DATABASE DESIGN AND TRANSLATION FOR MULTIPLE DATA MODELS

by

Randy Howard Katz

Memorandum No. UCB/ERL M80/24

9 June 1980

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

CHAPTER 1

INTRODUCTION

1. Introduction

1.1. Statement of the Problem

Database systems available today typically support a single model of data, either tabular (relational) [CHAM76], graph structured (network) [TAYL76], or tree structured (hierarchical) [TSIC76a]. Arguments have raged about which model is "best," yet there is a growing realization that no model is best under all conditions, and that systems built in the future will have to support more than one model of data organization. These are called heterogeneous database management systems. This thesis is concerned with techniques for constructing heterogeneous data management systems from existing homogeneous databases and systems.

Consider a hypothetical database system that supports both relational and network databases. Such a system must provide several services. First, the semantic content of the database, i.e. its real world meaning, should be specified independently of the models supported by the system. It should provide tools for both logical and physical database design that are largely independent of the

underlying models and systems. A single design methodology can be used for a variety of different data models. Second, the system should provide facilities for migration of data from one underlying model to another. This makes it possible for data to remain independent of particular data models. Finally, the system should provide facilities for program conversion. An existing application should not become obsolete because a database has been rerepresented in terms of a new data model. Methods for implementing these facilities are explored in this thesis.

1.2. Motivation

There are two primary motivations for our work. The first is that distributed database research will stimulate interest in the support for heterogeneous data models. The second is that no existing model is appropriate for the design and translation problems presented by such a system.

As the trend towards distributed database systems continues to gain in momentum, the tasks of database design and translation in a heterogeneous environment are becoming pivotal. A distributed database system is built on top of existing systems available at the local sites of a computer network. It provides the ability for an applications program to access databases stored at another site. In addition, similar problems exist in a single

presented in this dissertation can be used as the basis for an automated database design tool. The techniques for data and program translation make it possible to migrate data between different data models and systems without the need to reprogram user applications. These methodologies and techniques are needed to implement a heterogeneous system on top of existing systems.

ACKNOWLEDGEMENTS

It is my pleasure to express my gratitude for the warm encouragement and support of my thesis advisor, Professor Eugene Wong. I also wish to thank the members of my thesis committee, Professors Lawrence Rowe and Andrew Shogan, for their prompt reading of this thesis and their insightful comments.

I thank my colleagues and friends on the INGRES Project who read this (long) dissertation even though they didn't have to. In particular, Robert McCord, Professor Michael Stonebraker, and Julio Kaplan have provided many comments which have resulted in a more readable final product.

I wish to thank Professors Rowe, Stonebraker, and Wong for creating the supportive environment at Berkeley in which to pursue database related research. Undoubtedly, Berkeley has become the premier institution in which to conduct such research.

Finally, I acknowledge the generous financial support of the Army Research Office (Contract DAAG29-76-G-0245) and the I.B.M. Corporation for their predoctoral fellowship.

TABLE OF CONTENTS

Database Design and Translation for Multiple Data Models

Randy Howard Katz

ABSTRACT

Database experts have debated over which model of data organization is best under various criteria. We believe that no model is clearly best and that future systems must support more than one data model. This dissertation explores methods and techniques for constructing a heterogeneous database management system from existing database systems. Logical and physical database design is studied in an environment which supports multiple data models and systems. Techniques are formulated to translate a database to an equivalent organization under a different data model and to convert the query portions of programs.

A data model is developed to capture the semantic interrelationships supported by a database. The purpose of a logical design is to derive a schema which is well-behaved under the update operations of a particular model. A database specification in the semantic data model is mapped into the constructs of a target model, while preserving design goals related to the desirable behavior of a schema under update.

Research sponsored by Army Research Office, Contract DAAG29-76-G-0245, and an I.B.M. Predoctoral Fellowship.

The semantic data model is augmented with logical access paths to represent how semantic objects are interconnected. Physical storage structures are characterized by a small number of basic properties. The physical design method proceeds in two phases. Properties are first assigned to the logical access paths. Then, the logical access paths are implemented by choosing storage structures which support the assigned properties.

Data translation is accomplished by recognizing constructs within the source database that correspond to a semantic object, and then mapping these into an equivalent realization in the target model.

The logical access paths supported in the physical database direct the program translation. A sequence of low level operations are identified as a semantic access. A composed sequence of these accesses is mapped into a single high level query specification. The inverse translation is accomplished by mapping a high level specification into a sequence of low level operations which take advantage of the access paths efficiently supported by the database.

A semantic data model was used to integrate the processes of database design and translation in an environment which supports multiple data models. The methodology for logical and physical database design

site heterogeneous database management system constructed on top of existing data managers.

The underlying data models we have chosen to investigate are the Relational model of [CODD70] and the CODASYL/DBTG model of [CODA71, CODA73, CODA78]. The relational model has been the focus of much recent research in data management systems, while the CODASYL model has received interest as a potential national standard. Both models are likely to influence systems built in the near future. Unfortunately, neither model is effective in solving the problems of design and translation.

We briefly define each model in turn. The relational model consists of domains and relations. A domain is a set of values of similar type. Let D_1, D_2, ... , D_n be n ($n > 0$) domains, that are not necessarily distinct. The Cartesian product $D_1 \times D_2 \times \ldots \times D_n$ is the set of all n tuples $\langle t_1, t_2, \ldots, t_n \rangle$ such that $t_i \in D_i$ for all i. A relation defined on these domains is a subset of the Cartesian product. We may associate with each tuple component a distinct index, which is called its attribute. A relational database is a time-varying collection of data organized as a collection of tabular relations.

The CODASYL model consists of data items, records, and sets. A data item is the smallest unit of named data. A record type is a named collection of data items. There

may exist an arbitrary number of occurrences for each record type. A set is a named collection of 1 or more record types, in which one record type is distinguished as the owner and the others are the members. A set occurrence consists of one owner record and zero or more members. A CODASYL/DBTG database is a time-varying collection of data organized as record and set occurrences.

The underlying data models we have chosen do not provide appropriate data object types for modelling aspects of the real world. It is difficult, for example, to examine a relational schema and determine what real world objects and their interrelationships are being modelled. A method of semantic specification is needed which is independent of the underlying data models. The notion of semantic equivalence, i.e. whether two schemas in different data models represent the same facts about the real world, is crucial in achieving schema conversion. The semantic specification must be rich enough to make it possible to identify equivalent constructs in different data models. The proper form of the semantic specification is an important consideration for supporting heterogeneous data models.

An architecture for constructing a heterogeneous system on top of existing relational and CODASYL data managers is given in figure 1.1. The schema translation

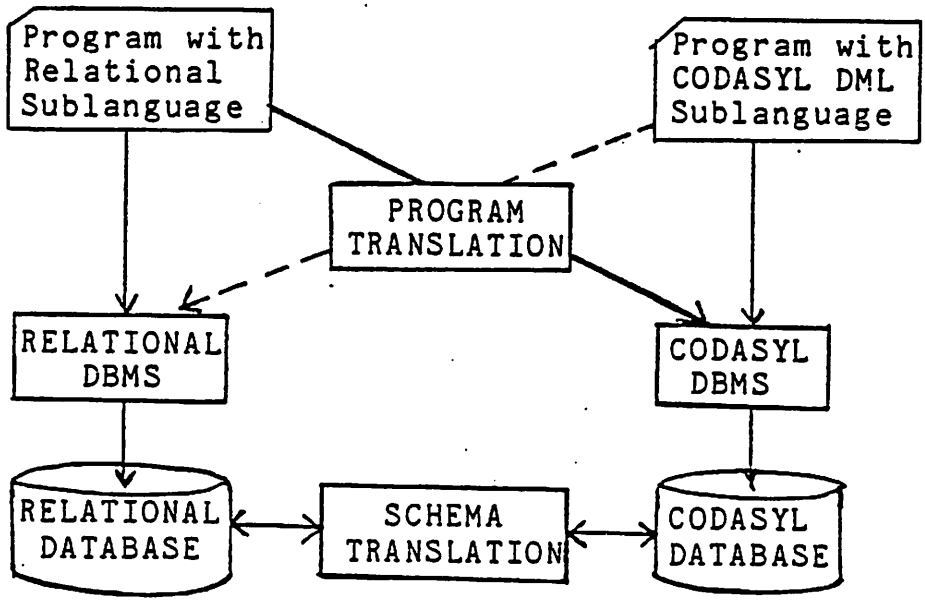


fig. 1.1 - Coexistence Architecture for a Heterogeneous DBMS

component supports conversion between representations of semantically equivalent databases organized under different data models. If a program is written to access data in a particular model, the program translation component makes it possible for it to access data organized under different data models. The program need not know how the data is actually stored. Multiple manipulation language interfaces are supported by program translation.

In the following, we briefly describe the design and translation problems that must be solved in a heterogeneous database system.

In the heterogeneous system proposed above, it is desirable to provide design tools that are highly independent of the specific models and systems that are the targets of our designs. Logical design is independent of specific system details because of its data model orientation. This property has been called "data independence." Nonetheless, a methodology for achieving logical designs that are model independent is desirable in a heterogeneous environment. Such a methodology would give precise rules for constructing a logical design with desirable properties for each model.

Physical database design, on the other hand, tends to be more concerned with the details of a specific system. In a heterogeneous environment, the task is difficult because of the requirement to design for the variety of storage structures supported by the underlying systems. Physical design has been partitioned into two tasks [CARD75]: implementation-oriented design, e.g. selection of which logical access paths to support, and implementation-dependent design, e.g. selection of implementation structures for collections of data. As much of the design process as possible should be implementation-oriented. For example, a design can be specified in terms of basic properties of storage structures without making a commitment to an actual implementation. The

implementation-dependent aspects can be handled by specialized tools keyed to a specific underlying system. Logical and physical database design should be supported by a single integrated tool.

Another motivation for heterogeneous databases is that they allow the user to migrate towards new data models and systems, while still supporting existing applications programs. New programs and databases can be developed while existing applications still function, whether or not the original database is represented in terms of the original data model. New applications should be able to access databases represented in the original model. Facilities should be provided to support several data manipulation languages simultaneously. Providing these facilities is difficult because the different data manipulation languages may be at different levels of procedurality, e.g. languages that support procedural, record at a time access and those that support set oriented access. Data and program conversion must be supported to facilitate this migration.

1.3. Previous Work

In this section, we review the previous research in database design and translation.

Logical design is the process by which real world objects and their interrelationships are represented in terms of a data model, independent of physical structure. This representation of the real world must be faithful. The behavior of the schema under the update operations of the model should reflect the semantics of the real world. If it does not, the schema is said to suffer from update anomalies.

For the relational model, the concept of normalization was introduced in [CODD71] in an attempt to derive relations which are free from update anomalies. The semantics of the relational model are insufficient in this regard, and must be augmented with functional dependencies [BERN76, CODD70, CODD71] and multivalued dependencies [ZANI76, FAGI77a]. These additional semantic structures make it possible to define normal forms for relations which indicate the kinds of update anomalies present in the relational schema (see section 3.3.2 for a discussion of normal form theory).

Normal form theory has led to two techniques for designing relational databases: decomposition and synthesis. Decomposition [CODD71], which has been generalized in [FAGI77b], starts with a collection of relations in which all attribute values are atomic and a specification of the multivalued dependencies among these attri-

butes. Each relation is decomposed into smaller relations until the resulting relations are in a form which is well-behaved under update operations. Unfortunately, multivalued dependencies represent complex concepts. These must be defined within the context of a particular relation, and require considerable effort on the part of the database designer to specify completely. Further, the approach is not constructive, i.e. primitive semantic objects are not combined to design the schema.

Synthesis [BERN76] begins with a set of attributes and a list of functional dependencies defined among them. A relational schema is derived by grouping the attributes together to form relations. The shortcomings of the approach are detailed in [FAGI77b], but can be summarized here. For synthesis to succeed, an artificial assumption must be made about how attributes can be related, i.e. only a single functional dependency can exist between any two attributes. In addition, synthesized relations exhibit update anomalies not found in decomposed relations, and the latter can not be constructed using the synthesis procedure.

A synthesis procedure based on functional dependencies and "static set dependencies" is proposed in [MIJA76]. An entity oriented data model is introduced and algorithms are given to design logical schemas for this

model. The authors do not show how their procedures can be applied to relational databases, although this does not appear difficult. The approach suffers from the same shortcomings as [BERN76].

Another approach to logical design is called abstraction [SMIT77a, SMIT77b]. "Aggregation" turns a relationship between objects into an aggregate object, while "generalization" turns a collection of objects of similar type into a generic object. They discuss how to map these abstract objects into relational and DBTG schemas. Again, the design process is complicated by the introduction of more complex semantic objects.

Methodologies specifically oriented towards CODASYL logical design have received less interest. [GERR75] describes a logical design aid that creates a CODASYL schema to support a specified set of anticipated queries. The problems of constructing a design that minimizes update anomalies is not addressed. [ADIB76] proposes a methodology for CODASYL logical design that maps a CODASYL schema into a relational schema, converts the relational schema into 3NF, and then maps the resulting schema back into CODASYL. The problem here is how to specify the original CODASYL schema in the first place. Another proposal is to convert the problem of designing a database into a mathematical programming problem to achieve an "optimal"

solution [MITO75, BERE77]. These models are limited in the number of aspects of logical design they can handle and can become expensive to use for the large problems encountered in real design efforts.

Another method, similar to the one explored in this thesis, is the structural transformation method of [CHEN76]. In this approach, information is specified in terms of a design data model which is transformed into either a relational or DBTG schema. These transformation rules have been chosen in an ad-hoc manner. Our contribution is to treat the mapping approach more formally by introducing goals of design and specifying mapping rules which preserve those goals.

Physical database design maps logical database structures into the access methods and storage structures of a particular database system. Surveys of physical design techniques can be found in [CHEN77] and [SCHK78].

[SCHK78] divides physical database design into four problem areas: file structuring, access path selection, record segmentation and allocation, and reorganization. File structuring is the problem of choosing an implementation structure for a single collection of records that satisfies a specified set of user requirements. Access path selection is concerned with which of a variety of access paths are chosen for efficient support. Record

segmentation techniques are ways to partition records to provide better access to the more heavily used "segments." Reorganization algorithms determine when the storage structures of a physical database should be reevaluated, as records are inserted and deleted which may cause the database's performance to deteriorate over time. This thesis deals only with file structuring and access path selection.

Two approaches have been proposed for selecting the implementation structure of a file. The first evaluates a preselected library of candidate implementations. It accepts input parameters which describe: (1) the contents of the records, (2) how the records are accessed and with what frequency, (3) how the secondary storage is organized and (4) how it performs under access requests. A variety of different storage organizations are evaluated, and the one that achieves minimum cost, under a preselected cost formula, is chosen. Simulation [CARD73, CARD75, SILE76] or analytic [GOTL74, DUHN78, TEOR78] techniques can be used to evaluate the costs of alternative designs. This method is hampered by restricting the selection to a predefined set of possible structures, thus making the methodology dependent on the details of a specific system. The second approach is to formulate general models of file structures which are parameterized to encompass a range of

possible implementations [HSIA70, SEVE72, YAO 77]. By varying the parameters, the designer can evaluate the relative costs of his designs.

As reported in [SCHK78], little work has been done to date on integrating logical and physical database design. This is crucial when choosing access paths for a database. Most approaches assume that the logical design has already selected certain paths to support. One such approach is [GAMB77], which describes a decision support system for physical database design. It accepts as input a logical access path description of the database and information about how frequently these access paths are used. The system determines location modes, record clustering, and set implementations for the CODASYL model. Storage and access costs are determined by a complex analytical model and are reported to the designer. Certain parameters can be varied by the designer to examine alternative designs. A similar approach is taken in [BUBE76], except that only an update-oriented and a retrieval-oriented design are derived. Heuristic design techniques are used in the latter to direct the design process. Our contribution is to use an integrated framework for logical and physical design and to propose a novel, implementation-oriented methodology for physical design. Storage structure properties are assigned to logical assign paths, and these

are used to direct a system specific expert to choose storage structures.

Next we review the recent research in schema conversion. Schema conversion is the process by which data represented in one model is mapped into an organization suitable for another model. [SHOS75] partitions the data translation problem into three levels. The logical level is concerned with mappings between groupings of data fields. The storage level involves mappings between different file organizations and access paths. The physical level deals with mappings between record placement and blocking strategies. Schema conversion is concerned exclusively with data mappings at the logical level.

A common approach to data translation is to provide the user with facilities to support the conversion process, while leaving it to him to explicitly specify the data mapping. [SHU 77] describes a system which provides high level languages for both defining data and for programming the restructuring transformations.

A second approach is based on performing the conversion by mapping a schema into a "semantically equivalent" schema for a different data model. [ZANI79a, ZANI79b] describe algorithms for mapping a CODASYL schema into an equivalent relational schema. He does not address the problem of mapping in the other direction, partially

because he is only interested in providing multiple model views on top of a CODASYL schema. The semantics-based approach is further examined in this thesis. Algorithms are formulated to map between target schema realizations of the same semantic objects, identified by the semantic specification used for the logical design.

Finally, we review the work in program conversion. Program conversion is concerned with the changes that must be made within an application program whenever a change is made in the representation of the database which it accesses. These changes can occur because a database is either converted from one database system to another or restructured within a single database system. This thesis focuses on the changes in applications programs that are caused by converting between database systems that support a different level of procedurality in the data sub-language.

[HOUS77] proposes a system wherein programs are converted by "decompiling" a source program into an abstract specification. The specification can be reexpressed in terms of the target data model and tailored to the target database. It can then be recompiled into the host and manipulation languages of the target system. However, it is difficult to formulate ways to specify the reexpression of the program in terms of the target data. The paper sug-

gests that a translation analyst be responsible for this.

Another approach is being investigated at the University of Florida [SU 76, SU 77, NATI78, SU 78a, SU 78b]. A semantic data model is proposed to characterize the way in which data is used by an application program. Language templates [SCHI77] are matched against the program to recognize instruction sequences which correspond to particular types of semantic accesses. The matching associates with each instruction sequence an access path graph which is a model dependent representation of a semantic access pattern. A query graph is constructed from a sequence of these graphs. The query graph can then be decomposed into access path graphs for the target system. The program is recompiled by finding the instruction sequences in the target host and manipulation languages that correspond to the access patterns of the graphs. Unfortunately, the authors do not articulate actual algorithms for realizing the conversion process outlined above. We agree that a semantics based approach is the most appropriate and is the approach taken in this thesis. Our contribution is to develop the actual algorithms to map sequences of procedural operations into a non-procedural query specification, and to integrate the conversion process with information gleaned from the database's physical design specification.

The mapping of non-procedural queries into a sequence of procedural operations is a classical problem in query processing for relational systems. [SELI79] describes a technique for enumerating all the likely processing strategies and choosing the one with the expected minimum cost. However, many possible strategies must be evaluated before the best one can be found. [WONG76, YOUS79] describe a collection of heuristic query processing tactics for reducing a query step by step into low level operations. Their approach does not take advantage of information about the physical structure of the database at the early stages. Our point of departure is to make use of information specified in the logical and physical design to aid in the selection of a processing strategy.

1.4. Contributions and Outline of Thesis

Chapters 1 and 2 cover introductory material. This chapter has provided an introduction to the need for heterogeneous database management systems and the facilities that such a system must support, namely database design and translation for multiple data models. Previous research in the areas of logical and physical database design, database translation, and program conversion was discussed. The overall goal of this thesis is to propose methodologies and techniques for supporting these facilities in a heterogeneous database environment.

In chapter 2, three interrelated data models are proposed as our framework for design and translation. The design model forms the basis of our approach to logical design and schema conversion. It is compared with other proposals for semantic data models. The integrity model is the basis of our approach to physical design, and introduces the concept of logical access paths. The access path model supports our approach to program translation. It consists of the physical access paths used by a program when navigating through its data.

Chapters 3 and 4 describe our methodology for design. Chapter 3 introduces the concept of design goals and shows how design model schemas can be mapped into relational or CODASYL schemas while preserving the design goals, which represent desirable properties exhibited by the designed schemas. The schemas that result from an application of our rules are characterized. The relational schemas are shown to be in fourth normal form, i.e. the relations are well-behaved under update operations. The usefulness of certain constructs in the CODASYL model are made clear by characterizing them in terms of the concepts of the design model.

Chapter 4 introduces our characterization of storage structure and proposes an algorithm for implementation-oriented physical design. The methodology is based on

assigning storage structure properties to access paths in the schema. Implementation algorithms are given which map a physical design specification into the storage structures of two specific database systems.

Chapters 5 and 6 describe our techniques for translation. Chapter 5 describes schema conversion. The mapping rules used in logical design can also be used to formulate mapping rules between different target models. The rules for mapping from CODASYL to relational and vice versa are derived. Problems associated with semantic restructuring of the database are also addressed.

Chapter 6 describes program conversions. Decompilation is the process by which a sequence of procedural operations are mapped into a single set oriented query specification. Algorithms are developed for analyzing a program written for a CODASYL database, and converting it to a program with a relational interface. Compilation is the process of mapping a set oriented query into a sequence of procedural, access path oriented operations. An optimization procedure is developed to convert a relational calculus query into an internal form which describes how the query manipulates the access path schema. An algorithm is given for compiling this internal form into CODASYL data manipulation operations.

Chapter 7 describes our conclusions and indicates areas for future research.

CHAPTER 2

DESIGN MODELS

2. Design Models

2.1. Introduction

In this chapter we define a design data model as the focus for our approach to logical design and schema conversion. The model is based on a variation of the Entity-Relationship Model [CHEN76] and has been influenced by the semantic data model of [SCHM75]. Because we are interested in dealing with databases supported by the currently popular models, in particular the relational and CODASYL approaches, the design model has been formulated to capture the kinds of structural constraints supported by these models, yet remains independent of them. It forms a semantic bridge between the underlying models. An alternative view of the design model, which emphasizes logical access paths derived from constraints on the interrelationships of semantic objects, is the integrity model. It closely resembles the functional data models of [SHIP80] and [BUNE79]. It is the basis of our approach to physical database design. Lastly, the decision of which access paths are to be supported in the physical realization of a database is reflected in the access path model. This

information is crucial for understanding the semantics of the procedural operations to accomplish decompilation, and for representing the available access paths during compilation.

2.2. A Design Data Model

2.2.1. Definitions

The term data model is used in a generic sense to mean a collection of data object types, while a schema is a specific choice of data objects to represent a database. For example, The relational model consists of the data object types domains and relations; a CODASYL data model consists of data items, records, and sets. A relational or CODASYL schema would consist of a specific collection of such objects. The design model is a semantic data model used to specify a design schema.

The design model is composed of entity sets, relationships, and properties. For each instant of time t, let $E_1(t)$, $E_2(t)$, ... , $E_n(t)$ be n distinct sets, which are called entity sets. An entity set, usually referenced by name, is a one parameter family of sets which change as members are inserted and deleted.

A property of an entity set E(t) is a one parameter family of functions f_t, which at each time t maps E(t) into some set V of values. A property is a total function:

for every e in E(t), $f_t(e)$ is single-valued (functional), and at each time t, f_t is defined over each member of E(t) (total).

An identifier is a distinguished property of an entity set which maps 1-to-1 into a value set. The identifier is actually a surrogate for the entity [CODD79], and as such, uniquely represents each entity with a value that can not change over time. If an existing property can not be adapted for use as an identifier, an artificial one must be introduced.

Explicit provisions for value set definitions have been omitted in our model. A domain definition subsystem such as that proposed in [MCLE76] could be included. A simpler approach is to use primitive data types, e.g., integer, char(10), etc. A given value set may appear in the range of more than one property.

A relationship R_t among the entity sets $E_1(t)$, $E_2(t)$, ... , $E_n(t)$ is a time-varying relation; i.e., R_t is a subset of the Cartesian product $E_1(t)$ X $E_2(t)$ X ... X $E_n(t)$ at time t. A relationship may optionally have a property defined on it. It is assumed that relationships specified in a design schema are both independent and indecomposable; i.e., no relationship is derivable from other relationships and no relationship can be decomposed into subrelationships of smaller degree and then recombined to

form the original relationship without loss of information (no relationship is equal to the join of two of its projections into subrelationships of smaller degree for all time).

A binary relationship R_t on entity sets $E_1(t)$ and $E_2(t)$ is single-valued in $E_1(t)$ if each entity of E_1 occurs in at most one instance of R_t. Intuitively, R_t is a partial function with domain $E_1(t)$ and range $E_2(t)$. If each entity in $E_1(t)$ occurs in exactly one instance of R_t, then R_t is called an association. Intuitively, R_t is a total function which maps each element of $E_1(t)$ into an element of $E_2(t)$. Neither an association nor a single-valued relationship may have a property defined on it. A property of an association is necessarily a property of the domain entity set; thus, the concept is superfluous. The models under investigation do not provide constructs for automatically supporting a single-valued relationship when it has a property defined on it (see chapter 3).

The design model distinguishes among the following semantic objects:

(a) entity set
(b) properties of entity sets
(c) associations
(d) single-valued relationships
(e) relationships
(f) properties of relationships
(g) value sets

A discussion of the allowable design model operations will

be deferred until Chapter 3.

We close this section with two examples of schema design using the design model. Consider the schema which represents a simple employee database: employees work in departments which have managers, employees are qualified to hold certain jobs of which only one is the job they are currently assigned, and job positions are allocated to departments.

Which real world objects do we wish to describe in our database? These are Employees, Departments, and Jobs, and they constitute the entity sets. Next we determine how these objects are interrelated. An employee Works-in a department, an employee is Assigned a current job, an employee is Qualified for several jobs, a department has a Manager employee, and a department is Allocated job positions. Works-in, Assigned, Qualified, Manager, and Allocated constitute the relationships.

Which of these relationships are functional? If the semantics indicate that an employee can not be assigned more than one job, or work in more than one department, or that a department can not have more than one manager, then these constitute the single-valued relationships. Qualified is a general relationship because an employee can be qualified for multiple jobs, and many employees can be qualified for a particular job. Allocated is a general

relationship for similar reasons.

Which of these functional relationships are total? If an employee must work in a department at all times, and he must be assigned to some job at all times, then Works-in and Assigned are associations. This imposes an existence dependency on employees: no employee can exist unless he works within a department or is assigned a job. Thus, to maintain the totality of the function, the deletion of a department or job must cause all the associated employees to be deleted as well.

The only decisions left are what properties are needed to describe the entity sets and general relationships. Employees are described by their names and year of birth. An employee number is introduced as a unique identifier. Departments have the properties of department name and location. A department number is introduced to uniquely represent a department. Jobs are described by a title and an associated salary. A job-id is included as the identifier. Of the general relationships, only Allocation has a property, i.e., the number of job positions of a particular job allocated to a given department. These constitute the properties. In summary, the design schema for the employee database is shown in figure 2.1.

In the second example, we shall discuss a design schema specification for a university database. Students

entity sets	properties
Emp	eno,ename,birthyr
Dept	dno,dname,location
Job	jid,title,salary

associations
Works-in(Emp,Dept)
Assignment(Emp,Job)

relationships	type	properties
Mgr(dept,emp)	single-valued	---
Qualified(emp,job)	general	---
Allocation(dept,job)	general	number

fig. 2.1 - Employee Database

are enrolled in courses taught by professors who are faculty of some departments. Courses have prerequisites, and departments have chairmen. The objects of interest are Students, Courses, Professors, and Departments. These may be described by properties. Students are identified by their registration number and described by their name and class year. Professors are identified by their social security number and described by their name and professorial rank. A course is identifed by a course number and is described by a title. Departments have names and locations, and a department number must be introducted as a unique identifier.

Students must Major in departments and departments must have a Chairman. These are represented as associations if we assume that every student must have some major

(including the major "undeclared"), and a department can not exist without a chairman. If professors can be on the faculty of more than one department, then Faculty is modelled as a general relationship. Similarly, a course can be a Prerequisite of several courses and can itself have several prerequisites.

Now consider the relationship Enrollment among Students and Courses. It is described by the property Grade. Because a given course can be taught by more than one professor, the relationship should be defined over professor as well. The relationship models reality if the database recorded the enrollment of a student for a particular semester. If Enrollment is to represent the transcript of a student throughout his college career, then Enrollment as formulated above is deficient, because it rules out the possibility that a student repeats a course from the same professor. To remedy this situation, an entity set Times is introduced to represent the time at which courses may be given (e.g. "Fall/1979/MWF10-11") and Enrollment is defined over Professors, Courses, Times, and Students. Times is described by properties for the term, year, and time of week, and a timeid is included as a unique identifier. The complete schema is shown in figure 2.2.

entity sets	properties
Students	regno,sname,classyr
Professors	ssno,pname,rank
Courses	cno,title
Dept	dno,dname,location
Times	timeid,term,year,time

associations
Major(Student,Dept)
Chairman(Dept,Professor)

relationships	type	properties
Faculty(Professor,Dept)	general	---
Prereq(Course,Course)	general	---
Enrollment(Course,Professor, Times,Student)	general	grade

fig. 2.2 - University Database

2.2.2. Comparisons with Other Models

Many data models have been proposed which model the real world in terms of objects of interest (entities) and how they are interrelated (relationships). In this section, our design model is compared with these other models. Our purpose is not to argue that the design model is "better," but rather to characterize it with respect to the other proposals.

The work on a "conceptual schema" to model the real world, by the ANSI/X3/SPARC committee on database architecture [TSIC77], has led to an increased interest in semantic data models. [KERS76] compares and classifies twenty-three different data models, while [NIJS76] con-

tains sixteen proposals for candidate models to represent the conceptual schema. We restrict the following discussion primarily to entity oriented models.

The basic object types of the design model consist of entity sets, relationships, properties, value sets, associations, and single-valued relationships. Several models have been based on similar concepts, although they use different terminology. Several limit all relationships to be binary [DEHE77, BRAC76]. Others allow n-ary relationships, but require them to be irreducible or indecomposible [HALL76]. We have adopted this approach. The model which most closely resembles the design model, in terms of the object types supported, is the Entity-Relationship Model of [CHEN76]. His model, however, does not require that entities be members of distinct entity sets. [MOUL76] argues that in order to provide unambiguous naming for entities, it is necessary to have entity sets which are disjoint. The entity sets of the design model are non-overlapping.

We have extended the Entity-Relationship Model by distinguishing functional binary relationships which may be total (associations) or partial (single-valued relationships). Few models make this distinction, with the exception of the functional data model of [SIBL77] and the CODASYL/DBTG model [NIJS75]. In the latter, the CODASYL

set construct can be used to model either total or partial functions between the member and owner record types.

We use associations to model the concept of existence dependency, i.e. that an entity can only exist so long as it is associated with some other entity. In [DEHE74], entities can be declared as "obligatory" members of a relationship. In [SCHM75], certain objects can only be used to characterize other objects, and must be deleted if the object they describe is deleted. [CHEN76] introduces the concept of "weak entity" to describe entities which are only of interest because they are related to some other entity, e.g. children of employee.

Because the concept of an entity is abstract, how does one determine whether an entity exists? Usually, an identifier is introduced to represent the entity within the data model. [HALL76, CODD79] require that an artificial identifier, i.e. a "surrogate", be introduced. Other models allow a property which is 1-to-1 to be selected as an identifier, or require an artificial identifier to be introduced if no such property exists [SENK73, SCHM75, BENC76, CHEN76]. We have adopted the latter approach.

In summary, the design model is a data model based on the concepts of disjoint entity sets and n-ary relationships. The model allows the functionality of binary relationships to be specified. It also provides constructs to

model existence dependencies. Identifiers are used to unambiguously represent entities.

2.3. Integrity Model

2.3.1. Definitions

The integrity model is so named because it illuminates the structural integrity constraints that must be enforced for the interrelationships of the design model. Certain types of interrelationships have been defined in terms of total functions (properties, associations) and partial functions (single-valued relationships). The general relationship can be represented in terms of functions as well. For each entity set that participates in a relationship, we can define a total function that maps the relationship instance into the particular entity that participates in that instance. These functions are called relationship associations. From the standpoint of the integrity schema, the relationship object is superflous. It can be modelled by an entity set with associations that relate it to the entity sets over which the relationship is defined. However for semantic reasons, it is still desirable to distinguish between entity sets and relationships.

The integrity model consists of objects and functions between objects. We distinguish between value objects and

non-value objects. Value objects can only appear in the range of a function. Functions can be either total or partial. There is a strong resemblance between the integrity model and the functional data models of [SHIP80] and [BUNE79]. However, those models do not distinguish between total and partial functions and have been proposed as an aid to data manipulation rather than database design.

The functional form of the interrelationships (i.e. 1:N) imposes a constraint which is simple to support and which can be supported within the data models popular today. The CODASYL set construct can be used to model 1:N relationships between owner and member records [NIJS75]. Hierarchies are natural representations of 1:N integrity constraints. Although not explicitly supported by the relational model, 1:N relationships can be supported by such underlying storage structures as links [TSIC75]. When the functional interrelationships are used as the basis of a manipulation language, they can be used as logical access paths between semantic objects (see section 2.3.3).

The examples of the previous section are reproduced in terms of the integrity model in figure 2.3.

Employee Database

Total Functions

```
eno: emp --> integer
ename: emp --> char(20)
birthyr: emp --> integer
dno: dept --> integer
dname: dept --> char(10)
location: dept --> char(10)
qual-emp: qual --> emp
qual-job: qual --> job
jid: job --> integer
title: job --> char(20)
salary: job --> integer
works-in: emp --> dept
assignment: emp --> job
alloc-dept: alloc --> dept
alloc-job: alloc --> job
number: alloc --> integer
```

Partial Functions

```
mgr: dept --> emp
```

University Database

Total Functions

```
regno: students --> char(10)
sname: students --> char(20)
classyr: students --> integer
cno: courses --> integer
title: courses --> char(30)
timeid: times --> integer
year: times --> integer
term: times --> char(3)
time: times --> char(10)
fac-prof: faculty --> prof
fac-dept: faculty --> dept
prereq-course1:
       prereq --> course
prereq-course2:
       prereq --> course
ssno: prof --> char(9)
pname: prof --> char(20)
rank: prof --> char(2)
dno: dept --> integer
dname: dept --> char(10)
location: dept --> char(10)
major: students --> dept
chairman: dept --> prof
enroll-course:
       enroll --> course
enroll-prof:
       enroll --> prof
enroll-times:
       enroll --> times
enroll-students:
       enroll --> students
```

Partial Functions

---

fig. 2.3 - Integrity Schemas for Sample Databases

2.3.2. A Graphical Representation

We introduce a graphical representation of the integrity schema. Let G = (V,E) be a directed graph with

set V of vertices and set E of edges. There is a vertex in V for each object in the integrity schema. For each function from $object_1$ into $object_2$, there is a directed edge from the vertex for $object_1$ to the vertex for $object_2$. Identifier property edges are represented with a double arrow.

The graphical representations of the example schemas are given in figures 2.4 and 2.5.

2.3.3. Integrity Model Manipulation Language

The functions of the integrity schema can be used as a basis for navigating among the objects of the schema.

Employee Database

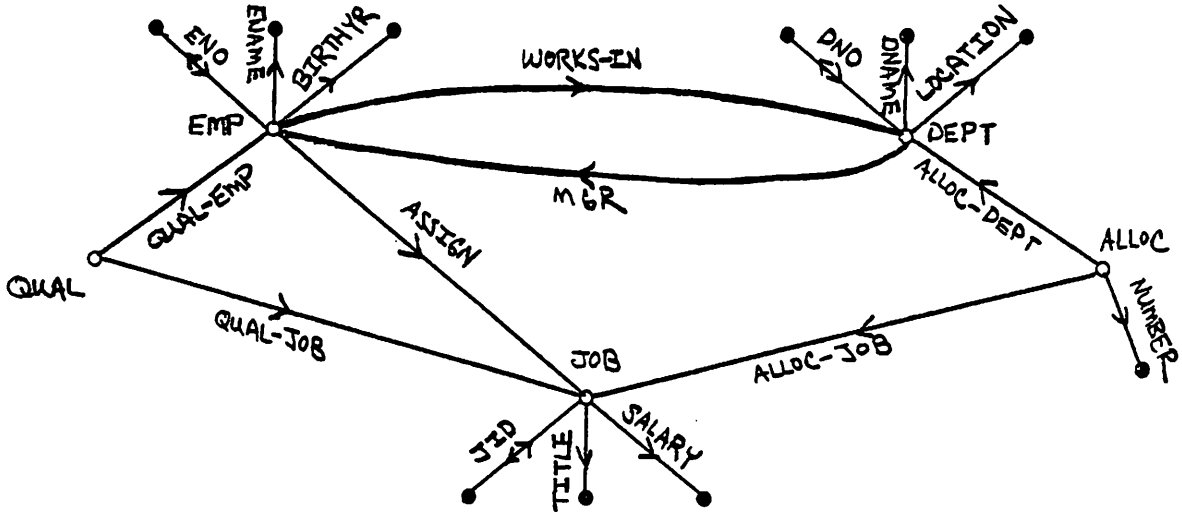


fig. 2.4 - Graphical Representation of Employee Database

University Database

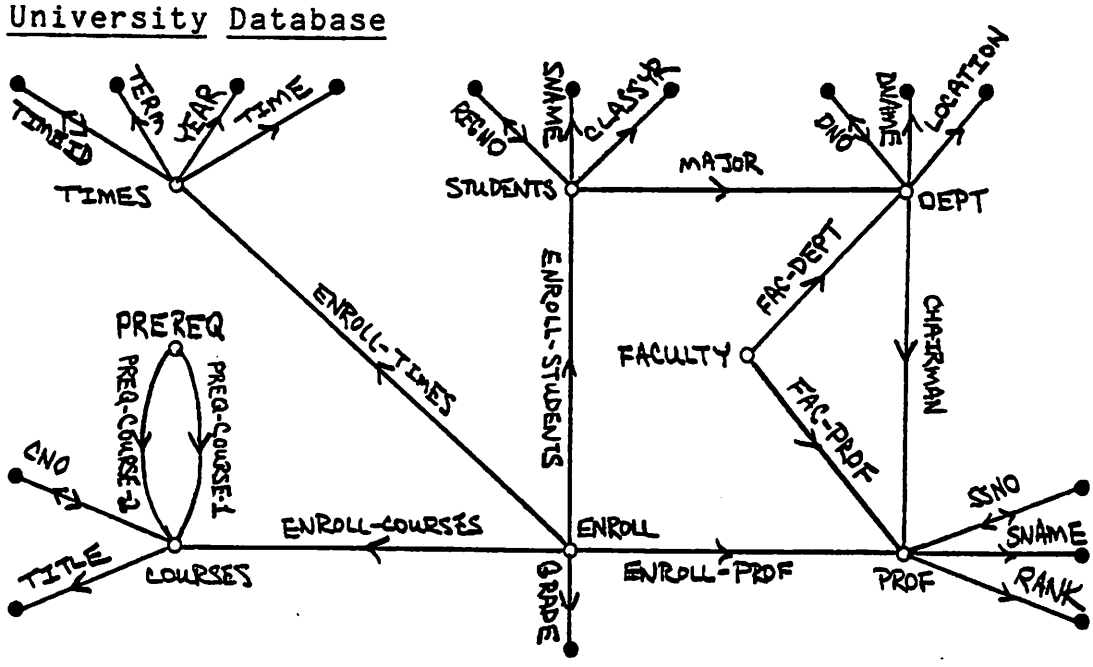


fig. 2.5 - Graphical Representation of University Database

This has led to the functional query language proposals of [BUNE79] and [SHIP80]. When integrity functions are used to access objects, we call them access mappings. The purpose of this manipulation language is not to provide a full function query language. Rather, it is used to specify a path from a starting set of objects to a destination set upon which data manipulation operations (e.g. retrieval, update, delete) will be performed. A similar defintion of "query" can be found in [NATI78].

An access mapping can be defined for either an integrity schema function or its inverse. We extend the

definition to allow an access mapping to map from a set into a set. This will facilitate the composition of access mappings. More explicitly, if f: A --> B is an integrity function defined over the integrity model objects A and B, then $F(\mathfrak{A}) = \mathfrak{B}$ where $\mathfrak{B} = \{f(a) \mid a \in \mathfrak{A} \text{ and } \mathfrak{A} \subseteq A\}$ and $F^{-1}(\mathfrak{B}) = \mathfrak{A}$ where $\mathfrak{A} = \{a \mid b = f(a) \text{ and } b \in \mathfrak{B} \text{ and } \mathfrak{B} \subseteq B\}$ are access mappings over f. Note that integrity schema functions are written in lower case, while access mappings are written in upper case.

It is useful to define several more kinds of access mappings. We extend the definition of inverse mapping to allow the subset of A to be accessed which is related to the instances of $\mathfrak{B}$ in some way other than equality, i.e. $<, \leq, \geq, >$. The definitions become:

$F_{>}^{-1}(\mathfrak{B})$ where $\mathfrak{A} = \{a \mid b > f(a) \text{ and } b \in \mathfrak{B} \text{ and } \mathfrak{B} \subseteq B\}$
$F_{<}^{-1}(\mathfrak{B})$ where $\mathfrak{A} = \{a \mid b < f(a) \text{ and } b \in \mathfrak{B} \text{ and } \mathfrak{B} \subseteq B\}$
$F_{\geq}^{-1}(\mathfrak{B})$ where $\mathfrak{A} = \{a \mid b \geq f(a) \text{ and } b \in \mathfrak{B} \text{ and } \mathfrak{B} \subseteq B\}$
$F_{\leq}^{-1}(\mathfrak{B})$ where $\mathfrak{A} = \{a \mid b \leq f(a) \text{ and } b \in \mathfrak{B} \text{ and } \mathfrak{B} \subseteq B\}$

In addition, for each non-value object of the integrity schema there is a constant inverse mapping whose value is the set of all instances of the object, e.g. EMP^{-1} is the set of employee entities.

Access mappings can be composed and intersected in order to form access expressions. If f: A --> B and g: B --> C, then f o g: A --> C and F o G can be defined as $\{c \mid c \in G(\{b\}) \text{ and } b \in F(\mathfrak{A}) \text{ and } \mathfrak{A} \subseteq A\}$. Similarly, if f: A

--> C and g: B --> C, then f $\sqcap$ g: A X B --> C and F(α) $\sqcap$ G(β) can be defined as {c | c $\in$ F(α) and c $\in$ G(β)}. An access expression is evaluated strictly from left to right. Parenthesis may be used to clarify the order of evaluation, but not to change it.

Although the manipulation language has no construct for boolean predicates, the objects accessed through an access expression can be qualified by intersecting them with inverse property mappings. For example, intersecting an access expression over employees with the inverse BIRTHYR^{-1}({1950}) will restrict the employees to those that were born before 1950.

A BNF description of the Integrity Model Manipulation Language is:

```
<access query> ::= <access mapping> |
                   <access mapping> o <access query>
                   <access mapping> ⊓ <access query>

<access mapping> ::= F | F^-1 |
                     F^-1({ <constant-list> }) |
                     ( <access query> )

<constant-list> ::= constant |
                    constant <constant-list>
```

Examples:

1) find the department that Smith works in.

   ENAME^{-1}({"SMITH"}) o WORKS-IN

   ENAME^{-1}({"SMITH"}) results in the set of employees

whose name is Smith. This is composed with WORKS-IN to access their associated departments.

2) find the names of departments with accountants who were born before 1950.

$((TITLE^{-1}(\{"ACCOUNTANT"\}) \circ ASSIGNED^{-1}) \cap BIRTHYR<^{-1}(\{1950\})) \circ WORKS\text{-}IN \circ DNAME$

$TITLE^{-1}(\{"ACCOUNTANT"\})$ accesses the job entity for accountant. Applying $ASSIGNED^{-1}$ to that gives us all the employees who are currently working as accountants. Intersecting that set with $BIRTHYR<^{-1}(\{1950\})$ restricts the accountants to those that were born before 1950. WORKS-IN accesses their departments, and DNAME accesses the names of those departments.

2.4. Access Path Model

2.4.1. Definitions

The access path model is concerned with those logical access paths that are supported by the storage structures of the underlying database system to promote the efficient access of data. The purpose of the access path model is to form a bridge between the logical interconnections of the integrity schema and the physical interconnections of the stored data, without committing the schema to particular implementation structures. Thus, the model is

implementation-oriented rather than implementation-dependent.

An access path schema consists of the objects of the integrity schema plus a selected subset of the possible access mappings described in the previous section. An access mapping is supported in the storage structure if the database system can efficiently perform the desired access; i.e., the time to access a set of desired objects is less via a supported access mapping than the time to exhaustively scan the entire set. If an access mapping is not supported, it is unsupported. Supported access mapping is our terminology for the usual notion of physical access path. The access mappings included in an access path schema must all be supported.

To illustrate these concepts, consider the employee database, and assume that the records which represent employees are hashed on the property ename. Thus, there is a fast path between ename values and the associated employees. $ENAME^{-1}$: char(20) --> emp would be supported by the database system and would be represented in the access path schema. $ENAME>^{-1}$: char(20) --> emp may or may not be supported, depending upon how the access method for hashing is implemented. Most implementations of hashing do not support efficient access by range. If an indexed sequential organization is used to support the first map-

ping, then it is likely that the second would also be supported. The access path schema specifies only those access mappings that are supported. It does not matter which implementation structure has been chosen to provide that support. Different choices may well result in different access path schemas.

The logical access paths of the integrity schema must be maintained whether or not those functions are supported. For example, WORKS-IN associates with each employee a single department. If WORKS-IN is not supported, we must still be able to access the associated department, albeit not as efficiently as if it were supported. This can be accomplished by making use of the identifier of the range entity set. An unsupported access mapping between employees and departments can be represented instead as an access mapping between employees and the identifier value set of department:

```
(supported)   WORKS-IN: emp --> dept
(unsupported) WORKS-IN: emp --> id value set if dept
```

Without support, it is no longer possible to navigate directly between employees and their departments. Instead, we must (1) navigate along the property to access the associated department identifier, (2) use an inverse identifier mapping to access the department, if such a mapping is supported, or, alternatively, exhaustively scan depart-

ment entities for the one with the appropriate identifier value.

The access path schema captures the effects of storage structure support without committing the schema to a particular implementation. Rather, a family of implementations are possible, in which each of the access mappings of the schema are supported. We are free to implement an access mapping with a variety of different storage structures. The logical access paths of the integrity schema have not been sacrificed. They are still supported by augmenting the schema with properties and identifiers as outlined above.

2.4.2. Access Path Manipulation Language

In this section, we define a data access language for access path schemas. The language is based on iterative constructs for enumerating elements of sets formed from intersections of access path mappings. Actually, the access path manipulation language is a reformulation of the Integrity Model Manipulation Language which emphasizes the procedural aspects of set enumerations.

The host language, e.g. COBOL, is extended by statements for (1) assigning access mapping results to program variables and (2) applying a block of statements to each element of an enumerated set:

```
<access assignment> ::= value-variable <-
                            <functional access mapping>

<access enumeration> ::= <for clause> range-variable
                          IN <set specification>
                          ST <mapping clause>
                          DO <statement list>

<for clause> ::= FOR EACH | FOR FIRST

<set specification> ::= <access mapping> |
                        <access mapping> ∩
                             <set specification> |

<mapping clause> ::= <clause> |
                     <clause> AND <mapping clause> |

<clause> ::= <access mapping> <relational-op> <value> |
             range-variable = range-variable

<access mapping> ::= F( <variable> ) |
                     F^-1( <variable> )

<variable> ::= value-variable | range-variable

<value> ::= value-variable | constant

<relational-op> ::= ≤ | < | = | > | ≥
```

We restrict the range variables that appear in the mapping clause to be the access enumeration range variable, except when comparing the current range variable to a previous one.

As an example, consider the query that requests the location of John Smith's dept. Assume we have the access path schema of the previous section, and that $WORKS\text{-}IN^{-1}$ is not included in the access path schema.

```
FOR FIRST e in EMP^-1() ST ENAME(e) = "John Smith" DO
    w <- WORKS-IN(e)
    FOR FIRST d in DEPT^-1 ST DNO(e) = w DO
        l <- LOCATION(d)
```

PRINT l

The program scans the employee objects until it finds one with employee name John Smith. W is set to the number of Smith's department, and departments are scanned to find the one with the matching dno. That department's location is found and printed.

If an inverse mapping $ENAME^{-1}$: char(20) --> EMP had been available, then the first line of the program would be:

FOR FIRST e in $ENAME^{-1}$("John Smith") DO ...

If $WORKS\text{-}IN^{-1}$ is supported, then the assignment to w can be omitted and the second FOR statement becomes:

FOR FIRST d in $WORKS\text{-}IN^{-1}$(e) DO ...

The advantage of the APML is that it represents procedural aspects of object enumerations without sacrificing the advantages of using access mappings, which are based on logical access paths.

2.5. Conclusions

In this chapter we have laid the groundwork for the rest of this thesis. A design model has been proposed to facilitate logical design and schema conversion for heterogeneous data models. It was shown how an integrity model can be defined to represent the constraints on

object interrelationships and to introduce logical access paths. The integrity schema will be the input to our methodology for physical database design. Finally, an access path model was proposed to aid the process of program translation. The model consists of only those logical access paths which are efficiently supported by the underlying database. Manipulation languages for the latter two models were defined.

CHAPTER 3

LOGICAL DESIGN

3. Logical Design

3.1. Introduction

In this chapter, we address the problem of how to design a logical schema for a specific data model. Our methodology is to characterize the objects of the real world by describing them in terms of the object types of the design model. The design schema description is then mapped into the object types of the underlying model. Design goals are introduced to help formalize this mapping. These specify the behavior expected of designed schemas under the update operations of the particular model. The mapping rules must preserve the design goals.

We present mapping rules for mapping a design schema into both relational and CODASYL logical schemas. The schemas derived from the mapping rules are also characterized.

3.2. Design Goals

Logical database design is the process by which information about the real world is transformed into objects supported by a particular data model, without

regard to how that information is actually encoded in storage. The primary objectives of logical design are (1) to adequately capture the semantics of the information while (2) divorcing the specification from physical implementation considerations. The data models available today concentrate more on the latter objective than the former. The relational model is a case in point. Although it does a good job of isolating the schema from implementation considerations, it is a non-trivial process to assign a meaning to a given schema. In this section, goals for logical design are formulated to achieve desirable semantics in the designed schema.

An important aspect of logical design is to minimize the unexpected side effects of applying operations to the logical structure of a database. For example, the process of relational normalization has been formulated to remove anomalies from updates to relational schemas. Update anomalies take one of two forms. Atomic operations are <u>fragmented</u> when a semantically atomic operation, e.g. deleting an entity, can not be performed atomically within the target schema. Update side effects are <u>uncontrolled</u> when an update operation simultaneously affects more than one independent object within the schema, e.g. deleting an entity inadvertently causes entities related to it via general relationships to be deleted as well. Some side

effects are controlled. For example, all relationship instances associated with a deleted entity should also be deleted. The design model has been formulated in part to clarify the issue of update anomalies and to control them.

For the design model, the following can be defined as atomic update operations:

(1) inserting or deleting an entity

(2) inserting or deleting a relationship instance

(3) modifying the range value of a function with an entity or relationship instance in the domain.

Note that it is not semantically meaningful to alter an entity or relationship instance. To record a change of information about such an object, it is necessary to modify one of the functions, i.e. properties or associations, that describe that object.

The application of an atomic update operation to a design schema may leave the database in an inconsistent state. For example, consider the employee database. If the Toy department is deleted, then all the employees who work in that department are no longer associated with an existing department. For the maintenance of semantic consistency, additional update operations must be performed. These induced updates are called side effects.

associations must be disallowed. The associations (including the relationship associations) impose a partial ordering on the entity sets and relationships for the purposes of insertion, i.e. X < Y if X must be inserted before Y. The insertion order for the employee database, represented as a topological sort, is shown in figure 3.1. Jobs and Departments must be inserted before Employee or Allocation instances. Jobs and Employees must be inserted before Qualified instances. In the university database (see figure 3.2), Professors must be inserted before Departments, Faculty instances, or Enrollments; Departments before Students or Faculty instances; Students before Enrollments; and Courses before Enrollments or Prerequisite instances.

Our approach to logical database design is to (1) specify a schema in terms of the design model and in so doing, completely describe the requirements for the atomi-

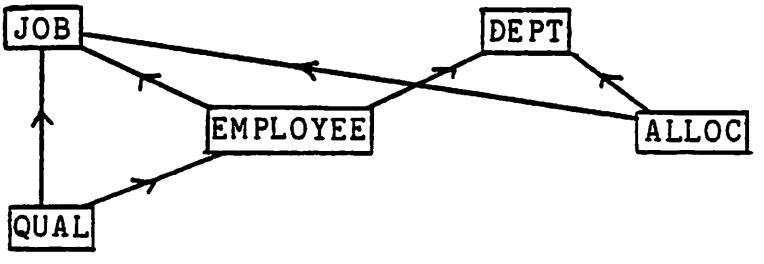


fig. 3.1 - Insertion Order DAG for Employee Database

The advantage of the design model is that the side effects are obvious and limited. The constructs of the design model limit the kinds of updates that can cause side effects to those that insert or delete an object in the range of a total function, i.e. associations and relationship associations:

(1) Deleting an entity e causes:

    (a) the deletion of an instance of a relationship in which e participates

    (b) the deletion of any entity in the domain of an association with e in the range

(2) Inserting an entity e (or relationship instance r) requires any entity in the range of an association (or relationship association), with e (or r) in the domain, to already exist.

Case (1b) can cause updates to propagate further through the schema, because the deletion of a single entity may spawn additional deletion updates to associated entities. Note that this is precisely the situation modelled by the association. If propagated deletions are not desired, i.e. the domain object can exist independently of the range, then the relationship should be specified as single-valued.

The insertion side effect (2) constrains the order in which entities may be inserted. In particular, a cycle of

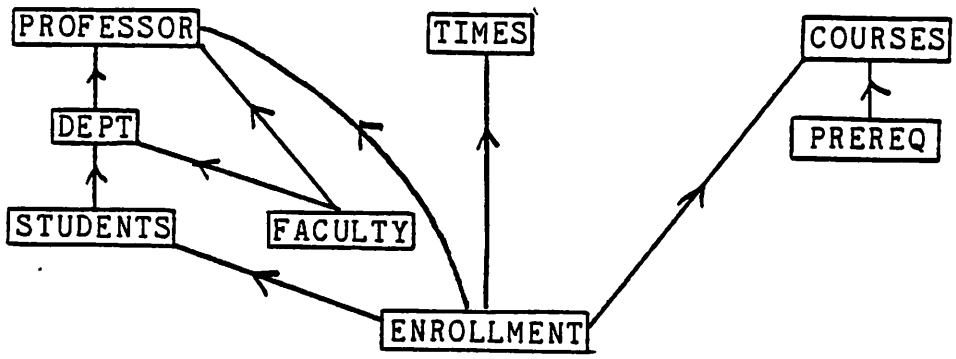


fig. 3.2 - Insertion Order DAG for University Database

city and side effects of update operations, and (2) formulate mapping rules to transform a design schema into either a relational or CODASYL schema while preserving both atomicity of update operations and controlled side effects. The design goals are stated to stress the support for these concepts:

(1) Preservation of Atomicity

An atomic update operation affects a single logical object in the target schema. In a relational schema, this means that a single update affects a single tuple, and in a CODASYL schema, a single update affects a single record instance. Intuitively, we desire a logical schema in which information about a single object is stored in a single place, so that operations that update the information need only affect a single logical object within the target

schema. This property can be described as "minimum update", and leads to logical schemas with a minimum amount of redundant information.

(2) Preservation of Side Effects

Any additional induced atomic operations are exactly those described for the design model.

The CODASYL data model provides primitive constructs to automatically support the design model's associations and single-valued relationships. The latter can only be supported when it is devoid of properties. To take advantage of this possibility, the mapping rules for CODASYL should preserve the following design goal in addition to the above:

(3) Automatic Integrity Support

The side effects of association and single-valued relationship should be handled automatically.

This support can be provided by using the set-membership options for removal and storage class in the CODASYL data definition language.

3.3. Relational Design

In this section, we describe a methodology for relational database design. It is based on mapping a design schema specification into a relational schema while

preserving the design goals of the previous section.

3.3.1. Mapping Rules

We would like to formulate rules for mapping that preserve the atomicity of update operations and avoid undesirable update side effects. We require that all information about entities and relationship instances, i.e. property and association values, be inserted and deleted at the same time as the creation or destruction of the object. Furthermore, this should be accomplished with a single update operation within the relational model. Similarly, a single relational update operation should be sufficient to modify the representation of the value of a property or association.

An identifier has already been defined as a 1-to-1 property of an entity set which is used to uniquely represent the entities of the set. Similarly, a relationship instance can be identified by the identifiers of the entities involved in the relationship. A primary function is a property or an association specified in the design schema. For relational design, we assume that associations are represented as functions which map into the identifier value set of the range object. This has no implication on the structure of the physical schema. A primitive object is either an entity set or relationship in its role as the domain of a primary function. Intuitively, by grouping an

object with its functionally related range values in a single place, i.e. in a single tuple, the fragmentation of operations can be avoided. Undesirable side-effects can be avoided by insuring that independent objects of the design schema are represented by independent constructs in the relational schema, i.e. no more than one primitive object is assigned to a single relation. The mapping rules are as follows:

1) Each entity set has an explicit identifier which represents it globally throughout the relational schema.

2) The identifier(s) of a primitive object together with the range values of its primary functions are grouped within the same relation of the relational schema.

3) There is one and only one primitive object per relation of the relational schema.

An entity or relationship instance is represented by a single tuple in the relational schema. The object's property values and association values are represented as attribute values in the tuple. The object is created by inserting the tuple and destroyed by deleting it. Property and association values are modified by updating the appropriate attribute value. The atomicity of operations has been preserved.

Consider the third mapping rule. Suppose that more than one primitive object is assigned to a relation. Then it is the case that in deleting an instance of one of the assigned objects, an instance of another assigned object will be inadvertently deleted, because both objects share the same tuple. By placing one primitive object per relation, information about a primitive object can not be lost by updating another object, except as the result of the controlled side effect discussed in the previous section. Uncontrolled side effects are avoided.

We should note that the relational model does not provide logical constructs for automatically supporting the integrity constraints implied by the design model. There are no facilities to automatically support the deletion semantics of associations or the insertion semantics of relationships, i.e. the entities participating in the relationship must exist before the relationship instance is created. In addition, no mechanism exists for automatically maintaining the functionality of a single-valued relationship. The mapping rules treat a single-valued relationship as though it were a general binary relationship. The kinds of mechanisms we need are those that enforce: (1) the uniqueness of relational keys, (2) subset constraints, i.e. values in the attribute of one relation are a subset of the values in an attribute of another

relation (foreign keys), and (3) functional interrelationships among the attributes of a relation.

An alternate set of mapping rules which are more algorithmic in nature is:

1') Each entity set has an explicit identifier which represents it globally throughout the relational model.

2') For each entity set E, define a relation R(E). The attributes of R(E) are made up by (i) a key attribute for the identifier of E, (ii) value attributes for the properties of E, and (iii) foreign key attributes for the associations of E.

3') For each general relationship R over E_1, E_2, ... , E_n, define a relation R(R). The attributes of R(R) are made up of foreign key attributes for the identifiers of E_1, E_2, ... , E_n and the value attributes for the properties of R.

4') For each single-valued relationship S over E_1 and E_2 (E_1 is single-valued), define a relation R(S). The attributes of R(S) are made up of foreign key attributes for the identifiers of E_1 and E_2.

Examples of Relational Design

(1) Employee Database

Emp, Dept, Alloc, Qual, and Mgr are each mapped into a relation. ENO, DNO, JID are the identifiers of Emp, Dept, and Job respectively. Hence (DNO, JID) and (ENO, JID) are the identifiers of Alloc and Qual. The above relational schema can be represented in terms of the data definition language of the INGRES system [EPST79]:

```
create EMP (ENO = i2, ENAME = c20, BIRTHYR = i2,
            WORKS-IN = i2, ASSIGNMENT = i2)
create DEPT (DNO = i2, DNAME = c10, LOCATION = c10)
create JOB (JID = i2, TITLE = c20, SALARY = i4)
create ALLOC (DNO = i2, JID = i2, NUMBER = i2)
create QUAL (ENO = i2, JID = i2)
create MGR (DNO = i2, ENO = i2)
```

Associations are represented by foreign key attributes, which are defined over the same domain as the identifier of the range entity set (e.g., WORKS-IN, ASSIGN).

(2) University Database

Students, Professor, Courses, Dept, Times, Faculty, Prereq, and Enrollment are each mapped to a single relation. REGNO, SSNO, CNO, and TIMEID are the identifiers of Student, Professor, Courses, and Times. The relational schema is:

```
create STUDENTS (REGNO = c10, SNAME = c20, CLASSYR = i2,
                 MAJOR = i2)
create COURSES (CNO = i2, TITLE = c30)
create PROF (SSNO = c9, PNAME = c20, RANK = c2)
create DEPT (DNO = i2, DNAME = c10, LOCATION = c10,
             CHAIRMAN = c9)
create TIMES (TIMEID = i2, TERM = c3, YEAR = i2,
              TIME = c10)
create FACULTY (SSNO = c9, DNO = i2)
create PREREQ (CNO = i2, PRECNO = i2)
```

```
create ENROLL (CNO = i2, SSNO = c9, TIMEID = i2,
               REGNO = c10, GRADE = c2)
```

Note that in certain cases, the naming of the relational attributes can be ambiguous. This is the case whenever a relationship involves the same entity set more than once, e.g. Prereq(courses,courses). A meaningful renaming of the attributes can be suggested by the database designer.

3.3.2. Characterization of Relational Schemas

In this section, we show that the schemas that result from an application of the mapping rules are in 4NF. Thus, the intuitive claim that a 4NF relational schema minimizes undesirable side effects is shown to be justified.

Our discussion of normal form theory closely follows that of [DATE77]. Relational normal form theory has evolved in an attempt to achieve the design of relations which are free from undesirable properties. Central to the theory is the concept of functional dependence (within a relation). Given a relation R, attribute Y of R is functionally dependent on attribute X of R if for all time, a single Y value is associated with each X value; i.e., there is a function (which can change over time) that maps X values into Y values. The definition can be extended to the case where X and Y are composite. Suppose that X is composite, i.e. consisting of more than one attribute. Then Y is fully functionally dependent on X if it is

functionally dependent on X but not functionally dependent on any subset of X.

Consider the relation EMP-DEPT of figure 3.3, constructed from information about employees and their jobs. ENAME, BIRTHYR, and DNO are functionally dependent on ENO. DNAME and LOCATION ARE functionally dependent on DNO, and by transitivity, also on ENO. The sample relation exhibits several update anomalies. If Paul is fired (deleted from the relation), then information about department 002 is inadvertantly lost as well (uncontrolled side effect). Changing the TOY department's location from SAN JOSE to BERKELEY affects more than one tuple in the database. Information about the TOY department is redundantly stored (fragmentation of atomic operations).

We are now ready to define the relational normal forms. A relation R is in 1NF if and only if all of its

EMP-DEPT

ENO	ENAME	BIRTHYR	DNO	DNAME	LOCATION
001	JOHN	1950	001	TOY	SAN JOSE
002	JEFF	1951	001	TOY	SAN JOSE
003	PAUL	1949	002	SHOE	BERKELEY

fig. 3.3 - Sample Relation

underlying domains contain atomic values only. In other words, the domains can not consist of aggregated values, such as sets. EMP-DEPT is in 1NF. A primary key is a collection of attributes whose values uniquely identify the tuples of the relation for all time. ENO is the key of EMP-DEPT. A relation R is in 2NF if it is in 1NF and every non-key attribute is fully dependent on the primary key. Since every attribute of EMP-DEPT is functionally related to ENO, the relation is in 2NF. If Y is functionally dependent on X and Z is functionally dependent on Y, then by transitivity, Z is functionally dependent on X. Such dependencies can lead to update anomalies. A relation R is in 3NF if it is in 2NF and every non-key attribute is non-transitively dependent on the primary key. EMP-DEPT is not in 3NF because of the transitive dependencies of DNAME and LOCATION on ENO through DNO.

Note that in the above we have assumed that no relation has more than one candidate key. The definitions can be relaxed to include this case.

Unfortunately, 3NF relations do not avoid all update anomalies. 4NF has been defined to eliminate these. [FAGI77a] introduces the concept of multivalued dependence, which intuitively means that although a given X value is not associated with a single Y value, it is associated with a well defined set of Y values. Functional

dependence is a special case of multivalued dependence. A normalized relation R is in 4NF if and only if whenever there exists a multivalued dependency in R, e.g. between attributes A and B, then all the attributes of R are also functionally dependent on A. [DATE77] give the following intuitive definition of 4NF:

> A relation R is said to be in 4NF if and only if, for all time, each tuple of R consists of a primary key value that identifies some entity, together with a set of mutually independent attribute values that describe that entity in some way.

For example, relation EMP is in 4NF: each EMP tuple consists of an ENO value, which uniquely identifies a particular employee, together with four pieces of information which describe the employee - employee name, birthyear, department worked in, and job assigned. Furthermore, each of the descriptive items is independent of the others, i.e. they are not functionally dependent on each other. The identifiers of entity sets will always coincide with the concept of primary key. The identifier of a relationship, however, may be a superset of the primary key. For example, the primary key of the single-valued relationship relation MGR(DNO,ENO) is DNO, not (DNO, ENO).

We can show that violations of mapping rule (3) always lead to violations of some normal form. In what follows, it is important to precisely define what is meant by assigning more than one independent design schema

object to the same relation of a relational schema.

We say that an entity set and a relationship overlap if and only if the relationship is defined over the entity set. Two relationships overlap on an entity set E if and only if the subset of E that participates in the first relationship is the same as the subset that participates in the second, for all time. The relation formed from the join of the relationship relations, on the attribute that represents the identifier of the overlapped entity set, can be decomposed without loss of information into the original relations.

The normal form violations can be classified as:

A. Putting two primitive objects together in the same relation which are unrelated by functions in the design schema. This results in a relation which is not in 2NF.

Example: Consider what would happen if the entity sets Job and Dept were assigned to the same relation, e.g., JOB-DEPT. This relation could be formed by taking the Cartesian product of JOB and DEPT from the the example of the previous section. The key of JOB-DEPT is (JID, DNO), but not all attributes are fully functionally determined by the composite attribute, e.g. JID functionally determines TITLE. This is a violation of 2NF.

B. Putting an entity set and a relationship involving it within the same relation, i.e. the entity set and the relationship are overlapped. This too results in a 2NF violation.

Example: Consider what would happen if the entity set Job and the relationship Qual were assigned to the same relation, e.g. JOB-QUAL. This relation could be formed by taking the join of JOB and QUAL on the attribute JID. The key of JOB-QUAL is (JID, ENO), but not every attribute is functionally determined by these, e.g. JID functionally determines TITLE. This is a violation of 2NF.

C. Putting two functionally related entity sets within the same relation. This is a 3NF violation.

Example: Suppose that the entity sets Emp and Job are assigned to the same relation, e.g. EMP-JOB. This is formed by joining EMP on ASSIGN to JOB on JID. The key of EMP-JOB is ENO. However ENO functionally determines JID, through the Assign association, which in turn functionally determines TITLE. Thus, although ENO functionally determines TITLE, this functional dependency is transitive. EMP-JOB is not in 3NF.

D. Putting two relationships together which overlap on a common participating entity set. This will result in

a violation of 4NF.

Example: Suppose that the relationships Alloc and Qual have been assigned to the same relation, e.g. ALLOC-QUAL. This can be formed from the non-loss join of ALLOC on JID with QUAL on JID. The key of ALLOC-QUAL is (ENO, JID, DNO). ALLOC-QUAL is already not in 2NF because of the partial dependence of NUMBER on (ENO, JID, DNO), i.e., (JID, DNO) functionally determine NUMBER. Assume that Alloc does not have the property Number, i.e. consider the projection of ALLOC-QUAL on (ENO, JID, DNO). This relation is in 3NF, because it is "all key," but not in 4NF. ALLOC-QUAL contains two multivalued dependencies which are not functional: "eno on jid" and "dno on jid."

In the above, we have assumed that relationships can only be assigned to the same relation if they overlap. Assume that Alloc and Qual have been assigned to the same relation, but do not overlap, i.e. the allocated jobs need not be the same as those assigned. Then the above example will result in a relation which is in 4NF but which can not have been derived from a design model schema. Thus, schemas derived by our mapping rules are strictly contained within the class of all 4NF schemas. The results of our characterization of update anomalies are summarized in figure 3.4.

	ENTITY	RELATIONSHIP
ENTITY	non-overlapping = 2NF	non-overlapping = 2NF
	association = 3NF	overlapping = 2NF
RELATION-SHIP	non-overlapping = 2NF	non-overlapping = 2NF
	overlapping = 2NF	overlapping = 4NF

fig. 3.4 - Summary of Normal Form Violations

Intuitively, functional dependencies not only represent integrity constraints, but also relationships which are modelled within the schema. Therefore we can define a correspondence between the constructs of the design model and functional dependencies. Let ID_E be the attribute derived from the identifier of entity set E, and P_V be the attribute derived from the property mapping P which ranges over the value set V. Then the following functional dependencies are implied:

Property	P: E --> V	==> ID_E --> P_V
Association	A: E_1 --> E_2	==> ID_{E_1} --> ID_{E_2}
S.V. Relationship	S: E_1 --> E_2	==> ID_{E_1} --> ID_{E_2}
$R \subseteq E_1 \times \ldots \times E_n$ Property	P: R --> V	==> $ID_{E_1}, \ldots, ID_{E_n}$ --> P_V

No other dependencies are implied except for those above and their logical consequences.

Within the relations derived by the mapping rules, it is possible to show that the above dependencies imply that the schema is in 4NF.

Theorem: A relational schema derived from a design model schema using the mapping rules is in 4NF.

Proof: Because all value sets are defined over atomic values, and designed relations are defined over domains derived from these value sets, the relational schema must be in 1NF.

A relation is derived from an entity set by defining it over attributes for the identifier, properties, and associations of the entity set. The only dependencies are of the form ID_E --> P_V or ID_E --> ID_{E_1}. There is no possibility for transitive, partial, or multivalued dependencies in such a relation. Thus the relation is in 4NF.

A relation is derived from a single-valued relationship by creating a binary relation over the identifiers of the entity sets involved. The only functional dependency that can hold is ID_{E_1} --> ID_{E_2}. Again the relation is in 4NF.

A relation is derived from a general relationship by defining it over attributes for the identifiers of the entity sets over which it is defined, together with attributes for the properties of the relationship. The depen-

dencies are then of the form ID_{E_1}, ... , ID_{E_n} --> P_V. No functional dependencies can hold among the ID_{E_i}, because a relationship is defined over the Cartesian product of the entity sets, i.e. a relationship instance can not be identified by any proper subset of the entities which take part. ID_{E_1}, ... , ID_{E_n} form a key of the relation, and every non-key attribute is a function of the key. Suppose that contrary to the assertion of the theorem, the relation is not in 4NF. Then it is equal to the join of two subrelations [FAGI77a]. Either the identifiers making up the key are split between these two relations or they are not. If they are split, then the relationship must also be decomposible, contradicting the assumption that each relationship in the design schema is indecomposible. If the key resides in one of the component relations, then attributes of the relation can not be functions of the key, contradicting mapping rule (2). QED.

The relational schemas derived by our mapping rules are in 4NF. Because our mapping rules have been formulated to preserve the design goals, and thus to control update anomalies, this correspondence can be viewed as a justification of why 4NF schemas are desirable. If a 4NF schema has not been derived by the mapping rules, it can still exhibit update anomalies.

In this section we have shown that there is a correspondence between our designed schemas and 4NF schemas. This indicates why 4NF schemas are often well-behaved under the update operations of the relational model. The importance of the "one object per relation" rule was illustrated by showing that in violating this rule, normal form violations will occur. This indicates that a schema designed without this rule would suffer from undesirable update anomalies.

3.4. CODASYL Design

In this section, rules are formulated for designing a "logical" CODASYL schema that preserves the design goals. The CODASYL/DBTG proposals [CODA71, CODA73, CODA78] include many aspects of physical design within the schema definition language. This makes it particularly difficult to choose what is meant by a logical schema. For the purposes of this section, we shall assume that a logical schema consists of the specification of record types, data items, keys (identifying data items), set types, and set membership options.

Another difficulty with the CODASYL model is that it has evolved over time. The newest proposal [CODA78] has eliminated many of the shortcomings of the previous ones. However most of the working implementations are based on these earlier proposals. We describe the mapping rules for

the 1973 proposal and indicate how these should be modified for the 1978 proposal.

3.4.1. Mapping Rules

The same requirements that were placed on the relational mapping rules are again needed here. Creation and destruction of entities and relationship instances must be accomplished with a single CODASYL operation. Information about a primitive object should be stored in one place to avoid fragmentation of updates. In addition, the facilities of the CODASYL model should be used to provide automatic support for the insertion and deletion semantics of relationships and associations, and the functionality constraint of single-valued relationships and associations.

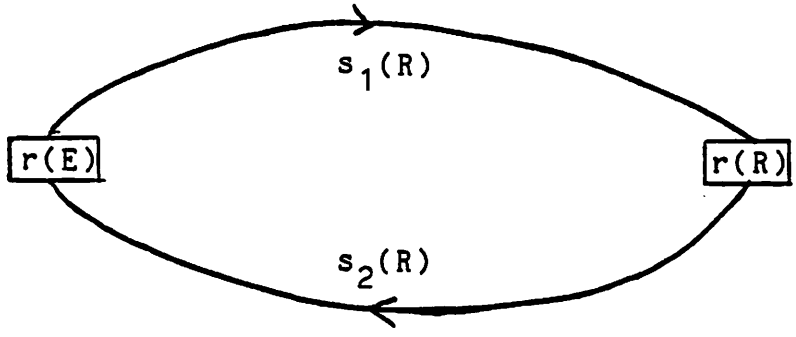
The mapping rules are as follows:

(1) Each entity set has an explicit identifier.

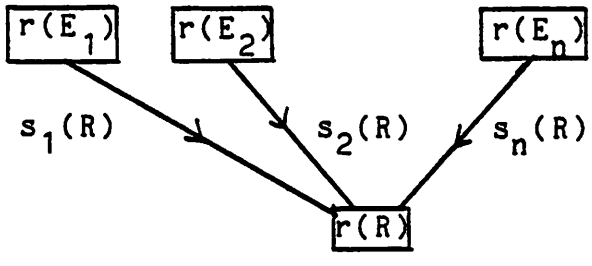
(2) For each entity set E define a record type r(E). The data items of r(E) are made up of the identifier of E and the properties of E.

(3) For an association or single-valued relationship $R(E_1, E_2)$ where $E_1 \neq E_2$, define a set type s(R) with $r(E_2)$ as the owner record type and $r(E_1)$ as the member record type.

(4) For a single-valued relationship R(E,E), define a record type r(R) having no data items, and a pair of set types $s_1(R)$ and $s_2(R)$ forming a cycle between r(E) and r(R). The assignment is depicted below:



(5) For a general relationship $R(E_1, E_2, \ldots, E_n)$ define a confluent hierarchy, consisting of a record type r(R) with only the properties of R as its data items, and n set types $s_1(R)$, $s_2(R)$, ... , $s_n(R)$ as shown below:



Rule (4) is necessary because the [CODA73] proposal forbids sets in which the same record type participates as both member and owner. This restriction has been lifted in the newest proposal, and Rule (4) is no longer necessary.

All the object types of the design model have been mapped into object types of the DBTG data model. Two kinds of record types have resulted from the mapping: those which contain an identifier data item, and those which do not. The former are called self-identified record types, and the latter link record types. Self-identified record types always represent entity sets, while link record types represent relationships and, in the case of the 1973 proposal, self single-valued relationships.

We introduce the concept of total and partial set membership. A record is a total member of a set type if every record occurrence is a member of a set occurrence. A member that is not total is said to be partial. The membership of a link record type in any set type should be total. The membership of a self-identified record type is total in any set which represents an association and partial in any set which represents a single-valued relationship.

Natural enforcement for total membership is provided by specifying mandatory/automatic for the removal/storage class of the member subentry (more will be said about this in section 3.4.2). If a sequence of set types form a cycle, then one set is required to have manual specified for the storage class. A cycle of associations can not exist because of the constraints on the order of inser-

tion. In the case of self single-valued relationships, the link record is a total member of $s_1(R)$ while r(R) is a partial member of $s_2(R)$ and therefore has a manual storage class.

A final mapping rule for DBTG schemas applies to the correct assignment of set membership specifications:

(6) The membership of a link record in any set type is total. The membership of a self-identified record type in any set that represents an association or relationship association is total, but not otherwise. All set memberships that are not total are optional/manual. All total memberships are mandatory/automatic.

By placing all the information about an entity in one place (within a single record instance, its data items and set memberships), we insure that the atomic operations of creating and destroying an entity or relationship instance are supported by the DML operations STORE and ERASE. A property of an entity can be atomically changed by a MODIFY operation. An association can be changed by using a MODIFY operation with "ONLY association set name MEMBERSHIP" specified. Similarly, all the information about a relationship instance is encoded in its associated record, data items, and set memberships.

The choice of set memberships is crucial in achieving the controlled side effects of the design model. The domain of an association is represented by a record type that is a mandatory/automatic member of a set with the owner record type representing the range. The mandatory specification for removal guarantees that if the owner (range) is erased, then the member (domain) will automatically be erased. These deletions can propagate. In addition, the automatic specification for storage class guarantees that a member record will be placed in the appropriate association set when first created. Support for total membership is insured by making it impossible for a member record instance to exist unless it participates in the association set. The total functionality of associations is achieved. Because CODASYL sets already model functions [NIJS75], the optional/manual specification for sets which represent single-valued relationships insures that partial functionality is maintained.

A general relationship is represented by its link record type, which is a mandatory/automatic member of sets which link it to the record types for the participating entity sets. Again, the record for a relationship instance can not be created unless the records of the participating entities already exist. Also, an instance can not exist unless it is a member of all the relationship sets.

Further, the mandatory specification guarantees that a relationship record will automatically be deleted if any of the entity records which own it are deleted.

In summary, the derived CODASYL schema will preserve the atomicity of update operations, avoid undesirable side-effects, and guarantee the functionality of associations and single-valued relationships.

Examples of CODASYL Design

(1) Employee Database

Each entity set is mapped to a single self-identified record type, e.g. DEPT, EMP, JOB. The data items represent the identifier and properties of each entity set. The 1978 DDL specification is:

```
RECORD NAME IS DEPT
  KEY DEPT-KEY IS DNO DUPLICATES ARE NOT ALLOWED
  01 DNO TYPE IS BINARY
  01 DNAME TYPE IS CHARACTER 10
  01 LOCATION TYPE IS CHARACTER 20

RECORD NAME IS EMP
  KEY EMP-KEY IS ENO DUPLICATES ARE NOT ALLOWED
  01 ENO TYPE IS BINARY
  01 ENAME TYPE IS CHARACTER 20
  01 BIRTHYR TYPE IS BINARY

RECORD NAME IS JOB
  KEY JOB-KEY IS JID DUPLICATES ARE NOT ALLOWED
  01 JID TYPE IS BINARY
  01 TITLE TYPE IS CHARACTER 15
  01 SALARY TYPE IS BINARY
```

Similarly, each relationship is mapped to a single link record type, e.g. QUAL, ALLOC:

```
RECORD NAME IS QUAL

RECORD NAME IS ALLOC
01 NUMBER TYPE IS BINARY
```

The record types are interrelated by sets which represent associations, relationship associations, and single-valued relationships. In the first two kinds, the membership is total, in the latter, the membership is partial:

```
SET NAME IS WORKS-IN
  OWNER IS DEPT
  ORDER IS PERMANENT INSERTION IS SYSTEM DEFAULT
  MEMBER IS EMP
  INSERTION IS AUTOMATIC RETENTION IS MANDATORY
  SET SELECTION IS THRU
      WORKS-IN OWNER IDENTIFIED BY KEY DEPT-KEY

SET NAME IS MGR
  OWNER IS EMP
  ORDER IS PERMANENT INSERTION IS SYSTEM-DEFAULT
  MEMBER IS DEPT
  INSERTION IS OPTIONAL RETENTION IS MANUAL
  SET SELECTION IS THRU
      MGR OWNER IDENTIFIED BY KEY EMP-KEY

SET NAME IS QUAL-EMP
  OWNER IS EMP
  ORDER IS PERMANENT INSERTION IS SYSTEM-DEFAULT
  MEMBER IS QUAL
  INSERTION IS AUTOMATIC RETENTION IS MANDATORY
  SET SELECTION IS THRU
      QUAL-EMP OWNER IDENTIFIED BY KEY EMP-KEY

SET NAME IS QUAL-JOB
  OWNER IS JOB
  ORDER IS PERMANENT INSERTION IS SYSTEM-DEFAULT
  MEMBER IS QUAL
  INSERTION IS AUTOMATIC RETENTION IS MANDATORY
  SET SELECTION IS THRU
      QUAL-JOB OWNER IDENTIFIED BY KEY JOB-KEY

SET NAME IS ALLOC-JOB
  (similar to QUAL-EMP, QUAL-JOB)

SET NAME IS ALLOC-DEPT
  (similar to QUAL-EMP, QUAL-JOB)
```

The schema is depicted in figure 3.5. Several DDL clauses have been omitted in the above because they deal more with physical than logical design.

(2) University Database

Here we will only show the graphical form of the schema (see figure 3.6). It should be obvious from the preceding example how to derive the DDL specifications. As in the relational case, the sets that are used to form the confluent hierarchy for the relationship Prereq would normally be assigned the same name. The designer can aid the design system by specifying more meaningful names for these sets.

3.4.2. Characterization of Networks

The CODASYL model, in spite of its complexity, enables the designer to exercise a considerable amount of

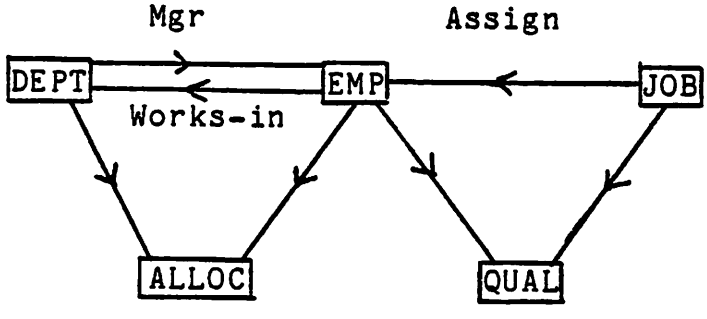


fig. 3.5 - Data Structure Diagram for Employee Database

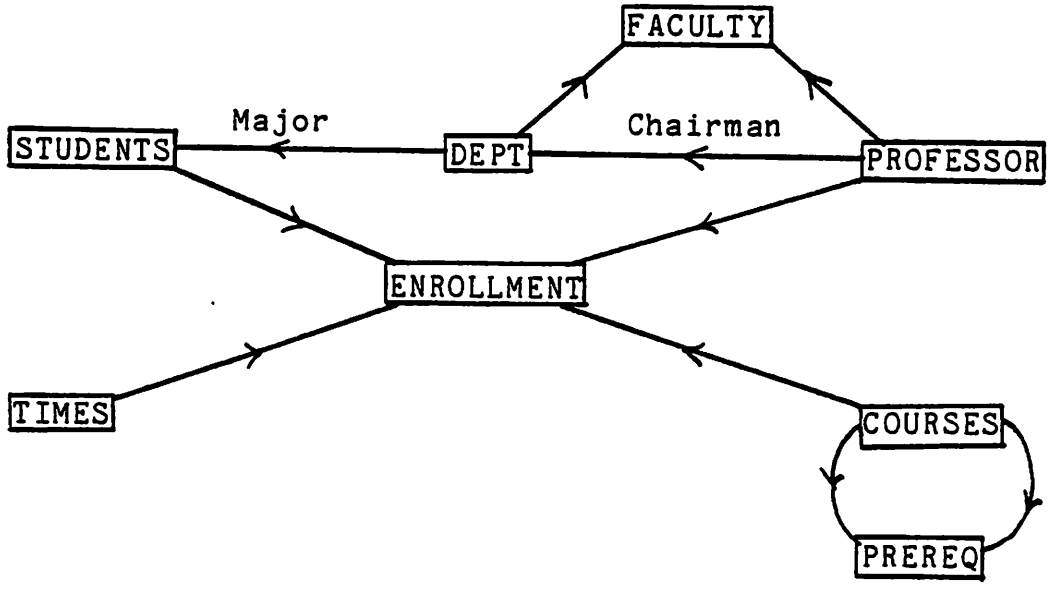


fig. 3.6 - Data Structure Diagram for University Database

control in designing his schema. The utility of the often confusing array of features of the model become clearer in light of the design goals and the design model.

The CODASYL set construct is a natural mechanism for supporting functional interrelationships among record types. This enables CODASYL to support the functional relationships, including relationship associations, of the design model in a straight-forward way. Thus the design goal of automatic integrity support for functional relationships can be achieved.

The CODASYL retention and storage class specification for set membership provides a mechanism to support the

semantics of update side-effects. Mandatory retention means that once a record becomes a member of a set, it can not be removed from the set without being deleted. This is a consequence of the existence dependency implied by associations within the design model. Further, support is provided to delete all members of a set when the owner is deleted. Thus the membership options can be used to support deletion side-effects. Optional retention implies no such dependency among the owner and the members, and hence no deletion side-effects are needed.

Automatic storage class insures that when a record is entered into the database, it is made a member of each set for which its automatic membership is specified. This mechanism supports our insertion semantics, because a record can not be entered into the database until its owners are already entered. By simultaneously specifying mandatory/automatic, a record can never exist outside of its automatic set memberships. This is precisely the semantics implied by the association. Optional/manual supports the semantics of the single-valued relationship. The semantic utility of mandatory/manual or optional/automatic are not clear. Update side-effects can be supported automatically with the constructs of the CODASYL model.

Atomicity of update is a consequence of CODASYL's choice of the record instance as the atomic unit of

update. This provides support for the concept of a "primitive object." Further, the model supports the uniqueness of identifiers by allowing a data item to be specified with the option "duplicates not allowed."

In summary, several constructs of the CODASYL model are more easily understood in terms of the concepts of the design model. CODASYL sets implement relationships and the set membership options specify the update semantics for these relationships.

3.5. Conclusions

The purpose of the design model is to provide an unambiguous specification and description of the primitive objects of a database. In this chapter, we have proposed a system independent methodology for logical database design that maps this description into either a relational or DBTG schema, while preserving desirable update properties of the original schema. Note that there is nothing about this approach that limits it to these models above.

In addition, the designed schemas have been characterized and were shown to be in a form that avoids anomalous behavior under update operations. In the relational case, the subset of 4NF schemas that correspond to a schema designed by the design rules is well-behaved under update, thus indicating why 4NF schemas are desir-

able. In the CODASYL case, features of the model are clarified in terms of their utility in supporting the concepts of the design model.

CHAPTER 4

PHYSICAL DESIGN

4. Physical Design

4.1. Introduction

The access path schema provides a useful interface between the user's logical view of the data and its physical implementation. In this chapter, we will describe an implementation-oriented physical design methodology which is largely independent of the specific database system and data model. The implementation-dependent aspects will be discussed in section 4.4.

The approach is to specify the requirements of the design in terms of the support assigned to each logical access path in the physical schema. A specification for a particular access path cannot be made if it would conflict with the specifications for other paths of the schema; i.e., certain requirements cannot be met simultaneously. The user's expected access patterns are used to direct the design process. A system specific mapping is then invoked to implement the access path schema in terms of the storage structures available in the target system.

4.2. Algebraic Structure for Physical Design

4.2.1. Storage Structure Properties and Constraints

For the purposes of implementation-oriented design, we shall use the logical access paths of the design schema. An access path schema will be used to represent those paths actually chosen for support. Properties of an access mapping can be formulated to capture the desirable characteristics of traversing the mapping in either the functional or inverse functional direction. Essentially, we are concerned with how data should be placed and physically interconnected in order to achieve fast access among data objects. Consider the schema function f: A --> B. The following properties of the mapping can be defined:

(1) Evaluated: given a in A, f(a) can be found without an exhaustive scan of B, i.e. the cost to access f(a) is less than the cost to access every element of B.

Example: The function WORKS-IN is "evaluated" if a fast access path exists between EMP and DEPT, e.g. a child to parent pointer from EMP to DEPT. The property mapping 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, e.g. 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, i.e. the cost to access the elements in the inverse is less than the cost to access an arbitrary subset of the same cardinality.

Example: WORKS-IN is "clustered" if EMP records within the same DEPT are placed together, e.g. 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, i.e. the cost to access both is less than the cost to access each of them separately.

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, i.e. name values are removed by

compression.

We assume that each object of the schema, be it a value, an entity, or a relationship instance, is assigned to a single stored record. Replication, e.g. the replication of data item values to record instances, will be made explicit by introducing new objects into the schema. The usual concept of "record" can be represented as a concatenation of the stored records of the values that make up the fields of the record. Our approach does not preclude the record segmentation and allocation techniques described in [SCHK78]. Given this assumption, certain implication rules can be formulated:

(i) well placed ==> 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 ==> indexed

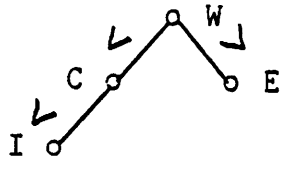
By placing the elements of $f^{-1}(b)$ together, an exhaustive scan of all the domain objects of f is not necessary. Once an object in the cluster has been found, the entire cluster has been found.

(iii) well placed ==> 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, it is possible to have a mapping which is 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 opposite is possible too. 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.

The implication rules can be used to impose a partial ordering among the properties (properties are denoted by their first letter):



W > C > I

W > E

A label is an assignment of properties to an edge of the integrity schema. There are six distinct labels: W, <C,E>, <I,E>, C, I, and E. We desire to generate schemas

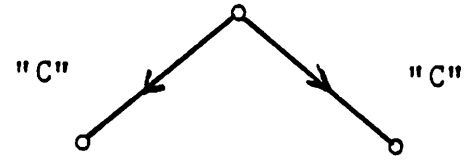
with the maximum possible support for each path. Therefore, we assume that at least each access path is evaluated. The task then is to assign one of the first three labels, denoted as "W", "C", and "I", to each access path of the logical schema. This indicates the requirements for data placement and interconnection to be realized within the physical schema.

A labelling is an assignment of a label to each edge of the schema, denoted as an n-tuple $(l_1, l_2, \ldots, l_n)$ where n is the number of edges in the schema. The assignment is subject to constraints which are shown below. The partial ordering among properties induces a partial ordering among labels as well: "W" > "C" > "I". A partial ordering can be defined for labellings. Let L_1 and L_2 be two labellings over the same schema. We say that $L_1 = L_2$ if for each edge in the schema, L_1's assigned label is the same as L_2's assigned label. We say that $L_2 > L_1$ if for each edge in the schema, either L_1's assigned label is the same as L_2's or L_2's label > L_1's, and $L_1 \neq L_2$. Note that under this definition, some labellings are incomparable, e.g. L_1 = ("W","C") and L_2 = ("C","W"). A maximal labelling is a labelling L for which there exists no labelling L' such that L' > L.

An obvious approach to achieving a maximal labelling is to assign "W", the label that represents the highest

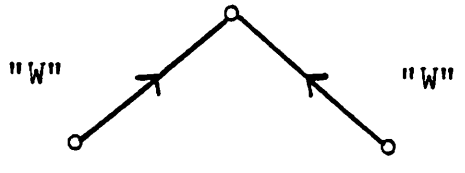
degree of support, to each edge. Unfortunately, certain labellings represent a choice of properties which can not be supported simultaneously within a schema. There are four constraints which conflict-free labellings must meet:

(i) cluster constraint:



It is not possible to label more than one outedge of a node with a "C" or "W". Clustering places together all domain objects which share the same range object. It is not possible to partition the domain on more than one function and still achieve this advantageous placement. Note that 1-to-1 properties do not cause a conflict because a 1-to-1 function partitions the domain objects into clusters of size one. This can always be supported regardless of additional clustering.

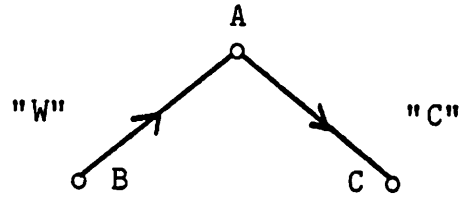
(ii) placement constraint:



It is not possible to label more than one inedge of a node with "W". Well-placement places clusters of domain objects with a common range object near that range object. It is

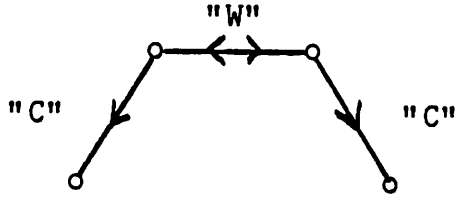
not possible to achieve this advantageous placement simultaneously for domain objects from more than one function.

(iii) path constraint:



It is not possible to simultaneously label an inedge of a node "W" while labelling an outedge "C". The placement of B object clusters near their associated A objects destroys the advantageous clustering of the A objects. 1-to-1 functions do not cause the constraint to be violated.

(iv) implied constraints:



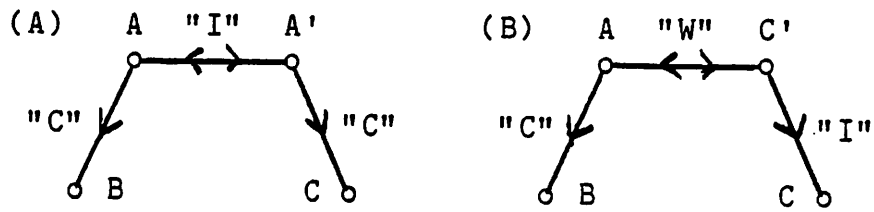
Certain compositions of functions and their properties result in the violation of one of the above constraints. For example, this schema would cause a violation of an implied cluster constraint.

4.2.2. Conflicts and Replication

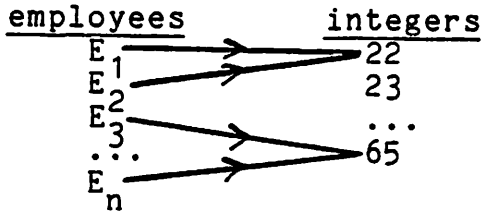
A conflict is a violation of a cluster, placement, or path constraint. A labelling algorithm is permitted to

introduce conflicts into a labelled schema, but these must eventually be resolved by replicating schema objects; i.e., new objects are created which are copies of existing schema objects. The degree of a schema is the number of conflicts that an algorithm may introduce during labelling. The expansion factor of a schema is the ratio of the size of a replicated schema to the original schema size. Replication can also be controlled by placing a limit on the expansion factor.

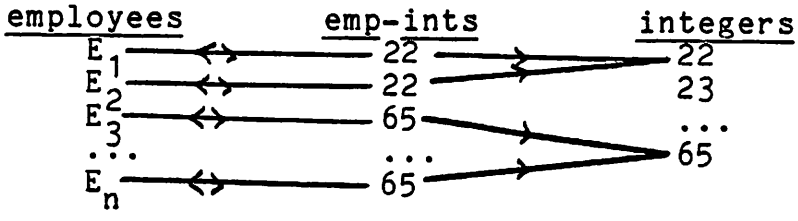
A cluster conflict can be resolved by one of the following methods:



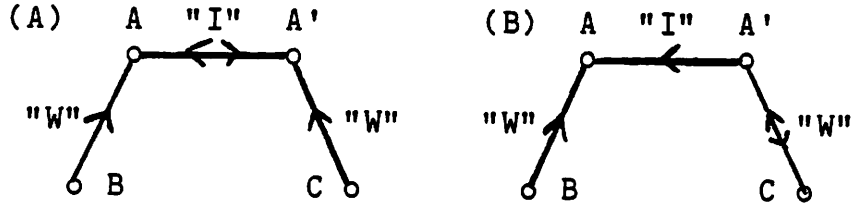
In (A), a copy of the domain object is made, and both the original and the copy are clustered on the appropriate ranges. In (B), a copy of the range is made and placed in 1-to-1 correspondence with the original domain object. Further, the domain and the copy are placed near each other. To illustrate this, consider the entity set employees and the value set integers, interrelated by the property function age. Schematically, the following situation can arise:



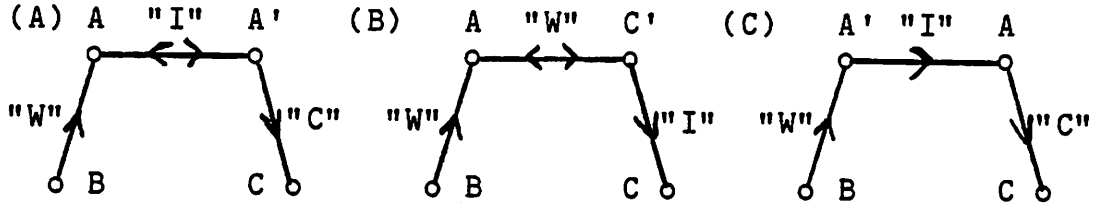
The effect of type (B) cluster resolution is to replicate the age values so there is one age value per employee:



A placement conflict is resolved analogously:



In addition, path conflicts can also be resolved via replication:



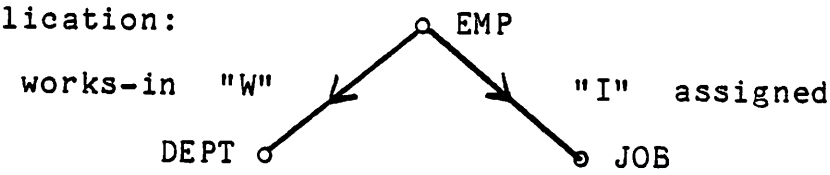
To simplify the labelling algorithms, we shall avoid path resolution by enforcing all path constraints.

The labelling schemes for resolving conflicts have been chosen to avoid implied constraints. A 1-to-1 edge

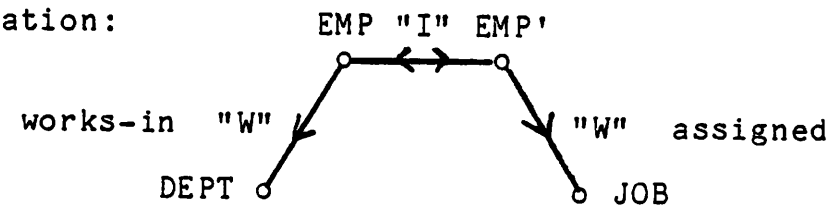
can only appear as either an identifier property or as the result of replication. Thus implied constraints are guaranteed to be avoided.

A conflict can be resolved through replication. The degree of a schema is therefore a measure of the amount of replication we are willing to tolerate during the labelling process. 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:



Further, we assume that all property mappings of EMP are replicated to EMP'. Access from JOB to EMP' to access these properties is improved in the schema with replication. Meanwhile, access from JOB to DEPT, or to any object via a non-property edge, is not impaired and is at no worse cost than in the original schema. However, updates to properties of 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 properties of EMP exceeds the frequency of updates of EMP, then replication is cost-effective.

Certain types of replication incur only storage cost. Replication of property values to domain objects can be accomplished without incurring an additional update cost, although there is an increase in storage requirements. When modifying a property 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 "San Jose" to "Sunnyvale", then the name of the location has not been recoded (San Jose has not been renamed to Sunnyvale), but rather LOCATION now maps into a different value. Thus 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 1-to-1 correspondence with the A objects. Because the same number of objects are created in either method, the choice then depends on which of A or C is updated most frequently. Further, under type (B) replication, each update to C must be propagated to $\frac{n(A)}{n(C)}$ replicates, where n(X) is the number of X objects in the schema. If U_A and U_C are

the update frequency for objects A and C respectively, then type (A) is chosen when $U_A * 1 < U_C * \frac{n(A)}{n(C)}$, and vice versa for type (B). Type (B) is always chosen to resolve conflicts involving property edges, because the update cost in that case is zero.

It is never advantageous to choose type (B) replication to resolve placement conflicts. A objects are placed in 1-to-1 correspondence with C objects, and the functional relationship implies there are more C objects than A objects. Thus more data must be replicated. Further, since A objects are replicated in both types, the same update frequencies apply. Thus type (B) will incur higher update costs.

We envision replication being used incrementally. First, a design is formulated with no replication, i.e. 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 highly retrieval oriented databases, replication is a useful technique for improving performance. In general, this may not be the case.

4.3. Labelling Algorithms

Because not all labellings are comparable, there may be many maximal labellings for the same schema. Rather

than generate all the possible maximal labellings for a given schema, usage information can be use 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 labelling that specifies superior support for the access paths most heavily travelled. Labelling can be formulated in terms of a pure integer linear program which can be solved by standard branch and bound methods [TAHA75]. Because of the potentially high computational costs associated with this approach, a hill-climbing algorithm is presented which procedurally assigns labels to edges. The latter is suboptimal in that only a local optimum is found. Throughout this section, we shall limit our formulation so that path conflicts are avoided.

4.3.1. Integer Programming Formulation

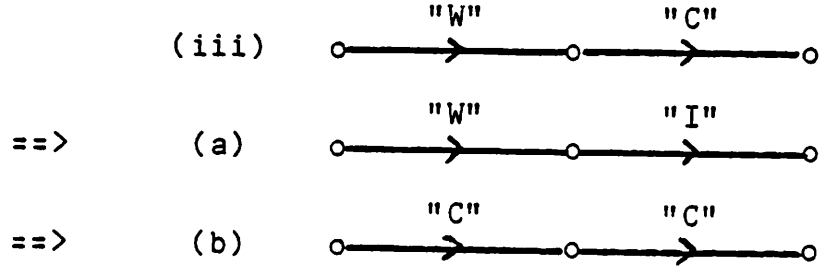
Labelling can be viewed as an assignment problem in which we assign the labels "W", "C", or "I" to each edge of the integrity schema subject to the constraints and the allowable number of placement or cluster conflicts that may be made during the label assignment, i.e. the degree of replication. The latter must be resolved using the techniques of the previous section.

The input to the algorithm is a graphical representation $G = (V,E)$ of the schema to be labelled, a ranking of

the edges (access mappings) according to frequency of traversal, and the degree of replication N. From this information, an integer linear program can be formulated.

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

The objective function maximizes a weighted sum of the frequencies of the edges labelled "W" or "C". A violation of a type (iii) constraint can be resolved in one of two ways:



We must be able to quantify the tradeoff between labelling an edge "W" or "C".

Let k_1 and k_2 be two constants which represent the tradeoff between accessing an object via an edge labelled "W" and one labelled "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

savings in processing time is considerable because 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 $\frac{k_1}{k_2}$ is the same as $\frac{\text{time to access a page + time to process a page}}{\text{time to process a page}}$. In general, $k_1 >> k_2$. In the following, we assume that $k_1 = 30$ and $k_2 = 1$.

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

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

Each of the labelling constraints can be reexpressed in terms of linear constraints in W_i and C_i. The first set insures that no edge may simultaneously be labelled "W" and "C":

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

The cluster constraints can be represented by a set of inequalities that make sure that at most one outedge of a given node may be labelled "W" or "C":

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

The placement constraints can be represented as a set of inequalities that makes sure that at most one inedge of a given node may be labelled "W":

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

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

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

At first glance, the number of constraints appears to be rather large, i.e. $|E| + 3|V|$. However constraints of the second and third type are only required if the node has more than one outedge or inedge respectively. Many nodes do not, e.g. 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, identifer edges can be ignored in the formulation, because such edges can always be labelled "W".

Also, the fourth type of constraint is a linear combination of the second and third types. This is only 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 non-negative integers, and we modify the constraints as follows:

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

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

for each $j \in V$,

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

The $X_{j,k}$ variables are used to 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:

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

such that

for each $i \in V$,

$$W_i + C_i \leq 1$$

for each $j \in V$ st. $|outedge(j)| > 1$,

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

for each $j \in V$ st. $|inedge(j)| > 1$,

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

for each $j \in V$,

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

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

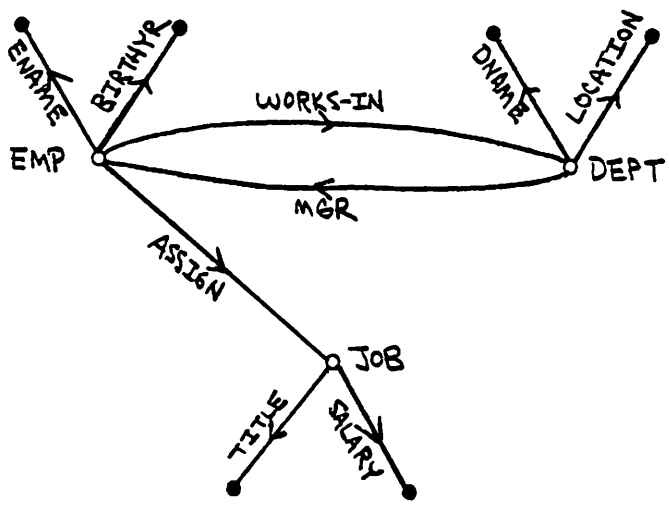
$$1 \geq C_i, W_i \geq 0, \text{ integer}$$

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

The optimal solution can be found by applying one of the many branch and bound algorithms developed for solving integer programming problems. However, further study may discover an algorithm which is particularly well-suited to solving the kinds of formulations found in practise.

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 non-negative and integer.

Our approach can be illustrated with an example. Consider the following subschema of the Employee database:



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

This ranking could have been derived from a set of user queries in conjunction with an indication of relative frequency, or simply specified by the designer. For a degree of replication = 0, the formulation becomes:

$$\text{Max } 30 \left[\sum_{i \in E} f_i W_i \right] + \sum_{j \in E} f_j C_j$$

st.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The program can easily be solved using Balas' algorithm for zero-one integer programs [TAHA75]. The resulting labelling is:

$$W_{WORKS\text{-}IN} = W_{TITLE} = 1$$

all others 0

WORKS-IN is the highest frequency edge. By labelling it "W", the outedges of DEPT and the other outedges of EMP are forced to be labelled "I". This allows the most frequently used outedge of JOB, i.e. TITLE, to be labelled "W".

If the degree is made 1, then the two highest frequency edges, WORKS-IN and ASSIGN, can both be labelled "W", even though this introduces a cluster conflict. All other edges are forced to be labelled "I" due to path and cluster constraints.

4.3.2. Suboptimal Labelling Algorithm

In general, the integer programming formulation of the previous subsection is a costly method for determining an optimal labelling. In this section we present another algorithm for labelling which is more procedural, but is not guaranteed to find an optimal solution. It differs from the previous algorithm mainly in that conflicts are resolved as soon as they occur.

The input to the algorithm is: a schema G = (V,E) to be labelled, a ranking of edges and their associated frequencies, update frequencies and object cardinalities for conflict resolution, and a limit L on the expansion factor to control replication. The algorithm has been formulated to always enforce path constraints.

The algorithm proceeds in two phases. The first phase produces a conflict-free labelling. It ignores all identifier edges, because these can always be labelled "W". Initially, all edges are labelled "I". The edges are visited in frequency order and are labelled "C" as long as the labelling does not cause a cluster conflict. Edges that are labelled "C" are again visited in frequency order and are labelled "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 longest 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 labelling, the entire path is examined, rather than each pair of conflicting edges. In the following, we show that paths can be resolved independently.

Lemma 4.1 - After phase 1, at most one path can pass through a given node.

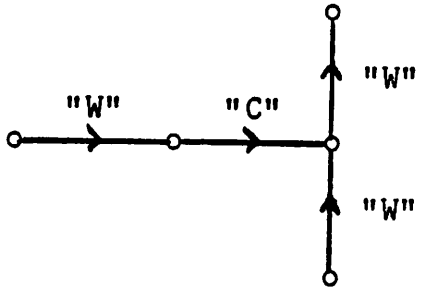
Proof: We must show that no node can appear as an internal node on more than one path. After phase 1, the schema is free of placement and cluster conflicts. Thus at most one outedge of any node is labelled "W" or "C" and at most one inedge is labelled "W". Then at most one path can pass

through a node.

Theorem 4.1 - Conflict-free labellings can be derived for paths independently.

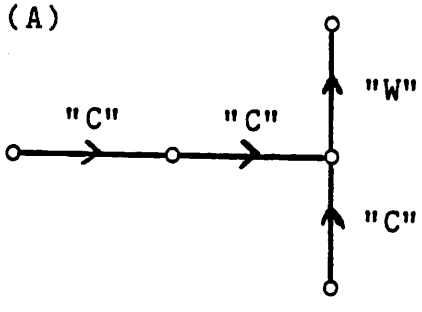
Proof: By lemma 4.1, it is not possible for two paths to cross. In that case, the order in which the paths are resolved can not affect the final labelling.

Lemma 4.1 does not exclude the case in which two paths form a "T" intersection as follows:

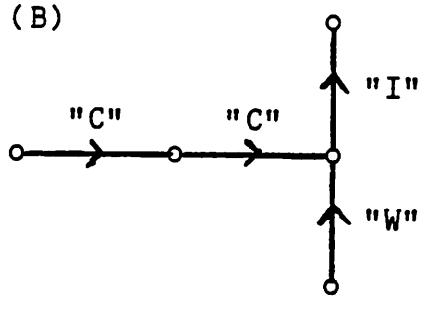


If the paths are relabelled independently, four final labellings are possible:

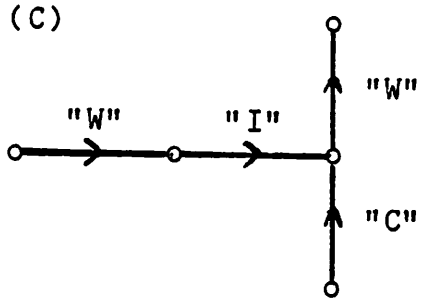
(A)



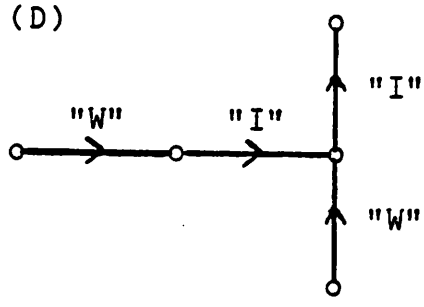
(B)



(C)

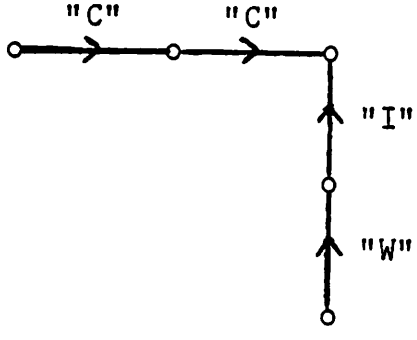


(D)



In the four situations above, the relabellings can take place along each path independently. A conflict-free labelling is guaranteed, and no edge can be improved without introducing a new conflict. QED.

However, in one case the labelling may not be maximal. Consider what labelling (B) would look like for a path intersection which forms an "L":



In this case the intersecting edge on the horizontal path can be improved by relabelling it "W" without introducing a new conflict. This situation can only happen for paths which intersect at a value node:

Lemma 4.2 - After phase 1 labelling, every non-value node has one outedge labelled at least "C" (i.e. the clustering

outedge).

Proof: Every non-value node must have at least one non-identifier outedge. Then the lemma is an obvious consequence of the cluster constraint. QED.

Thus, a path cannot terminate with a non-value node. We can show that the above situation is the only one in which edges can be improved without conflict after phase 1 and path resolution. The following lemmas are useful:

Lemma 4.3 - After phase 1 labelling and path resolution, no outedge labelled "I" can be improved without causing a new conflict.

Proof: By lemma 4.2, every node with outedges has one outedge labelled at least "C". Every other outedge must be labelled "I". Either a path passes through the node or it does not. If not, then the clustering outedge can never be relabelled "I". Thus no other outedge can be improved without causing a cluster conflict.

Suppose that the node is on a path. Then some inedge must be labelled "W" before path resolution. Afterwards, either the clustering outedge is relabelled "I" or the "W" inedge is relabelled "C". In the first case, an "I" outedge can not be improved without causing a path conflict. The second case is identical to the above. QED.

Lemma 4.4 - After phase 1 labelling, and before path resolution, if an inedge of a node is labelled "C" then some other inedge must be labelled "W".

Proof: The algorithm improves each "C" edge as long as the placement constraint is not violated. If an edge can not be improved, then there must be another inedge already labelled "W". QED.

We are now ready to present the main result:

Theorem 4.2 - After phase 1 labelling and path resolution, only the inedges of value nodes are candidates for improvement without introducing a new conflict.

Proof: We consider the cases for non-value nodes. There are three:

(1) A path passes through the node - By theorem 4.1, path resolution for connected nodes on different paths can be performed independently. Assume that these have been resolved. By lemma 4.3, no edges labelled "I" before path resolution can be improved. Because the node is on a path, one inedge is labelled "W". After path resolution, if this inedge is relabelled "C", then the clustering outedge is still labelled at least "C". Therefore, no "C" inedges can be can be improved without violating a path constraint. If the inedge remains "W", then no "C" inedge can be relabelled "W" without causing a placement conflict. Thus, no

inedges or outedges can be improved.

(2) The node is the source of a path - The node can have no inedge labelled "W". By lemma 4.4, it can have no inedge labelled "C" either. Then by lemma 4.3, no inedges or outedges can be improved, because all such edges, except for the clustering outedge, are labelled "I".

(3) The node is not on a path - The same argument as in case (2) applies. QED.

To resolve a path conflict, the path must first be detected. Lemma 4.1 guarantees that at most one path passes through any node. If a node is on a path, i.e. it has an inedge labelled "W" and an outedge labelled "W" or "C", then the path is extended in either direction until the longest path of conflicting edges is found. Because an edge can only appear on a single path, once an edge has been associated with a path, it is excluded from further consideration. Thus it is sufficient to visit each node in the schema to unambiguously determine all paths.

The next step is to generate all conflict-free labellings for each path. The labelling that contributes the most to the objective function is chosen as the final labelling. These can be generated recursively, starting with paths of two edges. The possible resolved labellings are [W,I] and [C,W] (we assume that the path and its

labelling are written from left to right). To extend these to higher degree, the last label in each previously generated labelling is examined. If the label is "I", the new labelling becomes [...,I,W]. If it ends in "W", two new labellings are added: [...,W,I] and [...,C,W]. Note that it may be the case that the very last edge in a path is labelled "C". Then the final new labelling would be [...,C,C]. In this way, all path conflict-free labellings of a given degree are derived. It is possible to generate these beforehand for common path lengths.

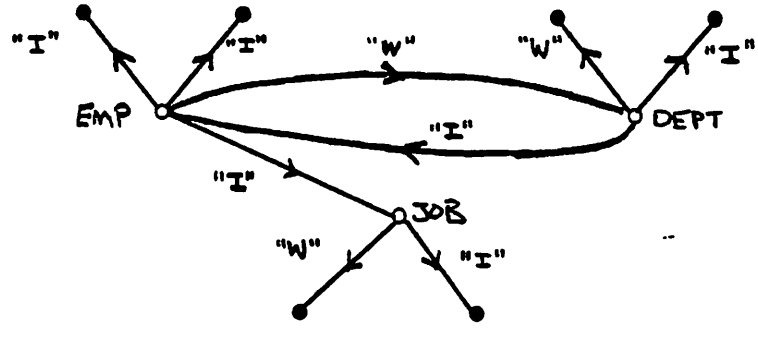
The second phase introduces replication and relabels those edges which can be improved without introducing new conflicts. Each edge is visited in frequency order. If it is labelled "I", it is temporarily relabelled "C". The relabelling stands as long as (1) no new conflicts are introduced, (2) a cluster 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 insure that the labelling does not oscillate. The cluster conflict is immediately resolved by choosing one of the replication methods of the previous section, given the update frequencies and object cardinalities.

Next, the edge is relabelled "W". This relabelling 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. The complete algorithm appears in figure 4.1.

By performing conflict and placement resolution "on the fly," we guarantee that no node can appear on more than one path during path resolution. Theorem 4.1 is still valid after phase 2.

The algorithm will be illustrated with the example of the previous subsection. It proceeds by first labelling WORKS-IN with "C". ASSIGN, ENAME, and BIRTHYR are forced to remain labelled "I" due to cluster constraints. TITLE is labelled "C", causing SALARY to be labelled "I". Finally, DNAME is labelled "C". All edges labelled "C" can be relabelled "W" because the schema can not have a placement conflict. The schema before path resolution is:



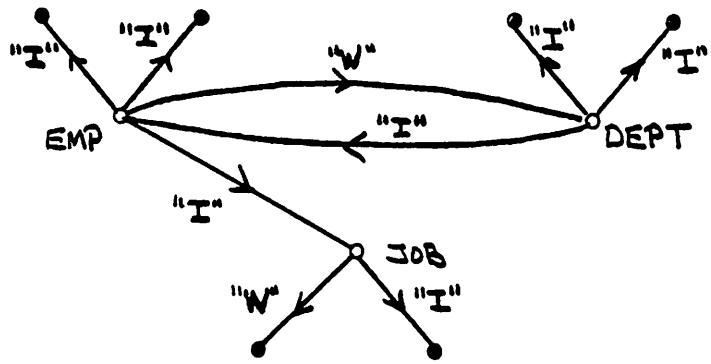
```
LABELSCHEMA:
   /* Phase 1 Labelling */
   all edges are initially labelled "I"
   FOR EACH edge in frequency order DO
      IF no cluster conflict THEN label <- "C"
   FOR EACH edge labelled "C" in frequency order DO
      IF no placement conflict THEN label <- "W"

   /* Path Resolution */
   RESOLVEPATHS

   /* Phase 2 Labelling */
   FOR EACH edge in frequency order DO
      IF edge is labelled "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 labelled "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 <- GETPATH(n)
      IF L ≠ ϕ THEN
         GENLABELS(length(L), labels)
         evaluate each l ∈ labels
         maxlabel <- maximum cost l
         relabel the schema using maxlabel
```

```
GETPATH(n,L):
   /* Find the path thru node n and place its edges in L.
      L is a sequence and || is the concatenation operator.
      Once an edge has been visited, it is removed from E. */
   L <- ø

          "W"    "W","C"
   IF o--->---o--->---o THEN
      i <- n
      WHILE there is an outedge(i) ∈ E labelled "W" DO
         L <- L || {outedge(i)}
         E <- E - {outedge(i)}
         i <- range(outedge(i))
      IF there is an outedge(i) ∈ E labelled "C" DO
         L <- L || {outedge(i)}
         E <- E - {outedge(i)}
      j <- n
      WHILE there is an inedge(j) ∈ E labelled "W" DO
         L <- L || {inedge(j)}
         E <- E - {inedge(j)}
         j <- domain(inedge(j))

GENLABS(n,labels):
   /* Generate all conflict-free path labellings of length n
      and store in labels */
   IF n = 2 THEN
      labels <- {[W,I],[C,W]}
   ELSE
      GENLAB(n-1,labels')
      FOR EACH l ∈ labels DO
         IF l[n-1] = "I" THEN
            l[n] <- "W"
            labels' <- labels' U {l}
         ELSE IF l[n-1] = "W" THEN
            l[n] <- "I"
            labels' <- labels' U {l}
            l[n-1] <- "C"
            l[n] <- schema's original label of n^th edge
            labels' <- labels' U {l}
      labels <- labels'
```

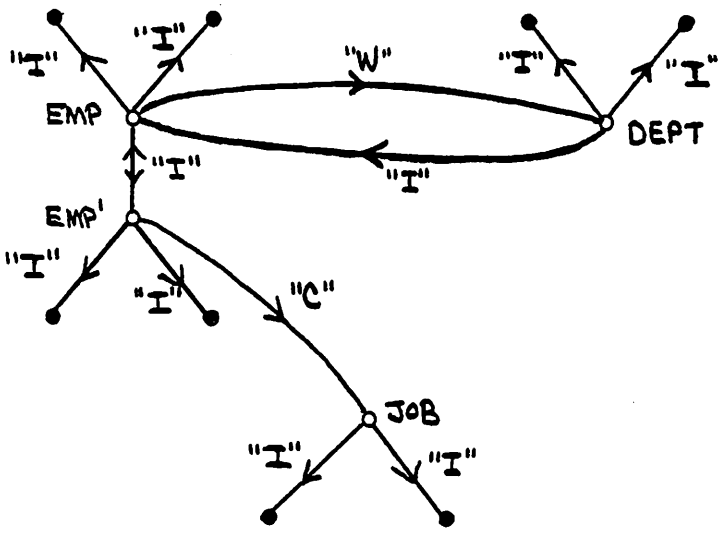
fig. 4.1 - Labelling a Schema

The path consisting of WORKS-IN and DNAME must be resolved. The labelling [W,I] maximizes the objective function and is chosen. The maximal, conflict-free schema

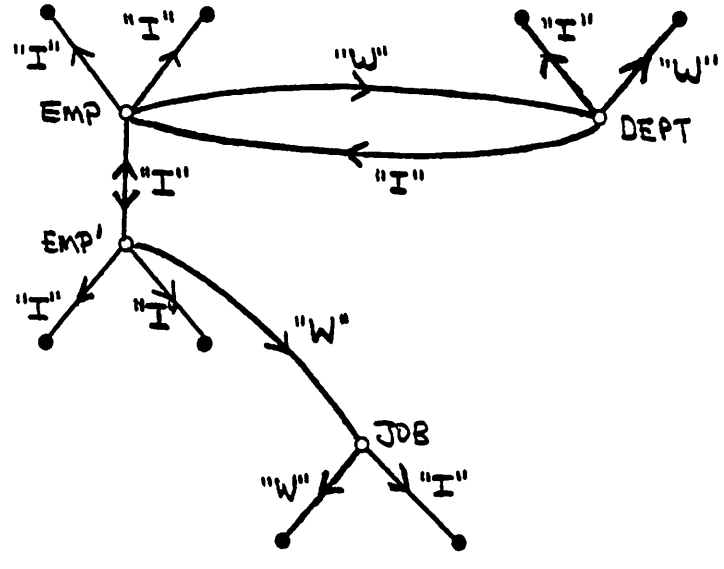
of degree 0 and expansion factor 1 is:



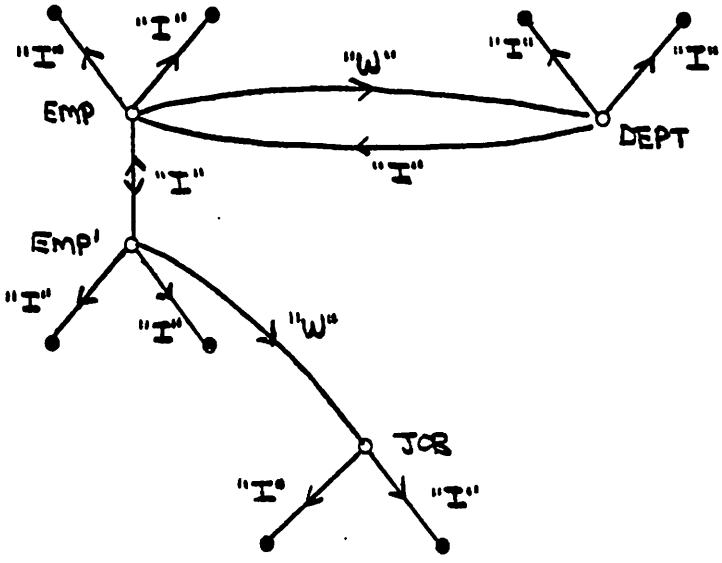
If the expansion factor limit is made greater than 1, then the algorithm will attempt to label ASSIGN with a "C", causing a cluster conflict with WORKS-IN. Assume that $U_{EMP} = 50$, $U_{JOB} = 10$, n(EMP) = 1000, and n(JOB) = 100. The type (A) is chosen because because $U_{EMP} < U_{JOB} * \frac{n(EMP)}{n(JOB)}$. 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 becomes:



ASSIGN can be immediately relabelled "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 relabelled "W" because it has not been previously involved in a path conflict. The schema before final path resolution is:



The labelling [W,I] maximizes the objective function for both paths. The final maximal, conflict-free labelling of degree 1 is:



4.4. Implementing a Schema

Up to this point, the design has been independent of the actual data model and system. In this section we briefly discuss the considerations involved in mapping a labelled schema into either relational or DBTG storage structures. We use the relational structures supported by INGRES [STON76] and those supported by the Data Structure Description Language of the 1978 CODASYL Journal of Development [CODA78].

The quality of the mapping depends on the detail of usage information specified. In the following, we assume that information has been specified at the level of the

previous section. All property mappings are at least "evaluated" by first replicating property values and then placing them together within the single record that represents the domain object. In the relational case, we are forced to assume that the schema is of degree 0 because of the difficulty in supporting replication in the relational model. Support for schemas which involve non-value node replication is illustrated in the CODASYL case.

4.4.1. Relational Implementation

INGRES provides three choices for the primary structure of a relation. It can be stored sequentially but unstructured (heap), stored sequentially and sorted and indexed on an attribute (ISAM), or hashed on an attribute. In the latter case, collisions are resolved by hashing to buckets. Thus tuples with the same attribute values are clustered. The secondary structure of a relation consists of optional indexes on the attributes.

Physical access paths between relations are not supported in INGRES. These are represented within the logical schema by foreign key attributes, i.e. attributes whose domain is the same as a key attribute of another relation. Thus the frequency of traversing the identifier mapping of an entity set must include the frequencies of all association mappings with the entity set in the range. This is represented by inedges in the integrity schema

graph. Further, interrelational placement is not supported. Tuples from different relations can not be placed near each other. Thus "W" edges and "C" edges are treated the same.

Due to the constraints of the labelling, at most one non-identifier outedge of a node can be labelled "W" or "C". Normally this edge should be chosen as the basis of the relation's primary structure. However, if the sum of the frequencies of all inedges (associations) plus the frequency of the identifier edge exceed that of the edge labelled "W" or "C", then the identifier is chosen. In any case, the chosen edge will represent either an association or a property. In the former case, it is represented within the relational schema by a foreign key attribute; in the latter, by a value attribute. We still must choose whether to base the primary structure on ISAM, which supports range access, or HASH, which supports only equality access. Information on how the access path is used will aid in making this determination. Identifier edges are frequently used in equijoin clauses of relational queries, thus HASH is the appropriate choice. Other property edges are frequently used in range clauses, e.g. AGE > 30, thus ISAM is the appropriate choice, although HASH should be used if equality access predominates. In the absence of such information, we always elect to choose ISAM struc-

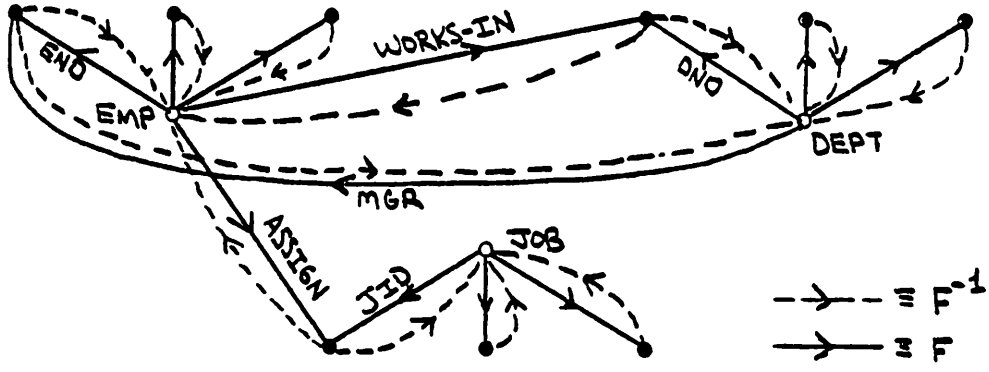
tures. Edges labelled "I" are supported by secondary indices. Rules to determine a relation's structure are given in figure 4.2.

Consider the maximal, conflict-free schema of degree 0 of the previous section. Applying the rules of figure 4.2 results in the following schema:

```
Algorithm RelationalPhysicalDesign
FOR EACH non-value node n ∈ V DO
   IF n represents an entity set THEN
      LET i = identifier outedge
      LET j = other outedge labelled "W" or "C"
      LET {k_1, ... , k_n} = inedge(n)

                  n
      IF f_i + Σ f_k_i > f_j
                  i
         THEN hash the relation on identifier attribute
         ELSE
            IF j is a property edge
               THEN isam the relation on value attribute
            ELSE IF j is an association edge
               THEN hash the relation on that foreign key
      FOR EACH outedge labelled "I" DO
         secondary index on that attribute

   ELSE /* node represents relationship */
      FOR outedge labelled "W" or "C"
         IF it is an association outedge
            THEN hash relation on that foreign key
            ELSE isam relation on the value attribute
      FOR EACH outedge labelled "I" DO
         secondary index on that attribute
```

fig. 4.2 - Relational Structure Choice

```
DEPT(DNO,DNAME,LOCATION)
   * hashed on DNO
   * indexed on DNAME, LOCATION

JOB(JNO,TITLE,SALARY)
   * hashed on JID
   * indexed on TITLE, SALARY

EMP(ENO,ENAME,BIRTHYR,WORKS-IN,ASSIGN)
   * hashed on WORKS-IN
   * indexed on ENO, ENAME, BIRTHYR, ASSIGN
```



The above access path schema represents maximum support for access mappings. Usage information may indicate that certain paths are not worth the overhead of supporting them.

INGRES does not provide mechanisms to support replicated data. Replication can be introduced by creating relations which are not in 4NF. For example, an EMP tuple can be extended with the attributes for the associated DEPT. However, without automatic support, it is not possible to conveniently propagate updates to the replicated data. We have elected not to allow replication for relational schemas, except for property value replication.

4.4.2. CODASYL Implementation

In the new CODASYL proposal [CODA78], many aspects of the physical database design have been removed from the schema DDL and localized in data storage definition. The DSDL provides facilities for the specification of 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 non-identifier outedge of a node may be labelled "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

property, then the record type is stored sequentially and sorted and indexed on the appropriate data item. Otherwise the outedge represents an association or a single-valued relationship, and the record type is clustered on the associated set. If "W" is specified, the records are placed near the owners. Indices are created for data items whose associated property mappings are labelled "I". The rules of figure 4.3 can be used to determine the record type's structure.

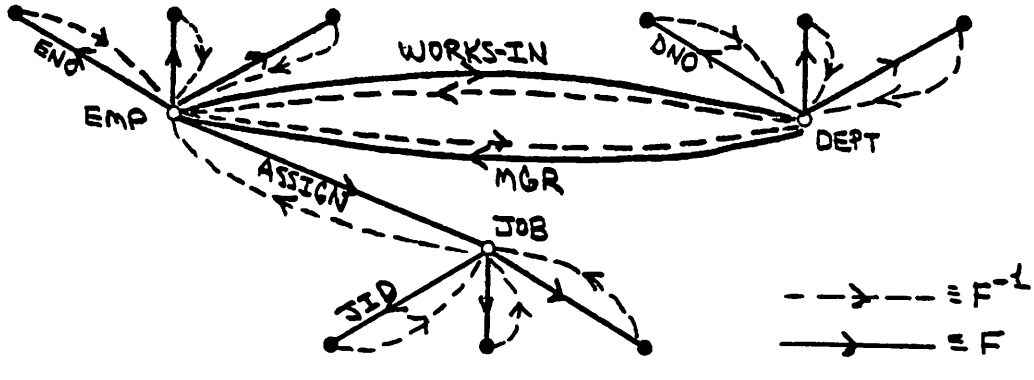
```
Algorithm CodasylPhysicalDesign
   FOR EACH non-value node ∈ V DO
      IF node represents an entity set THEN
         LET i = identifier outedge
         LET j = other outedge labelled "W" or "C"

         IF f_i > f_j
            THEN calc record type on key data item
            ELSE
               IF j is an association edge
                  THEN cluster record type on set membership
                     IF edge label is "W"
                        THEN place near owner
         FOR EACH property outedge labelled "I" DO
            index on that data item

      ELSE /* node represents a relationship */
         FOR outedge labelled "W" or "C"
            IF it is an association outedge
               THEN cluster record type on set membership
                  IF labelled "W" THEN place near owner
               ELSE sort and index on value data item
         FOR EACH property outedge labelled "I" DO
            index on that data item
```

fig. 4.3 - CODASYL Structure Choice

The DSDL specification for the degree 0 schema of the previous section would be:

```
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

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. The access schema for the above is:

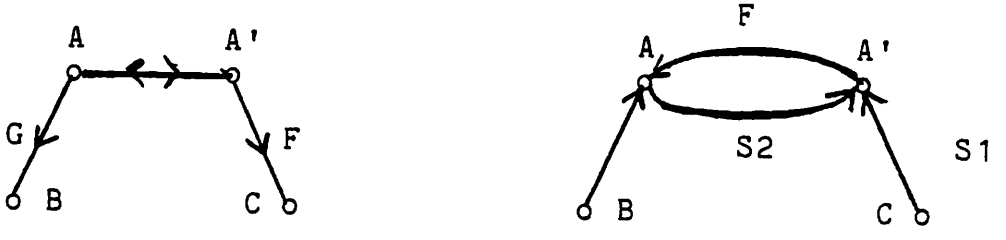


Again, all access mappings are maximally supported.

Support for paths that are infrequently travelled may be dropped at the choice of the designer.

Replication can be introduced by modifying the logical schema. Mechanisms exist to support the automatic propagation of update to all replicated objects. In each type of replication, a new record type is introduced along with new sets to implement the 1-to-1 edge. 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:



A' is made a total member of set S1. If C is deleted, then all associated A' are also deleted. A' is made a total member of S2. If A is deleted then its A' is also deleted. A is made a member of set F with A' as owner. A is a total member only if F was derived from an association. Thus a deletion of A' through S1 propagates to A. Because a cycle of associations is not allowed, the membership of A within F must have manual specified rather

than automatic. The following sequence is necessary to insert A into the schema:

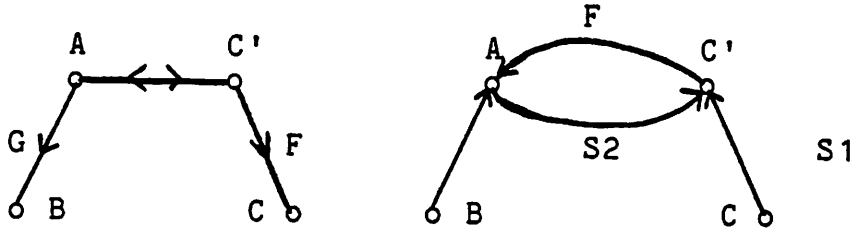
(1) STORE A [A is made a member of every set derived from one of its associations except for F]

(2) STORE A' [A' is connected to C via S1 and A via S2 automatically]

(3) CONNECT A TO F [A is connected to A' manually]

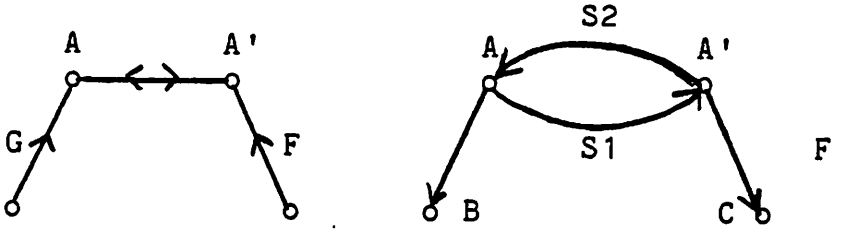
If F is an association, then (2) and (3) must be executed at the same time as (1). Otherwise they are executed when A first participates in the single-valued relationship.

The schema conversion for type (B) is similar:



The set membership options and insertion sequence are as before.

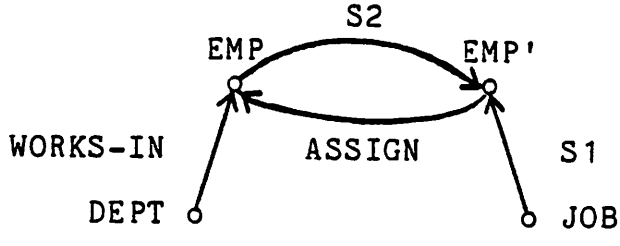
For type (A) placement resolution, the conversion is:



A' is a total member of S1. A is a total member of S2,

although manual must be specified. If F is an association, then the deletion of A will cause A' to be deleted via S1, which in turn causes C to be deleted. A must be stored first. Then A' is stored and A is connected to A' 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 of section 3.4.1 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 S1and placed near the owner JOB record.

4.5. Conclusions

In this chapter we have presented a characterization of storage structure in terms of implementation-independent properties: "evaluated", "indexed", "clustered", and "well-placed". We have used the characterization to specify an implementation-oriented physical design by assigning these properties to the logical access paths of the integrity schema. The implementation-oriented design represents the kind of support we would like to assign to each path.

Algorithms were presented to assign properties to paths in a conflict-free way, subject to certain constraints on the labelling. Replication was introduced explicitly and can be controlled by the database designer. It is used to achieve better retrieval time for access paths at some cost in update and storage overhead.

Finally, algorithms were given to realize an actual implementation from the implementation-oriented design for a specific database system.

CHAPTER 5

SCHEMA CONVERSION

5. Schema Conversion

5.1. Introduction

Schema conversion is a process of mapping a database schema between representations in different data models while preserving its semantic content. Conversion may also become necessary because of a change in the semantics modelled by the schema. The problems of constructing actual programs to perform the conversion, i.e., to access the data in one representation, manipulate it, and store it in a different representation, are not addressed. Schema conversion only tackles a subset of the total data translation problem.

In this chapter, rules will be formulated for mapping directly between database schemas that have been derived from the same design schema. This way, the preservation of semantics can be guaranteed. Further, the rules are formulated so that the design goals of chapter 3 are preserved. Rules are given for mapping between relational and DBTG schemas. The problems of database redesign and evolution are also addressed.

5.2. Derivation of Mapping Rules

Our logical design methodology is based upon formulating mapping rules to derive a target schema from a design model schema while preserving the design goals. If the design mapping rules could be inverted, it would be possible to map the target schema objects back into the design schema objects from which they were derived. Once these inverted mapping rules are determined, it is a simple matter to compose them with the design rules for a different target data model to arrive at rules to map directly between the two target models. This is the approach taken in this chapter, and is summarized in figure 5.1.

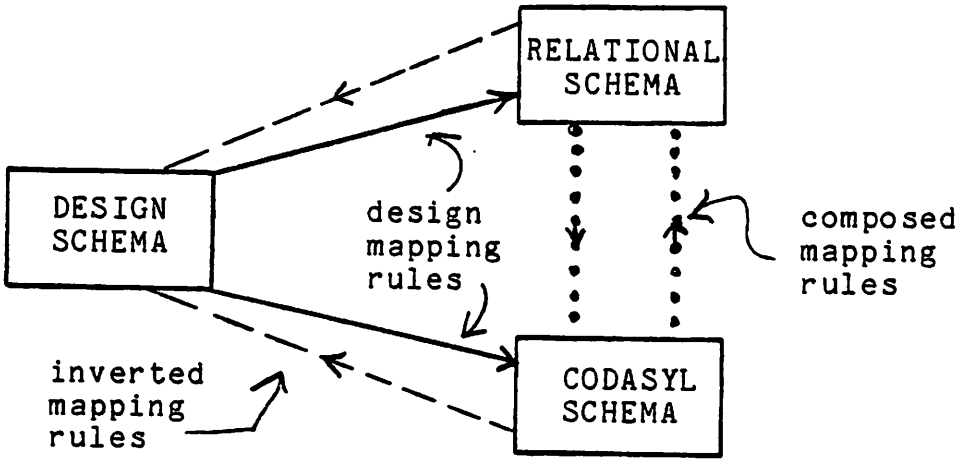


fig. 5.1 - Composed Mapping Rules

The inversion of the mapping rules is difficult because of the lack of semantic distinction in the relational and DBTG models. For example, it is not possible to determine, by simply looking at the relation as a collection of attributes over domains, which real world object is represented by the relation. This information can only be determined by enhancing the semantic content of the target schemas with augmented information.

The information we need for augmenting relational schemas is (1) which attributes are keys? (uniquely identify the tuple) and (2) which attributes are foreign keys? (values taken from the key domain of another relation). The former are called key attributes and the latter ID attributes. This information can be represented within the schema by functional dependencies and subset constraints. This approach was independently taken by [KLUG79]. The other attributes of the relation are called value attributes. The key of an entity set's relation is the attribute derived from its identifier property. The key of a relationship relation is the collection of foreign key attributes associated with the relations that represent the entity sets involved. The key of a single-valued relationship is the foreign key attribute associated with the single-valued entity set. Associated with each foreign key is the relation of which it is a key.

Given this semantic augmentation, how do we determine whether a relation represents an entity set, a single-valued relationship, or a general relationship? A general relationship relation (R-relation) has no single key attribute, only a collection of foreign key attributes and value attributes. A single-valued relationship (S-relation) consists of two foreign key attributes, one of which is also a key. The latter is a subset of the key of the single-valued entity set's relation. An entity set relation (E-relation) consists of one key attribute, which is not also a foreign key, and a collection of value attributes and foreign key attributes. This is summarized in figure 5.2.

The distinctions between objects in a DBTG schema derived by our mapping rules are not as ambiguous as in the relational case. However it is still desirable to

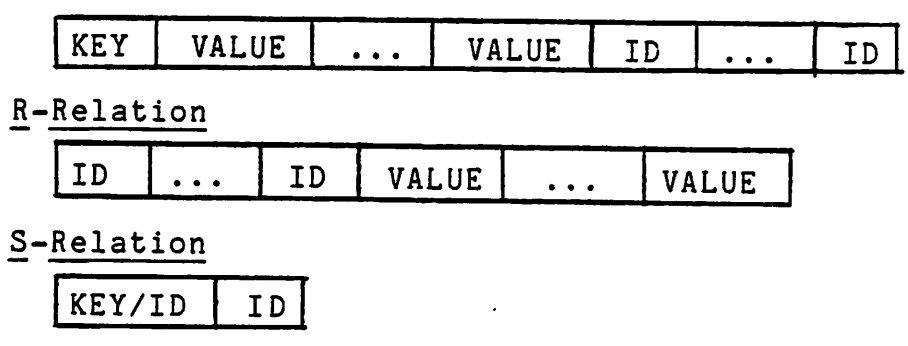


fig. 5.2 - Augmented Relation Types

make the distinctions explicit. In chapter 3, the concepts of self-identified and link records were introduced as well as total and partial set membership. These concepts can be used to identify design model constructs within a DBTG schema. Self-identified record types are derived from entity sets, and link record types from relationships. Associations are represented by sets with total members, and single-valued relationships by sets with partial members. The equivalences between objects in the augmented relational and DBTG schemas and the design model are summarized in figure 5.3.

Because of the 1-to-1 correspondence between the objects of the target schemas and the objects of the design schema, the mapping is straight-forward. The design

Design	DBTG	Relational
Entity Set	Record Type(SI)	Relation(E)
Identifier	Data Item(ID)	Key Attribute
Property	Data Item	Attribute(value)
Association	Set Type(Total)	Attribute(ID)
S.V. Relationship	Set Type(Partial)	Relation(S)
E_1 (S.V.)	Member	Key Attribute(ID)
E_2	Owner	Attribute(ID)
Relationship	Record Type(L)	Relation(R)
Property	Data Item	Attribute(value)
	+ Set Types(Total)	
$E_1, E_2, \ldots, E_n$	Owners	Attributes(ID)

fig. 5.3 - Equivalent Constructs in the Models

mapping rules for design schema objects are applied to the equivalent target schema objects to form the composed mapping rules.

Note that there is a strong resemblence between our view of schema conversion and the problem of creating relational views on top of CODASYL schemas [ZANI79a, ZANI79b]. For each object type in the CODASYL schema, there is an equivalent object type in a relational schema. The mapping of update operations against a relational schema into update operations against an underlying CODASYL schema derived from the same design schema is straight-forward:

Relational Operation	CODASYL Operation
APPEND tuple	STORE record
DELETE tuple	ERASE record
REPLACE value attribute	MODIFY data item
REPLACE ID attribute	MODIFY ONLY set MEMBERSHIP

Many more delete operations are needed for the relational schema because of the automatic support for associations in the CODASYL schema. Thus a sequence of DELETEs may be mapped into a single ERASE statement.

In general, the "view update problem" is much more difficult. It is made easier in our case because of the correspondence of object types and the augmentation of schemas with semantic information to facilitate the identification of the semantic objects.

5.3. Relational to CODASYL Mapping

In this section we formulate the rules for mapping an augmented relational schema to a DBTG schema. Associated with each identifier attribute is the E-relation from which its foreign key values are derived. The rules for mapping are:

(1) For each E-relation R define a self-identified record type r whose identifier is the key attribute of R. Each value attribute of R is a data item of r. The representation of the E-relations associated with the identifier attributes of R are owners of sets in which r is a total member.

(2) For each R-relation R define a link record type l. Each value attribute of R is a data item of l. The representation of the E-relations associated with the identifier attributes of R are owners of sets in which l is a total member.

(3) For each S-relation R, the representation of the E-relation associated with the key attribute of R is the partial member of a set owned by the representation of the E-relation associated with the identifier attribute of R.

Mapping rule (1) identifies an entity set, (2) a relationship, and (3) a single-valued relationship. Again,

the design goals are preserved: independent objects of the relational schema are mapped into independent objects of the DBTG schema, i.e. relations are mapped into record types, and all information about an object, represented by a single tuple, is mapped into a single record and its associated sets.

The conversion process can be illustrated with the relational schema of section 3.3.1. for the employee database. The augmented schema is:

E-Relations	Key	ID Attribute	Value Attribute
EMP	ENO(EMP)	WORKS-IN(DEPT) ASSIGN(JOB)	ENAME,BIRTHYR
DEPT	DNO(DEPT)	---	DNAME,LOCATION
JOB	JID(JOB)	---	TITLE,SALARY

R-Relations	ID Attribute	Value Attribute
QUAL	ENO(EMP),JID(JOB)	---
ALLOC	DNO(DEPT),JID(JOB)	NUMBER

S-Relations	Key	Foreign Key
MGR	DNO(DEPT)	ENO(EMP)

Rule (1) maps each E-Relation to a record type and a group of set types. The record type's data items include the value attributes and the key attributes of the E-relation. The set types represent the associations in which the entity set takes part as a domain. Graphically, the schema after applying rule 1 is given in figure 5.4.

Rule (2) maps each R-relation into a link record type and a group of set types. The link record contains data items derived from the value attributes of the R-relation.

fig. 5.4 - Schema After Rule (1)

The set types connect the link record type to the self-identified records that represent the entity sets involved in the relationship. The schema after applying (2) is shown in figure 5.5.

Finally, rule (3) maps each each S-relation into a set type that represents the single-valued relationship. The resulting schema is shown in figure 5.6.

5.4. CODASYL to Relational Mapping

In this section, rules will be proposed for mapping an augmented DBTG schema into a relational schema. Even though the semantics of the CODASYL schema are rich enough

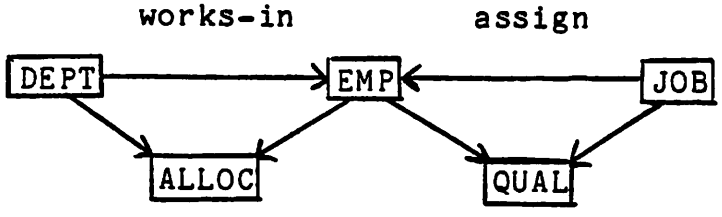


fig. 5.5 - Schema After Rule (2)

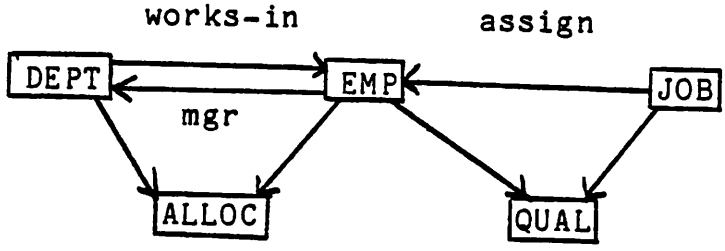


fig. 5.6 - Schema After Rule (3)

to allow us to determine from which design model objects a particular construct has been derived, we will assume that the schema has been augmented with the information.

For a self-identified record type, define its key to be its identifier. For a link record type, define its key to be the collection of the keys of the owners of all sets in which the link record is a member. For a DBTG schema obtained by using our mapping rules, link records can only be owned by self-identified record types, so that the definition of key is not circular. The rules for conversion are as follows:

(1) For each self-identified record type r define a relation R(r). Each data item of r is a attribute of R(r). The key of the owner of every set in which r is a total member is also a attribute of R(r).

(2) For each link record type l that is the member of two or more set types, define a relation R(l). The attri-

butes of R(l) consist of the data items of l plus the keys of the owners of the sets in which l is a member.

(3) For each set type s which has a partial member, define a binary relation R(s). The attributes of R(s) are the keys of the owner and the member of s.

In terms of the constructs of the design model, (1) identifies an entity set, (2) a relationship, and (3) a single-valued relationship. The design goals are preserved by mapping the independent objects of the CODASYL schema into independent objects in the relational schema and by mapping all information about an object, as represented in its record type and associated sets, into a single tuple of the relational schema. Of course, automatic support for the update semantics of associations and relationships has been forfeited. These must be enforced procedurally or through an integrity subsystem of the relational system.

There is a slight difference in the way self single-valued relationships are represented in the 1973 and 1978 CODASYL proposals. Rule (3) correctly handles schemas represented in either proposal. For the 1978 proposal, the rule is directly applied to a set with the same record type as owner and member. For the 1973 proposal, the key of the link record type is the same as the key of its

owner (see definition of key in the above). That SI-record type is a partial member of the set owned by the link record. Thus the keys of the owner and the member records are the same, as should be the case for a self single-valued relationship.

The conversion process can be illustrated with the CODASYL schema of section 3.3.2., for the employee database. The augmented schema is:

SI record types	Keys	Value Data Items
EMP	ENO	ENAME,BIRTHYR
DEPT	DNO	DNAME,LOCATION
JOB	JID	TITLE,SALARY

Link record types	Keys	Value Data Items
QUAL	ENO(EMP),JID(JOB)	---
ALLOC	DNO(DEPT),JID(JOB)	NUMBER

Total sets	member	owner
WORKS-IN	EMP	DEPT
ALLOC-DEPT	ALLOC	DEPT
ALLOC-JOB	ALLOC	JOB
QUAL-EMP	QUAL	EMP
QUAL-DEPT	QUAL	DEPT
ASSIGN	EMP	JOB

Partial Set	member	owner
MGR	DEPT	EMP

Rule (1) maps the DBTG representation of an entity set into its relational representation. Each SI-record type is mapped into an E-relation whose attributes are derived from the data items of the record type and the

keys of the owners of sets in which the record type is a total member. The schema after applying rule (1) is:

EMP(ENO, ENAME, BIRTHYR, WORKS-IN, ASSIGN)
DEPT(DNO, DNAME, LOCATION)
JOB(JID, TITLE, SALARY)

Rule (2) recognizes relationships and maps the link record type into an R-relation with the appropriate ID attributes to represent the E-relations involved. The additions to the schema after applying rule (2) are:

QUAL(ENO, JID)
ALLOC(DNO, JID, NUMBER)

Rule (3) handles single-valued relationships, mapping the partial set type into an S-relation. The final addition to the schema is:

MGR(DNO, ENO)

5.5. Schema Conversion Due to Database Changes

In this section, we discuss methods for handling the case in which schema conversion is necessary because the semantics of the real world have changed, e.g. a given entity set is no longer of interest. First we present mapping rules to convert an augmented relational or CODASYL schema into the design schema from which they have been derived. Then we discuss the admissible operations for redesigning a schema. The design mapping rules of chapter

3 can again be invoked to create a new schema for a target data model.

5.5.1. Mapping Into the Design Schema

First we give the rules for mapping from a relational schema to a design schema. Attribute names become the names of functions in the design schema, where the domain of the function is the object we are mapping and the range is the design model representation of the underlying domain in the case of a value attributes, or a relation in the case of an ID attribute. The rules are:

(1) For each E-relation R, define an entity set E whose identifier is the key attribute of R. Each value attribute of R becomes a property of E. The E-relations associated with ID attributes of R become the range of associations in which E is the domain.

(2) For each R-relation R, define a relationship R_D. Each value attribute of R becomes a property of R_D. R_D is defined over the entity sets derived from the E-relations associated with the ID attributes of R.

(3) For each S-relation R, the entity set derived from the E-relation associated with the attribute of R that is both the key of R and an ID attribute, becomes the domain of a single-valued relationship S. The range is the entity set derived from the non-key

ID attribute of R.

A similar set of rules can be derived for mapping an augmented CODASYL schema into a design schema. Again, data item and set names are used to derive the names of the design schema objects. The rules are:

(1) For each self-identified record type r, define an entity set E. Each data item of r becomes a property of E. Associations which involve E in the domain are derived from the sets in which r is a total member. The range entity set is derived from the owner record type of the set.

(2) For each link record type l that is a member of two or more set types, define a relationship R. The properties of R are derived from the data items of l. R is defined over the entity sets derived from the owners of sets in which l is a member.

(3) For each set type s which has a partial member, define a single-valued relationship S. The entity set derived from the member record type is the domain of the relationship (the single-valued entity set). The entity set derived from the owner record type (or its owner if it is a link) is the range of the relationship.

Again, in terms of the constructs of the design model, rule (1) identifies entity sets within the target schema, rule (2) relationships, and rule (3) single-valued relationships.

5.5.2. Redesign Operations

As a database evolves over time, the semantics it models may change as well. New real world objects and their interrelationships with existing objects may become of interest. In this subsection, we explore the kinds of semantic redesign operations that are supported by the design model.

We define a database redesign operation as the addition or removal of one of the following semantic object types:

(a) entity set
(b) property
(c) association
(d) single-valued relationship
(e) relationship

In addition, we define two evolution rules:

(a) association => single-valued relationship
(b) single-valued relationship => relationship

Each will be described in turn.

The addition of a new entity set to the schema causes no significant problems. All properties and associations with the new entity set in the domain must be specified at

the same time it is added. The deletion of an existing entity set is more complicated. All properties, associations, and relationships which involve the removed entity set must also be removed. The removal of an association may spawn additional redesign operations which can propagate through the schema.

A property which is not an identifier can be added or removed without side effect. The value set is specified at the time of property creation, and is removed with the property if it does not appear in the range of any other properties. An identifier can not be removed except through the removal of its associated entity set.

An association may only be added to a schema if it does not cause a violation of the insertion order. In particular, an association can not be added if it causes a cycle in the insertion order. To remove an association, we must also remove the entity set in its domain, because of the existance dependency that is modelled.

A single-valued relationship can be redesigned at any time without side effect. The entity sets must exist at the time of creation.

A general relationship can be added at any time, subject to the constraint that the entity sets over which it is defined must exist at the time of its creation. Similarly, it can be removed at any time, causing its

associated properties to be removed at the same time.

In addition to the redesign operations, there are two rules which deal with the evolution of relationships over time. An association is the most constrained type of relationship because it is required to be both functional and total. A single-valued relationship is less constrained, but still must meet the requirement of being functional. The general relationship is the least constrained.

We allow a relationship to become less constrained over time, but not vice versa. This is because of the inclusion of associations within single-valued relationships, and single-valued relationships within general binary relationships. The evolution can take place without loss of information.

For example, if the semantics of the employee database change so that employees can now exist outside of departments, then the association WORKS-IN can evolve into a single-valued relationship. No information is lost when performing this redesign, because at the point of applying the rule, every employee is associated with one department. If employees can now be assigned to more than one department, then WORKS-IN can evolve into a general relationship. WORKS-IN can not be "devolved" back into a functional relationship without the possibility of losing information from the database. For example, suppose that

an employee works in both the TOY and SHOE departments. To make WORKS-IN a function again requires a choice to be made for which department he currently works in. The choice can not be made unambiguously.

Once a schema has been redesigned and its relationships evolved, the design mapping rules must be reapplied to create a schema for a particular data model. Note that in the above, we have only been concerned with the structural changes that must be made to the database. We have not addressed the problem of how to construct programs to perform the data conversion. This has been widely studied elsewhere, e.g., [SHU 77].

5.6. Conclusions

In this chapter, we have shown how to derive schema conversion rules to map between relational and DBTG database schemas. This has followed naturally from the mapping rule approach to logical design explored in chapter 3. Rules can be formulated to map between additional models as long as the following assumptions hold: (1) each target schema was derived from a design schema by a mapping that preserves the design goals, and (2) the target schemas are augmented with semantic information where necessary to make it possible to determine which schema objects were derived from what types of design model objects.

We have also shown how to map target schemas into a design schema for the purpose of database redesign and evolution. The operations supported by the design model to redesign a database were defined.

CHAPTER 6

PROGRAM CONVERSION

6. Program Conversion

6.1. Introduction

In this chapter, we address the problem of how to convert the query language statements, embedded within an applications program, when the underlying database is translated to an equivalent schema for a data model with a different data sublanguage. We do not address the problem of modifying the program when the schema is altered due to database redesign and evolution. We concentrate on those changes that must be made because of a disparity in the level of procedurality of the source and target data sublanguages. The data manipulation language (DML) of the CODASYL model [COBO78] is "procedural" because a query is specified by describing how to navigate through a network of records, one record at a time. Non-procedural query languages for relational systems describe the data to be accessed in terms of a set-oriented specification (relational calculus). The detailed procedure used to access the data is not specified by the programmer, but is left for the system to determine.

In this chapter, algorithms are developed for mapping between DML and relational calculus queries embedded within a host language program. Decompilation maps between the procedural DML and the non-procedural relational calculus, while compilation maps in the other direction. In addition, not every meaningful sequence of DML operations can be converted into a relational calculus specification. Those programs that can be "decompiled" are characterized.

6.2. Query Specification

In this section, we briefly introduce the way queries are specified in CODASYL DML and the relational calculus. In the CODASYL model, a query is specified in terms of a sequence of FIND statements which identify the "current record of the database." In addition, that record can optionally become the current record of its record type, of its realm (logical partitioning of the database), and of any set in which it participates. The current record of the database can be deleted (ERASE), updated (MODIFY), or retrieved (GET). The program communicates with the database system via a user work area (UWA), which contains a record structure for each record type in the database schema. The current record of a record type can be transfered into the UWA by executing a GET. A new record is entered into the database by storing values for its

data items into the UWA and then executing a STORE operation. Values of data items in the UWA or in current records can be used as parameters in subsequent FIND statements. The current state of the database consists of all currency indicators (for each record type, set type, and realm), and the values of data items in the UWA. DML statements are embedded within the program as a syntactic extension of the host language.

In the relational model, the data accessed or updated is specified in terms of a predicate calculus-like qualification over a Cartesian product of relations. The qualified tuples are then projected to obtain the desired attributes. For a concrete syntax, we shall use the QUEL sublanguage, but restrict it to be aggregate-free [STON76]. For example, the query to find the department that employee Fred works in can be written as:

```
RANGE OF E IS EMP
RANGE OF D IS DEPT
RETRIEVE (D.DNAME) WHERE E.ENAME = 'Fred'
                   AND   E.WORKS-IN = D.DNO
```

E and D are called tuple variables and are shorthand names for specific relations. The list of attributes within parenthesis is called the "target list," and specifies the attributes to retrieve. The qualification following WHERE is called the "where clause." The Cartesian product EMP X DEPT is restricted to those tuples for which the qualifi-

cation is true. The qualified tuples are projected onto the attribute DNAME.

A relational calculus query is embedded in a program as follows. A query cursor is associated with each RETRIEVE statement. When OPENed, the query's results are computed. When SELECTed, the next tuple for processing is made ready for transfer into the user work area. When FETCHed, the data is actually transfered. Finally, the cursor must be CLOSEd before it can be reopened and the query reexecuted. A tuple id (TID) is associated with each tuple within a relation to encode its location within secondary storage. It is automatically set in the UWA as a side-effect of the SELECT operation. The interface is similar to the one defined for System-R [BLAS79], and although the data to be accessed is specified non-procedurally, the program still accesses a tuple at a time for processing.

6.3. Decompilation

Decompilation is the inverse of compilation; it is the process of grouping a sequence of procedural record at a time statements, which represent a plan to process a query, into a non-procedural set at a time specification. We are interested in decompiling CODASYL DML programs into relational calculus programs with a tuple a time interface.

The complete analysis of the applications portion of the program is an expensive and difficult process. Our approach requires a minimum amount of information about the host program: its control structure, the embedded sub-language statements, and the program's interaction with the database system via the user work area.

Decompilation proceeds in two phases. The first is analysis, during which the find operations of the program are grouped together into parameterized atomic access units. These units consist of sequences of DML and host language statements which correspond to a procedural enumeration of the set of objects in the range of an access mapping applied to a single argument, which is described by the parameter. The program's control structure is analyzed to determine which statements comprise a unit and how these units interact. This information is used to construct an access path expression graph which describes the access requests of the program in terms of composed access mappings. Two access mappings are composed if each object in the range of one mapping, the outer mapping, is an argument to the other, the inner mapping.

The second phase is embedding, in which the access expression is mapped into a relational query and interfaced with the original program. Two atomic access units can be combined if they are nested, and their associated

access mappings are composed. Further, no applications program statements can reside between the units. Each sequence of combined units corresponds to a sequence of edges in the access graph, which in turn can be mapped into a single relational query. A single access graph may have to be partitioned into several subgraphs if the associated atomic access units cannot be combined.

6.3.1. Analysis

6.3.1.1. Characterization of DML Operations

The difficult aspect of decompilation is determining which FIND statements constitute an atomic access unit. The object at a time FIND operations must be described in terms of the set oriented mappings of the access path schema. Cursors and object sets are introduced for this purpose. An object set is the complete set of objects potentially enumerated by a FIND operation, parameterized by the state of the database at a given point in time. For example, the object set associated with a FIND operation that accesses the records within a particular CODASYL set would consist of all member objects in the range of an inverse access mapping applied to the current owner object. The inverse mapping corresponds to the CODASYL set in the access path schema. The concept of an object set closely resembles the fan set of [DATE76]. A cursor is a

pair consisting of an index and an ordering specification, associated with a specific object set. The ordering specification is used to induce a sequence from the object set, and the index is the offset into this sequence. If the index is ever "out of range," it will take on the value zero. The cursor encodes the procedural aspects of the DML statement as it enumerates the elements of the object set. FIND operations that enumerate the same object set are candidates for forming an access unit.

The object set definition forms the bridge between the object oriented DML and the set oriented access mappings of the access path schema. It is the basis of the characterization of the CODASYL DML statements. Object set definitions are written in terms of the operations of the access path schema, and are parameterized by references to the current database state, i.e., UWA variable values; record, set, and realm currency indicators. An access expression is built by composing together an access mapping with the mapping that defines its parameters.

DML provides seven distinct ways of expressing a find operation. A skeletal object set definition is defined for six of these. Certain pieces of the definition are extracted from the statement. Others represent parameters which require further analysis before they can be determined. These are underlined. The objects in the skeleton

definitions are taken from the access path schema associated with the CODASYL schema. Their names are derived from the names of the corresponding records, data items, and sets in the CODASYL schema.

The type 1 find statement enumerates the instances of a record type which optionally match a specified UWA value for a data item or group of data items:

{FIRST, NEXT} record-name [USING data $item_1$, ...]

Object Set = { record-name $\in$ $RECORD\text{-}NAME^{-1}()$ $\sqcap$ DATA $ITEM_1^{-1}$(current UWA variable) $\sqcap$ ... }

cursor =

FIRST => c = <1, record type ordering specification>

NEXT => c = $<c_{index} + 1$, record type ordering specification>

Record-name and data $item_1$, ... are filled in from the find statement itself. For example, "FIND FIRST EMP USING BIRTHYR" requests that the current record become the employee record with a value for birthyr that matches the UWA. The object set definition is { emp $\in$ $EMP^{-1}()$ $\sqcap$ $BIRTHYR^{-1}$(emp.birthyr)}, i.e. the set of employee objects in the access mapping $BIRTHYR^{-1}$, parameterized by emp.birthyr. The cursor definition is <1, ordering specification>, where the specification is extracted from the description of the CODASYL schema, e.g., "ASCENDING ON

ENAME." The actual object accessed is the employee, born in the specified year, who appears first among the employees listed alphabetically.

In the definition of the NEXT cursor, c_{index} refers to the previous value of the cursor index. Thus the index of a NEXT cursor is one more than its previous value.

The type 2 find statement enumerates the record instances of a record type with a specified key data item value. It is also possible to access a record with the same key value as the one previously accessed:

{ANY, DUPLICATE} record-name USING key-name

Object set =

ANY => { record-name ∈ KEY DATA ITEM^{-1}(current UWA variable)}

DUPLICATE => { record-name ∈ KEY DATA ITEM^{-1}(KEY DATA ITEM(current record-name RECORD))}

Cursor =

ANY => c = <1, duplicate key ordering specification>

DUPLICATE => c = <c_{index} + 1, duplicate key ordering specification>

A request to find a record with a duplicate key value, results in a definition with a parameter for the key value from the current record of the record type. This kind of

information cannot be determined directly from the statement, but rather from how other statements interact with this one in terms of the program's control flow. In particular, we need to know the last statement executed which could have accessed a record of the specified type. The duplicate key ordering specification for the cursor definition is found in the schema definition.

The type 3 find statement will access a record within a set, that has the same selected data item values as the current record of that set:

DUPLICATE WITHIN set-name USING {data $item_1$, ...}

Object set = { member record name $\leftarrow$ SET $NAME^{-1}$(current owner record) $\cap$ DATA $ITEM_1^{-1}$(DATA $ITEM_1$(current member record))) $\cap$ ... }

Cursor =

c = $\langle c_{index} + 1$, set type ordering specification$\rangle$

The two parameters for the type 3 object set definition are for the current owner record of the set and the data item values from the current member record of the set. The set type ordering specification is extracted from the schema definition. The object set describes all member records of a set instance, identified by the current owner record, that have the specified data item values that

match those of the current member record.

The type 4 find statement is used to iterate over the members of a particular set instance:

{NEXT, PRIOR, FIRST, LAST, integer, variable} record-name WITHIN {set-name, realm-name}

Object set =

set-name => {record-name ∈ SET NAME^{-1}(current owner record)}

realm-name => {record-name ∈ REALM NAME^{-1}('realm name')}

Cursor =

NEXT => c = <c_{index} + 1, set/realm ordering specification>

PRIOR => c = <c_{index} - 1, set/realm ordering specification>

FIRST => c = < 1, set/realm ordering specification>

LAST => c = <max index, set/realm ordering specification>

integer => c = <c_{index} + integer, set/realm ordering specification>

variable => c = <c_{index} + variable, set/realm ordering specification>

The object set definition describes all the records in the current set instance or specified realm. The realm concept

must be simulated within the relational schema by appending to each relation an attribute which is defined over the domain of realm names. The ordering specifications are found in the appropriate places in the schema definition.

Type 5 find statements are not supported by decompilation. In this format, a record is accessed by providing the system with a database key, which is an encoding of the record's physical address. A database key is only guaranteed to reference the same record across a single execution of the program. This makes this type of statement unsuitable for the pre-execution analysis of the next section.

The type 6 find accesses the owner record of the current instance of a specified set:

OWNER WITHIN set-name

Object set = {owner record name ← SET NAME(current member record)}

Cursor = < 1, -- >

The object set contains the single owner record of the current member of the set instance. No ordering specification is needed for the cursor.

The type 7 find locates a record, within the current set instance of a specified set, which has data item

values that match those of the UWA:

record-name WITHIN set-name CURRENT [USING data item$_1$, ...]

Object set = { record-name $\leftarrow$ SET NAME^{-1}(current owner record) $\cap$ DATA ITEM$_1^{-1}$(current UWA variable) $\cap$ ... }

Cursor = < 1, set type ordering specification>

The parameters are the current owner record and the current values of UWA variables specified in the optional USING clause. The first record in the set ordering that simultaneously matches all the USING clause items is the one found.

The object set definition becomes more complicated if CURRENT is not specified. A SET SELECTION clause is invoked to specify a path of set instances. These trace through the schema to the record which is to become the owner of the current set. The set selection can be described in terms of access mappings, but because of its subtle side effects to currency information, we exclude it from consideration.

In the above, we have described the accessing operations of CODASYL in terms of sets formed from the access mappings of the associated access path schema. Some information cannot be filled in by simply examining the state-

ment itself. Further analysis of the execution dynamics of the program is required.

6.3.1.2. Flow of Currency Information Through a Program

In executing a find operation, a new record is made the current of the database and its record type, set type, and realm. Information about a current record may be used in subsequent find operations. This information is represented as parameters in the object set definitions defined in the previous subsection. In addition, data manipulation operations (MODIFY, GET, ERASE) operate on a current record.

We must understand how FIND operations interact, in terms of the execution dynamics of the program, to be able to group FIND operations into access units and then to compose the access units and their associated access mappings. Global data flow analysis [AHO 77, HECH77] is a technique used to determine the flow of information through a program. It is a process of pre-execution analysis which collects information about how "quantities" in a program are modified, preserved, and used. The quantities we are interested in involve the use and definition of currency information within the host language and data sublanguage statements of the program.

For analysis, we partition a program into blocks of statements for which there is no way to enter except at the first statement, and once entered, every statement must be executed sequentially. Information is collected for each block and then propagated to other blocks using the control flow of the program, as represented by the successor and predecessor blocks associated with each block.

We are interested in "reaching definitions." A currency is a currency indicator (one for the database, and for each record type, set type, and realm) or a UWA variable. A currency definition is a statement that can modify a currency, e.g., a FIND operation or an assignment statement to a variable in the UWA. A locally exposed definition is the last definition of a particular currency within a block. The set of all such definitions for a block x is called GEN(x). A definition of a currency c is killed by x if it reaches the top of block x and x contains a definition of c. The set of killed definitions is KILL(x).

The propagation of currency definitions through the program are described by the following two equations:

$$\text{For each block } x,\ IN(x) = \bigcup_{y \in PRED(x)} OUT(y)$$

$$\text{For each block } x,\ OUT(x) = [IN(x) - KILL(x)] \cup GEN(x)$$

where IN(x), OUT(x) are the sets of definitions that reach the tops and bottoms of block x respectively. Several methods have been developed to find solutions to these equations [AHO 77, HECH77].

Each FIND statement can create multiple definitions: one for each of the possible currencies associated with the found record. A GET operation will define a currency for each data item transfered into the UWA. Assignments to the UWA create a definition for those variables.

Besides the production of currencies, we are also interested in their usages. A currency is <u>used</u> if is referenced by a DML operation. For example, a type 1 find uses the current definition of data $item_1$. These usages are represented by underlined parameters within the object set definitions. A <u>locally exposed use</u> of a currency c is a use of c within x which is not preceded by a definition of c within x. The set of these uses is USES(x). By considering both IN(x), i.e., the definitions that reach a block, and USES(x) together, we can establish which definitions in the program can potentially be used by a specific operation.

These concepts can be illustrated in an example. Consider the "pseudo-COBOL" program which accesses the names of qualified accountants, shown in figure 6.1.

```
(1)    MOVE 'ACCOUNTANT' TO TITLE IN JOB.
(2)    FIND FIRST JOB USING TITLE.
(3)    FIND FIRST QUAL WITHIN QUAL-JOB.
   L.    IF END-OF-SET GO TO EXIT.
(4)        FIND OWNER WITHIN QUAL-EMP.
(5)        GET EMP.
           ...
(6)        FIND NEXT QUAL WITHIN QUAL-JOB.
           GO TO L.
   EXIT.
```

fig. 6.1 - Qualified Accountants

First, the find statements of the program can be characterized by their object set definitions:

(2) {job ∈ $TITLE^{-1}$(JOB.TITLE)}

(3) {qual ∈ $QUAL\text{-}JOB^{-1}$(current JOB)}

(4) {emp ∈ QUAL-EMP(current QUAL)}

(6) {qual ∈ $QUAL\text{-}JOB^{-1}$(current JOB)}

The program can be partitioned into four blocks. The sets of currency information after solving the data flow equations are shown in figure 6.2. The format <x,y> means that currency x is defined at statement y. A summary of definition and use information for each statement is:

(1) defines current TITLE
(2) defines current JOB; uses current TITLE
(3) defines current QUAL; uses current JOB
(4) defines current EMP; uses current QUAL
(5) defines current ENO,ENAME,BIRTHYR; uses current EMP
(6) defines current QUAL; uses current JOB

For example, the current QUAL used by statement (4) could

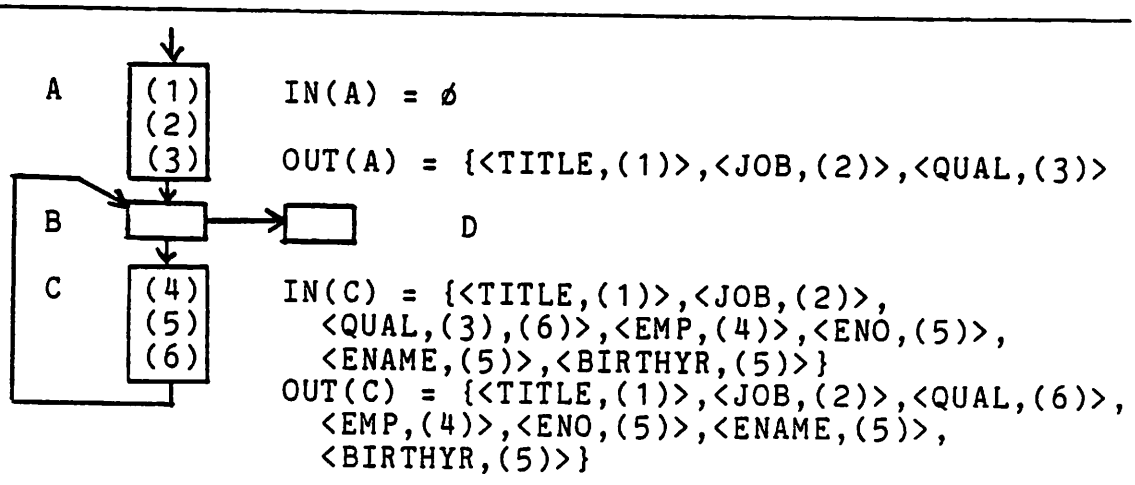


fig. 6.2 - Currency Flow

have been define by either statement (3) or (6), because those definitions of QUAL are found in IN(C), and no redefinition of QUAL appears in C before (4) is executed.

We define the currency flow graph to be G = (V,E), where V represents the statements of the program and E is a set of directed edges. An edge with tail at v_1 and head at v_2 is included in E if the statement represented by v_1 defines a currency used by the statement represented by v_2.

An algorithm to construct the currency flow graph is given in figure 6.3. Applying the above to the example results in the flow graph of figure 6.4.

Algorithm ConstructFlowGraph

```
FOR EACH Program Block x DO
   FOR EACH Statement j in x DO
      IF Statement j uses currency c THEN
         IF there is a local definition i of c that reaches j
            THEN add edge (i,j) to E
            ELSE
               FOR EACH definition i of c in IN(x) DO
                  add edge (i,j) to E
```

fig. 6.3 - Construct Flow Graph

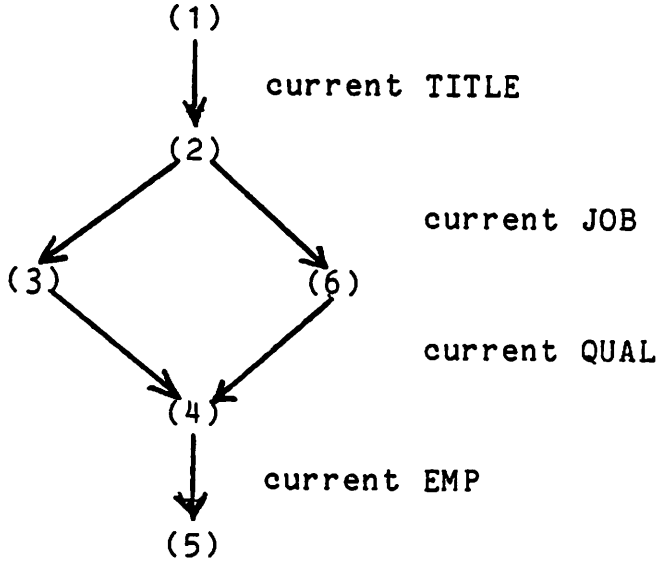


fig. 6.4 - Currency Flow Graph

In the next subsection, we examine how the currency flow graph is used to identify atomic access units and to construct an access path expression.

6.3.1.3. Formation of Access Path Expression Graph

The currency flow information of the previous subsection makes it possible to identify the object set parameters left unfilled. A difficulty is introduced whenever a parameter is ambiguously defined. For example, in the sample program, statements (3) and (6) provide the same currency information, i.e., the current QUAL record, used in statement (4). In general, an ambiguously defined parameter will cause decompilation to fail -- there is no unambiguous way to form an access path description for the query.

However, it is often the case that this ambiguity is caused by the way in which access mappings must be specified in DML. A pair of DML statements is needed to specify the enumeration of the records within a CODASYL set, i.e., to specify an F^{-1} access mapping. Consequently, each statement of the pair provides currency information for nested statements, which results in an ambiguity even though the pair corresponds to a single access operation. The statement pairs must be identified, and the currency flow graph modified, to eliminate the ambiguity. Three generic classes of statement pairs can be found. Associated with each class is a transformation to be applied to the currency flow graph to group the statements of the pair into a single node. If the transformation cannot be

applied, then the statements do not enumerate the same object set and cannot be used to form an atomic access unit.

A group of DML statements and host language statements which implement an access mapping are called atomic access units. The atomic access units for inverse and functional maps are shown in figures 6.5 and 6.6 respectively. Each access unit is parameterized by state information, such as record and data item currencies, which must be furnished by enclosing units. The contours drawn around the access units are henceforth called C-diagrams. Note that a functional access can often omit the NOT FOUND test if the access mapping is known to be an association. In that case the contour is drawn to the next enclosing C-diagram, or the end of the program, whichever comes first.

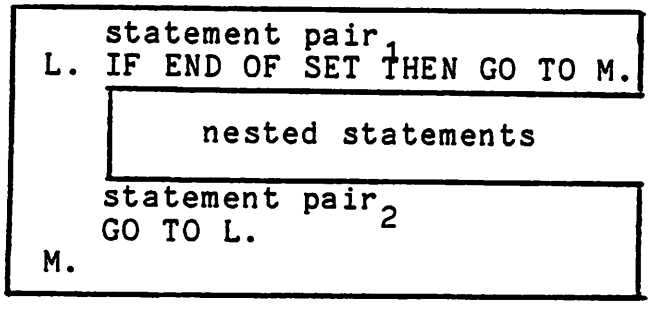


fig. 6.5 - Inverse Mapping Atomic Access Unit

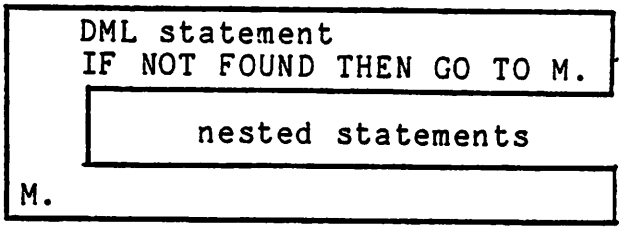


fig. 6.6 - Functional Mapping Atomic Access Unit

The three classes of statement pairs are:

(1) FIND FIRST/NEXT

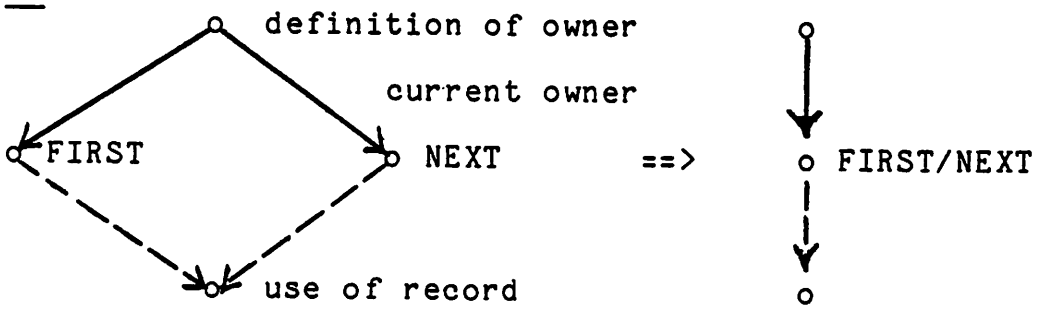
This class consists of the following three pairs of operations:

* FIND {FIRST, NEXT} record-name WITHIN set-name
* FIND {FIRST, NEXT} record-name WITHIN realm-name
* FIND {FIRST, NEXT} record-name [USING data $item_1$, ...]

The statements comprise an atomic access unit, and their nodes in the currency graph can be combined, if the following conditions hold: (1) each statement uses the same definition of an owner record currency (FIND WITHIN SET) or UWA data item currency (FIND USING), and (2) definitions from both statements reach all subsequent uses of this record currency. In other words, in terms of the flow of currency information through the program, the statements are completely identical, and they enumerate the

same object set. Schematically, the transformation can be represented as:

T1



("current owner" edge replaced by "current data item$_1$ value" for the last statement pair)

A dashed edge represents a subsequent use of a currency defined by the pair.

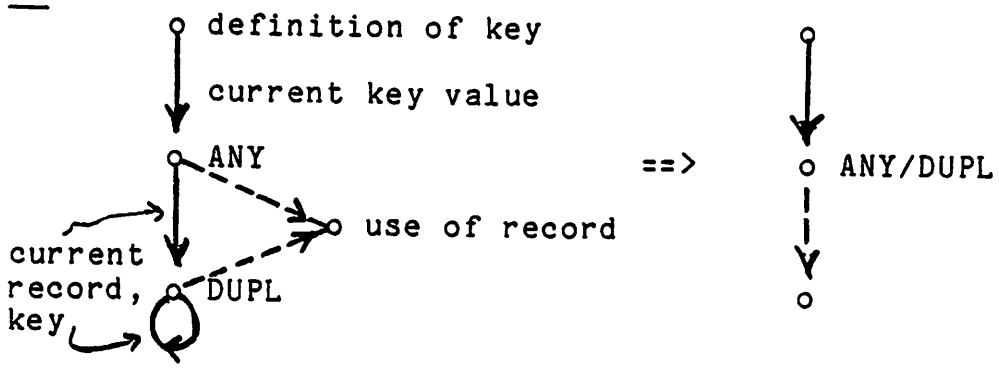
(2) FIND ANY/DUPLICATE

This class consists of the following find operation pair:

* FIND {ANY, DUPLICATE} record-name USING key-name

The statements form an access unit if: (1) the FIND ANY statement defines the record currency used by the FIND DUPLICATE statement (which also defines and uses its own record currency), (2) the key name is the same for both statements and the key data item definition that reaches the second statement is defined by the first, and (3) definitions from both statements reach all subsequent uses of the record currency. The associated transformation is:

T2



(3) FIND WITHIN/DUPLICATE

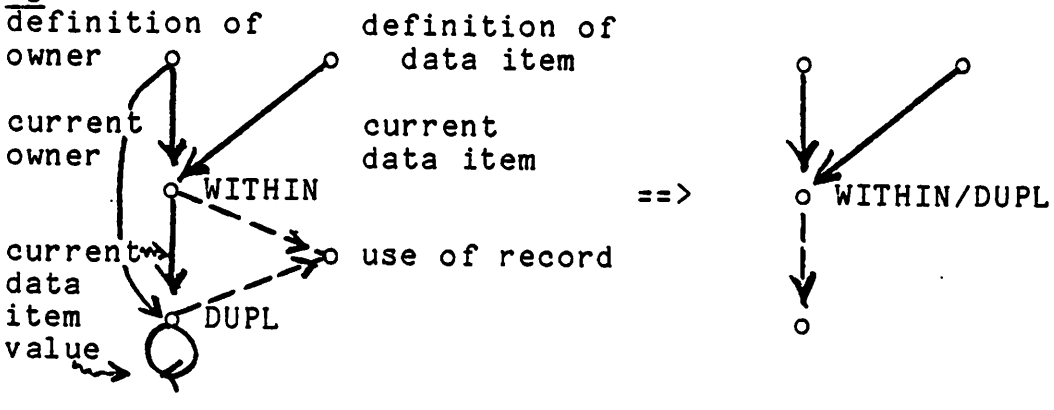
This class consists of the single operation pair consisting of the following two statements:

* FIND record-name WITHIN CURRENT set-name USING data item$_1$, ...

* FIND DUPLICATE WITHIN set-name USING data item$_1$, ...

These statements can be combined if: (1) they use the same definition of owner record, (2) the data items specified are identical, and (3) definitions from both statements reach all subsequent uses of the set's member record type. The transformation is:

T3



The sequence of data items in the USING clause must be the same for both statements.

The currency flow graph must be examined to determine whether the ambiguous definitions have been caused by one of the statement pairs. Ambiguous parameter definitions are determined by grouping together inedges of the same node which represent the same currency definition. Two nodes are candidates for combination if both provide identical currency information to subsequent nodes. In addition, the statements must conform to the format of an atomic access unit. Inedges that represent ambiguous definitions are grouped together and placed into a partition named by the nodes of which they are outedges. These edges are represented by the dashed lines in the transformations. If all the dashed edges do not appear in the same partition, i.e. do not emanate from the same pair of nodes, then the nodes cannot be combined and decompilation cannot proceed. The algorithm is given in figure 6.7.

In the sample program, the partition <(3),(6)> contains the edges (3)-->(4) and (6)-->(4) labelled "current QUAL." All other partitions are empty. Transformation T1 can be applied to nodes (3) and (6) to obtain the reduced flow graph of figure 6.8. Further, (3) and (6) are in the appropriate form for an inverse access unit.

```
Algorithm ApplyTransformations
   STEP 1: identify ambiguous parameters
      FOR EACH node DO
         IF inedges e1, e2 define the same currency THEN
            add e1, e2 to partition
               <tail(e1),tail(e2)>
   STEP 2: combine nodes
      FOR EACH partition <node1,node2> DO
        IF partition is not empty THEN
           IF class 1 THEN apply T1
           ELSE IF class 2 THEN apply T2
           ELSE IF class 3 THEN apply T3
           ELSE halt decompilation -- failure
```

fig. 6.7 - Apply Transformations

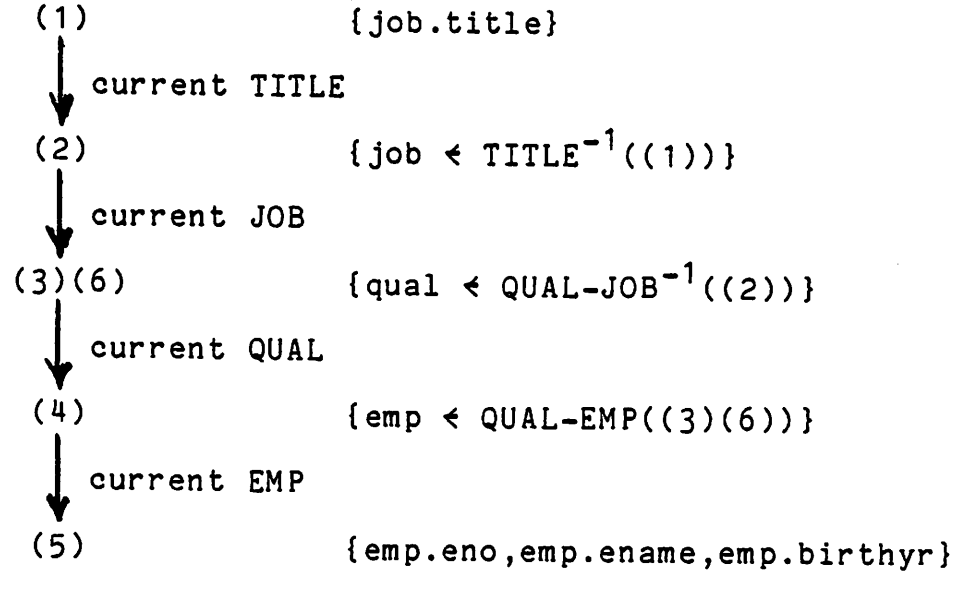


fig. 6.8 - Reduced Flow Graph

Associated with each (composite) node of the flow graph is an atomic access unit and a parameterized object set definition. The associated access unit enumerates the

elements of the object set definition and is parameterized by state information provided by other nodes and their access units. The access expression graph is constructed from the nodes and edges of the currency flow graph. The nodes are labelled with the object defined by the object set definition of the corresponding node of the currency flow graph. The edges are labelled with the access mapping in the object set definition whose parameter is defined at the tail of the edge. The graphical representation of the query is shown in figure 6.9.

The access path expression graph represents a description of the access mappings used to formulate a query. By representing the access requirements of the

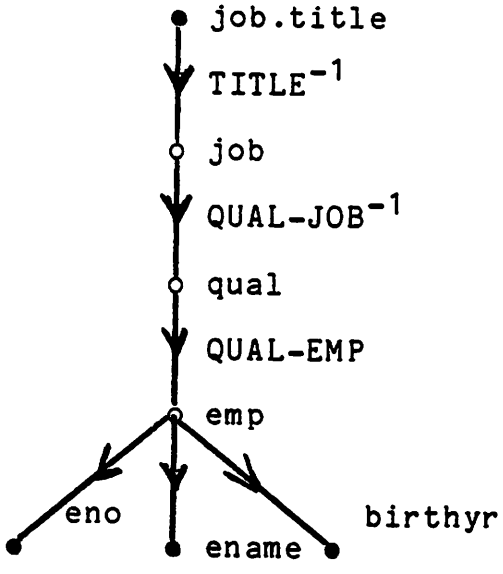


fig. 6.9 - Access Path Expression Graph

program in terms of the set at a time operations of the access path schema, rather than the object at a time operations of the DML, the translation into a relational calculus representation is much simplified.

6.3.2. Embedding

Once an access expression has been derived, it must be mapped into a relational query and interfaced with the host language program. If program instructions, including those to retrieve data, are potentially executed between nested access units, then the associated expression must be partitioned into subexpressions. Each of these are mapped into a relational query. In this subsection, algorithms are presented for determining where to "break" the access path expressions, how to map them into a relational calculus query, and how to embed a query into the original program.

6.3.2.1. Procedural Break Analysis

The currency flow analysis of the previous section is based solely on an examination of DML statements and the way they interact with respect to the program's control flow. An equivalent relational query makes all the data collected over a path available at a single time. If the CODASYL program accesses intermediate results, i.e., data which is a result of applying a subexpression of the

access expression, then this data must be made available to the program at the places where it is needed. The access expression must be partitioned to achieve this.

In addition, program instructions between accessing operations may induce a procedural break. Consider the following code skeleton.

```
   FIND FIRST EMP WITHIN WORKS-IN.
A. IF END-OF-SET  GO TO B.
      --- application code1 ---
      FIND FIRST QUAL WITHIN QUAL-EMP.
   C. IF END-OF-SET  GO TO D.
         --- application code2 ---
         FIND NEXT QUAL WITHIN QUAL-EMP.
         GO TO C.
   D. --- application code3 ---
      FIND NEXT EMP WITHIN WORKS-IN.
      GO TO A.
B.
```

$Code_1$ and $code_3$ are executed once for each employee, while $code_2$ is executed once for each employee by each qualified job. If the access mappings are composed, there is no way to combine their access units so that the code is executed the correct number of times. If $code_1$ and $code_3$ are not empty, then the expression must be partitioned.

As an example, consider the program that finds the departments to which accountants born after 1950 are assigned (see figure 6.10).

By following the steps of analysis given in the previous section, the access path expression graph of figure 6.11 is derived. However, the expression cannot be mapped

```
        MOVE 'ACCOUNTANT' TO TITLE IN JOB.
(1)     FIND FIRST JOB USING TITLE.
     L. IF NOT-FOUND GO TO EXIT.
(2)    FIND FIRST EMP WITHIN ASSIGN.
     M. IF END-OF-SET GO TO O.
        GET EMP.
        IF EMP.BIRTHYR <= 1950 GO TO N.
        <other code that accesses emp in UWA>
(3)     FIND OWNER WITHIN WORKS-IN.
        GET DEPT.
        ...
     N. FIND NEXT EMP WITHIN ASSIGN.
        GO TO M.
     O. FIND NEXT JOB USING TITLE.
        GO TO L.
  EXIT.
```

fig. 6.10 - "Young" Accountants

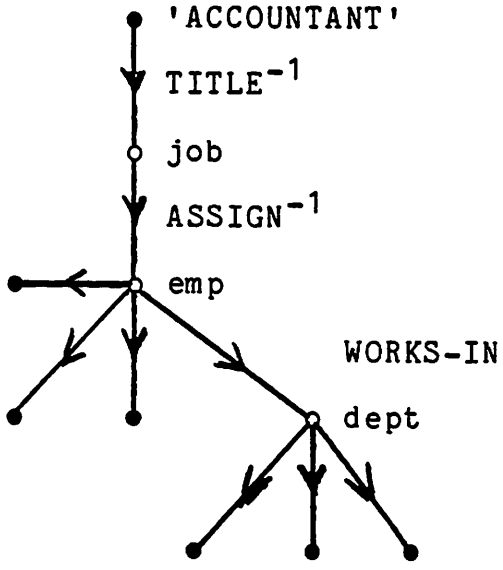


fig. 6.11 - Access Path Expression Graph for 6.10

directly into a relational query, because the program accesses emp data values for computation before the entire access expression has been completed.

Two composed access mappings of the access expression graph are <u>adjacent</u> if the access unit of the inner mapping is nested within the access unit of the outer mapping, and the only code that can intervene between these is associated with other atomic access units. No applications statements may appear. To map an access expression graph into a single relational query, every access mapping must be adjacent to its successor in the graph. If this is not the case, then the graph must be partitioned along the node that both edges share in common. A procedural break

is caused.

In the sample program, C-diagram (1) is associated with $TITLE^{-1}$, C-diagram (2) is associated with $ASSIGN^{-1}$, and C-diagram (3) is associated with WORKS-IN. $TITLE^{-1}$ and $ASSIGN^{-1}$ are adjacent because their C-diagrams of their access units are. However $ASSIGN^{-1}$ and WORKS-IN are not because of the program statements, including a GET statement, between their units. The access expression is partitioned along EMP (see figure 6.12). The segment on the left of the partitioning edge is called the predecessor segment, because it is executed first. The right segment is the successor.

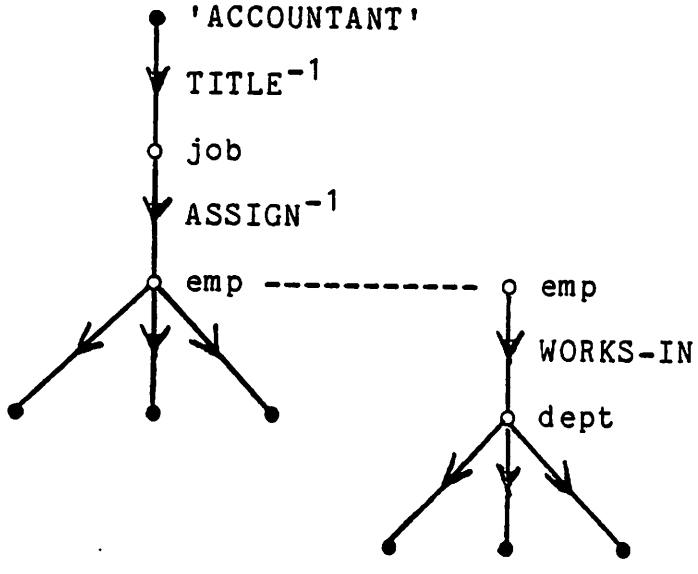


fig. 6.12 - Partitioned Access Path Expression

6.3.2.2. Formation of Relational Queries

A program may contain several queries, each of which is mapped into a possibly partitioned access path expression. The next step in decompilation is to derive a relational calculus query from each expression.

A linkage term is a clause of the relational qualification, i.e. the WHERE clause, that represents how interobject relationships are implemented within the relational schema. For example, the association WORKS-IN is represented by a foreign key attribute WORKS-IN of the EMP relation. The mapping is represented by including the linkage term E.WORKS-IN = D.DNO. All edges of the access path expression, except for those that represent property functions, will be mapped into a linkage term.

The mapping between edges and linkage terms is:

(1) property inverse: If P: A --> B, then P^{-1}(value) is mapped into the linkage term "range variable$_A$.P = value". For example, if J is the range variable for JOB, then $TITLE^{-1}$('ACCOUNTANT') is mapped into J.TITLE = 'ACCOUNTANT'.

(2) association: If F: A --> B, then F^{-1}(B) or F(A) is mapped into the linkage term "range variable$_A$.F = range variable$_B$.ID$_B$". For example, if E is the range variable of EMP, then ASSIGN(E) or $ASSIGN^{-1}$(J) is

mapped into E.ASSIGN = J.JNO.

(3) relationship association: If F: R --> A, then $F^{-1}(A)$ or F(R) is mapped into the linkage term "range $variable_R.ID_A$ = range $variable_A.ID_A$". For example, if Q is the range variable of QUAL, then QUAL-EMP(Q) or $QUAL\text{-}EMP^{-1}(E)$ is mapped into Q.ENO = E.ENO.

(4) single-valued relationship: If S: A --> B, then $S^{-1}(B)$ or S(A) is mapped into the linkage term "range $variable_B.ID_B$ = range $variable_S.ID_B$ AND range $variable_B.ID_A$ = range $variable_S.ID_A$". For example, if M is the range variable for MGR, then MGR(E) or $MGR^{-1}(D)$ is mapped into E.ENO = M.ENO AND D.DNO = M.DNO.

The algorithm for mapping an expression graph into a relational query is:

Algorithm CreateRelationalQuery

For each partitioned segment, perform the following mappings:

(1) Associate a range variable with each non-value vertex of the graph.

(2) Value vertices with a property mapping inedge appear in the target list as relational attributes derived from the label of the inedge.

(3) Property inverse edges or edges which are derived from property mappings become linkage terms in the qualification, depending upon the type of interrelationship represented.

(4) For a partitioned edge, append the following clause to the qualification of the successor query: "range variable$_{LEAF}$.TID = leaf record.TID", i.e., use the TID to reaccess the previous tuple.

Examples:

Access Path Expression of figure 6.9:

```
RANGE OF J IS JOB
RANGE OF Q IS QUAL
RANGE OF E IS EMP
RETRIEVE (E.ENO, E.ENAME, E.BIRTHYR)
WHERE J.TITLE = job.title
AND   Q.JID = J.JID
AND   Q.ENO = E.ENO
```

Access Path Expression of figure 6.12:

```
RANGE OF J IS JOB
RANGE OF E IS EMP
RETRIEVE (E.ENO, E.ENAME, E.BIRTHYR)
WHERE J.TITLE = job.title
AND   E.ASSIGN = J.JID

RANGE OF E IS EMP
RANGE OF D IS DEPT
RETRIEVE (D.DNO, D.DNAME, D.LOCATION)
WHERE E.TID = emp.tid
AND   E.WORKS-IN = D.DNO
```

6.3.2.3. Query Embedding

The final step of decompilation is to interface the relational query with the original host program. An important objective of embedding is to determine which DML statements and their associated control structures, e.g., tests for NOT FOUND or END OF SET, have been rendered superfluous by decompilation. The approach is based on collapsing together the access operations of the program, as represented by the access path expression graphs and their access units, by starting with the innermost nested operations and working outwards. In terms of the access path expression, this means starting with the root and working towards the leaves.

We present four program transformations for the cases (1) $f_1 \circ f_2$, (2) $f_1^{-1} \circ f_2^{-1}$, (3) $f_1 \circ f_2^{-1}$, and (4) $f_1^{-1} \circ f_2$. The transformations are repeatedly applied until there are no more operations to compose.

The first case involves the composition of two functional mappings. Note that an inverse function that is not implemented by a DML statement pair is considered to be a function for the purposes of embedding. Statements which appear on the left of the transformation, but not on the right, are considered to be deleted from the program. The transformation is given in figure 6.13. Note that the composed mapping can be treated as a functional access in

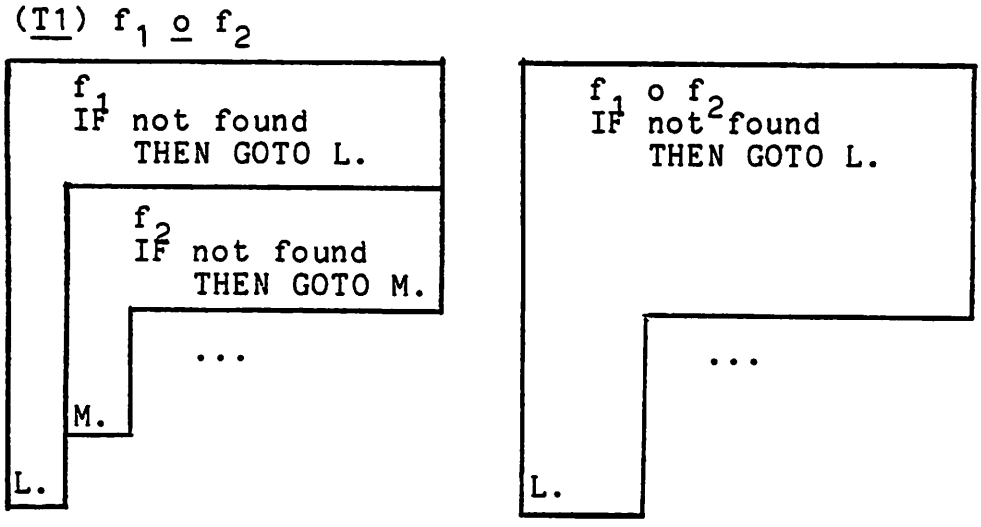


fig. 6.13 - Transformation T1

subsequent applications of transformations.

The second case deals with the composition of two inverse mappings. The transformation is shown in figure 6.14. The resulting flow chart can be treated as an inverse mapping for subsequent transformations.

The third case handles the composition of a function and an inverse function and is given in figure 6.15. The result can subsequently be treated as an inverse mapping.

The final case is the inverse of the above: composition of an inverse mapping and a functional mapping. The transform is shown in figure 6.16.

Once the access units have been combined as much as possible for a single expression graph, it is a simple

(T2) $f_1^{-1} \underline{o} f_2^{-1}$

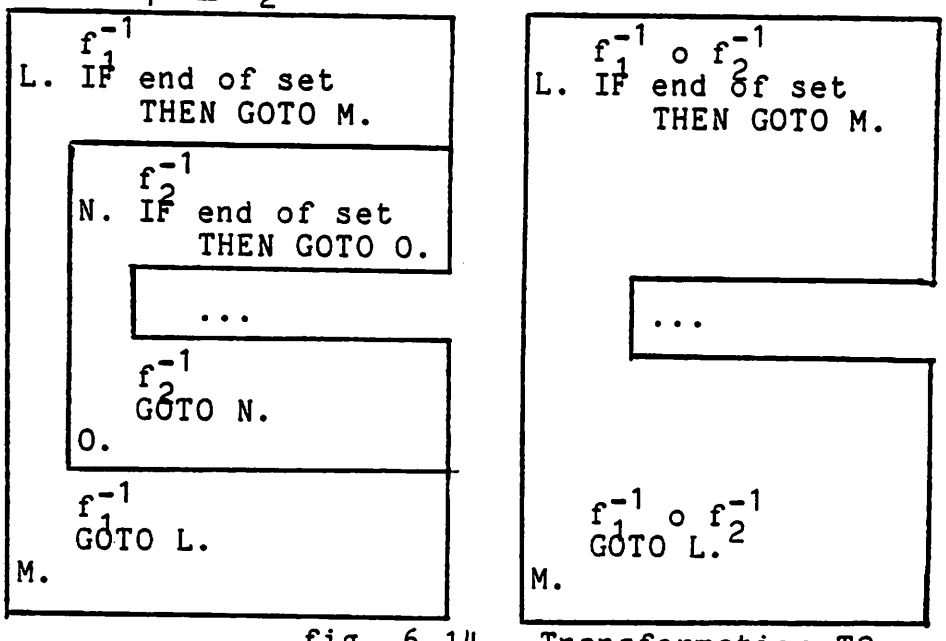


fig. 6.14 - Transformation T2

(T3) $f_1 \underline{o} f_2^{-1}$

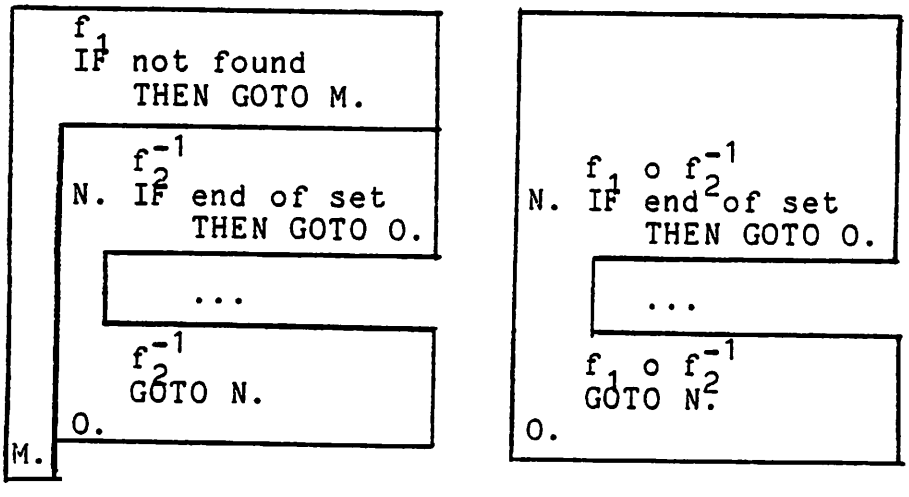


fig. 6.15 - Transformation T3

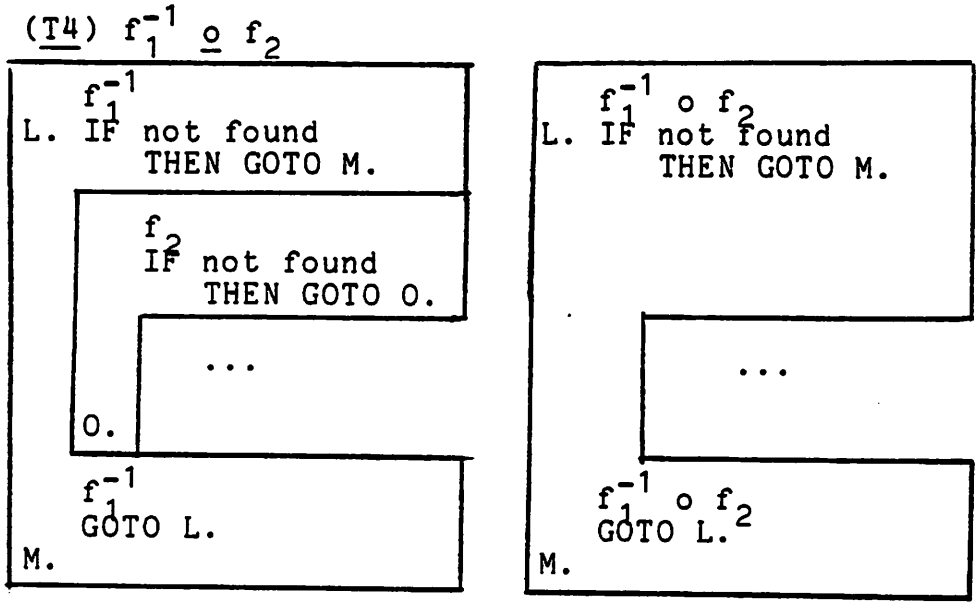


fig. 6.16 - Transformation T4

matter to locate where to embed the open and close statements (see figure 6.17). OPEN is embedded just before the first access of the expression and CLOSE is embedded just upon exit. Data accesses become SELECT statements, and get statements are replaced by FETCHes. In the latter, all GETs for data collected over the access expression must be clustered together so they can be replaced by a single FETCH statement.

The algorithm for embedding proceeds in three steps:

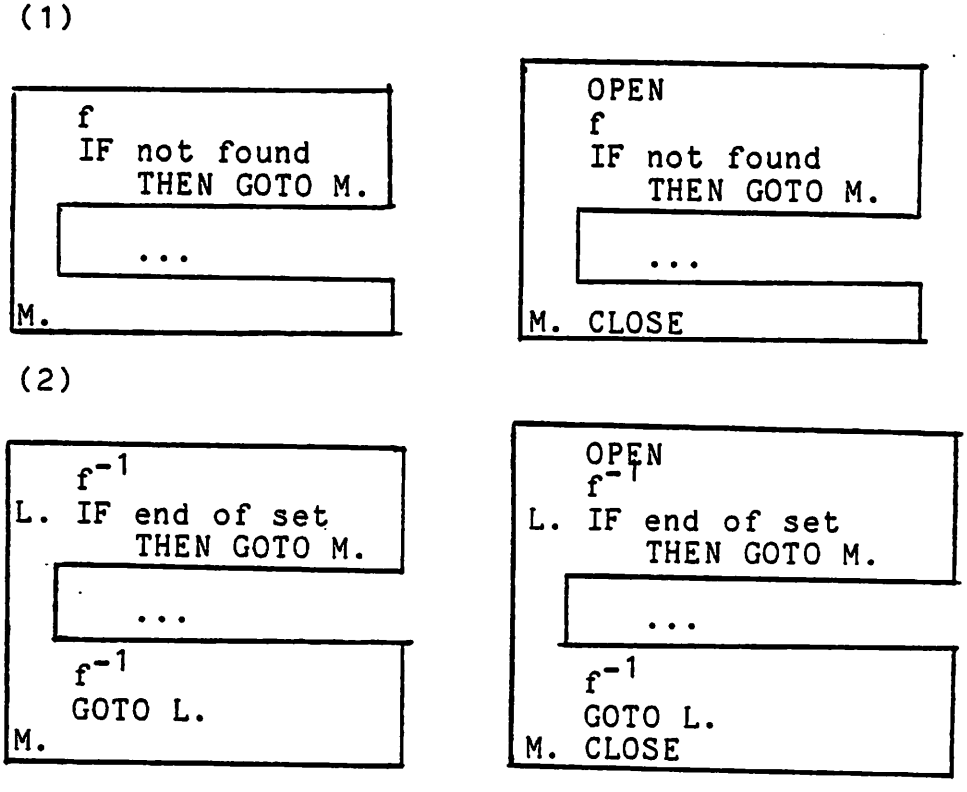


fig. 6.17 - Open/Close Embeddings

Algorithm Embed

Step 1: Create flow chart

Step 2: FOR EACH access path expression graph DO
FOR EACH mapping pair, starting with the
root's inedge DO apply transform
apply OPEN/CLOSE embedding transform

Step 3: Embed the interface statements into the host program

We will illustrate the concepts of embedding with the program of section 6.3.2.1. The program is initially represented as in figure 6.18. DML find statements have been replaced by the name of their associated access map-

pings. For the predecessor segment's query, transformation 2 is applied to $TITLE^{-1}$ o $ASSIGN^{-1}$, and OPEN and CLOSE are embedded (see figure 6.19). For the successor segment's query, no transformation need be applied. OPEN and CLOSE statements are embedded (see figure 6.20). The embedded program becomes:

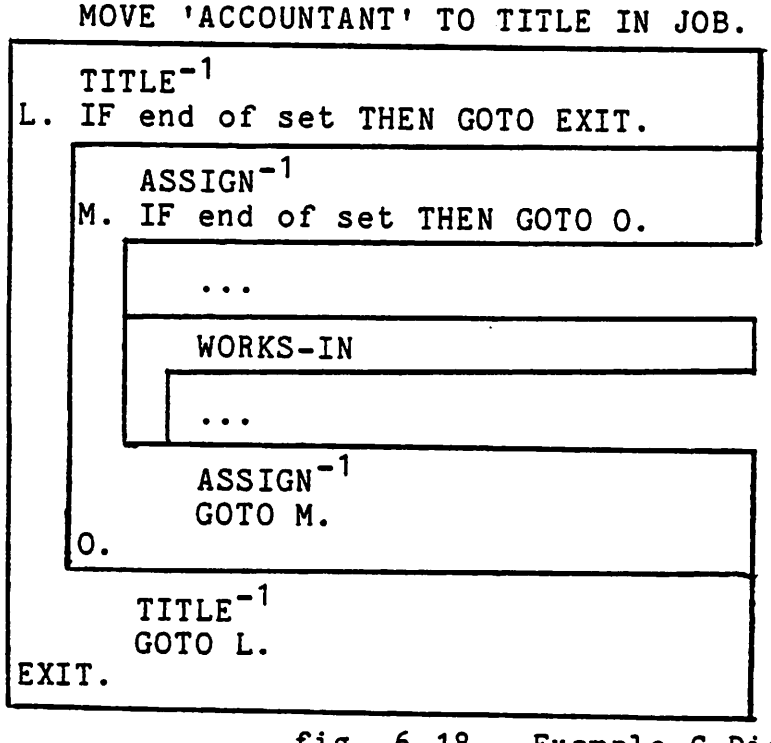


fig. 6.18 - Example C-Diagrams

```
   MOVE 'ACCOUNTANT' TO TITLE IN JOB.
   OPEN
   TITLE^-1 o ASSIGN^-1
L. IF end of set THEN GOTO EXIT.
      ...
      WORKS-IN
         ...
      TITLE^-1 o ASSIGN^-1
      GOTO L.
EXIT. CLOSE
```

fig. 6.19 - Compose Predecessor Query

```
   OPEN
   TITLE^-1 o ASSIGN^-1
L. IF end of set THEN GOTO EXIT.
      ...
      OPEN
      WORKS-IN
         ...
      CLOSE
      TITLE^-1 o ASSIGN^-1
      GOTO L.
EXIT. CLOSE
```

fig. 6.20 - Final Flow Chart

```
LET C1 BE <predecessor query>
LET C2 BE <successor query>
```

```
   MOVE 'ACCOUNTANT' TO TITLE IN JOB.

L.
   OPEN C1.
   SELECT C1.
M. IF end of set  GO TO EXIT.
      FETCH C1.
      IF EMP.BIRTHYR <= 1950  GO TO N.
      ... other code ...
         OPEN C2.
         SELECT C2.
         FETCH C2.
         ...
         CLOSE C2.
N.    SELECT C1.
      GO TO L.

EXIT. CLOSE C1.
```

6.3.3. The Class of Decompilable Programs

Not every CODASYL DML program can be decomposed into one with a relational access specification. Intuitively, if the data accessed by the program can be generated by a collection of parameterized non-procedural relational queries, then the program is decompilable.

Several properties of a program are necessary for its decompilablity. First, orderings among objects must be logical rather than physical. For example, it is possible to insert a new member record into a set instance after all previously inserted members. This is tantamount to sorting the members by time of insertion. Decompilation will successfully preserve the semantics of the program only if such orderings are supported logically, e.g., by specifying a set order of ascending insertion date, with

date included as a data item in each record. Physical orderings are not guaranteed to be maintained should the underlying database be converted to a different data model. In practise, this causes difficulties because existing programs often take advantage of physical orderings. In these cases, human intervention may be required to direct the translation.

Second, the decompilation algorithm is based on the assumption that data access is requested via atomic access units consisting of standard code sequences. These have been selected because of their semantic relationship to access path operations. Other combinations are possible, but they cannot be decompiled. For example, consider the pair

(1) FIND EMP WITHIN CURRENT WORKS-IN USING BIRTHYR.

(2) FIND DUPLICATE WITHIN WORKS-IN USING SALARY.

The above pair requests all employees in the current department who make the same salary as the first employee born in a specified year. The object set associated with (1) consists of all employees within the current department born in the specified year, while associated with (2) is an object set that consists of all employees within the current department who make the same salary as the previously accessed employee. Because the object sets are not the same, the statements cannot be combined.

Lastly, a canonical form exists for a decompilable program. The program can be represented as a sequence of composite C-diagrams. Each composite C-diagram consists of nested C-diagrams of adjacent access units. Each composite diagram can be mapped into a single relational query. If the program cannot be placed into this form, either because of misused statement pairs or poorly structured access, then it is not decompilable.

A program can be written, or transformed, so as to minimize the number of decompiled relational queries. First, all initializations of variables in the UWA should be performed as early as possible, i.e., before the sequence of FIND statements of the query are executed. Second, GET statements should be deferred as long as possible to insure they will not cause a premature procedural break. Applications code should be moved to the innermost loop feasible. Determining when such transformations do not alter the semantics of the program is beyond the scope of this thesis.

Decompilation is most successful for queries which involve long paths, i.e., few or no intersections, and for which the facilities of the DML are sufficient for expressing the query. Any necessity to perform computation outside of the DML, e.g., data item tests which are not based on equality or intersections of paths, will result

in a break and thus a less complete decompilation. Note that the DML does not support the intersection of a functional access mapping with property inverse mappings, even when the latter involve equality. This will always lead to a break.

We have characterized the class of programs that can be decompiled by the algorithm of this chapter. Not every "decompilable" DML program is successfully handled. Ideally, the goal of decompilation is to replace as much of a procedural program by non-procedural operations as possible. This can be achieved if the relational interface supports some, if not most, of the enumeration capabilities of the DML. The formulation of the appropriate interface is still an open question. A possible candidate is the link and selector language of [TSIC76b]. It combines aspects of DML enumeration with a non-procedural query specification.

Even for portions of a program that cannot be represented non-procedurally, it may still be possible to use non-procedural operations to severely limit the set of records over which the record-at-a-time operations are applied. In short, the goal is to do as much non-procedurally as possible. This objective is not completely obtained with our algorithm.

Our algorithm succeeds when the currency flow graph can be reduced, because the conditions for applying the graph transformations are met. A failure is caused by a malformed atomic access unit, as indicated by nodes of the flow graph that should be combined but cannot. A malformed access unit can often be supported procedurally as outlined above. However, we have not investigated methods for finding the covering non-procedural operations. Note that procedural enumeration operations will cause breaks analogous to those encountered during embedding. Thus a query which contains several procedural access units may be broken into several subqueries.

6.4. Compilation

The compilation process consists of two stages. The first stage takes a relational calculus query, written for a particular relational schema, and maps it into an internal form called the iterative query language (IQL). The cost of the query, in terms of number of pages accessed, is sensitive to the order in which relations are iterated in the IQL. Information in the associated access path schema is used to find an "optimal" iteration order. The second stage maps the IQL query into a CODASYL DML routine to access data from a semantically equivalent CODASYL schema.

We assume that the relational interface described in the previous section is being used here. A compiled relational program will have a CODASYL DML coroutine associated with each query. These coroutines communicate with the user program through a program UWA, and with the database system through their own UWA. An OPEN statement has the effect of copying the values from the program's UWA into the coroutines' UWA. SELECT passes control from the user program to the DML coroutine to actually access the data. A FETCH will cause data in the UWA to be copied into the program UWA.

6.4.1. Iterative Query Language

In this section, we define an intermediate form for compilation. The IQL is a semi-procedural relational query language. It is procedural in that an order of iteration is imposed on the enumeration of tuples to answer a query. It is non-procedural in that access paths are represented by boolean connectives rather than access mappings. The access paths used to access an object are not explicitly specified.

An IQL query consists of a nested sequence of FOR EACH statements. The format of the i^{th} statement in the nesting is:

FOR EACH $v_i \in R_i$ ST $B(v_1,\ldots,v_i)$
  DO <statement block>

R_i is a relation. V_i is a range variable, which is used to represent the current tuple in the enumeration of R_i. B is a boolean condition which is applied to v_i and its predecessor range variables in outer nested FOR EACH statements. The statement block is only executed if B is true for the current tuples represented by v_1, ... , v_i. The semantics of the FOR EACH can be described as:

```
    v_i <- first tuple of R_i
λ1: IF v_i = ø THEN GO TO λ2
        IF B(v_1,...,v_i) THEN <statement block>
        v_i <- next tuple of R_i
        go to λ1
λ2:
```

As an example, consider the query that requests the location of John Smith's department. Assume we have the schema of the previous section. The query might be expressed as:

```
FOR EACH E ∈ EMP ST E.ENAME = "John Smith" DO
    FOR EACH D ∈ DEPT ST D.DNO = E.WORKS-IN DO
        PRINT D.LOCATION
```

The program scans the employee relation until it finds a tuple with the employee name of John Smith. E.WORKS-IN is the department number of his associated department. The department relation is scanned to find a tuple with the matching department number. This access can be efficiently implemented if the WORKS-IN access path is supported in the access path schema. Once found, the department's location is printed.

Once an order has been determined, it is straightforward to map a relational calculus query into an IQL query. In order to make maximum use of logical access paths, some clauses are expanded by introducing transitivities. For example, if Q ranges over QUAL and A over ALLOC, then "Q.JID = A.JID" does not correspond to a logical access path. However if the clause is replaced by "Q.JID = J.JID AND J.JID = A.JID," then each clause represents a logical access path. This transformation can be applied to equality clauses between matching identifiers of relationship relations. Next, we associate a FOR EACH statement with each range variable of the relational calculus query. One-variable clauses or clauses which involve the range variable and previously iterated variables are associated with the FOR EACH. The above query would have been derived from the relational calculus query:

```
RANGE OF E IS EMP
RANGE OF D IS DEPT
RETRIEVE (D.LOCATION) WHERE E.ENAME = 'John Smith'
                      AND   E.WORKS-IN = D.DNO
```

6.4.2. Cost Model for IQL Queries

The objective of optimization is to determine an iteration order which minimizes the number of pages accessed to solve a query. Suppose that a query involves the relations R_1, ... , R_n. Let an ordering $\underline{i} = (i_1$, ... ,

i_n) be a permutation of 1, ... , n. For a given $\underline{i}$, let

$q_j(\underline{i})$ = Average number of R_{i_j} tuples selected for each assignment of tuples to (v_{i_1}, ... , $v_{i_{j-1}}$)

$p_j(\underline{i})$ = Average number of R_{i_j} pages accessed to determine the qualified tuples

$N_n(\underline{i})$ = total number of pages of R_{i_1}, ... , R_{i_n} accessed

Then $N_n(\underline{i}) = \sum_{j=1}^{n} p_j(\underline{i}) \prod_{k=1}^{j-1} q_k(\underline{i})$.

The $q_j(\underline{i})$ and $p_j(\underline{i})$ can be determined from the access path schema associated with the relational schema. In section 6.3.2.2 an algorithm was described to map access path expression graphs into relational linkage terms. Here we must invert that process: we wish to recognize the underlying access paths represented by the clauses of the FOR EACH statement. The mapping between linkage terms and access paths is as follows:

(1) "range variable$_A$.P = value" can be mapped into the property P: A --> Value Set. For example, J.TITLE = 'ACCOUNTANT' is mapped into the property TITLE: JOB --> CHAR(20).

(2) "range variable$_A$.F = range variable$_B$.ID$_B$" is mapped into the association F: A --> B. For example, E.ASSIGN = J.JID is mapped into the association ASSIGN: EMP --> JOB.

(3) "range variable$_R$.ID$_A$ = range variable$_A$.ID$_A$" is mapped into the relationship association R-A: R --> A. For example, Q.ENO = E.ENO, with Q ranging over QUAL and E ranging over EMP, is mapped into the association QUAL-EMP: QUAL --> EMP.

$P_j(\underline{i})$ and $q_j(\underline{i})$ depend on information about the objects in the schema, and the functionality and label of the access path. For a schema object A, let $p_F(A)$ be the expected number of pages accessed to find the desired subset of A via path F. Let $\pi(A)$ be the number of pages on which elements of A can be found, and n(A) be the cardinality of A. Then $p_F(A)$ is defined as follows:

F^{-1}: B --> A
Label is "W" then $p_F(A) = 0$
"C" then $p_F(A) = \frac{\pi(A)}{n(B)}$
"I" then $p_F(A) = \frac{n(A)}{n(B)}$

F: A --> B
Label is "W" then $p_F(A) = 0$
else $p_F(A) = 1$

In the above, we have assumed that objects in the domain of a mapping are uniformly distributed among the objects in the range. This assumption is often not true in practise. In addition, it is difficult to maintain accurate counts of object cardinalities in real databases. Therefore the page costs should be considered as estimates at best. The $p_j(\underline{i})$ is determined by evaluating all

candidate access paths, and choosing F with the minimum value for $p_F(R_{i_j})$. The access path F is called the minimum access path (MAP).

Consider the WORKS-IN association of the employee database. Let n(EMP) = 100 and n(DEPT) = 10. The average number of employees per department is 10, or restated, $\frac{1}{10}$ of the employees are in a given department. The fraction $\frac{1}{10}$ is called the selectivity of WORKS-IN, denoted by $f_{WORKS-IN}$. The number of departments per employee is 1, which is 10 * $\frac{1}{10}$. Thus the expected fraction of objects accessed is given by $\frac{1}{n(Range_F)}$. This observation holds only for mappings derived from equality clauses. The value of f_F for mappings derived from inequality clauses is arbitrarily approximated by $\frac{1}{3}$, i.e., one third of the accessed objects are expected to meet the range qualification. The selectivity of intersected mappings, derived from conjunctive clauses, is the product of the selectivities of each mapping. Thus $q_j(\underline{i}) = n(R_{i_j}) * \prod_{K \in S} f_K$, where S is the set of paths derived from the clauses of the j^{th} FOR EACH statement.

6.4.3. Determining Iteration Order

The operations of CODASYL DML facilitate the expression of queries that trace a path through the schema. However, the intersection of paths is not adequately

supported, except for the trivial case of the intersection of an inverse mapping (owner to member access) with property inverses (USING clause). An intersection over two paths can be formed by enumerating all elements on one path for each element on the other. To determine if the same element appears on both paths, we must escape to the host language and procedurally compare their database keys. Therefore the concept of a path query appears basic in the CODASYL environment. In addition, in the previous section we saw that queries which are intersection-free, except for property mapping intersections, are the basic unit of decompilation. If we desire our compilations to be reversible, we must limit ourselves to queries which can be expressed without intersections. Path queries are a class for which all the intersection-free orderings can be easily found.

Let Q be a query expressed in the relational calculus with qualification q in conjunctive normal form. Define its associated query graph $G_Q(V_Q,E_Q)$ as:

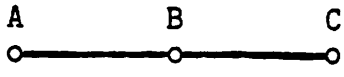
V_Q = set of relations mentioned in the range statements of Q.

E_Q = $\{(i,j) \mid i \neq j$ and some clause of q references both R_i and $R_j\}$.

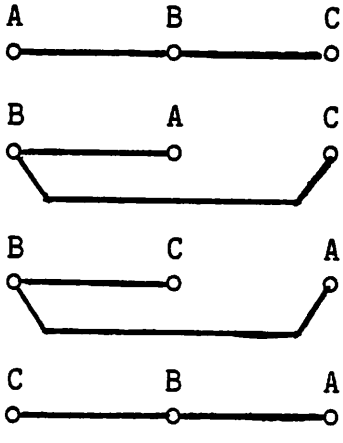
A path query is one in which (1) all clauses are at most two variable and based on equality comparison (equi-join)

and are supported by the underlying access path schema, and (2) its associated query graph is acyclic, and no node is connected to more than two other nodes.

An intersection-free order is one in which the nodes are arranged in a sequence so that no node has been preceeded in the sequence by more than one connected node. For example, the path:



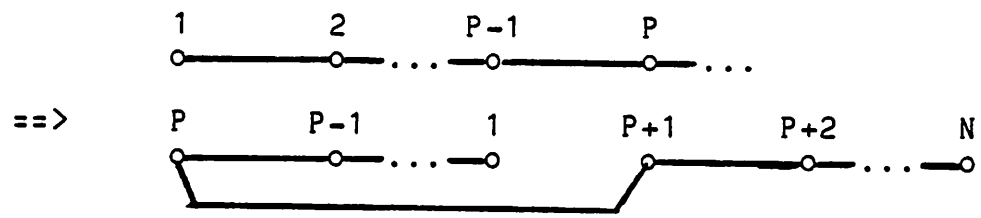
has four intersection-free orderings:



In general, there are N! ways to arrange N nodes. However, there are only 2^{N-1} intersection-free orderings of a path with N nodes. This represents a considerable reduction in the number of orders that must be evaluated:

N	2^{N-1}	N!
1	1	1
2	2	2
3	4	6
4	8	24
5	16	120
6	32	720
7	64	5040
8	128	40320

An algorithm to generate all intersection-free orders in figure 6.21. A node in the path is chosen as the first node in the iteration, partitioning the nodes to its left and right in the path. All nodes to the left must be visited in inverse order to avoid an intersection:



For example, if P-2 is visited before P-1, an intersection is caused among the edges from P and P-2. Each partition is partially ordered by this restriction. The problem then becomes one of shuffling two sequences together in all ways which preserve the order of each individual sequence. The algorithm does this by selecting the first element of one sequence as the node to visit next, and recursing on the remaining sequences. The process is repeated with the first node of the other sequence.

Another class of queries are those whose query graphs form an acyclic graph. A tree query is a query in which

```
GenerateOrders
/* algorithm chooses the i^th node for the pivot.
   Path order is represented by the array ORDER. The
   pivot partitions it into ORDER[1:i-1] and ORDER[i+1:n] */
FOR i <- 1 TO N DO
   RECURSE(1,i-1,i+1,n,{i})

/* parameters are index of start of left partition, end of
   left partition, start of right partition, end of right
   partition, and result, is a sequence of nodes (|| is
   sequence concentration operator) */
RECURSE(leftstart,leftend,rightstart,rightend,result)
   IF ORDER[leftstart:leftend] = ø THEN
      result <- result || ORDER[rightstart:rightend]
      PRINT result
   ELSE
     IF ORDER[rightstart:rightend] = ø THEN
        result <- result || ORDER[leftstart:leftend]
        PRINT result
   ELSE
     result' <- result || ORDER[rightstart]
     RECURSE(leftstart,leftend,rightstart+1,rightend,result')

     result' <- result || ORDER[leftstart]
     RECURSE(leftstart+1,leftend,rightstart,rightend,result')
```

fig. 6.21 - Intersection-free Order Generation

(1) all clauses are at most two variable and based on equality and are supported by the underlying access path schema, and (2) its associated query graph is acyclic and connected. Any preorder traversal of a tree query's graph will result in an intersection-free order. However, because a query graph for a tree query has a more complex structure, there does not appear to be a straight-forward algorithm to enumerate all intersection-free query orders.

6.4.4. Compilation into CODASYL DML

A query written in IQL imposes an iteration order for processing the query. Given this information, the query can be mapped into the accessing operations supported by CODASYL DML. The approach is to choose an access path to access the object, and to use the primitives of DML to take advantage of the selected access path.

Certain restrictions are imposed on the query to simplify the algorithm. The boolean clause associated with each FOR EACH statement is assumed to be in conjunctive normal form. Further, each conjunct involves at most two variables, and an OR clause can only be specified for alternative values of a value attribute (e.g. J.SALARY < 10K OR J.SALARY > 15K), not a key attribute.

6.4.4.1. Algorithm for Compilation

In this section, we present a recursive decent algorithm for generating a CODASYL DML program from an IQL query. The algorithm proceeds from the outermost loop towards the innermost loop. It is invoked by calling CODEGEN(1). The algorithm is given in figure 6.22.

For the i^{th} nested FOR EACH statement, CODEGEN determines whether any clause can be supported by a set access. If none can, it attempts to find a one-variable equality clause on which to base the access. The cases are: (1) the

---

Algorithm GenerateDML

```
procedure CODEGEN(i)
  IF i > number of FOR EACH statements THEN
    ⌈generate code to interface results with user program
    ⌊RETURN
  LET B(v1,...,vi) = C1(v1,...,vi) AND C2(v1,...,vi)
                     AND C3(vi) AND C4(vi)
    WHERE
      C1(v1,...,vi) = supported 2-var equality clauses
      C2(v1,...,vi) = 2-var clauses not in C1
      C3(vi) = 1-var equality clauses
      C4(vi) = 1-var clauses not in C3
    NOTE: clauses in C2 are treated as 1-var by substitut-
          ing values from the current r1,...,ri-1

  IF C1 = ø THEN /* access by equality clause */
    ⌈LET C3(vi) = C31(vi) AND C32(vi) AND C33(vi)
      WHERE
        C31(vi) = indexed key attribute clause
        C32(vi) = indexed non-key attribute clauses
        C33(vi) = non-indexed attribute clauses

    IF C31 ≠ ø THEN /* access by identifier */
      ⌈K <- key name of identifier data item
      generate
        "       MOVE value TO ri.identifier data item.
                FIND ANY ri USING K.
                IF NOT FOUND THEN GO TO λi.
                  GET ri.
                  IF NOT (C2 AND C32 AND C33 AND C4)
                    THEN GO TO λi."
      CODEGEN(i+1)
      generate
        "λi: "
      ⌊

    ELSE IF C32 ≠ ø THEN /* access by indexed property */
      ⌈MAP <- minimum cost property mapping
      K <-key name of MAP
      C32' <- C32 - MAP clause
      generate
        "       MOVE value TO ri.value data item.
                FIND ANY ri USING K.
         λi+2:  IF NOT FOUND THEN GO TO λi.
                  GET ri.
                  IF NOT (C2 AND C32' AND C33 AND C4)
                    THEN GO TO λi+1."
      CODEGEN(i+1)
```

```
    generate
      "λi+1:   FIND DUPLICATE r_i USING K.
               GO TO λi+2.
       λi: "

  ELSE /* complete record enumeration */
    data item names derived from C33 clauses
    FOR EACH clause j in C33 DO
      generate
        "     MOVE value TO r_i.data item_j."
    generate
      "       FIND FIRST r_i USING data item_1,...
       λi+2: IF NOT FOUND THEN GO TO λi.
               GET r_i.
               IF NOT (C2 AND C4) THEN GO TO λi+1."
    CODEGEN(i+1)
    generate
      "λi+1:   FIND NEXT r_i USING data item_1,...
               GO TO λi+2.
       λi: "

ELSE /* supported 2-var clauses */
  MAP <- association mapping
  S <- set name of MAP

  IF MAP is functional THEN
    generate
      "       FIND OWNER WITHIN S.
              IF NOT FOUND THEN GO TO λi.
                GET r_i.
                IF NOT (C2 AND C3 AND C4)
                  THEN GO TO λi."
    CODEGEN(i+1)
    generate
      "λi: "

  ELSE /* MAP is inverse */
    IF C3 ≠ ∅ THEN /* use USING clause FIND */
      data item names derived from C3 clauses
      FOR EACH clause j in C3 DO
        generate
          "  MOVE value TO r_i.data item_j."
      generate
      "       FIND r_i WITHIN S CURRENT
                USING data item_1,...
       λi+2: IF NOT FOUND THEN GO TO λi.
               GET r_i.
               IF NOT (C2 AND C4)
                 THEN GO TO λi+1."
      CODEGEN(i+1)
      generate
```

```
    "λi+1:   FIND DUPLICATE WITHIN S
               USING data item1,...
             GO TO λi+2.
     λi: "
ELSE /* USING clause does not apply */
    generate
    "       FIND FIRST ri WITHIN S.
     λi+2: IF NOT FOUND THEN GO TO λi.
             GET ri.
             IF NOT (C2 AND C4)
               THEN GO TO λi+1."
    CODEGEN(i+1)
    generate
    "λi+1:   FIND NEXT ri WITHIN S.
             GO TO λi+2.
     λi: "
```

fig. 6.22 - Compiling into DML

key attribute of the relation appears in a one-variable equality clause, (2) indexed attributes appear in one-variable equality clauses, or (3) value attributes appear in equality clauses. In case (1), code is generated to access the CODASYL record by its key data item. In (2), an indexed minimum cost access path is found and used to access the record. In (3), a FIND USING statement is employed to access the record, automatically implementing the equality tests for the value data items.

Otherwise, at most one clause is supported by set access because the ordering is intersection-free. The underlying access mapping is selected to access the records. If the access mapping is functional, then a FIND OWNER statement is used. If it is an inverse mapping, and

there are one-variable equality clauses in the condition, then the FIND CURRENT / FIND DUPLICATE statement pair is used. This is to take advantage of the USING clause to implement the value data item equality tests. Otherwise a FIND FIRST/NEXT WITHIN SET operation pair is generated. The complete details can be found in the algorithm.

6.4.4.2. Some Peep-hole Optimizations

In the compilation algorithm, not much consideration has been given to optimizing the program produced, outside of selecting the most appropriate DML primitives to implement a given access. Several improvements can be made:

(1) If the clauses involved in the predicate test are all null, then the IF statement can be removed.

(2) The entire record need not be transfered into the UWA. Only those data items which are subsequently used within the iteration need be accessed by the GET statement. If no data item values are subsequently used, then the GET statement can be deleted.

(3) If a functional MAP is known to be an association, then the test for NOT FOUND can be dropped, because the range must exist under the definition of an association.

(4) If a record type must be completely enumerated, and no equality clauses can be used to restrict the

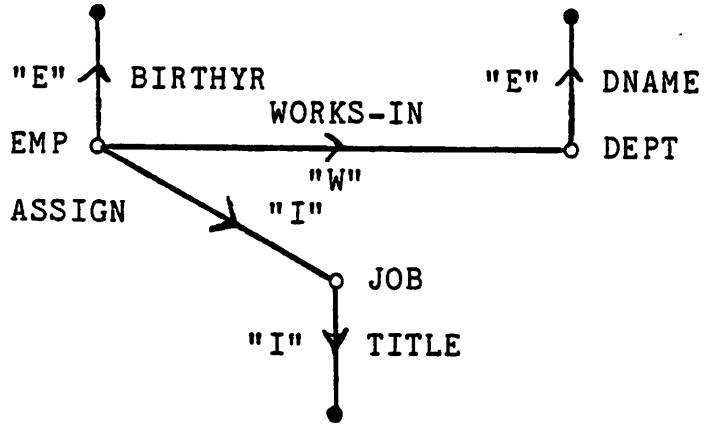
enumeration, then the USING clause in the FIND FIRST/NEXT record operation can be dropped.

6.4.4.3. Example Compilation

Consider the relational query to find the names of all departments which employ accountants born after 1950. The APML query might look like the following:

```
FOR EACH J ∈ JOB ST J.TITLE = 'Accountant' DO
  FOR EACH E ∈ EMP ST E.ASSIGN = J.JID
                      AND E.BIRTHYR > 1950 DO
    FOR EACH D ∈ ST D.DNO = E.WORKS-IN DO
      ...
```

The relevant portion of the employee access path schema is:



The first FOR EACH involves an indexed one-variable clause, i.e. J.TITLE = 'Accountant'. The code generated is:

```
     MOVE 'ACCOUNTANT' TO JOB.TITLE.
     FIND FIRST JOB USING TITLE.
λ2:  IF NOT FOUND THEN GO TO λ1.
       ---
       FIND NEXT JOB USING TITLE.
       GO TO λ2.
```

Note that the data items of JOB are not needed by the rest of the query, so there is no need to actually move the JOB records into the UWA.

The second FOR EACH involves a two-variable clause which is implemented by a set, i.e. E.ASSIGN = J.JID. The other clause, E.BIRTHYR > 1950, is one-variable but not equality, and thus cannot be used within the DML. The code generated is:

```
          FIND FIRST EMP WITHIN ASSIGN.
λ5:       IF NOT FOUND THEN GO TO λ3.
            GET EMP; BIRTHYR.
            IF NOT (EMP.BIRTHYR > 1950) THEN GO TO λ4.
            ---
λ4:         FIND NEXT EMP WITHIN ASSIGN.
            GO TO λ5.
```

This section of code is nested within the previous one.

The last FOR EACH involves a single clause supported by a set, i.e. D.DNO = E.WORKS-IN. Because the access is functional, the FIND OWNER operation is used. Because the mapping is an association, the test for NOT FOUND can be omitted:

```
FIND OWNER WITHIN WORKS-IN.
  GET DEPT; DNAME.
  ---
```

This is nested within the above. At the very innermost level of nesting, code is incorporated to return results to the user program upon request. The compiled program is the same as that of figure 6.10 in section 6.3.2.1.

6.5. Conclusions

In this chapter, we have presented detailed algorithms for translating programs whose associated databases have themselves been translated. The algorithms make extensive use of the information contained in the database's access path schema.

A decompilation algorithm to map CODASYL DML operations into a relational query was presented. The program is analyzed to determine which code sequences correspond to set-oriented mappings of the access path schema. The composed mappings can be translated into a relational query. The algorithm is most successful, i.e. combines together the longest sequences of DML operations, when the associated iteration order is intersection-free.

An algorithm to compile relational queries into DML was also presented. Again, the algorithm is limited to those iteration orders which are intersection-free. Information in the access path schema is used to help estimate the costs of access and to take advantage of supported access paths.

CHAPTER 7

CONCLUSIONS

7. Conclusions

In this final chapter we briefly review the contributions of this dissertation and point to areas for future work.

7.1. Contributions

The major thrust of this thesis has been to apply a semantic database specification to several related problems in database design and translation. Its contributions can be classified into the four categories of: (1) logical design methodology, (2) physical design methodology, (3) schema conversion, and (4) program conversion. Each will be treated in turn.

A logical design methodology was developed which maps a semantic database specification of what is to be modelled into a logical schema for a particular data model. Design goals were formulated to aid in the derivation of the mapping rules. These represent requirements on how the logical schema is to behave under the update operations of the model. The methodology was shown to yield schemas which are well-behaved under update opera-

tions.

A physical design is integrated with the logical design by choosing implementation structures which support the logical access paths implied by the semantic specification. The storage structures and access methods of database systems were characterized by four implementation independent properties of a physical access path: "Evaluated", "Indexed", "Clustered", and "Well Placed." Methods of assigning these properties in a conflict-free manner to the access paths of a schema were presented. The schema, with assigned properties, is then used as a requirements specification for choosing actual implementation structures from among those supported by a specific system. The methodology partitions the physical design process into access path specification and implementation choice.

Methods were presented to augment target schemas with semantic information to facilitate the automatic translation of logical schemas between different data models. Translation algorithms for the CODASYL and Relational data models were derived. Redesign operations and evolution rules were defined for modifying the semantic content of a database schema.

An algorithmic approach to decompiling programs written with low level procedural manipulation operations into

a high level non-procedural query specification was presented. The class of "decompilable" programs was characterized. Methods for compiling non-procedural queries into low level operations were also explored.

7.2. Future Work

The original motivation for this work was to explore techniques for constructing a heterogeneous database management system on top of existing database managers. The methodologies for logical and physical database design and the techniques for schema and program conversion are a step toward achieving this goal. More work is needed to actually achieve it.

In the realm of physical database design, several problems are still unsolved. The integer linear programming formulation for assigning properties to access paths is computationally expensive to use. Does a special purpose branch and bound or dynamic programming algorithm exist for its efficient solution? Further, the algorithms for implementing a schema given the requirements specification require a minimum amount of information from the user. Is it possible to propose a family of algorithms, that when given more detailed usage information, will generate better physical designs? More work is needed on choosing optimal structures to support access paths.

For schema conversion, methods are needed for constructing programs to actually perform the translation between different data models. These methods should be implemented in order to answer questions of efficiency. Can the translation be done in real time? Then it might be possible to support multiple manipulation languages by translating the data into the data model upon which the query language is defined. How can the access path specification be used to aid in the translation of a schema at the physical level? The specification could be used to choose new storage structures for the translated data within the target database system.

The techniques developed here for program conversion should be extended to a larger class of programs. A more extensive analysis of the program, involving special cases and general program transformations, is a fruitful, albeit difficult, area for future research.

Much works remains in formulating efficient query processing techniques which compile non-procedural queries into procedural access path operations. Heuristics should be developed for selecting a "good" order in which to visit the objects mentioned in a query.

Finally, an attempt should be made to actually implement a heterogeneous database management system on top of existing relational and CODASYL systems. Such a system

should also be extended to a distributed environment. Several research projects are already pursuing this direction [CCA 79, KIMB79]. Perhaps a distributed design model could be formulated to aid in designing databases in this environment. Of particular interest is the use of the design model to help integrate existing databases. Work in this direction is reported in [ELMA79].

REFERENCES

[ADIB76] Adiba, M., et. al., "A Unified Approach for Modelling Data in Logical Data Base Design," in [NIJS76].

[AHO 77] Aho, A. V., Ullman, J. D., Principles of Compiler Design, Addison Wesley, Reading, Mass., 1977.

[BENC76] Benci, E., et. al., "Concepts for the Design of a Conceptual Schema," in [NIJS76].

[BERE77] Berelian, A., Irani, K., "Evaluation and Optimization," Proc. Intl. Conf. on Very Large Data Bases, 1977.

[BERN76] Bernstein, P. A., "Synthesizing Third Normal Form Relations from Functional Dependencies," A.C.M. Trans. on Database Sys., V 1, N 4, (Dec 76).

[BLAS79] Blasgen, M. W., et. al., "System R: An Architectural Update," IBM Research Report RJ2581(33481), (Jul 79).

[BRAC76] Bracchi, G., et. al., "Binary Logical Associations in Data Modelling," in [NIJS76].

[BUBE76] Bubenko, J. A., et. al., "From Information

Requirements to DBTG Data Structures," A.C.M. SIGPLAN-SIGMOD Conf. on Data, (Mar 76).

[BUNE79] Buneman, P., Frankel, R. E., "FQL -- A Functional Query Language," Proc. A.C.M. SIGMOD Conf., (May 79).

[CARD73] Cardenas, A. F., "Evaluation and Selection of File Organization - A Model and System," Comm. A.C.M., V 16, N 9, (Sep 73).

[CARD75] Cardenas, A. F., "Analysis and Performance of Inverted Data Base Structures," Comm. A.C.M., V 18, N 5, (May 75).

[CCA 79] Computer Corporation of America, "MULTIBASE - A Research Program in Heterogeneous Distributed DBMS Technology," (Sep 79).

[CHAM76] Chamberlain, D. D., "Relational Data-Base Management Systems," A.C.M. Computing Surveys, V 8, N 1, (Mar 76).

[CHEN76] Chen, P. P., "The Entity-Relationship Model - Toward a Unified View of Data," A.C.M. Trans. on Data Base Sys., V 1, N 1, (Mar 76).

[CHEN77] Chen, P. P., Yao, S. B., "Design and Performance Tools for Data Base Systems," Proc. Intl. Conf. on VLDB, 1977.

[COBO78] CODASYL COBOL Committee Journal of Development, 1978.

[CODA71] CODASYL Data Base Task Group, April 1971 Report, A.C.M. (Apr 71).

[CODA73] CODASYL Data Description Language, Journal of Development, NBS Handbook 113, (Jun 73).

[CODA78] CODASYL Data Description Language, Journal of Development, (Jan 78).

[CODD70] Codd, E. F., "A Relational Model of Data for Large Shared Data Banks," Comm. A.C.M., V 13, N 6, (Jun 70).

[CODD71] Codd, E. F., "Further Normalization of the Data Base Relational Model," in Courant Computer Science Symposia 6, Data Base Systems, Prentice-Hall, New York, (May 71).

[CODD79] Codd, E. F., "Extending the Data Base Relational Model to Capture More Meaning," IBM Research Rep RJ2472, (Jan 79).

[DATE76] Date, C. J., "An Architecture for High-Level Database Extensions," A.C.M. SIGMOD Conf., (June 76).

[DATE77] Date, C. J., An Introduction to Database Systems, 2 ed., Addison Wesley, Reading, Mass., 1977.

[DEHE74] Deheneffe, C., et. al., "Relational Model for a Data Base," Proc. IFIP 1974 Conf., 1974.

[DUHN78] Duhne, R. A., Severence, D. G., "Selection of an Efficient Combination of Data Files for a Multiuser Database," Proc. AFIPS Natl. Comp. Conf., 1978.

[ELMA79] El-Masri, R., Wiederhold, G., "Data Model Integration Using the Structural Data Model," Proc. A.C.M. SIGMOD Conf, (May 79).

[EPST79] Epstein, R., et. al., "INGRES Version 6.2 Reference Manual," Memo No. UCB/ERL M79/43, (May 79).

[FAGI77a] Fagin, R., "Multivalued Dependencies and a new Normal Form for Relational Databases," A.C.M. Trans. on Database Sys., V 2, N 3, (Sep 77).

[FAGI77b] Fagin, R., "The Decomposition Versus the Synthetic Approach to Relational Database Design," Proc. Intl. Conf. on VLDB, 1977.

[GAMB77] Gambino, T. J., Gerritsen, R., "A Database Design Decision Support System," Proc. Intl. Conf. on VLDB, (Oct 77).

[GERR75] Gerritsen, R., "A Preliminary System for the Design of DBTG Data Structures," Comm. A.C.M., (Oct 75).

[GOTL74] Gotlieb, C. C., Tompa, F. W., "Choosing a Storage Schema," Acta Informatica, V 3, pp. 297 - 319, 1974.

[HALL76] Hall, P., et. al., "Relations and Entities," in [NIJS76].

[HAMM78] Hammer, M., McLeod, D., "The Semantic Data Model: A Modelling Mechanism for Data Base Applications," Proc. A.C.M SIGMOD Conf., (May 78).

[HECH77] Hecht, M. S., Flow Analysis of Computer Programs, North Holland, New York, 1977.

[HOUS77] Housel, B. C., "A Unified Approach to Program and Data Conversion," Proc. Intl. Conf. on VLDB, 1977.

[HSIA70] Hsiao, D., Harary, F., "A Formal System for Information Retrieval from Files," Comm. A.C.M., V 13, N 2, (Feb 70).

[KERS76] Kerschberg, L., et. al., "A Taxonomy of Data Models," in Systems for Large Data Bases, P. C. Lockermann, E. J. Neuhold, eds., North-Holland Pub. Co., Amsterdam, 1976.

[KIMB79] Kimbelton, S. R., et. al., "XNDM: An Experimental Network Data Manager," Proc. Fourth Berkeley Conference on Distributed Data Management and Computer Networks, (Aug 1979).

[KLUG79] Klug, A., "Entity-Relationship View Over Uninterpreted Enterprise Schemas," Proc. Intl. Conf. on Entity-Relationships, U.C.L.A., (Dec 79).

[LEFK69] Lefkovitz, D., File Structures for On-Line Systems, Spartan Books, N. Y., 1969.

[MCLE76] McLeod, D. J., "High Level Domain Definition in a Relational Data Base System," 1976 A.C.M. SIGMOD - SIGPLAN Conf. on Data, (Mar 76).

[MEAL67] Mealy, G. H., "Another Look at Data," Proc. AFIPS 1967 FJCC, V 31, AFIPS Press.

[MIJA76] Mijares, I., Peebles, R., "A Methodology for the Design of Logical Database Structures," in [NIJS76].

[MITO75] Mitoma, M. F., Irani, K., "Automatic Data Base Schema Design and Optimization," Proc. Intl. Conf. on Very Large Data Bases, 1975.

[MOUL76] Moulin, P., et. al., "Conceptual Models as a Data Base Design Tool," in [NIJS76].

[NATI78] Nations, J., Su, S. Y. W., "Some DML Instruction Sequences for Application Program Analysis and Conversion," Proc. A.C.M. SIGMOD Conf., 1978.

[NIJS75] Nijssen, G. M., "Set and CODASYL Set or Coset," in Data Base Description, B. C. M. Douque, G. M.

Nijssen, eds., North Holland Pub. Co., Amsterdam, 1975.

[NIJS76] Nijssen, G. M., ed., Modelling in Database Management Systems, North-Holland Pub. Co., Amsterdam, 1976.

[ROUS75] Roussopoulos, N., Mylopoulos, J., "Using Semantic Networks for Data Base Management," Proc. VLDB Conf., (Sep 75).

[SCHI77] Schindler, S. J., "Templates for Structured DML Programs," Working Paper ST 2.2, Data Translation Project, Graduate School of Business Administration, Univ. of Michigan, (Feb 77).

[SCHM75] Schmid, H. A., Swenson, J. R., "On the Semantics of the Relational Data Model," Proc. ACM SIGMOD Conf., (May 75).

[SCHK78] Schkolnick, M., "A Survey of Physical Database Design Methodology and Techniques," Proc. of Conf. on Very Large Data Bases, 1978.

[SELI79] Selinger, P., et. al., "Access Path Selection in a Relational Database Management System," Proc. of A.C.M. SIGMOD Conf., 1979.

[SHIP80] Shipman, D., "The Functional Data Model and the

Data Language Daplex," to appear in A.C.M. Trans. on Database Sys., 1980.

[SHOS75] Shoshani, A., "A Logical Level Approach to Database Conversion," Proc. A.C.M. SIGMOD Conf., 1975.

[SHU 77] Shu, N. C., et al, "EXPRESS: A Data EXtraction, PRocessing, and REStructuring System," A.C.M. Trans. on Database Systems, V 2, N 2, (Jun 77).

[SILE76] Siler, K. F., "A Stochastic Model for Database Organizations in Data Retrieval Systems," Comm. A.C.M., V 19, N 2, (Feb 76).

[SIBL77] Sibley, E. H., Kershberg, L., "Data Architecture and Data Model Considerations," Proc. AFIPS National Computer Conf., 1977.

[SMIT77a] Smith, J. M., Smith D. C. P., "Database Abstractions: Aggregation," Comm. A.C.M., V 20, N 6, (Jun 77).

[SMIT77b] Smith, J. M., Smith D. C. P., "Database Abstractions: Aggregation and Generalization," A.C.M. Trans. on Database Systems, V 2, N 2, (Jun 77).

[STON76] Stonebraker, M. R., "The Design and Implementation of INGRES," A.C.M. Trans. on Database Systems, V 1, N 3, (Sep 76).

[SU 76] Su, S. Y. W., "Applications Program Conversion Due to Data Base Changes," Proc. Conf. on Very Large Data Bases, 1976.

[SU 77] Su, S. Y. W., Liu, B. J., "A Methodology of Applications Program Analysis and Conversion Based on Database Semantics," Proc. A.C.M. SIGMOD Conf., 1977.

[SU 78a] Su, S. Y. W., et. al., "Application Program Conversion Due to Semantic Changes," CIS Report 7879-2, Dept. of Computer and Information Sciences, (Mar 78).

[SU 78b] Su, S. Y. W., Reynolds, M. J., "Conversion of High-Level Sublanguage Queries to Account for Database Changes," Proc. AFIPS Natl. Comp. Conf., 1978.

[TAYL76] Taylor, R. W., Frank, R. L., "CODASYL Data-Base Management Systems," A.C.M. Computing Surveys, V 8, N 1, (Mar 76).

[TAHA75] Taha, H. A., Integer Programming - Theory, Applications, and Computations, Academic Press, New York, 1975.

[TEOR78] Teorey, T. J., Oberlander, L. B., "Network Database Evaluation Using Analytic Modeling," Proc. AFIPS Natl. Comp. Conf., 1978.

[TSIC75] Tsichritizis, D. C., "A Network Framework for Relational Implementation," in Data Base Description, B. C. M. Douque, G. M. Nijssen, eds., North-Holland Pub. Co., Amsterdam, 1975.

[TSIC76a] Tsichritizis, D. C., Lochovsky, F. H., "Hierarchical Data-Base Management," A.C.M. Computing Surveys, V 8, N 1, (Mar 76).

[TSIC76b] Tsichritizis, D. C., "LSL: A Link and Selector Language," Proc. A.C.M. SIGMOD Conf., June, 1976.

[TSIC77] Tsichritizis, D. C., Klug, A., eds., "The ANSI/X3/SPARC DBMS Framework - Report of the Study Group on Data Base Management Systems," 1977.

[WONG76] Wong, E., Youseffi, K., "Decomposition -- A Strategy for Query Processing," A.C.M. Trans. on Database Systems, V 1, N 3, (Sep 76).

[YAO 75] Yao, S. B., Merten, A., "Selection of File Organization Using an Analytic Model," Proc. Intl. Conf. on Very Large Data Bases, 1975.

[YAO 77] Yao, S. B., "An Attribute Based Model for Database Access Cost Analysis," A.C.M. Trans. on Database Systems, V 2, N 1, (Mar 77).

[YOUS79] Youseffi, K., Wong E., "Query Processing in a

Relational Database Management System," Proc. Intl. Conf. on Very Large Data Bases, 1979.

[ZANI76] Zaniolo, C., "Analysis and Design of Relational Schemata for Database Systems," Ph.D. thesis, U.C.L.A., (Jul 76).

[ZANI79a] Zaniolo, C., "Multimodel External Schemas for CODASYL Data Base Management Systems," Proc. IFIP TC-2 Working Conf. on Database Arch., Venice, 1979.

[ZANI79b] Zaniolo, C., "Design of Relational Views over Network Schemas," Proc. A.C.M. SIGMOD Conf., (Jun 79).