dominate in smaller query regions, whereas bucket occupancy is crucial for large regions. The balance of the two effects depends on the bucket size. Small buckets have a better adaptability. On the other hand the relation of data to header information is worse, mainly due to higher object fragmentation. It shall be emphasized that the number of disk accesses would be crucial only if a processor about thirty times as fast was used.

The number of index accesses is not of significant influence, but it may be noted that the indexed grid method needs less accesses than EXCELL. Supposing the same cell partition of space, ordering the cells according to the Peano path should allow more uninterrupted sequences that can be read in sequential order. Why did the bintree scheme lead to more interruptions with need of index accesses? The average cell size of the grid implementation with the allocation of possibly several buckets is much larger than that of the bintree solution.

Among the grid solutions the index method with index can be preferred, since the large data file is substantially reduced and the influence of the index query on time is very small. If - in contrast to the examples used - searches are done in sparsely occupied areas, the indexed solution eliminating empty cells is even better.

The results of single-level and double-level EXCELL are almost equal. The internal index depth of the latter was chosen small, and it should give the same effect as a very large data set with a deep first-level index. However, in the present examples mostly external accesses occurred, together with many changes between different external indices. This should not be expected with larger files and greater internal depth, and the two-level variant can be expected to be superior in general.

The use of the B-tree instead of external hashing seems to be a valid alternative in the present implementation, but this result cannot be generalized to data sets of arbitrary size because the tree and the search time grow with the size of data.

3. METHODS FOR STORING GEOGRAPHIC OBJECTS

3.1 Organizations for Background Information

Point objects are stored in one of the above organizations. The common solution for objects of spatial extension (lines, areas) is to use these same methods, but to cut each object at cell boundaries. Queries thus require the extra effort of composing all object parts.

3.2 Overlapping Pyramid Hashing

The Field Tree was proposed by Frank (1983) to overcome the problem of cutting objects. It is a hierarchical structure, where each object is stored in a cell large enough to accept it in its integrity. The size of the cells is not determined by the spatial distribution of data, but by the size of its objects. Thus for the same region, cells of different levels of the tree can exist. The Field Tree is similar to a quadtree. Because in a quadtree partition objects cutting a main cell boundary can only be stored in the root, each level of the tree is displaced by half a cell. Small objects crossing high level borders now fit lower level cells because the cells overlap.

Similar to the background data organizations, the simple grid concept should be tried for the Field Tree concept, leading to a hierarchy of grids. It is implemented as a pyramid of displaced grid indices, allowing access by hashing instead of a tree traversal. Each level corresponds to the structure described earlier and is accessed by row/column hashing, however with different origins on each level. As the number of empty cells is very large and it is important to store only non-empty cells, the grid solution with index is used. The high-level indices may be kept in memory. Each query is performed independently on all levels. An overlapping pyramid applied to image representation is mentioned by Samet (1984).

The structure is not very flexible with respect to data distribution, and typical geographical objects are not very small. Therefore, variable size buckets (in multiples of a block) are used in the implementation. One main restriction to the use of this scheme is its limitation to object types of relatively small size. For example, it would be nonsense to store the coastlines of the oceans in such a structure, with the whole Pacific in one gigantic cell, and then ask for the coasts of Hawaii in a query.

3.3 Test Results

The test example uses administrative boundaries of Switzerland as line objects (arcs). Table 3 shows a typical result. The effort to search for identical beginning and end points of cut objects is far greater than the time for the query of the initial data. With the simple concept of different grids, the problem of the complex tree of cells is eliminated and overall performance is clearly better than with the methods using one level of cells. The search time is even shorter than that of the bintree methods. The test shows that **avoiding the composition of object parts is worthwhile when using the fast pyramid hashing**. Only with a substantially different relation between processor and disk speed, the low load

factor of the overlapping pyramid buckets could be of significance.

4. CONCLUSIONS

The use of cell methods to organize large spatial data sets is important, but by concentrating on external read operations, the internal processing effort must not be neglected. The results show that grid hashing performs faster than bintree methods with Peano key indexing due to its simple internal processing. Even with a combination of a very fast processor and slow optical disks leading to a crucial influence of disk accesses, bintree methods are superior only with small query windows. These results are valid for line and area data in vector format, where bucket occupancy turns out to be more economical with overflow chaining than with geometrical splitting of cells. For reasons of storage efficiency the grid method with index should be preferred. The use of the overlapping pyramid scheme to avoid cutting objects along cell borders is worthwhile unless strong processing power would reduce the composing effort drastically.

ACKNOWLEDGEMENTS

Prof. Kurt E. Brassel has kindly reviewed the text. This contribution is gratefully acknowledged.

REFERENCES

Abel, D. J. (1984): "A B+-Tree Structure for Large Quadtrees", *Computer Vision, Graphics, and Image Processing*, 27, pp. 19-31.

Comer, D. (1979): "The Ubiquitous B-Tree", *Computing Surveys*, Vol. 11, pp. 121-294.

Davis, W. A. (1986): "Hybrid Use of Hashing Techniques for Spatial Data", *Proceedings Auto Carto London*, Vol. 1, pp. 127-135.

Frank, A. (1983): "Probleme der Realisierung von Landinformationssystemen, 2. Teil: Storage Methods for Space Related Data: The FIELD TREE", *Institut für Geodäsie und Photogrammetrie, Bericht*, Nr. 71, Zurich: ETH, 63 pp.

Kleiner, A. and K. E. Brassel (1986): "Hierarchical Grid Structures for Static Geographic Data Bases", *Proceedings Auto Carto London*, Vol. 1, pp. 485-496.

Kleiner, A. (1986): "Data Structures for Spatial Data Bases", in: R. Sieber and K. E. Brassel (ed.): *A Selected Bibliography on Spatial Data Handling: Data Structures, Generalization and Three-Dimensional Mapping*, Zurich: University of Zurich, pp. 3-19.

Orenstein, J. A. and T. H. Merrett (1984): "A Class of Data Structures for Associative Searching", *Proceedings of the Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pp. 181-190.

Samet, H. (1984): "The Quadtree and Related Hierarchical Data Structures", *ACM Computing Surveys*, Vol. 16 (2, June).

Tamminen, M. (1981): "The EXCELL Method for Efficient Geometric Access to Data", *Acta Polytechnica Scandinavica, Mathematics and Computer Science Series, No. 34*, Helsinki: Univ. of Technology, 57 pp.

Table 1: File size with different methods

	WDB	G	G/I	E/1	E/2	B/B
index size (K)	81	-	27	4097	540	179
data size (K)	3173	8913	3612	5357	5357	5357

Table 2: Query of line data with different methods

	G	G/I	E/1	E/2	B/B
query window: 0.5° x 0.5°					
search time (s)	1.03	0.99	6.05	6.53	6.57
internal portion (s)	0.11	0.12	5.43	5.91	5.91
external portion (s)	0.92	0.87	0.62	0.62	0.66
disk accesses	42	44	26	26	28
index accesses	-	2	2	2	4
data accesses	42	42	24	24	24
bucket occupancy (%)	45	45	31	31	31
used data (%)	25	25	76	76	76
query window: 1.0° x 1.0°					
search time (s)	1.59	1.46	10.33	11.14	11.24
internal portion (s)	0.13	1.12	8.96	9.78	9.77
external portion (s)	0.46	0.34	1.37	1.36	1.47
disk accesses	70	72	70	66	69
index accesses	-	2	6	2	5
data accesses	70	70	64	64	64
bucket occupancy (%)	45	45	32	32	32
used data (%)	41	41	76	76	76
query window: 2.0° x 2.0°					
search time (s)	4.16	3.63	16.58	17.76	18.03
internal portion (s)	0.19	0.17	12.35	13.47	13.44
external portion (s)	3.97	3.46	4.23	4.29	4.59
disk accesses	172	174	213	213	212
index accesses	-	4	13	13	12
data accesses	172	170	200	200	200
bucket occupancy (%)	43	44	32	32	32
used data (%)	66	66	85	85	85
query window: 4.0° x 4.0°					
search time (s)	7.44	6.66	20.40	21.44	21.74
internal portion (s)	0.21	0.23	12.30	13.46	13.45
external portion (s)	7.23	6.43	8.10	7.98	8.29
disk accesses	318	318	427	424	425
index accesses	-	6	15	12	13
data accesses	318	312	412	412	412
bucket occupancy (%)	41	42	32	32	32
used data (%)	88	88	94	94	94

Table 3: Query of line objects with different methods

query window: 130 x 130 km	G	E	OP
search time (s)	1.22	4.87	2.42
object processing time (s)	33.83	40.55	8.59
total time (s)	35.05	45.42	11.01
disk accesses	68	88	92
index accesses	-	8	-
data accesses	68	80	92

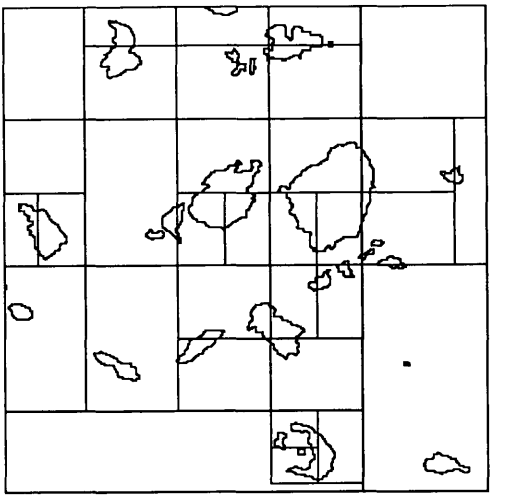

Figure 1: Greek islands organized in a bintree

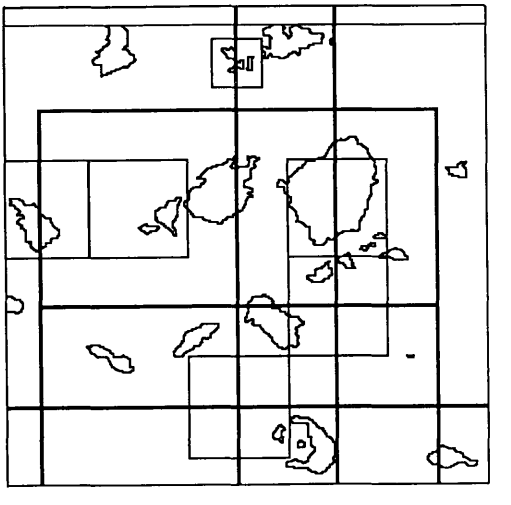

Figure 2: Greek islands organized in a pyramid of overlapped cells