Resolving Conflicts in Global Storage Design through Replication

R. H. KATZ
University of Wisconsin
and
E. WONG
University of California

We present a conceptual framework in which a database's intra- and interrecord set access requirements are specified as a constrained assignment of abstract characteristics ("evaluated," "indexed," "clustered," "well-placed") to logical access paths. We derive a physical schema by choosing an available storage structure that most closely provides the desired access characteristics. We use explicit replication of schema objects to reduce the access cost along certain paths, and analyze the trade-offs between increased update overhead and improved retrieval access. Finally, we give an algorithm to select storage structures for a CODASYL 78 DBTG schema, given its access requirements specification.

Categories and Subject Descriptors: H.2.1 [**Database Management**]: Logical Design—*data models*; H.2.2 [**Database Management**]: Physical Design—*access methods*

General Terms: Design

Additional Key Words and Phrases: Functional data model, access path selection, storage structure choice

1. INTRODUCTION

A large body of work on physical database design focuses on choosing efficient structures for implementing a set of homogeneous records (index selection, record segmentation, etc.). What is missing is a conceptual framework in which "global access design" for an entire database can be achieved, that is, a specification of storage structures to support access paths between sets of records. Our goal is to develop a design method for global access design based on abstract characteristics of storage structures and freeing us from dependence on low-level implementation details. Specifically, we propose a precise way of describing the access require-

This research was conducted at the University of California, and supported by U.S. Army Research Office Grant DAAG29-76-G-0245, the U.S. Air Force Office of Scientific Research Grant 78-3596, the Honeywell Corporation, and an IBM Predoctoral Fellowship for the first author.
Authors' addresses: R. H. Katz, Computer Science Department, University of Wisconsin, 1210 West Dayton Street, Madison, WI 53706; E. Wong, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720.

ments of a database, called the *access path schema*, a way of determining whether this description can be implemented by existing storage structures, and a method of comparing alternative descriptions to find the most desirable one.

The access path schema is an interface between the logical view (conceptual schema) of a database and the specific access methods and storage structures chosen to support that view (internal schema). We are not the first to recognize the need for an access path description. The DIAM (Data Independent Access Method) framework [12] provides for four levels of description of a database (entity set, string, encoding, and physical device), among which the string-level description most closely corresponds to our access path schema. While similarities exist, we believe that our approach, based on assigning abstract characteristics of storage structure support to logical access paths, represents a more direct response to the problem of global storage design.

We propose a *data model* for defining access path schemata. A *data model* is a collection of data types and constraints used to describe a database and the operations on those data types (e.g., relational and network data models). In contrast, a *schema* is the description of a specific database structured according to some data model.

The paper is organized as follows. First, we introduce a functional data model to describe the logical access paths of a database. We describe their support by assigning them abstract characteristics. Certain combinations of characteristics "conflict," that is, they cannot be supported simultaneously by any conceivable storage structure. Thus we introduce constraints on how the access characteristics are assigned. Any access path schema satisfying these can be implemented by a judicious choice of storage structures. Paths with undesirable access characteristics can be improved by data replication, and we present methods for introducing controlled replication. The access path data model is a functional data model augmented with access characteristics, constraints on their assignment, and rules for introducing and controlling replication of schema objects.

Our approach to access path selection, founded on a constrained assignment of abstract access characteristics to paths, is novel. While it is suited to a "pencil and paper" design method, a more rigorous approach is needed for databases with many schema object types. We describe an integer programming formulation for optimizing the assignment of characteristics to the most frequently traveled paths. An optimum can be found, but the running time is exponential in the number of schema object types. We describe an alternative heuristic algorithm in the appendix.

This paper is a revised and expanded exposition of the ideas first appearing in [9], with a new emphasis on the role of replication to resolve conflicting specifications for the support of access paths.

2. AN ACCESS PATH DATA MODEL

2.1 Introduction

We introduce the access path data model in this section. First, we specify the logical access paths of a database in terms of a functional data model. *Access characteristics* are introduced to describe support for access paths abstractly (Section 2.2). Next, we define a partial ordering on schemata whose logical access

Total functions		Partial functions
ename:	emp → char (20)	mgr: dept → emp
birthyr:	emp → integer	
works-in:	emp → dept	
assignment:	emp → job	
title:	job → char (15)	
salary:	job → integer	
dname:	dept → char (10)	
location:	dept → char (20)	

Fig. 1. Functional specification of example database.

paths have assigned access characteristics, subject to certain constraints. The ordering allows us to define *maximal* schemata, which are those whose assigned characteristics cannot be "improved" (Section 2.3). Finally, we present rules for introducing replicated objects to improve a path's access characteristics (Section 2.4). The access path data model consists of (1) a model for logical access paths with assigned access characteristics, (2) a set of assignment constraints, and (3) rules for introducing new schema objects to improve this assignment. The set of logical access paths among original and replica schema objects, with assigned access characteristics, constitutes the access path schema for a database.

Throughout the paper, we use an example database that stores information about employees, the departments they work in, and the jobs they are assigned. Each employee is described by name (a character string) and year of birth (an integer), each department by name (a character string) and location (a character string), and each job by title (a character string) and salary (an integer).

2.2 Logical Access Paths and Access Characteristics

Our model of logical access paths is based on a reformulation of the Entity-Relationship model that emphasizes functional interrelationships among schema objects [16]. We call it the *functional data model*, and it strongly resembles the models of [1, 6, 13, 14]. However, these other models have not been used for physical database design.

The primitive data types of the model are *object sets* and *functions* between them. Object sets are either *value sets* or *entity sets*. The primary difference is that value sets never appear in the domain of a function. For example, the set of employees is an entity set, while the set of employee names is a value set.

Functions are either *total* or *partial*. The distinction is a constraint on whether every element of the domain set must participate in the functional relationship. For example, since every employee must have a name, the function ENAME: EMP → CHAR(20) is total. If not every department has a manager, then MGR: DEPT → EMP is partial. The functional specification for the example database is given in Figure 1.

Additional constraints are possible. We could distinguish between surjective and injective functions, but since we never use the information, it is not part of our model. We do distinguish between functions that are *one-to-one* and those that are *many-to-one*. Multivalued (many-to-many) relationships are represented by confluent hierarchies as in the CODASYL model, that is, we introduce a "relationship" set and functions to map it into the participating entity sets. For

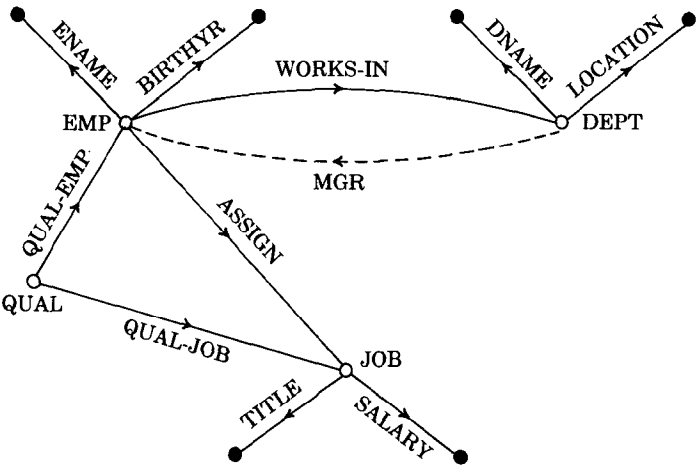


Fig. 2. Example functional database schema. Solid arrows, total; broken arrow, partial.

example, the many-to-many relationship "qualified" among employees and jobs they are qualified to hold is represented by an object set QUAL and the total functions QUAL-EMP: QUAL → EMP and QUAL-JOB: QUAL → JOB. Each QUAL object represents a single interrelationship between an employee and a job. An obvious graphical representation of the schema is shown in Figure 2.

A *query* is the set formed by a composed sequence of functions and inverses. For example, John's department is LOCATION(WORKS-IN(ENAME^{-1}({John}))). This amounts to traversing from "John" to EMP via ENAME, from EMP to DEPT via WORKS-IN, and from DEPT to locations via LOCATION. In the CODASYL model, this is implemented by record and set type traversals, and in the relational model, by selections and joins.

A functional schema only describes which access paths are available, not how well they are supported by the physical schema. An access path schema is the functional schema augmented with zero or more *access characteristics* (defined below) assigned to each of its functions (logical access paths). The access characteristics abstractly describe the properties of the storage structures supporting the access paths. These characteristics are defined in terms of the concepts of *exhaustive scan* and *close proximity.*

An *exhaustive scan* of a set S is one which requires every element of S to be accessed. In terms of actual storage structures, this implies that the cost to perform an exhaustive scan of S is proportional to the cost to access every page (potentially) containing an element of S.

The elements of a set are *in close proximity* if the entire set can be accessed as a unit with minimal cost. Again, in terms of physical structures, this means that the elements of S are placed on as few pages as possible and, further, that these pages are arranged on secondary storage so that they may be read together. Thus, records in the same disk block, track, or cylinder are in close proximity.

Our purpose in defining these concepts in such qualitative terms is to describe a physical database design in an abstract way, without committing ourselves to

too much detail in the specification. The definitions of the access characteristics are

(1) *Evaluated.* Given a in A, $f(a)$ can be found without an exhaustive scan of B.

Example. The function WORKS-IN is "evaluated" if a fast access path exists between EMP and DEPT, for example, a child-to-parent pointer from EMP to DEPT. The function ENAME is "evaluated" if name values are stored in a field within the record that represents the individual employee.

(2) *Indexed.* Given b in B, $f^{-1}(b)$ can be found without an exhaustive scan of A.

Example. WORKS-IN is "indexed" if a fast access path exists between DEPT and EMP, for example, a linked list of EMP records within the same DEPT. ENAME is "indexed" if the file of employees is inverted on the ENAME field.

(3) *Clustered.* The elements of $f^{-1}(b)$ are in close proximity.

Example. WORKS-IN is "clustered" if EMP records within the same DEPT are placed together, for example, EMP sorted by the WORKS-IN attribute or WORKS-IN set membership. ENAME is "clustered" if the file is sorted on employee names.

(4) *Well-Placed.* a and $f(a)$ are stored in close proximity (this is often called a "clustered link").

Example. WORKS-IN is "well-placed" if EMP records with the same DEPT are placed near the associated DEPT record. ENAME is "well-placed" if the file is clustered on name, with common name values factored out of the records, that is, name values are removed by compression.

There are direct analogs for each of these in the CODASYL model. Although we have developed them independently of any particular model, the correspondence reveals the intuitive ideas behind CODASYL's storage structures. While we believe these characteristics to be descriptive of important aspects of storage structure, some concepts are missing. We have no way of describing structures that support access to sequences, that is, the notion of object ordering is not captured in the above.

To illustrate the implications of assigning particular access characteristics to access paths, we present a simple page cost model. A "page" is really a set of hardware pages that can be accessed as a unit, such as a disk track. Let $F: A \rightarrow B$ be a logical access path, $n(A)$ and $n(B)$ be the cardinalities of sets A and B, and PAGES(A) and PAGES(B) be the number of pages containing elements of A and B. Let the cluster factor of function F, denoted as $C_F(A, B)$, be defined as PAGES(A)/$n(B)$, that is, the number of pages on which a particular cluster of A objects can be found.

The worst case page cost to access a B object from a particular A object, assuming F has no assigned characteristic, is the cost to access every page containing B objects: PAGES(B) (i.e., an exhaustive scan). If evaluated, the cost is one page access. If well-placed, the cost is $C_F(A, B) - 1$ (B and its related A's are on pages in close proximity). If pages are large enough to hold the entire domain object cluster and the range object, then the cost of a well-placed access

is zero. We will assume this to be the case. Indexed and clustered characteristics are not relevant when accessing the range from the domain set.

The worst case page cost to access an A object from a particular B object, assuming F has no assigned characteristics, is again an exhaustive scan: PAGES(A). If indexed, the cost is $n(A)/n(B)$, that is, each associated A object is on a different page. If clustered, the cost is $C_F(A, B)$. Again, if well-placed, the cost is $C_F(A, B) - 1$. The evaluated characteristic is not relevant when accessing the domain set from the range.

Assume that every element of every set in the functional schema is assigned to a unique stored record. Replication is explicitly represented by introducing additional data objects into the schema. Under this assumption certain implication rules can be formulated for the characteristics defined above:

(i) well-placed $\Rightarrow$ evaluated

By placing $f(a)$ near a, a fast way to get from the domain to the range is automatically provided. It is no longer necessary to scan the entire set of range objects to find the desired one.

(ii) clustered $\Rightarrow$ indexed

By placing the elements of $f^{-1}(b)$ together, an exhaustive scan of all the domain objects of f is not necessary. Once an object in the cluster has been found, the entire cluster has been found.

(iii) well-placed $\Rightarrow$ clustered

Let $b = f(a)$. "Well-placed" means that a and b are stored together. Since there is one record for each b instance, all A objects with b in the range of f will be placed near b and hence near each other. Thus, clustering is achieved.

"Evaluated" need not imply "indexed," and vice versa. For systems without index storage structures, a mapping can be evaluated but not indexed. For example, an employee's name may be stored in the same record that represents the employee, with no storage structures available to access the record via an employee name. The converse is possible as well. Some inverted file systems allow access to a record through a value associated with the record that is not accessible from it. For example, an employee's name may not be stored with the record that represents the employee, but an index on employee name is available.

Even though "well-placed" implies "evaluated" and "clustered," an access path that is both evaluated and clustered need not be well-placed. If WORKS-IN is well-placed, then employees are clustered on department and placed near their departments. If it is clustered and evaluated, then once again employees are clustered by departments and the employee and department records are interrelated by fast access paths, perhaps on the basis of pointers. The latter organization does not imply the close proximity of employee and department records, and does not benefit from that proximity.

2.3 Partial Ordering and Constraints

The implication rules lead naturally to a partial ordering among the four access path characteristics, shown in Figure 3. As a consequence, an access path schema can be completely described by labeling each edge with one of the six distinct

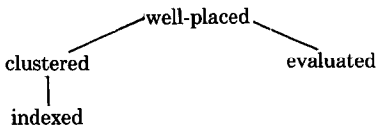


Fig. 3. Partial ordering among characteristics.

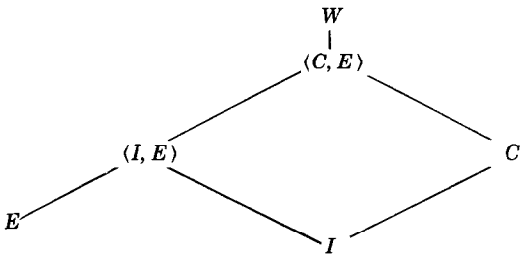


Fig. 4. Partial ordering among labels.

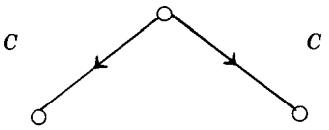


Fig. 5. Cluster constraint.

labels: W, $\langle C, E\rangle$, $\langle I, E\rangle$, C, I, and E, where the letters represent the four characteristics in an obvious way. The partial ordering among labels is shown in Figure 4.

The partial ordering among the labels induces a natural ordering among the access path schemata with the same underlying functional schema, that is, among different labelings of the same schema. We say that S_1 is *greater than* S_2 if S_1 and S_2 have the same functional schema and for each edge e, label$(e, S_1) >$ label(e, S_2).

Intuitively, a "greater" access path schema represents a better global storage design, because access for every path is faster. The "greatest" schema is achieved by labeling every edge with W, but such an assignment cannot be implemented with known storage structures. We characterize "implementability" in terms of conservative (and somewhat pessimistic) constraints on the labels. There are four constraints that *implementable* labelings must meet:

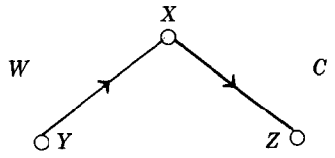
(i) *Cluster Constraint.* At most one outedge of a node can be labeled by a C or W (see Figure 5). Clustering places together all domain objects that share the same range object. It is not possible to partition the domain on more than one function and still achieve this advantageous placement.[1] One-to-one properties do not cause a conflict because such a function partitions the domain objects into clusters of size one. This can always be supported regardless of additional clustering.

(ii) *Placement Constraint.* At most one inedge of a node can be labeled by W (see Figure 6). Well-placement places clusters of domain objects with a common

[1] Admittedly, it may be the case that two or more functions partition the domain into precisely the same partitions. The detection of such a situation requires more semantic information than is available in the access path schema.

Fig. 6. Placement constraint.

Fig. 7. Path constraint.



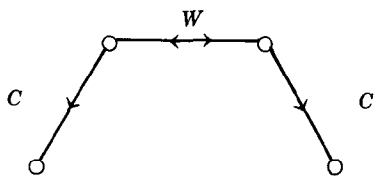


Fig. 8. Example of implied cluster constraint violation.

range object near that range object. It is not possible to achieve this advantageous placement simultaneously for domain objects from more than one function.

(iii) *Path Constraint.* If an inedge of a node is labeled W, then no outedge can be labeled C (see Figure 7). The placement of Y object clusters near their associated X objects destroys the advantageous clustering of the X objects with respect to Z. One-to-one functions do not cause the constraint to be violated.

(iv) *Implied Constraints.* Certain compositions of functions and their properties result in the violation of an above constraint. We say that two nodes are *twins* if they are joined by an edge that is both one-to-one and well-placed. An implied constraint is one that must hold for an equivalent schema in which the twins are combined into a single node. For example, Figure 8 depicts a schema that would cause a violation of a cluster constraint.

The decision of whether to assign "evaluated" to an edge or one of "indexed," "clustered," or "well-placed" is independent. There are no constraints on making access paths indexed or evaluated: conflicts only arise when assigning "clustered" or "well-placed." Throughout the rest of the paper we assume that for the purposes of global storage structure design, each edge is at least indexed and evaluated ($\langle I, E \rangle$). The final stage of design (given in Section 4) maps a labeled schema into the storage structures of a particular database system. It uses traditional techniques to determine if logical access paths assigned evaluated and/or indexed characteristics should receive support. **W**, **C**, and **I** represent the labels $\langle W, E \rangle$, $\langle C, E \rangle$, and $\langle I, E \rangle$, respectively.

A labeling is *maximal* if no constraint is violated, and no other violation-free labeling is greater. Good storage designs must correspond to maximal labelings. Unfortunately, a schema may have many alternative maximal labelings. We

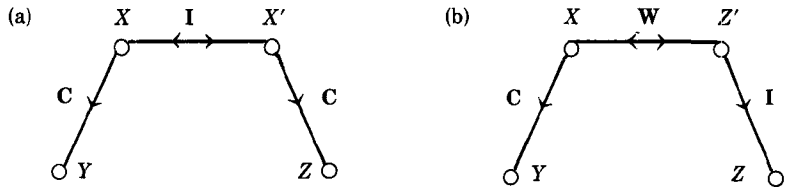


Fig. 9. Resolution of cluster conflicts.

envision global access design to be an interactive procedure involving the following subproblems:

(1) *Optimization.* For a given integrity schema, impose additional criteria for selecting an optimal schema, such as maximization of the degree of support given to the most frequently traversed access paths. The idea is to limit the alternatives to a unique maximal labeling.

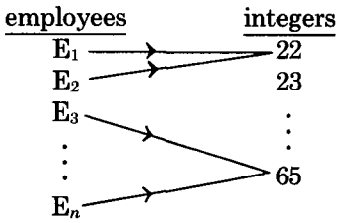
(2) *Resolution of Violations.* If the optimum labeling is not good enough, find the specific paths that have inadequate support and improve their labeling. This necessarily introduces violations of the constraints. These are resolved by data replication, possibly followed by reoptimization.

We now present algorithms to deal with the two key components: optimization and resolution of violations. Because of the length of the optimization algorithm, we first deal with resolution by replication.

2.4 Conflicts and Replication

A *conflict* is a violation of a cluster, placement, or path constraint. A labeling algorithm can introduce conflicts into a labeled schema, as long as these are eventually resolved by replicating schema objects. Copies of existing schema objects are created and are incorporated into the schema as follows. A *replica* is a new object set, derived from an original object set. New functions are defined to map the replica into (some of) the same range sets as the original. An additional function is defined to interrelate the replica and its original.

A cluster conflict is resolved by two types of replication (see Figure 9; replicated objects are primed). In type (a), a replica of X is made, X', and both the original and the copy are clustered on the appropriate ranges. In type (b), a copy of Z is made, Z'. There is a Z' object for each X to represent the Z instance associated with that X. Further, the X and Z' are placed near each other. To illustrate this, consider the entity set employees and the value set integers, interrelated by the function age. Schematically, the following situation can arise:



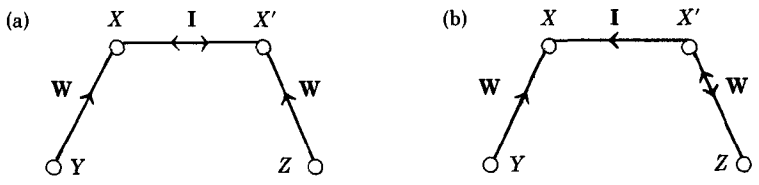


Fig. 10. Resolution of placement conflicts.

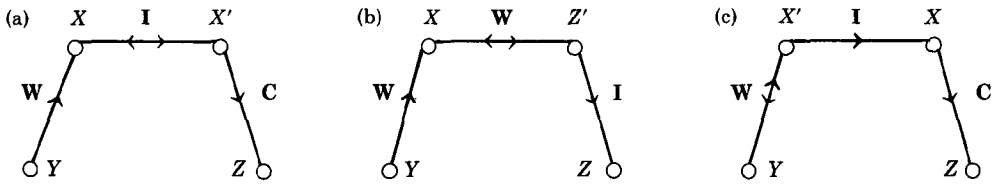
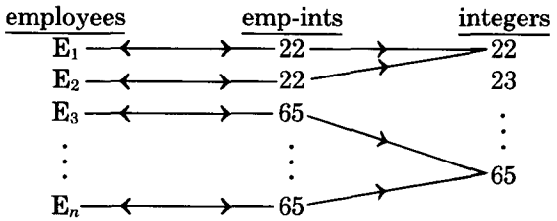


Fig. 11. Resolution of path conflicts.

The effect of type (b) cluster resolution is to replicate the age values so there is one age value per employee:



A placement conflict is resolved analogously. The methods appear in Figure 10.

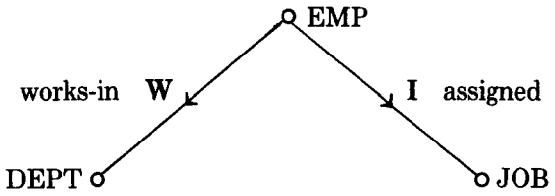
Path conflicts are also resolved via replication (see Figure 11). However, to simplify the labeling algorithms, we always enforce path constraints.

In the above, we introduced replicated nodes to resolve conflicting labelings. The assignment of edges of the original node to the replica is a difficult optimization problem. Some edges, such as the one that causes the conflict, are *partitioned* among them. Any of the other edges can also be *replicated.* Edge partitioning involves the same considerations as the segment partitioning problem described in [11]. We assume that only conflicting edges are partitioned, with the replica receiving the conflicting edge of lower access frequency. In addition, we always replicate the value set mappings of the original to the replica, since replication of values is easy to support automatically. The frequency with which these edges are accessed is assumed to be the same as in the original schema.

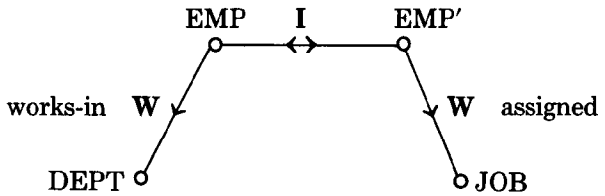
The schemes for conflict resolution are guaranteed to satisfy implied constraints. A one-to-one edge only appears as the result of replication or as a distinguished "identifier" attribute of an object (there is at most one per object type).

As shown above, a conflict is always resolved through replication. The *degree* of a schema is the number of conflicts that an algorithm may introduce during labeling. It is therefore a coarse measure of the amount of replication we are willing to tolerate during the labeling process, since it does not take into account the cardinalities of the replicated objects (the appendix contains an algorithm based on a more practical measure of replication: the expansion factor). Replicated information introduces increased costs for storage and update, while reducing retrieval costs. For example, consider a resolved cluster conflict among EMP, DEPT, and JOB:

Without replication:



With replication:



Access from JOB to EMP′ to access these replicated value set mappings is improved in the second schema. Meanwhile, access from JOB to DEPT, or to any nonvalue set object, is not impaired and has no worse cost than in the original schema. However, updates to values associated with an EMP, or the insertion or deletion of EMP objects, must be propagated to all copies. This can result in a considerable update overhead. If the frequency of retrieval from JOB to values of EMP exceeds the frequency of updates of EMP, then replication is cost-effective (we assume that the cost of retrieval and update are the same).

Replication of value set objects to domain objects can be accomplished without incurring additional update cost, although there is an increase in storage requirements. When modifying a value, the relationship between the domain and the value set is changed, rather than the value itself. For example, if a department's location is changed from Berkeley to Madison, then the name of the location has not been recoded (Berkeley has not been renamed Madison), but rather LOCATION now maps into a different value. Updates need not be propagated to replicated location values.

The choice of which type of replication to use is based on update frequencies and object cardinalities. For cluster resolution, the replicated objects are placed in one-to-one correspondence with the X objects. Because the same number of object instances is created in either method, the choice depends on which of X or Z is updated most frequently. Further, under type (b) replication, each update to

Z must be propagated to $n(X)/n(Z)$ replica instances, where $n(X)$ is the cardinality of X schema objects in the schema. If U_X and U_Z are the update frequencies for objects X and Z, respectively, then type (a) is chosen when $U_X * 1 < U_Z * n(X)/n(Z)$, and vice versa for type (b). Type (b) is always chosen to resolve conflicts involving value set edges, because the update cost in that case is zero.

It is never advantageous to choose type (b) replication to resolve placement conflicts. X objects are placed in one-to-one correspondence with Z objects, and the functional relationship implies that there are more Z objects than X objects. Thus, more data must be replicated. Further, since X objects are replicated in both types, the same update frequencies apply. Type (b) must incur higher update costs.

We use replication incrementally. First, a design is formulated with no replication, that is, degree = 0. Then the degree is increased, and the design is recomputed. The designer must evaluate the increased retrieval efficiency versus the increased update costs. For retrieval-oriented databases, replication is a useful technique for improving performance. Under many other situations, this will not be the case.

3. OPTIMIZATION

3.1. Introduction

Because not all labelings are comparable, there may be many maximal labelings for the same schema. Rather than generate them all, usage information is exploited to restrict the enumeration to those that best support the expected usage patterns of the database. In this section we present an algorithm for generating a maximal labeling that specifies superior support for the access paths most heavily traveled. Labeling can be formulated as a pure integer linear program, and it is solved by standard branch-and-bound methods [15]. Because of the high computational costs associated with this approach, we present an alternative hill-climbing algorithm, which heuristically assigns labels to edges, in the appendix. Although the latter is suboptimal in that only a local optimum is found, our limited experience indicates that the approach often finds optimal labelings for typical database schemas. However, a complete evaluation of the heuristic algorithm is beyond the scope of this paper. Throughout this section, we limit the integer programming formulation to avoid path conflicts.

We model usage information by assigning access frequencies to schema edges. The access paths of the functional schema are logical and imply nothing about the physical organization of the database. They represent ways in which schema objects are meaningfully joined, yet they need not be efficiently supported in the underlying schema. A database administrator assigns access frequencies by analyzing the correlation between the access of a particular object, say, employees, and other related objects, such as departments worked in, in the expected mix of queries. If there is a high correlation between accessing employees and accessing their departments, then WORKS-IN should receive a high access frequency. The database administrator assigns frequencies to expected queries, which are paths through the schema. The frequency of an edge is the sum of the frequencies of the queries that traverse it.

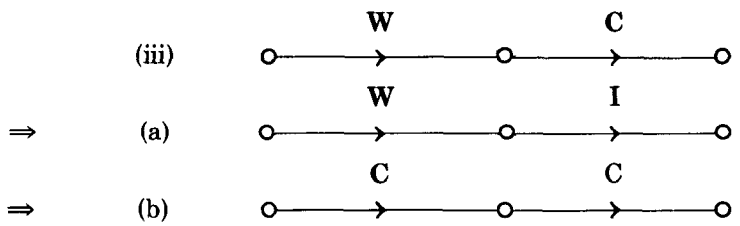
3.2. Integer Programming Formulation

We view labeling as an assignment problem in which the label **W** ($\langle W, E \rangle$), **C** ($\langle C, E \rangle$), or **I** ($\langle I, E \rangle$) is assigned to each edge of the functional schema, subject to the constraints and the allowable number of placement or cluster conflicts that may be made during the label assignment (the degree of replication). Induced conflicts must be resolved using the techniques of the previous section. An integer program is one of several ways of formulating a labeling algorithm. Another is found in the appendix.

The input to the algorithm is a graphical representation $G = (V, E)$ of the schema to be labeled, a ranking of the edges according to frequency of traversal, and the degree of replication N. We formulate an integer linear program from this information.

For each edge i, we introduce the decision variables W_i and C_i which are restricted to being zero or one. If W_i is set to one, then the edge has been labeled **W**; if C_i is one, then the edge has been labeled **C**. Otherwise, both variables are zero, and the edge has been labeled **I**.

Since we seek to maximize the number of well-supported edges, the objective function maximizes a weighted sum of the frequencies of the edges labeled **W** or **C**. A violation of a type (iii) constraint is resolved in one of two ways:



We must quantify the trade-off between labeling an edge **W** or **C**. Let k_1 and k_2 be two constants that represent the trade-off between accessing an object via an edge labeled **W** and one labeled **C**. The advantage of a well-placed edge is that both the range and domain of the mapping reside near each other in secondary memory. Thus, in accessing one, the other is accessed for "free." This is a significant savings, since the time to access a new object from secondary storage is often tens of milliseconds, versus tens of microseconds to access data already in storage. The ratio k_1/k_2 is the same as (time to access a page + time to process a page)/(time to process a page).

If f_i is the traversal frequency of edge i, then the objective function is

$$\max k_1 \sum_{i \in E} f_i W_i + k_2 \sum_{j \in E} f_j C_j.$$

Each of the labeling constraints is expressed in terms of linear constraints in W_i and C_i. The first set ensures that no edge may simultaneously be labeled **W** and **C**:

$$W_i + C_i \leq 1 \qquad \text{for each} \quad i \in E.$$

The cluster constraints are represented by a set of inequalities that guarantee that at most one outedge of a given node may be labeled **W** or **C**:

$$\sum_{i \in \text{outedge}(j)} (W_i + C_i) \leq 1 \qquad \text{for each} \quad j \in V.$$

The placement constraints are represented as a set of inequalities that guarantee that at most one inedge of a given node may be labeled **W**:

$$\sum_{i \in \text{inedge}(j)} W_i \leq 1 \qquad \text{for each} \quad j \in V.$$

The path constraints are represented by a set of inequalities that ensure if an inedge is labeled **W**, then no outedge may be labeled **W** or **C**:

$$\sum_{i \in \text{inedge}(j)} W_i + \sum_{k \in \text{outedge}(j)} (W_k + C_k) \leq 1 \qquad \text{for each} \quad j \in V.$$

The number of constraints appears large: there are $|E| + 3|V|$ of them. However constraints of the second and third type are only required if a node has more than one outedge or inedge, respectively. Many nodes do not, for example, those that represent value objects. The fourth type is only needed for nodes with both inedges and outedges. Again, value nodes can be ignored. Further, identifier edges can be ignored in the formulation, because such edges are always labeled **W**.

The fourth type of constraint is a linear combination of the second and third types. However, this is because we have not yet included the degree of replication in the formulation. For each node j, we introduce the decision variables $X_{j,1}$, $X_{j,2}$, constrained to be nonnegative integers, and we modify the constraints as follows: for each $j \in V$,

$$\sum_{i \in \text{outedge}(j)} (W_i + C_i) - X_{j,1} \leq 1,$$

$$\sum_{i \in \text{inedge}(j)} (W_i) - X_{j,2} \leq 1,$$

$$\sum_{i \in \text{inedge}(j)} (W_i) + \sum_{i \in \text{outedge}(j)} (W_i + C_i) - X_{j,1} - X_{j,2} \leq 1.$$

The $X_{j,k}$ variables count the number of violations of the cluster and placement constraints. An additional constraint must now be added:

$$\sum_{j \in V} (X_{j,1} + X_{j,2}) \leq N,$$

where N is the degree of replication.

The complete formulation is reproduced here:

$$\max k_1 \sum_{i \in E} f_i W_i + k_2 \sum_{j \in E} f_j C_j$$

such that for each $i \in E$,

$$W_i + C_i \leq 1;$$

for each $j \in V$ such that $|\text{outedge}(j)| > 1$,

$$\sum_{i \in \text{outedge}(j)} (W_i + C_i) - X_{j,1} \leq 1;$$

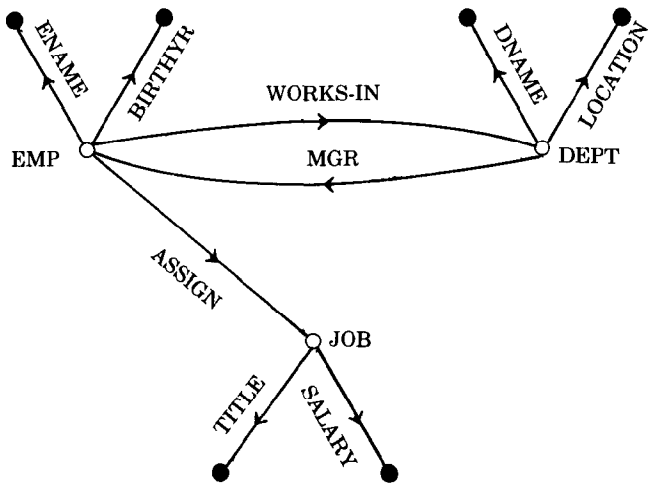


Fig. 12. Employee database subschema.

for each $j \in V$ such that $|\text{inedge}(j)| > 1$,

$$\sum_{i \in \text{inedge}(j)} (W_i) - X_{j,2} \leq 1;$$

for each $j \in V$,

$$\sum_{i \in \text{inedge}(j)} (W_i) + \sum_{j \in \text{outedge}(j)} (W_k + C_k) - X_{j,1} - X_{j,2} \leq 1$$

$$\sum_{j \in V} (X_{j,1} + X_{j,2}) \leq N$$

$$1 \geq C_i,\ W_i \geq 0, \quad \text{integer}$$

$$X_{j,1}, X_{j,2} \geq 0, \quad \text{integer}.$$

The optimal solution is found by applying one of the generalized branch-and-bound algorithms developed for solving integer programming problems. Further study may lead to an algorithm particularly well suited to solving these kinds of formulations.

The formulation requires only a minor change if path conflicts are permitted. Type (ii) and (iii) constraints are dropped, and $X_{j,1}$ and $X_{j,2}$ are combined into a single variable X_j, constrained to be nonnegative and integer.

We illustrate our approach with an example. Consider the subschema of the Employee database in Figure 12. A ranking of edges and their frequencies is

(1)	WORKS-IN,	frequency = 30
(2)	ASSIGN,	21
(3)	TITLE,	15
(4)	ENAME,	10
(5)	DNAME,	8
(6)	LOCATION,	7
(7)	SALARY,	5
(8)	MGR,	5
(9)	BIRTHYR,	1

We assume that $k_1 = 30$ and $k_2 = 1$. For a degree of replication equal to zero, the formulation becomes

$$\max 30 \left(\sum_{i \in E} (f_i W_i) \right) + \sum_{j \in E} f_j C_j$$

such that

$$W_{\text{WORKS-IN}} + C_{\text{WORKS-IN}} \leq 1$$

$$W_{\text{ASSIGN}} + C_{\text{ASSIGN}} \leq 1$$

$$W_{\text{TITLE}} + C_{\text{TITLE}} \leq 1$$

$$W_{\text{ENAME}} + C_{\text{ENAME}} \leq 1$$

$$W_{\text{DNAME}} + C_{\text{DNAME}} \leq 1$$

$$W_{\text{LOCATION}} + C_{\text{LOCATION}} \leq 1$$

$$W_{\text{SALARY}} + C_{\text{SALARY}} \leq 1$$

$$W_{\text{MGR}} + C_{\text{MGR}} \leq 1$$

$$W_{\text{BIRTHYR}} + C_{\text{BIRTHYR}} \leq 1$$

$$W_{\text{ENAME}} + C_{\text{ENAME}} + W_{\text{BIRTHYR}} + C_{\text{BIRTHYR}} + W_{\text{WORKS-IN}} + C_{\text{WORKS-IN}} + W_{\text{ASSIGN}} + C_{\text{ASSIGN}} - X_{\text{EMP},1} \leq 1$$

$$W_{\text{DNAME}} + C_{\text{DNAME}} + W_{\text{LOCATION}} + C_{\text{LOCATION}} + W_{\text{MGR}} + C_{\text{MGR}} - X_{\text{DEPT},1} \leq 1$$

$$W_{\text{TITLE}} + C_{\text{TITLE}} + W_{\text{SALARY}} + C_{\text{SALARY}} - X_{\text{JOB},1} \leq 1$$

$$W_{\text{WORKS-IN}} + W_{\text{DNAME}} + C_{\text{DNAME}} + W_{\text{LOCATION}} + C_{\text{LOCATION}} + W_{\text{MGR}} + C_{\text{MGR}} - X_{\text{DEPT},1} \leq 1$$

$$W_{\text{MGR}} + W_{\text{ENAME}} + C_{\text{ENAME}} + W_{\text{BIRTHYR}} + C_{\text{BIRTHYR}} + W_{\text{WORKS-IN}} + C_{\text{WORKS-IN}} + W_{\text{ASSIGN}} + C_{\text{ASSIGN}} - X_{\text{EMP},1} \leq 1$$

$$W_{\text{ASSIGN}} + W_{\text{TITLE}} + C_{\text{TITLE}} + W_{\text{SALARY}} + C_{\text{SALARY}} - X_{\text{JOB},1} \leq 1$$

$$X_{\text{EMP},1} + X_{\text{DEPT},1} + X_{\text{JOB},1} \leq 0$$

The program is solved by Balas' algorithm for zero-one integer programs [15]. The resulting labeling is

$$W_{\text{WORKS-IN}} = W_{\text{TITLE}} = 1, \qquad \text{all others} = 0$$

WORKS-IN is the highest frequency edge. By labeling it **W**, the outedges of DEPT and the other outedges of EMP can be labeled no better than **I**. This allows the most frequently used outedge of JOB, that is, TITLE, to be labeled **W**.

If the degree is increased to 1, then the two highest frequency edges, WORKS-IN and ASSIGN, can both be labeled **W**, even though this introduces a cluster conflict. All other edges are labeled **I** owing to path and cluster constraints. The conflict is resolved by replicating EMP objects as described in Section 2.3.

4. IMPLEMENTING A CODASYL SCHEMA

Up to this point, the design is independent of the actual data model and system. In this section we briefly discuss the considerations involved in mapping a labeled

schema into DBTG storage structures [3]. In [7], Katz describes how the mapping can be performed for standard relational storage structures.

The quality of the mapping depends on the detail of usage information specified. In the following, we assume that information is specified with the same detail as the previous section. All value set mappings are at least "evaluated" by first replicating values and then placing them together within the single record that represents the domain object. This is the conventional way to represent value sets in CODASYL.

In the most recent CODASYL proposal [3], many parameters of physical database design have been removed from the schema DDL and localized in data storage definition. The DSDL provides facilities for specifying the pagination of the storage media, schema to storage record mapping, record pointer implementation, set representation, and storage record placement. We do not deal with the specification of the storage media, and assume that all sets are represented by chains with direct pointers. Additional usage information could be used to make a more sophisticated choice for these parameters.

The DSDL provides three choices for the record placement strategy. A record may be calc'd (hashed) on a key specified in the DDL, clustered by set membership and optionally placed near the owner, or stored in sequential sorted order. Indexes can be specified separately for keys specified in the DDL. Again, at most one nonidentifier outedge of a node may be labeled **W** or **C**. This should be selected as the edge to determine the record type's primary structure, unless the traversal frequency of the identifier outedge is greater than this edge's frequency. In that case, the identifier outedge is selected. If the selected outedge represents an identifier, the record type is calc'd on the related key data item. If it represents a value set mapping, then the record type is stored sequentially and sorted and indexed on the appropriate data item. Otherwise the outedge represents a mapping into an entity set, and the record type is clustered on the CODASYL set used to implement that mapping. If **W** is specified, the records are placed near the owners. Indexes are created for data items whose associated value-set mappings are labeled **I**. The rules of Figure 13 are used to determine the record type's structure.

To simplify the following discussion, we introduce the identifiers JID, DNO, and ENO for JOB, DEPT, and EMP, respectively. The DSDL specification for the degree 0 schema of the previous section is

```
STORAGE RECORD NAME IS DEPT
  PLACEMENT IS SEQUENTIAL ASCENDING DNO
  SET WORKS-IN ALLOCATION IS STATIC
    POINTER FOR FIRST, LAST RECORD EMP
      IS TO EMP
  SET MGR ALLOCATION IS STATIC
    POINTER FOR NEXT, PRIOR
    POINTER FOR OWNER
STORAGE RECORD NAME IS JOB
  PLACEMENT IS SEQUENTIAL ASCENDING JID
  SET ASSIGN ALLOCATION IS STATIC
    POINTER FOR FIRST, LAST RECORD EMP
      IS TO EMP
```

```
Algorithm CodasylPhysicalDesign
FOR EACH nonvalue node V DO
  IF node has an identifier (one-to-one) outedge THEN
    LET i = identifier outedge
    LET j = other outedge labeled W or C
    IF fi > fj
      THEN calc record type on key data item
      ELSE
        IF j is a mapping into a nonvalue set
          THEN cluster record type on set membership
            IF edge label is W
              THEN place near owner
          ELSE sort and index on value data item
    FOR EACH mapping labeled I DO
      IF value set mapping THEN index on that data item
      ELSE owner-coupled set implements the indexing
  ELSE /* node has no distinguished identifier */
    FOR outedge labeled W or C
      IF it is a mapping into a nonvalue set
        THEN cluster record type on set membership
          IF labeled W THEN place near owner
        ELSE sort and index on value data item
    FOR EACH mapping labeled I DO
      IF value set mapping THEN index on that data item
      ELSE owner-coupled set implements the indexing
```

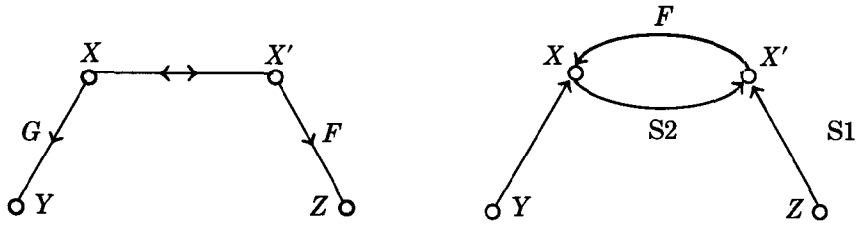
Fig. 13. CODASYL storage structure choice.

```
STORAGE RECORD NAME IS EMP
  PLACEMENT IS CLUSTERED VIA SET WORKS-IN NEAR OWNER DEPT
  SET WORKS-IN ALLOCATION IS STATIC
    POINTER FOR NEXT, PRIOR
    POINTER FOR OWNER
  SET MGR ALLOCATION IS STATIC
    POINTER FOR FIRST, LAST RECORD DEPT
      IS TO DEPT
  SET ASSIGN ALLOCATION IS STATIC
    POINTER FOR NEXT, PRIOR
    POINTER FOR OWNER
```

plus specification for INDEXES for each data item not covered in the above. All access mappings are maximally supported. Support for infrequently traveled paths that have been labeled I may be dropped at the choice of the designer.

Replication is introduced by modifying the logical schema. CODASYL provides mechanisms to support the automatic propagation of updates to all replicated objects. For each replication method, a new record type is introduced to represent the replica, along with new sets to implement the interrelationship with the original. CODASYL allows the data items of a member record to be inherited from its owner. Thus the original record can own its replicate(s) and provide it (them) with their data item values. Changes can propagate from owners to members, and vice versa.

Replication type (a) for cluster conflicts results in the following schema modification:

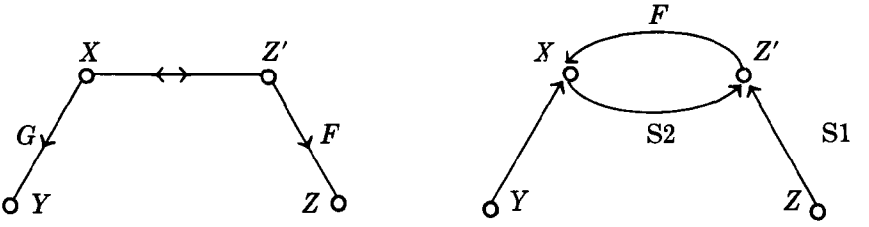


CODASYL sets with automatic/mandatory members implement a total function from the member record type to the owner record type. The member record type is called the *total member* of the set. X' is made a total member of set S1. If a Z record is deleted, then all associated X' records are also deleted. X' is also made a total member of S2. If an X record is deleted then its X' replicas are also deleted. X is made a member of set F with X' as owner. X is a total member only if F was derived from a total function. Thus a deletion of X' through S1 propagates to X. Because a cycle of total functions is not allowed, the membership of A within X must have manual specified rather than automatic. The following sequence is necessary to insert an X record into the schema:

(1) STORE X [X is made a member of every set derived from one of the total functions in which it appears in the domain, except for F]
(2) STORE X' [X' is connected to Z via S1 and X via S2 automatically]
(3) CONNECT X TO F [X is connected to X' manually]

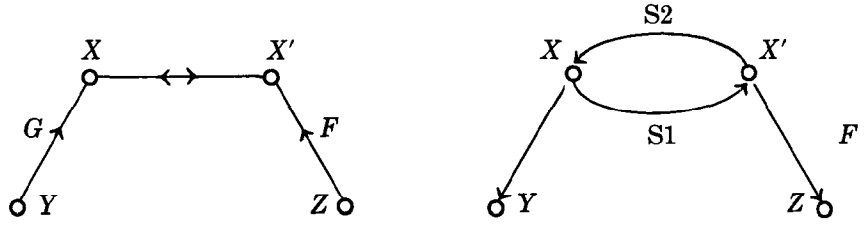
If F is a total function, then (2) and (3) must be executed at the same time as (1). Otherwise they are executed when A first participates in the domain of a partial function.

The schema conversion for type (b) is similar:



The set membership options and insertion sequence are as before.

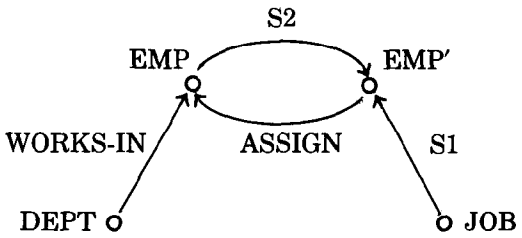
For type (a) placement resolution, the conversion is



X' is a total member of S1. X is a total member of S2, although manual must be

specified. If F is a total function, then the deletion of an X record will cause its replica X' to be deleted via S1, which in turn causes Z to be deleted. A must be stored first. Then X' is stored and X is connected to X' through S2.

In the replicated schema of the previous section, type (b) cluster resolution was applied to EMP. The schema becomes



Additions to the DDL for the employee database are

```
RECORD NAME IS EMP'
  KEY IS EMP'-KEY IS ENO DUPLICATES ARE NOT ALLOWED
  01 ENO TYPE IS BINARY
     SOURCE IS ENO IN OWNER OF S2
  01 ENAME TYPE IS CHARACTER 20
     SOURCE IS ENAME OF OWNER IN S2
  01 BIRTHYR TYPE IS BINARY
     SOURCE IS BIRTHYR OF OWNER IN S2
SET NAME IS S1
  OWNER IS JOB
  MEMBER IS EMP'
  INSERTION IS AUTOMATIC RETENTION IS MANDATORY
  SET SELECTION IS THRU
    JOB OWNER IDENTIFIED BY KEY JOB-KEY
SET NAME IS S2
  OWNER IS EMP
  MEMBER IS EMP'
  INSERTION IS AUTOMATIC RETENTION IS MANDATORY
  SET SELECTION IS THRU
    EMP OWNER IDENTIFIED BY KEY EMP-KEY
SET NAME IS ASSIGN
  OWNER IS EMP'
  MEMBER IS EMP
  INSERTION IS MANUAL RETENTION IS MANDATORY
  SET SELECTION IS THRU
    EMP OWNER IDENTIFIED BY KEY EMP'-KEY
```

The placement and cluster attributes of the set S1 are specified in the DSDL as given above for the degree 0 schema. EMP' is clustered on its membership in set S1 and placed near the owner JOB record.

5. CONCLUSIONS

We have presented a novel approach to physical database design, based on assigning implementation independent properties of storage structures to logical access paths, and then using this specification to derive a physical database

schema. Redundant data were explicitly introduced into the schema to improve the cost of retrieval along certain heavily traveled paths. Rather than introduce replication in a haphazard way, we have integrated it with the logical schema. This enables us to control the propagation of updates to replicas.

The access path data model not only provides a mechanism to capture the information needed for physical database design, but also serves as a system independent description of the physical schema. The latter aids in program and schema translation activities as well as database design. For example, we have studied how to translate queries written in CODASYL DML into relational queries by examining how the DML object-at-a-time operations manipulate and traverse among the objects of an access path description of the CODASYL schema [10]. A related paper describes how to use the access path schema description to optimize the mapping of relational queries into CODASYL DML [8]. The access path description can also be exploited in choosing storage structures for the target schema during schema translation.

A student of Eugene Wong has implemented an interactive logical database design tool in the spirit of the work in [4, 5]. We hope to extend this tool with the techniques described here to demonstrate the feasibility of our approach to physical database design.

APPENDIX. SUBOPTIMAL LABELING ALGORITHM

The integer programming formulation of Section 3.2 is a costly method for determining an optimal labeling. In this appendix we present another algorithm for labeling that is more procedural, but is not guaranteed to find an optimal solution. Conflicts are resolved as soon as they occur, and information about data volumes is used to control replication.

The input to the algorithm is: a schema $G = (V, E)$ to be labeled, a ranking of edges and their associated access frequencies, update frequencies, and object cardinalities for conflict resolution, and a limit L on the *expansion factor*. The latter is the ratio of the size (in bytes) of a schema with replicated objects to the size of the original schema. The algorithm is formulated to always enforce path constraints.

It proceeds in two phases (see Figure 14). The first phase produces a conflict-free labeling. It ignores all one-to-one edges, because these can always be labeled **W**. Initially, all edges are labeled **I**. The edges are visited in frequency order and are labeled **C** as long as the labeling does not cause a cluster conflict. Edges that are labeled **C** are again visited in frequency order and are labeled **W** if no placement conflict would result. At this point, the schema is free of placement and cluster conflicts, but not path conflicts. These must be eliminated.

A *path* is defined as the maximal sequence of distinct edges, adjacent in the functional direction, such that each adjacent pair of edges causes a path conflict. To derive a "good" final labeling, the entire path is examined, rather than each pair of conflicting edges. Paths can be resolved independently (see [7] for proof).

To resolve a path conflict, the path must first be detected. At most one path passes through any node (see [7] for proof). If a node is on a path, that is, it has an inedge labeled **W** and an outedge labeled **W** or **C**, then the path is extended in either direction until the maximal path of conflicting edges is found. Because an edge can only appear on a single path, once an edge has been associated with

```
LABELSCHEMA:
  /* Phase 1 Labeling */
  all edges are initially labeled I
  FOR EACH edge in frequency order DO
    IF no cluster conflict THEN label ← C
  FOR EACH edge labeled C in frequency order DO
    IF no placement conflict THEN label ← W

  /* Path Resolution */
  RESOLVEPATHS

  /* Phase 2 Labeling */
  FOR EACH edge in frequency order DO
    IF edge is labeled I THEN
      label ← C
      IF path conflict & edge appeared on resolved path
        THEN label ← I
        ELSE
          IF cluster conflict THEN
            choose replication method
            IF expansion factor > L
              THEN label ← I
              ELSE replicate to resolve conflict
    IF edge is labeled C THEN
      label ← W
      IF path conflict & edge appeared on resolved path
        THEN label ← C
        ELSE
          IF placement conflict THEN
            choose replication method
            IF expansion factor > limit
              THEN label ← C
              ELSE replicate to resolve conflict

  /* Path Resolution */
  RESOLVEPATHS

RESOLVEPATHS:
  /* Find the paths through the schema, and relabel them to eliminate path conflicts */
  FOR EACH node n DO
    L ← node n's path (DON'T revisit nodes on this path)
    IF L ≠ { } THEN
      generate all labels for path L
      evaluate each l ∈ labels
      maxlabel ← maximum cost l
      relabel the schema using maxlabel
```

Fig. 14. Labeling a schema.

a path it is excluded from further consideration. Thus, it is sufficient to visit each node in the schema once to determine all paths unambiguously.

The next step is to generate all conflict-free labelings for each path. For a path of length n, the valid labelings are strings of length n from the regular language $((WI) * C *) * [W]$, that is, strings over W, C, and I such that no W is followed by a C or W. The labeling that contributes the most to the objective function is chosen as the final labeling.

The second phase introduces replication and relabels those edges that can be improved without introducing new conflicts. Each edge is visited in frequency

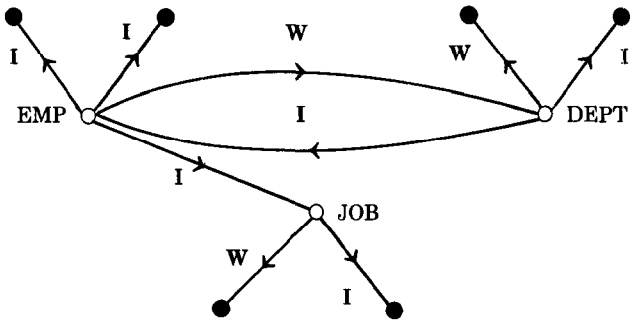


Fig. 15. Schema before path resolution.

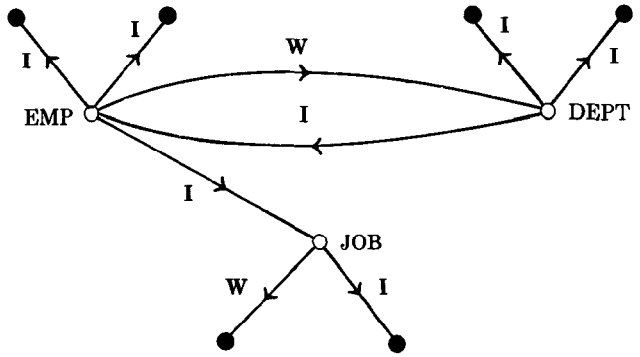


Fig. 16. Conflict-free labeling of schema.

order. If it is labeled **I**, it is temporarily relabeled **C**. The relabeling stands as long as (1) no new conflicts are introduced, (2) a cluster conflict is introduced and the expansion factor does not exceed the limit, or (3) a path conflict has been introduced and this edge has not been involved in a previous path conflict. The latter case is included to ensure that the labeling does not oscillate. The cluster conflict is immediately resolved by choosing one of the replication methods of Section 2.4, given the update frequencies and object cardinalities.

Next, the edge is relabeled **W**. This relabeling holds as long as (1) no new conflict is introduced, (2) a placement conflict is introduced and the limit has not been exceeded, or (3) a path conflict has been caused and this edge has not been involved with a path conflict before. The placement conflict is immediately resolved by introducing replication.

We illustrate the algorithm with the example of Section 3.2. It proceeds by first labeling WORKS-IN with **C**. ASSIGN, ENAME, and BIRTHYR must remain labeled **I** because of cluster constraints. TITLE is labeled **C**, causing SALARY to remain labeled **I**. Finally, DNAME is labeled **C**. All edges labeled **C** can be relabeled **W** because the schema cannot have a placement conflict. The schema before path resolution is given in Figure 15.

The path consisting of WORKS-IN and DNAME must be resolved. The labeling [*W*, *I*] maximizes the objective function and is chosen. The maximal, conflict-free schema of degree 0 and expansion factor 1 is shown in Figure 16.

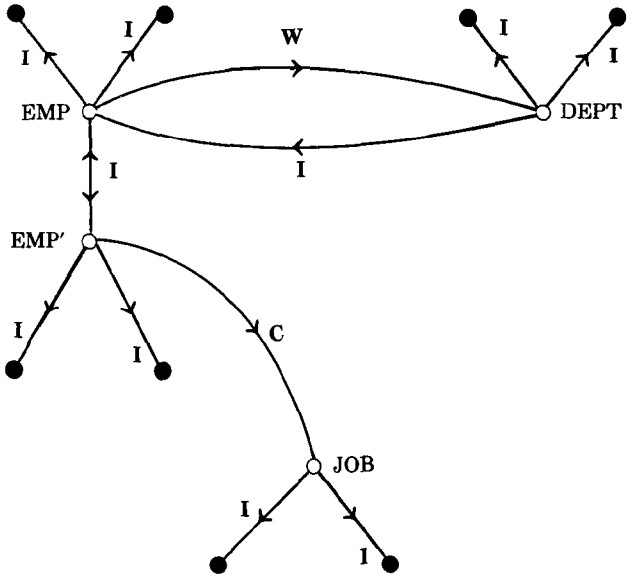


Fig. 17. Resolution of conflicting labeling.

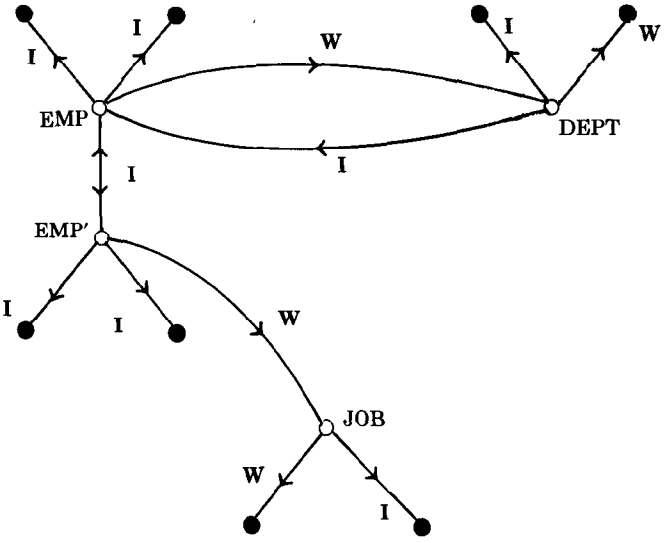


Fig. 18. Labeling before path resolution.

If the expansion factor limit is made greater than 1, then the algorithm will attempt to label ASSIGN with a C, causing a cluster conflict with WORKS-IN. Assume that $U_{EMP} = 50$, $U_{JOB} = 10$, $n(EMP) = 1000$, and $n(JOB) = 100$. Type (a) is chosen because $U_{EMP} < U_{JOB} * n(EMP)/n(JOB)$. The size of the schema has increased by $n(EMP) * [SIZE(ENO) + SIZE(ENAME) + SIZE(BIRTHYR)]$ where $SIZE(X)$ is the number of bytes needed to encode an object X. If the expansion factor does not exceed the limit, the schema is labeled as in Figure 17.

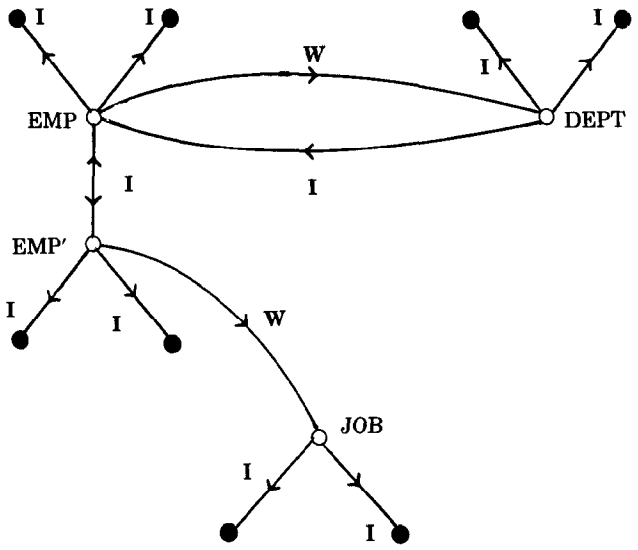


Fig. 19. Final labeling.

ASSIGN can be immediately relabeled **W**, causing a path conflict with TITLE. For simplicity, we assume that no further replication is possible without exceeding the limit. Even though a path conflict is introduced, LOCATION is relabeled **W** because it has not been previously involved in a path conflict. The schema before final path resolution is given in Figure 18.

The labeling [*W*, *I*] maximizes the objective function for both paths. The final maximal, conflict-free labeling of degree 1 is shown in Figure 19.

ACKNOWLEDGMENTS

We thank the referees and Professor Philip Bernstein for their many insightful comments and criticisms. The result has been a much improved and more readable paper.

REFERENCES

(Note. Reference [2] is not cited in the text.)

1. BUNEMAN, P., AND FRANKEL, R.E. FQL—A functional query language. In *Proc. ACM—SIGMOD Int. Conf. Management of Data* (Boston, Mass., May 30–June 1), ACM, New York, 1979, pp. 52–57.
2. CHEN, P.P., AND YAO, S.B. Design and performance tools for data base systems. In *Proc. 3rd Int. Conf. Very Large Data Bases* (Tokyo, Oct. 6–8), ACM, New York, 1977, pp. 3–15.
3. CODASYL data description language. *J. Dev.*, Jan. 78.
4. GAMBINO, T.J., AND GERRITSEN, R. A database design decision support system. In *Proc. 3rd Int. Conf. Very Large Data Bases* (Tokyo, Oct. 6–8), ACM, New York, 1977, pp. 534–544.
5. GERRITSEN, R. Tools for the automation of database design. *Proc. NYU Symp. Database Design* (May) 1978.
6. HOUSEL, B., YAO, S.B., AND WADDLE, V. FDM—Functional data model. In *Proc. 5th Int. Conf. Very Large Data Bases* (Rio de Janeiro, Oct. 3–5), ACM, New York, 1979, pp. 194–208.
7. KATZ, R.H. Database design and translation for multiple data models. Ph.D. dissertation, Dep. of Electrical Engineering and Computer Science, Univ. of California at Berkeley, June 1980.

8. KATZ, R.H. Compilation of relational queries into CODASYL DML. In *Proc. 2nd Int. Conf. on Databases—Improving Usability and Responsiveness* (Jerusalem, Israel, June), P. Scheuermann (Ed.), Academic Press, New York, 1982.
9. KATZ, R.H., AND WONG, E. An access path model for physical database design. In *Proc. ACM—SIGMOD Int. Conf. Management of Data* (Santa Monica, Calif., May 14-16), ACM, New York, 1980, pp. 22-29.
10. KATZ, R.H., AND WONG, E. Decompiling CODASYL DML into relational queries. *ACM Trans. Database Syst. 7*, 1 (March 1982), 1-23.
11. SCHKOLNICK, M. A survey of physical database design methodology and techniques. In *Proc. 4th Int. Conf. Very Large Data Bases* (West Berlin, Sept. 13-15), ACM, New York, 1978, pp. 474-487.
12. SENKO, M.E., ET AL. Data structures and accessing in data-base systems. *IBM Syst. J. 12*, 1 (1973).
13. SHIPMAN, D. The functional data model and the data language DAPLEX. *ACM Trans. Database Syst. 6*, 1 (March 1981), 140-173.
14. SIBLEY, E.H., AND KERSHBERG, L. Data architecture and data model considerations. In *Proc. Nat. Computer Conf., 1977.* Vol. 46, AFIPS Press, Arlington, Va., 1977, pp. 85-96.
15. TAHA, H.A. *Integer Programming—Theory, Applications, and Computations.* Academic Press, New York, 1975.
16. WONG, E., AND KATZ, R.H. Logical design and schema conversion for relational and DBTG databases. In *Entity-Relationship Approach to Systems Analysis and Design*, P. P. Chen (Ed.), Elsevier North-Holland, New York, 1980.