Frank, A. U. "Probleme Der Realisierung Von Landinformationssystemen 2. Teil." 63. Zürich: Eidesgenössische Technische Hochschule Zürich, Institut für Geodäsie und Photogrammetrie, 1983.

Eidgenössische Technische Hochschule Zürich

Institut für Geodäsie
und Photogrammetrie

Bericht Nr.

71

PROBLEME DER REALISIERUNG VON LANDINFORMATIONSSYSTEMEN

2. Teil

Storage Methods for Space Related Data: The FIELD TREE

André Frank

Juni 1983

Anmerkung des Herausgebers

Die Beiträge, die in der Schriftenreihe 'Berichte des Instituts für Geodäsie und Photogrammetrie' erscheinen, dienen vor allem dem Unterricht und der Dokumentation. Sie sind deshalb in erster Linie für Mitarbeiter des Instituts und für Studenten bestimmt. Einzelne Hefte können auch einem weiteren Kreis von Interessenten zur Verfügung gestellt werden. Die Auflage ist auf den besonderen Zweck des Heftes abgestimmt.

VORWORT

In Landinformationssystemen werden Daten, die sich auf geometrisch begrenzte, im Raum fixierte Objekte beziehen, gespeichert. Diese Daten werden kurz 'raumbezogene Daten' genannt. Benützer von Landinformationssystemen wollen meistens Teilmengen dieser Daten als Plan darstellen und geben dazu einerseits ein Gebiet an und anderseits Daten von Objekten in diesem Gebiet. Solche räumlichen Zugriffe müssen rasch erfolgen, damit interaktives geometrisches Arbeiten am Bildschirm attraktiv wird. Raumbezogene Zugriffe sind aber auch zur Ueberprüfung der geometrischen Konsistenzbedingungen häufig erforderlich. Diese 'inneren' Funktionen sind dem Benützer kaum bewusst; ihre Ausführungsgeschwindigkeit beeinflusst aber direkt die Leistungsfähigkeit des Systemes bei Daten-Mutationen.

Dr. André Frank hat schon früh begonnen, diese Zusammenhänge zu erforschen (IGP Bericht Nr. 26, Probleme der Realisierung von Landinformationssystemen, 1. Teil: Datenstrukturen und Speicherung). Im nun vorliegenden Bericht wird eine überarbeitete, verallgemeinerte Methode für den räumlichen Zugriff beschrieben. Insbesondere werden dabei Ideen von Prof. J. Nievergelt (Institut für Informatik ETH) einbezogen, die eine systematische Darstellung der komplexen Sachverhalte erlauben und sie so der Untersuchung besser zugänglich machen.

Dieser Bericht bringt zwar noch keine abschliessende Behandlung des Problems; er dokumentiert den derzeitigen Stand. Es sind zusätzliche Arbeiten notwendig, die die Verteilung räumlicher Daten untersuchen und Wege zeigen, wie sie mit (statistischen) Parametern beschrieben werden könnten. Ich hoffe, dass eine weitere Zusammenarbeit zwischen unserem Institut und der University of Maine at Orono, wo Dr. Frank heute 'Computer Assisted Mapping' lehrt, in dieser Richtung weitere Fortschritte bringt.

Der Bericht wurde von Dr. Frank für ein Seminar des Lincoln Institute for Land Policy, Cambridge (Massachusetts) geschrieben; er wird am IGP als internes Arbeitspapier verwendet.

R. Conzett

Dr. Andrew U. Frank
University of Maine at Orono

STORAGE METHODS FOR SPACE RELATED DATA:
THE FIELD-TREE

ABSTRACT

Retrieval of data based on location is the most important access path in spatial data collection, e.g. Land Information Systems. In such systems large amounts of data related to varying, geometrically described objects are stored. Users most often need map-like graphics for which all objects falling in a particular area have to be retrieved. To allow interactive work a fast access method for this special two-dimensional range query is needed.

Field-tree is a storage structure which allows fast retrieval for spatially extended objects within a two-dimensional window. Access time is essentially **linear in the number of objects retrieved** and nearly independent of the amount of data stored. The field-tree method is physically clustering objects with similar location, thereby preserving neighborhood in limited areas. This is beneficial for other algorithms of computational geometry which process data locally, too.

Clustering objects in fields, field-tree is similar to hashing methods and their applications to multi-dimensional data. Using a specific adaptable method for subdivisions of fields, the fields can be arranged in a tree, which proves advantageous for range queries. A number of parameters permit the adaptation of the method to specific applications and hardware. It is independent of the way objects are described and can, therefore, be included in a generalized database management system.

TABLE OF CONTENTS

	PAGE
1. INTRODUCTION	5
2. APPROACH	7
3. FORMAL DESCRIPTION OF A LAND INFORMATION SYSTEM	9
3.1 LIS as a Data Collection	9
3.2 LIS as a Spatial Data Collection	10
3.3 Typical Queries in a LIS	10
3.4 Approximate Size of a LIS	11
3.5 LIS Requires a Database System which Includes Special Methods for Spatially Related Data	12
4. ABSTRACTION TO FIND A LOWER LEVEL GENERAL ALGORITHM	13
4.1 Restricted Set of Spatially Accessible Data Types	13
4.2 Generalization of Spatially Accessible Objects	14
4.3 Generalized Spatial Retrieval	15
5. INVESTIGATION INTO STORAGE METHODS	18
5.1 Performance Goals	18
5.2 Storage Structure Influence	20
5.3 Storage Devices	20
5.4 Principals of Storage Structures	22
6. ACCESS METHODS FOR SPATIAL RETRIEVAL	27
6.1 Quad-Trees	27
6.2 Dividing Address Space in Regular Fields	28
6.3 Dividing Space in Fields with Varying Size	30
6.4 Treatment of Extended Objects	32
6.5 Goals for Storage Methods Using Overlaplping Fields	35
6.6 Methods for Discussion	36
6.7 Basic Method for Spatial Retrieval	39
6.8 Storage and Retrieval	48

PAGE

7. EXAMPLE IMPLEMENTATION 51
7.1 Specific Method 51
7.2 RAUML 53
7.3 FIELD 54
7.4 Module GRID 55
7.5 Module QTREE 56
7.6 Module VIERECK 57
7.7 Results 58

8. CONCLUSION 58

GLOSSARY 59

REFERENCES 59

1. INTRODUCTION

Land Information Systems as defined in [FIG 1981] are systems "which consist, on the one hand, of a data-base containing spatially referenced **land-related data for a defined area,** and on the other hand of procedures and techniques for the systematic collection, updating, processing and distribution of the data" (Figure 1).

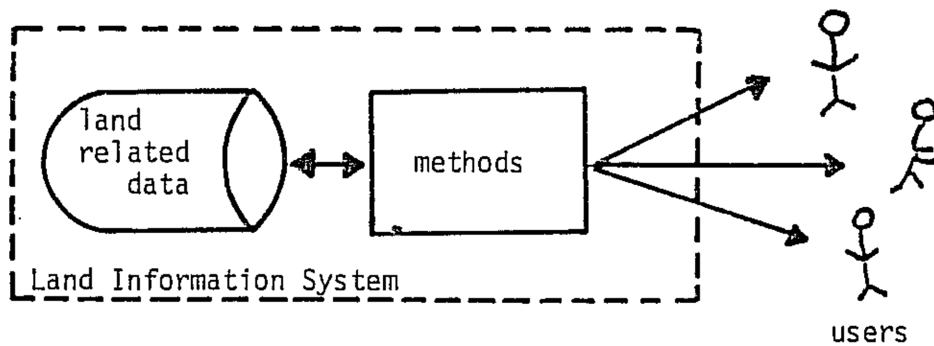


Figure 1

This paper concentrates on methods for storage and retrieval of spatially referenced land-related data for different applications. Such methods should be included in generalized database management systems to make them applicable for the management of land-related data. The discussion here will be restricted to operations related to the spatial reference of such data, namely **the fast retrieval of all data elements related to a certain area of interest.** Such operations are extremely important for Land Information Systems, not only, but most obviously, in order to quickly satisfy demands for graphical, map-like presentation of data contained in the system [Herot 1980]. Such operations are equally important for many internal operations like data aggregation for certain areas and eventually for checks on consistency of newly input data with previously stored data (many references in [Dutton 1978]).

Data structures supporting such retrieval operations will also be beneficial for other geometrical algorithms, as many of them can be formulated in order to take advantage of locality in the data [Dutton 1978a]. The retrieval of all data related to a certain area of interest is a special form of a two dimensional range query, for which only a few general purpose methods are known [Leuker 1976] [Lee 1977] [Bentley 1979] [Lee 1980].

Data structures must be laid out independently of the hardware needed and in order to be applied in micro computer based systems as well as in programs, running on mainframe computers

In each case they have to be tailored to suit the characteristics of the hardware and the operating system used. In order to discuss generic transportable software, such adaptations, even if crucial to the performance of the system, should be clearly separated from the basic logic of the method.

In this paper only methods for fast spatial access to spatially referenced data are covered. A complete package for storage and retrieval of data in a Land Information System (or any logically coherent part thereof) must include many more access paths to retrieve data (e.g. based on key values, on logical connection between data as from parcel to owner, etc.).

Generalized database management systems, as they are available from different sources and for different hardware, ranging from micro computers [Everest 1982] to mainframes [DEC 1977 a & b] usually provide this sort of functionality. None of them, however, provides the functionality for access based on spatial reference. It has been shown to be feasible to implement the methods described hereafter on top of a commercially available, CODASYL type [CODASYL 1971 and later] database system [Frank 1981], thus enhancing the database management system with additional functionality. It seems more advantageous, however, to build this function right into the database system, resulting in increased performance and lower overhead.

This is not easily done with commercially available systems for which access to source code usually is barred by commercial interest. A completely transportable database management system including space-related functions has been built [Frank 1982] [Frank 1982a]; this system has been successfully implemented on different mainframes and micro computers (e.g. IBM 370, PERQ produced by Three Rivers Computer Corp., DECsystem-10).

2. APPROACH

In order to discuss the methods for storage and retrieval of space-related data in a most generic form independent of specific hardware characteristics, we must concentrate on discussing **what** has to be performed but refrain from detailed description of **how** to do it. This is an advocated practice in modern software engineering [Liskov 1979], known as 'program specifications' or 'abstract data types' [Parnas 1972a] [Isner 1982]. These specifications, originating at the description of the user's needs, are then broken down into functions of lower layers. The more restricted the interfaces between the layers, the easier the maintenance and adaptation of such a program package. Each step down leads closer to the functions computer hardware performs. If each layer is only relying on the functionality of lower layers (the **what**) and is independent of how this lower layer performs these functions, it is much easier to adapt such a program to a specific task and to transport it to different hardware (e.g. [ISO 1979]).

The exposition of the storage and retrieval method in this paper is guided by these principles. Decisions about how to perform a certain task are intentionally postponed as much as possible to preserve adaptability to different applications.

These techniques will allow us to define very small and easily programmable modules. For the description of these modules we shall use a non-formal way of presentation and not strive for a formal description [Guttag 1977] [Parnas 1972]. Experience shows that formal descriptions - at least the forms proposed to date - are hard to read and understand without a great deal of study and experience.

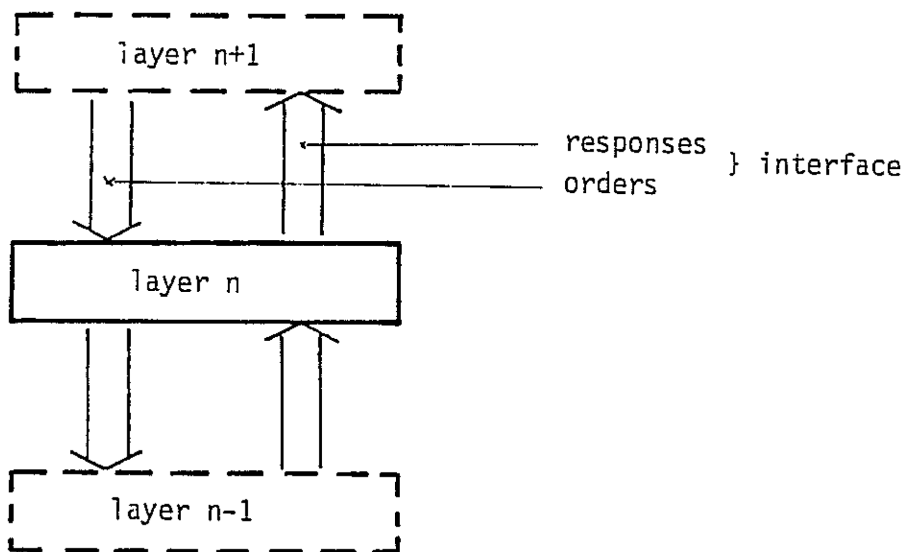


Figure 2

The paper will therefore first give a formal description of a Land Information System as it is assumed for this paper. The basic space-related query for map drawing purposes, which forms the primary aim of the method here presented, is then introduced.

A close investigation into the typical query and how to treat it on one hand, and the desire to find abstract properties which hold for a larger class of data types without modification, on the other hand, will enable us to identify generalized, abstract properties. These make it possible to build lower level functions which can be used uniformly by higher level, more application-specific procedures. This leads to a restatement of the problem in more abstract terms.

Once a formal description of the task is available, we may proceed to assess published storage and retrieval methods.

Much consideration is given to stating goals for solutions so that different methods can be assessed and a rational choice is possible. We have, however, only limited experience with treatment of spatially referenced data and no systematic investigations into statistical properties of real data are known. Such

studies should be encouraged as they would ultimately lead to better adapted algorithms.

Finally, the modules needed for the implementation (based on a database system) are explained and the code is joined in the appendix.

3. FORMAL DESCRIPTION OF A LAND INFORMATION SYSTEM

3.1 LIS as a Data Collection

Without considering the meaning of the data stored and excluding any discussion of the relation between the data stored and the facts in real world (which can be found in [Frank 1983]), we may state that a Land Information System L is a (probably large) set of (logical) records d.

$$L = \{d\}$$

The records may be partitioned in disjoint sets of records of the same **type.***

<intersect over i> D.i = L
<union over i> D.i = ∅

with

$$D.x = \{d.x\}$$

The difference between a simple occurrence of a record d.x describing a simple fact (e.g. a description of my house) and the generalized set of records D.x (the type 'house' in general) must be clearly borne in mind for the following discussion.

----

*To avoid typographic problems, index subscripts are given here in the 'dot' notation customary for record field selection in modern programming language. Angle brackets are chosen to denote operations, for which no convenient signs are available.

3.2 LIS as a Spatial Data Collection

Most of the data in a LIS (this follows from the cited definition of the LIS) are related to objects located in space. These objects may have arbitrary shape and location. It is not necessary to store for all objects an explicit description of location and extension, because the spatial reference may be a reference to other objects whose location and extension are known. This is called **indirect** spatial reference.

The logical data records d consist therefore of two parts, namely geometric data and other attribute data. The geometric data may either be a description of location and shape or a reference to another data record which provides a geometric description (either directly stored or by reference to another record).

d.i = d.i. description <union> d.i. geometry

It is not assumed that this reference to location and extension/shape is of the same type for all data records in the LIS. The spatial reference is expected to be different for different types of data, depending on the typical shape of the objects referenced.

The smaller amount of data in a LIS which is not describing geometric objects, is treated with standard methods and is of no special concern in the context of this paper.

3.3 Typical Queries in a LIS

Users of a LIS will very often retrieve data based on spatial criteria. Most obvious is the request for a map on a CRT screen which requires retrieval of all data of certain types related to objects within the map boundary. Such queries should be answered fast enough to support interactive work [Martin 1968].

This is more difficult to achieve for this type of query than for queries in commercial systems. Answers in commercial systems most often are based on only a few logical records whereas a map of the size of a CRT screen easily consists of data from 1000 - 2000 logical records. Screens with more data are too

crowded to be easily understood, whereas screens with less data look empty and are difficult to interpret.

Similar retrieval operations are used internally to check consistency of new or updated data which have to be stored in the system. It is obvious that consistency checks in a LIS will involve spatial relations (e.g. a house must lie within the geometrical bound aries of its lot). To check such consistency constraints requires the retrieval of data within a limited, usually small, area.

Formally we characterize the retrieval operation as:

```
RETRIEVE DATA OF ALL OBJECTS
   OF TYPE      type-list
   WITHIN       area-description
   WITH         additional-condition.
```

Where

- the type-list is the set of data types required in the answer (all these must be of objects with a geometric description)

- the area-description is a definition of the geometric form and location of the area of interest. This description can be specified either as an explicit area or as reference to a stored object which has an area description associated with it (e.g. the name of a county).

- the additional-condition contains the other than location based conditions the data retrieved have to fulfill.

3.4 Approximate Size of a LIS

In order to develop a storage structure, certain very rough assumptions about the size of the application are needed. These figures are not critical as they may vary over several orders of magnitude without affecting the methods; parameters within the algorithms allow to adapt them to specific cases.

Investigations in Switzerland for LIS at the local level - mainly treating property and utility line data - let us expect the following numbers:

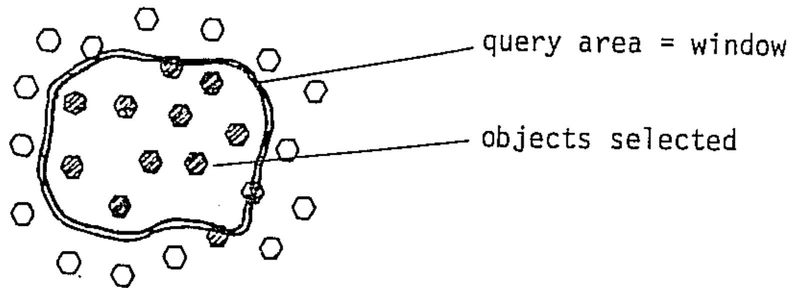


Figure 3

Up to 40,000 points per square mile, and up to 1 million objects (property data, line data, etc.) per system. Remarkable is the variance in the density of objects per area, which is over 1:1000 (for America probably even higher). About 60 different object types can be identified and the distribution seems to follow the familiar 20:80 rule.

As the geometric objects described are not changing often, a few percent of the stored data are changed annually only, the rate of updating is very low. The ratio between update and retrieval is therefore less than 1:1000.

3.5 LIS Requires a Database System Which Includes Special Methods for Spatially Related Data

The data in a LIS are related to each other in many ways not involving spatial relations. These connections may be exploited by the users with the same methods usually needed in

information systems for other applications (e.g. financial, personnel or equipment information systems). Methods to achieve this are published in textbooks for database management systems, and it is therefore not necessary to discuss those techniques here [Date 1977] [Ullman 1982]. But it should be clearly understood that a Land Information System needs this functionality together with special methods to deal with spatial queries. A full database system should thus be included in a LIS program.

4. ABSTRACTION TO FIND A LOWER LEVEL, GENERAL ALGORITHM

A method for storage of data related to objects with geometrical properties (location and shape) and retrieval of such data based on location is needed in a LIS. Such algorithms should be generally applicable to all kind of data without regard to their internal structure or the specific form of the description of their geometric properties. It is clearly not appropriate to devise a specific method for each data type in a database. This would hamper the adaptability of a database system to new situations considerably. To prepare a generalized algorithm we have to find general traits of the objects we deal with. There are two different ways we can proceed:

- either we restrict the objects which are spatially retrievable to a few limited types and build all other objects from these base types

or

- we find general characteristics of all the objects to be treated which are sufficient for the task.

4.1 Restricted Set of Spatially Accessible Data Types

The geometric properties of all objects in a LIS may be described using only a few geometric data types (e.g. points and lines). It is possible then to restrict spatial access to these geometric data types. In consequence spatial access to other data has to access the geometric description first.

This seems barely practical for mapping purposes when our interest is focused on geometric properties, but is clearly not adequate in cases in which we need spatial access to complex objects in order to produce derived data (e.g. averages for an

area or similar) without being interested in the geometric properties.

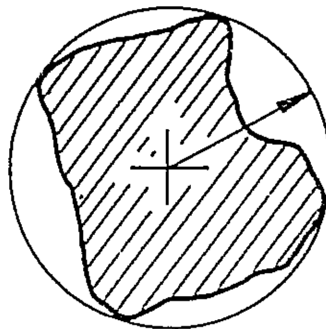
Furthermore, there are clearly two performance disadvantages:

- access to additional data is indirect, making response slower,

- the algorithms will need decisions of the 'point in polygon' type. These decisions require computationally expensive algorithms. (Faster auxiliary algorithms of the type described in the next approach certainly be used to increase speed.)

4.2 Generalization of Spatially Accessible Objects

In 3.2 we stated that all data in a LIS refer to objects with known location and shape. The different types of shapes may be generalized to a geometrically simple **circumscribing figure.** A minimal description with three values, two values indicating the location and one the size of the circumscribing circle of the object would suffice (e.g. center point (x,y) and radius of a circumscribing circle).



center point (x,y)
&
radius (r)

Figure 4

Non-minimal descriptions may describe the shape of the object more adequately and therefore be more advantageous for the search procedure (cf. next paragraph).

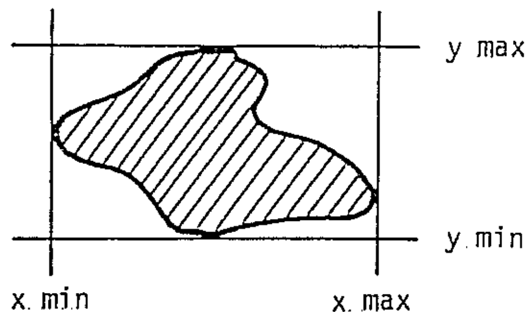


Figure 5

A rectangle with sides parallel to the coordinate axis gives a computationally simple description needing four values (e.g. two coordinate values for each of two diagonally opposite corners or two coordinate values for the center point and two for the length of the sides). More elaborate descriptions (e.g. boxes with sides not parallel to coordinate axis) describe the objects more accurately but the operations needed involve time consuming, trigonometric functions . This seems appropriate for very specific applications only.

Generally each object d.i (i a geometric object type) has a circumscribing figure v in order that

for all points x $(x \subseteq d.i \Rightarrow x \subseteq v)$

is true.

4.3 Generalized Spatial Retrieval

Reconsidering the spatial access requirement (cf 3.3) we can split it in a generalized part, which could be carried out by the database system and a type-specific part, which has to be programmed specially for each data type.

The spatial query (3.3) can be stated formally as a restriction

on L yielding the set of objects forming the answers R

R = <all> d <in> L <which>

(d.i $\subseteq$ type-list) AND
(d.i.geometry $\subseteq$ area-description) AND
(d.i.description $\subseteq$ additional-conditions).

These conditions may be checked in any order and the rules regarding the outcome of AND-connections allow evaluation of the conditions to stop as soon as one yields a result of 'FALSE'.

Furthermore, we may split one condition (a) in two (a1 and a2) provided

(not a1 => not a)	or	a => a1
(not a2 => not a)		a => a2

Applied to the geometrical condition it may be split in a condition on circumscribing figures for the window and the objects and an exact condition:

(x.i.geometry $\subseteq$ area-description) =
(x.i.v $\subseteq$ area-description.w) AND
(x.i.geometry $\subseteq$ area-description)

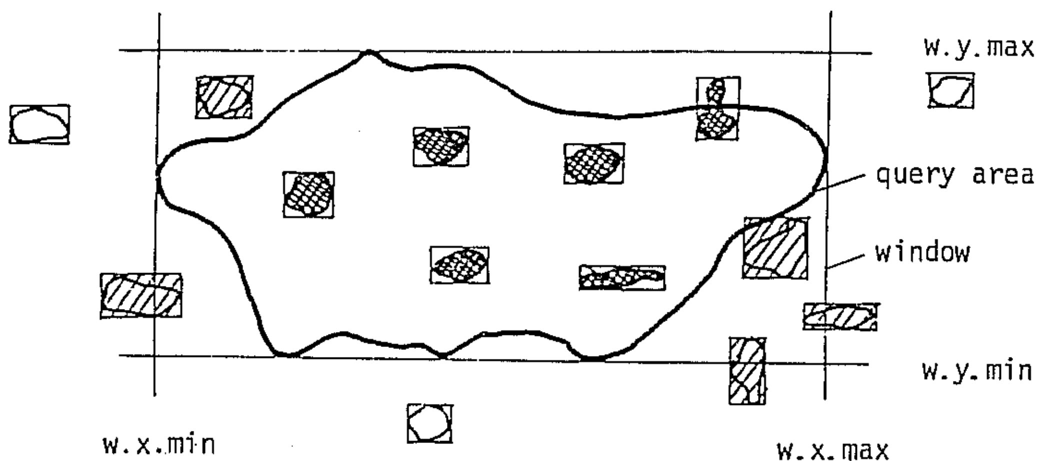


Figure 6

Testing the circumscribing figure (rectangle in Figure 6) for overlapping is simple and can be done fast whereas testing for exact inclusion in arbitrary figures is much more complicated and time consuming.

The decision in which order to test these conditions depends on their selectivity (ratio between objects accepted and total objects tested [Wedekind 1976]) and the costs (in the form of waiting time for the user).

Reasonable assumptions about the number of objects and the type of queries most likely to be asked yield:

Test	Selectivity	Cost
type-list	medium 10%	low
circumscribing figure	very high (<1%)	low
exact area-description	very high (<1%)*	very high
additional-condition	variable (100-10%)	medium

It is clearly advantageous to perform the two tests with low cost and high and medium selectivity first thus excluding the maximum number of data from the costly tests for exact geometric inclusion and the additional-conditions.

Moreover, these two tests are defined for generalized objects (cf 4.2) and can therefore be included in a generalized storage and retrieval method for data related to spatial objects, not restricting treatment to special geometric data.

This generalized method therefore features two functions:

---

*This selectivity is assumed if the test is applied to all data stored; if it is applied only to data which passed the test with circumscribing figures, the selectivity may drop to 50%.

- STORE OBJECT (o.v, o.type, o.data)

  Where - o.v is a description of the circumscribing figure of the object,
  - o.type the type of the object
  - o.data the data associated with the object (these data are not further interpreted by the method and can be thought of as an unstructured stream of bits)

and

- RETRIEVE ALL OBJECT
  OF TYPES type-list
  WITHIN window

  where - type-list is a set of types describing which object types should be included,
  - window is the circumscribing figure of the area of interest.

The latter operation will in the sequel be called a **spatial retrieval.**

The rest of this paper will be restricted to discussing methods for storage and retrieval of data of space-related type, which allows fast spatial retrieval of the sort described above. This operation is generally applicable to any type of data related to space and can be implemented without assumption about the internal structure of the data.

5. INVESTIGATION INTO STORAGE METHODS

5.1 Performance Goals

The performance of retrieval operations is dependent on the structure in which the data are stored . The simplest storage structure is a sequential file which, consequently, is also searched in sequential manner. This search method requires an amount of time which increases linear with the number of data objects stored. Such performance is clearly not acceptable for any data collection which may eventually contain a large amount of data.

An optimal system would respond to any query in a short, essentially constant time.

Factors influencing response time for the spatial retrieval are (cf. 4.3):

- card* (L) amount of data stored
- area (w) area of the query window
- card (R) amount of data needed for the response
- t. set up time to set up the query

And the response time is a function

t. response = f.L (card (L)) + f.w (area (w)) +

f.R (card (R)) + t.setup

To build systems which are useful for the storage of large amounts of data, the influence of the amount of data stored f.L must increase much slower than linearily, otherwise users will experience unacceptable long response times when the amount of data stored in the system, has grown.

On the other hand, it seems logical to assume that users will expect and thus accept the response time to be longer in cases where a larger area has to be searched and/or more data records are needed for the answer.

The number of records which can be graphically presented on a CRT with a given resolution does not vary too much (estimated about 2,000 records) - more data can not be visually analyzed, whereas muchfewer data results in 'empty' maps which are difficult to interpret and not useful to work with. The influence of the amount of data in the response f.R may approach linearity without devasting effects.

----

*card is an abbreviation for cardinality, the number of objects in a set.

The size of the query window may vary greatly with the application. Some users may only be interested in some landmarks over a wide area (e.g. all interstate highways in a state), whereas others are looking for very detailed information about a small area (e.g. all roads in a village). In both cases, the answer will include about the same amount of data (card (R)) - and therefore the user will expect approximately the same response time. Thus it will be important to prepare the retrieval methods to deal with queries requiring different information density in their answers; the influence of the area of the window f.w must therefore be kept low (for a given card (R)).

5.2 Storage Structure Influence

In order to improve the retrieval performance we have to adapt the storage structure in such a way as to be prepared to respond to the most often used query. All such preparations increase the cost of storage operations and increase the storage space requirements - there is clearly nothing obtained free; a trade-off is always involved.

The additional cost incurred during storage - additional operations needed to keep the structure up-to-date - are not important in typical LIS applications where retrievals are much more often performed then storage of new data. Typically space-related data is very stable and changes are seldom.

All storage methods are geared towards certain access requirements; they may be indifferent to (or even hinder) access based on other attributes.

5.3 Storage Devices

We can assume that the data stored in a LIS occupy more storage space than can be reasonably used in a computer's main storage (solid state or core memory). Long time storage of data has to use secondary or mass storage devices permitting random access (preferably disk, but floppy disk may be used, too). Storage space on such devices is even for small systems readily available and relatively cheap. Thus the increase in storage size caused by a specific storage method is not of great importance to us.

Access to data stored on mass storage devices is relatively slow (compared to access in main memory about 10,000 times slower). Experience with database systems shows that practically all response time delay is due to access to data stored on mass storage devices and the influence of processor speed is minimal. This causes us to investigate the different storage methods considering only the average number of accesses to main storage and disregard other processing needed.

The performance difference between larger and smaller systems should not differ much. Mass storage devices for small systems are not much slower than the best available for mainframes (30..100 msec* compared to 15.. 30 msec for mainframes; if floppy disks are used, it amounts to 300 msec). Performance is expected to be similar provided the small system has enough main storage (and large enough an address space to take advantage of this main storage) for all programs and the necessary buffers to reside in main memory all the time (e.g. 1/2 M bytes). In practice performance is much more influenced by the interaction between database and operating system; with a certain database example, we observed better performance on a PERQ micro computer than on an IBM 370.

Before different storage and retrieval algorithms can be discussed, we have to investigate briefly the typical access characteristics of mass storage devices.

Mass storage devices have a typical access time characteristic of the form

$$t.access = t.Position + t.Transfer \cdot l.data$$

where

- t.Position is the time needed for positioning the device's reading/writing heads to the point where the data required are stored,

- t.Transfer is the transfer time per unit length of data required, and

---

*msec = millisecond = 1/1000 second

- l.data is the length of the data to be accessed.

Generally t.Position is much larger than t.Transfer (30 m sec compared to 1 microsec). It is therefore faster to access data on a mass storage device in large blocks (typically 1 kbytes). Access to individual data elements within the block may then be done much faster in main memory.

If we are able to transfer with one access to the main storage device more than one useful data element, we can save that many times the long positioning time.

5.4 Principals of Storage Structures

Storage structures and access methods can either be based on an organization of data space or of the address space [Nievergelt 1980].*

Basically three different broad categories of data structures may be discerned [Fagin 1979]:

- sequential storage, with minimal organization,
- tree structures, organizing the data space,
- hash methods, organizing the address space.

Retrieval in sequential storage needs in the order of O(n) operations, where n denotes the number of elements stored. This is clearly not acceptable for this application.

Typical methods based on organization of data space are using tree structures [Knuth 1973]. Each data element at the same time represents itself and marks a point in the data space. These marked points are then used during retrieval to navigate thru the stored data. Retrieval needs in the order of O (log (n)) operations.

---

* In this context, the address space is formed by the data records keys on which access is required and should not be confused with the physical address space used for addressing storage cells where data are stored.

Typical methods based on organizing the address space are using hash storage. The address space is divided into fields; to each field a storage bucket is attributed. All data whose addresses fall into a certain address field are stored in the corresponding bucket. Retrieval of an element needs then O(1) operations, but certain other drawbacks have to be considered. An extensive discussion of such methods can be found in [Fagin 1979].

More advanced methods achieve better performance in adapting to local differences in the population of address space by stored records [Nievergelt 1981] [Tamminen 1981] [Tamminen 1982].

5.4.1 Performance Estimates - Given that the data in a LIS may not be completely stored in main memory but must be stored on mass storage devices and considering the access time of these devices (cf. 5.2) estimates of performance of different storage and access methods can not only count operations, but must differentiate between operations in main memory and those needing access to mass storage. The latter consume so much more time that valid response time averages can be computed using only the number of accesses to mass storage. The number of operations using data in main memory only is hardly influencing the performance.

It is clearly advantageous if the number of physical accesses to mass storage can be reduced and methods using more than one data element of a block of data accessed from mass storage are faster. Response time is reduced roughly by a factor averaging the number of useful data elements found in each block accessed.

The goal is therefore to physically and logically cluster data which are often used together in one block of mass storage. Storage and access methods which allow such clustering are clearly at an advantage if data are stored on mass storage devices and usually a number of records are retrieved together (e.g. index-sequential access method ISAM, B*-trees [Knuth 1973], and EXHASH [Fagin 1979]). Methods based on organization of the data space have usually more problems in clustering the data. The methods for clustering may either use preset guidelines or may adapt themselves to the data stored. Extensive research into clustering data for fast retrieval in multi-key queries has been done in the area of document retrieval [Schkolnick 1977] [Salton 1978].

Performance of retrieval of large amounts of related data as it occurs during the spatial retrieval in LIS is obviously much influenced by the physical clustering of the data elements.

Physical clustering however may only be done for one, the most important, access path. In a LIS, this will be in most cases the access based on location, given that this access may benefit most from physical clustering. Experience shows that response time may be 10 .. 100 times shorter compared to an unclustered storage structure.

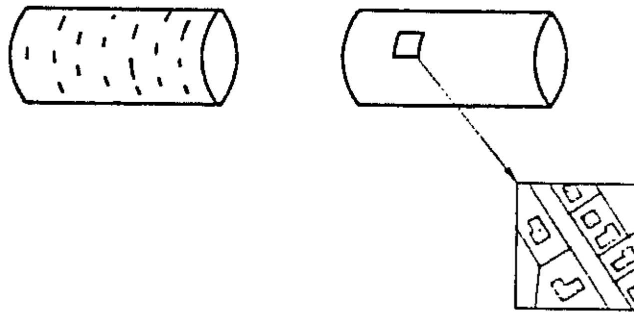


Figure 7

Within methods exploiting clustering of data, we can differentiate between the access methods for the clusters on the one hand, and the access methods for the data within the cluster on the other hand.

Therefore the following analysis of methods is divided into two questions:

- which data are clustered together
- how to access a cluster.

It must be pointed out that these are purely logical discussions of storage and access methods and do not imply a specific implementation. Comparisons are based on the average number of accesses to clusters with a minimum retrieval of data not needed. Details of the implementation, especially how physical clustering is achieved, are not considered. Certain implementations

which take advantage of the physical characteristics of storage devices, however, may prove superior to others.

5.4.2 Maintenance of Storage Structures - Nearly always when we use storage and access methods, we do not know beforehand the contents of the data to be stored. Most storage algorithms, however, build different storage structures according to the sequence in which the data are stored.

Certain sequences of input may produce storage structures which are non-optimal for access. Methods based on data space organization need usually permanent reorganization to maintain optimal (or at least near to optimal) data structures (e.g. balancing of trees [Knuth 1973]).

Methods based on address space organization need reorganization when the amount of data stored grows (globally or locally) over a certain margin. Sometimes limits are set for how much data a cluster may contain at the most. If this limit is overrun, the cluster is divided into some subclusters.

Reorganization of data structure is part of the additional effort we must pay for faster access. On the average, this does not influence performance of storage much and even less so average performance in a LIS, expecting much retrieval and a few storage operations (cf. 5.2).

Reorganization may, however, incidentally hold up storage operations more than acceptable for interactive work - this happens seldom, but it is too bad if it happens.

If a storage method is well divided in a logical clustering which is used for retrieval and a physical clustering, which speeds retrieval, shortcomings in physical clustering can be tolerated. They influence only performance, but not the results of retrieval operations. In such cases the time consuming physical reorganization does not have to be carried out during an interactive session but may be postponed until resources are available (e.g. during the next night).

5.4.3 Problems of Linear Address Spaces - Storage space in computers is essentially linear; storage cells may be ordered in sequence and numbered with integers.

Most access methods are designed for accessing data based on keys in a linear data address space where an ordering relation '<' exist, that for each three keys, a,b,c holds:

- a < a
- (a < b) and (b < a) => a = b
- (a < b) and (b < c) => (a < c)

This ordering relation is relied on for decisions during search.

For two-dimensional space with coordinates represented as real numbers, there is no such ordering relation. For a representation of coordinates with limited precision using computer internal representation of REALS, there are mappings to the realm of integer where an order relation exist. (A possible mapping would be to number all possible points in two-dimensional space in a row by row fashion as:

```
301  302  303  ....
201  202  203  204  ..
101  102  103  104  ...
 1    2    3    4    5   6   ...
```

Such a mapping can be used for retrieving a specific point with given coordinate values but is not efficient for two-dimensional range queries because the address space is very large (e.g. (10**9)**2 = 10**18) and therefore only sparsely populated (e.g. 10**6 points; only one in 10**12 addresses are in use).

Range queries are characterized by at least one condition for selecting data not given as a discrete value but as a range of values. The answer to a range query consists generally of a multitude of data, each with a data value within the range. The data in the answer form a neighborhood in the address space. The spatial retrieval is a two-dimensional range query whereby the x and y location are specified as ranges.

In order to answer range queries fast, we must exploit neighborhoods in the data space (cf. Figure 6a). The mapping in Figure

7 preserves neighborhood in one direction (x) but not in the other (y).

Carnap proved that there is **no** mapping function possible to map a two-dimensional topological space to a one-dimensional preserving neighborhood. The best we can do is to find a compromise preserving neighborhood for limited areas. Dividing the address space in **fields** and clustering the data of a given field into one storage bucket maintain neighborhood within the fields but not across their boundaries. This is the basic idea of the following **Field-Tree** algorithms.

6. ACCESS METHODS FOR SPATIAL RETRIEVAL

After going thru the more theoretical background, we can start to discuss methods for spatial retrieval as they are proposed in the literature and work out a method best suited for a LIS.

To provide a framework for discussing different issues, we start admitting three fundamental simplifications which will be revoked one after another during the exposition. To begin with we shall assume

- that all data concern point-like objects,
- that objects are uniformly distributed in space
- that all objects are of the same type.

The discussion is intentionally on a very abstract level, trying not to fix details in too early a stage. This prevents us from ruling out possibilities by premature decisions.

Figures must be interpreted as examples, as alternate implementations are possible. The next paragraph will give a description of a possible implementation.

6.1 Quad-Trees

Quad-trees ([Finkel 74] - in the meantime a derived type of quad-trees for compressed storage of areas has appeared in literature) are a storage structure for fast access to data elements based on two-dimensional keys. Quad-trees are fundamentally quaternary trees (trees which branch at each level in four), each node dividing the data space in four quadrants and

all data pertaining to any quadrant are stored in the respective (smaller) quad-tree.

Quad-trees, as all the methods organizing data space, do not provide for the clustering of data needed when data are stored on mass storage devices. If data are stored in a quad-tree in an arbitrary order, quad-trees need, as do all tree structures, balancing. Deletion in quad-trees seems fairly complicated and time consuming [Samet 1980]. For large data collections access time in quad-trees grows with

(base 4) log (card (L)),

resulting for card (L) = 10xx6 in 10 accesses to mass storage for finding a point. To search for several points in a range yields better results, approaching eventually one access per point found.

As quad trees do not provide for methods for physical clustering, they do not seem adequate for storage of large data collection with the retrieval characteristics of LIS. Range queries yielding 2,000 points with one physical access per point will result in a response time of 2,000 * 30 m sec = 60 sec, which is too much for interactive work.

6.2 Dividing Address Space in Regular Fields

A very obvious method for fast spatial access consists of dividing the address space (the coordinate space which points are located in) in regular fields. For each field we collect all data elements related to points within that field and store them within a physically contiguous storage bucket. This way data of nearby points are clustered and neighborhood is at least piecewise preserved. Access is now a two-step operation:

- to find field
- to find point within field.

Different geometric figures providing a partition of the address spaces may be used for fields; most simple algorithms result if we use a rectangular grid, parallel to coordinate axes.

We may number fields in a way similar to Figure 8, and it is thus simple to calculate the number of the fields touched by a

query window. Different access methods may then be used for finding the bucket related to that field (e.g. quad-trees [Denert 1977]).

11	12	13	14	15
6	7	8	9	10
1	2	3	4	5

Figure 8

Implementation is simple. As we assume data to be uniformly distributed in space, each field will contain the same amount of data. Thus we can attribute a physical storage bucket (a number of consecutive disk blocks) to each field.

Access time is then

T = n.field * t.field + n.objects * t.object + t.set up

Where n.field = area (window)/area (field)

When we physically cluster the object data together with the data for the field in one block of mass storage, access to the field data needs time consuming physical access. The following processing of the object data is then much faster as no additional dsata has to be brought in from mass storage (thus t.object <<+.field).

Furthermore

n.object = area (window) * object-density

the total access time is therefore of the order

$$O(T) = \text{area (window)}$$
$$= \text{n.objects.}$$

This is linear in the number of objects retrieved, therefore exactly fulfilling our requirement.

6.3 Dividing Space in Fields with Varying Size

If we have to accommodate for varying distribution of data in space, fields will contain different amounts of data. This may be solved by assigning additional storage buckets to fields, which, however, degrades access time because for some fields we have now to access several buckets. This influence is strongly felt in cases where data density vary very much. In a LIS it must be expected that data density may vary more than 1:1000 between densely populated areas and areas of very extensive use.

Instead of extending the storage space for densely populated fields, we may reduce the field size (e.g. dividing a field of the first level into four smaller fields for the second level and so forth) (Figure 9). This bears some resemblance to [Burton 1978].

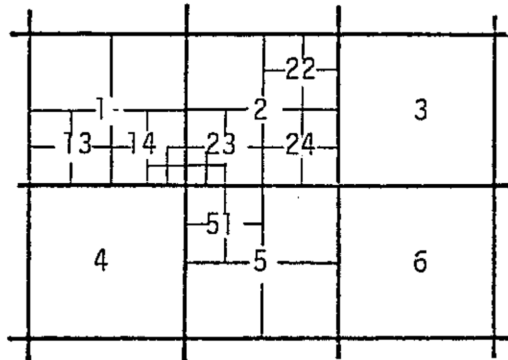


Figure 9

This allows us to accommodate even large differences in a few levels of subdivisions.

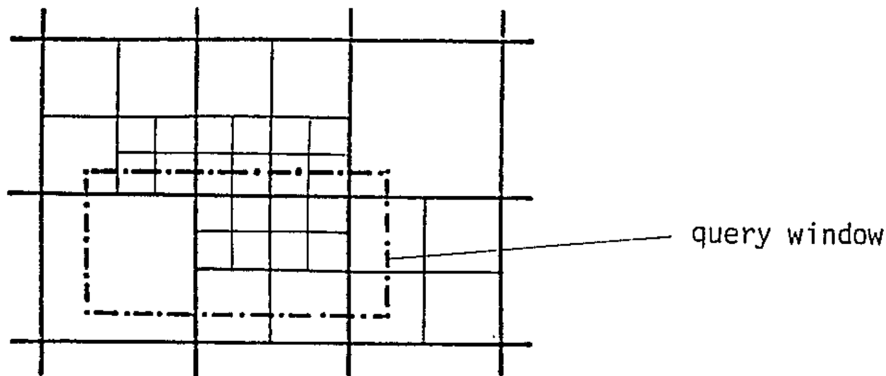


Figure 10

The access method to find the fields in a range query (e.g. area outlined in Figure 10) needs some consideration. It is still possible to find numbering schemes (e.g. as in Figure 9) which allow us to calculate all field numbers for all fields touching a query window. The set of such potential fields is large because it includes all fields of all levels potentially within the query window even if these levels are not yet used - even if only a few levels are foreseen this may sum up considerably.

For the case of square grids it is approximately

2** (area (window)/area (smallest field))

or if squares at level 1 are 1 square mile and the area of the window is 6 square miles, with 10 levels it amounts to

2 * (6/(1/4**10)) = 12 * 2**20 = 12 * 10**6.

It is then necessary to try to access all these fields to search them if they exist or to find out that they do not exist (i.e. they are empty), which takes about the same amount of time.

For access to the clusters it seems more appropriate to exploit the structural relation between the larger and the smaller

fields in a tree structure. Thus the name **FIELD-TREE** for the method. This requires that the large fields remain, even when the smaller, lower level fields are used, and it also requires that all lower level fields existing are connected to larger fields (which may somewhat limit the storage algorithms).

A range query can then be processed very similar to a range query in a quad-tree [Denert 1977]. Performance of retrieval in this structure is similar to the previous case; we know that only accesses to fields (resulting in physical accesses to storage blocks) have to be counted. The number of objects in the answer is proportional to the number of fields accessed since each field contains on the average the same number of objects. Therefore the response time is linear

$$O(T) = \text{card}\ (R)$$

in the number of data elements retrieved. The area of the window does not enter because the number of fields overlapping a window is variable and depends on the density of the data. (Small deviations from linearity are caused by the effects of the limits of the window, where fields have to be accessed which do contribute less than an average number of data elements within the window.)

6.4 Treatment of Extended Objects

To store data concerning spatially extended, non-point-like objects is not possible in the previous model. Objects may cross the boundaries of the fields and can therefore not always be unequivocally assigned to a field.

6.4.1 Cutting Methods choose the fields relatively large (approximately like traditional map sheets) and cut objects at the sheet boundaries in pieces which then can be stored within the respective storage buckets (usually files of the operating system). The advantage of this latter solution is that the drawing of map sheets is simplified as all objects are already clipped at the borders. This approach is therefore useful for the storage of graphical information as it is currently stored on paper map sheets [Wild 1980]. It is not suitable for storage of more sophisticated data collections where the geometric properties are only, although but a very important, aspect of the data stored.

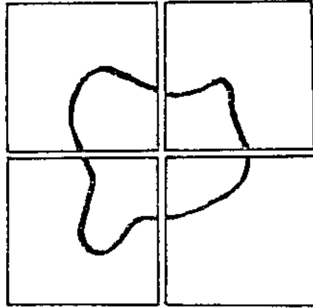


Figure 11

If we are not content with the storage and treatment of graphical data alone but are interested in exploiting other properties of the objects, this method can not be applied. The resulting objects can not be interpreted, they do not have meaning in a real world sense: what is the address of half a house?

Furthermore, it does not seem possible to program general algorithms to cut objects without knowledge of the internal representation of the objects, including but not limited to the form of the geometric description. It is also not clear what would be the best way to treat indirect spatial reference.

This method is also not flexible enough to adapt to a varying distribution of data in space. The limits of the sheets have to be chosen beforehand and can not be adapted easily. The retrieval method is unsuitable when the query window is much larger than the sheets, since a great number of sheets have to be searched for the few objects answering the query and the objects retrieved have to be recombined at the sheet boundaries.

It is not suitable neither when the query window is much smaller, since all objects within the sheet have to be acccessed and tested.

6.4.2 Storing Duplicate References - Instead of cutting objects in each field the object is touching, a reference to the object may be stored. This requires either a storage method with which objects and references to object may be stored intermixed or we store only object references within the fields and put the objects to another place [Tamminen 1981]. The latter method requires an extra (probably physical) access to storage and therefore increases the response time.

Any system which allows for multiple storage of objects (or object references) needs a filter to eliminate objects included in the answer twice. This merely requires sorting of references to object and slows processing and makes programming more complicated; as no additional data are required from mass storage, this does not increase average response time by much.

6.4.3 Overlapping Fields - If the fields we use for clustering objects overlap, we may find ways to distribute objects to fields in order that each object goes entirely in one field (without crossing that field's border).

An obvious division in fields fulfilling this requirement is given in Figure 12. If the largest field is larger than the area out of which we collect data, we are sure that every object at least fits into this largest field.

For fields containing too many objects the division method in Figure 9 is still operational, but now some small objects may not be transferred to the smaller lower level fields because they cross the boundary of the lower level field. A method to increase bucket size for a given field where too many objects have to be stored which can not be transferred down, is needed.

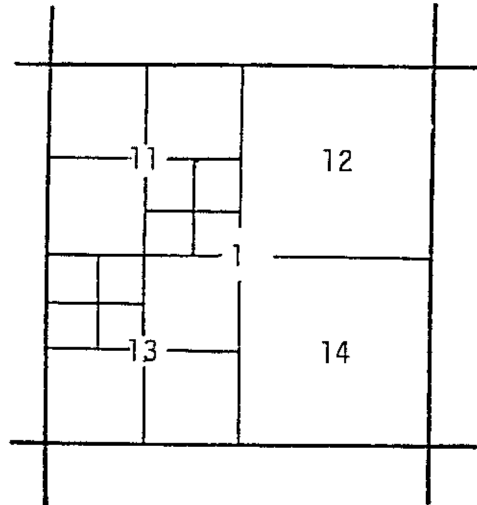


Figure 12

The method of division for fields given in Figure 12 is not optimal since any object crossing the boundaries between 11, 12, 13, and 14 has, independent of its size, to be stored in field 1. In general, a field will have an amount of such objects crossing the boundaries of the next smaller fields, proportional to the length of its side, which results in too many objects assigned to the upper level fields. As these fields have to be searched more often than smaller ones, this will increase the number of objects included in the answer, thus increasing the number of accesses to mass storage and ultimately response time.

6.5 Goals for Storage Methods Using Overlapping Fields

In order to improve performance for spatial retrieval using clustering based on overlapping fields, guidelines for estimating performance have to be established.

Other things being equal, the number of clusters to be accessed increases response time linearly. A method which for a given query window results in processing fewer fields is therefore superior.

The fewer data about objects not falling into the query window accessed during a retrieval, the better, because if more data have to be brought into main memory, more accesses to mass storage are needed. If more data are stored on smaller fields, the spatial selectivity is enhanced and therefore fewer unwanted data are accessed.

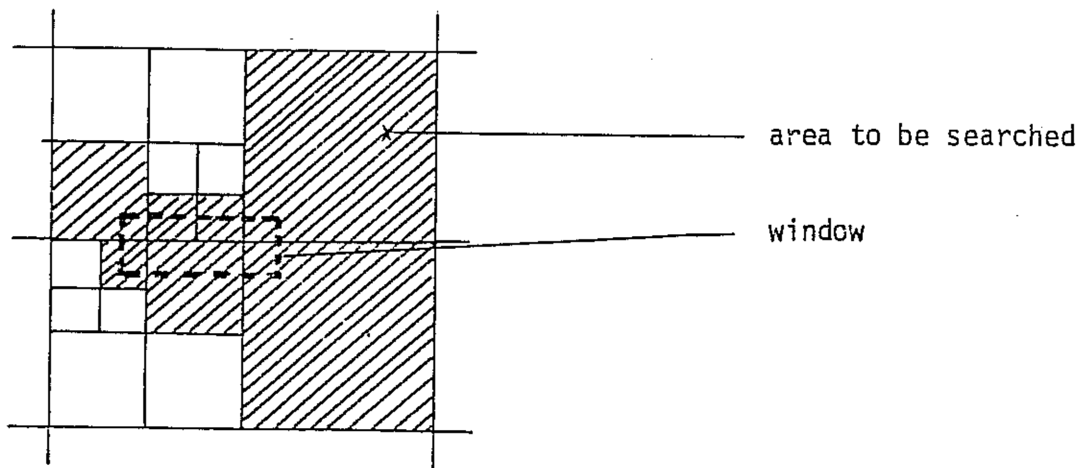


Figure 13

For a given small query window, the larger fields cover large additional areas with data unrelated to the query at hand (Figure 13). Generally, spatial selectivity of a method is better if objects are assigned to smaller fields.

6.6 Methods for Discussion

The circumscribing figure to indicate location and extension of a geometric object is described by a number of parameters. If we choose rectangles with sides parallel to the coordinate axes, four parameters are needed. These may be either the four coordinates of the sides (Figure 14) or, more useful, two coordinates of the center point and the two lengths of the sides (Figure 15).

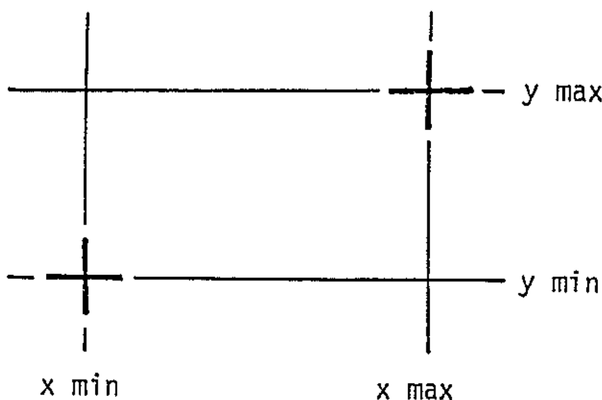


Figure 14

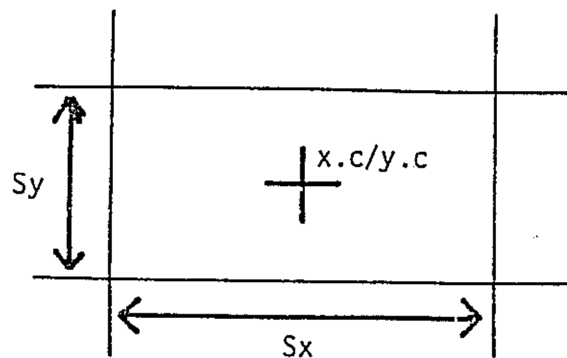


Figure 15

Each characteristic figure can thus be represented as a point in 4-dimensional space [Nievergelt 1981]. This center/side space is most useful for the following discussion.

In order to make further explications simpler, all examples are chosen from the realm of one-dimensional, extended objects, which are points in a two-dimensional center/side space and therefore drawn easily.

In fact, for the methods presented hereafter, it is possible to discuss the two projections on the x and y coordinate axes independently. Thus we can discuss these methods using two-dimensional center/side diagrams and apply the results for each dimension independently.

Example: The objects a, b, c, and d (in Figure 16) characterized by the center point - half side pair

a	(2,2)
b	(4,1)
c	(6,2)
d	(4,3)

are in center/side space points (Figure 17).

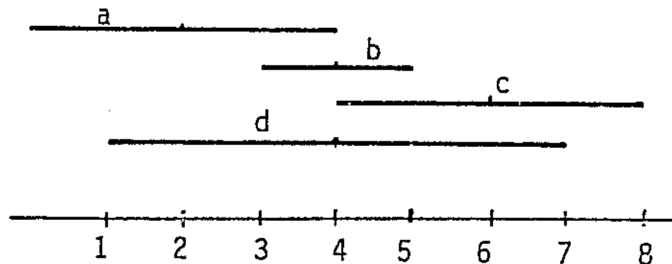


Figure 16

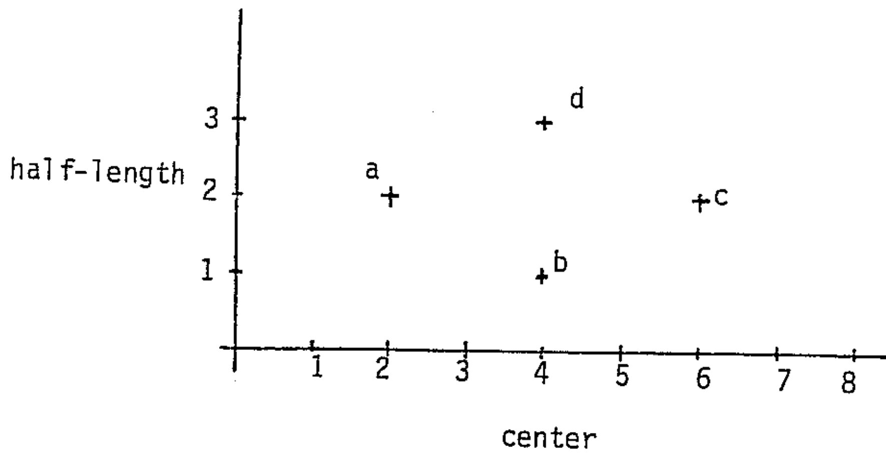


Figure 17

The spatial query is now not an ordinary range query in center/side space, but from the condition for overlapping

Object <intersect> window <> Ø

follow for each coordinate the conditions for the center-point half-side figures

$$o.x + o.s > w.x - w.s$$

OR

$$o.x - o.s < w.x + w.s$$

(Figure 18).

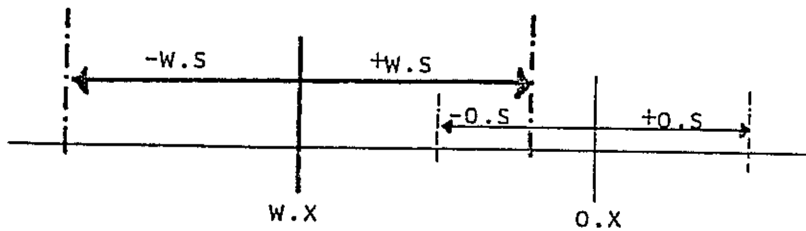


Figure 18

The derived expression

$$o.x > w.x - w.s - o.s$$

or

$$o.x < w.x + w.s + o.s$$

describes the search cone in center/side space (hatched in Figure 19 for the window 2.5 to 3.5). The similar conditions for each coordinate in two-dimensional applications have to be combined by AND.

Different division of two-dimensional space into overlapping fields together with the necessary rules how objects are assigned to fields may now be mapped as grids in center/side space. These grids may be used to assess different methods, as they make visible how many fields have to be searched and how large these are.

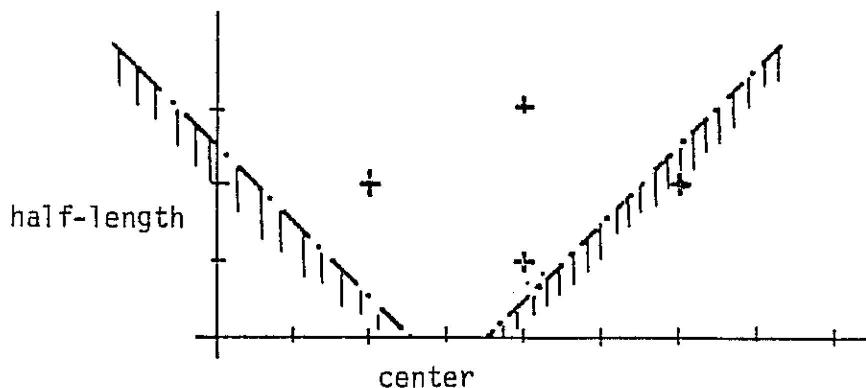


Figure 19

6.7 Basic Method for Spatial Retrieval

6.7.1 EXHASH - In [Nievergelt 1981] it is proposed to use the EXHASH storage method which is a very flexible method for multi-key data. In [Hinrichs 1982] these ideas are further discussed.

EXHASH methods use a hash type directory for access to grid cells (fields). This seems however not optimal in applications where the range of the keys is very large and the density of population varies greatly (cf 6.3). To avoid this drawback, an implementation with a few levels of indirection for the directory has been completed but no report of performance is yet available.

EXHASH is based on dynamic adaptation of fields to the amount of data stored. This dynamic adaptation occurs independently for every key axis (center point coordinates as well as side length) and may be guided by some parameters. It makes an arbitrary pattern of convex fields in the center/side space possible (Figure 2Ø). In a LIS we may expect that the number of large objects is much smaller than the number of small objects. It is not obvious how these parameters must be selected to push data towards the smaller fields to speed the special spatial retrieval (cf. 6.6).

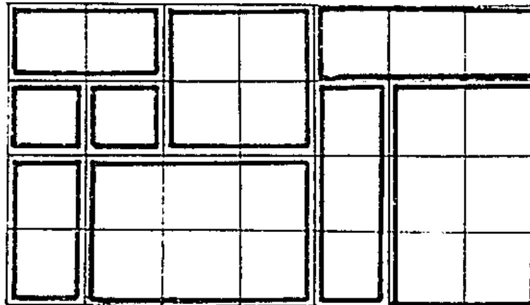


Figure 2Ø

It seems, therefore, that the more general method EXHASH may be improved by specializing for the specific spatial retrieval operation.

6.7.1 Field-Tree - Instead of dividing the center/side space in an arbitrary pattern by the way incoming data direct it, we can divide it beforehand in a way reflecting the goals mentioned before (cf 6.5).

Such a prepared fields-division also allows simpler ways of access to the clusters. The regular properties of the division

may be exploited to use more specialized and faster access methods - preferably trees.

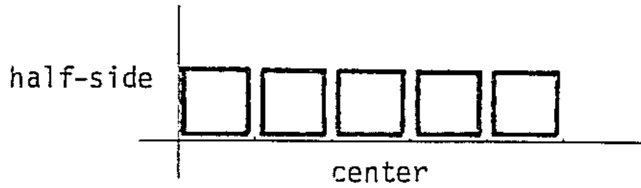


Figure 21

Figure 21 shows one level of such a division in center/side space. The ratio middle-point-distance to half-side is 1:1, which results in square grid cells (grid cells is used in conjunction with the division of center/side space, whereas fields are describing the corresponding division of real world space). The (one-dimensional) fields for that level are given in Figure 22, showing that each point is laid over by 3 fields.

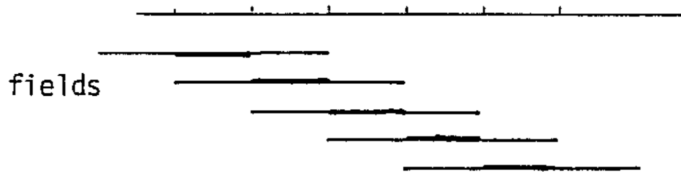


Figure 22

Applied to the two-dimensional space this means an overlay of 9; in Figure 23 are the nine fields, all overlapping the hatched area given.

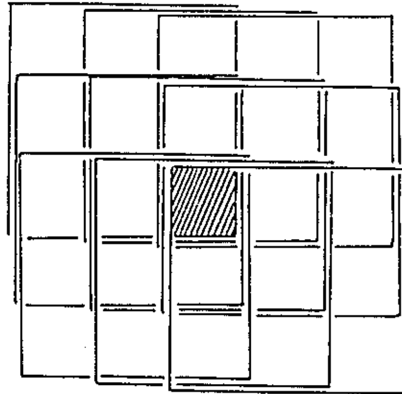


Figure 23

On the other hand we may ask what grid cell in center/side space is formed by all the objects which may be placed within a field without being cut (Figure 24 for one-dimensional space).

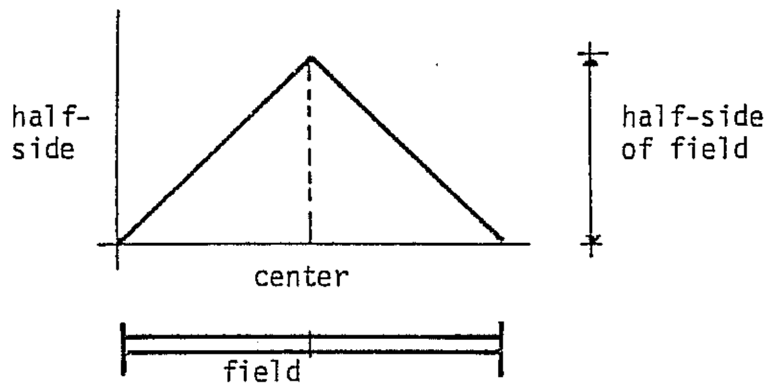


Figure 24

In order to have coverage for all objects up to a certain size, the field's triangular grid cells in center/side pace have to overlap (Figure 25).

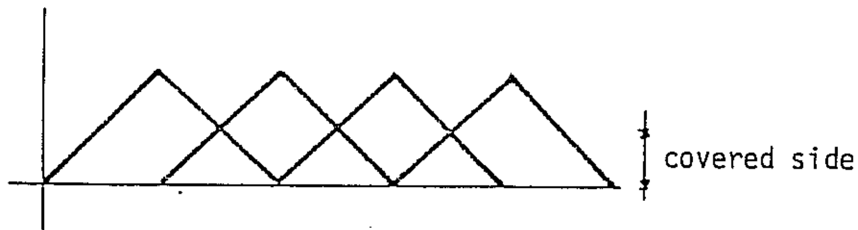


Figure 25

Different ways of assigning objects which might be placed in more than one field are possible resulting in different grids (e.g. based on left corner, on center, Figure 26).

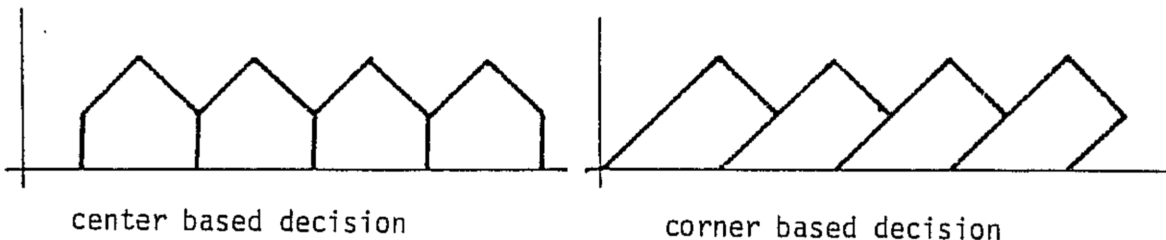


Figure 26

No principal differences between such methods are apparent.

If we look at two levels of fields, overlapping with different side lengths, we may base the decision to assign objects to the

lower level on the side of the object or on the fact that an object fits on the smaller field (Figure 27).

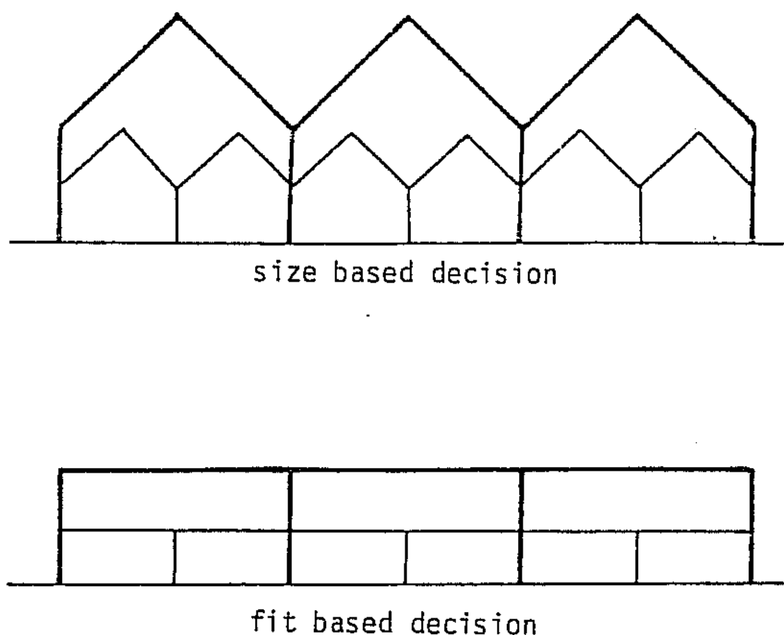


Figure 27

As apparently more objects will be forced down to lower levels, the 'fit'-based method should perform better.

Finally we can arrange the lower level grid cells in a manner such that they are always fully included in one higher level grid cell. This ensures a regular (quaternary) tree structure for the fields and facilitates the programming of reorganization algorithms.

Before it is possible to discuss the choice of the different parameters for the definition of the grid in center/side space, more investigation into the statistical properties of real world objects location and extension is required.

All methods of this type however can be expected to perform spatial retrievals in a response time which is essentially linear in the number of data elements stored for the query

window. The next paragraph will discuss an additional consideration for improvement. To conclude this paragraph a previously published method for spatial retrieval [Frank 1981] should be compared.

This method is based on an overlaying of square fields with sides doubled for each level; the overlay is, however, not symmetric (as Figure 27) but each set of fields is slightly translated (Figure 28). Any number of levels of such fields may be overlayed without the boundaries of fields of different levels ever coinciding.

For this type of division it has been proven that any object may be placed undivided in a field with a side smaller than 16 times the maximum extension of the object; this is an upper bound as most objects may, of course, be placed on smaller fields.

The previously used analysis tools applied to this method reveal a number of complications resulting in clumsy programs.

Projected on one dimension and transformed to center/side space (Figure 29), the grid shows that some object may only be placed in fields of level 1 and 3, but not in level 2. This results in cases during reorganization when an object may not be transferred from level 1 to 2, but only directly to level 3 ("jumping over a level"). If we take into consideration the combined effects of both coordinate axis which are not independently treated in this method, even great 'jumps' are possible.

Nevertheless, this method was fully programmed and tested and resulted in performance in the expected range. The programs written in COBOL, however, were long clumsy and difficult to maintain (approximately 3100 lines of code). This may be compared to the approximately 400 lines of PASCAL code given in the appendix, fulfilling essentially the same role.

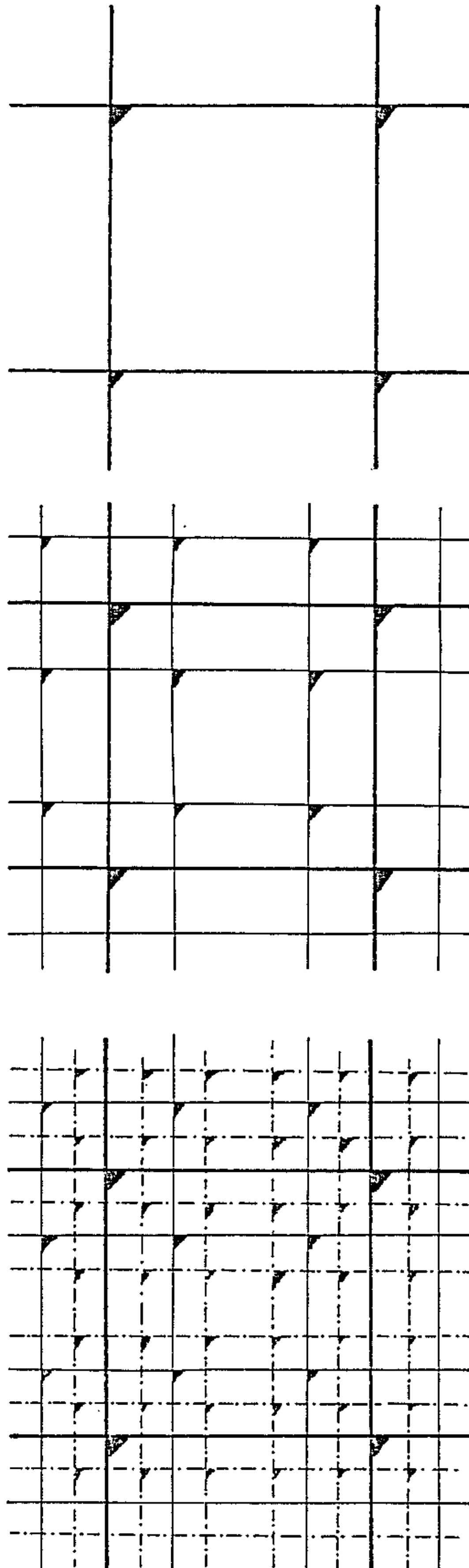


Figure 28

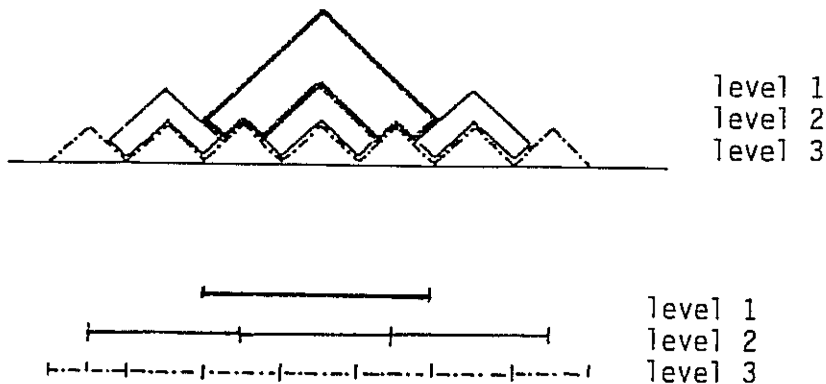


Figure 29

6.7.3 Improvement Using Types - In 4.3 it was pointed out that type based conditions may have a high selectivity in the spatial retrieval. It is important to speed up inquiries concerning large areas but only requiring data of a few landmarks. The previously developed method results in slow response in such cases, as the response time is essentially linear in the amount of data stored for the area of interest. This is slow for queries regarding large areas, even when these retrieve only a few data elements of landmarks.

It is reasonable to expect that queries for large areas will select the data to be retrieved in first line on data type (e.g. 'RETRIEVE ALL counties WITHIN Maine', or'RETRIEVE ALL school-buildings WITHIN Penobscot County') and that the types of data which will be required in such 'overview' retrievals can be identified beforehand.

For each data type an 'importance' is fixed which indicates the area which we expect data of this type will influence. This importance number fixes the minimal side of the field data of this type may be assigned to.

For retrieval it is then not necessary to search all fields to the bottom of the tree but only down to the minimal size of fields as indicated by the importance of the type searched for (as conventional in computer science, 'up' in the tree is the root and the leaves - our fields - are 'down'). The response

time for such queries is then only linear with the amount of data stored for that area with size or importance larger than this cut-off limit. If the importance numbers are well chosen, this will result for usual queries in about the same amount of data and therefore in constant response time.

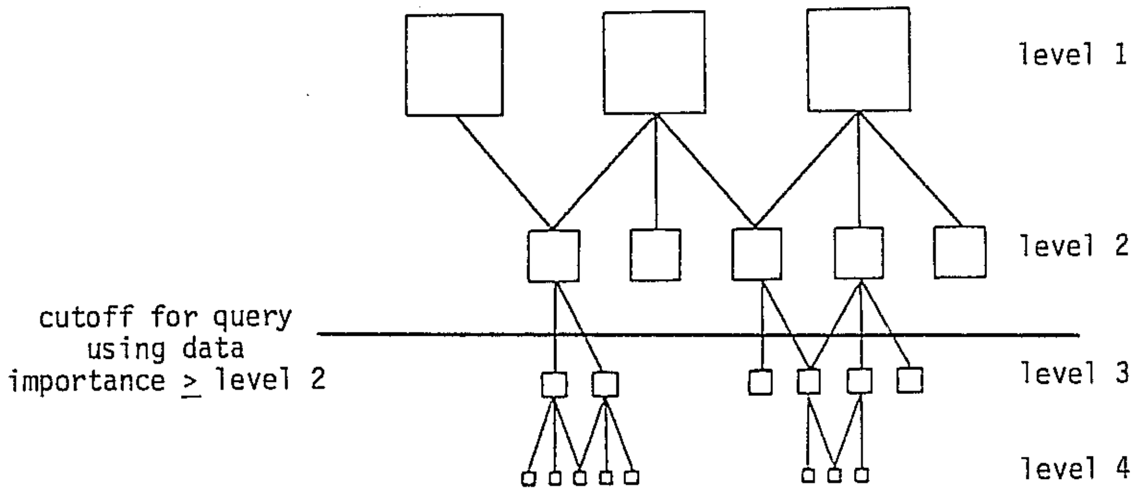


Figure 30

6.8 Storage and Retrieval

After the extensive discussion of the storage structures, the formulation of the algorithms used for storage and retrieval is simple.

6.8.1 Storage - Each object is always stored on the smallest field possible. The smallest field possible means:

- the object is not cut by the field boundaries
- the field size is large enough for the object's importance (of 6.8)
- the field exists.

This leads to the algorithm

1. Descend the tree as long as a smaller field exists, in which the object fits and which is larger than the importance of the object demands.

2. Store the object on that smallest field.

This in itself would never make the tree of fields grow, so we have to add a rule, when to open a new field:

- If the tree is empty open the first, largest field; this is the root of the tree.

- If the field on which to store the object contains already a certain number of objects (i.e. the data for the new object will not fit in the storage bucket), reorganize the field.

Reorganization is the means by which all fields are equally filled with data; if a field becomes overfilled because more data about that part of the world is stored in the LIS, it has to be divided into fields of the lower level and the data must be distributed on these smaller fields provided it 'fits' there in order to maintain the rules stated about placement of objects.

The division of fields and the ensuing distribution of data previously stored with the larger field to the smaller ones reduces the amount of data stored there, opens new fields and makes therefore the tree grow.

To assure best performance of the algorithm all fields should be filled exactly. Division of fields guards against overfilling. Fields which contain too few data have a deteriorating influence on performance too. Provisions in the division procedure may check that only those fields of the lower level are opened for which a certain amount of data are already stored. (This results in grid cells in center/side space which are not convex; it has no negative effect but marks a difference to the EXHASH method which relies on convex grid cells.)

With overfilling and underfilling limits well chosen, average storage utilization may be kept reasonably high.

With the experiments carried out we have not been able to formulate the relationship among reorganization strategy, storage utilization, and performance. The new system now built should facilitate more revealing tests.

As the overfilling of a field is only a defect in physical clustering but not in the logical data organization, spatial retrieval yields correct answers in spite of overfilled fields. We can, therefore, postpone reorganization and set a mark only. The effective reorganization may be carried out later (e.g. at night when computer utilization is low).

6.8.2 Retrieval - The recursive algorithm for retrieval is simple.

1. Initialize: start with the root of the field tree

2. Test a field (recursion)
   If the field under consideration is above the minimal level of importance for objects which must be retrieved, test if the field touches the query window, if yes,

   - the objects on the field must be included in the answer and

   - all sub-fields of this field must be tested

7. EXAMPLE IMPLEMENTATION

This paragraph explains an example implementation First the specific method choosenand the corresponding parameters are explained and then the most important points in the modules discussed. Before the central module can be presented, a number of logically lower level auxiliary modules are introduced. The PASCAL code of these modules is available from the author.

7.1 Specific Method

For the circumscribing figure a rectangle with sides parallel to the coordinate axes was felt to result in simplest code. The fields were chosen as squares, with a level Ø field of side length 8Ø km (coordinates throughout are expressed as integers, unit is one millimeter).

The ratio of distance of middle-points of fields to half side of fields was chosen as 2:1.

The decision where to place an object is based on the left lower corner of the objects and on the fact that an object 'fits' in the field.

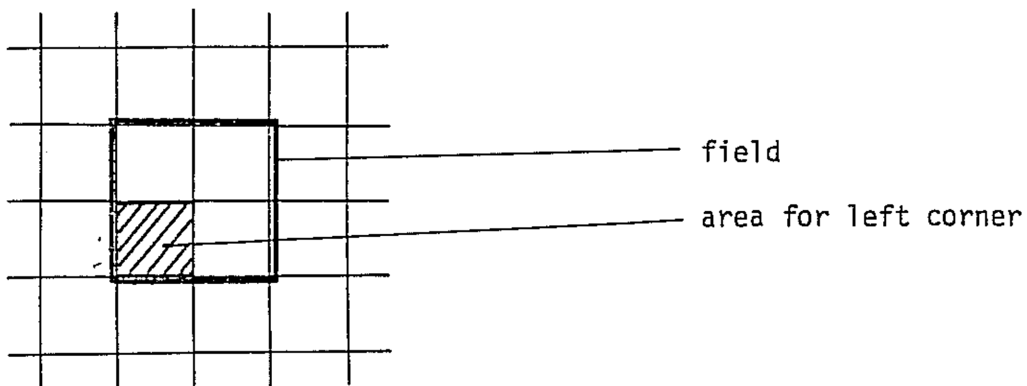


Figure 31

As a rule this says:

each object is placed in the smallest field

in which it may be contained undivided
and its left lower corner is therein.

This field has for no object a side length larger than the maximal side length of the object.

As long as fields are not overflowing, objects may be stored on larger fields.

Retrieval is guided by the rule:

Check all fields which touch the search window.

For experimental purposes the number of objects per field (overfill limit) was set very low (10) and reorganization was not allowed to open a new lower level field before more than 5 objects to be placed there were already stored on the larger field. These are arbitrary settings whose effect on performance will be discussed below.

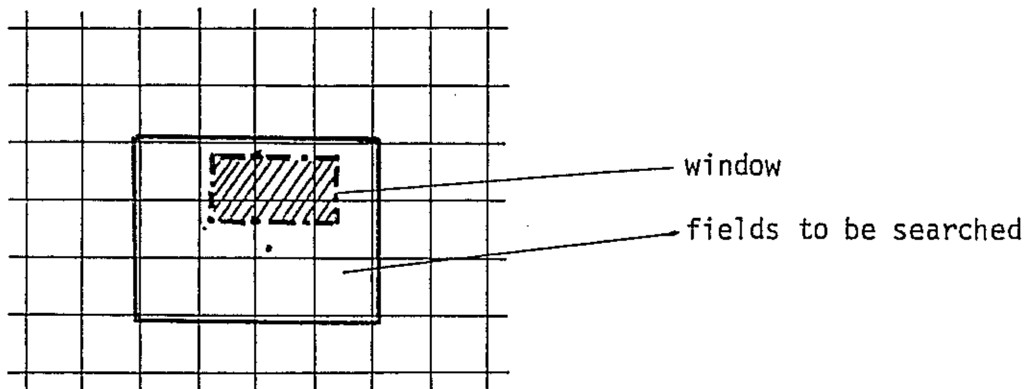


Figure 32

7.2 RAUML

The functions of this module are used for storage and retrieval of objects, as described in 4. These are the highest level functions and may either be used directly by the application program or be built in a generalized database management system.

Three operations provide for storage, update and deletion of objects for which access based on location is required.

rIn: Store the object with given viereck
rDelete: the object from the spatial access system
rChange: change the objects viereck and move the object to the correct grid cell

Two functions are used for retrieval. The first one to initialize the search, passing the query parameters like window, and object-types required. The second function must be called as long as it returns the value true - it delivers then for each call one object. This seems to be the simplest way to implement a function which returns a varying, but large number of objects.

rSearchInitialize: set parameters like window (a viereck), set of object-types requested, and initialize the search

rSearch: as long as TRUE, returns an object for each call

The semantics of these operations are that rSearch returns all the objects previously stored and touching the query window set by rSearchInitialize.

The implementation can be as follows (to keep explications simple 'importance levels' and object-type related considerations are left out):

```
rIn:     find smallest existing field on which to place the
         object (use fFind)
         while there is no room for the object (fSpace)
             distribute the objects on this field (rDistribute)
                  find the smallest existing field again
                  (this may have been changed thru the
                  growth of the tree by distribution)
```

Store object on that field (establish a logical connection between the field and the object and cluster the objects of one field physically).

update free space of field (fStore).

rDelete: destroy the logical connection between the object and make its space available for further use (fDelete)

rChange: do a rDelete followed by a r/n.

rSearch: controls the use of two internal operations, one to get the net field (fNextField) and one to get the next object on the present field (rNext Object)

The code for rSearch would be essentially two nested loops

```
while fNextField do
      While rNextObject do
            treat object;
```

if it were not necessary to return with each object to the caller; this is a typical application for Jackson's program inversion.

rSearchInitialize: Initializes the query window and initializes the search in the field-tree (fSearchInit).

rDistribute: Is an internal routine which distributes the object of a field onto the smaller subfields. An implementation may choose to count first the objects which may be pushed down (fWhichPart) and then only open the fields which will receive a minimal number. fAddField is called to create the new fields.

7.3 FIELD

The module FIELD manages the fields.

fStore:
fDelete:
fSpace: are used to control the space utilization on the field

fFind: Starts with the field tree's root and descends the tree until it finds the smaller existing field where the given viereck fits (uses fSubField at each mode to select the next lower field).

fWhichPart: uses gWhichPart to decide into which subfield of a field a given viereck fits (if any).

fSubField: uses gWhichPart to find out into which subfield a viereck fits and returns then that subfield.

fSearch: is similar as rSeach returning each time called the next field in the tree. It is basically a tree walking algorithm as described in [Knuth 1973], where for each field gInWindow is called to decide where this field (and all its subfields) have to be included. A recursive formulation would be:

```
fSearch (field, window):
   process field;
      for each possible subfield of field
         if subfield exist then
            if gInWindow (subfield, window) then
               fSearch (subfield, window).
```

Coding complications arise from the fact that each field found has to be handed back to the caller; more sophisticated forms of Jackson's inversions are needed.

fAddTree: adds a subfield to the tree (use gPartGrid to make the respective grid cell).

FIELD uses throughout the services of QTREE to move in the quaternary tree structure and to maintain it.

7.4 Module GRID

GRID contains the code to manipulate the grid cells. Grid cells are described by center point location and level. For a given set of parameters, the relation between level and side length is fixed.

The functions and procedures used externally are:

gViereck: returns the field of the grid cell as a viereck

gInWindow: tests of a grid cell touches a window and has therefore to be searched for objects (uses gViereck and vierCut)

gWichPart: decides in which smaller grid a viereck may fall find which quarter of the grid cell the lower left corner of the viereck falls in, then check if the upper right corner is contained in the same grid

gPartGrid: makes a specific smaller grid

Internally GRID uses two functions

gVier: to return the viereck describing the area where the left lower corner of an object has to be placed (the decisions in this implementation are left corner based cf. Figure 26)

gExtend: to return the viereck indicating the area where the right upper corner of the object may lie;

The viereck returned by gExtend is few times as large and includes the viereck from gVier (cf. Figure 31).

7.5 Module QTREE

This module manages the tree of grid cells. It is based on a standard, CODASYL-like database management system [Frank 1982]. In such systems it is not allowed to connect many entities of one type to one entity of the same type and therefore trees must be represented by a two entity-type structure, one entity-type for the tree's leaves, another for the downward connection (Figure 33). If the connecting entities are stored on the same page as the upper level fields, a test if a subfield exist is possible without accessing any additional page.

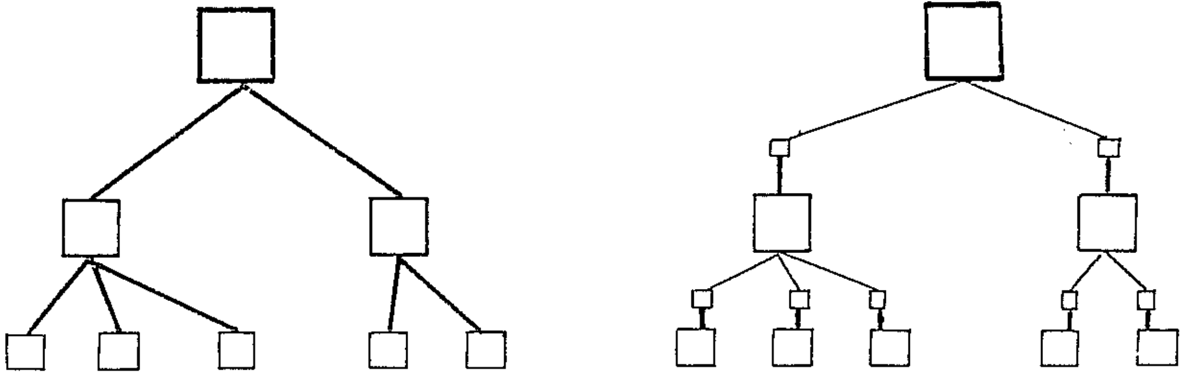


Figure 33

This is however completely hidden in this module and the adaptation to another database system should only need changes here. The functions and procedure needed are

gRoot: find the root of the tree of fields

gNextLower: return in turn each of the grid description of the lower level grid cells

gGetNext: access a lower level grid cell

gAdd: add a grid cell as a new leafe to a given one in the tree

gupp: } gupf: to move up in the tree, one to find each of the two entity-types needed.

7.6 Module VIERECK

VIERECK is the data abstraction for the circumscribing figure for the object geometry and at the same time for the query window.

The functions and procedures used here are:

vierMake: make a viereck given two opposite corner points

vierCornerLL:
vierCornerUR: for a given viereck return a corner point, either lower left or upper right

vierInVier: test if a viereck is within another one

vierCUT: test if two viereck are overlapping.

7.7 Results

The test implementation mainly confirmed the general principles as they were laid out in paragraph 6. For testing purposes the bucket was chosen extremely small (10 objects) and the minimal number of objects required to open a subfield high (5 objects). These settings resulted in a high utilization of storage space; on the average 65% of the space in a bucket was occupied. The additional cost for maintaining the structure of the field tree, namely the distribution of objects of filled fields to subfields seemed relatively small - only 80% of the records stored were affected by distribution operations. This means a storage operations cost on the average twice as much in a field tree as in a storage structure without spatial access path.

Performance for retrieval showed an average of more than 50% of the objects brought into main memory useful. With a more practical bucket size of e.g. 100 objects per field, this will result in retrieval of 2000 objects for a screensized map in about 60 accesses to mass storage (2.. x20 seconds depending on hardware and the operating system).

Additional research is needed before the influence of the differentt parameters can be quantitatively predicted, and guidelines for optimal selection can be established.

8. CONCLUSION

This discussion has first elaborated on the general properties of space-related data collections. It was possible to identify abstract properties of data stored in such a system. Based on the properties, generalized operations, namely space related storage and retrieval operations, were formulated.

The second part investigated which storage structures will support fast and efficient retrieval of space-related data. Finally, a method is described which implements this method with fairly simple programs. By choosing values for certain parameters, this method can be adapted to the statistical properties of specific land related data. Not enough investigations have been carried out so that systematic advice on the selection of these parameters could be given.

As the method is based on very general properties of data, it can be easily adapted to any type of data where spatial retrieval is needed. It is best to incorporate it in a generalized database management system. The author is currently building such a system which runs on different computers from mainframe to supermicro. Results so far are encouraging enough to continue this work.

GLOSSARY

Object — Data describing a real world object of which location and spatial extension is known. Objects can be retrieved based on location.

Field — An area in real world space data of which are stored together. Fields are divided in subfields.

Grid Cell — An area in center/side space which determines clustering of data; each grid cell corresponds to a field.

Bucket — Contiguous space in mass storage, used to store the data of objects in a cluster. (Sometimes also called 'page'.)

REFERENCES

Bentley, J.L., Friedman, J.H. Data structures for range retrieval ACM Computing Surveys, Vol. II No. 4, December 1979, p. 397.

Burton, W. Efficient retrieval of geographical information on the basis of location. In [Dutton 1978].

CODASYL Data Base Task Group (DBTG) Report, April 1971.

CODASYL DDL Journal of Development, June 1973.

CODASYL Data Description Language Committee, Journal of Development, January 1978.

CODASYL Draft Specification of a Data Storage Description Language, BCS/CODASYL DDLC Data Base Administration Working Group, 1978.

Digital Equipment Corp., Data Base Management System (DBMS-10) Programmer's Procedures Manual, Maynard, MA 1977.

Digital Equipment Corp., Data Base Management System (DBMS-10) Administrator's Procedures Manual, Maynard, MA 1977.

Denert, E., Franck, R. Datenstructure, Mannheim, Bibliographisches, Institut 1977.

Date, C.J. An Introduction to Database Systems. Addison Wesley Publishing 1977 2nd edition.

Dutton, G. First International Advanced Study Symposium on Topological Data Structures for Geographic Information Systems. Harvard Papers on Geographic Information Systems, Harvard University, Cambridge, MA 1978.

Dutton, G. Navigating ODYSSEY, in [Dutton 1978].

Everest, G. C. and Lawrence, D.T. Comparative Survey of Data Base Management Systems, on Microcomputers. ACM SIGSMALL Newsletter, VOl. 7, No. 2, Oct. 1981, p. 77.

Fagin, R., Nievergelt, J., Pippenger, N., Strong, H.R. Extendible Hashing - A fast access method for dynamic files. ACM TODS, Vol. 4, No. 3, Sept. 1979, p. 315.

FIG Commission 3. Resolutions. FIG Congress 1981, Montreux, Switzerland.

Finkel, R.A., Bentley, J.L. Quad Trees; A data structure for retrieval on composite trees. Acta Information Vo. 4, 1974.

Frank, A. Applications of DBMS to Land Information Systems. 7th International Conference on Very Large Data Bases, Cannes, France 1981.

Frank, A. PANDA. Bericht, Institut fur Geodesie und Photogrammetrie, ETH, Zurich, 1982.

Frank, A. PANDA A Pascal Network Database Management System. SIGSMALL Conference 1982, Colorado Springs, August 1982.

Frank, A. Conceptual Framework for Land Information System - A First Approach. Paper presented at the FIG Commission 3 meeting, March 1982, Rome.

Guttag, J. Abstract Data Types and the development of data structures. Comm. ACM, Vol. 20, No. 6, June 1977.

Herot, C. Spatial Management of Data. ACM TODS, Vol. 5, No. 4, 1980.

Hinrichs, K., Nievergelt, J. Managing Geometric Objects with the GRID file. Working paper, Institut fur Informatik Swiss Federal Institute of Technology, Zurich, Switzerland 1982.

Knuth, D. The Art of Computer Programming, Vol. 1 Fundamental Algorithms, Vol. 3 Sorting and Searching, Addison Wesley, 1973.

Isner, J. A FORTRAN Programming Methodology based on Data Abstraction. Comm. ACM, Vol. 25, No. 10, Oct. 1982, p. 686.

ISO/TC917/SC16 N227. Reference Model of Open Systems Interconnection. Version 4 as of June 1979.

Lee, D.T., Wong, C.K. Quintary Trees: A file structure for multidimensional database systems. ACM Transactions on Database Systems, Vol. 5, No. 3, Sept. 1980, S. 339.

Lee, D.T., Wong, C.K. Worst-case analysis for region and

partial region searches on multidimensional library search trees and balanced quad trees. Acta Inf. g 1977, p. 3.

Liskov, B., Zilles, S. An introduction to formal specifications of Data Abstractions. In Yeh, R. (Ed.). Current Trends in Programming Methodology, Vol. 1, Prentice Hall, 1977.

Lueker, G.S. A data structure for dynamic range queries. Proc. 19th IEEE Symposium Foundations of Computer Science, 1978, p. 8.

Martin, J. Design of Man-Computer Dialogues. Englewood Cliffs, Prentice-Hall, 1973.

Nievergelt, J., Hinterberg, H., and Sevcik, K.C. The GRID File: an Adaptable, Symmetric Multi-Key File Structure. Institu fur Informatik, ETH Zurich, 1981.

Parnas, D.L. On the criteria to be used in decomposing systems into modules. Comm. ACM, Vol. 15, No. 12, Dec. 1972. p. 1Ø53.

Parnas, D.L. A technique for software module specification with examples. Comm. ACM, Vol. 15, No. 5, May 197, p. 33Ø.

Salton, G., Wong A. Generation and search of clustered files. ACM TODS Vol. 3, No. 4, Dec. 1978.

Samet, H. Deletion in two-dimensional Quad Trees. Comm. ACM. Vol. 23, No. 12, Dec. 198Ø, p. 7Ø3.

Schkolnick, M. A clustering algorithm for hierarchical structures. TODS, Vol. 2, No. 1, March 1977, p. 27.

Shamos, Michael I., Bentley, J.L. Optimal algorithms for structuring geographic Data. In [Dutton 1978].

Tamminen, M. Expected Performance of some Cell Based File Organization Schemes. Report HTKK-TKØ-B8, Helsinki University of Technology, Espoo 1981.

Tamminen,, M. Efficient spatial access to a database. Proceedings, ACM SIGMOD Conference 1982.

Ullman, J.D. Principle of Database Systems, Computer Science Press, 2nd ed., 1982.

Wedekind, H., Harder, T. Datenbanksysteme II. Reihne Informatik 18, BI Wissenschaftsverlag, Bibliographisches Institute, Mannheim, 1976.

Wild. Interactive Photogrammetric Data-Base and Mapping System. Operator's Procedure Manual. Wild (Heerbrugg), 1980.