Implementing a Generalized Access Path Structure for a Relational Database System

THEO HAERDER
Technische Hochschule Darmstadt, West Germany

A new kind of implementation technique for access paths connecting sets of tuples qualified by attribute values is described. It combines the advantages of pointer chain and multilevel index implementation techniques. Compared to these structures the generalized access path structure is at least competitive in performing retrieval and update operations, while a considerable storage space saving is gained. Some additional features of this structure support m-way joins and the evaluation of multirelation queries, and allow efficient checks of integrity assertions and simple reorganization schemes.

Key Words and Phrases: database, relational model, access path structures, index structures, B^*-trees
CR Categories: 3.50, 3.74, 4.33

1. INTRODUCTION

A relational database system accessible to nontechnical users provides an interface which allows queries to be expressed with a number of powerful relational operators using a simple conceptual framework. Since the complexities of the access path organization are hidden from the user, the interface itself has to translate queries into the actual sequence of storage and data references. Therefore, the system should be capable of optimizing access and selection of data. For this goal only a limited number of operators has to be supported at this level, e.g. join, selection, and projection. Nevertheless, it is crucial for the efficiency of the system which kinds of access aids are chosen to implement the set of relational operators.

As opposed to conventional file systems where access to tuples is only provided for a unique key (primary key) using key-to-address transformation (e.g. hashing schemes) or key comparison techniques, e.g. binary, indexed, or sequential search, there exists the additional requirement for fast associative and sequential access

This work was performed while the author was visiting at IBM Research Laboratory, San Jose, CA 95193.

Author's address: Technische Hochschule Darmstadt, Fachbereich 20 (Informatik), Hochschulstr. 1, D-6100 Darmstadt, West Germany.

to sets of tuples and also for fast navigation from one tuple to others which are related in some way. Usually, these design goals are reflected by introducing two different kinds of access path implementations, e.g. secondary indexes (inverted lists) and pointer chains. Secondary indexes support sequential and fast associative access to single tuples and to sets of tuples qualified by their content, while pointer chains are usually used for navigational access. Several alternatives to pointer chains for navigation on access paths connecting sets of tuples are described in [2].

At any rate, the provision of two different kinds of access path implementations together with the appropriate operators on them increases the complexity of system implementation. Therefore, it seems desirable to find a single implementation technique which supports both types of applications and which can compete with indexes and pointer chains from a performance point of view.

In this paper we propose such an implementation technique, called a "generalized access path structure," which combines the advantages of index and pointer chain access. We discuss this structure as an access aid for the relational model of data [9], that is, as a totally redundant access path bearing only "inessential" information [10]. It will be shown that this structure can also be used for implementing access paths providing essential information for the logical data structures. Some additional features of this structure support m-way joins and the evaluation of multirelation queries, and allow efficient checks of integrity assertions and simple reorganization schemes.

2. THE RELATIONAL MODEL: FORMALISM AND TERMINOLOGY

We use the well-known terminology of the relational model of data [7, 9] which can be viewed as a schema with a number of interpretation rules.

Let a database schema be a finite collection of relation schemata and a set of domains. Each relation schema consists of a relation name RN, a finite set of attribute names $(A1, A2, \ldots, An)$, which are all unique within a distinct relational schema, and a functional mapping F_{RN} of this set of attribute names into the set of domains. If an attribute name A maps into a domain $F(A)$, then all values for that attribute must belong to that domain. More than one attribute name may map into the same domain.

An n-ary mathematical relation over the sets $D1, D2, \ldots, Dn$ is a subset of the Cartesian product $D1 \times D2 \times \cdots \times Dn$. Let u be an element of such a relation; then u is called an n-tuple or tuple (for short) $u = \langle u1, u2, \ldots, un \rangle$ with $ui \in Di$.

An instance of a relation schema $RN(A1, A2, \ldots, An)$, or a relation R for short, is a finite subset of $F(A1) \times F(A2) \times \cdots \times F(An)$, where F is the domain mapping of the schema.

Generally, the elements of a domain are homogeneous, that is, each domain has a particular value associated with it, e.g. binary, numeric, character.

An attribute or minimal group of attributes which guarantees the uniqueness of tuples within a relation R is called a candidate key. One of the candidate keys of a relation is chosen to be the primary key. A foreign key is an attribute (or group of attributes) of a relation that has to be defined on the same domain as a candidate key of another relation.

An important observation leading to our proposal of an implementation technique of a generalized access path structure is the following: The domains of the relational names specified as part of the database schema definition carry important interrelational information. One of the important issues of these domains is to indicate comparability of attributes, within the same relation schema or across schemata, which is needed to achieve various relational operations, e.g. join.

Since the relationship between different relations Ri is based on the matching of domain values, this fact can be used in constructing an access path to the related tuples of different relations.

3. ACCESS PATHS FOR RELATIONS

The relational data model differs from other models in that all information is stored in terms of data values (domain values) within tuples, i.e. no essential information is represented by connections between tuples or by ordering of tuples. In order to increase the performance of the system in case of associative access to the tuples or required value ordering of the tuples, specific access aids (access paths) are introduced additionally.

An access path giving value ordering and associative access by one or more attributes to one relation is called an "image" following the terminology introduced in [1].

Definition. Let R be a relation with attributes $A1, \ldots, An$. An image of the attribute Ai of R, $i \in \{1, \ldots, n\}$, is a mapping from values in Ai to those tuples in R which have that value for the ith attribute, i.e. a mapping $I_i: F(Ai) \rightarrow 2^R$. Additionally, these sets of tuples qualified by values of Ai are ordered according to the sorted sequence of values of Ai. The generalization of the term "image" to compound attributes is straightforward.

Access paths relating tuples of one relation to tuples of another relation are called binary links. In the paper we shall use special binary links according to the following definition.

Definition. Let R be a relation with attributes $A1, \ldots, An$, S be a relation with attributes $B1, \ldots, Bm$; $F(Ai) = F(Bk)$ for the domains $F(Ai)$, $F(Bk)$, $i \in \{1, \ldots, n\}$, $k \in \{1, \ldots, m\}$; Ai be a candidate key of R. The link between R and S with regard to Ai, Bk is defined as the set $L(R(Ai), S(Bk)) := \{(r, s) \mid r \in R, s \in S, \mathrm{pr}_{Ai}(r) = \mathrm{pr}_{Bk}(s)\}$, where $\mathrm{pr}_{Ai}(r)$ and $\mathrm{pr}_{Bk}(s)$ are the projections to the components of r and s which correspond to attributes Ai and Bk, respectively. The term "link" may be generalized for compound attributes similarly.

The reference from the access path structure to the actual tuple is usually done by means of *TID*'s (Tuple Identifiers [1]) or physical pointers. An appropriate implementation technique for *TID*'s being a concatenation of a page number along with a byte offset from the bottom of that page combines the speed of a byte address pointer with the flexibility of indirection. The page number allocated in a logical address space allows an indirect reference to the actual physical storage block. The offset denotes a special slot which contains the byte location of the referenced tuple in the page. Hence, the *TID* concept offers two different kinds of indirection—at the page level and within the page.

All links and images should be "automatic" [8] to preserve the properties of the relational model, that is, the placement of tuples in images and links is only based on matching values and not on the decision of the user ("manual" [8]).

3.1 Implementation Technique for Images

An image is conveniently implemented and maintained through the use of a multipage index structure which contains pointers to the tuples themselves. The pages of a given index can be organized into a balanced hierarchic structure using the concept of B^*-trees [3, 13]. For nonleaf nodes, an entry consists of a key value and a pointer pair. The key itself can consist of values of single or compound attributes and can be represented in encoded form [5] allowing a particular sort order on each attribute value in case of compound attributes. The pointer addresses another nonleaf page or a leaf page in the same structure.

For the leaf nodes an entry is a combination of key values, along with a variable length ascending list of *TID*'s for tuples having exactly those key values. In order to identify the length of the *TID* list an additional length information field is kept with each stored key. In addition, the leaf pages are chained in a doubly linked list, so that sequential access can be supported from leaf to leaf.

If the total storage space for the *TID* lists of a particular key exceeds one leaf page, overflow pages can be introduced optionally which can keep the overflowing part of the lists. These overflow pages are chained with the leaf pages only, and they are not pointed to by the nonleaf pages to reduce the increase of the height of the B^*-tree.

If a mechanism is provided for enforcing the uniqueness of keys, e.g. for specified candidate keys, this structure can also be used to implement an access path for primary keys. The "image" of the relation is represented by the particular value ordering when accessing the leaves of the B^*-tree from left to right (in post order). When a relation is created, one image of the relation may be designated as the "clustering image," with the result that tuples near each other according to a chosen order relation will be stored physically near.

Figure 1 shows schematically an image on an attribute Ai of some relation R with $\{K1, K2, \ldots, K99\} \in F(Ai)$. Let us assume that the relation EMP(ENO, DNO, ...) with the attributes employee number (ENO), department number (DNO), etc., is given and inverted on DNO. The image on the attribute DNO is denoted by I(EMP(DNO)).

3.2 Implementation Technique for Links

A binary link connects tuples in one or two relations on matching attribute values. Usually, it is implemented by using chaining techniques with *TID*'s or physical pointers (storage addresses). The *TID* chaining gives one level of indirection compared to physical chaining of addresses.

For example, links are maintained in the Relational Storage System by storing the *TID*'s of the NEXT, PRIOR, and OWNER tuples in the prefix of the child tuples and by storing at least the *TID* of the first child tuple in the parent tuple according to Figure 2. In this example one tuple of the OWNER relation DEPT(DNO, ...) is linked to n tuples of the MEMBER relation EMP(ENO, DNO, ...). The binary link is denoted by L(DEPT(DNO), EMP(DNO)).

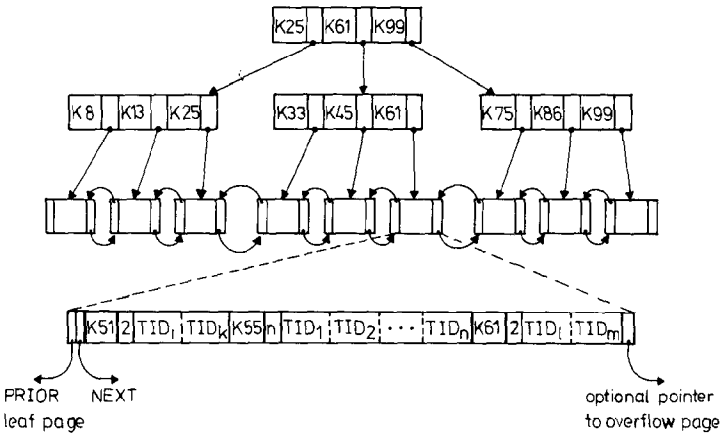


Fig. 1. Image implementation for I(EMP(DNO))

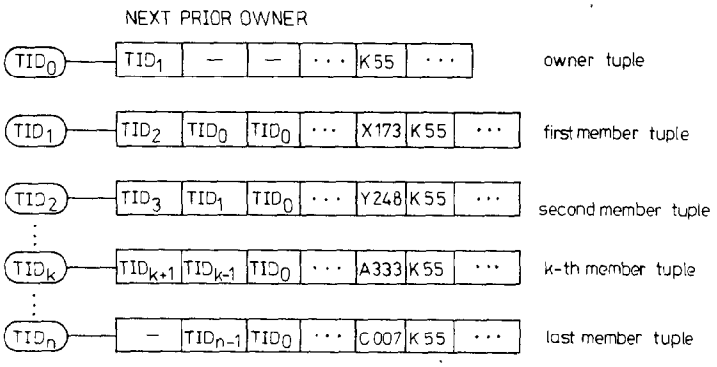


Fig. 2. Link implementation of L(DEPT(DNO),EMP(DNO)); link occurrence for domain value $K55$

3.3 Implementation Technique for a Combined Access Path Structure

A binary link provides a direct path from single tuples (parents) in one relation to sequences of tuples (children) in another relation. Usually it is argued that the main advantage of a link is the direct access to a tuple of either relation coupled by a binary link, while use of an image may involve a complete traversal of a B^*-tree structure consisting of several page accesses in order to find the child or parent tuple [1]. The relative gain of a link over an image is even enhanced when the child tuples have been clustered on the same page as the parent tuple. In this case no additional page has to be touched using the link, while a couple of pages may be accessed in a large index.

It should be pointed out that the relationships between tuples of one or different relations are expressed explicitly by attribute values in the relational model. This key property allows combined images on the same domain serving also as link structures. Therefore, the advantages of image and link access can be combined using a different kind of organization of the leaf nodes of the B^*-tree. The nonleaf nodes look exactly as in the single image implementation. In the leaf nodes separate *TID* lists for both relations together with the related length information fields are stored for each key. The lists for the parent relation contain

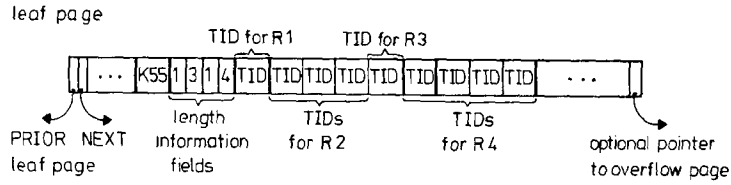


Fig. 3. Combined implementation of link L(DEPT(DNO),EMP(DNO)), and the images I(DEPT(DNO)) and I(EMP(DNO))

only one *TID* entry, while each variable length list for the child relation contains the sequence of *TID*'s for the children related to a particular parent tuple. The order in these lists can be exactly the same as in the binary link. In Figure 3 the discussed examples for the EMP and DEPT relations are treated in a unified way. The various attribute values for DNO in EMP and DEPT are the keys in the images and the matching DNO's also establish the link occurrences between the two relations.

With this access path structure the striking disadvantage of separate images can be avoided, that is, the traversal of an additional B^*-tree structure, when the child tuples are to be accessed after the parent tuple is located. In either case it must be assumed that the owner tuple is found via an image access I(DEPT(DNO)). If the leaf page containing the required key (candidate key) for the tuple of the OWNER relation is fixed in core, then the subsequent navigational accesses to the tuples of the MEMBER relation are at least as fast as the accesses via the binary link. In case of clustering, even more tuples can be stored in a particular page, because the storage space of three *TID*'s per tuple and link is saved. On the other hand, the access to the linked tuples in determined sequence enforced by the embedded *TID* chain is not necessary. Furthermore, having the combined access path structure, there is no need to fetch the tuples of a binary link sequentially, e.g. if it happens that the tuples are stored on different devices, seeks and rotational delays may be overlapped.

Therefore, the proposed access path structure can be considered in this particular case as a combination of

—an image for DEPT, I(DEPT(DNO)),
—an image for EMP, I(EMP(DNO)),
—a binary link for DEPT – EMP, L(DEPT(DNO), EMP(DNO)), with direct access from
  —OWNER to each MEMBER,
  —each MEMBER to each other MEMBER,
  —each MEMBER to OWNER.

In all cases DNO is used to search the B^*-tree so that only one B^*-tree structure is needed. Furthermore, the pointers NEXT, PRIOR, and OWNER are not stored. They are expressed implicitly by their relative position in the variable length *TID* list.

3.4 Generalization of the Combined Access Path Structure

3.4.1 Implementation Technique for the Generalized Access Path Structure. The combined access path structure replaces different access path types like image and binary link by joining the various characteristics of these access paths in one unified structure. A considerable advantage is gained, there-

fore, with regard to implementation complexity. Instead of supporting specialized modules for each of the access path types, only one unified set of modules working on this combined structure is necessary. The proposed approach reduces the extent of implementing various operations on access paths.

The proposed concept of the combined access path structure can be extended in the following way leading to the "generalized access path structure": All variable length *TID* lists belonging to the various attributes in different relations which are all defined on the same domain are stored with their related domain value (key value). This concept is not restricted to a single domain with single attributes defined on it. It can be applied to given sequences of attributes (compound attributes) corresponding to one particular domain sequence.

The format of the nonleaf pages is the same as for the image and combined access path. All kinds of optimizations, e.g. key compression, which are available for single access path implementation can be applied to them. The leaf pages contain for each key up to m variable length *TID* lists together with m length information fields. If an actual domain value is not defined for attribute Ai, then the corresponding *TID* list does not exist and the corresponding length information field indicates this fact by having a zero entry. At least one *TID* list must exist for a specified domain value; otherwise this domain value is currently not used in any tuple of the related relations and doesn't appear as a node in the access path.

The implementation of the generalized access path structure is shown in Figure 4. The particular example is chosen for four relations related by domain DEPARTMENT NUMBER. Let us assume that the relation $R1$ is DEPT with DNO being the primary key. $R2$ may be considered as the EMP relation with the inverted attribute DNO. $R3$ and $R4$ are introduced as the MANAGER and EQUIPMENT relations:

MGR(MNO, DNO, JCODE, ...)

EQUIP(INO, DNO, TYPE, ...)

The attributes DNO of the relations MGR and EQUIP are also inverted. DNO in relation MGR is specified as a candidate key, additionally. The graphical representation of this example describing all existing binary links in it is shown in Figure 5.

Here the same attribute name DNO is chosen for convenience. In principle each relation can have a different attribute name defined on the same domain, e.g. DEPARTMENT NUMBER. In the case of domains with numeric values each attribute can carry a different unit of the same or different unit types. By accessing the index the appropriate conversion rule must be applied to map the particular attribute value to the corresponding domain value.

Each node in the leaf page, e.g. the particular node with domain value $K55$ in Figure 4, contains up to four variable length lists with four length information

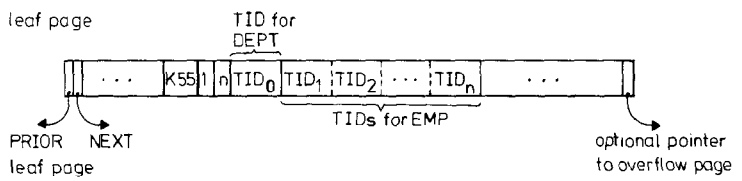


Fig. 4. Implementation of the generalized access path; example for four relations on domain DNO

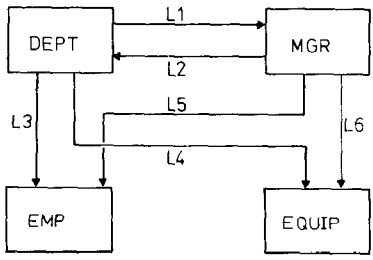


Fig. 5. All existing binary links based on DNO

fields describing the tuples of the four different relations with DNO = $K55$. If a particular attribute is specified as a candidate key, the corresponding list length of the *TID* list is restricted to 1, shown in the example for domain value $K55$ for $R1$ and $R3$. All other attributes are not restricted at all.

3.4.2 Determination of All Different Access Paths. In order to reduce the complexity of the following analysis, we discuss the case of single attributes defined on one domain. The generalization of that concept to compound attributes is straightforward. The concatenation of the attributes $A1, A2, \ldots. Ak$ in case of compound attributes can be viewed as a new attribute A' defined on a domain D.

Let us assume that m single attributes $A1, A2, \ldots, Am$ are defined on a domain D. For convenience, we denote the corresponding relations by $R1, R2, \ldots, Rm$. (Ri and Rj are the same if Ai and Aj belong to one relation.)

In order to evaluate the number of images and binary links which may be represented by a generalized access path structure, we assume that the first n attributes $A1, A2, \ldots, An$, $n \leq m$, are candidate keys of their relations. The attributes $An+1, \ldots, Am$ are dependent attributes.

In the general case OWNER and MEMBER of a binary link can be the same relation, e.g. if a hierarchical structure is defined on the tuples of a relation using two different attributes.

Note that there is always a functional dependency between the tuples of the MEMBER and the OWNER relation. If there is a one-to-one correspondence between two attributes a binary link can be defined in either direction: $L(Ri(Ai), Rj(Aj))$and $L(Rj(Aj), Ri(Ai))$. Clearly, an image can be established for each attribute:

$I(R1(A1)),$
$I(R2(A2)),$
$\vdots$
$I(Rm(Am)).$

Additionally, the access via the following different binary links is conceivable:

$L(R1(A1), R2(A2)),$
$L(R1(A1), R3(A3)),$
$\vdots$
$L(R1(A1), Rm(Am)),$
$L(R2(A2), R1(A1)),$
$L(R2(A2), R3(A3)),$
$\vdots$

$L(Rn(An), R1(A1)),$
$\vdots$
$L(Rn(An), Rn-1(An-1)),$
$L(Rn(An), Rn+1(An+1)),$
$\vdots$
$L(Rn(An), Rm(Am)).$

Therefore, the total number of images is m and the total number of binary links is $n \cdot (m-1)$, which the generalized access path structure may be used for.

Note. Since each *TID* list per attribute value of each attribute is stored only once, only one particular ordering can be represented in such a list. If a *TID* list is used for a MEMBER relation in different binary links only one particular ordering can be chosen.

4. EVALUATION OF THE GENERALIZED ACCESS PATH STRUCTURE

The presented proposal is based on the observation that the height of the B^*-tree describing the necessary page fetches to access a leaf page remains constant for a wide variety of different leaf page numbers, e.g. for a page size of 4 K bytes the height of a B^*-tree is constant for 2 up to 400–500 leaf pages ($h = 2$). In most cases the larger amount of access path data of the generalized access path structure does not increase the height of the B^*-tree. Therefore, access path data such as twin and parent pointers can be factored out from the tuples and stored more economically in variable length lists of the generalized access path structure without increasing the path length to a leaf node of the B^*-tree. Since these pointers are clustered in single pages, the various kinds of conceivable pointers, such as FIRST, NEXT, PRIOR, LAST, *N*th, OWNER, etc., can be represented implicitly by their relative position in the variable length *TID* list. As a result a substantial overall saving of storage space is gained with this structure.

In order to evaluate the generalized access path structure a large number of parameters of its actual implementation has to be considered. Additionally, a detailed performance study for all important operation types using or modifying this access structure is necessary. A sensitive parameter affecting the performance of accesses to sets of tuples is the fact whether the requested tuples are clustered or not; that is, whether they are stored together in a physical block (page) or not. Therefore, a particular emphasis has to be put on this case.

The cost factors determining the utility of an access aid are the following:

—storage space of the access path data,
—selecting single tuples based on their content,
—selecting single tuples based on their relative position in an access path (NEXT, PRIOR, FIRST, LAST, *N*th),
—selecting sets of tuples based on their content,
—accessing all tuples of a relation sequentially in value-determined order,
—performing a join operation on matching values
  —of single tuples in two relations,
  —of two entire relations,
—maintaining the access path for

—insertion,
—update,
—deletion of tuples.

A detailed study of the generalized access path structure considering all these cost factors in terms of actual page accesses is described in [12].

The results comparing the access costs for retrieval operations on clustered and unclustered tuples show that the generalized access path structure is at least competitive for all types of the considered access primitives compared to other access path types.

The sequential access costs for a relation in value-determined order are lower for the link structure as far as the number of pages touched is concerned. The reason for this behavior is the increased number of leaf pages which have to be fetched for the generalized access path structure. On the other hand, this quantity of access path pages is negligible with regard to the entire number of data pages transferred. Furthermore, some storage space is saved in the data pages due to elimination of chain pointers, so that more related tuples may be clustered in a page.

The comparison of the maintenance costs for the various access path structures show that the generalized access path structure is insensitive to the number of MEMBER tuples per attribute value, while the costs are increasing linearly with the chain length of a link occurrence in case of unclustered tuples.

5. ADDITIONAL FEATURES OF THE GENERALIZED ACCESS PATH STRUCTURE

5.1 Support of Relational Operations

The generalized access path structure supports in a natural way the join of relations, because the access information for tuples of different relations having matching values is stored close together. Although binary link structures implemented as embedded chains allow only one-to-many joins in a natural way, the generalized access path structure also supports many-to-many joins (natural joins [9]) easily. The algorithms for joining relations need not be restricted to two relations at a time, if more than two relations are to be joined on matching values. For the general case linking m different relations together on matching domain values an m-way join may be defined. A simple approach for this type of join would be to execute $(m - 1)$ subsequent 2-way joins. However, with the generalized access path structure, appropriate algorithms can easily be designed which perform an m-way join at a time.

Having the generalized access path structure, some query types can be answered without looking at the actual data. For example, there is no need to add an attribute NEMP (number of employees) to the DEPT relation, because the values of this attribute can be derived from a generalized access path without additional costs. The number of employees belonging to a given department may be counted using the access path on domain DEPARTMENT NUMBER (DNO) for DEPT and EMP.

Let us assume that one wants to list all departments with DNO > 50 which have less than 5 employees. The corresponding SEQUEL query [1] referring to DEPT and EMP relations is stated as follows:

```
SELECT DNO
FROM DEPT X
WHERE DNO > 50
AND
  (SELECT COUNT (*)
  FROM EMP
  WHERE DNO = X.DNO) < 5;
```

To evaluate this query no tuple has to be accessed because all information can be extracted from the access path data.

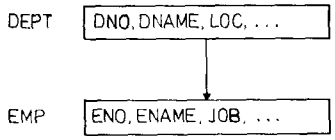
A useful feature in a database system is the availability of mechanisms to check existing integrity assertions [11] which describe properties of data objects such as type, value range, unit, update rules, etc., and also their relationships to other data objects. Such time-independent assertions about the relations in a database are formulated to preserve the integrity of data. The system has to prevent update operations which would violate an assertion or to trigger further maintenance operations to preserve the integrity of the database.

Those actions should be achieved without degrading the performance of the system. Hence, performance oriented aids such as access paths are required for efficient checks of assertions in case of insertion, deletion, and update transactions.

The generalized access path structure offers some powerful means to support this goal. For example, the uniqueness of a defined primary key can be guaranteed easily through the index mechanism. During the creation of such an access path, checks for duplicates may be included without additional costs. Owing to the centralized and clustered access path data, checks on the membership and ownership of sets (functional dependencies) can be obtained without further page accesses. For example, an OWNER tuple with existing MEMBER tuples can be prevented from deletion, or alternatively can trigger propagated deletion of all MEMBER tuples, by looking only at the particular node in the access path.

5.2 Implementation of Access Paths Bearing Essential Information

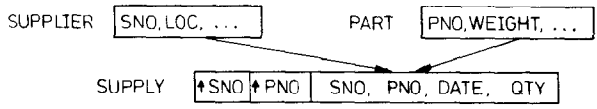
In hierarchical and network models of data often some information is represented by suitable access path structures. For example, the DNO attribute in the EMP relation indicating the membership of a particular employee to his department is dropped from the EMP relation. Instead, a special access path relating each OWNER to his MEMBER tuples is established. Now, this access aid is bearing information relevant to the user [10]. In this case access path data are used to complete the logical data structure of the user. Assume, for example, the following two relations connected by an appropriate path (e.g. pointer chain) with the meaning that a referenced employee belongs to the connected department:



If we drop the access path in this example, the fact that an employee belongs to a particular department is lost, because this kind of information is represented

by the access path. If the connection is implemented through a generalized access path structure all information is preserved when navigational access is from the OWNER tuple. However, if an arbitrary member tuple is located, it is hard to find the OWNER tuple. For this case the following addition may be considered. Each MEMBER tuple keeps the OWNER pointer in its prefix such that the OWNER tuple can be accessed directly. By using the generalized access path structure, all MEMBER tuples can be found without touching the OWNER tuple. In this respect the MEMBER relation can be treated independently. It can be considered as a "complete" relation in which a particular attribute, e.g. DNO, is factored out. Maintenance is only slightly more difficult in this case. A MEMBER tuple can be inserted or deleted without fetching the OWNER tuple. The OWNER pointer is found in the generalized access path structure.

The scheme discussed above can be applied to access paths representing network relationships in the essential and inessential cases. Assume we have three relations, SUPPLIER, PART, and SUPPLY, with the following OWNER-MEMBER relationships:



Each MEMBER tuple in the SUPPLY relation has stored the OWNER pointers to both OWNER relations. Assume that generalized access paths exist for the domains SNO and PNO. Then the location of the OWNER tuples and navigational access from the OWNER tuples to the related MEMBER tuples is achieved via the corresponding generalized access path structures. The access to the different OWNERS of a particular MEMBER tuple is gained by following the OWNER pointers stored in the prefix of the tuple. In our example the attributes SNO and PNO in SUPPLY can be dropped resulting in a change from "inessential" to "essential" OWNER pointers.

Of course, the scheme of keeping two OWNER pointers in the prefix of a MEMBER tuple can be generalized to the representation of all existing OWNER relationships by OWNER pointers in the essential and inessential cases.

5.3 Simple Reorganization Algorithms

Link structures having distributed access path data are difficult to reorganize. The various pointers pointing to a tuple to be moved have to be found and reset properly. Instead of controlling and supplying three different types of pointers in a link, only one *TID* has to be provided per tuple in a generalized access path. The OWNER and MEMBER tuples can be removed independently without affecting each other. Only one *TID* has to be maintained in a generalized access path structure if a tuple is moved to another page.

Because of the centralized access path information, load, reload, and restructuring of tuples should be achieved in a simpler and more efficient way. The design of appropriate algorithms seems to be facilitated with the aid of generalized access path structures. It will be studied in detail in a separate paper.

5.4 Concurrency of Operations

In a multi-user environment concurrent operations on access paths should be allowed without interfering with other operations being performed. Serializing the access along some heavily used access paths can create an unacceptable bottleneck of the entire database system.

It is shown in [4] that concurrent operations on B-trees can be carried out using simple locking protocols. Thus, this solution is adequate for images using B-tree structures. For generalized access path structures the solution presented in [4] should be extended in order to support concurrency on different access paths in one B^*-tree. The author plans to describe this locking protocol in a separate paper.

6. CONCLUSIONS

In summary we have described a new kind of implementation technique of access paths leading to the generalized access path structure. It combines the advantages of link and image structures in retrieval and update operations, and is competitive from a performance point of view.

The presented proposal is based on the observation that the height of the B^*-tree describing the necessary page fetches to access a leaf page remains constant for a wide variety of different leaf page numbers. Therefore, access path data such as twin and parent pointers can be factored out from the tuples and stored more economically in variable length lists of the generalized access path structure. Since these pointers are clustered in single pages, the various kinds of conceivable pointers such as FIRST, NEXT, PRIOR, OWNER, etc., can be represented implicitly by their relative position in the variable length *TID* list. As a result a substantial saving of storage space is gained with this structure.

Finally, this unified approach to access path implementation should reduce the complexity of the system implementation.

ACKNOWLEDGMENT

I would like to thank Rudolf Bayer, Mario Schkolnick, and Irving Traiger for their interest in my work and for helpful discussions of this paper. The comments of the referees are gratefully acknowledged.

REFERENCES

(Note. Reference [6] is not cited in the text.)

1. ASTRAHAN, M.M., ET AL. System R: Relational approach to database management. *ACM Trans. Database Syst. 1,* 2 (June 1976), 97–137.
2. BACHMAN, C.W. Implementation techniques for data structure sets. In *Data Base Management Systems,* D.A. Jardine, Ed., North-Holland Pub. Co., Amsterdam, 1974, pp. 147–157.
3. BAYER, R., AND MCCREIGHT, E. Organization and maintenance of large ordered indexes. *Acta Informatica 1,* 3 (1972), 173–189.
4. BAYER, R., AND SCHKOLNICK, M. Concurrency of operations on B-Trees. IBM Res. Rep. RJ 1791, IBM Res. Lab., San Jose, Calif., May 1976. To appear in *Acta Informatica.*
5. BLASGEN, M.W., CASEY, R.G., AND ESWARAN, K.P. An encoding method for multi-field sorting and indexing. IBM Res. Rep., RJ 1753, IBM Res. Lab., San Jose, Calif., March 1976.
6. BLASGEN, M.W., AND ESWARAN, K.P. On the evaluation of queries in a relational database

system. IBM Res. Rep. RJ 1745, IBM Res. Lab., San Jose, Calif., 1976.
7. CADIOU, J.M. On semantic issues in the relational model of data. Proc. 5th Symp. on Math. Foundations of Compt. Sci. 1976, Gdansk, Poland, *Lecture Notes in Computer Science 45*, Springer-Verlag, pp. 23–38.
8. CODASYL DATA BASE TASK GROUP (DBTG) Report, April 1971 (available from ACM, New York).
9. CODD, E.F. A relational model of data for large shared data banks. *Comm. ACM 13*, 6 (June 1970), 377–387.
10. CODD, E.F., AND DATE, C.J. Interactive support for nonprogrammers: The relational and network approaches. IBM Res. Rep. RJ 1400, IBM Res. Lab., San Jose, Calif., June 1974.
11. ESWARAN, K.P., AND CHAMBERLIN, D.D. Functional specifications of a subsystem for data base integrity. Proc. Int. Conf. on Very Large Data Bases, Framingham, Mass., Sept. 1975, pp. 48–68 (available from ACM, New York).
12. HAERDER, T. An implementation technique for a generalized access path structure. IBM Res. Rep. RJ 1837, IBM Res. Lab., San Jose, Calif., Oct. 1976.
13. WEDEKIND, H. On the selection of access paths in a data base system. In *Data Base Management*, J.W. Klimbie and K.L. Koffeman, Eds., North-Holland Pub. Co., Amsterdam, 1974, pp. 385–397.

Received December 1976; revised October 1977