INFORMATION TO USERS

This reproduction was made from a copy of a document sent to us for microfilming. While the most advanced technology has been used to photograph and reproduce this document, the quality of the reproduction is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help clarify markings or notations which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure complete continuity.

2. When an image on the film is obliterated with a round black mark, it is an indication of either blurred copy because of movement during exposure, duplicate copy, or copyrighted materials that should not have been filmed. For blurred pages, a good image of the page can be found in the adjacent frame. If copyrighted materials were deleted, a target note will appear listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed, a definite method of "sectioning" the material has been followed. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For illustrations that cannot be satisfactorily reproduced by xerographic means, photographic prints can be purchased at additional cost and inserted into your xerographic copy. These prints are available upon request from the Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases the best available copy has been filmed.

University Microfilms International
300 N. Zeeb Road
Ann Arbor, MI 48106

Learmonth, Gerard Paul

ADAPTIVE DATA MANAGEMENT

The University of Michigan PH.D. 1983

ADAPTIVE DATA MANAGEMENT

by
Gerard Paul Learmonth

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Business Administration)
in The University of Michigan
1983

Doctoral Committee:

Professor Alan G. Merten, Co-chairman
Professor Thomas J. Schriber, Co-chairman
Assistant Professor Marilyn Mantei
Associate Professor F. Brian Talbot
Associate Professor Toby J. Teorey

Why take the style of those heroic times?
For nature brings not back the mastodon,
Nor we those times; and why should any man
Remodel models?

Alfred Lord Tennyson
Morte d'Arthur

For Maryellen, Rod, and Colin

ACKNOWLEDGMENTS

I would like to thank all of the members of my dissertation committee for their time and effort in helping me to bring this dissertation to completion. Special thanks are due to Alan Merten who patiently guided me as some rather vague ideas eventually came together to form the basis of this research.

Most importantly, I want to thank my family for their understanding, love, and support.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

CHAPTER 1

INTRODUCTION

1.1 Background

Mankind for thousands of years has been an intelligent observer of his environment. Through the millenia he has felt the need to encode and record his observations for his own sake and for communicating these observations to others. From the cave drawings of the past to the very large computer-based databases of the present, the recording of facts about objects and events in the environment has been a characteristic human endeavor.

Encoded and recorded facts are referred to collectively as data. Data, in and of themselves, have no particular meaning unless they are interpreted. The interpretation process is essentially one of transforming data into information. Information then constitutes an increment of knowledge about the environment which is inferred from data. An organized collection of information about a particular subject, in turn, represents knowledge about that subject. This chain from the simple recording of data to its interpretation as information and

eventually to the acquisition of knowledge is carried out in virtually every area of human interest.

The digital computer, as a data processor, has enabled the mechanization of the recording and organizing of data. Through the application programs which the computer processes, some of the transformation of data into useful information is similarly automated. The human, however, is still an integral part of the process: a human decides which data are to be recorded; how they are to be organized based on perceived needs for information; and designs the application programs which affect the transform of data into information.

The work which is to follow is concerned with one aspect of the progression from data to information, that is, the organization and management of computer-based data. The goals will be to provide additional mechanisms for incorporating more of the meaning and interpretation of the data into its structure and to permit the resulting structure to adapt to changing patterns of usage over time.

1.2 The Problems and the Approach to Their Solution

Concern for the effective and efficient management of an enterprise's data has increased dramatically since the introduction of digital computers as data processors. Initially, attention was focused on the development of a comprehensive portfolio of application programs to support

routine, day-to-day operation of the enterprise. The acquisition, organization, and storage of data in machine-readable form was of secondary importance.

With the explosive growth in demand for information system support, attention has now shifted to the management of data [GIBS74, NOLA79]. It is recognized that an effective information system depends in large measure on a cohesive and well-managed base of data. These data are a valued strategic resource to the enterprise.

Coincidentally, this shift toward a data orientation came at a time when generalized database management systems became widely available. Prior to their introduction, an enterprise's computer-based data resource was typically stored and managed as a collection of separate files. Each file would contain instances of data records of a single type. One or more of these files would serve each application which in turn supported the interpretation of the data solely through the application procedure. This state of affairs naturally lead to considerable data redundancy, lack of data accuracy, and worst of all, lack of control over the vital data resource.

A generalized database management system provides a framework and a software tool for integrating data records of many different types into one, logically homogeneous file. The logical (and physical) structure of these record types within a database does provide some of the

interpretation of the data. One or more stored databases can then be used to satisfy many of the information requirements of the enterprise. Among the advantages of the database approach to the management of data are the minimization of redundancy, the ability to enforce accuracy standards, and the centralization of control over data [MART75].

The problems to be examined in the chapters to follow involve two substantively different, though essentially interrelated, issues. The first issue concerns the need to incorporate more of the interpretation of data into the design and implementation of a database for an enterprise. Presently, the design of a database, whether at the abstract level or physical design level, is conceptualized in two dimensions. The objects of interest to the enterprise are envisioned as lying in a plane with edges connecting them in semantically meaningful ways. These two design dimensions are adequate for capturing and representing some of the interpretation of the enterprise's data but certainly not all.

The principal thesis of this work is that there exists another important dimension to consider in modeling an enterprise's information requirements - the occurrence dimension. In this third dimension, the notion of instances of data and the relationships among them may be conceptualized. In order to demonstrate the effectiveness

of extending the art of database design into this dimension, two new database abstractions are introduced.

The second issue concerns the need to develop a consistent database design methodology which proceeds from an abstract, relatively unconstrained modeling environment to the ultimate implementation of the model in some database management system. This problem has not been fully addressed in the literature. Rather, specific portions of the design problem have been extensively examined without an attempt to fully integrate the entire process (viz., [TSIC82] and [DATE83]).

The solution to be offered here involves the development of an integrated, three level database design methodology which proceeds from an abstract data model, to a particular generic data model (i.e., the relational data model), and lastly to a proposal for an implementation of that generic data model. Not only will this methodology provide a consistent database design environment, but will also incorporate the notion of the occurrence dimension throughout.

By recognizing the occurrence dimension of data modeling and integrating it in all of the phases of database design, it will be shown that:

1. considerably more of the meaning and interpretation of an enterprise's data can be explicitly represented in the design of a database;

2. additional semantic constraints on the integrity of a stored database can be enforced;

3. the stored database can be manipulated more efficiently and the operational performance of the database can be improved; and

4. the operational life of the stored database can be extended because the database will be capable of adapting to changing requirements.

The next chapter will introduce and motivate the concept of the occurrence dimension in data modeling and data management and will outline the three levels of the proposed design methodology. Additionally, a case study in database design and implementation will be described. This case study will be utilized throughout the remainder of the work to demonstrate how the occurrence dimension can be effectively represented in each of the levels of database design and its eventual implementation and use. The succeeding three chapters will then focus on each design level in depth.

CHAPTER 2

ENTERPRISE INFORMATION MODELING

2.1 Introduction

Enterprise information modeling refers broadly to the art and practice of describing an enterprise in terms of its data and its information requirements. In general, the types of data collected and maintained by an enterprise are easily determined and will be relatively constant over time. The simple structuring of data types, however, does not necessarily represent all of an enterprise's information structure. The data need to be interpreted to convey information.

Traditionally, all of the interpretation of computer-based data has been embodied in the application programs which were designed to process the data. The organization and structure of the data were predicated on the efficiency of its storage and retrieval. With a generalized database management system, it is possible to organize an enterprise's data types so that their structure does convey some of the necessary interpretation. Consequently, the art of enterprise information modeling involves the consideration of the

semantics of data as well as the syntax of its representation. The goal of enterprise information modeling is the effective design of a database which will serve as an operational model, not just of an enterprise's data, but of its information requirements as well.

The practice of enterprise information modeling typically encompasses a number of discrete levels [TSIC82]. Each level involves a certain degree of abstraction. At one extreme - the level of abstract data modeling - a maximum degree of flexibility is obtained by suppressing the details (limitations) of the target generic data model. Generic data models, such as the hierarchical, network, and relational, impose restrictions on design alternatives due to the limited data structuring mechanisms which they support. In abstract data modeling, these data structuring limitations are temporarily ignored.

Similarly, at the level of generic data modeling, while the data structuring limitations are in effect, the low level, physical details of the internal data model are suppressed. Issues of data record volume, placement, and retrieval patterns are ignored. It is only at the last design level - the internal data model - that concern for the occurrences of the various data records becomes a design issue.

Abstraction is an important aspect of enterprise information modeling. Without the ability to suppress

detail, the task of designing a database for an enterprise would be prohibitively complex. However, there is the danger of oversimplifying and thus, missing important aspects of the design. Because one of the goals of designing a database for an enterprise is the representation of the meaning of data, suppressing the notion of occurrences of data objects until the last level of the design can have serious consequences.

This chapter will be concerned with motivating the concept of the occurrence dimension throughout the art and practice of enterprise information modeling. The next section will:

1. delineate the three discrete levels of the design process to be considered;

2. describe how the occurrence dimension fits within this framework; and

3. introduce a case study in database design which will serve as the vehicle for demonstrating the effectiveness of this dimension.

2.2 A Three Level Database Design Methodology

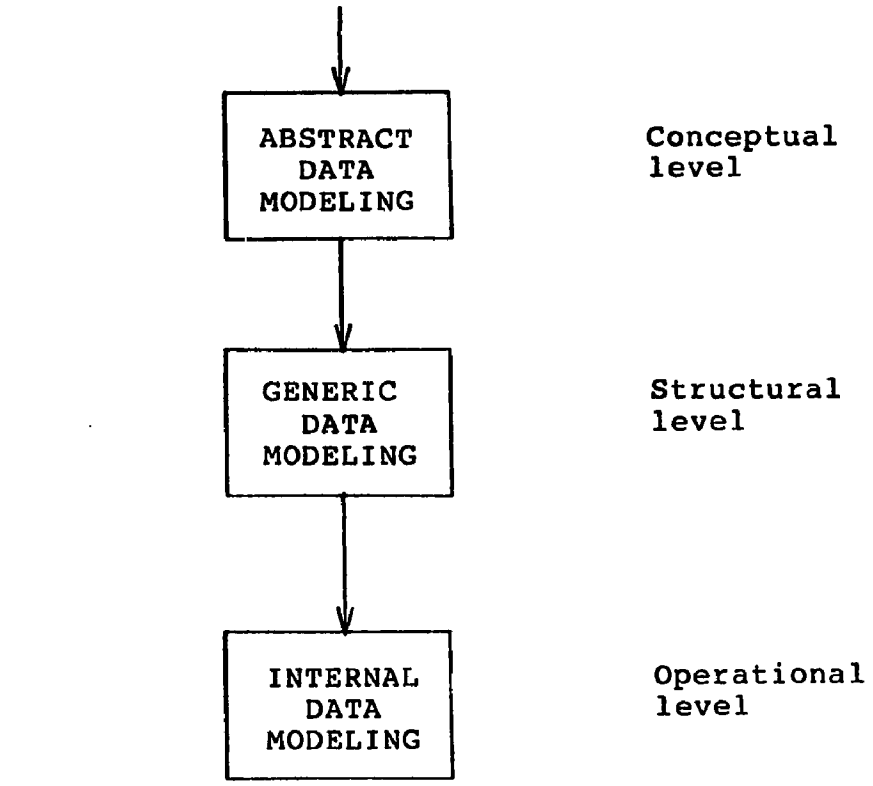
The process of designing a database for an enterprise consists of a sequence of activities leading from the perception of need to the eventual implementation and operational use of the database. However, several discrete points along this continuum have been the focus of attempts at developing rigorous modeling methodologies. The nature of the modeling activity at any point can be characterized by the structures, operators, and constraints available to the database designer [TSIC82].

Figure 2.1 shows the sequence of three discrete levels in the database design process which will be examined in detail.

The highest level is concerned with the development of an abstract data model of the enterprise. At this level, a database designer is free to utilize virtually any representation that is suitable for capturing both the syntax and semantics of the enterprise's information structure. The principal objective of abstract data modeling is to develop a complete and logically consistent model of the enterprise's information structure in terms of the objects of interest to it and the meaningful associations which exist among them.

The objects of interest to an enterprise are commonly called entities. An entity represents any object, real or abstract, about which the enterprise collects and maintains data. Entities may, in turn, be defined as a collection of atomic attributes which correspond to specific data types used to characterize the entity as a class. The concept of an entity is an abstraction because the designer is not concerned about individual occurrences of entities but merely their existence in general and their characterization in terms of data types. Therefore, the level of abstract data modeling has been labelled as the conceptual level in the figure.

ENTERPRISE INFORMATION MODELING



A three level design methodology

Figure 2.1

Each entity in an abstract data model serves only to indicate its existence as a conceptually meaningful object to the enterprise. Semantically, entities of different types, when associated with one another, capture added meaning over and above their individual representation in the abstract model. The abstract modeling mechanisms used to characterize the various relationships arising in an abstract data model are called database abstractions [SMIT77a].

Because an abstract data model is intended to be descriptive in nature, there is no particular need to provide operators to manipulate the model. Also, abstract data modeling is virtually devoid of constraints. The designer has considerable flexibility in choosing the design and representation most suitable for modeling an enterprise's information structure. Part of Chapter 3 will include an extensive survey of the state-of-the-art in abstract data modeling.

The next discrete level in the design process, shown in Figure 2.1, concerns generic data modeling. This particular level corresponds to mapping an abstract data model onto the structures allowed by a particular generic data model.

The notion of a generic data model is a generalization, or idealization, of database management systems which share to some degree a common set of structures, operators, and constraints. The first such

data model to be described was the relational model of data [CODD70]. This generic model consists of a simple structure, the flat tabular structure; a set of operations on these structures - the relational algebra; and certain constraints, for example, the Referential Integrity Constraint [DATE81]. Numerous implementations of this data model now exist. Ironically, the two other major generic models of data, the hierarchical and the network data models, had implementations in existence prior to the formulation of their conceptual bases [DATE83].

Although the generic data model does limit the database design alternatives in terms of structures, operators, and constraints, it is the structures available to the designer that are the most significant consideration at this level. The entities represented in the abstract data model map rather directly, though not necessarily on a one-to-one basis, onto record types and relation schemes of a generic data model.

The relationships portrayed among the entities must be representable within the more limited structuring discipline prescribed by the generic data model. The database designer must be able to make the necessary transformations from the relatively unconstrained abstract modeling environment to a structural representation supported by the generic data model. In doing so, the resulting generic data model structure must still preserve the semantic intent of the abstract design. To emphasize

this concern, the generic data model level has been alternatively labelled the structural level in the figure.

The structural representation of a database design at the generic data model level is again portrayed as a two-dimensional arrangement of the required record types (hierarchical or network) or relation schemes. In the hierarchical and network models, it is implicitly assumed that the necessary operators are available to navigate through the structural representation of relationships to satisfy the enterprise's information requirements. In the relational data model, the relationships indicated in the abstract model have no corresponding representation in terms of structure. Rather, certain key attributes must be redundantly represented in relation schemes to enable the operators of the relational algebra to effectively materialize these relationships when needed.

Once the transformation of the abstract data model to the generic data model has been accomplished, the generic model is then formally defined in the data definition language facility of some database management system which effectively implements that data model. Chapter 4 will be concerned with examining this process of transforming an abstract data model into a generic data model.

The last level of the database design process shown in Figure 2.1 deals with the internal data model. Having brought the design of a database from the abstract to the generic level, the last step in the design process is to

make appropriate decisions regarding the physical implementation of the database. It is at this level that consideration of the actual occurrences of data is traditionally first given. Abstract entities and their representation as record types or relation schemes at the generic level, are purely descriptive devices. Each expresses the intent of how data occurrences are to be stored. Likewise, relationships in either the abstract data model or a generic data model, indicate potential connection paths among semantically related entities.

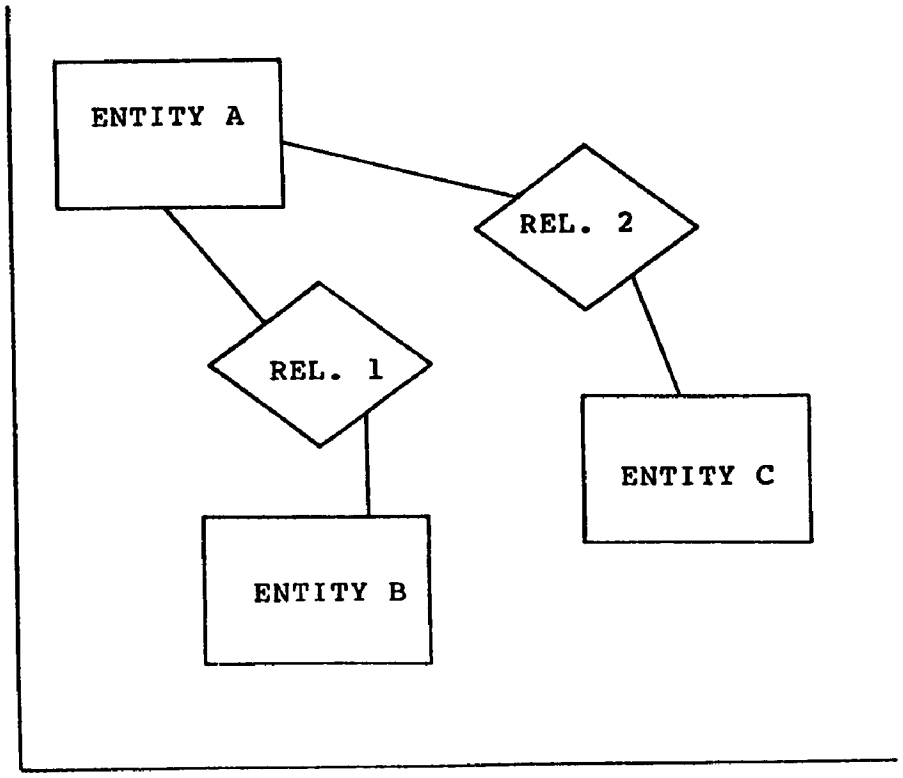
The internal data model corresponds to representing the database design utilizing the facilities of a particular database management system. This involves choosing strategies for the physical layout of record types and relation schemes; the allocation and organization of secondary storage; and the choice of physical storage structures to implement relationships. While the abstract data model and the generic data model are concerned primarily with the static, permanent representation of an enterprise's information structure, the internal data model must be concerned with the on-going, continuous use of the database.

If the operational life of a database is expected to be long, on the order of years, the assumptions made in its original design and the anticipated pattern of usage will be likely to change. The design decisions made in the internal data model representation of a database must at

least be cognizant of the dynamics of operational performance. At best, the internal data model should be equipped to monitor, analyze, and adapt as necessary to changing patterns of use. Chapter 5 will conclude the examination of the proposed three level database design methodology by looking at the issues relevant to designing an internal data model.

2.3 The Occurrence Dimension in Information Modeling

The representation of a database design, especially at the abstract and generic data model levels, typically takes the form of a two-dimensional diagram. Figure 2.2 shows a representative two-dimensional diagram of part of a hypothetical abstract model. Three entities are portrayed as rectangles with two relationships indicated by diamonds and edges. This diagraming convention was proposed by Chen [CHEN76] in the Entity Relationship Model.



An abstract data model diagram

Figure 2.2

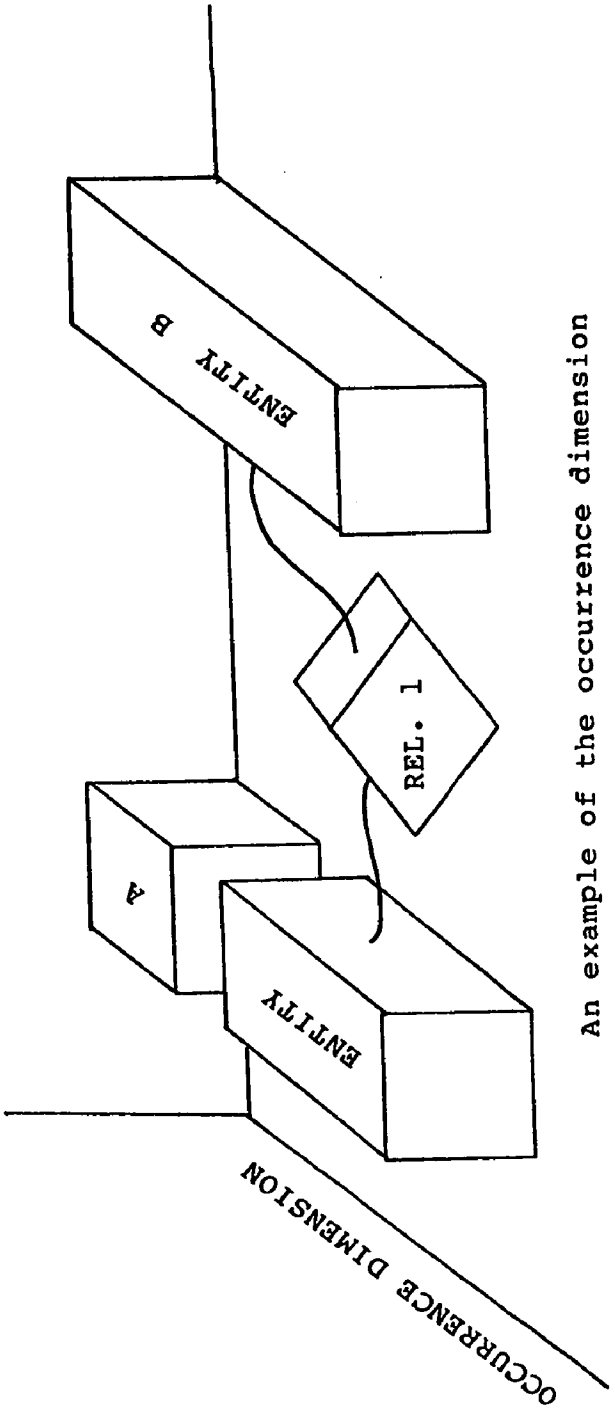
The entities, as characterized by an aggregation of attributes (not shown here), serve to classify objects and events of concern to the enterprise. An entity merely asserts the potential for instances or occurrences of objects of this type to exist in the database at some point in time. Relationships impart added meaning to the entities. Like an entity, a relationship only asserts the potential for an association among entity occurrences at some point in time, it does not imply that all entity occurrences will necessary participate in an occurrence of the relationship.

This two-dimensional portrayal of entities as class objects and relationships as potential associations limits the designer's ability to represent many of the semantically meaningful aspects of an enterprise's information structure because there is no way to conceptualize occurrences. Similarly, the concept of a database as a time-varying collection of data is generally ignored in the process of database design. Again, the chief reason for this is that the time-varying nature of a database is manifest in its instances.

In order to be able to represent the concept of occurrences of entities and relationships in either the abstract or generic models, another design dimension is needed. This third dimension will then permit the representation of occurrences as a conceptually meaningful aspect of both entities and relationships. Figure 2.3

shows how this third (occurrence) dimension can be used to model an aspect of an enterprise's information structure which would otherwise not be representable in two dimensions.

The entity on the left of the figure is portrayed as two disjoint subsets of occurrences of this entity. Although all occurrences of this entity will be logically identical, there is a semantically meaningful reason to introduce the partitioning. As the diagram indicates, the relationship between the two entities is constrained to occur only between a subset of the occurrences of one entity while all occurrences of the other entity may potentially participate in the relationship.



An example of the occurrence dimension

Figure 2.3

Situations such as that portrayed in the figure can arise naturally and frequently when interpreting an enterprise's information requirements. For example, when references to the occurrences of an entity (SELECTion queries) are accompanied by a common selection criterion (WHERE clause), this may indicate the existence of a semantically meaningful partitioning of the entity occurrences which should be explicitly recognized in the design. Additionally, as the figure indicates, this natural partitioning may be a consequence of the meaning associated with a particular relationship. The next section will suggest several such situations in the context of a database design problem.

2.4 A Database Design Problem

In order to demonstrate the importance of the occurrence dimension in enterprise information modeling and to describe its impact on the three level database design methodology, a case study of a database design for a small college will be used. This hypothetical design problem has been specifically devised as a vehicle for identifying the types of situations where recognition of the occurrence dimension can significantly enhance the database designer's ability to capture meaningful aspects of the enterprise's information structure. Also, by following the case through the three level design process,

the operational performance improvements which will accrue as added benefits will be demonstrated by examples.

It is assumed that the college does not presently employ a database management system in its administrative data processing activities. A decision has been made to design a database to organize, structure and manage a part of its data resource. This database will serve a number of existing, structured applications of the college's administration and will also provide ad hoc access to the college's data. As a result of the collection of requirements for this database, the information summarized in Table 2.1 has been obtained.

The objects of interest to the college (entities) for this particular database application are listed in the first column. Beneath each entity name is the approximate number of occurrences of each which will be stored in the database. The second column indicates the relevant attributes which are to be used to characterize the occurrences of each entity. The last column then shows the number of bytes needed to represent each attribute value.

ENTITY	ATTRIBUTES	BYTE LENGTH
DEPARTMENT (25)	Department name	15
	Office number	3
	Phone number	4
MAJOR (45)	Major name	25
	Degree awarded	3
	Required credits	2
STUDENT (4000)	Student number	5
	Student name	20
	Address	40
	Class	9
STUDENT-ACCOUNT (4000)	Student number	5
	Account balance	7
ENROLLMENT (14000)	Grade	1
COURSE (500)	Course number	3
	Course name	30
	Description	100
SECTION (200)	Section number	1
	Room	3
	Time	12
FACULTY (250)	Employee number	4
	Employee name	20
	Title	9
COMMITTEE (30)	Committee name	50
	Number of members	2

Entity and attribute requirements

Table 2.1

In addition to identifying the entities and attributes, the following semantic information has been determined:

1. Each MAJOR course of study is offered by a single DEPARTMENT.

2. STUDENTs, with the exception of freshmen, are required to elect a MAJOR.

3. Each COURSE is given by a single DEPARTMENT but not every COURSE is given in every term.

4. Offerings of a COURSE in a particular term are associated with SECTIONs.

5. Each SECTION of a COURSE is taught by one FACULTY member.

6. FACULTY members may serve on many college COMMITTEEs. In addition, FACULTY members who hold the rank of Professor serve as an advisory body to the president of the college.

7. The college has many different COMMITTEEs.

8. Each STUDENT has a STUDENT-ACCOUNT for financial transactions although only a relatively small number of STUDENTs actively use their STUDENT-ACCOUNTS.

9. STUDENTs enroll in several SECTIONs of COURSEs each term and SECTIONs typically have many ENROLLMENTs. A grade is assigned and recorded for a particular STUDENT in a particular SECTION of a COURSE.

The objective of the college is to design and implement a database which adequately represents its data and information requirements. The data requirements are completely listed in Table 2.1 while some of its information requirements (that is, the interpretation of these data types) are implicit in the semantic statements above. For the most part, these semantic rules will result in the creation of relationships between the entities in

the table. However, some of the semantics expressed in these rules cannot be directly represented in the database design.

For example, rule 6 implies that among all occurrences of FACULTY members, those of professorial rank are to be viewed separately for certain purposes. Similarly, rule 2 states that only those STUDENTs who are not freshmen may participate in a relationship with a MAJOR course of study. While these two rules are intuitively plausible, there are no mechanisms to explicitly represent them in the database design because each rule depends on an attribute value rather than any structural difference.

Likewise, rule 8 calls for associating a STUDENT-ACCOUNT occurrence with each STUDENT occurrence but it also states that only a small number of the STUDENT-ACCOUNT occurrences will be frequently referenced. Rule 4 constrains the association of SECTION occurrences to those COURSEs given in a particular term. Both of these rules imply the partitioning of the occurrences of the respective entities based on a temporal criterion. Again, neither rule can be fully represented in the database design with currently available design mechanisms.

Additional information requirements can be obtained explicitly by examining the requirements of structured application types which must be supported by the database. Typical applications include preparing class lists for

each section; student grade reporting; faculty teaching assignment reporting; committee membership lists; and student account posting and billing. While these application types are not exhaustive, they are representative of the types of applications which would be required in a college and would provide important information to the design process.

This particular database design case is obviously a simplification of what would be involved in a real design of this kind. However, it sufficiently complete to demonstrate the necessity of the occurrence dimension in adequately capturing the information structure of the college. The scenario presented here will be used throughout the remainder of this work.

CHAPTER 3

THE OCCURRENCE DIMENSION IN ABSTRACT DATABASE DESIGN

3.1 Introduction

At the present, the art of abstract database design is primarily concerned with capturing and representing the static, time-invariant aspects of the enterprise's information structure. Design methodologies such as the Entity Relationship Model [CHEN76] provide a framework in which to assemble the relevant data items (attributes) into meaningful units (entities) and to represent associations among them (relationships). While it is recognized that the actual content of the database - the occurrences - will change over time, there is the implicit assumption that the overall design will remain constant for a relatively long time.

The lack of recognition of occurrences of attributes, entities, and relationships in the abstract design limits the ability of the designer to adequately portray many of the semantic information requirements gathered prior to the design effort. Any such information requirements not captured in the abstract design will similarly be ignored

or overlooked in subsequent stages of the design methodology. This chapter will then be concerned with:

1. identifying when semantic information requirements call for recognizing the occurrence dimension in abstract data modeling; and

2. the introduction of new modeling mechanisms which effectively enable the representation of these semantic information requirements in an abstract data model.

In order to accomplish these goals, the next major section will present an overview of abstract data modeling by developing an integrated framework based primarily on the work of Chen [CHEN76]. This framework will also include the contributions of Codd [CODD79] and Smith and Smith [SMIT77a, SMIT77b]. This particular framework represents to a large extent the state-of-the-art in abstract database design as it is currently practiced. Also, the framework will provide a suitable basis for the principal contribution of this work, that is, the introduction of the occurrence dimension. The two following major sections will then be concerned specifically with introducing two new database abstractions which permit the recognition and representation of semantic information requirements in the occurrence dimension.

3.2 A Framework for Abstract Data Modeling

Perhaps the best known and most widely used methodology for abstract data modeling is the Entity Relationship Model, or ERM, [CHEN76]. This model presents

a framework for organizing data in a way which carefully avoids any of the constraints imposed by generic data models or their implementation as database management systems. With a small set of modeling constructs, the designer can portray data and some of its semantics in a model known as the enterprise view.

Recently, considerable attention has been given to ways in which more of the semantics of data can be represented in an abstract data model. The aggregation and generalization hierarchies of Smith and Smith [SMIT77a, SMIT77b] provide two such mechanisms. Codd [CODD79], Hammer and McLeod [HAMM78], and Tsichritzis [TSIC76] also provide modeling constructs to represent additional semantic aspects of data. However, none of these mechanisms explicitly recognize occurrences of data.

Before introducing the two new semantic modeling tools which are defined in the occurrence dimension, a review of the art of abstract data modeling in two dimensions, based on these authors contributions, will be presented. This two-dimensional framework will then be used as a basis for extending the art of abstract data modeling into the occurrence dimension.

Virtually all abstract data modeling methodologies employ three basic building blocks: attributes, entities, and relationships. Named attributes are the smallest units in the abstract design process. Although attributes are defined on domains (value sets), this detail is

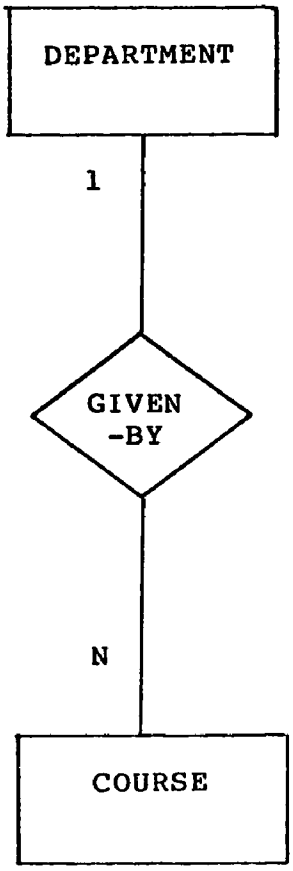
usually suppressed in the abstract model. An attribute, by itself, is unstructured. The name of an attribute signifies the role that it plays in describing a higher-level object in the model. Additional semantics associated with an attribute are represented in the dependencies (functional or multivalued) in which it participates.

Entities are the simplest structural objects in abstract data modeling. Although the exact definition of an entity is the subject of some debate [KENT78], it is generally agreed that an entity serves to represent some object of interest to the enterprise. This object can be thought of as a whole and has a number of properties that are described by attributes. An entity, then, is a named collection of attributes. Its structure is frequently, though not necessarily, considered to be an ordered set of attribute names. Some of the semantics associated with an entity are inherited from the dependency structure of its attributes. Other semantic aspects of the entity are conveyed through the named relationships in which it may participate.

By associating entities through a named relationship, the designer can express more meaning than is conveyed simply by isolated entities and their attributes. The structure of relationships in an abstract model may be either implicit or explicit. Implicit relationships are portrayed in the model as labelled edges connecting two or

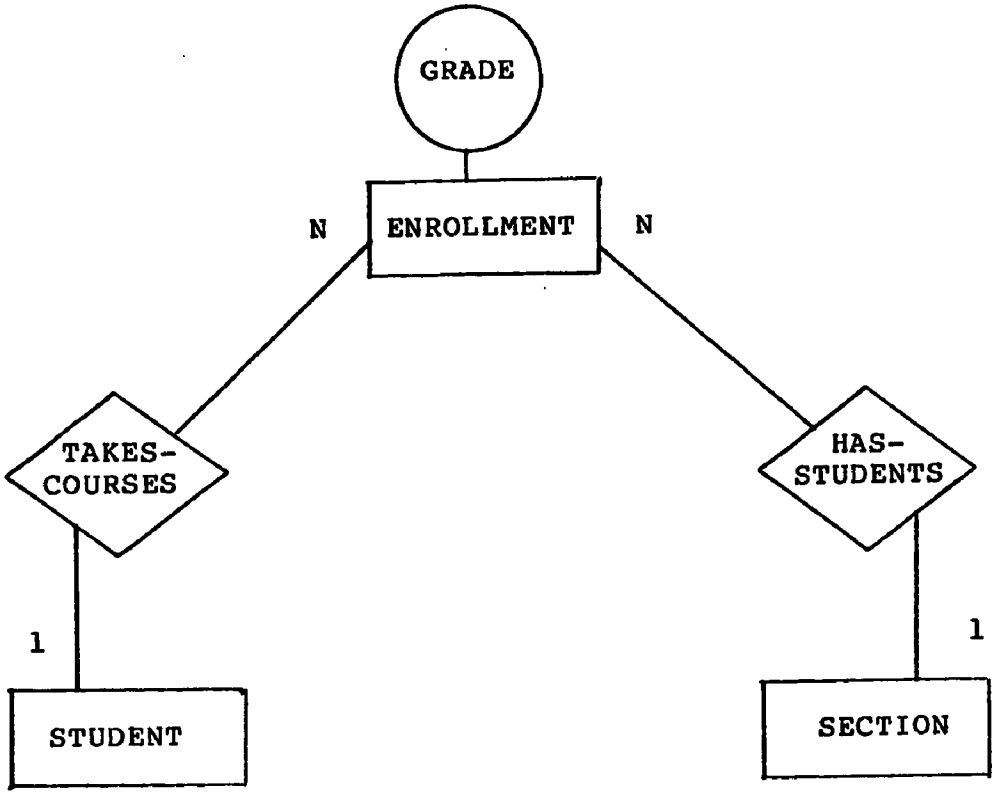
more entities in a data model diagram. While the desired association is represented, the relationship itself has no objective reality, i.e., it has no properties (attributes) of its own. For example, in Figure 3.1, the two entities DEPARTMENT and COURSE are joined by the implicit relationship GIVEN-BY which asserts that COURSEs are associated with DEPARTMENTs which have responsibility for, and control over them. Note that the relationship is bidirectional and could have been labelled "HAS-COURSE" depending on the perspective of the designer.

Explicit relationships, on the other hand, do have objective reality and are represented as entities in their own right. Figure 3.2 portrays a relationship between a STUDENT entity and a SECTION entity as the entity ENROLLMENT. This relationship is made explicitly because the attribute GRADE characterizes the ENROLLMENT of a particular STUDENT in a particular SECTION. The attribute GRADE is not a characteristic of a STUDENT or a SECTION individually. Note that two implicit relationships serve to connect the STUDENT and SECTION entities to ENROLLMENT.



An implicit relationship

Figure 3.1



An explicit relationship with attribute

Figure 3.2

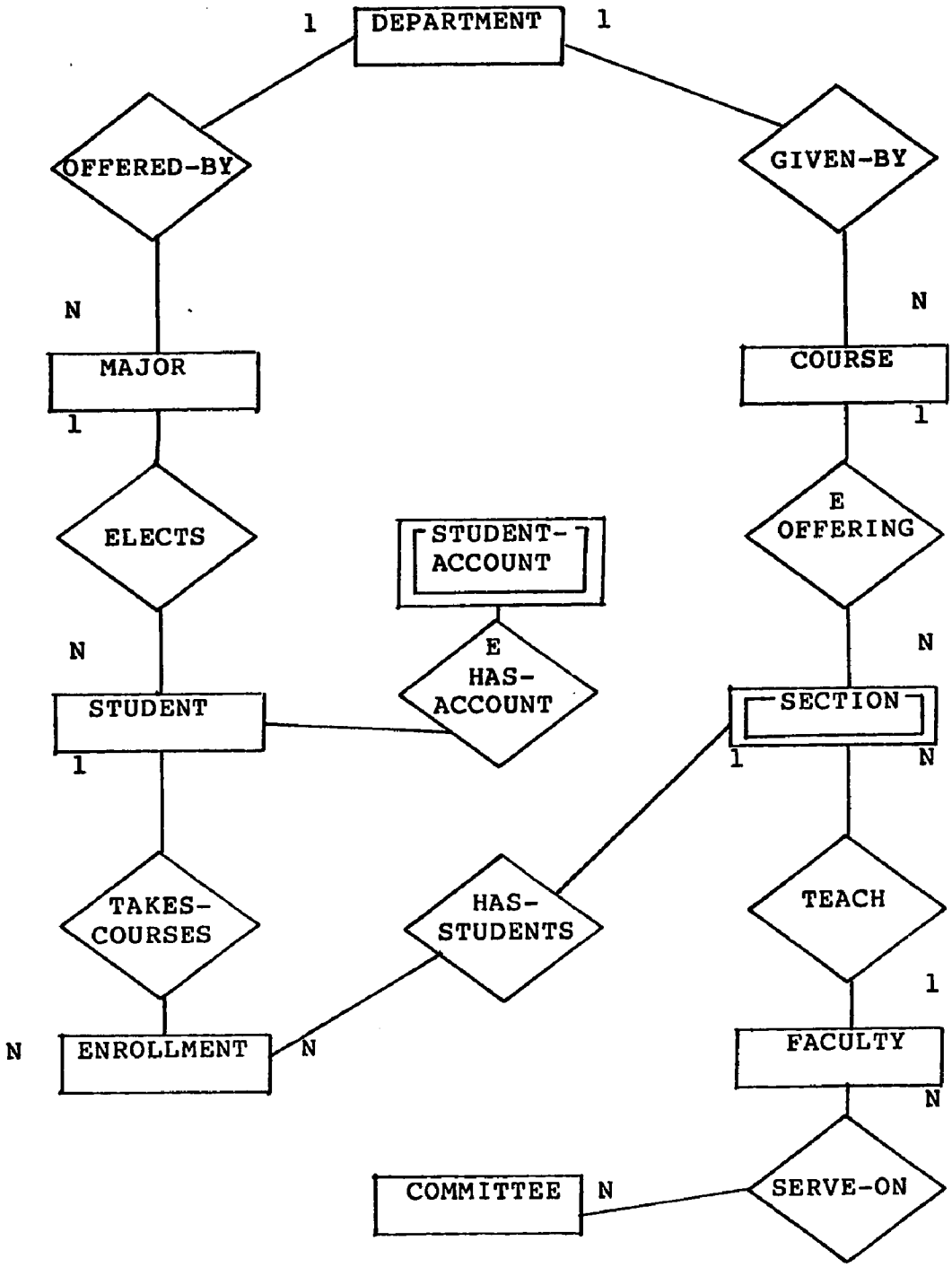
One other semantically meaningful aspect of a relationship is the cardinality of the relationship. A relationship, either implicit or explicit, can be classified as being functional (one-to-one or many-to-one) or complex (many-to-many). Although this classification of relationships has consequences on physical design decisions, in the abstract model the cardinality is simply noted on the model diagram as in Figure 3.2. Both implicit relationships are functional (many-to-one) from ENROLLMENT.

These three basic building blocks of abstract data models are well-known. In the quest to capture more meaning in the abstract modeling process, different authors have enhanced the notions of entities and relationships by defining different types with very special meanings.

In the ERM, Chen [CHEN76] differentiates between regular entities and weak entities. Regular entities exist regardless of their association with other entities in the model. In Figure 3.3, the entity DEPARTMENT is a regular entity in that it exists independently of COURSEs. The entity SECTION, however, is a weak entity because it depends on the existence of a particular COURSE. It does not make sense to have a SECTION without a COURSE. The weak entity SECTION is portrayed diagrammatically as a double rectangular box with the letter "E" in the relationship OFFERING indicating the existence dependency

on COURSE. Likewise, STUDENT-ACCOUNT derives its existence from a STUDENT.

Similarly, explicit relationships are either regular or weak depending on whether the entities they join are regular or weak. The explicit relationship MAJOR in Figure 3.3 is a regular relationship. It serves to connect the regular entities STUDENT and DEPARTMENT based on the election of a particular course of study. A MAJOR can exist without a particular STUDENT or DEPARTMENT and it has its own properties such as a name, a degree awarded, and the number of credits required for successful completion. The weak relationship ENROLLMENT joins the regular entity STUDENT with the weak entity SECTION. This particular relationship cannot exist without a SECTION.



An Entity Relationship Model of the college

Figure 3.3

In the Relational Model/Tasmania (RM/T), Codd [CODD79] presents a slightly different classification of entities and relationships. Entities are categorized as being either kernel, characteristic, or associative. A kernel entity is like a regular entity in that it has independent existence. A characteristic entity derives its existence from a superior entity (either characteristic, kernel, or associative) in the same way as a weak entity. An associative entity is like a regular relationship in that it serves to join two or more other entities (kernel or associative) and it has independent existence. An associative entity may have characteristic entities subordinate to it as well. For completeness, Codd also mentions the possibility of weak relationships which he calls "nonentity associations." These objects, while possibly having their own attributes, do depend on other entities for their existence, whence the play on terms.

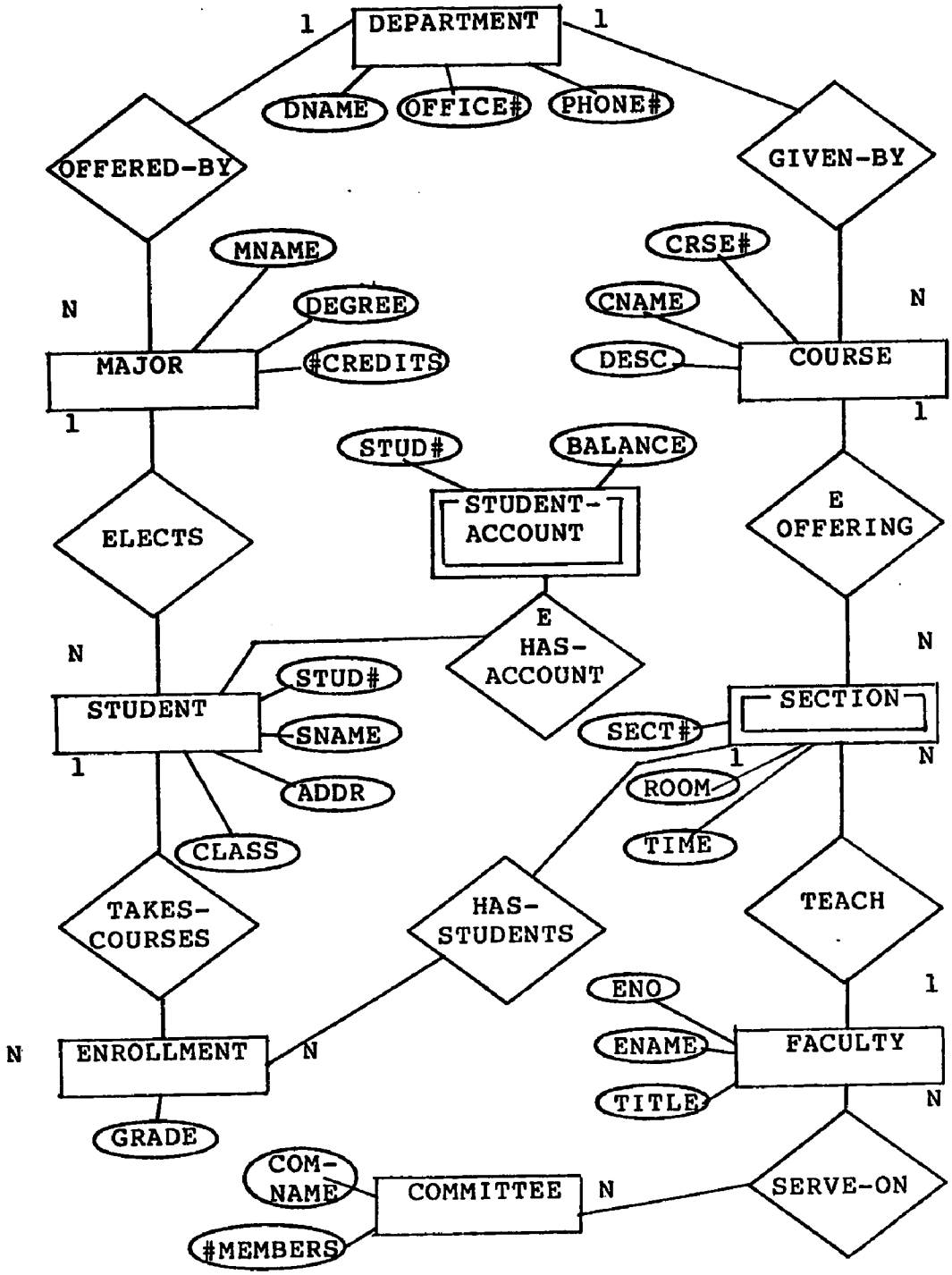
In Figure 3.3, DEPARTMENT, STUDENT, and COURSE are kernel entities. SECTION and STUDENT-ACCOUNT are characteristic entities. MAJOR is an associative entity while ENROLLMENT is a nonentity association. This categorization of entities and relationships in RM/T is quite useful in abstract data modeling, however, it should be reiterated that Codd's intention is to extend the relational model of data, not to propose an abstract data modeling methodology.

These refinements on the traditional concepts of entity and relationship enhance a database designer's ability to incorporate more meaning into the abstract design of a database. Smith and Smith [SMIT77a, SMIT77b], however, have introduced two database abstractions which extend these concepts even further. The aggregation and generalization abstractions enable the database designer to express more complex semantic interpretation about the entities and relationships in a model. Both of these abstractions are related to concepts already used in knowledge-based systems (artificial intelligence) and abstract data types (programming languages). Aggregation is related to the PART-OF notion from AI and corresponds to the cartesian product abstract data type. Generalization comes from the IS-A notion in AI and corresponds to the discriminated union data type.

The aggregation abstraction involves taking two or more objects in an abstract data model and forming a higher-level object from them, hence the term aggregation hierarchy. The lower-level objects do not cease to exist but gain added meaning through the aggregate object. In their original work, Smith and Smith [SMIT77a, SMIT77b] described aggregation in several different aspects. Codd [CODD79] prefers to call their aggregation abstraction a cartesian aggregation to differentiate it from other forms of aggregation such as the cover aggregation of Hammer and McLeod [HAMM78] and statistical aggregation.

In this integrative summary of abstract data modeling, three forms of cartesian aggregation will be described. The first will be called simple cartesian aggregation. The aggregate object resulting from simple cartesian aggregation is the aforementioned entity. By collecting a set of related attributes together in a semantically meaningful way, either a kernel or characteristic entity is formed. Figure 3.4 portrays the same abstract model as in Figure 3.3 with the addition of attribute names. The kernel entity DEPARTMENT is a simple cartesian aggregation of the attributes DNAME, OFFICE#, and PHONE#. Similarly, the characteristic entity SECTION is a simple cartesian aggregation of the attributes SECT#, ROOM, and TIME.

The second form of cartesian aggregation is associative cartesian aggregation. In this form of database abstraction, the aggregation involves forming a high-level object from two or more entities along with any attributes which serve to characterize it. The high-level object is also treated as an entity with independent existence, i.e., it represents a regular relationship [CHEN76] or an associative entity [CODD79]. The entity MAJOR is an example of an associative cartesian aggregation in that it plays a superordinate role in relating STUDENTs to DEPARTMENTs through their elected MAJOR. Its existence, however, is not dependent on either subordinate entity.



An abstract model with attributes

Figure 3.4

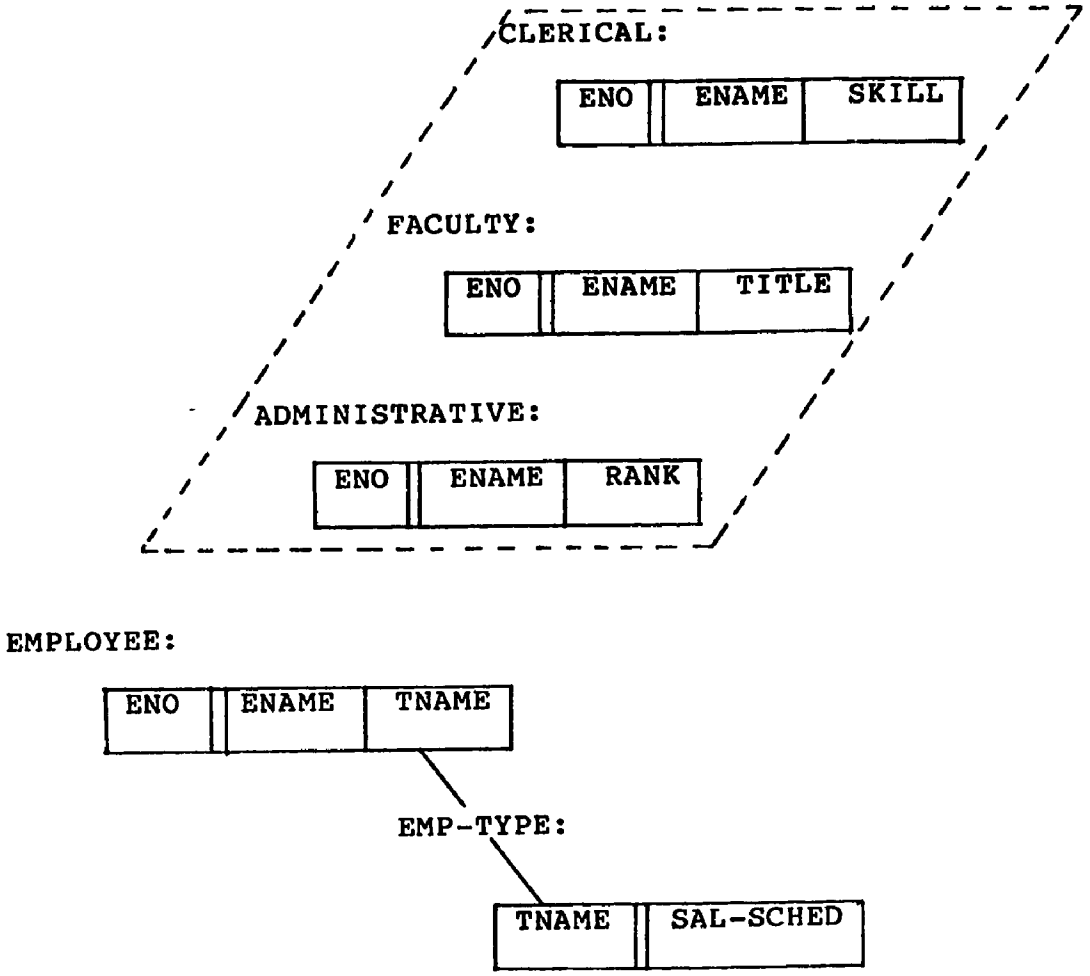
Simple cartesian aggregation and associative cartesian aggregation formalize the concepts of entity and explicit relationship. Cover aggregation is much more general. A cover aggregate object (another entity) serves to relate objects of either the same or different types. For example, the entity COMMITTEE in Figure 3.4 is a cover aggregate which serves to relate (not necessarily disjoint) subsets of FACULTY members by virtue of their COMMITTEE assignments. If STUDENTs were allowed to be members of certain COMMITTEEs, then the cover aggregate concept would span two entities. It is evident that the cover aggregate captures an important, albeit subtle, semantic aspect of an enterprise's information structure.

The generalization abstraction involves identifying a collection of entities, abstracting away their individual differences, and forming high-level generic objects (entities) which represent their common properties. The inverse of generalization is called specialization.

In forming a generalization hierarchy, two new entities are involved. The first (required) entity stands for the generic object as a whole. It has the properties (attributes) common to all members of the hierarchy. The second (optional) entity contains the properties which are relevant to each specialized entity in the hierarchy as they are viewed collectively. Figure 3.5 portrays a generalization abstraction using the diagraming technique proposed in Smith and Smith [SMIT77b].

Three mutually exclusive subtypes of employees are represented in the diagram. These three subtypes constitute a categorization of employees. The entity EMPLOYEE is the generic object in this hierarchy and contains the attributes which are common to all employees regardless of their subtype. The characteristic entity EMP-TYPE is included in the generalization hierarchy because there is an attribute (SAL-SCHED) which is a property of each subtype viewed collectively. The lowest level in the hierarchy consists of the entities representing each subtype and it contains those attributes which are relevant to each subtype individually.

With this particular generalization hierarchy, all instances of employees stored in the database will be required to be represented as an EMPLOYEE. If it is clear that an instance of an employee is also a member of one of the subtypes then it must be entered into that subtype as well. Note that membership in a subtype of a category may be optional. While the subtypes in a category are mutually exclusive within the category, they are not necessarily collectively exhaustive.



A generalization hierarchy

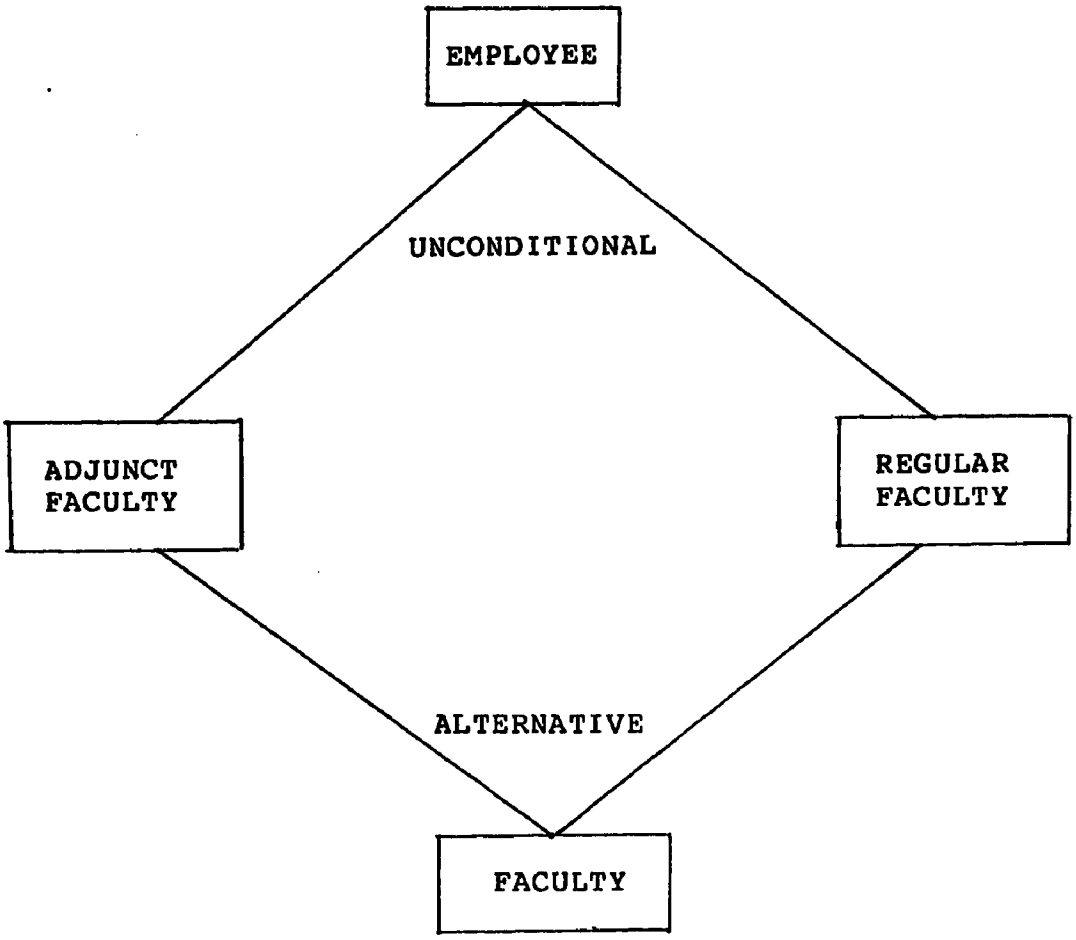
Figure 3.5

The generalization hierarchy appears to have a great deal of redundancy because attributes are repeated down the hierarchy. In the abstract data model this merely conveys the idea that lower level entities in the hierarchy inherit the properties (attributes) of higher level entities. Smith and Smith have noted, however, that in a physical implementation this redundancy may be effectively controlled. The semantic notion captured in a generalization hierarchy is that different users have different views of data depending on their level of abstraction. A dean may be interested in FACULTY employees only while the director of personnel may be interested in all employees regardless of their subtype. The generalization abstraction permits these multiple views to be explicitly represented in the abstract data model.

A particularly important aspect of this abstraction is the ability to represent relationships, either explicit or implicit, among entities at lower levels (say, the subtype level) which may not be applicable at higher levels. For example, in Figure 3.5, an implicit relationship TEACHES between FACULTY and SECTION is not appropriate to either ADMINISTRATORs or CLERICALs. Smith [SMIT78] discusses this in more detail.

In RM/T, Codd refines the generalization abstraction by describing two different kinds. The first kind is called an unconditional generalization and is exactly as described above. Each subtype entity is constrained to

belong to a single parent generic object in a generalization hierarchy. The second kind is known as an alternative generalization. In this case, a subtype entity may be generalized into any of several parent generic objects. Figure 3.6 portrays both of these forms of generalization. The entity FACULTY may alternatively be generalized into either ADJUNCT FACULTY or REGULAR FACULTY. Both of these are unconditionally generalized into EMPLOYEE as before.



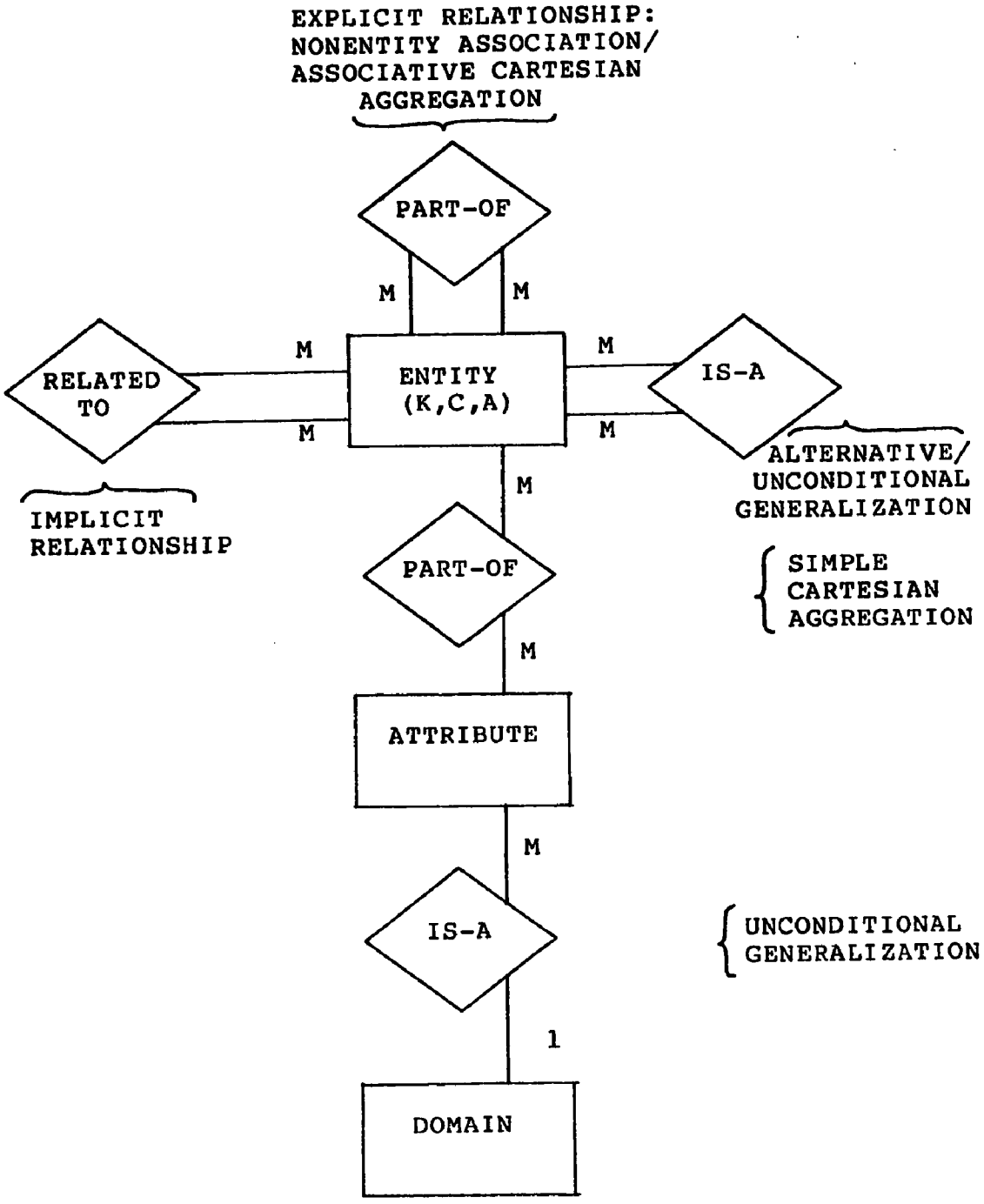
Unconditional and alternative generalization hierarchies

Figure 3.6

This section has integrated a number of concepts and ideas into a modeling framework which represents, to a large extent, the state-of-the-art in abstract data modeling. The framework, however, does not explicitly account for occurrences of the relevant objects - either entities or relationships. To summarize this framework, Figure 3.7 portrays most of the concepts covered utilizing the diagraming conventions of the ERM.

The key element of the figure is the notion of the entity, whether kernel(K), characteristic(C), or associative(A). The entity is represented as a simple cartesian aggregation of attributes which, in turn, are unconditional generalizations of domains. Implicit relationships among entities are represented directly by the implicit relationship RELATED-TO. Explicit relationships, either associative cartesian aggregations or nonentity associations, are shown using the PART-OF relationship. Lastly, unconditional generalizations or alternative generalizations among entities are shown with the IS-A relationship.

This framework will now be used as the basis for extending the art of abstract data modeling into the occurrence dimension. The next two sections will present new database abstractions which are defined only in the occurrence dimension.



An abstract model of the abstract data modeling process

Figure 3.7

3.3 A Static, Intra-Entity Abstraction: the Selector

The two-dimensional, abstract data modeling framework reviewed in the preceding section is adequate to capture and represent the majority of the semantic information requirements that would arise in a database design effort. However, some of the semantic rules associated with an enterprise's data cannot be represented with the available abstract modeling tools. The reason is that these semantic rules are defined on the occurrences of entities or relationships and the nature of the abstraction process is to suppress the consideration of occurrences, at least until the very last stage of database design.

One type of situation where this problem might arise occurs when it is necessary to specialize an entity on the basis of one of its attribute values. Unlike the specialization that takes place in a generalization hierarchy, this form of specialization does not create any new entities or relationships but rather calls for subsetting the occurrences of an entity based on the particular attribute value.

Tsichritzis [TSIC76] has addressed this type of subsetting at the level of the internal data model in the LSL (Link and Selector Language) database management system. The device used in LSL to represent such a partition is called a selector. This same term will also be used here but with a more precise definition.

Two different forms of selector will now be introduced which provide the database designer with the ability to capture this type of semantic information requirement at the level of the abstract data model by extending the modeling environment into the third, occurrence dimension. It will be shown in the succeeding chapters that these two new forms of database abstraction have important consequences at the lower levels of database design as well as in data manipulation operations on an actual stored database.

As an abstract data modeling tool, each form of selector abstraction is defined as the specialization of the occurrences of an entity based on a boolean qualification involving a single attribute and a constant selected from its value set. As stated previously, the characteristics of the underlying domain of an attribute are generally suppressed in the process of abstract data modeling. However, for a selector abstraction it is important to consider two important characteristics of the attribute.

One characteristic has to do with the cardinality of the attribute's value set, that is, the number of possible values contained in its domain, or more precisely, what is the cardinality of the range of values that the attribute will assume. For example, the attribute SNAME of the STUDENT entity has a domain and a range which are quite large assuming there are many STUDENT's in the college.

The attribute CLASS of STUDENT however has a domain and range consisting of only four values (FRESHMAN, SOPHOMORE, JUNIOR, SENIOR).

The second characteristic is concerned with the volatility of the attribute values. Certain attributes are such that once a value is assigned, it is very likely to be a "permanent" characteristic of the associated entity occurrence. The attribute SNAME is an example of a "permanent" characteristic of a STUDENT. Similarly, CLASS may be considered "permanent" even though it may change annually. Conversely, certain attributes will be the object of frequent value changes. Although not portrayed in Figure 3.4, if the DEPARTMENT entity also had the attributes BUDGET-ALLOCATED, BUDGET-SPENT, and BUDGET-COMMITTED, the last two attributes would likely be the subject of frequent changes through updates. Also, the underlying domains and ranges are very large.

When a selector is defined on an entity type in an abstract design, there may be some implications concerning the relationships in which that entity participates. In one case, the selector merely defines a partition over the entity occurrences based on a constant attribute value and any relationships involve the entity as a whole. This form of selector will be referred to as a simple selector.

On the other hand, it may be the case that one or more relationships involve only the selected subset while other relationships may be directed to the entity as a

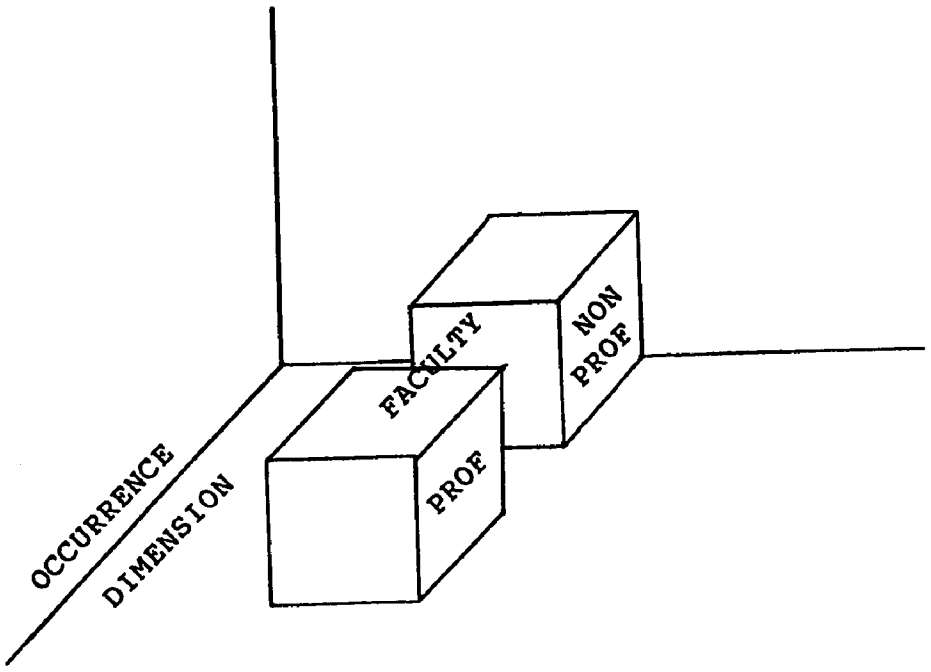
whole. When a relationship involves only the selected subset, its definition and subsequent manipulation will have to be treated differently. This form of selector will be referred to as a relationship selector. This term reinforces the notion that the purpose of the selector is to subset an entity in order to support a relationship which is meaningful only to the selected subset.

For the database design problem described in the preceding chapter, semantic rule 6 implied that among all occurrences of the entity FACULTY, those FACULTY members with professorial rank were to be viewed as logically separate from FACULTY members in general. This rule could not be represented in the two-dimensional framework as is evident in the representation of the entity FACULTY in Figure 3.4.

Figure 3.8 shows how this semantic information requirement would be portrayed in the occurrence dimension. Figure 3.9 then shows a possible diagrammatic representation for a simple selector abstraction defined on the entity FACULTY in two dimensions. The rectangular box directly above the entity contains a name for the simple selector, in this case SENIOR, as well as its formal definition in terms of an attribute of FACULTY. The selected subset of FACULTY consists of those occurrences of FACULTY whose attribute TITLE equals PROFESSOR. This simple selector enables the designer to explicitly represent the fact that some users' information

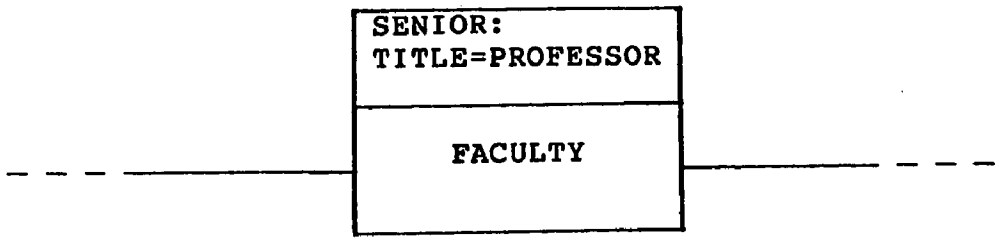
requirements involve only senior faculty members (i.e., full professors) and not all faculty members. The two relationships involving the entity FACULTY, however, are directed to all faculty members.

It should be noted that while a simple selector does explicitly capture a semantic notion in the abstract design, it is not a necessity. Without the concept of a simple selector, applications which reference the entity in question can always determine the desired subset dynamically by checking all of the entity occurrences and verifying the boolean qualification at that time. The implication of defining a simple selector in the abstract design is that presumably there will be some mechanism for representing the simple selector at lower levels of database design. The next chapter will discuss not only the mechanics of defining simple selectors in the conceptual schema, but will also introduce new data manipulation operators which will exploit their existence in responding to queries.



A simple selector in the occurrence dimension

Figure 3.8



A diagram of a simple selector abstraction
in two dimensions

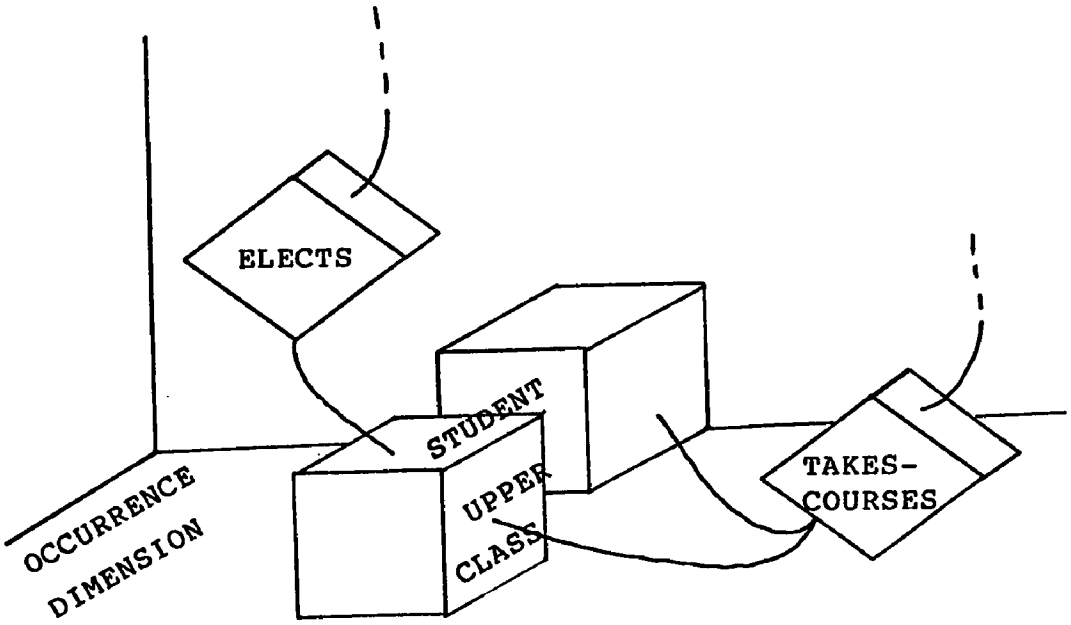
Figure 3.9

In the database design problem being followed here, semantic rule 2 states that it is only legitimate for STUDENTs who are not freshmen to participate in the relationship with a MAJOR course of study. Again, the two dimensional framework for abstract data modeling does not provide a mechanism for representing this semantic information requirement. Observing the situation portrayed in Figure 3.4, it is not apparent that only a subset of STUDENTs may be related to MAJORs. The implication is that any STUDENT may ELECT a MAJOR.

Figure 3.10 shows how this rule would be effectively represented in the occurrence dimension. Figure 3.11 then shows the diagrammatic representation for a relationship selector in two dimensions. In the rectangular box above the entity STUDENT is the name of the selector, UPPERCLASS, followed by the attribute name on which the selector is defined and the constant attribute value to which its compared. In this case, the attribute is CLASS and the relevant value is FRESHMAN. Relationships involving STUDENTs may be directed to all STUDENTs or to only those STUDENTs meeting the qualification. For example, only STUDENTs who are not FRESHMAN may participate in the relationship ELECT while all STUDENTs may participate in the relationship TAKES-COURSES.

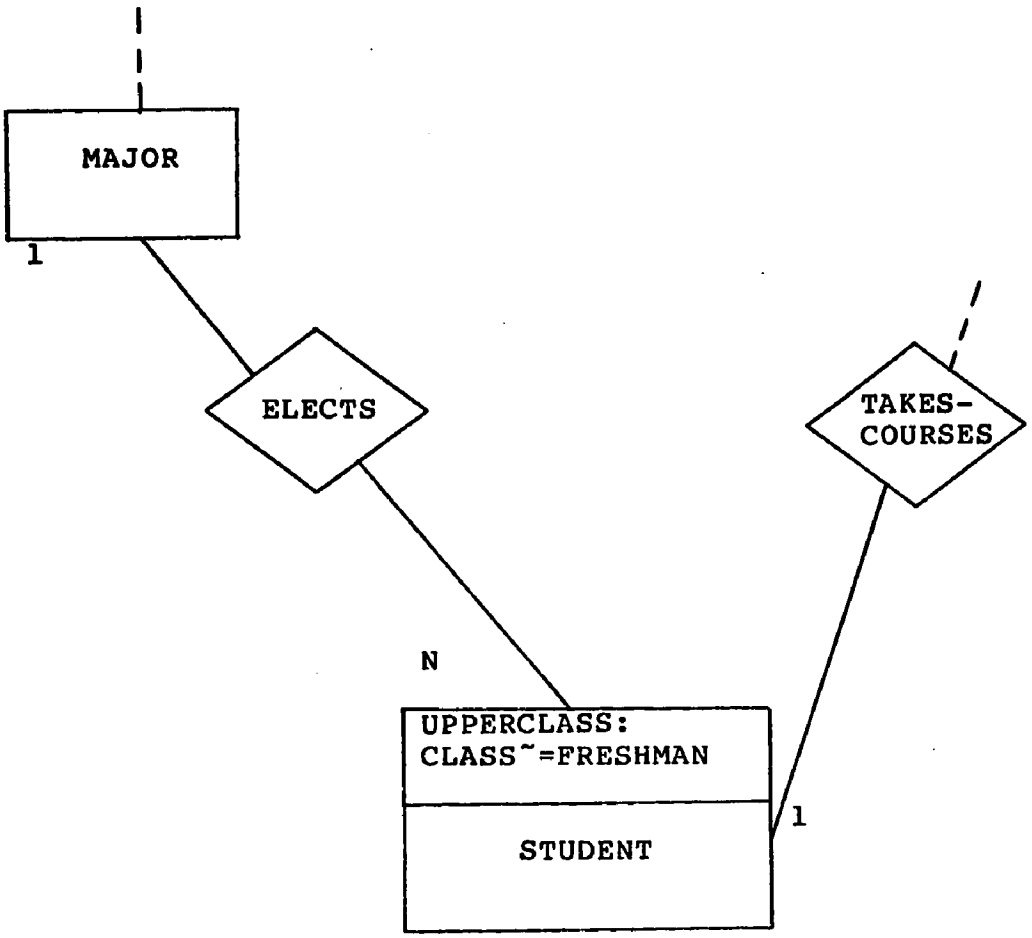
While a simple selector abstraction is not a necessary aspect of abstract design, a relationship selector does play a very important role in the design

process. A requirement that a relationship is valid only for a particular subset cannot be incorporated into a design without such a mechanism. In the absence of a relationship selector, it is incumbent on the database administrator to ensure that the proper precautions are taken to preserve the integrity of the stored database. If such precautions are not taken, it is possible that the stored database will violate the semantic rule concerning the relationship.



A relationship selector in the occurrence dimension

Figure 3.10



A relationship selector abstraction in two dimensions

Figure 3.11

Adding the concept of selectors to the abstract data modeling process provides yet another tool to incorporate more meaning into the earliest stage of database design. The selector mechanism does not affect the structure of the abstract data model nor does it create any new entities or relationships. The selector abstraction does, however, enable the explicit representation of certain semantic information requirements in the abstract design of a database. Inserting, deleting, or modifying instances of FACULTY or STUDENTs in the physical database implementation will be constrained to abide by their qualification with respect to the defined selectors.

3.4 A Dynamic, Intra-Entity Abstraction: the Adaptive Selector

The art of abstract data modeling, even with the selector abstractions just defined, is limited to representing the static, relatively time-invariant properties of an enterprise's information structure. Because an enterprise is a dynamic, on-going concern, its information requirements will likely contain dynamic, time-varying properties as well.

Among the reasons that time-varying properties of data are not presently representable in the abstract data modeling methodologies is that these properties involve the concept of occurrences and they typically arise through circumstances external to the modeling

environment. In collecting the requirements for the database design, certain semantic rules concerning the data may be postulated in a narrative fashion which do not specifically refer to any concrete structural characteristic of the model, for example, an attribute value. Rather, it may be that the stated requirement is temporal in nature.

A new database abstraction is needed to permit the representation of these dynamic, time-varying semantic rules in abstract data modeling. The nature of this abstraction will be to explicitly recognize that at any given moment in time, a proper subset of an entity has some special meaning within the model. Unlike the selector abstraction introduced in the preceding section, the membership of this subset is in no way "permanent" and may possibly be empty at certain times.

This new database abstraction introduced here for the first time will be called an adaptive selector. The term "selector" is used because of the resemblance of this mechanism to the selector abstraction described above. The principal difference is that the adaptive selector is not defined on an attribute of the entity. The adjective "adaptive" conveys the notion that the abstraction is intended to represent a time-varying property of the entity. In this way, adaptability may be explicitly recognized in the abstract data model as a semantically

meaningful characteristic of an enterprise's information structure.

The aggregation and generalization abstractions were based on the abstract data types of cartesian product and discriminated union, respectively. The adaptive selector can be directly related to the abstract data type of powerset [HOAR72]. In mathematics, a powerset is defined as the set of all subsets of a given set. As an abstract data type, the powerset is defined with respect to some other data type called the base type. A variable defined on a base type is single-valued; at any instant in time it may take on only one value from the base type. A variable defined on a powerset is set-valued; at any instant in time it may take on a set of values selected from the base type. In abstract data modeling, the role of the base type will be played by an entity and the adaptive selector will be defined as a powerset over occurrences of the entity.

As with the selector abstraction, an adaptive selector will be defined on a specific entity and this naturally leads to questions concerning the relationships in which the entity participates. Two forms of selector were defined depending on whether a relationship could be directed exclusively to the selected subset. Regardless of the form of selector used, its definition remained permanent with respect to the attribute name and the constant value to which the attribute is compared. In the adaptive selector abstraction, two forms will also be

defined depending on whether or not a relationship is involved.

In the first form of adaptive selector, the intent is to identify and represent that subset of an entity which is currently of more interest to the enterprise. A well-known folk theorem in computer science, with corollaries in numerous other disciplines, states that 80 percent of the references to the records in a file will be directed at only 20 percent of the record occurrences. This first form of adaptive selector will explicitly recognize this phenomenon in the abstract design. While there is nothing magic about the 80/20 split, this "theorem" has been empirically verified in data processing environments by Heising [HEIS63].

A way to conceptualize this form of adaptive selector is to think of it as a modified "push-down stack." While a conventional push-down stack maintains a first-in, first-out discipline with the most recently referenced item at the top of the stack, the modified stack referred to here will have a joint criterion for a stack maintenance policy. On the one hand, those entity occurrences most recently referenced will be in the stack, but the stack maintenance policy will also tend to favor those entity occurrences which have been most frequently referenced over the recent past. In this way, not only will the content of the stack be changing dynamically, but the stack will be reasonably assured of always holding the

subset of entity occurrences of most interest to the enterprise at any point in time. The details of how this form of adaptive selector will actually be implemented will be discussed in Chapter 5.

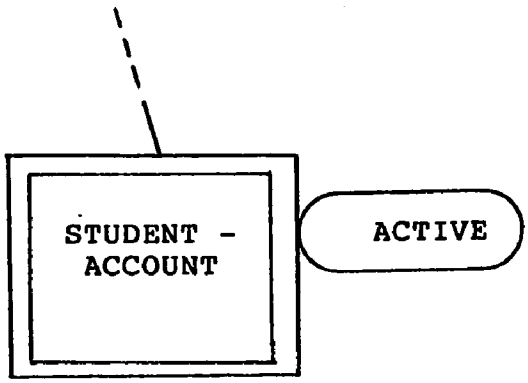
The highly dynamic nature of this first form of adaptive selector along with the fact that its membership is determined by criteria external to the model (that is, observed recency and frequency of access), precludes any relationships from being directed to it. To differentiate this form of adaptive selector from the second form, it will be called a simple adaptive selector.

In the database design problem described in the preceding chapter, semantic rule 8 calls for a STUDENT-ACCOUNT entity occurrence to be associated with each STUDENT entity occurrence. The STUDENT-ACCOUNT entity contains data about a STUDENT's billing activity. At a minimum, a STUDENT-ACCOUNT occurrence will be debited and credited once a term for tuition, room, and board charges. However, certain STUDENTs make regular, frequent use of their accounts for such things as bookstore purchases, theater and sports event tickets, and snack-bar items. The STUDENT-ACCOUNTs for these STUDENTs are most likely to be the object of the vast majority of references during certain periods of time.

Figure 3.12 portrays how, in two dimensions, a simple adaptive selector, named ACTIVE, could be defined on the entity STUDENT-ACCOUNT. With the simple adaptive selector

ACTIVE, the existence of such a temporally defined subset is explicitly recognized.

In describing the simple selector abstraction in the preceding section, it was noted that while this abstraction provided a useful abstract design tool, it was not essential to the abstract design process. The boolean qualification used to determine the selected subset could always be applied when the actual stored database is manipulated. Similarly, the simple adaptive selector is not essential in abstract design. When one is incorporated into an abstract design, the database designer is simply recognizing the fact that a certain time-varying subset of the entity will be the object of more frequent reference. The effect of having declared the existence of a simple adaptive selector will be manifested in the storage structure and access path decisions at the level of the internal schema. These issues will be discussed in detail in Chapter 5.



A diagram of a simple adaptive selector abstraction

Figure 3.12

The definition of the second form of adaptive selector is somewhat more precise than that of the simple adaptive selector. Rather than using the highly dynamic joint criterion of recency and frequency of reference to identify a particular subset of an entity, this second form will rely on the entity occurrence's participation in a relationship to qualify it for membership. In the relationship selector, the converse was true, that is, an entity occurrence's membership in that form of selector was a necessary condition for its participation in a relationship.

This form of adaptive selector will be referred to as a relationship adaptive selector. The intent of this database abstraction is to permit the database designer to explicitly represent the semantic information requirement that participation in a relationship occurrence is a temporal characteristic of entity occurrences. The membership of the subset identified by a relationship adaptive selector share the temporal quality of "currency." That is, those entity occurrences which are selected at any given moment in time are of particular importance to the enterprise at that time.

While the simple adaptive selector assumed a relatively small proportion of an entity's occurrences would be selected (e.g., 20 percent), the relationship adaptive selector has no such limitation on its membership size. As a matter of practicality, however, it would be

virtually useless to define one when it was expected that nearly all entity occurrences would qualify for membership. This would be have to be decided by the designer.

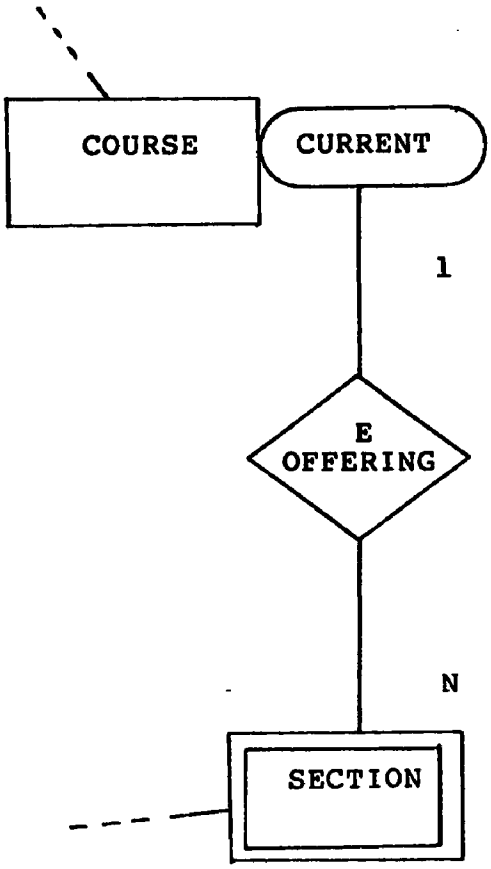
As with the simple adaptive selector, the implications of defining a relationship adaptive selector will be manifest at the level of the internal data model where storage structure selection and access path determination decisions are made. The presence of a relationship adaptive selector in the abstract data model, however, does convey the required additional semantic meaning from the outset of the design process.

The type of situation which would give rise to the need for a relationship adaptive selector is contained in semantic rule 4 of the database design problem of Chapter 2. This rule states that, while the college has many COURSEs GIVEN-BY DEPARTMENTs, only those COURSEs offered in a given term may be related to SECTIONs through the relationship OFFERING. Creating and associating a SECTION occurrence with a COURSE occurrence makes the COURSE "current." Numerous information requirements may be dependent on this notion. For example, the preparation of class lists by COURSE and the transitive association of FACULTY with COURSEs is only meaningful for those COURSEs which are current, i.e., have SECTIONs.

Figure 3.13 shows the diagrammatic representation of a relationship adaptive selector in two dimensions. In

this case, a relationship adaptive selector named CURRENT is defined on the entity COURSE and the relationship OFFERING is directed to it rather than to the COURSE entity as a whole.

Smith and Smith [SMIT77b] require that aggregate objects and generic objects created by the aggregation and generalization abstractions be namable by simple English nouns. Although this requirement is somewhat imprecise [CODD79], it provides an intuitive way to express the meaning of these objects. For the adaptive selector abstraction, an adjective can be used to name it. The adjective should denote the temporal property which characterizes the subset of an entity which is being identified. This, too, is an imprecise requirement and should be accompanied by an external (to the model) statement of its exact definition.



A relationship adaptive selector

Figure 3.13

The two forms of adaptive selector just described serve similar purposes in identifying a meaningful subset of an entity which would otherwise not be representable. They are, however, quite different in the way they are defined. In the case of the relationship adaptive selector CURRENT, a predicate may be formulated to test whether a particular entity occurrence "belongs" to the adaptive selector at a point in time. This predicate involves the observation of some real world fact. For example, a COURSE necessarily belongs to CURRENT if an existing SECTION is associated with it. When a SECTION is inserted for a non-CURRENT COURSE, it automatically becomes CURRENT. Similarly, when the last SECTION of a CURRENT COURSE is removed, the COURSE is no longer CURRENT.

A simple adaptive selector does not have such a predicate to determine its membership. At the abstract design level, the designer may want to recognize that among the occurrences of an entity, a certain subset of them will be more "meaningful" than the others at any given time. In the case of the simple adaptive selector ACTIVE, the designer wishes to express the notion that there will be a subset of STUDENT-ACCOUNTs about which information is more frequently needed. Membership in this simple adaptive selector must then be based on observed frequency of reference to particular entity occurrences over a period of time. The exact composition of this ACTIVE subset will presumably change over time as well.

In the abstract design process it is sufficient to be able to indicate the existence of adaptive selectors in the model. The details of how these adaptive selectors will be implemented are left to later stages of the design process and will be addressed in the next two chapters.

3.5 Summary

This chapter has been focused on the art and practice of abstract database design. This constitutes the highest level of the design process in the three level methodology portrayed in Figure 2.1. To provide a basis for extending abstract data modeling into the occurrence dimension, a review of the present state of abstract modeling methodologies was presented first.

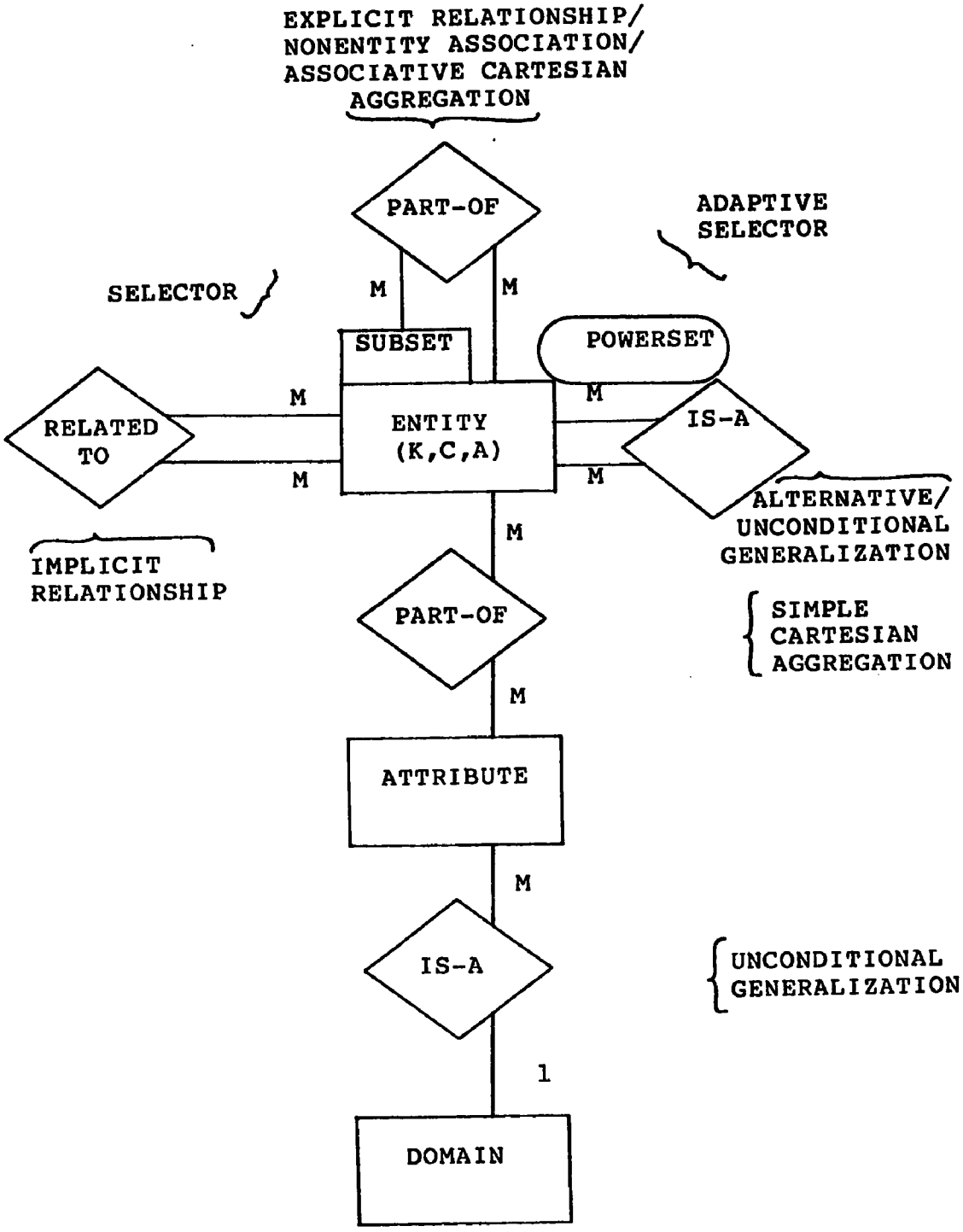
Beginning with the Entity Relationship Model [CHEN76], the contributions of Smith and Smith [SMIT77a, SMIT77b] and Codd [CODD79] were integrated to provide a comprehensive, two-dimensional abstract data modeling framework. This framework incorporates not only the basic syntatic elements of abstract data modeling but also includes the semantic notions of aggregation and generalization hierarchies, cover aggregation, and conditional and unconditional generalization. This particular framework is summarized in Figure 3.7.

The major contribution of this chapter toward the art and practice of abstract data modeling has been the introduction, definition, and demonstration of two new

database abstractions which permit the representation of certain semantic information requirements in the occurrence dimension. Both the selector and adaptive selector database abstractions were presented in two different forms:

- The simple selector subsets the occurrences of an entity based on a boolean qualification of one of its attribute values. Any relationships in which the entity participates involve all occurrences.
- The relationship selector also subsets the occurrences of an entity based upon a boolean qualification of an attribute value. Here, however, the participation of entity occurrences in a particular relationship is predicated on the selection criterion.
- The simple adaptive selector subsets the occurrences of an entity based on the temporal, and externally defined, criterion of recency and frequency of reference. All relationships in which the entity participates may be directed to all occurrences regardless of their selection by this abstraction.
- The relationship adaptive selector also subsets the occurrences of an entity based upon a temporal criterion; however, this criterion is related to participation of the entity occurrences in a particular relationship.

Using the database design problem defined in Chapter 2, representative situations which call for the explicit recognition of the occurrence dimension were described. Appropriate selector and adaptive selector abstractions were then defined to enable the formal representation of these semantic information requirements in an abstract data model.



An abstract model of the abstract data modeling process with selector and adaptive selector database abstractions

Figure 3.14

Figure 3.7 renders a concise description of the two-dimensional framework for abstract data modeling. Figure 3.14 then adds the selector and adaptive selector database abstractions to this description. While the diagram is itself two-dimensional in nature, it is to be understood that these two abstractions are clearly defined in the third, occurrence dimension.

CHAPTER 4

THE OCCURRENCE DIMENSION IN A GENERIC DATA MODEL

4.1 Introduction

Abstract data modeling is an important tool for designing databases for several reasons. First, it enables the designer to temporarily suppress the inherent limitations of the target database management system and concentrate on the issue of defining the enterprise's information structure. Secondly, the abstract model, or enterprise view, serves as a vehicle of communication between the designer and the end-users. The diagrammatic representation of the enterprise view is easily understood by non-technical and technical personnel alike. Lastly, and perhaps most importantly, abstract data modeling allows the designer to incorporate more of the semantic meaning of the data into the database design.

Most database management systems are based on one of the three major models of data: the hierarchical, the network, or the relational data model. The data structuring capabilities of these models, although more restrictive than the abstract data model, can directly

represent all of the attributes, entities, and functional relationships of the abstract model. Aggregation and generalization hierarchies can also be represented in a straightforward manner while complex relationships typically require special handling. The selector and adaptive selector abstractions, however, have no direct representational form in any of these models.

In the first stage of transforming an abstract data model to a generic data model, the designer is concerned with converting the relatively unrestricted abstract data model into either a collection of tree structures (hierarchical), a network of owner-coupled sets (network), or a collection of normalized relations. The second stage is to express this generic data model design in the data definition language (DDL) facility of the target database management system. The result of this process is a conceptual schema which consists of a complete description of the entire database as it is intended to be structured, stored, and maintained by the target DBMS.

This chapter will be concerned with two specific aspects of representing the occurrence dimension in a generic data model. The first has to do with extending a particular generic data model to support selectors and adaptive selectors. This will involve choosing a generic data model for this purpose and then making the necessary enhancements to its data definition language facility. The second aspect has to do with demonstrating the

manipulative power of selectors and adaptive selectors. This will involve the addition of operators to the data manipulation language component of the chosen generic data model which will allow utilizing the selectors and adaptive selectors in responding to general queries.

The next major section will present arguments for choosing the relational model of data for both of these purposes. The following sections will present a formal syntax for a relational DDL which allows the definition of selectors and adaptive selectors. The last section will be devoted to the introduction of a set of new relational algebra operators which facilitate the manipulation of a stored database containing defined selectors and adaptive selectors. The potential performance gains in processing queries with these new operators will also be demonstrated.

4.2 Choosing a Generic Data Model

The three major data models have been examined extensively and there are arguments which are frequently made to support a preference for one over the others. The question arises as to which of these data models would be the most suitable to extend in order to implement the selector and adaptive selector abstractions.

The hierarchical and network models place heavy emphasis on the explicit structuring of data. Relatively straightforward operators are provided for insertion,

deletion, and updating and for navigation through these structures for retrieval. Once an information structure is represented as a hierarchy or a network of owner-coupled sets, it is effectively frozen in that form.

In the relational model there is only one simple mechanism for structuring data - the normalized relation. Entities are represented by a collection of named base relation schemes and all relationships are represented by the replication of attributes. Emphasis is shifted in this data model to the operators which manipulate relations. New (unnamed) relations can be formed dynamically by joining existing relations, projecting subsets of the attributes of a relation, or selecting subsets of the tuples (instances) of a relation. The relational operators are applied by users from outside of the environment (the database) and provide the ability to manipulate and alter the underlying structure of the data for any particular need. Any new view of data formed by the application of relational operators, however, is not permanent. Only the defined, base relations actually exist.

Selector abstractions represent, in effect, a permanent selection over an entity. Unlike the selection operations that a user may invoke as the need arises, a selector abstraction implies that a certain selected subset of the occurrences of an entity has permanent and universal meaning within the database. Because the relational data model supports dynamic selection through

its operators, it would be quite suitable for the implementation of the selector abstractions.

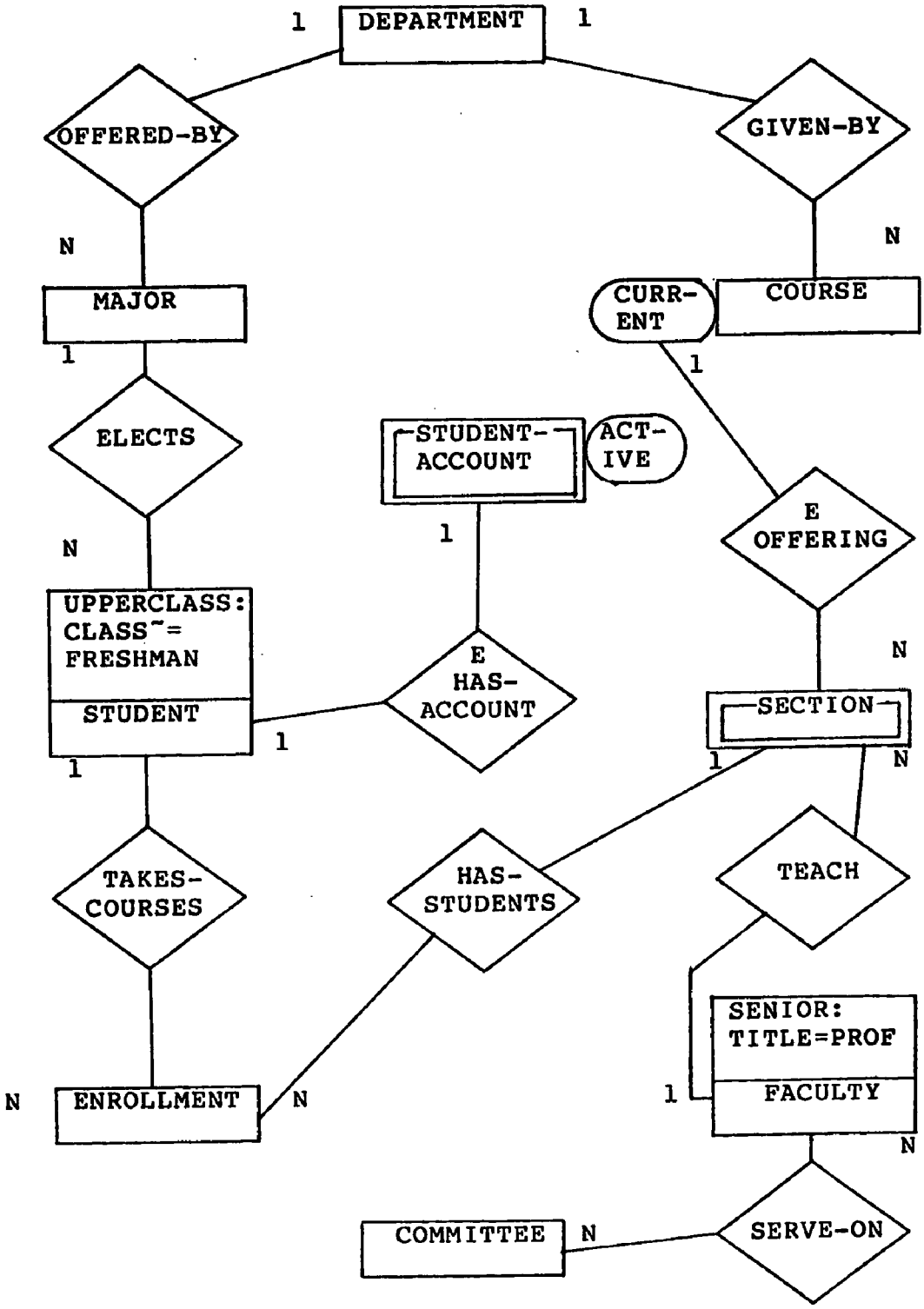
The concept of adaptability is defined as a time-varying quality of an already existing object (entity), consequently, the adaptive selector database abstractions will require a special kind of representation in a generic data model. Facilities are needed to express this quality for particular objects; to give a semantically meaningful name to it; and to provide operators to manipulate it. Again, the relational data model would appear to be the best choice for incorporating the adaptive selector database abstractions as well. Additionally, Smith and Smith [SMIT77b] have already described the aggregation and generalization abstractions in terms of the relational model and Codd [CODD79] has verified that these two abstractions could also be represented directly in the extended relational model, RM/T.

4.3 A Relational Data Definition Language Facility

Unlike the hierarchical or network data models, the relational model of data has no formal diagraming technique to portray the result of transforming an abstract data model into an equivalent collection of normalized relations. The resulting relations are simply listed showing the relation name, the attributes contained in each relation, and an indication of which attribute(s)

form a key for the relation. Because selector and adaptive selector abstractions are non-structural in nature, they do not result in the formation of any relations in this transformation process. Therefore, the list of normalized relations used to represent a database design will have to be augmented with "artificial" relations which will serve as surrogates for any selectors or adaptive selectors appearing in the abstract design. By representing selectors and adaptive selectors in this way, data manipulation operations may be performed on them. This will be examined in more detail later in this chapter.

Figure 4.1 portrays the final abstract data model diagram of the college database as developed in the preceding chapter. The attribute names have been omitted for clarity. However, the simple selector, relationship selector, simple adaptive selector, and relationship adaptive selector of Figures 3.9, 3.11, 3.12, and 3.13, respectively, have been included. While the diagram is two-dimensional in nature, it is understood that these selector and adaptive selector abstractions imply the existence of the occurrence dimension in the model.



The complete abstract data model of the college database

Figure 4.1

Figure 4.2 shows the nine entities of the abstract model of Figure 4.1 portrayed as a collection of ten base relation schemes. The base relation scheme, SERVEON, is a nonentity association arising from the need to handle the complex relationship (actually a cover aggregation) between COMMITTEE and FACULTY. Also, several of the base relation schemes contain redundant attributes which are required to enable the functional relationships to be materialized by joining the various relations over the common domains.

The artificial relation scheme *SENIOR represents the simple selector defined on the entity FACULTY. The only attribute of this artificial relation scheme is the key attribute (ENO) of FACULTY. The relation scheme *UPPERCLASS is the artificial relation which represents the relationship selector abstraction portrayed in Figure 3.11. The attributes of this artificial relation are the key attribute (STUDNO) of the STUDENT base relation scheme and the foreign key MNAME to enable the functional relationship ELECTS to be represented. The asterisks are used to indicate that these are artificial relations representing selector abstractions.

DEPARTMENT (DNAME, OFF, PHONE)

MAJOR (MNAME, DEGREE, CREDITS, DNAME)

STUDENT (STUDNO, SNAME, ADDR, CLASS)

STUDENT-ACCOUNT (STUDNO, BALANCE)

COURSE (CRSENO, CNAME, DESC, DNAME)

SECTION (SECTNO, CRSENO, ROOM, TIME, ENO)

ENROLLMENT (STUDNO, CRSENO, SECTNO, GRADE)

FACULTY (ENO, ENAME, TITLE)

COMMITTEE (COMNAME, NUMMEM)

SERVEON (COMNAME, ENO)

*UPPERCLASS (STUDNO, MNAME)

*SENIOR (ENO)

$ACTIVE (STUDNO)

$CURRENT (CRSENO)

Relation schemes for a hypothetical college database

Figure 4.2

The relation scheme $CURRENT is the artificial relation representing the relationship adaptive selector shown in Figure 3.13. The dollar sign distinguishes this artificial relation as an adaptive selector and its only attribute is the key (CRSENO) of COURSE. Similarly, the artificial relation scheme $ACTIVE defined on the entity STUDENT-ACCOUNT and is shown with its only attribute STUDNO.

Given the collection of relation schemes for a particular design, these must then be coded in the data definition language provided with the target database management system. Ordinarily, the DDL facility is a separate, stand-alone, non-procedural language. Its sole purpose is to allow the formal definition of the database requirements in terms of the structure and characteristics of the base relations, that is, the intention of the stored database. The data manipulation language component of the target database management system, either in query language form or as host language interface, is usually completely separate from the DDL.

For expository purposes, both the DDL and DML components of the target database management system will be embedded in a single, high-level, procedural programming language. Smith and Smith [SMIT77b] chose to describe the DDL syntax of their aggregation and generalization abstractions in a variation of the programming language PASCAL. This choice has two important

advantages. First, PASCAL is a well-defined and widely used programming language. Describing the structure of the two abstractions in a PASCAL-like syntax has an obvious pedagogical advantage. Second, and more importantly, standard PASCAL already supports the abstract data types upon which aggregation and generalization are based. PASCAL's record type and record variant type are precisely the cartesian product and discriminated union abstract data types.

Smith and Smith [SMIT77b] describe only how aggregate and generic objects might be defined as variables in a PASCAL program. This section will develop an extended syntax for a relational data definition language facility for PASCAL. The syntax, and associated semantics, will provide a suitable set of constructs to define a relational database schema including facilities to represent the aggregation, generalization, selector, and adaptive selector abstractions.

4.3.1 Notational Conventions

The programming language PASCAL is considered to be a strongly typed language. That is, in addition to the basic data types found in nearly all general purpose programming languages, PASCAL provides facilities for the creation of user-defined data types. Simple variables and complex structures can be defined on any data type and PASCAL

includes special operators to facilitate manipulating variables defined on these types.

Separating the concepts of data type and data definition in PASCAL is similar to the idea of separating the concept of a domain from an attribute defined on the domain in the relational data model. A data type or domain is simply a well-defined set of values which can be represented on a computer system. The set of values remains uninterpreted until a named attribute or variable is defined on it. In fact, several variables or attributes may be defined on the same domain, however, their names ascribe semantic meaning to their different uses in representing real world objects. In describing a relational data definition language facility for PASCAL, the distinction between data type and data definition will be made explicit not only for domains and attributes but for relations as well.

In PASCAL, data typing can be accomplished in two ways. First, a named data type can be defined explicitly in a type statement. Variables or structures can then be defined on the named type. This is useful when several different variables or structures are to be defined on the same type. Alternatively, the data type can be expressed implicitly when a variable or structure is defined. Explicit data typing will be used here.

Rather than completely defining the syntax and semantics of PASCAL, only those portions of the language

necessary for defining a relational database schema will be described. Although the syntax to follow bears a resemblance to PASCAL's syntax, certain new key words and conventions are used to differentiate the relational database definition facilities from the ordinary PASCAL facilities. Also, the syntax used by Smith and Smith [SMIT77b] to define aggregation and generalization hierarchies has been modified.

For notational purposes, key words (terminal symbols) will be underlined and non-terminal symbols will be enclosed in angle brackets (< and >). Symbols, either terminal or non-terminal, whose presence in a definition is optional will be enclosed in curly brackets ({ and }) and if there is a choice among symbols, they will be separated by vertical bars (|). Small letters will be used for naming data types while capital letters will be used for naming actual attributes and relations as they will be referenced by users.

4.3.2 Unstructured Data Types

In the relational data model, considerable importance is attached to the concept of a domain. A domain consists of a well-defined set of atomic data values. Although these data values are in turn defined as certain patterns of bits on the computer system, for the purposes of defining a relational database schema the domain values are considered to be non-decomposable. Because the set of

values defined for a given domain are atomic they are unstructured.

Named attributes are associated with particular domains to give meaning to the data values as they are used to characterize some real world object. Larger, structured objects are then formed from a collection of named attributes. For the data definition language to be presented here, four unstructured data types are sufficient for defining domains. Three of these unstructured data types are standard PASCAL data types [JENS76] and the fourth was introduced by Smith and Smith [SMIT77b] for use with generalization hierarchies.

The first of the standard PASCAL data types is the base data type. Its syntax is the following:

type <d-name> = {integer|real|char|boolean};

The non-terminal <d-name> stands for the name given to the domain being defined. When a domain is defined as being either of type integer or type real, its range will be the set of numbers representable within the limitations of the host computer system's word size. The type char is used to denote a domain which ranges over all of the symbols in the computer system's character set, e.g., the ASCII character set. Additionally, although not shown in the syntax, a domain typed as char may also have a length attached to it to permit the definition of strings of characters which are to be treated as a whole and not

further decomposed. Lastly, the type boolean ranges over only two values, TRUE and FALSE.

The base data type definition of domains is most frequently used. The second unstructured data type definition is used when a particular domain is intended to range over only a limited set of values. The syntax of an enumeration data type is the following:

type <d-name> = (<v_1>,<v_2>,...,<v_n>);

Again, <d-name> is the name associated with the domain being defined. Following the equal sign is an ordered list of values contained within parentheses. The values are considered to be constants, either numbers or character strings or any combination of the two. When a variable (attribute) is defined on this type, PASCAL will automatically verify that actual values assigned to the variable (attribute) are taken from this finite list. Also, the order in which the list of values is defined is important.

The enumeration data type is useful when the number of legal values is relatively small. The subrange data type can be used to define a domain which may take on a somewhat larger set of values which constitute an ordered, consecutive subset of a base data type. The syntax of a subrange data type is:

type <d-name> = <start>..<end>;

Here, <start> and <end> are constants defined on either the integer, real, or single character char base data types. All values which logically fall within the inclusive subrange are legitimate.

These first three unstructured data types are available as an integral part of any implementation of the programming language PASCAL. The enumeration and subrange data types also offer particularly attractive advantages for incorporating a relational database management system interface into PASCAL.

Figure 4.3 represents a list of unstructured data type definitions for the attributes shown in the relation schemes of Figure 4.2. Base, enumeration, and subrange data types are used to define the necessary domains. One of the domains, "names," will be used later to provide the domain definition for two separate attributes: SNAME and ENAME. The actual definitions are arbitrary and have been chosen only to demonstrate the different possibilities available for data typing in PASCAL.

```
{   base data type domain definitions    }

type deptnam=char(15);
type majnam=char(25);
type deg=char(3);
type names=char(20);
type add=char(40);
type crsenam=char(30);
type des=char(100);
type hour=char(12);
type committ=char(50);

{   subrange data type domain definitions    }

type office=100..650;
type phonenum=1000..9999;
type cred=1..40;
type studnum=10000..99999;
type crsenum=100..999;
type sectnum=1..9;
type classroom=100..509;
type score='A'..'E';
type enum=1000..2500;
type memnum=1..15;

{   enumeration data type domain definitions    }

type rank=(ASSTPROF,ASSOCPROF,PROFESSOR);
type yrgroup=(FRESHMAN,SOPHOMORE,JUNIOR,SENIOR);
```

Unstructured data type definitions

Figure 4.3

The fourth unstructured data type was introduced by Smith and Smith [SMIT77b]. When a generalization hierarchy is created in an abstract model, a characteristic entity may also be defined to represent all of the attributes which are common to each subtype. This characteristic entity has as one of its attributes the name of the entity which represents each subtype. The domain of the attribute will consist of the names of the base relations which define the subtypes.

A special data type is required to define this domain. Smith and Smith refer to such a domain as an image domain. The proposed syntax for defining image domains is the following:

type <id-name> = rel(<s_1>,<s_2>,...,<s_m>);

The non-terminal symbol <id-name> will be replaced by the actual name to be used for the image domain. After the equal sign is the finite list of structured data type names. Each structured data type, to be discussed in the next section, will provide the necessary data typing for the entities (relations) which represent the subtypes of the generalization hierarchy. This list is preceeded by the terminal symbol "rel" to differentiate it from an enumeration data type.

Figure 4.4 shows the one image domain that is required in the generalization hierarchy of Figure 3.5.

```
{    image domain definition    }

type typename=rel(fac,admin,clerk);
```

An image domain definition

Figure 4.4

Structured data types with the names "fac","admin", and "clerk" will then have to be defined. These structured data types will eventually be associated with relations named FACULTY, ADMINISTRATIVE, and CLERICAL respectively.

The four unstructured data types just described are sufficient to define all necessary domains in a relational database schema. These domains serve only to specify the set of legitimate values which an attribute may assume. The smallest meaningful unit in the definition of a database, however, remains the named attribute. Each attribute in the abstract data model must then be given a name according to the conventions of the PASCAL language and be associated with a previously defined data type.

In conventional PASCAL programming, named variables are defined in var statements to make the association between a data type and an object which can be referenced and manipulated within a program. In this extension to the programming language PASCAL, the naming of attributes and their association with domains will take place when structured data types are defined to represent the various

relation schemes which will comprise the database definition.

The next sections will describe the two structured data types which will provide the necessary facilities for defining relation schemes. In these structured data types, attribute definitions will be made explicit.

4.3.3 Structured Data Types

Given a set of relation schemes which adequately portray the intent of the abstract data model, their representation in the data definition language is straightforward. In the PASCAL-based DDL being described here, all of the attributes can be named according to the rules of the particular PASCAL compiler and they may be associated with their domains by defining an appropriate unstructured data type. Relations, being essentially structured objects, can be defined using one of two structured data types which will now be described.

The first of the two structured types will be concerned with defining aggregate relation types. This structured data type will be the most commonly used data type in constructing a relational data model definition. In keeping with the concept of separating a data type from a data definition, each aggregate relation resulting from the transformation of an abstract data model into a set of relation schemes will have a separate aggregate relation type defined for it.

The syntax of an aggregate relation type is as follows:

type <agg-name> =

aggregate [<keylist>]
<a-name$_1$> : {key} <d-name>;
<a-name$_2$> : {key} <d-name>;
.
.
<a-name$_n$> : {key} <d-name>

end;

The non-terminal symbol <agg-name> stands for the name of the aggregate relation type being defined in the DDL. Unlike the unstructured data types which may possibly be associated with different named attributes, each aggregate relation type will be associated with exactly one defined relation scheme. Following the key word aggregate is a required list of attribute names (<a-name>s) which are intended to comprise a user-defined primary key for the relation. In the relational data model, each stored instance of a relation (row or tuple) must be unique and one or more attributes are selected to enforce this requirement.

The list of named attributes for the aggregate relation type being defined then follow. Each <a-name> conforms to the naming rules of the particular PASCAL compiler implementation. The capitalization of each <a-name> indicates that an actual attribute is being defined. This then is the name by which the attribute will be referenced in applications.

After the required colon, the optional keyword key will appear if the defined attribute is in fact the key of another aggregate relation (i.e., it is a foreign key). This requirement is necessary to insure that the Referential Integrity Rule [CODD79] can be enforced. If, in a stored instance of this aggregate relation, a tuple has a non-null value for this attribute, then in the relation for which this attribute is a primary key, a tuple must also appear with the same value. Date [DATE81] discusses this rule and its implications. Lastly, the <d-name> of the domain on which the attribute is defined is provided.

The aggregate relation type described here is patterned very closely after that given in Smith and Smith [SMIT77b]. Among the important differences, however, is that the definition of an aggregate relation type is clearly separated from the definition of a relation scheme as it is to be used in the procedural portion of a PASCAL program. The benefit of this is that aggregate relation types can be constructed and maintained by a database administrator and kept in a central schema library. Different applications can then copy those aggregate relation types to which they have authorized access. The same aggregate relation type may be associated with possibly different relation scheme names in the individual applications.

```
type dept=
     aggregate [DNAME]
     DNAME : deptnam;
     OFF : office;
     PHONE : phonenum
     end;

type maj=
     aggregate [MNAME]
     MNAME : majnam;
     DEGREE : deg;
     CREDITS : cred;
     DNAME : key deptnam
     end;

type stud=
     aggregate [STUDNO]
     STUDNO : studnum;
     SNAME : names;
     ADDR : add;
     CLASS : yrgroup
     end;

type stud-acct=
     aggregate [STUDNO]
     STUDNO : studno;
     BALANCE : real
     end;

type crse=
     aggregate [CRSENO]
     CRSENO : crsenum;
     CNAME : crsenam;
     DESC : des;
     DNAME : key deptnam
     end;

type sect=
     aggregate [SECTNO,CRSENO]
     SECTNO : sectnum;
     CRSENO : key crsenum;
     ROOM : classroom;
     TIME : hour;
     ENO : key enum
     end;
```

Figure 4.5 (Part 1)

```
type enroll=
     aggregate [STUDNO,CRSENO,SECTNO]
     STUDNO : key studnum;
     CRSENO : key crsenum;
     SECTNO : key sectnum;
     GRADE : score
     end;

type fac=
     aggregate [ENO]
     ENO : enum;
     ENAME : names;
     TITLE : rank
     end;

type comm=
     aggregate [COMNAME]
     COMNAME : committ;
     NUMMEM : memnum
     end;

type assign=
     aggregate [COMNAME,ENO]
     COMNAME : key committ;
     ENO : key enum
     end;
```

Aggregate relation type definitions

Figure 4.5 (Part 2)

Figure 4.5 contains the aggregate relation types for the ten base relation schemes of Figure 4.2. In each definition, the key attribute names are identified; all of the attributes are listed with their corresponding domains as typed in Figure 4.3; and where appropriate, the presence of foreign keys is noted.

The ten aggregate relation types specify the structure and intention of each aggregate relation scheme. It remains to associate each aggregate relation type with a named relation scheme by which it will be referenced in

an application. To accomplish this, a variation on the PASCAL var statement will be used. The general form of this definition will be the following:

var <rel-name> collection of <agg-name>;

The non-terminal symbol <rel-name> will be replaced by a desired name for the relation scheme. In general, this will be the name that was given to the relation scheme when the abstract data model was transformed into a set of relation schemes. However, different applications which reference this database may choose different names as the need arises. The fundamental definition of the relation scheme remains with the aggregate relation type regardless of what name is chosen. Again, by separating data typing from data definition, a degree of data independence is provided to the database administrator.

The named relation scheme then stands as a definition for a set of actual stored tuple occurrences all of which have the exact same attribute structure and underlying domains. The key words collection of imply that the named relation scheme is to be associated with the entire set of stored occurrences. The reason for using "collection" rather than "set" is that PASCAL already has a data type for "sets" which is not at all related to database usage. Lastly, the aggregate relation type name is specified to complete the definition of the relation scheme. Figure 4.6

presents the relation scheme definitions for the ten relation schemes of Figure 4.4.

```
var DEPARTMENT collection of dept;
var MAJOR collection of maj;
var STUDENT collection of stud;
var STUDENT-ACCOUNT collection of stud-acct;
var COURSE collection of crse;
var SECTION collection of sect;
var ENROLLMENT collection of enroll;
var FACULTY collection of fac;
var COMMITTEE collection of comm;
var SERVEON collection of assign;
```

Relation scheme definitions

Figure 4.6

In transforming a generalization hierarchy from the abstract model to the relational model of data, several relation schemes result. For each subtype, a separate aggregate relation scheme must be defined assuming that each is not itself the generic object of another generalization hierarchy. An aggregate relation scheme may also be required for the object which represents the attributes of each subtype collectively. Lastly, a relation scheme must be defined for the generic object itself.

Because a generic object is essentially redundant with respect to its attributes and it serves a special role in defining the underlying model, a separate structured type definition is required in the DDL. It is assumed that the DDL processor will effectively control the redundancy in terms of the actual storage and representation of data values, however, the database administrator must be able to express the intent of the generalization hierarchy when defining the database.

The syntax to be used here is a modification of the work of Smith and Smith [SMIT77b]. Generic relation schemes will be separately typed as with aggregate relation types and the specification of the range of aggregate relation types which comprise the generalization hierarchy will be handled by the use of image domain types. The syntax for representing generic relation types is the following:

```
type <gen-name>=

generic {[<agg-name>]}
<c-name_1> : <id-name>;
<c-name_2> : <id-name>;
   .           .
   .           .
<c-name_m> : <id-name>

of aggregate [<keylist>]
<a-name_1> : {key} <d-name>;
<a-name_2> : {key} <d-name>;
   .           .
   .           .
<a-name_n> : {key} <d-name>

end;
```

The non-terminal symbol <gen-name> will be replaced by the actual name of the generic relation type being defined. The keyword generic then appears to differentiate this type definition from the aggregate relation type definition. The square brackets contain the name of the (optional) aggregate relation type which defines the attributes common to each subtype collectively. Next is a list of "cluster names" (<c-name>s) which specify each category of the generalization hierarchy.

Typically, a generalization hierarchy will contain only one category, that is, only one "cluster" of relation schemes will be generalized into a single generic object. However, it is possible that a single generic object may be specialized into several disjoint "clusters." Following each <c-name>, then, is the name of the appropriate image domain. Once the cluster names of the generic relation type have been specified, the key word aggregate denotes the beginning of the attribute definitions for the generic relation type. In the syntax given here, each cluster name is also considered to be an attribute of the generic relation type defined on an image domain. The remaining attribute definitions are the same as in the aggregate relation type syntax.

Figure 4.7 contains the necessary type definitions for the generalization hierarchy of Figure 3.5. Three additional unstructured data types are provided for the domains required in "admin" and "clerk". The fourth

required domain, for the attribute ENAME, has already been defined as "names" in Figure 4.3. Next, three aggregate relation types are defined. The first two are for two of the subclasses in the hierarchy. The aggregate relation type for the subclass FACULTY has already been defined as "fac" in Figure 4.5. The third aggregate relation type is for the characteristic entity EMP-TYPE. Lastly, the generic relation type for EMPLOYEE is defined.

Once the necessary aggregate and generic relation types have been defined, it remains to complete the definition by associating these data types with named relation schemes. Figure 4.8 portrays the resulting relation scheme definitions.

```
{   additional domain types   }

type skillnam = char(15);
type ranknam = char(20);
type sal = 1..9;

{   additional aggregate relation types   }

type admin =
aggregate [ENO]
ENO : enum;
ENAME : names;
RANK : ranknam
end;

type clerk =
aggregate [ENO]
ENO : enum;
ENAME : names;
SKILL : skillnam
end;

type etype =
aggregate [TNAME]
TNAME : typename;
SAL-SCHED : sal
end;

{   the generic relation type   }

type emp =
generic [etype]
TNAME : typename
of aggregate [ENO]
ENO : enum;
ENAME : names
end;
```

Structured data types for a generalization hierarchy

Figure 4.7

```
var ADMINISTRATIVE collection of admin;
var CLERICAL collection of clerk;
var EMP-TYPE collection of etype;
var EMPLOYEE collection of emp;
```

Relation scheme definitions for the generalization hierarchy

Figure 4.8

4.3.4 Data Types for the Selector and Adaptive Selector

The two new database abstractions introduced in the preceding chapter will require additional DDL facilities. This section will propose a syntax for the definition of these two abstractions at the level of the conceptual schema. While both abstractions are non-structural in the abstract data model, they will require a concrete representation in the conceptual schema. The physical details of how these will be implemented will be discussed in the next chapter.

The following syntax represents the way in which a selector data types will be defined in the relational DDL:

```
type <sel-name> =

selector [<a-name>]
of <agg-name> | <gen-name>
 <a-name1> : key <d-name>;
 <a-name2> : key <d-name>;
     .          .       .
     .          .       .
 <a-namei> : key <d-name>

{with
 <a-namej> : key <d-name>;
 <a-namek> : key <d-name>;
     .          .       .
     .          .       .
 <a-namen> : key <d-name> }

 end;
```

The non-terminal symbol <sel-name> will be assigned an actual name for the selector type. The keyword selector differentiates this type definition from aggregate types, generic types, and the unstructured types. The named attribute which determines the selector is included in the brackets. Next, the aggregate type or generic type which contains the named attribute is identified. Note that the constant attribute value which serves to partition the aggregate type or generic type is not specified in the type definition nor is the boolean qualifier. This will be done when a relation extension is defined for the selector type.

Regardless of whether a simple selector or relationship selector is being defined, the key attributes of the aggregate type or generic type are replicated in the selector type definition. In the case of a simple

selector, these will be the only attributes of the selector definition. If a relationship selector is being defined, then the key word with is included followed by a list of the key attributes of the other aggregate or generic type which participates in the relationship. Figure 4.9 portrays the simple selector and relationship selector type definitions for the selectors of Figure 3.9 and Figure 3.11.

```
type profs =
selector [TITLE]
of fac
ENO : key enum
end;

type standing =
selector [CLASS]
of stud
STUDNO : key studnum
with
MNAME : key majnam
end;
```

Selector type definitions
Figure 4.9

The selector data type definitions above alert the DDL processor that selectors will be defined on the attribute TITLE of the aggregate type "fac" and on the attribute CLASS of the aggregate type "stud". The actual values which will determine the partitioning will be provided when the selector extensions are defined. Also, the foreign key "MNAME" is included in the definition of the relationship selector "standing" because the relationship ELECTS between MAJOR and STUDENT is defined

only on the those STUDENTs who are selected on the basis of their class standing. Without the relationship selector abstraction, the foreign key MNAME would have to be included in the aggregate type "stud" with null values eventually stored for all freshmen. The semantic rule that freshmen cannot ELECT a MAJOR would then have to be enforced externally.

Once a selector data type has been declared, it remains to define the selector on an actual relation and to provide the attribute qualification which determines the selected subset. The following syntax may be employed:

var <sel-rel-name>
selects <sel-name> of <rel-name>
where <a-name> <op> <domain-value>;

The non-terminal symbol <sel-rel-name> will be replaced by the name of the selector as it was defined in the abstract data model. After the key word selects is the name of the selector type definition followed by the name of the relation scheme on which the selector is to be defined. Lastly, is the qualification clause which specifies the attribute name, a boolean operator, and a constant drawn from the underlying domain of the attribute. Figure 4.10 shows the definition of the selectors *SENIOR and *UPPERCLASS using this syntax.

```
var *SENIOR
selects profs of FACULTY
where TITLE = "PROFESSOR";

var *UPPERCLASS
selects standing of STUDENT
where CLASS ~= "FRESHMAN";
```

Selector artificial relation scheme definitions

Figure 4.10

The adaptive selector abstraction has been defined as a temporal partitioning of an entity based on certain externally declared criteria. In one case, the membership of the adaptive selector is defined on the basis of recency and frequency of use. As an example, the simple adaptive selector ACTIVE was defined on an entity STUDENT-ACCOUNT. This simple adaptive selector is meant to express the fact that at any point in time a certain subset of STUDENT-ACCOUNTs may be "more meaningful" to the enterprise than the entire set of STUDENT-ACCOUNTs. In cases such as this, neither attributes nor relationships are involved in defining the simple adaptive selector.

In the second case, relationship adaptive selectors are defined in terms of a time-varying participation in a relationship. For example, the relationship adaptive selector CURRENT of Figure 3.13 was defined as selecting those COURSEs which are being OFFERED at any particular instant in time. This temporal criterion for selecting instances of COURSEs is intuitively more understandable but still must be controlled externally.

In either case, the typing of an adaptive selector in the relational DDL is both simple and straightforward. The criteria needed to precisely determine membership will be handled externally. The DDL processor need only be made aware of the existence of an adaptive selector. The syntax for typing either form of adaptive selector is the following:

```
type <ad-sel-name> =

adaptive selector over <agg-name> | <gen-name>
 <a-name1> : key <d-name>;
 <a-name2> : key <d-name>;
     .          .       .
     .          .       .
 <a-namen> : key <d-name>

end;
```

The definition of an adaptive selector requires that a name be given to it and that the aggregate or generic type over which it is defined be indicated. As with the selector type, the key attribute(s) of the named aggregate or generic type are included as the key attributes of the adaptive selector type. The following figure exhibits the type definition for the adaptive selectors ACTIVE and CURRENT of Figure 3.12 and Figure 3.13.

```
type busy =
adaptive selector over stud-acct
STUDNO : key studnum
end;

type offered =
adaptive selector over crse
CRSENO : key crsenum
end;
```

Adaptive selector type definitions

Figure 4.11

Figure 4.11 portrays a simple adaptive selector type definition ("busy") which will serve as the basis for ACTIVE. Because the STUDENT-ACCOUNT entity is related to the STUDENT entity functionally (one-to-one), the key attribute of "stud-acct" is also STUDNO. Finally, the relationship adaptive selector type definition "offered" is given with key attribute CRSENO for "crse."

After an adaptive selector type definition has been defined, it remains to provide a means to associate the type definition with a relation extension. The following syntax will enable the definition of an artificial relation extension to represent adaptive selectors in the DDL:

var <ad-sel-rel-name>

powerset <ad-sel-name> of <rel-name>;

The non-terminal symbol <ad-sel-rel-name> will be the name by which the adaptive selector will be referenced within the data manipulation language portion of an

application. The key word powerset is used for two purposes. First, it denotes immediately the nature of the abstract data type on which adaptive selectors are based. And secondly, it serves to emphasize that, by definition, a powerset of the occurrences of a relation may at any point in time contain all of the occurrences, a subset of the occurrences, or possibly be empty. Figure 4.12 portrays the formal definition of the artificial relations $ACTIVE and $CURRENT using this syntax.

```
var $ACTIVE powerset busy of STUDENT-ACCOUNT;
var $CURRENT powerset offered of COURSE;
```

Adaptive selector artificial relation scheme definitions

Figure 4.12

4.4 A Relational Data Manipulation Language Facility

The preceding section proposed and demonstrated the syntax and semantics of a relational data definition language embedded in the programming language PASCAL. The purpose was to provide the necessary extensions for representing the occurrence dimension in a particular generic data model. As a result, the formal definition of the requirements of the database design problem of Chapter 2 was achieved. Figures 4.3, 4.4, 4.5, 4.9, and 4.11 provide the necessary data type definitions while Figures 4.6, 4.10, and 4.12 associate these data types with PASCAL variable names.

Recognizing the semantic information requirements which give rise to the occurrence dimension and formally defining selectors and adaptive selectors enhances the ability of the database designer to effectively represent the enterprise's information requirements. The ability to manipulate selectors and adaptive selectors in response to database queries, however, provides demonstrable justification for their use.

Originally, Codd [CODD70] proposed an algebraic data manipulation language facility for the relational model. This approach followed from the fact that extensions of relations are simply sets as they are understood in mathematics. Relation schemes define the members of these sets. It was quite natural then to propose a language made up of algebraic operators which manipulate relations viewed as sets. Subsequently, non-procedural, calculus-like data manipulation languages evolved. However, the relational algebra remains the standard by which all relational data manipulation languages are judged. If a relational DML can be shown to possess a set of operators at least as powerful as the relational algebra, it is said to be relationally complete.

The relational algebra, in its minimal form, consists of only five primitive set operators. Of these, two operators are unary, that is, they manipulate only one entire relation at a time. These two are the selection operator and the projection operator. The remaining three

operators are binary - operating on pairs of relation extensions. These are union, difference, and extended cartesian product. A comprehensive treatment of these five operators can be found in Codd [CODD79], Date [DATE81], or Ullman [ULLM80].

In addition to these five primitive operators, the relational algebra has certain more powerful operators which are based on them. The most important of these operators is the join operator. In joining two relation extensions, there must be at least one attribute in each defined on a common domain. The attribute names may be different as long as the underlying domains are identical. The result of the join operation consists of the concatenation of the columns of each relation as in the product operator, however, a result tuple is formed only if the values of the common domains satisfy some boolean qualification. Codd [CODD79] contains a complete discussion of the various forms of join operator.

In this extension of the relational model, the notion of "artificial relations" has been introduced. These artificial relations stand for the selectors and adaptive selectors defined in the relational data definition language facility. The first type of artificial relation represents the simple selector where the only attribute(s) contained in it are the key attribute(s) of the base relation over which it is defined. This type of artificial relation does not participate directly in any

relationships and will be denoted SS. The second type represents relationship selectors where, in addition to the key attribute(s) of the base relation, foreign keys are present as well. This type of artificial relation will be denoted RS.

Artificial relations which represent either form of adaptive selector contain only the key attribute(s) of the base relation on which they have been defined. These types of artificial relations will be denoted SAS and RAS for simple adaptive selector and relationship adaptive selector, respectively. Lastly, conventional base relations will be denoted simply as R.

All five basic operators and the various forms of join are clearly applicable to base (R-type) relations. The two unary algebraic operators, selection and projection, operate on SS-, RS- and RAS-type relations in the conventional manner with projection being appropriate only when there is more than one named attribute in the relation. Neither operator, however, is applicable to relations of the SAS-type. The membership of the SAS-type relation is determined by the external factors of recency and frequency of use. The size and membership of such a relation is also changing dynamically.

In considering the binary algebraic operators, each possible combination of the five different relation types must be examined. This would lead to 15 pairings. However, the dynamic nature of the SAS-type precludes its use in

any of the binary operators, therefore, only ten possible combinations remain. In these ten combinations, the binary algebraic operators are all applicable subject to the ordinary restrictions placed on them, such as union compatibility and the presence of common domains where necessary.

With the exception of the simple adaptive selector then, the relation schemes representing the other database abstractions may be manipulated by the relational algebra in a straightforward manner. However, it will generally be the case that these artificial relation schemes are not manipulated directly, but that they will be manipulated by utilizing the new, extended operators and facilities to be described in the next section.

4.4.1 Extended Operators for the Relational Algebra

The five primitive relational algebra operators along with the various forms of the join operator, provide a rich collection of data manipulation language facilities. Even though these facilities may be used to manipulate the artificial relations employed to support the selector and adaptive selector abstractions, additional operators are needed to fully exploit their presence. Five such additional operators have been identified and will be described in this section.

Both selector and adaptive selector abstractions are intended to define subsets of the occurrences of an

entity. Their representation by artificial relations in the conceptual schema signals the intent that the appropriate subset will be required for manipulation apart from the base relation as a whole. An extension of one of these artificial relations does not physically contain the selected tuples but rather provides a mechanism to identify the subset when necessary.

The first of the five extended operators is intended to enable the selected subset to be the designated operand in any DML operation. Because the artificial relation extensions themselves do not contain the desired subset but only contain the key attributes of the selected tuples, these artificial relation extensions may be used as a filter or a "mask" when manipulating the base relation extension. When viewing a base relation extension through the filter only the appropriately selected tuples may be "seen."

The second extended operator is the logical inverse of the first. While the selected subset has been singled out because of its special semantic meaning, it will often be the case that the complement of the selected subset will be the desired operand in a DML operation. In such cases, the tuples not qualified for membership in the artificial relation extension are to be manipulated.

In either form of selector, a particular tuple could be tested for membership in the artificial relation by simply applying the boolean qualification to the base

relation itself. Likewise, in the relationship adaptive selector, participation in the defining relationship could be tested quite readily. However, in a simple adaptive selector, because of its temporal criterion and dynamically changing membership, an operator is needed to ascertain whether a particular tuple is presently selected. The third extended operator will then be a set membership test operator.

The fourth extended operator will be a set size operator. That is, this operator will yield as its result the cardinality of the artificial relation to which it is applied. Although there is usually a DML operation which counts the tuples in a relation extension, this operator will be particularly useful with the adaptive selectors.

The fifth, and last, extended operator is actually not an operator in the sense that it may be applied at will by a user of the DML. Rather it is intended to be employed directly by the DML processor as a natural part of its determination of the optimum manner in which to satisfy a DML query. In deciding how to respond to a query involving a join of two base relations in the presence of a relationship selector or a relationship adaptive selector, an implied join will be incorporated into the query syntax to utilize the artificial relation where appropriate.

	ARTIFICIAL RELATION TYPES	SS	RS	SAS	RAS
OPERATORS	FILTER	YES	YES	NO	YES
	COMPLEMENT	YES	YES	NO	YES
	SET MEMBERSHIP	NO	NO	YES	YES
	SET SIZE	YES	YES	YES	YES
	IMPLIED JOIN	NO	YES	NO	YES

Applicability of the extended relational operators to the artificial relation types
Table 4.1

Table 4.1 indicates the applicability of these five extended operators to the four artificial relation types. Where the operator is applicable a "YES" is entered otherwise a "NO" is entered.

Again, because of the highly dynamic, time-varying nature of the simple adaptive selector relation, the filter, complement, and implied join extended operators would be inappropriate for use on it. Set membership and set size, however, are useful operators with this type of artificial relation. Set membership is not appropriate for use with either form of selector because the test of membership could be applied directly to the base relation by examining the designated attribute value.

4.4.2 Data Manipulation with the Extended Operators

These five extended operators will now be demonstrated using sample DML queries posed in the context of the college database example. The particular syntax to be employed here to demonstrate the manipulative power of selectors and adaptive selectors will follow closely the SEQUEL data manipulation language described in Date [DATE81]. Although SEQUEL is classified loosely as a transitional DML falling somewhere between the algebra and calculus, it is much closer to the algebra in its syntax. Also, SEQUEL is used in several widely used relational database management systems including SYSTEM/R, SQL/DS, ORACLE, and RIM. It will be assumed that the SEQUEL syntax is embedded within the PASCAL programming language in order to be consistent with the DDL presented above.

Table 2.1 indicates the expected number of entity occurrences to be stored in this database. In order to evaluate the effectiveness of using selectors and adaptive selectors, certain additional assumptions will be necessary. First, among the 250 FACULTY member occurrences, 100 will be of professorial rank. Of the 4000 STUDENT occurrences, it is assumed that they are approximately evenly distributed across classes, that is, there are 1000 STUDENTs in each class. Although there are 500 COURSEs listed by the college, only 100 will be offered in any given term. Each such COURSE offering will

have an average of two SECTIONs. Finally, of the 4000 STUDENT-ACCOUNTs, only 800, or 20 percent will be actively used.

Although consideration of the internal data model will not be taken up until the next chapter, assumptions regarding storage structure support will also be needed here to compare the effectiveness of the query formulations. It is assumed that each stored relation extension has associated with it a dense hierarchical index for primary access to its tuples based upon the primary key. Similarly, simple selectors, relationship selectors, and relationship adaptive selectors are implemented with dense hierarchical indexes as well. These assumptions permit some of the query evaluations to be carried out by operating on the storage structures instead of the actual tuples.

The results to be offered in the following comparisons are only indicative of the magnitude of the potential efficiencies obtainable by using the extended operators with selectors and adaptive selectors. The assessment of the true magnitude of the relative gains would necessarily involve considerably more information about the query processor, any optimization strategies used, and the characteristics of the physical environment.

To utilize both the filter and complement operators, a new clause - "USING" - will be introduced into the

SEQUEL syntax. The syntax of a USING clause is the following.

USING {COMPLEMENT OF}
<sel-rel-name> | <ad-sel-rel-name> :
SELECT . . .

Without the optional key words "COMPLEMENT OF", the USING clause indicates that either form of selector or a relationship adaptive selector is to be used as a filter to qualify the subsequent SELECT statement. For example, consider a retrieval request for the course numbers (CRSENO) and course names (CNAME) of all COURSEs which are presently being offered. There exists a relationship adaptive selector which identifies these course instances. This query could be posed with the following modified SEQUEL statements.

```
USING $CURRENT:
 SELECT CRSENO CNAME
 FROM COURSE
```

The use of the relationship adaptive selector $CURRENT in this particular query guarantees that only presently offered COURSEs will be retrieved. In fact, the dense hierarchical index created to implement $CURRENT would be used to access the tuples of the COURSE relation extension instead of the primary index. A total of 100 COURSE tuples would then be retrieved to extract the requested information.

Without the relationship adaptive selector and the filter operator, this query would be considerably more difficult to formulate. In this case, using standard SEQUEL syntax, the query would be formulated as follows.

```
SELECT CRSENO CNAME
FROM COURSE
WHERE EXISTS
        (SELECT *
         FROM SECTION
         WHERE SECTION.CRSENO=COURSE.CRSENO)
```

The first observation is that this formulation is considerably more complex. Not only is it lengthier in its written form, and consequently more difficult to understand, but it requires more effort to execute. The simplest strategy for executing this particular query formulation would be to perform a join of the two primary indexes as indicated in the last WHERE clause. The result of the join would be the 100 (matched) keys of the offered COURSEs. The actual retrieval would be for these selected occurrences only. However, the join operation on the primary indexes represents the additional effort in this formulation without the filter operator.

Another possible query would be to retrieve, and presumably list, the names (SNAME) and addresses (ADDR) of all freshmen. It is known that there is a relationship selector which identifies all non-freshmen. The complement of this relationship selector would yield the desired result when applied to STUDENT. This query could be posed as follows.

```
USING COMPLEMENT OF *UPPERCLASS :
  SELECT SNAME ADDR
  FROM STUDENT
```

The execution of this query could be carried out by initially operating on the two hierarchical indexes associated with the STUDENT relation extension, that is, the primary index for all STUDENT tuples and the index for *UPPERCLASS. The set difference of these two indexes would yield the keys and pointers to freshman STUDENT occurrences. The actual retrieval of tuples would limited to a total of 1000 freshman STUDENT occurrences.

The alternative formulation would require that the qualification on the attribute CLASS be made explicit.

```
SELECT SNAME ADDR
FROM STUDENT
WHERE CLASS = "FRESHMAN"
```

In terms of the length and clarity of the two formulations, either one is acceptable. In fact, the latter, with its explicit qualification on the CLASS attribute, could be argued to be a clearer expression of the intent of the query. However, in this form, all 4000 STUDENT tuple occurrences would have to be retrieved in order verify freshman status. Consequently, the availability of the relationship selector *UPPERCLASS and the complement operator result in substantially less work in responding to this query.

The basic relational algebra operators and the extended operators of filter, complement, and implied join

all return as a result an unnamed relation containing those tuples which met the stated qualification. That is, these operators are set valued. The two extended operators for set membership and set size are not set valued.

The set membership operator will be implemented as a simple boolean test. After initializing the key attribute(s) of a base relation, a test may be made to determine if the specific tuple is currently a member of an adaptive selector defined on that base relation. The result of this operator will be either true or false. The filter and complement operators are, in effect, tests of set membership, however, they range over an entire base relation. Set membership deals only with specific tuples.

The set membership operator will be used in an "IF ... THEN ... " construct preceding a SELECT. The key word "IN" in the boolean qualification signals that a set membership test is being applied to an adaptive selector. The syntax is as follows.

IF <keylist> IN <ad-sel-rel>
THEN SELECT . . .

The non-terminal symbol <keylist> will contain the names of the key attributes of the base relation on which the adaptive selector is defined. Prior to the IF statement, these attribute names must be initialized to the appropriate values for the tuple in question. For example, a query which retrieves the BALANCE of a

particular STUDENT-ACCOUNT only if that account is among the active accounts would be formulated as follows.

```
IF STUDNO IN $ACTIVE
THEN SELECT BALANCE
     FROM STUDENT-ACCOUNT
```

Because of the nature of the simple adaptive selector $ACTIVE, there is no equivalent formulation of this query without the set membership operator. The membership of $ACTIVE is determined by the observation of recency and frequency of reference to STUDENT-ACCOUNTs and does not depend on attribute values or relationship participation. A failure of this query to return a BALANCE would be indicative that the account in question is not among the active set.

Another example of the set membership operator would be a query which lists the section numbers, rooms, and times of a COURSE which is currently being offered.

```
IF CRSENO IN $CURRENT
THEN SELECT SECTNO ROOM TIME
     FROM SECTION
```

This query requires that the key attribute CRSENO be initialized to a valid course number. If the COURSE in question is a member of the relationship adaptive selector $CURRENT, then the appropriate information is retrieved; otherwise, the SELECT will not be performed. Because the key of the SECTION relation scheme consists of the concatenation of CRSENO and SECTNO, the primary index for the SECTION relation extension cannot be used without

knowledge of the SECTNO(s). Therefore, the actual retrieval operation would involve a scan of the 200 SECTION occurrences.

The equivalent formulation of this query without the set membership operator and relationship adaptive selector is, in fact, simpler.

```
SELECT SECTNO ROOM TIME
FROM SECTION
WHERE SECTION.CRSENO = x
```

In this form, it is assumed that the relevant CRSENO is represented by "x". The difference in the two formulations is that in the former, if the COURSE occurrence is not being offered, only the relationship adaptive selector index is searched and no additional work is performed when the search fails. In the latter, all of the SECTION occurrences would be retrieved, at substantial cost, only to determine that there are no matches with the specified CRSENO.

Frequently, it is useful to know the size of a relation in terms of the number of tuples it currently contains. The syntax of SEQUEL's SELECT statement has a built-in operator, COUNT, which may be used for this purpose. The COUNT function, however, requires that the entire relation extension be retrieved to obtain the number of tuple occurrences. For the artificial relations described here, an explicit count of the number of tuples will be maintained. The value of this field may be

obtained for any artificial relation using the following form of SELECT.

SELECT SIZE (<sel-rel> | <ad-sel-rel>)

This brief form of SELECT simply returns an integer count of the number of tuples presently selected for membership in either the indicated selector relation or adaptive selector relation. If it were necessary to determine the total number of currently offered COURSEs, the next query would suffice.

```
SELECT SIZE ($CURRENT)
```

The last of the extended operators is the implied join. Unlike the preceding four operators, the implied join is not directly invoked when formulating a query. The syntax of the SELECT statement in SEQUEL embodies the relational algebra operations of selection, projection, union, and various forms of join. In the examples shown so far, only selection, projection, and join have been demonstrated.

When the FROM clause in a SELECT statement contains the name of more than one relation, these relations are joined. If the relations do not contain any common attribute names, an extended cartesian product is performed. If there are common attribute names, then either a theta join or natural join is performed depending on the WHERE clause.

In the presence of relationship selectors and relationship adaptive selectors, the joining of two base relations only makes sense for the tuples which are members of the appropriate artificial relations. In formulating queries involving base relations with either a relationship selector or relationship adaptive selector, the query parser should perform the implied join of one of the two base relations before affecting the final join with the second base relation.

Because joins (and implied joins) are not shown explicitly in the SEQUEL syntax, there is no change or addition to the syntax.

To demonstrate when and how an implied join would be employed, the following query examples will be used. The first query involves retrieving the student names and major names of all students in the engineering department. The attribute SNAME is contained in the relation STUDENT while the attributes MNAME and DNAME are contained in MAJOR. There is no need to retrieve the relation DEPARTMENT.

Occurrences of STUDENT may only be related to occurrences of MAJOR if they have qualified for membership in the relationship selector *UPPERCLASS. The proper sequence of relational algebra operations to handle this query would involve first performing an equijoin over STUDNO of the base relation STUDENT with the artificial relation *UPPERCLASS. This would result in an

intermediate, unnamed relation containing the tuples of all STUDENTs who may legitimately elect a MAJOR. Assuming none of the attributes are eliminated by a projection, this unnamed relation consists of all of the STUDENT attributes concatenated with the MNAME attribute from *UPPERCLASS. Next, this unnamed relation is to be joined (equijoined) with the base relation MAJOR on the basis of equal MNAME values. Lastly, a selection is made on the DNAME attribute and the desired tuples are projected over the SNAME and MNAME attributes. The SEQUEL SELECT statement which performs all of these operations is written thusly.

```
SELECT SNAME MNAME
FROM STUDENT MAJOR
WHERE DNAME = "ENGINEERING"
```

In conventional SEQUEL syntax, the only apparent join here would be on the base relations STUDENT and MAJOR. However, referring to Figure 4.2, there is no common attribute on which to accomplish the join. The result would then be a virtually meaningless extended cartesian product followed by a selection on DNAME and a projection over SNAME and MNAME. With the extended operator of implied join, however, the proper sequence of operations described above would be carried out to correctly respond to the query.

Without the relationship selector *UPPERCLASS, the relational schema of Figure 4.2 would have to propagate

the foreign key MNAME into the relation scheme STUDENT. The domain on which MNAME is defined would have to contain a special null value for all freshman. The question of how to treat null values in join or selection operations is currently a research problem (see [CODD79] and [DATE83]).

The definition of a relationship selector calls for including the key attribute(s) of one of the base relations as a foreign key in the artificial relation scheme. Relationship adaptive selectors contain only the key attribute(s) of the one base relation on which they are defined. This, however, does not preclude the applicability of the implied join to facilitate the response to a query. When a join operation is required between two base relations that have a relationship expressed through a relationship adaptive selector, the implied join using the relationship adaptive selector can avoid unnecessary work in responding to the query.

As an example, consider a query calling for the retrieval of the course number, section number, course name, and faculty member's name from the database. The response to this query would involve joining the COURSE relation with the SECTION relation to form an unnamed relation with the attributes of both. This would involve retrieving 500 COURSE occurrences and 200 SECTION occurrences to form a 200 occurrence unnamed relation extension.

Next, a join of this unnamed relation with FACULTY over attribute ENO would yield the ENAME attribute. This join operation would involve retrieving the 250 FACULTY occurrences at least once. Finally, a projection over the attributes CRSENO, SECTNO, CNAME, and ENAME would produce the desired result.

Although this sequence of operations could be carried out with the existing relation schemes, the presence of the relationship adaptive selector $CURRENT would simplify the entire process. In this case the implied join of COURSE with $CURRENT would initially restrict the first join to only those COURSE occurrences presently being offered. This reduces the number of COURSE occurrences retrieved to only 100 instead of 500. The query formulation would be as follows.

```
SELECT CRSENO SECTNO CNAME ENAME
FROM COURSE SECTION FACULTY
```

These demonstrations are not all inclusive of the manipulative power afforded by the presence of selectors and adaptive selectors, however, they do provide an insight as to how a relational data manipulation language such as SEQUEL might exploit them in response to certain types of queries. The incorporation of selectors and adaptive selectors in an abstract data model enhances the ability of a database designer to represent more of the semantics of data. Likewise, the use of the five extended

operators presented here enables a database user to express those same semantics when formulating a query.

4.5 Summary

The goal of this chapter has been to integrate the first two levels of the three level database design methodology. This requires mapping the concepts and principles of abstract data modeling onto the structures, operators, and constraints of a generic data model. In particular, the selector and adaptive selector abstractions which introduced the occurrence dimension into abstract data modeling, require special structures and operators not available in existing generic data models.

To accomplish this goal, the relational model of data was chosen as the target generic data model. The programming language PASCAL was chosen as the vehicle for developing the necessary data definition language facilities to permit the definition of selectors and adaptive selectors. The data manipulation language SEQUEL was employed to demonstrate the manipulative features of selectors and adaptive selectors.

The first contribution was the introduction of the concept of artificial relations to represent selectors and adaptive selectors when transforming an abstract model into a collection of normalized relation schemes, i.e., the conceptual schema. The next stage in conceptual schema

development involved the provision of data definition language facilities to allow the formal definition of the conceptual schema in a DBMS-processible form. An extended syntax was developed to completely define a conceptual schema including the necessary facilities to both type and define selectors and adaptive selectors. These two data definition stages then enable the formal representation of the occurrence dimension at the level of a generic data model.

The provision of DDL structures to represent the occurrence dimension in a generic data model constitutes only part of the mapping of an abstract data model onto a generic data model. Operators must also be provided to manipulate these new structures. To demonstrate the manipulative power afforded by the presence of artificial relations representing selectors and adaptive selectors, the relational algebra was extended to include five new operators.

A series of representative queries, drawn from the database design example, was used to show how these five new operators would be employed. Where appropriate, the alternative query formulation without the new operator or artificial relation was also presented. In most cases, it was shown that the formulation involving the new operators was more clear than the alternative formulation and in all cases the amount of work required to respond to the query was significantly less.

CHAPTER 5

THE OCCURRENCE DIMENSION IN INTERNAL DATA MODELING

5.1 Introduction

To complete the integration of the three level database design methodology shown in Figure 2.1, the generic data model representation of a database must finally be mapped onto the physical structures available with the particular DBMS and host computer system. The process of choosing a suitable implementation strategy is referred to as internal data modeling.

At the level of abstract data modeling, the existence of occurrences of entities and relationships has been conceptualized to enable the representation of certain semantic information requirements which would otherwise not be representable. The selector and adaptive selector abstractions have been introduced as mechanisms for the characterization of occurrences of data objects in an abstract data model. At the level of a generic data model, definitional forms (artificial relations) have been introduced to formally specify the intention of implementing and maintaining these special characterizations of occurrences. Additionally, a set of

data manipulation operators have been defined to facilitate the formulation of database queries in the presence of selectors and adaptive selectors. Finally, at the level of the internal data model, consideration must be given as to how any defined selectors and adaptive selectors will be physically implemented, maintained, and manipulated.

This chapter will be concerned with several issues related to the internal data modeling requirements of the occurrence dimension. The practice of internal data modeling is not a subject about which many generalizations may be made. Rather, it is quite intimately tied to the environmental factors surrounding a particular DBMS and its host computer system. Therefore, in addressing these issues, the approach and analysis will be more suggestive of solutions to the problems instead of prescriptive.

The next major section will explore the physical storage structure alternatives which would be suitable for the implementation, maintenance, and manipulation of selectors and adaptive selectors. For the most part, the suggested storage structures are available in many commercial database management systems and the host computer operating systems which support them. Specific reasons for each suggested storage structure will be offered in order to clarify the intent of physically representing a particular selector or adaptive selector.

The following section will then address some of the operational considerations which arise as a result of having selectors and adaptive selectors in a database design. The topics to be discussed are not themselves design issues and therefore are not technically part of the process of internal data modeling. However, they are important if the full potential of selectors and adaptive selectors is to be realized.

Among the claims for incorporating the occurrence dimension into a database design methodology were that it would result in potential operational performance improvements and that it would prolong the operational lifetime of the database. Several suggestions for the improvement of operational performance will be discussed and an entirely new database buffer management policy will be described and analyzed.

5.2 Storage Structures

In theory, the storage structure selection problem is quite complex. Prior to the development of generalized database management systems, files of records stored on secondary storage media were organized as flat files. Sequential, direct, or indexed access were the dominant storage and retrieval mechanisms. With the advent of database management systems, storage structures previously restricted to processor memory became viable structures for secondary memory organization as well. Linked lists,

inverted files, hash tables, and numerous pointer schemes greatly increased the storage structure alternatives available to a database designer.

In practice, a database designer is constrained to utilize only those storage structures supported by the particular database management system. In the case of hierarchical and network oriented database management systems, the choice is still formidable. However, there do exist a number of semi-automated design aids which, given the conceptual schema and estimates of usage patterns, can assist the designer in selecting a reasonable physical design for these types of systems (see Teorey and Fry [TEOR80] for an extensive bibliography of work in this area).

The relational data model assumes a very simple storage structure - the table. Each relation scheme contained in the schema is to be stored physically as a two-dimensional table with the columns representing the named attributes and the rows representing the individual occurrences of the relation extension. Any relationships are represented by (redundantly) stored data values. The power of the data manipulation language is then used to materialize these relationships when necessary (e.g., the join operation). Additionally, the relational data model imposes no ordering on the rows of the table.

The tabular storage structure just described is a conceptual ideal within the relational data model. In all

practicality, additional storage structure support is employed to facilitate efficient retrieval operations. It is clearly desirable to maintain some form of logical ordering on the tuples (rows) of a table if there are uses which need to access the tuples sequentially. Similarly, it is not unlikely that some retrieval operations require rapid, random access to individual tuples. In this case, some type of direct storage structure may be used.

To balance these two extremes, a form of hierarchic index storage structure is frequently used. Under the generic name of indexed sequential access methods (ISAM), this type of storage structure may be used for primary, secondary, and relationship access to tables [HAER78]. Primary access refers to locating specific tuples based on their key attribute values. Secondary access may optionally be provided to enable the location of tuples (or a set of tuples) based on non-key attribute values. Lastly, relationship access refers to explicitly maintaining a storage structure to represent relationship occurrences. In the relational database management system - SYSTEM R [ASTR76] - this type of storage structure is known as a "link" and effectively represents a permanent join on two relations (tables). All of these storage structures are, at least in theory, hidden from the user.

Without loss of generality, the following assumptions about physical design environment will be made. The relational database management system being used is

assumed to support the data definition language facility and data manipulation language extensions given in the preceding chapter. Each base relation extension will be stored as a simple tabular structure. Physically, the tabular structure will be organized as a set of fixed size blocks of secondary storage and each block will contain a number of tuple occurrences. The order of the tuples within blocks is immaterial, that is, it cannot be assumed that the tuples have been stored and maintained in any particular logical order.

Primary access to the stored base relations will be accomplished by means of a separate, dense hierarchic block index storage structure. This particular storage structure is commonly known as a B* tree [KNUT73]. No secondary storage structures will be considered. This assumption has no consequence on the design because secondary storage structures are always redundant and are incorporated strictly for performance reasons. Relationship access between or among base relations is accomplished by the data manipulation language operators (e.g., join or cartesian product).

These assumptions are quite simplistic on the surface but are really not that far from reality for commercially available database management systems (SQL/DS and INGRES, in particular). As an example, the ten base relations of the schema portrayed in Figure 4.2 will be assumed. Primary access to each base relation is provided by a

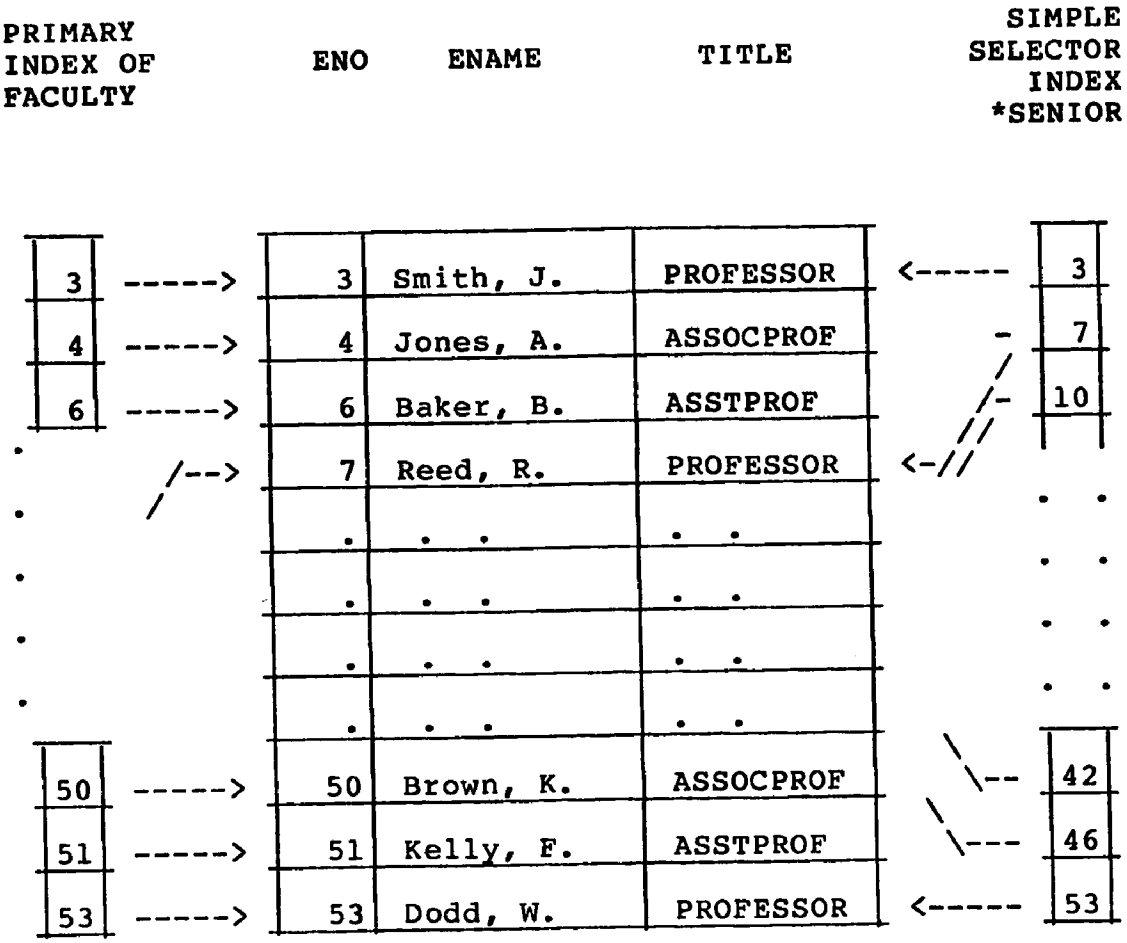
dense, hierarchic block index organized as a B* tree. Each base relation will then be physically stored as two separate "files." One file will hold the actual tuple occurrences which may or may not be in their key sequence order. The second file will contain the entire dense index to the base relation. Primary access to the stored base relation will be accomplished through these two files.

The remainder of this section will explore the possibilities for storage structure support for the selector artificial relations and the adaptive selector artificial relations.

5.2.1 A Storage Structure for Simple Selectors

A natural storage structure for the physical representation of a simple selector would be to build and maintain another hierarchic block index to the base relation. This index would contain the keys and pointers to the tuple occurrences which have been selected by the boolean qualification. When accessing the stored base relation via a filter operator (i.e., a USING clause is contained in the query), the simple selector index would be used in place of the primary index. Figure 5.1 shows a portion of such a storage structure.

FACULTY RELATION EXTENSION



A hierarchic index for a simple selector

Figure 5.1

In the center of the figure is a tabular structure of a part of the FACULTY relation extension. The fact that the tuples are ordered on increasing ENO is immaterial and has been done only for clarity. On the left side, the lowest level blocks of the primary index are shown pointing to their associated tuple occurrences. Higher level blocks of the primary index are not shown. The right side of the figure indicates the lowest level blocks of the index used for the simple selector *SENIOR defined on FACULTY. Higher level blocks of this index are also not shown. Because *SENIOR selects that subset of FACULTY who are full professors, only those tuples with the value "PROFESSOR" in the TITLE attribute are contained in this second index storage structure.

Although the concept of an artificial relation was introduced to represent the simple selector in a relational schema definition, at the physical level of implementation, there is no need to create a separate tabular structure. Given the availability of a hierarchic index storage structure, the simple selector may be represented efficiently and concisely by a second index structure associated with the base table.

From the point of view of data manipulation, the filter operator may be applied directly to this second index. For example, a SELECT operation with the USING clause would retrieve only the assumed 100 PROFESSORs from the total of 250 FACULTY occurrences. The complement

operator would require that both the primary and simple selector indexes be scanned at the same time. That is, when manipulating the subset of the tuples of the base relation which are not selected, the primary index structure would provide the necessary access path after verifying that each tuple considered is not found in the simple selector index.

The amount of work involved in this double indexing is not as much as might be assumed. The secondary storage blocks which are used for holding the index typically contain many more key/pointer pairs than a comparable block which contains actual tuple occurrences. Determining the complement of a relation extension with respect to a simple selector can be carried out more efficiently when operating on the indexes than could be accomplished by transferring all of the data (tuple) blocks and choosing the desired tuple occurrences. While dense hierarchic block indexes require more storage space than their non-dense counterparts, it is just this aspect of efficient manipulation prior to the actual retrieval of data blocks which make them so attractive as a database storage structure [ULLM80].

Storage structure support for a simple selector is then quite straightforward and efficient. The artificial relation defined to represent a simple selector does not itself become a stored relation (table), but merely becomes an alternative storage structure to the underlying

base relation table. The insertion and deletion of tuples in the base relation table would require that the index structure for the simple selector be modified accordingly. Also, the database management system would be required to monitor the updating of the selected attribute values should such an update cause a particular tuple to either join or leave the selected subset.

5.2.2 Storage Structures for Relationship Selectors

In abstract data modeling, relationships between entities were classified as being either implicit or explicit. Implicit relationships are represented in abstract data modeling as directed edges connecting entities. In the relational data model, implicit relationships are represented by the careful replication of key attributes as foreign keys in the appropriate base relation schemes. Fundamental to either the explicit or implicit relationships is the fact that once they are defined, any entity occurrence may participate in the relationship. There is no qualification associated with the relationship which systematically includes or excludes certain entity occurrences from legitimately participating in the relationship.

The relationship selector database abstraction was introduced to allow such a qualification to be expressed for an implicit relationship. The applicability or legitimacy of an implicit relationship between two

entities may depend on the value of a particular attribute contained in one of the entities. The relationship selector captures this qualification in the abstract data model and is represented in the by an artificial relation scheme as defined in the preceding chapter. Just as with the simple selector, the artificial relation scheme will be treated as an alternative storage structure for an existing base relation tabular structure rather than a stored base relation in its own right.

The artificial relation definition for a relationship selector contains two sets of attributes. The first set is the attribute(s) which comprise the primary key of the base relation scheme on which the relationship selector is defined. The second set is the primary key attribute(s) of the other relation scheme. These two sets combined may be considered a concatenated key for the artificial relation. There are no other attributes contained in the artificial relation for a relationship selector such as might be found in an associative entity or nonentity association.

A relationship selector artificial relation can be implemented as an alternative dense hierarchic index storage structure in a manner similar to the simple selector. The key attribute values of the base relation on which the relationship selector is defined constitute the "primary" portion of the concatenated key. The key attributes of the other base relation are of "secondary" importance. The reason for this dichotomy between the sets

of attributes stems from the way in which the relationship selector is to be operated upon.

The principal operations involving a relationship selector are the filter, complement, and implied join. The filter and complement operators deal specifically with the subset selected by the boolean qualification used in defining the relationship selector. Therefore, it is the "primary" portion of the concatenated key which is of principal importance in these operators. For the implied join, both portions of the concatenated key are required to effect the joining of the two base relations.

A storage structure suitable to meet these requirements would involve a modification to a dense hierarchic block index. The lowest level blocks of the index would maintain an ordered list of the concatenated keys. The "primary" portion of the concatenated key, by appearing first, would determine the lexicographic order of the index. These block entries would then be comprised of the "primary" portion of the concatenated key, the "secondary" portion of the concatenated key, and a pointer to the tuple corresponding to the "primary" key value. The higher level blocks would be based solely on the "primary" portion of the concatenated key.

Figure 5.2 shows a sample tabular representation of portions of the STUDENT and MAJOR base relations. Each base relation has a dense hierarchic block index associated with it for primary access to its tuples.

Again, the ordering of the tuples is merely for clarity. Figure 5.3 then shows a first (lowest) level block of the dense hierarchic block index used to represent the relationship selector *UPPERCLASS and a portion of the block immediately above it in the hierarchy.

In a conventional implicit relationship between entities such as these, the key attribute of the MAJOR relation would necessarily be propagated to the STUDENT relation as a foreign key. This would require the existence of a null value for the domain of the attribute MNAME. In this case, the null value would have to play a specially defined role in each of the relational algebra operators. The use of a relationship selector obviates the need for nulls and handles this aspect of the college's information structure in a more effective way.

STUDENT RELATION EXTENSION

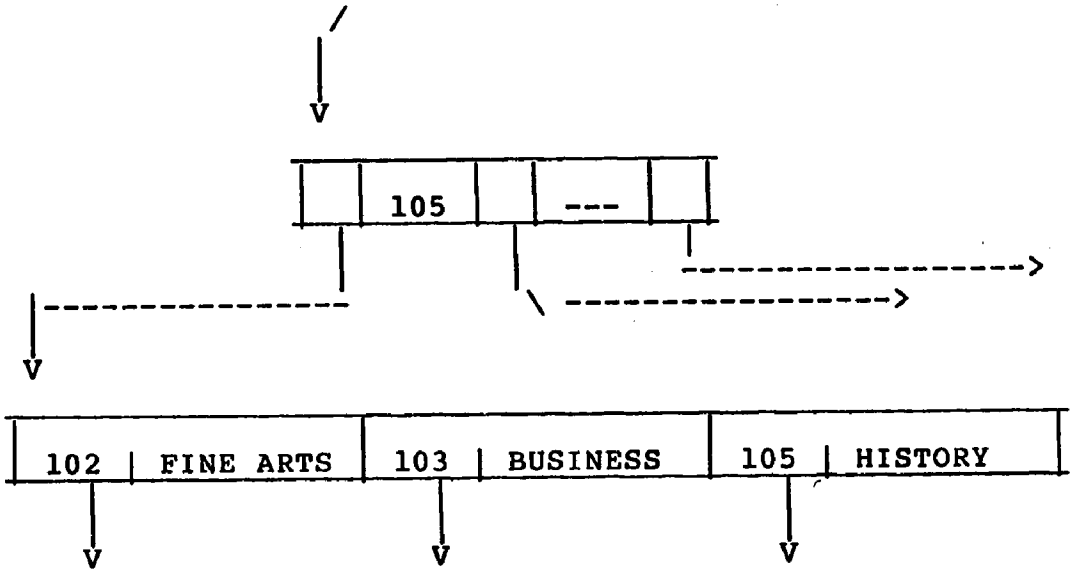
P	-->	101	- - -	FRESHMAN
R I	-->	102	- - -	JUNIOR
I N	-->	103	- - -	SENIOR
M D	-->	104	- - -	FRESHMAN
A E	-->	105	- - -	JUNIOR
R X	-->	-	- - -	- -
Y	-->	-	- - -	- -
	-->	999	- - -	FRESHMAN

MAJOR RELATION EXTENSION

P	-->	BUSINESS	- - -
R I	-->	ENGINEERING	- - -
I N	-->	FINE ARTS	- - -
M D	-->	- - -	- - -
A E	-->	- - -	- - -
R X	-->	HISTORY	- - -
Y	-->	PHILOSOPHY	- - -

A portion of two base relation extensions

Figure 5.2



Index storage structure for a relationship selector

Figure 5.3

In a relationship selector storage structure, the database management system would have to recognize the different format of the sequence set blocks. This formality would not be difficult to implement in a generalized hierarchic block index access method. In fact, similar hierarchic indexes have been designed for incorporation into the experimental relational database management system, SYSTEM R [HAER78]. The proposed implementation of "images" and "links" in SYSTEM R could be readily adapted to the requirements for supporting simple selectors and relationship selectors, respectively.

The suggestion of alternative hierarchic index structures to support simple selectors and relationship selectors appears to be the natural way to implement them. Even though a relational environment has been assumed throughout, this approach would be equally viable in a network or hierarchical environment. Virtually all commercially available database management systems support some form of dense hierarchic block index storage structure for either primary or secondary access paths. These could be easily adapted for this task.

Other storage structures could be used to implement both forms of selectors as well. For example, linked lists might be used for simple selectors. The selected tuples would be chained to one another in either a forward direction or possibly in both directions. Relationship selectors might be implemented by multilist chain

structures. Regardless of the implementation technique chosen, the cost of building and maintaining the storage structure must be considered. The dense hierarchic indexes suggested here may not be the best choice in particular circumstances, but they do offer a single, unified approach to satisfying all required storage structures.

5.2.3 Storage Structures for Simple Adaptive Selectors

In an abstract data model, the database designer may assert the existence of a simple adaptive selector for a particular entity. This serves to explicitly recognize a temporal characteristic of the entity which is not otherwise representable by attributes or relationships. In the generic data model of the preceding chapter, an artificial relation was employed to formally represent the simple adaptive selector. The attributes of this artificial relation are the key attributes of the underlying base relation. For data manipulation, the set membership and set size operators were described.

The subset of tuple occurrences of a base relation identified by a simple adaptive selector are the ones which have been the object of the majority of references relative to a specific point in time. For efficient retrieval from secondary memory, it would be useful if the selected tuples were physically managed so that rapid access could be provided with a minimum of costly secondary storage transfers.

In this sense, the simple adaptive selector is analogous to the Working Set Principle postulated by Denning [DENN67]. In a virtual memory environment, a program is allocated a conceptually large virtual address space organized into fixed size page frames. This address space usually exceeds the amount of available real memory page frames. During execution of the program, references to the virtual address space tend to cluster in a subset of its page frames for periods of time. The "working set" is defined as that subset of a program's pages which have been referenced most recently.

The working set of a particular program is determined by examining the pattern of distinct page references which have taken place over a fixed, backward looking window on time. A program is not considered eligible for execution unless its working set of pages is present in real memory. The supposition is that the program will likely reference this same set of pages in the near future. The content of this working set, however, changes dynamically as the program proceeds through the course of its execution.

The simple adaptive selector is intended to represent a similar phenomenon. Among the tuple occurrences of a particular base relation, it is expected that references will tend to cluster among a relatively small subset over periods of time. Whereas in the working set, the identity of the specific pages is immaterial except to the operating system, in the simple adaptive selector the

identity of the selected tuples is important. The set membership data manipulation operator, in particular, tests whether specific tuple occurrences are currently members of the simple adaptive selector at an instant in time.

There are two requirements then for the implementation of simple adaptive selectors. First, there is a need for a storage structure to continuously record the membership of the selector and to provide the information necessary for the set membership and set size operators. Secondly, there needs to be a policy regarding the physical organization of secondary memory which facilitates the management of the selected tuples so that efficient, rapid access to them may be rendered. The next major section will deal with this question.

As with the simple selectors and relationship selectors, the storage structure chosen for a simple adaptive selector should be separated from the base table on which it is defined. This is especially true for simple adaptive selectors because the physical organization of the underlying base table cannot be constrained by the requirements of any storage structure. Unlike either form of selector, however, the storage structure chosen for a simple adaptive selector will not provide an alternate access path to the base table. Rather, it will merely support the requirements of the set membership and set size operators.

It has been assumed that primary access to base relations has been supported by a dense, hierarchic block index structure. The lowest level of the index contains all of the keys of the base relation along with pointers to the tuple occurrences. In keeping with the philosophy of the B* tree, the entire index structure is kept balanced as insertions and deletions are made. Maintaining the balance of a hierarchic index is generally quite easy once the initial index is constructed. By carefully choosing certain implementation parameters, most insertions and deletions typically affect only one index block. Occasionally, an insertion or deletion requires that several blocks be altered to maintain the balance. When this occurs, the cost of re-balancing can be expensive.

The storage structure proposed for the simple selector and relationship selector is also a hierarchic block index. This second, alternative storage structure contains a subset of the keys and pointers of the base relation. Because both forms of selector are assumed to be relatively stable with respect to their membership, the cost of maintaining the alternative storage structure is minimal.

By choosing dense hierarchic block indexes for all required storage structures so far, a degree of uniformity and simplicity has been achieved. Usually the database designer is faced with a bewildering assortment of storage

structures to choose from. Relying on just one reasonably efficient storage structure simplifies the physical database design process considerably [HAER78].

Storage structure support for the simple adaptive selector presents an entirely different physical design problem. First, the dynamically selected subset is determined by on-going monitoring of references to the tuples of the base table. This monitoring process will identify the temporally active subset of the base table at any point in time. Secondly, the size of the selected subset is theoretically variable; however, in practice some upper bound must be established. The underlying assumption in the use of a simple adaptive selector is that a relatively small proportion of the tuples of a base relation are the most active at any point in time. And lastly, because the membership of the selected subset is expected to change over time, the storage structure used to represent it must be easily and inexpensively changeable, i.e., adaptable.

For these reasons, virtually all of the commonly used storage structures are not appropriate for representing simple adaptive selectors. Hierarchic indexes and pointer-based storage structures, such as linked lists, would suffer from excessive overhead in their maintenance requirements. Conventional direct storage structures, such as pointer arrays, would need to be sorted to provide rapid look-up for tests of set membership.

Severance and Lohman [SEVE76] and Aghili and Severance [AGHI82] have encountered a similar requirement to provide rapid retrieval for a selected subset of the record occurrences of a database. They propose that insertions and updates to existing database records be held in an alternative file called a differential file. At periodic intervals, these accumulated changes are made to the main, permanent database file. Between the reorganizations, requests for retrieval of database records may be made from the differential file (if the desired record is located there) or from the main database file. Because there is no primary access path to the differential file and an exhaustive search would be extremely expensive, a storage structure is needed to quickly determine whether a particular record is presently in the differential file. As with a simple adaptive selector, their storage structure need only determine whether the desired key is present, that is, satisfy a test of set membership. It need not provide an access path to the associated record occurrence (e.g., a pointer).

The storage structure they chose to employ for this purpose is known as a Bloom filter [BLOO70]. This structure consists of a bit vector of some suitable length and a number of independent hashing functions. The length of the bit vector and the number of hashing functions, in the case of a differential file, can be determined by the methods described in Aghili and Severance [AGHI82].

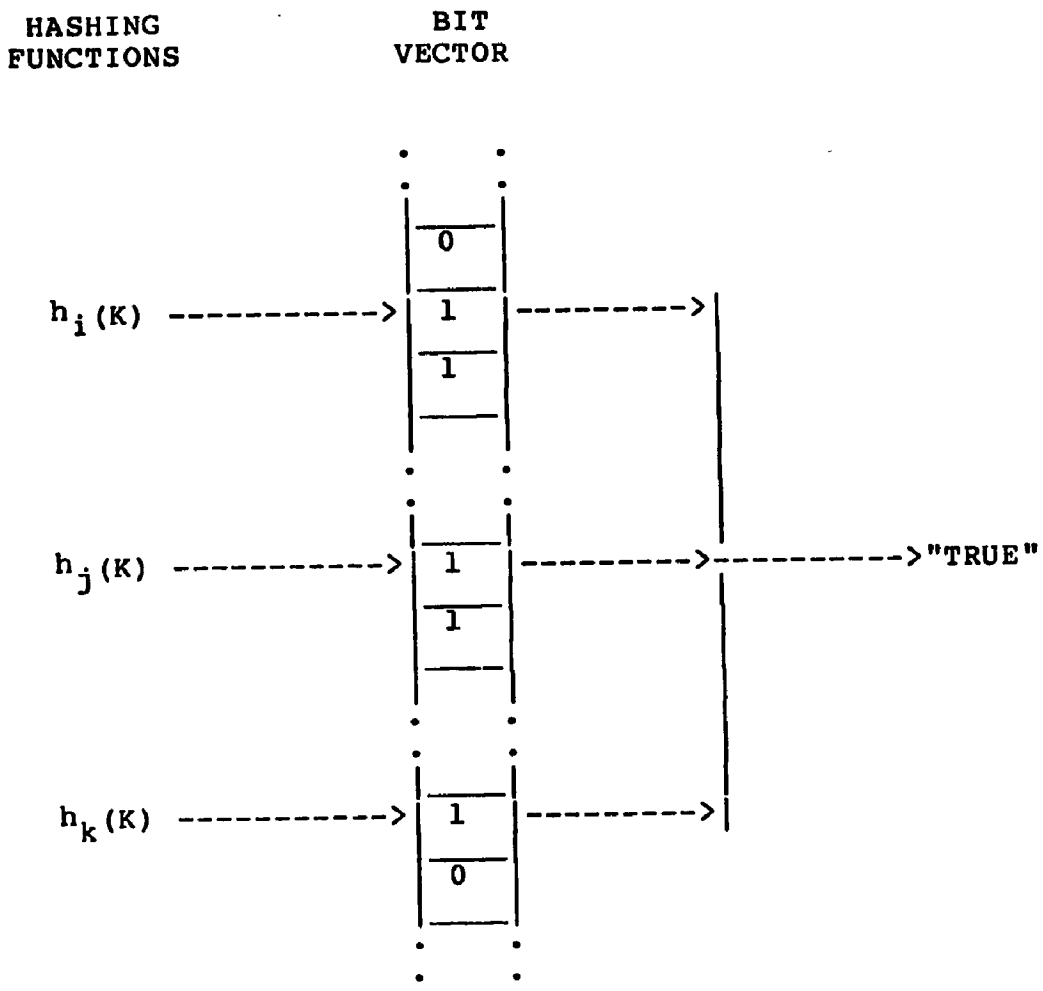
Initially, the bit vector is all zero. As records are inserted or updated and placed in the differential file, the record key is transformed by each hashing function which, in turn, selects a bit in the bit vector to set to one. On subsequent retrieval, the desired key is again transformed by each hashing function. If any of the selected bits is zero, the record is definitely not in the differential file and the main database is searched using its primary access path. If all of the bits are one, then the record is likely, though not necessarily, in the differential file. Periodically, the differential file is merged into the main database file and the process begins anew with the bit vector reset to zero.

One problem with a Bloom filter, as with most hashing schemes, is that it does not support deletions. In hashing different keys to the bit vector, it is common that the same bit might be selected to be set. By deleting a key from the Bloom filter, all of its selected bits would have to be reset to zero. This, in turn, may affect other keys which coincidentally select one or more of the same bits. However, with the proper choice of parameters, the Bloom filter can be designed so that the probability of two distinct keys selecting the same <u>set</u> of bits is arbitrarily small.

Because of the similarity of requirements between the differential file search mechanism and a simple adaptive selector, the Bloom filter will be the recommended storage

structure for simple adaptive selectors. For each base relation on which a simple adaptive selector has been defined, an appropriately parameterized Bloom filter will be constructed. At intervals to be determined by the performance monitor of the host database management system, each storage structure will be reinitialized to represent the subset of the most recently and frequently retrieved tuple occurrences. Between such intervals, the Bloom filter will serve to support any set membership operations for the underlying base table.

Figure 5.4 portrays the general nature of a Bloom filter storage structure. In this case, a portion of the bit vector is displayed with some of its individual bits set to one. There are three separate hashing functions shown, all of which are applied to some key value, K. Each of these hashing functions map onto values of one in the bit vector. The logical "and" of these bits then yields a value of TRUE for a test of set membership operation. In theory, this test might give an erroneous indication that the given key is present in the simple adaptive selector when, in fact, the selected bits have been set as a result of hashing other key values to the same bits. The likelihood of such an occurrence can be controlled by the careful choice of the size of the bit vector and the number of hashing functions employed.



A Bloom filter with three hashing functions

Figure 5.4

The next major section of this chapter will discuss the issues involved in determining the size of each Bloom filter bit vector and the number of hashing functions needed. This physical design problem will have a different objective function than that of a differential file. Specifically, the occurrence of "filtering errors" where the Bloom filter erroneously indicates that a particular key has been selected, must be extremely small.

5.2.4 Storage Structures for Relationship Adaptive Selectors

The simple adaptive selector relies solely on an external criterion for its definition. On-going performance monitoring of the database management system will periodically determine the subset of the tuples of a given base table which will be selected for incorporation in the storage structure used to represent it. Relationship adaptive selectors are also intended to capture a temporal characteristic of an entity. However, the definition of a relationship selector is determined by the entity's participation in a designated relationship at a given point in time.

In developing an abstract data model of an enterprise, various types of relationships are portrayed among entities. These relationships indicate only the potential for a relationship between instances of the associated entities. In an actual stored database, it is

possible that certain instances of the associated entities will not participate in the relationship. The reasons for the non-participation can be varied but one important reason may be that the relationship is meaningful only at certain times.

The example given here is of a relationship OFFERED between the entities COURSE and SECTION. A characteristic of a COURSE is whether or not the COURSE is being offered at a particular time. There is no attribute of a COURSE which conveys this fact but it may be inferred from the association of the COURSE with an instance of a SECTION. For abstract modeling purposes, the concept of a relationship adaptive selector was introduced to explicitly represent this aspect of COURSEs.

In the generic data model of the preceding chapter, relationship adaptive selectors such as CURRENT were represented by artificial relations with the key attributes of the corresponding base relation as their only attributes. This differs from the artificial relations used for relationship selectors where the keys of both base relations were included. The reason for this is that in the relationship selector, both entities are kernel entities with their own independent existence. Neither entity contains the key of the other as a foreign key because the relationship depends on a particular attribute value and is meaningless for those entity occurrences not meeting a boolean qualification on that

value. The inclusion of both keys in the artificial relation supports a join of the two base relations.

In the relationship adaptive selector, one of the entities is a temporal characteristic of the other and consequently derives its existence from it. Therefore the key attribute(s) of the "superior" kernel entity are required in the characteristic entity as part of its composite (concatenated) key. In the example here, CRSENO is the key of the kernel entity COURSE and it is an element of the concatenated key of the temporal characteristic entity SECTION (CRSENO + SECTNO).

This situation presents two somewhat conflicting requirements for a storage structure for the relationship adaptive selector. First there is a need for a storage structure to keep track of the kernel entity occurrences which are presently participating in the designated relationship. This storage structure will support both the set size and set membership operations and must be readily adaptable as the subset of participating occurrences changes over time. Secondly, the storage structure must explicitly represent the keys of the selected subset so that the filter, complement, and implied join operators may be applied to it.

The first requirement could be satisfied efficiently by a Bloom filter storage structure as proposed for the simple adaptive selectors. However, this storage structure would not satisfy the second requirement of

explicitly representing the necessary keys. A hierarchic index storage structure would satisfy all of the requirements but could potentially be expensive to implement and maintain. Because there is no presently available storage structure which simulataneously satisfies both requirements and the keys must be explicitly represented, the hierarchic index is the best choice for the physical representation of relationship adaptive selectors.

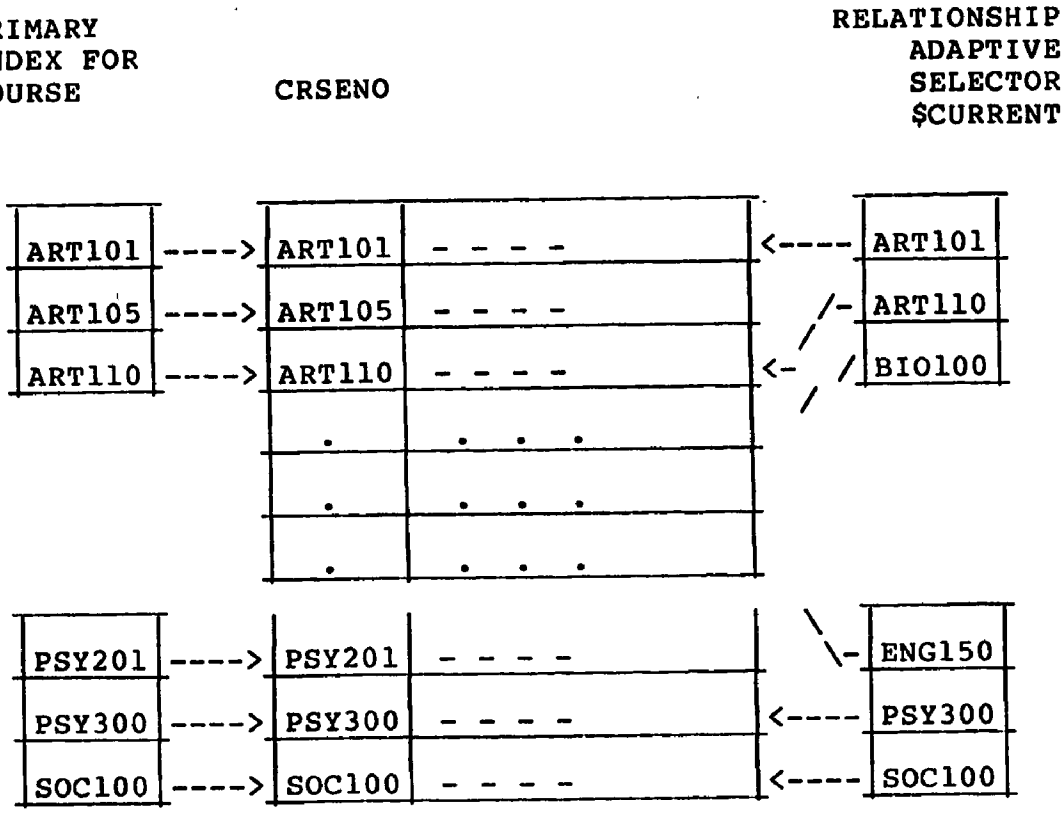
Unlike the simple adaptive selector where the membership is determined by on-going performance monitoring of frequency of retrieval, the membership of a relationship adaptive selector is determined by a data manipulation operation which stores or deletes a temporal characteristic tuple occurrence. The overhead in performing this operation is sufficiently large that the maintenance required to insert or delete a key from the hierarchic index is insignificant.

Figure 5.5 shows a portion of a hierarchic block index used to represent the relationship adaptive selector $CURRENT. The lowest levels of both the primary index and relationship adaptive selector index are portrayed on either side of the base table for the COURSE relation extension. Higher levels of the index set of each are not shown. Below the COURSE relation extension is a portion of the SECTION relation extension to give a flavor of how the two are related through $CURRENT. If there is at least one

occurrence of a SECTION associated with a particular COURSE, then the relationship adaptive selector index contains the appropriate key.

The format of this storage structure is exactly the same as for the simple selector hierarchic block index. The index set blocks each contain the key values and pointers to the associated tuple occurrences in the base table. All of the extended relational algebra operators can then utilize this structure.

COURSE RELATION EXTENSION



SECTION RELATION EXTENSION

CRSENO	SECTNO	
ART101	001	- - -
ART101	002	- - -
ART110	001	- - -
.	.	. . .
.	.	. . .

A hierarchic index for a relationship adaptive selector

Figure 5.5

5.3 Operational Considerations for Adaptability

Three of the storage structures recommended above were predicated on the availability of a hierarchic block index access method within the target database management system. This, in turn, assumed that the host computer's operating system provided some form of indexed sequential access method such as ISAM or VSAM. This assumption is not unreasonable for currently available database management systems (e.g., SQL/DS and INGRES). Consequently, the physical implications of incorporating either form of selector abstraction or a relationship adaptive selector in an existing database management system are not too great. For example, the generalized access path facility described by Haerder [HAER78] could be adapted to these needs quite readily.

Implementation of a simple adaptive selector, however, does require certain enhancements to the underlying database management system. A Bloom filter storage structure, for example, is novel and would not generally be found in an existing database management system. This section will examine some of the requirements needed for implementing simple adaptive selectors and also describe certain other facilities which may be added to a DBMS to improve its overall performance.

5.3.1 Performance Monitoring

Of primary importance to the implementation of a simple adaptive selector is the need for a mechanism to identify the the temporally active subset of a base relation. The fundamental assumption behind the use of a simple adaptive selector is that a relatively small percentage of the tuples in a base relation will be the object of a disproportionate amount of the retrievals directed to that base relation. Although there are no hard and fast rules to support such an assumption, it is a well-known folk theorem in commercial data processing as well as other fields that as few as twenty percent of the records in a file will be referenced as often as eighty percent of the time. This so-called 80/20 Rule as been empirically verified [HEIS63].

Knowing that such a phenomenon is likely to affect a given base table in a relational database is of no particular use to the physical design process unless the pertinent subset can be identified. Also, it is very likely that the membership of the active subset will change over the operational life of the stored database. The physical design aids presently available analyze known applications and attempt to formulate estimates of retrieval patterns in terms of volume and frequency. No attempt is made at identifying the individual occurrences.

After a database has been designed, implemented, populated, and in operation for a period of time, it is possible to both verify the initial assumptions concerning patterns of reference and to identify the specific tuples (records) which are being retrieved most frequently at that point in time. Database management systems are typically equipped to record this information during operation. References to tuples, especially for insertion and update are generally recorded in separate areas for back-up and recovery purposes as well as to provide an audit trail. This information can be easily augmented to capture reference patterns for simple retrievals also.

To effectively implement a simple adaptive selector, this kind of monitoring will be required. An analysis of the reference patterns directed at a given base table will reveal the high activity subset, if such exists. This information will then enable the construction or updating of the storage structure used to represent the simple adaptive selector. The question of how the reference pattern will be analyzed depends on the nature of the performance monitor available with a particular DBMS. It may be that the performance monitor maintains a reference count field for each occurrence. This field might be updated continuously (at great expense) or periodically to form an empirical estimate of the distribution based upon a sample of references. Alternatively, a transaction reference log may be examined for this purpose.

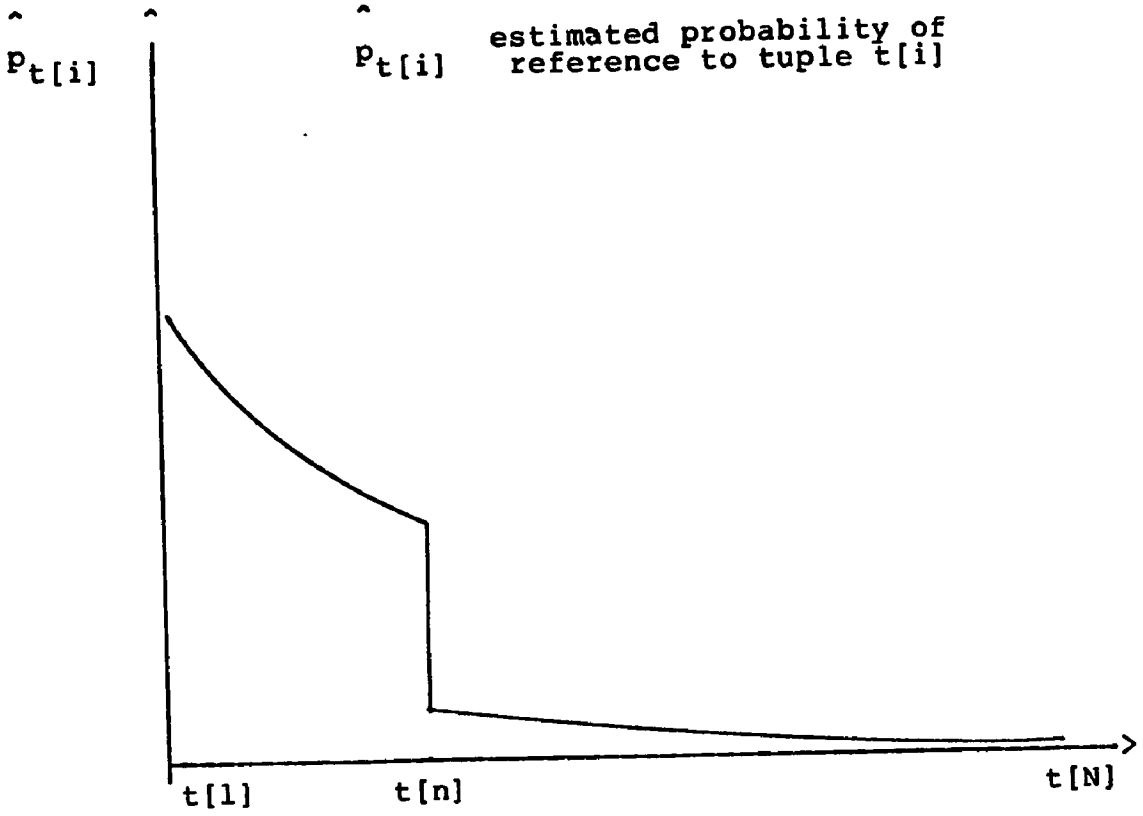
The frequency of the analysis and updating of the simple adaptive selector must be a decision made by a database administrator given the particular circumstances of the environment. This kind of decision represents the external criterion which completes the semantic definition of a simple adaptive selector. The database administrator, with knowledge of the nature of the data, the applications which process it, and the needs of the enterprise can establish the frequency of updating. Regardless of the frequency of updating, the full benefit of a simple adaptive selector will be realized if the identified subset can be physically clustered to improve I/O performance.

5.3.2 Bloom Filter Parameter Selection

The recommended storage structure for implementing simple adaptive selectors is a Bloom filter. This structure has the desirable qualities of being easy to build and maintain, providing very rapid response for the set membership operator, and not requiring too much storage. In order to implement a Bloom filter, three design decisions must be made. First, the size of the simple adaptive selector in terms of the number of keys to be stored must be established. Next, the size of the bit vector and number of independent hashing functions must be determined. These last two decisions depend on the first and are not independent of each other.

The implicit assumption underlying the use of a simple adaptive selector is that, at any point in time, a relatively small proportion of the tuple occurrences will be more likely to be referenced and that this subset will continuously change over the operational life of the database. From a probabilistic perspective, this would suggest that the reference process is non-stationary. For practical purposes, it will be assumed that the distribution of references is nearly stationary between evaluation intervals for the simple adaptive selector. The information obtained from the performance monitor can be used to develop an empirical estimate of the reference distribution. The size and membership of the simple adaptive selector can then be determined.

Figure 5.6 portrays the type of situation that might be encountered when analyzing performance monitor data collected during a simple adaptive selector evaluation interval. A probability plot of the empirical reference distribution is shown for convenience as a continuous function. The ordinate shows the relative frequency of reference for each stored tuple since the last sample (evaluation) interval. The abscissa represents the tuple occurrences in terms of their order statistics. A clear dichotomy between the frequently referenced tuples and the less frequently referenced tuples is evident.



Probability plot of an empirical reference distribution

Figure 5.6

If the empirical distribution appears to be flat, or nearly rectangular, indicating uniform frequency of access, then a simple adaptive selector is not warranted. This occurrence may be only temporary or it might indicate that a simple adaptive selector for the relation in question was inappropriate. However, it is expected that an empirical reference distribution of the type shown in Figure 5.6 will be obtained at each evaluation interval.

The point labelled "t[n]" on the abscissa demarks the selected subset from the remainder of the tuple occurrences. This point should obviously represent less than half of the total tuple occurrences for a simple adaptive selector to be useful. The cumulative mass under the curve up to and including this point will be denoted as p. The remainder of the distribution accounts for a cumulative mass of q=(1-p). The quantities p and q can be interpreted as the probability of a <u>random</u> reference being for a member of the selected subset or for a non-selected tuple, respectively. If the 80/20 Rule were applicable to this situation, then n=0.20*N, p=0.80, and q=0.20.

Having determined the number of keys, n, to be represented in the Bloom filter, it remains to establish the size of the bit vector, say M, and the corresponding number of independent hashing functions, X. If a given key has been selected as a member of the temporally active subset, the Bloom filter will always correctly indicate this fact regardless of the size of the bit vector and

number of hashing functions. The only potential problem with the storage structure is that it might incorrectly indicate membership for keys which have not been selected.

The analysis to follow is based on the original work of Bloom [BLOO70] and the subsequent research contained in Severance and Lohman [SEVE76]. An interesting example of the design of a Bloom filter for a differential file is given in Gremillion [GREM82].

The problem of determining a bit vector size and number of independent hashing functions may be formulated in the following way. Let n be the number of keys to be represented in the bit vector. Let p be the probability that a reference to the Bloom filter, i.e., a test of set membership, is for one of the n selected keys; then q=(1-p) is the probability that a reference is for one of the non-selected keys. In this latter instance, the Bloom filter may erroneously indicate that the key has been selected. The criterion for choosing M and X will be to minimize this potential occurrence.

The n selected key values represent a sample drawn from the underlying domain of the key attribute(s). As is often the case, these key values may not be uniformly distributed over this domain. Regardless, the X hashing functions will be assumed to be chosen such that they will map these keys uniformly into the bit vector.

For a Bloom filter bit vector of length M, after hashing n key values to bit addresses via X independent

hashing functions, the probability of any randomly selected bit being set to one is given by:

$$\Pr[\text{a random bit set}] = 1 - ((M - 1)/M)^{nX}$$

For M sufficiently large, the above expression may be approximated by:

$$\Pr[\text{a random bit set}] = 1 - \exp(-nX/M)$$

Performing a test of set membership on one of the non-selected keys, the conditional probability of finding all of the X bits set to one, i.e., the probability of a filtering error is:

$$\Pr[\text{all X bits are set} \mid \text{key not selected}] = [1 - \exp(-nX/M)]^X$$

The probability of referencing the Bloom filter storage structure with one of the non-selected keys is q, therefore the unconditional probability of a filtering error is:

$$P(n,X,M) = q\,[1 - \exp(-nX/M)]^X$$

With n a fixed constant, the probability of a filtering error is then a function of both M and X. As a practical matter, the size of the bit vector must be limited. An extremely large bit vector would enable the filtering error rate to approach zero but then other storage structure options would be more plausible, e.g., a

dense hierarchic index. Also, it is desirable for the simple adaptive selector storage structure to provide rapid response to tests of set membership, even at the expense of a modest filtering error rate. Consequently, the value of M can be fixed by the database designer after considering both the number of keys to be represented and the amount of processor memory available to store the bit vector.

With n and M both fixed, classical unconstrained optimization techniques can then be used to determine the appropriate number of independent hashing functions needed to minimize the probability of filtering errors. Severance and Lohman [SEVE76] have shown that differentiating P(n,X,M) with respect to X, setting the derivative to zero, and solving for X yields:

$$X' = M \ln(2)/n$$

where ln is the natural logarithm.

For X' chosen as above, the expected number of bits set in B, after storing n key values, is M/2. Therefore, the probability of any randomly selected bit being set to one is simply 1/2. The minimum probability of a filtering error, given n, M, q, and X', is then:

$$\begin{aligned} P'(n,X',M) &= q\ (1 - \exp(-nX'/M))^{X'} \\ &= q\ (1/2)^{M \ln(2)/n} \\ &\sim= q\ (0.6185)^{M/n} \end{aligned}$$

In practice, X' will frequently evaluate to a non-integer value. The last step in the process of parameterizing a Bloom filter would then be to examine P(n,X,M) at the two integers which surround the value of X'. The integer resulting in the smallest probability should be chosen.

This particular formulation of the design of a Bloom filter storage structure for simple adaptive selectors is applicable to a wide number of situations. The only quantities needed are n, q, and M. The first two can be determined readily from the analysis of the performance monitor data and the last can be determined on the basis of available processor memory. It is interesting to note that the size of the underlying base relation is immaterial. All that is necessary is that n be less than N/2.

As a practical example of this storage structure design process, consider the simple adaptive selector ACTIVE presented in Figure 3.12. In the abstract data model, explicit recognition is given to the fact that at any time a subset of all of the STUDENT-ACCOUNT occurrences will be the object of the majority of references. The exact membership of this subset is not known in advance and is likely to change over time, however, it is a semantically meaningful aspect of the college's information structure that such a pattern of reference will occur. This temporal partitioning in the

occurrence dimension of STUDENT-ACCOUNT will then have implications for the design of the conceptual schema, the data manipulation operations performed on the relation, and the physical storage of the relation extension.

The DDL definition of the simple adaptive selector $ACTIVE is shown in Figure 4.12 where an artificial relation is defined as a powerset over STUDENT-ACCOUNT. The data manipulation operators of set size and set membership can then be applied to this artificial relation. It remains to design a storage structure to physically represent this artificial relation.

It has been assumed that the college has 4,000 students and that each of them has a STUDENT-ACCOUNT occurrence. After an initial period of operation, the performance monitor detects that twenty percent, or 800, such STUDENT-ACCOUNTs are receiving a disproportionately high percentage of retrieval activity, say eighty percent of all references. A Bloom filter storage structure may then be designed to provide rapid response to tests of set membership for these tuples.

The database administrator may decide to allocate 500 bytes for the bit vector. In this case, the bit map can be locked in processor memory so that it is always available for data manipulation. The bit map will then contain 4,000 bits.

Using the approach described above, the relevant constants are:

$N = 4000$

$n = 800$

$M = 4000$

$p = 0.80$

$q = 0.20$

It remains to solve for X, the number of independent hashing functions, and the corresponding filtering error rate. The solution for X is obtained as:

$$X' = M \ln(2)/n = 4000 \ln(2)/800 \tilde{=} 3.466$$

The minimum filtering error rate for this choice of X is given by:

$$P'(n,X',M) = q\ (0.5)^{M \ln(2)/n}$$
$$= 0.20 * (0.5)^{3.466} \tilde{=} 0.01810$$

Because X' is not an integer, the two integer values nearest to this optimum must be examined. For three independent hashing functions, the filtering error rate is found to be 0.0250 and with four hashing functions the error rate is 0.0125. Therefore, given a bit vector of size 4,000 bits, four hashing functions can be used with the Bloom filter to yield a near optimum error rate of less than two percent.

To gain some insight into how the various design parameters interact, Table 5.1 shows the values of P(n,X,M) for various combinations of n, M, and q. The

left-most column indicates the ratio of M to n, that is, the size of the Bloom filter bit vector relative to the number of keys to be represented. In parentheses after this ratio is the integral number of independent hashing functions needed for this ratio of M to n. The columns indicate the probability of reference to a non-selected key and the body of the table contains the associated probability of a filtering error.

M/n(X) \ q = Pr[reference to non-selected key]	0.10	0.20	0.30	0.40	0.50
1.0(1)	0.06321	0.12642	0.18963	0.25284	0.31605
1.5(1)	0.04866	0.09732	0.14598	0.19464	0.24330
2.0(1)	0.03935	0.07870	0.11805	0.15740	0.19675
2.5(2)	0.03032	0.06064	0.09096	0.12128	0.15160
3.0(2)	0.02368	0.04736	0.07104	0.09472	0.11840
3.5(2)	0.01895	0.03790	0.05685	0.07580	0.09475
4.0(3)	0.01469	0.02938	0.04407	0.05876	0.07345
4.5(3)	0.01152	0.02304	0.03456	0.04608	0.05760
5.0(3)	0.00919	0.01838	0.02757	0.03676	0.04595

Typical filtering error rates

Table 5.1

The first observation is that the filter error rate decreases as the size of the bit vector, relative to the number of keys, increases. This is not altogether obvious because, given the formulation above, the expected number of bits set in the bit vector is M/2 regardless of the number of keys, n. This phenomenon is explained by the relative size of the bit vector in combination with the number of independent hashing functions employed.

The filter error rates shown in the table are calculated using the integral number of hashing functions closest to the minimum error rate achieved at a non-integer X. Even for relatively large bit vectors, the number of hashing functions remains quite modest. The last observation is that there is a marked increase in the filter error rate as the probability of referencing a non-selected key increases. This result is interesting in that the derivation of the values contained in the table do not involve either n or N, the total number of tuple occurrences, directly.

Although the Bloom filter storage structure offers several distinct advantages for implementing simple adaptive selectors, the results shown in the table suggest some guidelines for its use. First, a Bloom filter will be effective if the size of the bit vector can be kept relatively small. If a very large bit vector is required to achieve an acceptable error rate, alternative storage

structures may prove to be superior, e.g., a dense hierarchic index.

Secondly, regardless of the number of selected keys to be represented, if the probability of referencing non-selected keys is high, it may not be possible to achieve an acceptable error rate. The determination of n, p, and q in the design process comes from an analysis of the empirical reference distribution. A large value of q might be indicative of a nearly uniform reference distribution, in which case the simple adaptive selector may not be warranted. Or, the number of selected keys may be so small relative to the total number of tuple occurrences that other storage structures may be more appropriate.

Lastly, at each evaluation interval, the database designer may opt to change any of the parameters of the Bloom filter as the performance monitor evidence indicates. This includes not supporting a simple adaptive selector at periods of time when observed reference activity is effectively uniform over the tuple occurrences.

5.3.3 Secondary Memory Organization

At present, the process of physical database design only treats operational issues as they relate to pre-implementation decisions. Once made, these decisions are extremely difficult to change. Typically,

operationally oriented information such as data volume and transaction frequency are estimated in order to decide on efficient record segmentation, secondary memory block sizes, and file (device) allocation. The effectiveness of these design decisions will likely diminish as the database is installed and in operational use for a period of time.

The very nature of the selector and adaptive selector abstractions is that they are to provide a measure of adaptability to the database not only in its logical design but also throughout its operational lifetime. To accomplish this, a database management system should not only recognize and represent adaptability but should also provide a flexible environment for managing adaptability at the level of physical occurrences. This subsection and the next will explore two proposals which will enable a database management system to effectively respond to the dynamics of the occurrence dimension with an aim at improving the overall operational performance of the stored database.

The principal operational task of a database management system is to provide efficient access to the stored data occurrences. This involves the management and coordination of a hierarchy of storage devices. Two levels of the storage hierarchy will be focused upon here: primary (processor) memory and secondary memory. There are other levels in a typical storage hierarchy, such as a

high-speed cache memory and archival memory, but they are not of direct concern for what follows.

Secondary memory provides the permanent repository for a stored database. Data occurrences are organized into logical groupings such as tuples (records) and these are further aggregated into larger units, or blocks, for efficient transfer to primary memory. Because secondary memory devices are relatively slow, transfers of blocks of data to and from processor memory are quite expensive. The operational performance of a DBMS is then critically linked to the efficiency of secondary memory organization.

Traditionally, data occurrences are organized in secondary memory according to criteria not related to performance considerations. Tuples or records may be allocated to blocks in some logical order such as increasing primary key sequence. In some instances, tuples of different types participating in certain relationships may be clustered closely together. These criteria often require that the data occurrences be "pinned" [ULLM80] to their secondary memory locations. The storage structures used to support primary, relationship, and secondary access to the data occurrences depend on these fixed, permanent addresses. While these criteria are logically sensible, they are not necessarily the most efficient organizations from the perspective of efficient retrieval.

The secondary memory organization proposed here is to allow all data occurrences, at the aggregate level of

tuples, to be "unpinned." That is, tuple occurrences are free to be moved or relocated as long as all of the storage structures used to provide access to the tuples are maintained consistently. This particular organization is more costly from the storage structure perspective but can be highly efficient in overall performance. All required access paths to unpinned tuple occurrences can be provided by existing storage structures such as dense hierarchic indexes [HAER78].

The consequence of allowing tuple occurrences to be unpinned, and potentially relocatable, is that the placement of the tuple occurrences can be based on a criterion of efficient retrieval. Recognizing that the pattern of access to stored tuple occurrences is very unlikely to be uniform, the tuple occurrences can be distributed among blocks of secondary memory in a way which can reduce significantly the number of costly block transfers between secondary and primary memory.

To affect such an organization requires that performance monitoring data be kept on the relative frequency of access to individual tuple occurrences. These data would then provide the information to identify and collect the more frequently retrieved tuple occurrences into a set of high frequency of reference blocks which would serve to minimize the number of secondary memory references by maximizing the likelihood of finding a desired occurrence in processor memory. The periodic

relocation of tuple occurrences could be carried out as often as deemed necessary over the operational life of the database.

The price to be paid for maintaining this organization includes the cost of the reorganization as well as the additional overhead for the various storage structures. This price, however, is not necessarily excessive because database files and their associated storage structures require periodic reorganizations to recover space freed by deletions and to merge overflow areas into the main database files [HELD78].

A DBMS policy which periodically reorganizes its secondary memory on the basis of observed frequency of reference to individual tuples can be designed and implemented regardless of whether selector or adaptive selectors are used. The only requirements for adopting such a policy are that the secondary memory organization does not pin tuple occurrences to fixed locations and that some performance monitoring capability is provided to trace reference patterns. Selectors and adaptive selectors, however, provide a unique opportunity for exploiting this secondary memory organization.

In the case of a simple selector, the occurrences of tuples of a particular type are partitioned on the basis of a boolean qualification of an attribute value. With a flexible secondary memory organization, the selected tuple occurrences can be clustered into a set of logically

contiguous blocks. When the filter operator is applied to these occurrences, only those blocks containing the selected tuples need be transferred to processor memory. Similarly, the complement operator would reference only blocks containing non-selected tuple occurrences.

This would significantly improve retrieval performance when either of these two operators is used in conjunction with a query. Also, within either subset of blocks, individual tuple occurrences can still be ordered on observed frequency of reference. This same effect can be achieved when a relationship selector is defined.

The most obvious case for periodically reorganizing secondary memory on observed frequency of reference is the simple adaptive selector. The criterion for membership in a simple adaptive selector is based on the recency and frequency of reference to specific tuple occurrences. Maintaining a storage structure to support a test of set membership operator is of only marginal value if the selected tuples cannot be retrieved efficiently.

When establishing or evaluating (i.e., updating) the storage structure for a simple adaptive selector, tuple occurrences are selected for membership by examining performance monitor data. As individual tuple occurrences are selected, they can simultaneously be clustered into a set of blocks. These blocks, in turn, would have a much higher probability of reference and consequently would be

more likely to be found in the processor memory buffer area when requested.

Lastly, a relationship adaptive selector would be treated in much the same way as the simple selector or relationship selector. Those tuple occurrences selected because they are currently participating in a specified relationship would be physically clustered in secondary memory.

These assertions about measurable improvements in the retrieval performance of an operational database, although based on strong intuitive grounds, have not been proven analytically. There are simply too many factors to consider when trying to characterize the magnitude of any such improvement. For example, the stochastic nature of the reference process, the effect of multiple, concurrent database users, the operational characteristics of the DBMS and host operating system, and the frequency of the periodic reorganizations would all have an effect on any measure of performance.

5.3.4 An Adaptive Buffer Management Policy

The periodic relocation of data occurrences based on observed frequency of access has as its ultimate goal the improvement of database management system performance through the efficient and effective organization of secondary memory. There are numerous other opportunities to make improvements in DBMS performance ranging from

query optimization in the data manipulation language to the careful choice of additional, secondary storage structures. It is generally accepted, however, that the most significant improvements in performance will be achieved when the rate of data block transfers between processor memory and secondary memory is minimized.

Because of the great disparity in the speed of access between these two levels of memory, the cost of transferring blocks of data to and from processor memory is the dominant factor in DBMS performance. This issue was raised in the proposal for periodic tuple relocation. The purpose of the relocation of tuple occurrences in secondary memory was to cluster the high frequency of reference tuples into blocks which would have a correspondingly high probability of being found in processor memory thereby reducing the number of transfers needed to satisfy retrieval requests.

It was argued that this alone would lead to measurable performance gains. In order to fully realize these gains, consideration must be given to the management and organization of processor memory as well. The processor memory region allocated to hold the transferred blocks of data occurrences is the DBMS buffer area. This area is much smaller than the secondary memory area required to hold the entire stored database. Eventually, blocks of secondary memory to be brought into processor memory must displace previously transferred blocks because

the buffer will be found to be full. To the extent that the buffer area can be managed to increase the likelihood of finding a desired block in the buffer, the number of these costly transfers can be reduced.

The problem of organizing and managing a DBMS buffer area is quite similar to the problem encountered in virtual memory operating systems [DENN70]. In both cases, a relatively small (real) processor memory area is available to accommodate the contents of a rather large (virtual) secondary memory area. The key to efficiently managing a DBMS buffer area lies in the choice of the policy used to decide which buffer block to displace when a newly transferred block of data finds the buffer full.

Numerous buffer management policies have been proposed and analyzed (viz., Coffman and Denning [COFF73] and King [KING71]). They differ chiefly in the nature of the information they use to make the replacement decision. The comparative measure of efficiency of a buffer management policy is the observed rate of buffer faults, or block replacements, it experiences in servicing a suitably long sequence of independent data block references. The Appendix will describe in detail the assumptions usually made in calculating buffer fault rates for specific policies.

The most widely used buffer management policy is the well-known Least Recently Used (LRU) policy. In the LRU policy, the identity of the blocks currently in the buffer

is kept (theoretically) in a stack. Each time a buffer block is referenced its identifier is advanced to the top of the stack with the other block identifiers being pushed down in the stack. When a block is to be transferred to a full buffer, the block currently occupying the last (bottom) stack position is the candidate for displacement.

The information used in the LRU replacement policy reflects the most recent reference history of the buffer. Presumably, the block chosen for replacement has not been referenced recently and is therefore unlikely to be referenced in the near future. This policy does not take into account the relative frequency of reference to individual data blocks but relies solely on the recency of reference.

An interesting, albeit impractical, policy is the A0 buffer management policy of Denning, Chen, and Schedler [DENN68]. This policy assumes full and complete knowledge of the stationary block reference probabilities. The replacement decision rule is always to replace the buffer block which has the least, known probability of reference. For a database consisting of N data blocks in secondary memory, a processor memory buffer area of size n ($n < N$) will eventually contain the (n-1) highest probability of reference blocks. The remaining buffer block will then be used to service all references to the N-n+1 lower probability data blocks.

Because of the impossibly of knowing the exact, stationary block reference distribution, this particular buffer management policy is not implementable. Under this assumption, however, it is known to be the optimum buffer management policy. Therefore, the A0 policy provides a lower bound on the buffer fault rate and is used as a standard for comparison of other policies. Where the LRU policy was concerned with recency of reference to data blocks, the A0 policy is concerned strictly with the frequency (probability) of reference.

A buffer management policy which combines both of these criteria in its replacement decision, and is feasible to implement, should be able to offer substantial gains in the reduction of buffer faults. The following proposal for a Least Frequently and Recently Used (LFRU) buffer management policy will be shown to be no worse that the LRU policy and to approach the optimum A0 policy as the distribution of data block references becomes increasingly non-uniform.

To meet the criterion of recency of reference, the LFRU buffer management policy will maintain a modified LRU stack discipline. The identity of each data block will be entered into the top of the stack when referenced and will be pushed down in the stack as subsequent blocks are entered. Associated with each data block is an empirical estimate of its reference probability obtained from the performance monitor data. This estimate will be used to

incorporate frequency of reference into the replacement decision.

For every data block currently represented in the LRU stack, a quantity called the residual life expectancy will be calculated. Given the data block's position in the LRU stack, the residual life expectancy is an estimate of the probability that the block will be rereferenced before it would be removed from the LRU stack. The replacement decision rule is then to remove that block from the buffer which has the smallest residual life expectancy. This will frequently, but not always, be the block in the lowest (bottom) stack position.

The residual life expectancy is calculated in the following manner. Assume that there are a total of N data blocks in the database and there are n buffer slots available ($n < N$). For a data block j currently in the i-th stack position (the top stack position is 1) with estimated reference probability p_j, the residual life expectancy is given by:

$$1 - (1 - p_j)^{n-i+1}$$

When p_j is small, this can be approximated by:

$$1 - \exp(-(n-i+1)p_j)$$

The interpretation of this expression is that the block in question is (n-i) positions from the bottom of the LRU stack. The probability that block j is not

referenced in the next (n-i+1) independent references to data blocks, and consequently would leave the LRU stack, is $(1 - p_j)^{n-i+1}$. One minus this quantity is the expected probability that block j will be rereferenced. Using the estimated block reference probability in conjunction with the relative stack position then combines both the criteria of frequency of reference with recency of reference.

Figure 5.7 demonstrates the use of residual life expectancies in the LFRU buffer management policy. A simple five position LRU stack is used in each part of the figure. The relative stack position, block identifier, estimated probability, and computed residual life expectancy constitute the columns. The top portion of the figure illustrates a typical situation with a full buffer. A reference to a data block not present in the buffer would cause block 57 to be displaced to accommodate the entering block. This is exactly the decision that would be made in a strict LRU buffer.

STACK POSITION	BLOCK NUMBER	PROBABILITY ESTIMATE	RESIDUAL LIFE
1	43	0.0012	0.0060
2	87	0.0025	0.0100
3	16	0.0009	0.0027
4	93	0.0012	0.0024
5	57	0.0020	0.0020

(a) Initial configuration

STACK POSITION	BLOCK NUMBER	PROBABILITY ESTIMATE	RESIDUAL LIFE
1	93	0.0012	0.0060
2	43	0.0012	0.0048
3	87	0.0025	0.0075
4	16	0.0009	0.0018
5	57	0.0020	0.0020

(b) Internal stack reference

STACK POSITION	BLOCK NUMBER	PROBABILITY ESTIMATE	RESIDUAL LIFE
1	35	0.0008	0.0040
2	93	0.0012	0.0048
3	43	0.0012	0.0036
4	87	0.0025	0.0050
5	57	0.0020	0.0020

(c) External block reference

LFRU stack configurations

Figure 5.7

The middle portion of the figure shows the resulting stack arrangement after a reference to block 93 which is already in the buffer. The purpose here is to illustrate the subsequent stack configuration and the changes in the residual life expectancies. In this state, block number 16 in the next to last stack position is the candidate for displacement under the LFRU replacement policy. The bottom portion of the figure then shows the stack arrangement when block 35 (not presently in the stack) is referenced.

If the reference distribution to the data blocks is near uniform, the LFRU policy will behave exactly as the LRU policy. When the reference distribution shows marked non-uniformity, the LFRU policy will tend to behave as the A0 policy using estimated block reference probabilities rather than known probabilities. In addition to the potential for achieving near optimum performance, the LFRU policy will be able to adapt to changing patterns of reference as detected by the performance monitor.

To assess the effectiveness of the LFRU buffer management policy, a set of simulation experiments has been performed to measure the relative buffer fault rates under the LRU, A0, and LFRU policies. The Appendix describes the simulation model and its assumptions in detail. Briefly, however, the simulation experiments were based on the Independent Reference Model [COFF73]. This model consists of generating a sequence independent and identically distributed random variables which represent

database block references. A processor memory buffer of a given size is managed according to the chosen policy and the number of buffer faults observed in processing a sufficiently long reference stream is recorded.

In the experiments performed here the database size was chosen to be 20,000 tuple occurrences. These tuples were then distributed randomly over 2,000 database blocks. The processor memory buffer was limited to holding 20 blocks, or one percent of the database. Although these sizes are relatively small with respect to typical databases, the important factor is not the absolute size but rather the relative size of the buffer to the entire database. Increasing or decreasing either the database size or the buffer size would make no appreciable difference as long as the ratio of the buffer to the database remainder the same.

The mass function which assigns probabilities of reference to the individual tuples was taken from Knuth [KNUT69]. The mass attributable to the i-th (ordered) tuple is given by:

$$[\ i^k - (i-1)^k] \ / \ N^k$$

where N is the total number of tuples and k ($0 < k <= 1$) is the skewness parameter. Clearly, when k is equal to one, the mass function is uniform. As k approaches zero, the distribution becomes increasing skewed. With k = 0.1386, the distribution becomes exactly the 80/20 Rule,

that is, 80 percent of the mass is assigned to the first 20 percent of the tuple occurrences.

The experiments consisted of using each of three buffer management policies and observing the buffer fault rate over 100,000 independent and identically distributed random references to tuples. The references were generated using the probability mass function above with k set at 0.8 (very nearly uniform), 0.5 (modestly skewed), and 0.1386. For the A0 policy, the exact probabilities of reference to tuples, and consequently the cumulative block probabilities, were known and used in the replacement decision. In LRU, the tuple probabilities are not needed and under LFRU, the probabilities of reference were estimated over the course of the simulation runs.

Table 5.2 summarizes the results of these experiments. When the reference distribution is nearly uniform (k = 0.8), the observed fault rates are quite high which is consistent with the fact that the buffer can only accommodate one percent of the data blocks. The A0 policy establishes the lower bound for this particular reference stream. LRU and LFRU lie above the bound and are virtually identical.

SKEWNESS PARAMETER k	BUFFER REPLACEMENT POLICY	OBSERVED FAULT RATE
0.8	LRU	0.99078
	LFRU	0.99082
	A0	0.98893
0.5	LRU	0.98896
	LFRU	0.97999
	A0	0.96324
0.1386	LRU	0.79269
	LFRU	0.69282
	A0	0.66098

Experimental results

Table 5.2

As the distribution becomes more skewed with k = 0.5, all three policies begin to show slightly improved performance. A0 is again the lower bound on the fault rate for the reference stream in question. The decrease in its fault rate from the nearly uniform case is due to the fact that the replacement policy can take advantage of the known, non-uniform reference probabilities. With a moderate degree of non-uniformity the LFRU policy can be seen to be superior to LRU, albeit only slightly.

A marked difference in the three policies is evident in the highly skewed 80/20 case. Not only does the lower bound established by A0 drop significantly, but the performance of LFRU is approaching this optimum rate. In

fact, if the simulated reference stream was extended indefinitely the estimated probabilities would converge to the true, underlying probabilities and LFRU would eventually converge to the optimum fault rate.

These results do not constitute a formal verification of the superiority of LFRU over the standard LRU buffer management policy. This is due in large part to the fact that the results were obtained under ideal, and perhaps unrealistic, conditions. They do, however, provide some insight as to the LFRU policy could offer significant performance improvement when the distribution of data block references is non-uniform.

5.4 Summary

The third, and last, level of the database design methodology portrayed in Figure 2.1 concerns the design decisions which must be made at the time of physical implementation. The majority of these decisions are made in the context of the particular computer system environment. Among these decisions, however, the storage structure selection problem can be examined somewhat independently of the target DBMS and host operating system.

The first major section of this chapter addressed the storage structure selection problem within the framework of the generic data model developed in the preceding chapter. The primary goal was to suggest possible storage

structures which would be suitable for implementing selectors and adaptive selectors. It was assumed that a hierarchical dense block index storage structure (or access method) was available. This assumption is not unrealistic as many commercially available DBMSs provide such a structure. Primary access to all stored base relation extensions was then assumed to be implemented in this way. Secondary and relationship access paths were assumed, without loss of generality, to be non-existent.

The suggested storage structure to support simple selectors, relationship selectors, and relationship adaptive selectors was then to build and maintain separate hierarchical dense indexes. In each of these three cases, it was argued that the hierarchical dense index structure was not only an efficient, economical choice of storage structure but that it could be employed effectively by the filter, complement, and implied join extended relational algebra operators.

Storage structure support for simple adaptive selectors, however, presents a different set of requirements. First, the membership of a simple adaptive selector is determined by an externally defined criterion, that is, observed recency and frequency of reference. This then suggests that the membership of the simple adaptive selector will likely change over time. Secondly, with a simple adaptive selector, it is not necessary for its storage structure to provide an alternative access path.

Instead, it must efficiently support the set membership operator.

For these reasons, the recommended storage structure for the simple adaptive selector was the Bloom filter. Because of the similarity of the requirements of a simple adaptive selector and a differential file, it was argued that this storage structure would provide the necessary efficiency, simplicity, and flexibility to support a dynamically changing set membership.

Both the hierarchic dense index and Bloom filter storage structures represent only suggested storage structures. However, they do satisfy one of the goals of this chapter, that is, the recommendations demonstrate the feasibility and practicality of including the occurrence dimension in the internal data model representation of a database design.

The next major section of this chapter then addressed certain operational considerations which would necessarily affect an operational database management system which implements selectors and adaptive selectors. First, the requirement for on-going performance monitoring of reference to stored tuple occurrences was discussed. While this type of operational data is often routinely collected, in the case of simple adaptive selectors it is absolutely essential for constructing and parameterizing the Bloom filter storage structure.

Next, a proposal was advanced for the periodic relocation of tuple occurrences. This proposal called for periodically reorganizing the tuple occurrences of base relation extensions in order to cluster the high frequency of reference tuples into a relatively small number of storage blocks. The justification for this proposal was that by performing such relocations, the overall operational performance of the database would be improved by reducing the input/output load on the system.

The last operational consideration concerned the proposal of a new database buffer management policy. The Least Recently and Frequently Used (LFRU) policy was presented and contrasted with the most commonly employed buffer management policy, the Least Recently Used (LRU) policy, as well as the known (non-look-ahead) optimum policy. The results of a simulation study were offered as an indication of the prospective operational improvements which could be gained by utilizing this policy.

CHAPTER 6

CONCLUSION

6.1 Summary of the Research

The main theme of this work has centered on the concept of the occurrence dimension in the design of a database. The recognition of this dimension in the process of database design offers two principal advantages:

1. it enhances the ability of a database designer to capture and represent certain semantic information requirements which are not otherwise representable in the traditional two-dimensional design process; and

2. it offers the potential to significantly increase the operational performance characteristics of the database as well as to extend its operational lifetime.

In order to introduce the occurrence dimension and examine its implications in the process of database design, an integrated three level database design methodology was employed. This particular methodology builds upon existing research and practice in database design and then extends the process by incorporating the occurrence dimension.

The first level of the proposed design methodology is concerned with the art of abstract data modeling. This preliminary design phase consists of identifying the

relevant objects about which an enterprise collects and stores data and how these objects are to be organized so as to convey some of their meaning or interpretation within the enterprise. The Entity Relationship Model [CHEN76] along with the semantic modeling constructs of Smith and Smith [SMIT77a, SMIT77b] and Codd [CODD79], was reviewed as the basis for formalizing the abstract design process.

This review presented a summary of the state-of-the-art in abstract database design. Emphasis was placed on representing the meaning and interpretation of data at the expense of implementation details. It was then argued that there exist significant opportunities to capture and represent additional meaning within an abstract data model if the notion of occurrences of data objects is conceptualized at this level.

At first, the idea of considering occurrences in an abstract data model appears at variance with the nature of the abstraction process. That is, abstraction implies the suppression of detail while the consideration of data occurrences would ordinarily imply the inclusion of considerable detail. However, several specific instances were identified where it was essential to explicitly recognize the existence of occurrences of data objects in order to adequately capture their full meaning.

While the traditional diagrammatic portrayal of an abstract database design is rendered in two dimensions,

the art of abstract data modeling was then extended into a third modeling dimension - the occurrence dimension. Two new database abstractions, selectors and adaptive selectors, were introduced to provide mechanisms to formally represent semantic information requirements in this dimension.

The ability to model the semantics of an enterprise's information requirements in the occurrence dimension would be of little or no value unless there exist corresponding facilities at the lower levels of the database design methodology. The next phase of the three level design methodology involves the mapping of an abstract data model onto the structures supported by a generic data model. The generic data model chosen was the relational model of data. In order to accommodate any selectors or adaptive selectors employed in the abstract data modeling phase, the notion of "artificial" relations was introduced. These artificial relations serve as surrogates for the selectors and adaptive selectors in this transformation process.

Once artificial relations have been defined, there must then be a facility for their formal specification in a data definition language for some implementation of the relational data model. A syntax for a relational data definition language facility was proposed using the PASCAL programming language as a host. Each type of selector and adaptive selector was given a data type specification as

well as facility for defining artificial relation extensions based on that type.

Using this data definition language facility, a relational schema may then be developed for a database design which incorporates selectors and adaptive selectors. To complete the generic data model level of the methodology, a set of extended relational algebra operators was presented. These operators enable the manipulation of the artificial relation extensions by an application program or ad hoc query language such as SEQUEL.

The last level of the methodology concerns the internal, physical details of implementing the relational schema with selectors and adaptive selectors in some particular operating environment. Specifically, recommendations were made for providing storage structure support for selectors and adaptive selectors. Additionally, several proposals were advanced which concern the on-going management of an operational database.

6.2 Contributions

The single, most important contribution of this work lies in the recognition of the occurrence dimension in the art and practice of database design. First, by being able to conceptualize occurrences of data objects in the art of abstract data modeling, significantly more of the semantic

meaning of those data objects can be explicitly represented in the preliminary abstract design.

In order to demonstrate the nature and importance of the occurrence dimension, an example of a hypothetical college database design problem was presented. This design problem, while obviously over simplified, was constructed so as to motivate the kinds of situations which would give rise to the need for the occurrence dimension.

Four quite natural semantic rules were included in the information requirements for the college database which could not be explicitly represented in a two-dimensional abstract data model. The reason is that these semantic rules concern properties of the occurrences of the college's data objects and traditional abstract data modeling suppresses the notion of occurrences. To address this problem, the selector and adaptive selector database abstractions were introduced. These database abstractions are defined exclusively in the occurrence dimension.

The selector abstraction was presented in two forms - the simple selector and the relationship selector. Each form was defined as a partitioning of the occurrences of a given entity based on a boolean qualification of one of its attribute values. The definition of a simple selector on an entity does not affect its participation in any relationships. Rather, any relationships involving the entity are assumed to be potentially valid for all entity occurrences regardless of their selection. With a

relationship selector, however, membership in the selected subset is a necessary precondition for an entity occurrence to participate in certain relationships.

It was then shown how these two forms of selector abstraction could be used to adequately capture and represent two of the given semantic rules which were otherwise not representable. Although the examples were taken from the context of the sample design problem, they are representative of a large class of similar situations which occur frequently in database design.

Similarly, two forms of adaptive selector were presented - the simple adaptive selector and the relationship adaptive selector. In each form of adaptive selector, the membership of the selected subset was based on a temporal criterion. The simple adaptive selector is intended to identify a temporally active subset of the occurrences of an entity. The membership criterion is predicated on the observed recency and frequency of reference to the entity occurrences. In the relationship adaptive selector, membership is based on the entity occurrence's participation in a stated relationship at any given time.

These two forms of adaptive selector abstraction were also demonstrated in the context of the sample design problem. They were shown to be capable of adequately representing the remaining two semantic rules. Again, although the examples were contrived, it is evident that

the use of adaptive selectors enhances the ability of a database designer to capture semantically meaningful aspects of enterprise's information requirements in an abstract data model.

With the art of abstract data modeling extended into the occurrence dimension, it was then necessary to map this modeling environment into the structures, operators, and constraints of a generic data model, in this case, the relational model of data. Because the selector and adaptive selector database abstractions do not create any new entities or relationships within a particular database design, they would have no direct representational form as base relations. To address this problem, the concept of artificial relations was introduced. The artificial relations are intended to serve as surrogates for any selectors or adaptive selectors present in the abstract data model.

A complete relational schema definition corresponding to an abstract data model was then defined as a collection of base relations along with any necessary artificial relations. These artificial relations effectively represent the occurrence dimension in the relational model of data. In order to demonstrate the practicality of incorporating artificial relations, and consequently selectors and adaptive selectors, into the relational model of data, a relational data definition language facility was defined.

The programming language PASCAL was chosen as the vehicle for describing the proposed data definition language facility. This choice was motivated by similar approach taken by Smith and Smith [SMIT77a, SMIT77b]. The syntax of the data definition language was completely defined including data typing facilities for both selectors and adaptive selectors. The extensional counterparts of these data types were given in terms of PASCAL variable definitions. In particular, the variable definition of adaptive selectors was based on the abstract data type of the powerset [HOAR72].

In addition to providing data definition language capabilities for representing the occurrence dimension within the relational model of data, it was also necessary to provide data manipulation language facilities to operate on the selectors and adaptive selectors. For this purpose, five new operators were proposed for the relational algebra. Each of these operators was carefully defined and related to the original five primitive operators which constitute the basis for the relational algebra.

To demonstrate their usefulness, a series of sample database queries was proposed. Using the stated assumptions regarding the number of data occurrences in the sample database design problem, the queries were first formulated in an extended version of the query language SEQUEL which implements the five new operators. Each query

was then formulated without the appropriate artificial relation and extended operators. It was shown that significant performance gains could be realized when the artificial relations and extended operators were employed. These gains were measured in terms of the number of tuple occurrences which would have to be retrieved from secondary storage in order to satisfy the query request. Also, it was shown in several of the queries that the formulation with the extended operators was more concise and more easily understandable.

The definition and subsequent manipulation of the artificial relations which represent the occurrence dimension in the relational model of data presupposed that they will have some form of physical representation. It was assumed, in the context of the sample database design problem, that each base relation extension would be stored as a separate flat file. A hierarchic dense block index storage structure was then associated with each base relation in order to provide a primary access path to the tuple occurrences.

It was suggested that artificial relations be implemented as separate storage structures for the associated base relations. For each type of artificial relation (selector or adaptive selector), a recommended storage structure was given. In the case of simple selectors, relationship selectors, and relationship adaptive selectors, the hierarchic dense index was the

recommended choice. Arguments in support of this choice were given based upon the relative efficiency, simplicity, and flexibility of this storage structure. Additionally, it was shown that this recommended storage structure would be quite suitable for supporting the extended relational operators in responding to queries.

For the simple adaptive selector, the Bloom filter storage structure [BLO070] was recommended. This choice was predicated on the unique requirements of the simple adaptive selector, specifically the ability to rapidly adapt to changing patterns of usage and to efficiently support the set membership operator. A discussion of the Bloom filter paramenter selection problem as it relates to simple adaptive selectors was also given.

Lastly, several requirements for the operational support and maintenance of these storage structures were reviewed. Among these, a specific contribution was made in the proposal for a new DBMS buffer management policy. This policy (LFRU) offers the potential to greatly reduce the number of buffer faults which would be experienced during the operation of a database. In support of this claim, the results of a series of simulation experiments were presented which demonstrate the potential gains. Although this policy would be most effective in the presence of adaptive selectors, its usefulness extends to any operational DBMS environment where on-going performance monitoring would provide the necessary information.

6.3 Further Research

The recognition of the occurrence dimension in data modeling has created new opportunities for the representation of semantics in the art of database design. The introduction and definition of the selector and adaptive selector database abstractions represent but two possibilities for capturing and implementing semantically meaningful aspects of the real world as they might arise in a database design. There are undoubtedly other such abstractions which are still to be discovered.

One such possibility could be in the area of distributed databases. While the occurrences of an entity may be viewed globally as a set of homogeneous data objects, each occurrence has a location characteristic indicating the site at which it is physically stored. A form of database abstraction could be used to represent this aspect of the entity throughout the design process. Further examination of the nature of the occurrence dimension should lead to a much greater understanding of the process of data modeling.

While the selector and adaptive selector abstractions have been rather narrowly defined, there also exists the possibility of expanding upon their definitions. More general criteria could be used in the formulation of the predicates used to define selectors thereby extending their applicability. Similarly, recent work in the area of

representing the semantics of time in databases [CLIFF83] is quite interesting. The concept of historical databases intersects somewhat with the role of adaptive selectors and may prove to be a fruitful extension of both ideas.

Lastly, the possibility of implementing these ideas in an actual database management system should be undertaken. The true potential of the research presented here will be better understood when a database management system incorporating selectors and adaptive selectors is available for experimentation and observation.

APPENDIX

APPENDIX

SIMULATION MODEL ASSUMPTIONS

The simulation model used to derive the results contained in Table 5.2 was based on the Independent Reference Model [COFF73]. The simulated database is assumed to consist of a finite number of tuple occurrences organized into blocks which constitute the unit of transfer between levels of memory. Associated with each tuple occurrence is a known, stationary probability of reference. The simulation then involves generating a sequence of independent and identically distributed references to the stored tuples. The figure of merit is the observed rate of buffer faults obtained for a given buffer management policy.

The Independent Reference Model has been criticized for the assumption of independent requests to tuples. However, Easton [EAST75] and Fagin and Easton [FAGI76] have validated the Independent Reference Model against trace data taken from an operational (IMS/VS) database. A possible rationalization for the success of their validation experiments may lie in the observation that a DBMS services many users simultaneously and when their request streams are merged, the result is an apparently independent sequence.

Assume that the simulated database contains a total of N tuple occurrences numbered sequentially

(i=1,2,...,N). These tuples are then randomly assigned to one of B = N/n blocks where there are n tuples per block. The assignment of tuples to blocks was accomplished by using a simple hashing algorithm. The simulated reference stream consists of sequence of i.i.d. random variables denoted:

$$\{X_1, X_2, \ldots, X_j, \ldots, X_M\}$$

The probability of a reference to the i-th stored tuple at reference X_j is given in the following mass function:

$$Pr[\, X_j = i \,] = [\, i^k - (i-1)^k \,] \;/\; N^k$$

The parameter k determines the skewness of the mass function. The values of k=0.8 (nearly uniform), k=0.5 (modestly skewed), and k=0.1386 (the 80/20 Rule) were used here.

To generate a simulated sequence of references to the database, the probability integral transform method was used. By first generating a uniform random deviate, U, the ordinal number of the corresponding reference is obtained from:

$$i = N\, U^{1/k}$$

Each random tuple reference, X_j, is then transformed to the appropriate block reference by again using the hashing function.

The observed buffer fault rate for a sequence of M such random references is determined as the ratio of the number of faults occurring relative to the number of block switchings. A block switching occurs at reference j+1 when the block referenced by X_{j+1} is not the same as the block referenced by X_j.

The reason for using block switchings as the unit of discrete "time" is to avoid the unnecessary consideration of rereferencing blocks consecutively. Easton [EAST75] found that fault rates simulated over "real time" tended to be quite high when validating them against actual data. By ignoring consecutive references to the same block, he was able to validate his model.

For each stored tuple, two quantities were maintained. One was the block number to which it was assigned and the other was a reference count field. In the Least Recently Used (LRU) buffer management policy, the reference count field was not used because this information is not used in the policy. In the A0 policy, the reference count field contained the actual (constant) probability of reference because this assumed to be known in this policy. Lastly, in the Least Frequently and Recently Used (LFRU) policy, the field was incremented during the simulation to build up an empirical reference distribution which is used in calculating the residual life expectancy.

The actual simulation experiments consisted of storing a total of N = 20,000 tuple occurrences in B = 2,000 blocks. The database buffer region in processor memory was capable of holding at most 20 blocks (or 1%) of the database. A simulated reference stream of M = 100,000 references was then generated.

The transient period until the buffer was first filled was very short and was not considered to significantly contaminate the observed results. In the case of the LFRU policy, however, the transient period did have an effect because the time to develop a reasonably accurate empirical reference distribution was fairly long. If a warm-up period were allowed to enable the empirical reference distribution to begin to stabilize before recording buffer faults, then the empirical distribution would rapidly approach the true distribution and the LFRU policy would approximate the A0 (optimum) policy. It was decided not to allow a warm-period so as not to bias the results in favor of the proposed LFRU policy. The results for the LFRU policy shown in Table 5.2 then underestimate its true performance.

Each experiment corresponded to a choice of the skewness parameter k and one of the three buffer management policies resulting in a total of nine experiments. The model itself was coded as a PASCAL program.

BIBLIOGRAPHY

BIBLIOGRAPHY

[AGHI82] Aghili, Houtan and D.G. Severance, "A practical guide to the design of differential files for recovery of on-line databases," ACM TODS, Vol. 7, No. 4, December 1982, pp. 540-565.

[ASTR76] Astrahan, M.M., et al., "System R: relational approach to database management," ACM TODS, Vol. 1, No. 2, June 1976, pp. 97-137.

[BLOO70] Bloom, B.H., "Space/time trade-offs in hash coding with allowable errors," Comm. ACM, Vol. 13, No. 7, July 1970, pp.422-426.

[CHEN76] Chen, P.P-S., "The entity relationship model - toward a unified view of data," ACM TODS, Vol. 1, No. 1, March 1976, pp. 9-36.

[CLIF83] Clifford, James and D.F. Warren, "Formal semantics of time in databases," ACM TODS, Vol. 8, No. 2, June 1983, pp. 214-254.

[CODD70] Codd, E.F., "A relational model of data for large shared data banks," Comm. ACM, Vol. 13, No. 6, June 1970, pp. 377-387.

[CODD79] Codd, E.F., "Extending the database relational model to capture more meaning," ACM TODS, Vol. 4, No. 4, December 1979, pp. 397-434.

[COFF73] Coffman, E.G. and P.J. Denning, Operating systems theory, Prentice-Hall, 1973.

[DATE81] Date, C.J., An introduction to database systems, 3rd Edition, Addison-Wesley, 1981.

[DATE83] Date, C.J., An introduction to database systems, Volume II, Addison-Wesley, 1983.

[DENN67] Denning, P.J., "The working set model for programming behavior," Comm. ACM, Vol. 11, No. 5, May 1967, pp. 323-333.

[DENN68] Denning, P.J., Y.C. Chen, and G.S. Schedler, A model of program behavior under demand paging, IBM T.J. Watson Research Center, RC2301, September 1968.

[DENN70] Denning, P.J., "Virtual memory," ACM Computing Surveys, Vol. 2, No. 3, September 1970, pp. 153-189.

[EAST75] Easton, M.C., "Model for interactive data base reference string," IBM J. of Res. Dev., Vol. 19, No. 6, November 1975, pp. 550-556.

[FAGI76] Fagin, Ronald and M.C. Easton, "The independence of miss ratio on page size," J. of ACM, Vol. 23, No. 1, January 1976, pp. 128-146.

[GIBS74] Gibson, C.F. and R.L. Nolan, "Managing the four stages of EDP growth," Harvard Business Review, Vol. 52, No. 1, January-February 1974, pp. 76-88.

[GREM82] Gremillion, L.L., "Designing a bloom filter for differential file access," Comm. ACM, Vol. 25, No. 9, September 1982, pp. 600-604.

[HAER78] Haerder, Theo, "Implementing a generalized access path structure for a relational database system," ACM TODS, Vol. 3, No. 3, September 1978, pp. 285-298.

[HAMM78] Hammer, M.M. and D.J. McLeod, "Semantic integrity in a relational data base system," Proc. 1st VLDB, September 1975, pp. 25-47.

[HEIS63] Heising, W.P., IBM Systems Journal, Vol. 2, No. 2, 1963, pp. 114-115.

[HELD78] Held, Gerald and Michael Stonebreaker, "B-trees re-examined," Comm. ACM, Vol. 21, No. 2, February 1978, pp. 139-143.

[HOAR72] Hoare, C.A.R., "Notes on data structuring," in Dahl, O.-J., E.W. Dijkstra, and C.A.R. Hoare, Structured programming, Academic Press, 1972.

[JENS76] Jensen, K. and N. Wirth, PASCAL user manual and report, 2nd Edition, Springer-Verlag, 1976.

[KENT78] Kent, William, Data and reality, North-Holland, 1978.

[KING71] King , W.F., III, "Analysis of paging algorithms," Proc. IFIP Conference, August 1971, pp. 485-490.

[KNUT69] Knuth, D.E., The art of computer programming, Volume 2: Seminumerical Algorithms, Addison-Wesley, 1969.

[KNUT73] Knuth, D.E., The art of computer programming, Volume 3: Sorting and Searching, Addison-Wesley, 1973.

[MART75] Martin, James, Computer Data-Base Organization, Prentice-Hall, 1975.

[NOLA79] Nolan, R.L., "Managing the crises in data processing," Harvard Business Review, Vol. 57, No. 2, March-April 1979, pp. 115-126.

[SEVE76] Severance, D.G. and G.M. Lohman, "Differential files: their application to the maintenance of large databases," ACM TODS, Vol.1, No. 3, September 1976, pp. 256-267.

[SMIT77a] Smith, J.M. and D.C.P. Smith, "Database abstractions: aggregation," Comm. ACM, Vol. 20, No. 6, June 1977, pp.405-413.

[SMIT77b] Smith, J.M. and D.C.P. Smith, "Database abstractions: aggregation and generalization," ACM TODS, Vol. 2, No. 2, June 1977, pp. 105-133.

[SMIT78] Smith, J.M., "A normal form for abstract syntax," Proc. 4th VLDB Conference, September 1982, pp.156-162.

[TEOR80] Teorey, T.J. and J.P. Fry, "The logical record access approach to database design," ACM Computing Surveys, Vol. 12, No. 2, June 1980, pp. 179-212.

[TSIC76] Tsichritzis, D.C., "LSL: a link and selector language," Proc. ACM SIGMOD Conference, 1976, pp. 123-133.

[TSIC82] Tsichritzis, D.C. and F.H. Lochovsky, Data Models, Prentice-Hall, 1982.

[ULLM80] Ullman, J.D., Principles of database systems, Computer Science Press, 1980.