The Generalized Grid File: Description and Performance Aspects

Henk Blanken, Alle IJbema, Paul Meek, and Bert van den Akker

University of Twente, Dept. of Computer Science
P.O. BOX 217, 7500 AE Enschede, The Netherlands

Abstract

The grid file offers multi-attribute access and is a structure proposed for geometrical applications. This means that so-called point, partial match, partial range and range queries for the concerned attributes are supported. In this paper the generalized grid file (GGF) will be described, offering fast access for a single attribute too. The performance of the GGF structure is evaluated for the business environment and the results look promising. As such, the generalized grid file seems to be a good storage and access structure for computer integrated manufacturing (CIM), an environment where business and geometrical applications meet.

1. INTRODUCTION

Currently available database management systems (DBMSs) are suitable to support business applications like planning and accounting. In this environment a typical query, formulated in relational terms, might request all tuples of a relation having a given value for a certain attribute ("give the component with component number = 37"). These queries are called *point queries*. Also *range queries* where the attribute must have a value within a certain interval ("give components with a length between 15 and 30") occur frequently [16]. In order to process queries of this kind so-called *single-attribute access* is needed. Accessing a database can be speeded up by offering access paths. Among them index structures (and especially B^+-trees) are very popular.

New database applications involving geographical and design data are becoming more prominent [5, 12, 14, 18]. In those applications, geometrical objects (points, edges, faces) have to be represented and stored. Often users issue values for two or more attributes and request all tuples having the issued values for those attributes ("give the component(s) with a length of 13 and a width of 7"). In order to process such a point query *multi-attribute access* to the database is necessary. Range queries ("give the components with a length between 10 and 20 and a width between 10 and 25") occur probably even more frequently in the design than in the business environment. One of the more popular structures to speed up multi-attribute access is the so-called grid file [19, 17, 13, 20, 21].

One important way to achieve fast query processing is *clustering* [3, 7] meaning that tuples having an equal or almost equal value for a selected attribute are physically stored close to each other. So clustering can be implemented by *ordering* of a stored relation on the attribute concerned. Assuming that storage redundancy must be prevented, only one clustering can be prescribed for a relation. It is important to notice that not all access structures allow clustering, but that grid files and B^+-trees do.

In this paper a so-called *generalized grid file* (GGF) will be described. This structure has a parameter that defines the number of attributes for which an access path is offered (the GGF to support single-attribute access is noted as GGF(1)). It appears that the GGF behaves like a B^+-tree if single-attribute access has to be supported and like a grid file if the number of attributes is greater than one. In the literature many access structures have been defined, each with its strong and weak points. No structure is superior under all circumstances. It is generally accepted, however, that in the business environment the B^+-tree is a reasonably efficient 'overall' access structure. In the same way the grid file seems to be suitable for geometrical applications. In Fig. 1.1, which is derived from [15], a typical data flow in a company is shown. In the figure a geometrical and a business-oriented data flow are distinguished. Integration of data needed for product design and for the traditional database applications is typical for the Computer Integrated Manufacturing (CIM) environment. As the GGF can act like a B^+-tree or a grid file, the GGF seems to be a serious 'candidate' structure for databases supporting both kinds of applications, hence for the CIM environment.

After the introduction of the GGF (section 2), this paper treats sequential access as offered by the GGF (section 3). It appears that a dense as well as a non-dense index (offering sequential access for the indexed attribute) can be implemented easily using a GGF(1) structure. Section 4 reconsiders the index selection problem, that is the problem of selecting an optimal set of indexes for a file given a certain load (queries and updates). An index has the property of improving the retrieval and slowing down the updating process. The latter is a serious drawback, hence in the business environment a major problem in physical database design is the selection of an 'optimal' set of indexes. In section 5, a version of the GGF is discussed that uses less memory than the one described sofar. It appears that the optimized GGF(k) is a very good implementation of the k-d-B-tree, an other access structure for the geometrical environment [22]. Finally, some conclusions finish the paper.

2. GENERALIZED GRID FILE

In the sequel we consider a COMPONENTS relation with the attributes component number (CNO), component name (CNAME), component weight (CWEIGHT), component length (CLENGTH) and component width (CWIDTH). Suppose this relation is ordered (which implies clustering) on CNO. To the stored relation a directory is added and together they form a

CH2840-7/0000/0380$01.00 © 1990 IEEE

GGF(1) (see Fig. 2.1). The two-level directory contains pages on which two lists are stored: one list contains CNO values (and is called 'scale') and the other contains pointers. In order to obtain for instance the tuple with CNO = 37, the shaded pointers in Fig. 2.1 can be followed.

Now the GGF(2) is introduced. Suppose that the COMPONENTS relation is not clustered on one but on two attributes, for instance CWIDTH and CLENGTH. In Fig. 2.2. a one-level directory for the COMPONENTS relation is shown. If one requests the tuple(s) having CLENGTH=13 and CWIDTH=7, then the scales are used to find the shaded pointer in the two-dimensional pointer-array. This pointer gives access to the page with the tuple(s). It may happen that more than one tuple exists with the mentioned values; then more than one page may contain such tuples causing a certain value to occur more than once in a scale.

When a data page overflows due to insertions, a new data page is claimed and half of the tuples are moved to the new page (this is called a *page split*). In our example (see Fig. 2.3) the tuples with CLENGTH <= 11 remain on the 'old' page, while those with CLENGTH > 11 are moved to the 'new' page. Moreover the directory has to be updated, meaning the addition of an extra row to the pointer-array and the insertion of the value 11 into the CLENGTH scale. Some pointers of the directory refer to the same page; the part of the <CWIDTH,CLENGTH> space that is mapped to the same data page is called, as known, a *region*. Due to insertions the one-level directory may become larger than one page. Then the directory has to be stored on two pages. Moreover, an extra page (the root page) must be introduced, hence the directory becomes two levels deep. Also the root page contains two scales and one 2-dimensional pointer-array. In the beginning the pointer-array of the root page contains just two pointers, one for each of the second level pages. When more pages are added to the second level, the root page will contain more information. In Fig. 2.4. a schematic picture of a *two level* directory of the GGF(2) is given (see also [13]). A region is represented here as a dotted rectangle. Often the root page will be stored in internal memory, while (some or all) second level pages remain on external memory. Besides a second level also a *third level* may be added, and so on. A simple computation shows that a third level will only be necessary for very big files.

So the GGF consists of a directory and a data file. The directory is a tree of *arbitrary depth* with pages as nodes. Each page contains n scales and an n-dimensional pointer-array (n>=1). A pointer refers to either a lower level directory page or a data page. The speed of handling partial match or partial range queries remains equal, no matter which attributes participate in the selection condition, so so-called *symmetrical access* is obtained. Finally, the GGF is *flexible* as an easy adaptation to growing and shrinking data volumes is achieved.

In [22] the *k-d-B-tree* has been described. This k-dimensional access structure is also tree structured and the nodes consist of region, point and data pages. The point pages contain pointers to tuples on the data pages and form a kind of indirection between region and data pages. Neglecting this (not essential) difference, there is a great similarity between GGF and k-d-B-tree. In the k-d-B-tree each region page contains a description per region and a pointer to the lower level (region or point/data) page. It appears that a region page has the same function as a directory page of a GGF. Let us concentrate on the description of the regions in a region page. In [22] no implementation of such a description is given, while in fact the GGF does give such an implementation: a region is represented

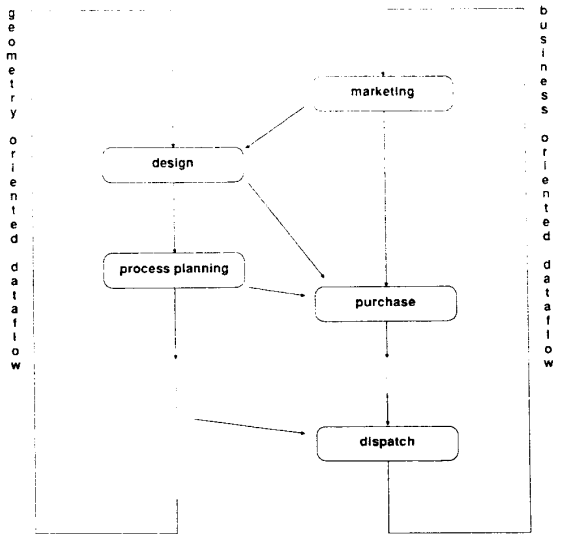

Fig. 1.1 A 'characteristic' dataflow within a company.

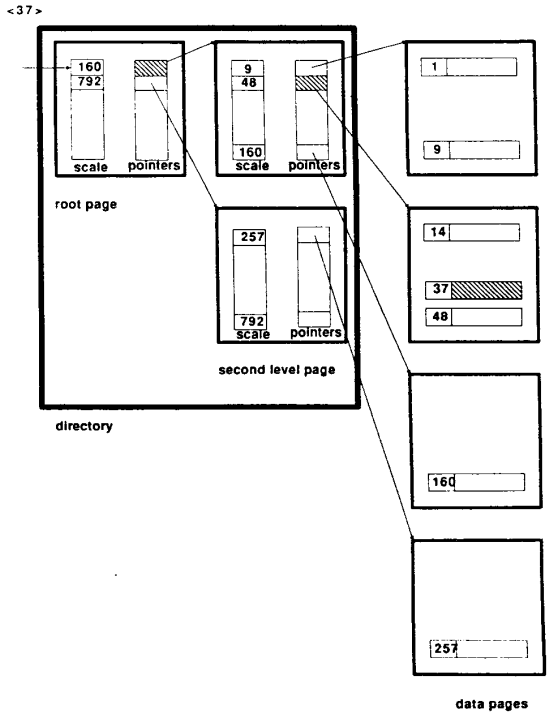

Fig. 2.1 GGF(1) for components relations (clustering on CNO).

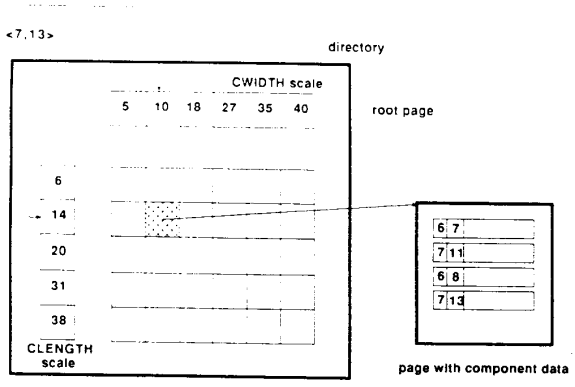

Fig. 2.2 GGF(2) for component clustering on CLENGTH and CWIDTH (one level directory).

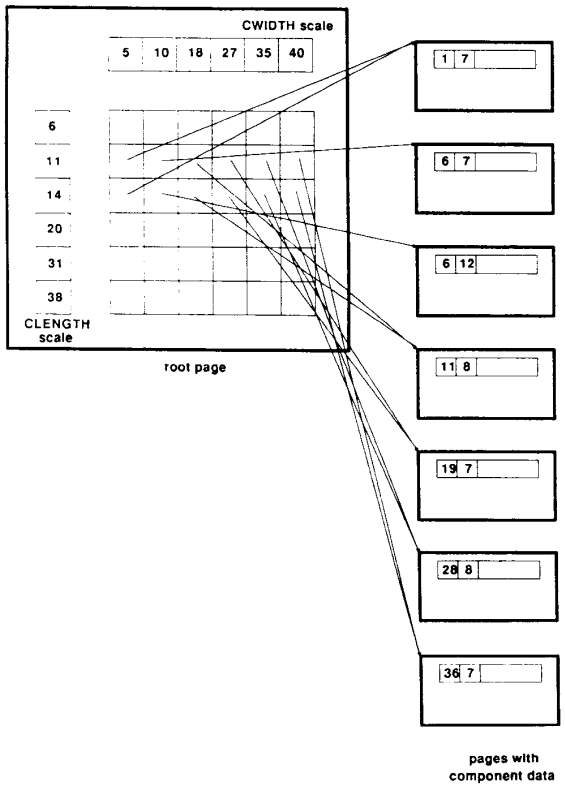

Fig. 2.3 Splitting of a data page.

by those pointers in the pointer-array referring to the same (directory or data) page, see the dotted rectangulars in Fig. 2.4. So the *GGF(k) is an implementation of the k-d-B-tree!*

A consequence is that the GGF-algorithms for range queries (including point, partial match and partial range queries), insertions and deletions of tuples have much in common with those described in [22]. For completeness' sake, they will be given in Appendix A. Moreover, topics treated in [22] like splitting strategy, merging of regions and reorganization hold for the GGF as well.

3. SEQUENTIAL ACCESS

Sequential access for an attribute means obtaining (a part of) the tuples *ordered on* that attribute. Sequential access has a variety of applications in processing queries; think for instance of range queries, equi-join queries or queries with an ORDER BY and/or GROUP BY clause.

In this section, three cases will be distinguished. The first case concerns sequential access for the attribute on which the tuples of a GGF(1) structure are ordered. In the second case sequential access is considered for an attribute on which the GGF(1) is *not* ordered. In the last case a relation is stored in a GGF(2). The question is whether fast sequential access can be offered to the attributes on which the tuples are clustered.

3.1 Ordering attribute of GGF(1)

Fig. 2.1 shows a file with COMPONENTS tuples ordered on attribute CNO. The file is enriched with a directory that is a non-dense index for CNO. The file and the directory constitute a GGF(1) as explained in section 2. It is clear that sequential access for CNO is very fast.

3.2 Non-ordering attribute of GGF(1)

Besides sequential access to the attribute on which the GGF(1) is ordered, often a need exists to have other access paths as well. Normally these paths are offered by so-called dense, non-clustered indexes. The GGF structure can be used to implement such an index. Consider again Fig. 2.1 and assume that the data pages contain two-tuples <CWIDTH,REFCOMP> where REFCOMP is a reference to a stored COMPONENTS tuple. So the data pages of Fig. 2.1 contain binary tuples ordered on CWIDTH instead of COMPONENTS tuples ordered on CNO. When the scale and the pointer list in a directory page are *combined* to form *one* list of two-tuples <CWIDTH-value,pointer> then we see that the GGF(1) is in fact a B^+-tree [9]. Hence, Fig. 2.1 shows then a non-clustered index on attribute CWIDTH for relation COMPONENTS.

3.3 One attribute of a GGF(2)

Suppose that the COMPONENTS tuples are stored in a GGF(2) clustered on CLENGTH and CWIDTH. Consider a query that asks for COMPONENTS tuples ordered by CLENGTH. As the GGF(2) offers clustering of the tuples on CLENGTH and CWIDTH, processing might be fast. First tuples with a CLENGTH value smaller than the first scale value are read, followed by sorting these tuples (hopefully and probably in internal memory). Then tuples from the next scale interval are read, and so on. The whole sort operation can be seen as a sequence of partial range queries that are followed by (small)

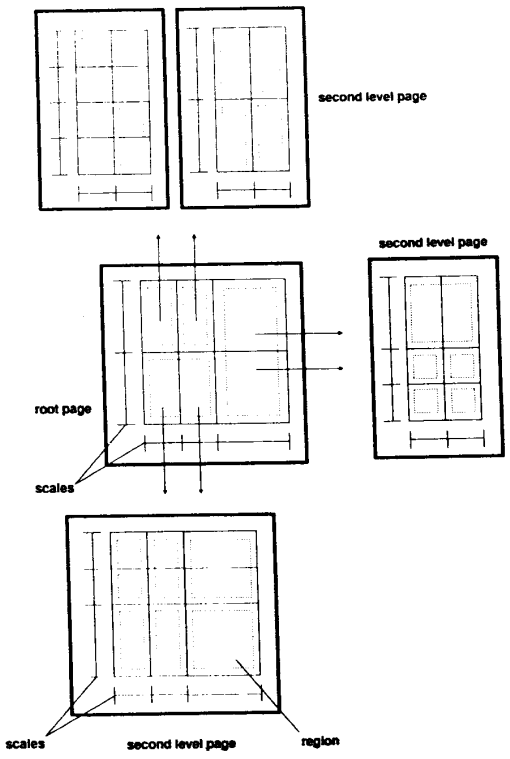

Fig. 2.4 GGF(2): Two level directory (schematic).

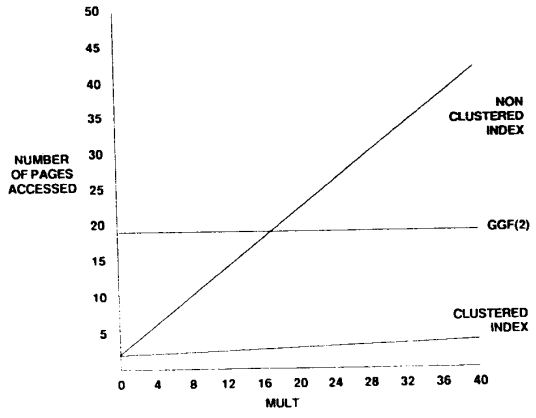

Fig. 4.1 Partial match query (NTUP = 10000).

sort operations. Each partial range query covers exactly one scale interval in the CLENGTH direction.

The strong point of the GGF(2) structure is that the mentioned processing can be done for the CLENGTH as well as the CWIDTH attribute.

4. INDEX SELECTION PROBLEM REVISITED

The index selection problem deals with the problem of determining the attributes (or attribute combinations) for which an index has to be defined, given a certain load (queries and updates) and given certain file characteristics. Many tools and algorithms have been developed to solve this problem [1, 8, 2, 6, 11]. In this section it will be shown that with the introduction of the GGF structure extra opportunities are offered to solve the index selection problem. To this end the following problem will be discussed.

Assume that the attributes CLENGTH and CWIDTH are often involved in the selection condition of queries. Such a query can be a point, partial match, partial range or range query. In order to speed up the processing of those queries the COMPONENTS relation can be provided with two indexes, one for the CLENGTH attribute and one for CWIDTH. (These indexes can be implemented by GGF(1)s, see previous section.)

Speeding up retrieval, however, can also be achieved by storing the tuples in a GGF(2) (or GGF(3), or GGF(4), ...) structure. An interesting question arises now: under which conditions will a GGF(2) structure be superior to two GGF(1) structures? This is the topic of this section. The criteria for the comparison are the number of page accesses needed to retrieve and insert tuples taking also the needed external memory space into account.

After introducing some assumptions and abbreviations, the results of the comparison are given in the Sections 4.2 - 4.7.

4.1 Assumptions and Abbreviations

Assume, without loss of generality, that the index for CLENGTH is clustered and for CWIDTH is not. Other assumptions are:

1. The values attained by an attribute are uniformly distributed over the corresponding domain. Moreover, attributes are assumed to be independent.
2. The file is steadily growing in size, so deletions will not cause merge operations to be executed.
3. The root page of a directory (or an index) is stored in internal memory.
4. A directory of a GGF(2) contains as many scales in CLENGTH as in CWIDTH direction.

Below some parameters and their abbreviations are introduced. Between parentheses the default values are given. The default values for the region size, which is defined as the average number of pointers in the directory referring to the same data page, differ significantly for GGF(1) and GGF(2). The reason is that in GGF(1) each data page is referred to only once, while in GGF(2) more than one pointer in the pointer-array may refer to the same page (see also Fig. 2.3). Based on own simulations and on results given in [19] the region size REGSZ(2) has been set to 2.5. OCCDAT gives the occupation factor for the data pages. Under the assumption of uniformly distributed attribute values an occupation factor of 70% is reasonable for GGF(1) and GGF(2). The appendix contains

formulas that are derived to estimate the number of page accesses for handling all kinds of queries.

NTUP	= number of tuples in file (10,000)
TUPLEN	= tuple length in bytes (100)
PAGLEN	= page length in bytes (4000)
PTRSZ	= size of a pointer to a page in bytes (4)
ATTSZ(n)	= size of attribute value in bytes in dimension n (10)
NSEL(n)	= number of scale elements of the axis in dimension n
OCCDIR	= occupation factor for directory (or index) pages (0.7)
OCCDAT	= occupation factor for data pages (0.7)
REGSZ(1)	= region size (the average number of GGF(1)-directory pointers referring to the same data page) (1)
REGSZ(2)	= idem for GGF(2) (2.5)
MULT	= average number of tuples having the same value for a certain attribute (for instance CLENGTH).

4.2 Point Query

In this section we assume that CLENGTH and CWIDTH constitute a composed key. If COMPONENTS tuples are stored in a GGF(2), values for CLENGTH and CWIDTH allow us to fetch a tuple. Depending on the depth of the directory this costs one or two page accesses (the root page is in internal memory!). In the two-indexes case, both indexes have to be accessed. If for instance NTUP = 100,000, then the non-clustered index is three levels deep and the clustered index only two. As the root pages are in internal memory the total number of page accesses is four. It is assumed that in the two-indexes case the intersection of the pointer lists obtained from the indexes can be performed in internal memory (so does not require extra page accesses). Table 4.1 gives the results.

NTUP	GGF(2)	Two-indexes
0 - 285	1	1
285 - 7,980	1	2
7,980 - 11,200	1	3
11,200 - 57,000	2	3
57,000 - 1,596,000	2	4
1,596,000 - 3,136,000	2	5

Table 4.1. Number of page accesses for point queries.

4.3 Partial Match Query

Consider the 2-dimensional space defined by the domains of the CWIDTH and CLENGTH attributes. A partial match query contains one equality condition, so two cases can be distinguished. Consider the case in which 'CLENGTH = v' is the selection condition. The attribute value v belongs to a certain scale interval on the CLENGTH axis, which determines a horizontal 'band' in the 2-dimensional space. This band contains the required tuples, hence in the GGF(2) the corresponding parts of the directory and data file have to be accessed. In the appendix formulas have been derived.

Let us now consider the two-indexes case. Tuples with equal CLENGTH value are stored near each other, so the partial match query with the condition 'CLENGTH = v' can be processed very fast. Processing of the query with the condition 'CWIDTH = w' uses the non-clustered index on CWIDTH. This results in a list of pointers that refer to tuples satisfying the condition. These tuples have to be fetched.

In Fig. 4.1. the multiplicity MULT is the average number of tuples having the same value for a certain attribute. This

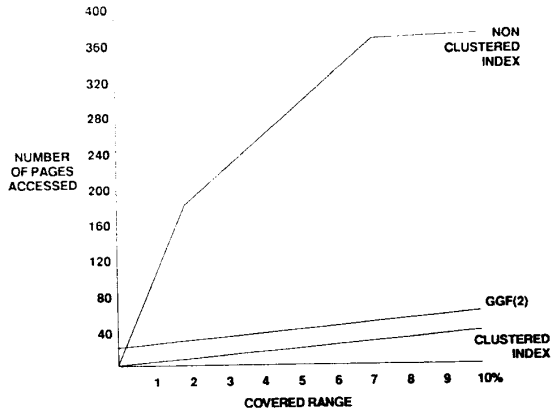

Fig. 4.2 Partial range query (NTUP = 10000).

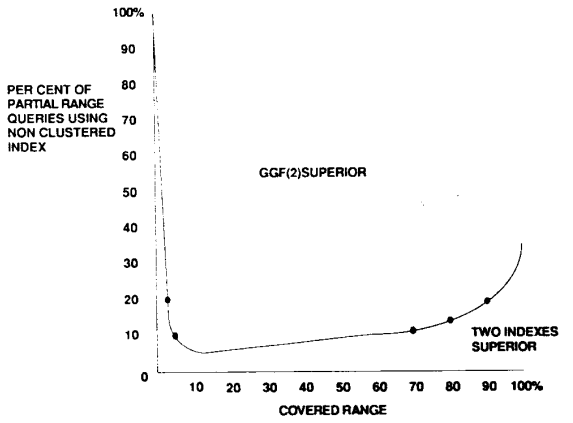

Fig. 4.3 Break-even points GGF(2) versus Two-indexes

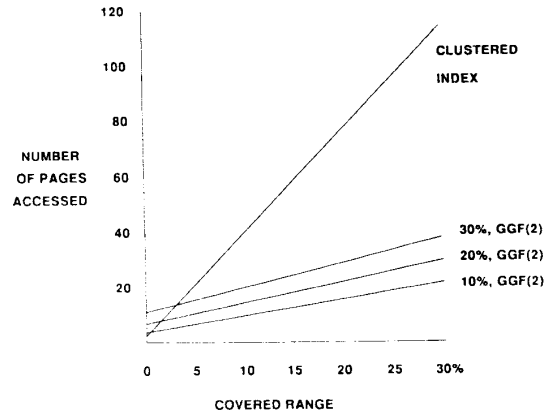

Fig. 4.4 Range query (NTUP = 10000).

attribute is CLENGTH for the CLUSTERED INDEX and CWIDTH for the NON CLUSTERED INDEX curve. The GGF is symmetrical, so it gives equal results for CLENGTH and CWIDTH. Fig. 4.1 shows that the GGF(2) often performs worse than the two indexes structure. Only when MULT > 17 and the partial match query concerns the CWIDTH attribute, the GGF(2) is superior.

4.4 Partial Range Query

Partial range queries in a GGF(2) either have the selection condition "v1 <= CLENGTH <= v2" or the condition "w1 <= CWIDTH <= w2". About the same processing as in the case of a partial match query can be followed. In Fig. 4.2 the results for a condition that is satisfied by 0 to 10% of the tuples are given. If the condition concerns the CLENGTH attribute, then the clustered index can be used, which gives a very good performance (see CLUSTERED INDEX graph). Moreover, the figures show that for very low values of the covered range even use of the non-clustered index (CWIDTH condition) gives better results than use of GGF(2). Fig. 4.3 gives a curve that shows the break-even points for the GGF(2) and the two-indexes cases. For instance, if the covered range is 10% and if 6% of the partial range queries concern the CWIDTH attribute, then the GGF(2) and the two-indexes perform about equally. If more than 6% of the queries concern the CWIDTH attribute then GGF(2) performs better.

4.5 Range Query

In a range query the selection condition is "v1<= CLENGTH <=v2 AND w1<= CWIDTH <=w2". In Fig. 4.4 the X-axis gives the percentage of the tuples satisfying the condition with "v1 <= CLENGTH <= v2" (from 0% to 30%). Although two indexes are available it is assumed that only the clustered one is used as for the considered number of tuples per page and the percentage of tuples satisfying the CWIDTH condition (10% to 30%), one may expect that at least one tuple per page satisfies the CWIDTH condition. Hence the non-clustered index does not help to decrease the number of accesses to data pages. It is clear that the GGF(2) structure, that exploits the CLENGTH as well as the CWIDTH condition, performs much better.

4.6 Insertion

In order to insert a tuple in a GGF(2) structure only the data page has to be updated giving rise to two or three accesses: one for the second level of the directory (if present) and two for reading and writing of the data page. In the two-indexes case the value of the CLENGTH attribute determines the physical position of the tuple. So, first the second level page of the (non-dense) clustered index has to be fetched, followed by reading and writing of the data page on which the tuple has to be stored. Moreover, the (dense) non-clustered index has to be updated, which amounts to a total of five page accesses (or six if the non-clustered index is three levels deep).

What about splitting of pages? We have assumed that the file is steadily growing, see assumption 4.1.1. Then the whole GGF(2) directory and the whole data file has been built up during the insertion of at least NTUP tuples (deletions are possible, but may not cause merge operations.) A simple computation shows that the influence of splitting on the GGF(2) performance is small, almost negligible. With two-indexes, however, the following problem occurs. If a data page has to be split, half of the tuples is normally moved to a new page. For these tuples the non-clustered index has to be updated. So it is clear that in the two-indexes case, splitting has a significant effect on performance (see Appendix B for formulas). In Table 4.2 splitting of data pages is taken into account.

NTUP	GGF(2)	Two-indexes
10,000	2.04	6.5
100,000	3.07	8.2

Table 4.2. Number of page accesses to insert a tuple.

4.7 Memory Occupation

As stated before the occupation factor has been set to 70%. The GGF(2) uses a directory, while in the two-indexes case we have a clustered and a non-clustered index. The number of pages occupied by an index depends on the size of the indexed attribute. Table 4.3 gives the results for varying attribute sizes. In the two-indexes case more space is needed (about 9-30%).

CLENGTH (in bytes)	CWIDTH	GGF(2)	Two-indexes
4	4	360	389
4	10	360	410
4	30	360	482
10	4	360	389
10	10	360	410
10	30	361	482
30	4	360	392
30	10	361	413
30	30	361	485

Table 4.3. External Memory occupation in pages. (NTUP = 10,000).

5. OPTIMIZATION

5.1 Indirection pointers

A region has been defined as a part of the data space that is mapped to one data page. As stated before, the region size is about 2.5 in the 2-dimensional case and assuming a uniform distribution; for dimensions greater than two and for non-uniform distributions, region sizes are even (much) greater.

Leaf directory pages contain pointers to data pages. In section 4 the pointer size PTRSZ has been set to four bytes, but in general it makes sense to store more information about a data page than just a pointer, for instance the number of tuples stored in a data page, locking information, etc. A reasonable estimate for the size of such a 'pointer' might be eight bytes. Given *big* region sizes, it is a good idea to store a 'pointer' to a page only *once* on a directory page, while so-called 'indirection pointers' of the pointer-array refer to these (long) 'pointers'; these indirection pointers are, considering realistic page sizes, etc. only one byte long (see Fig. 5.1). Observe that for GGF(1) the region size is always equal to 1, so that indirection pointers do not make sense in this case.

5.2 K-d-B-trees

We consider the k-d-B-tree. The most *straightforward* implementation of the description of a region consists of the

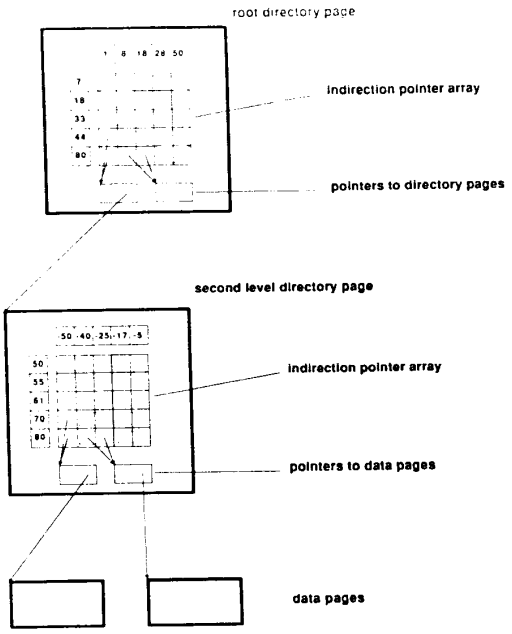

Fig. 5.1 The optimized Generalized Grid File GGF(2).

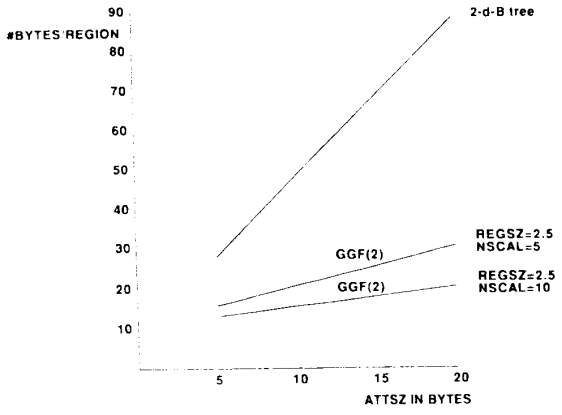

Fig. 5.2 Memory overhead per region for 2-d-B tree and GGF(2) for uniform data distribution and PTRSZ=8 bytes.

lower and the upper bound for each dimension of the region, resulting in the storage of (2*k) attribute values. Together with PTRSZ bytes for the pointer size this amounts to a memory overhead per region of:

(2 * k) * ATTSZ + PTRSZ bytes.

A directory page of the optimized GGF contains for each region a description and a pointer to the data page corresponding to the region. Assuming NSCAL scales in each of the k dimensions, we get a memory overhead per region of

k * NSCAL * ATTSZ / (NSCAL**k / REGSZ(k))
+ REGSZ(k) * 1 + PTRSZ bytes.

Fig. 5.2 compares the memory overhead for one region of a 2-d-B-tree in case of the straightforward region implementation and in case of the optimized GGF(2) implementation. The values for NSCAL are 5,10, which are (very) conservative values.

6. CONCLUSIONS

The grid file [19] offers multi-attribute access to data. This structure has two drawbacks. First, it does not support single-attribute access and secondly, very big files can not be handled as the internal memory occupation becomes too high. A generalization of the grid file, called the generalized grid file (GGF) eliminating those drawbacks has been described. GGF(i) (i>=1) is the notation for a generalized grid file that supports access to i attributes. The GGF(1) implements a file sorted on a certain attribute and provides a non-dense index for that attribute. The GGF(1) can also be used to implement a non-clustered, dense index for an arbitrary attribute: this use boils down to an implementation of the B^+-tree. Moreover, it appears that GGF(k) (k>=2) is an efficient implementation of the k-d-B-tree, so GGF integrates the B^+-tree, the grid file and the k-d-B-tree.

The problem of index selection in the business environment receives a new dimension because also GGF(2) (and GGF(3), ...) structures can be used. It appears that under certain circumstances the GGF(2) offers a significantly better performance than a file provided with two separate indexes, one of which is clustered. For instance, in a GGF(2) structure point queries, (partial) range queries and especially updates are handled (much) faster than in the two-indexes case.

The GGF structure can be used successfully in the business environment. However, it is also noticed that the GGF(2), GGF(3) and so on, are useful in supporting geometrical applications. Hence it can be concluded that GGF seems to be a good 'overall' access structure for the CIM environment where both kinds of applications meet.

REFERENCES

[1] Albano, A., De Antonellis, V. and Di Leva, A., Eds.,: Computer-aided Database Design. North-Holland Publ. Comp., Amsterdam, 1985.

[2] Anderson, H.D., Berra, P.B.: Minimum Cost Selection of Secondary Indexes for Formatted Files. ACM Transactions on Database Systems, Vol. 2, 1977, pp 68-90.

[3] Astrahan, et al.: SYSTEM R: Relational Approach to Database Management. ACM Transactions on Database Systems, Vol. 1, 1976, pp 189-222.

[4] Bernstein, P.A. et al.: Query Processing in a system for distributed databases (SDD-1). ACM Transactions on Database Systems, Vol. 6, No. 4, 602-625, 1981.

[5] Batory, D.S., Kim, W.: Support for Versions of VLSI CAD Objects. M.C.C. Working Paper, March 1985.

[6] Blanken, H.M.: Automatic Generation of Storage Structures for a DBTG Database System. International Conference on Databases, Aberdeen, July 1980, pp 99-118.

[7] Blanken, H.M.: Performance Aspects of Database Management Systems. PhD Thesis, TU Twente, 1984.

[8] Ceri, S.: Methodology and Tools for Database Design. North-Holland Publ. Comp., Amsterdam, 1983.

[9] Comer, D.: The Ubiquitous B-tree. Computing Surveys, Vol. 11, No. 2, June 1979, pp 121-137.

[10] Fagin, R. et al: Extendible Hashing: a fast access method for Dynamic Files. ACM Transactions on Database Systems, Vol. 4, No. 3, Sept 1979.

[11] Gambino, T.J., Gerritsen, R.: A Database Design Decision Support System. Proc. of the Conf. on Very Large Databases, 1977, pp 534-544.

[12] Gruendig, L., Pistor, P.: Land-Informations-Systeme und Ihre Anforderungen an Datenbank-Schnittstellen. J.W. Schmidt, Ed., Informatik-Fachberichte 72, Springer, Berlin, 1983, pp 61-75 (in German).

[13] Hinrichs, K.: Implementation of the Grid File: Design Concepts and Experience. BIT 25, 1985, pp 569-592.

[14] Katz, R.: Information Management for Engineering Design. ISBN 3-540-15130-3, Springer Verlag, 1985.

[15] Krause, F.L., Armbrust,P., Bienert,M.: Methodbases and Product Models as a Basis for Integrated Design and Manufacturing. Proc. of the 2nd International Conference on Manufacturing Science, Technology and Systems of the Future. Ljbjana, Yugoslavia, Sept. 1985.

[16] King, W.: Relational database systems: Where we stand today", Information Processing 80, S.H. Lavington (ed), IFIP, 1980, pp 370-381.

[17] Kriegel, H.-P.: Performance Comparison of Index Structures for Multi-key Retrieval. Proc. ACM SIGMOD '84, pp. 186-196.

[18] Lorie, R.A., Plouffe, W.: Complex Objects and their Use in Design Transactions. Proc. Annual Meeting - Database Week: Engineering Design Applications (IEEE), San José, May 1983, pp 115-121.

[19] Nievergelt, J., Hinterberger, H., Sevcik, K.D.: The Grid File: an Adaptable, Symmetric Multi-Key File structure. ACM Transactions on Database Systems, Vol. 9, No. 1, 1984, pp. 38-71.

[20] Ouksel, M.: The Interpolation-Based Grid File. Proc. of ACM-SIGMOD-SIGACT Symposium on Database Systems, 1985.

[21] Six, H-W. and Widmayer, P.: Spatial Searching in Geometric Databases. Sixth Data Engineering Conference, Feb. 1988, Los Angeles, Cal. USA.

[22] Robinson, J.T.: The K-D-B-tree: a Search Structure for large multidimensional dynamic indexes. In Proc. ACM SIGMOD'81, pp. 10-18, 1981.

Appendix A: GGF Algorithms

Below it is assumed that the GGF occupies at least one tuple. In the algorithms, cursive names indicate variables.

```
range_query (page: page_type; range: range_type);
    (* treats point, partial match, partial range and range queries;
       condition is given by range *)

    IF page is a data page
    THEN fetch tuples in page within range
    ELSE        (* page is directory page *)
        FOREACH region in page DO
            IF region ∩ range ≠ ∅
            THEN    determine page_l covering region;
                    range_query (page_l, range)
        END;


insertion (VAR root: page_type; tuple: tuple_type);
    (* inserts tuple in GGF indicated by root *)

    find data_page on which tuple must be inserted;
    IF data_page is full
    THEN        split_root (root, tuple);
                insertion (root, tuple)
    ELSE        insert tuple in data_page;


split_root (VAR root: page_type; tuple: tuple_type);
    (* expands GGF to store new tuple; the depth of directory
       is possibly incremented by 1 by creating a new root page *)

    split (root, tuple);
    IF root too full
    THEN        split root into two pages;
                create and update new root page;
                root becomes new root page;


split (VAR page: page_type; tuple: tuple_type);
    (* makes space for tuple in subtree indicated by page;
       updates page and directory pages below page *)

    determine region in page with key of tuple ∈ region;
    determine page_l covering region;
    IF page_l is directory page
    THEN        split (page_l, tuple);
                IF page_l is too full
                THEN    split page_l into two pages;
                        update region information in page
    ELSE        split data page page_l into two pages;
                update region information in page;


delete_tuple_root (VAR root: page_type; key: key_type);
    (* deletes tuple(s): many tuples may satisfy 'key value = key';
       tries to merge lower level pages and
       shrinks possibly the directory by one level *)

    FOREACH region in root with key ∈ region DO
                determine page_l covering region;
                delete_tuple (page_l, key);
    END;
    try to merge regions immediately below root;
    IF only one region remains in root AND depth of directory >= 2
    THEN        determine page_l covering region in root;
                delete page root;
                root becomes page_l;


delete_tuple (page: page_type; key: key_type);
    (* deletes tuple(s) and possibly merges pages
                one level below page *)

    IF page is directory page
    THEN        FOREACH region in page with key ∈ region DO
                        determine page_l covering region;
                        delete_tuple (page_l, key);
                END;
                try to merge pages below page and update page;
    ELSE        remove from page tuples with 'key value = key';
```

Appendix B: Formulas

NDATP = (* number of pages occupied by data file *)
NTUP / (PAGLEN * OCCDAT / TUPLEN)

NRACC(n,m) = (* number of page accesses needed to fetch m tuples randomly stored over n pages; the pointers to these tuples are given and ordered on page number; the formula derived by Bernstein [4] is used *)

m	if m <= n/2
(m+n)/3	if n/2 < m <= 2*n
n	if 2*n < m

NLFPCI = (* number of leaf pages occupied by clustered index *)
NDATP * REGSZ(1) * (ATTSZ(1) + PTRSZ) / (OCCDIR * PAGLEN)

NLFPNCI = (* number of leaf pages occupied by non-clustered index *)
NTUP * REGSZ(1) * (ATTSZ(1) + PTRSZ) / (OCCDIR * PAGLEN)

NLFPDIR = (* number of pages in the lowest level of directory of GGF(2); NSEL(1) and NSEL(2) are determined by the following relation: NSEL(1) * ATTSZ(1) + NSEL(2) * ATTSZ(2) + NSEL(1) * NSEL(2) * PTRSZ = OCCDIR * PAGLEN *)
NDATP / (NSEL(1) * NSEL(2) / REGSZ(2))

NEPTI = (* number of external pages two-indexes case (clustered and non-clustered index) *)
NDATP + NLFPCI + NLFPNCI

NEPGGF = (* number of external pages GGF case *)
NDATP + NLFPDIR

NLVCI = (* number of levels of clustered index *)

NLVNCI = (* idem for non-clustered index *)

NLVDIR = (* idem for directory of GGF(2) structure *)

Formulas for the latter three terms are not given as these formulas are well-known. For the considered cases holds that NLVCI and NLVDIR are equal to 2, while NLVNCI is 2 or 3 (depending on NTUP).

NACC(1) is the number of page accesses needed to process a query if the file has two indexes; the same holds for NACC(2) with respect to the GGF(2) structure. Remind that the root page is always stored in internal memory!

Partial Match query

For the GGF(2) holds that the number of scale values in both directions are equal (see assumption 4 in section 4.1). Given also the uniform distribution, an estimate of the expected number of pages, which must be accessed, is:

NACC(2) = sqrt(NLFPDIR) + sqrt(NDATP)

When two indexes are offered the following cases can be distinguished:

- If the partial match condition concerns the attribute for which a clustered index is present, then a good approximation is given by:

  NACC(1) = (* read clustered index + read some data pages *)
  (NLVCI - 1) + (1 + MULT * NDATP / NTUP)

- For the non-clustered index we get:

  NACC(1) = (* read part of non-clustered index + MULT tuples *)
  ((NLVNCI - 1) + MULT * NLFPNCI / NTUP) + NRACC(NDATP,MULT)

Partial Range Query

First we consider the GGF(2) case. Assume that in the partial match query, a % of the total interval is involved. The expected number of pages, which must be accessed is:

NACC(2) = (* read pages of directory + read pages of data file *)
(1 + a * sqrt(NLFPDIR) / 100) * sqrt(NLFPDIR) +
(1 + a * sqrt(NDATP) / 100) * sqrt(NDATP)

Now the two-indexes case can be considered.

- If the condition involves the attribute for which a clustered index is present, the expected number of pages, which must be accessed, is:

  NACC(1) = (* read pages of clustered index + read pages of data file *)
  (1 + a * NLFPCI / 100) + (1 + a * NDATP / 100)

- For the attribute with the non-clustered index we get:

  NACC(1) = (* read part of non-clustered index + some random accesses *)
  (NLVNCI - 1) + a * NLFPNCI / 100) + NRACC(NDATP,a*NTUP/100).

Range Query

Consider the GGF(2) case. Assume that in one direction a % of the total interval is involved and in the other b %. We get the formula:

NACC(2) = (1 + a * sqrt(NLFPDIR) / 100) *
(1 + b * sqrt(NLFPDIR) / 100) +
(1 + a * sqrt(NDATP) / 100) *
(1 + b * sqrt(NDATP) / 100)

In the two-indexes case only the clustered index is used; for the formulas, see the partial range query.

Insertion

The influence of splitting data pages has been taken into account; splitting of directory or index pages, however, is neglected!

NACC(2) = (* read dir page(s) + read and write data page + (in case of a split) write new data page and updated directory page *)
(NLVDIR - 1) + 1 + 1 + (1 / (OCCDAT * PAGLEN / TUPLEN)) * (1 + 1)

NACC(1) = (* read clustered index page(s) + read and write data page + (in case of a split) write data page and updated leaf page of clustered index *)
(NLVCI - 1) + 1 + 1 + (1 / (OCCDAT * PAGLEN / TUPLEN)) * (1 + 1)
(* read non-clustered index page(s) and write leaf page *)
+ (NLVNCI - 1) + 1
(* in case of a split of a data page, update of non-clustered index for all moved tuples *)
+ (1 / (OCCDAT * PAGLEN / TUPLEN)) * ((NLVNCI - 1) + 1) * PAGLEN / (TUPLEN * 2)