1992

Utilizing extended entity relationship modeling as a basis for logical relational database design

Randall E. Wambold
Lehigh University

AUTHOR:

Wambold, Randall E.

TITLE:

Utilizing Extended Entity Relationship Modeling as a Basis for Logical Relational Database Design

DATE: May 31, 1992

Utilizing Extended Entity Relationship Modeling as a Basis for Logical Relational Database Design

by

Randall E. Wambold

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Computer Science

Lehigh University

March 1992

This thesis is accepted and approved in partial fulfillment of the requirements for the Master of Science.

April 6, 1992
Date

Professor Donald Hillman
Thesis Advisor

Chairman of the Department

Table of Contents

Figures

Abstract

Extended Entity Relationship Modeling is an accepted standard method used during the analysis and design phases of system construction. It encompasses two activities. First, the methodology is used to create a semantic, or conceptual, model that provides an accurate representation of the information requirements necessary to satisfy a particular application arena. Then, second, the finalized model, which is composed of a conceptual diagram and supplemental data, is transformed into an initial logical relational database design that represents the meaning of the original model in a form compatible with, and understood by, the Relational Database Management System.

This thesis discusses a representative Extended Entity Relationship Modeling approach, including both the specification of a conceptual model and its associated database design derivation process. Design issues raised by the use of the Extended Entity Relationship Modeling technique are also examined. My intent, then, is to demonstrate the effectiveness of the basic technique. First, I show how the technique can be used to create an effective conceptual representation of a target domain. Second, I demonstrate how the completeness, flexibility and ease of understanding and use of the initial conceptual model can be preserved and transferred to the logical design. And, finally, I explore

the expected performance characteristics of the default logical relational database design derived from the Extended ER model, and how those characteristics might be improved by extending the derived design.

Chapter 1 Evaluating the Quality of a Database Design

1.1 Introduction

For the purpose of this thesis, I find it important to first discuss the objectives and features of a superior database design. The primary objective in creating a database is to store information that supports the efficient performance of functions. Databases are designed to facilitate reasoning and action. Descriptions of real world objects are organized and held by the structures of a database. The structures and their associated rules for manipulation provide a data model designed to address a problem area.

The features discussed here seek to ensure that a given database design can satisfy the problem objectives for which it was originally created. The design features that must be assessed can be divided into two sets. [1] The first set can be termed Operational Design Features which must be satisfied for the design to be functionally sufficient. If the operational features are incompletely satisfied, then the functions supported by the database will only partially be fulfilled. The second set can be termed Usability Design Features. These quality features assess the ease of use and the adaptability of the design.

In the discussion that follows, I do not draw conclusions about the absolute value of each design feature explored. Rather, I seek only to list those features that

should constantly be reviewed as a design evolves. The real importance of each factor will depend on the problem's objectives. I use many terms freely: conceptual design, database design, application design, etc. The interchangeability of these terms indicates the applicability of these general design feature assessments to all forms of the database design - from its conceptual foundation in a schema to its specific relational database definitions.

1.2 Operational Design Features

1.2.1 Intrinsicality and Completeness

These design features seek to verify that the basic components of the conceptual schema (entities, relationships, attributes - much more will be said about these later) are indeed relevant to the functions that the database will facilitate. Often as an application is designed, its original objectives evolve. As the objectives evolve, the components required to support those objectives evolve also. These design features verify that the final conceptual components match the final application objectives.

The intrinsicality feature requires that each conceptual component can be directly mapped to some function within the application. Strict adherence to this design requirement will cause the elimination of any component that does not support a function of the database. The unnecessary

inclusion of these components will introduce additional complexity into the design which may compromise its effectiveness. Each entity, relationship or attribute type must be measured against the set of system functions. (This type of test indicates the need for finely defined function definitions.)

The completeness feature is the opposite requirement of intrinsicality. With this test we establish that all of the conceptual components necessary to achieve the known functions of the database are indeed contained in the conceptual schema. While the completeness of the conceptual schema, as compared to the set of database functions known at a particular point in time, can be established, the degree of a database's completeness tends to vary over time. That is, a database's completeness tends to deteriorate as the functional use of that database expands over time.

1.2.2 Reliability

Given a set of conceptual components for a database, the reliability design feature assesses the ability of the database users, specifically those who gather the root information for entry into the database, to identify the data in the world. That is, is the required information available? "Availability" introduces many reliability factors. First, an appropriate and reliable source for the data must be found. Second, the specificity or granularity

of the data must be sufficient to meet the requirements of the application functions. And, third, the data must be available in a timely manner. If the reliability feature cannot be met, the conceptual design is flawed, and the ultimate functions of the database cannot be attained.

1.2.3 Representability and Resolution

While the reliability test specifies the quality of the information sources required to provide the basic data to populate a database, the representability feature applies further quality tests to that information. Chiefly, the representability tests assess whether the root data can be unambiguously mapped to the conceptual components of the database. Does the root data clearly identify the logical entities, relationships and attributes, or are the logical targets not precisely evident? [2] Additional data evaluation rules may be required to answer questions of representability. For example, when the root data shows that an employee has equal assignments to two projects, but the database requires the identification of each person's primary "assigned to" project, then the root data's representability is lacking. (That is, two equal assignments do not convey an apparent primary assignment, so we must require the data source to indicate which is the primary assignment, or provide a scheme where the required information can be

correctly determined.)

Resolution is the database-oriented design feature that is complimentary to the representability feature. On occasion, the existence of representability questions may indicate a flaw in the basic conceptual design. Resolution seeks to verify that each conceptual component of the database design is appropriately distinguishable from the other components. Entity, Relationship and Attribute names that are not semantically distinct, should be reviewed to see if the same root data is really sought. For example, the attribute Address and the entity Location seem suspiciously similar. Or, a database that contains both Transportation_Mode and Vehicle_Type entity types may have poor resolution if it needs to represent a real world object, such as my car. Additionally, indistinct attributes are quite common, occurring wherever more than one fact qualifies as an attribute value. So, in a marketing database, where the client entity type has an attribute entitled "Product_Interest", the fact(s) that I am interested in both their skiing and tennis products creates a situation where the available root data cannot be properly represented within the database as it is designed.

1.2.4 Consistency

Consistency is a design feature that measures and ensures the internal consistency of the conceptual model.

With this feature we seek to demonstrate that the definition of each component is consistent with the definitions of all other related components. By example, this design check would require the postal address of the Person entity to share a like definition with the postal address within the Business entity. Less obvious consistency checks, such as the use of standardized abbreviations for attribute values (such as Unit_of_Measure or Degree_Type), could also be enforced.

1.2.5 Normalization

The normalization feature [3] provides quality checks to verify that the conceptual components are indeed structured in a manner that is consistent with relational theory. Adhering to the set of normalization rules reduces redundancy in the database, and in turn removes the potential for database inconsistencies.

The accepted normal forms, and their rules for construction, extend common-sense notions. The first normal form verifies that each conceptual attribute takes only atomic values, i.e., that these values cannot be decomposed. The higher level normal forms verify the interactions or dependencies of the attributes in the database. In second normal form, we verify that every non-key attribute really does depend on the key attribute that it has been associated

with (this normal form establishes functional dependency). In third normal form, functional dependencies between non-key attributes are uncovered and rooted out (in this case, full functional dependency is established in each conceptual entity - usually through the introduction of new entities). Additional normal forms are defined which uncover other instances of dependency - such as transitive and multivalued dependency - and provide prescriptions to modify the conceptual components in a manner that allows the full representation of these dependencies in the database and is relationally sound.

Adhering to these rules of normalization, creates a high degree of data integrity in the database. [4] Update anomalies, where in an unnormalized database the change of a single real world fact might be represented in several database value changes, cannot occur. Likewise, insertion anomalies, where a new real world fact must be recorded in multiple database locations, will not occur. And, deletion anomalies, which are the inverse of insertion anomalies, will be prevented. Again, the removal of these anomalies is only accomplishable because of the systematic removal of data redundancy.

1.3 Usability Design Features

These design features cannot be attained with the same certainty associated with operational design features.

Usability features tend to be quality measurements that interact among themselves. Each cannot be maximized because there are tradeoffs between the features. Realizing that there are tradeoffs, it is necessary to establish the relative importance of each feature when seeking to satisfy it.

1.3.1 Flexibility and Clarity

The flexibility and clarity design features assess the ability of the conceptual schema components to adapt to changes in the real world that the database seeks to represent.

The quality of the flexibility feature can be explored by theorizing hypothetical changes in the problem domain and assessing the ease by which the database design can accommodate the change. Ideally, the database design should easily adapt to likely evolutions in the problem domain. For instance, an airline's aircraft maintenance application that does not anticipate (i.e., easily accommodate) the introduction of new aircraft types, such as a Boeing 797, clearly would receive low marks for flexibility. When evaluating database design flexibility, the designers must weigh both the likelihood and frequency of each real world change with the corresponding modifications in the database to accommodate that change.

The clarity, or preciseness of definition, associated

with each database component is another aspect of flexibility. On one hand, database users and designers seek clear and unambiguous definitions of database components. This enhances their ease of use, in that the appropriate components to be used in the development of information and actions is easily and correctly identified. Yet, complete precision of definition can bring with it a penalty. Complete clarity reduces a database object's flexibility or adaptability. Open-ended or vague database object definitions allow application users to exercise judgment and thus extend or continue a given database design.

1.3.2 Efficiency

The efficiency quality design feature seeks to determine the performance characteristics of a database design. This must be both a measurement of the expected database performance as well as the real performance requirements of the application users. The functional objectives of the application determine the database performance requirements. [5]

The quality of the efficiency of a database design can be construed in several ways. In one way, efficiency can be determined by assessing the success with which data redundancy has been minimized. (In chapter 6, I will discuss the use of redundant data - often in the form of summarized data - as an efficient method of addressing the needs of some

application functions.) In general, we should consider a less redundant database design as the more efficient design. In another way, efficiency may be assessed by determining the number of entity types, relationship types, and attribute types needed to support the functions of the database application. From this perspective, a flexible design will normally appear much less efficient than a less flexible design. That is, a conceptual schema with many components will perform less well (due to the increased physical actions required to access the data contained in those many components) than when compared to a conceptual schema that stores the same data in far fewer components. Additionally, the efficiency of a design can be significantly affected by some detail design decisions. For example, a given fact may be represented in the conceptual schema as either an attribute of one entity type, or as an independent entity, with its own series of attributes, associated with a root entity. In general, it is much more efficient to access an attribute rather than an associated entity.

1.3.3 Semantic Integrity

The semantic integrity quality feature seeks to assess a database's ability to support the creation of meaningful and useful inferences. That is, this quality feature seeks to verify that the results of using this database applica-

tion do indeed provide sufficient information for its users to reason and take action.

Meaningful inferences can only be provided if the data contained within the database is appropriately specific. That is, the conceptual components of the database - the entity types, relationship types and attribute types - should not be so broad (read, flexible) as to obscure their semantic meaning and thus undermine their usefulness. As the design process progresses, it must be assured that the conceptual schema continues to satisfy the absolute functional requirements of the database application. Coupled with this general requirement for "specificity", the domain definitions for each attribute type within the schema must be expressive and specific enough to capture the required real world meaning of the attribute. (I use "required" to differentiate the following. We cannot expect to capture the full real world meaning of an attribute in a simple database structure, but we must pursue the capturing of the necessary meaning - where "necessary" indicates the attribute domain specificity required to produce useful inferences from the database.)

Chapter 2 Semantic Modeling

2.1 Introduction

The process of semantic modeling [6] attempts, first, to identify that set of semantic concepts that are essential and useful when speaking of some real world problem domain. While these semantic concepts are not precise, their creation is the essential first step in the process of creating a database design. Given the identification of these concepts, the next step is to devise a set of symbolic objects, such as entity types and relationship types, that can be used as the building blocks for the creation of a formal model. Coupled with the development of these symbolic objects, a set of complimentary integrity rules, that in a formal way guide the manipulation of these objects, must also be developed.

2.2 The Analysis and Design Context

Extended Entity Relationship Modeling (ERM) is just one tool that is used during the analysis and design phases of an application project. Other complementary tools include Process Modeling and Function Definition. [7]

Process Modeling defines the movement and transformation of data through functions within a business or operation. Often dataflow diagrams are used to record this process information. They record the sources of inputs and the destinations of outputs. Dataflows indicate the trans-

fer of data through processes and between data stores.

Function definitions describe the elementary operational processes that must be implemented to achieve the objectives of the application. The documentation for each function would include: the triggering action that invokes the function, a frequency of activity, a detailed description of the function's logic, a list of data entities that are consumed / produced / or modified by the function and a list of related functions that may be invoked.

Information that is retained over time (as opposed to that which is generated and consumed within a function) for later use is what we seek to define with Extended ER modeling.

Process models and function definitions, alone, are ineffective as a basis for the design of databases. These techniques model data transformations, but a database is a designed representation of data. While I will not define these tools beyond the brief description that I have provided above (because they are not the focus of this paper), the reader should note that their development is an essential related task to the development of the conceptual schema.

2.3 Extended Entity Relationship Modeling (ERM)

ERM is a popular implementation of semantic modeling. ERM utilizes semantic objects such as entities, attributes,

relationships, supertypes and subtypes. ERM seeks to categorize all information elements into these semantic conceptual categories. [8] Tightly integrated with this modeling scheme is a corresponding diagraming technique. Thus the objects of a given model can be easily represented in a concise, but effective manner (in terms of its ability to convey the existence and interrelationships of all objects within the diagram). Supplemental documentation, consisting primarily of integrity rule information, is combined with the semantic object data to create a complete semantic model (extended data model). This complete semantic model is also referred to as the conceptual model.

The modeling technique discussed in the following pages is my aggregation of many distinct techniques. [9] I feel that this technique permits the development of a flexible, and yet precise, semantic model of a conceptual database.

2.4 Semantic Concepts

2.4.1 Entities

An entity is any object about which we need to hold information and which contributes to the satisfying of our application's objectives. [10] An entity can be a real person, place, object or event, or it can be a concept or activity. It must, in all cases, be significant to the objective. Each specific instance of an entity must be uniquely identifiable / distinguishable from all other

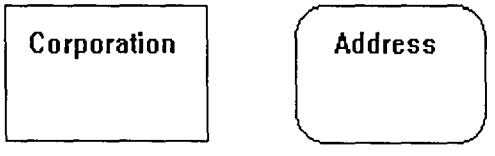
occurrences of the entity type. Entities can be classified as regular and weak entities. Regular entities have the characteristic that they are capable of existing independently from all other entities. Weak entities are dependent upon some other entity, and never exist outside of a relationship with this other entity. For example, an employee's emergency contacts would be weak entities, in that they should not exist if the relevant employee entity instance does not exist.

A regular entity is represented diagrammatically by a rectangle containing a capitalized entity name. A weak entity is shown diagrammatically within a softbox.

Figure 2-1 shows two example entities. The entity type labeled Corporation is a regular entity. And, the entity type labeled Address is a weak entity.

While I have included distinct graphical representations for regular and weak entity types, strictly speaking this is not necessary. It is the characteristics of the relationships between one entity and other entities that truly define its class. I retain the two graphical representations as a simple method of highlighting each entity type's characteristics to the users of the database.

Figure 2-1: Regular and Weak Entities



2.4.2 Relationships

A relationship represents a significant association between two entity types. [11] Each distinct relationship type has two properties (besides its name). Cardinality (e.g., one-to-one, one-to-many or many-to-many) defines the number of possible participant entity instances that can occur in a relationship. And, optionality defines the nature of the participation for each entity type. That is, if each and every entity instance of a particular entity type must participate in this relationship, then its optionality is mandatory, else it is termed optional. The identification of an entity type's participation as mandatory implies the invoking of an additional rule during the insertion of each instance of the entity. This rule requires that an instance of the specified relationship type be created simultaneously with the creation of each root entity instance. Corresponding update and delete constraints must be applied to retain at least one relationship instance for each entity instance in the mandatory relationship.

Diagrammatically, relationships can be shown in two ways. In most cases, a relationship is shown by two line segments and a diamond connecting the rectangles or softboxes of two entities. (This diagrammatic structure is used to associate all entities in any relationship except the special relationship case where a weak entity's existence dependence on another entity is being described.) The name

of the relationship is contained in the diamond. The cardinality of each entity type at the end points of the relationship is noted by the values of "1" (One) and "M" (Many). Regarding optionality, a single line segment between the rectangle of an entity type and the diamond of a relationship indicates optional participation. A double line segment indicates the entity type's mandatory participation in the relationship.

In the special case where a weak entity's existence dependence is being described, the diamond shape connector is replaced by a triangle. In this case, the base of the triangle faces the "subordinate" entity and the apex of the triangle points to the "superior" entity. (I use the terms "subordinate" and "superior" to permit the case where both entities are weak entities, and one weak entity controls the existence of another.)

Figure 2-2 illustrates the graphical representation of an optional many-to-many relationship between the regular entity types, Part and Vendor. The relationship type is named Quote. The relationship is optional because some instances of Part may exist for which we have no Quotes, and some instances of Vendor may exist which have not yet provided a Quote. The relationship is termed many-to-many because a single Part may have Quotes from many Vendors, and a single Vendor may Quote for many Parts.

Figure 2-3 represents an Existence Dependent Relationship. The weak entity Branch_Location is dependent on the existence of a relevant regular entity instance of Business_Unit. The relationship is labeled, Located_At / Contained_In (more will be said about the use of two names later), and the relationship's cardinality is one-to-many. That is, a single Business_Unit may be "Located_At" many Branch_Locations, but a single Branch_Location may be "Contained_In" only one Business_Unit. Each Business_Unit's participation in the relationship is optional, but each Branch_Location's participation is mandatory. The one-to-many cardinality and mandatory participation by the dependent entity are standard characteristics of an Existence Dependent relationship.

While I have included a distinct graphical representation of Existence Dependent Relationship types, strictly speaking this is not necessary. These relationships could be described as any other relationship. It is the relationship type's characteristics (the one-to-many cardinality and the mandatory participation of the "many" entity) that really designate a given relationship as an Existence Dependent relationship. I retain the triangular graphical representation to highlight this type of relationship to the users of the database.

Figure 2-4 illustrates a recursive relationship. The regular entity, Part, can optionally participate in a relationship with other instances of its type. The many-to-many cardinality indicates that each part may be "Composed_Of" many component parts, and that each part (simultaneously) may be "Used_In" the construction of many higher-level Parts, i.e., assemblies.

Figure 2-5 illustrates the General Relationship Representation. The solid and dashed lines indicate entity participation - either optional (a single solid line segment) or mandatory (a double solid line segment). The 1/M indicates the cardinality of each entity type in the relationship. I have provided two names for the single relationship type shown in the example. This is a common construct that highlights the bidirectional nature of a relationship. The graphic representation of each relationship can be described in a formal syntax, as follows:

Relationship Formal Syntax:

```
                                (must be)
   a) Each and every ENTITY-A <         > relation_name_1
                                (may be )

        (one and only one              ever)
        <                  ENTITY-B        > .
        (one or more                 plural)
```

and,

```
                                (must be)
   b) Each and every ENTITY-B <         > relation_name_2
                                (may be )

        (one and only one              ever)
        <                  ENTITY-A        > .
        (one or more                 plural)
```

The use of this formal syntax is helpful in verifying the structure of the diagram by allowing its contents to be verbalized. The relationship in figure 2-3 could be verbalized as follows:

a) Each and every Business_Unit may be Located_At one or more Branch_Locations.

and,

b) Each and every Branch_Location must be Contained_In one and only one Business_Unit ever.

Figure 2-2: Example Many-to-Many Relationship

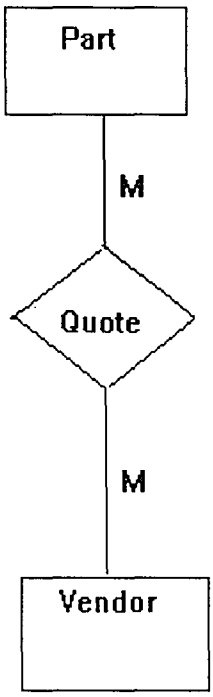


Figure 2-3: Example Existence Dependence Relationship

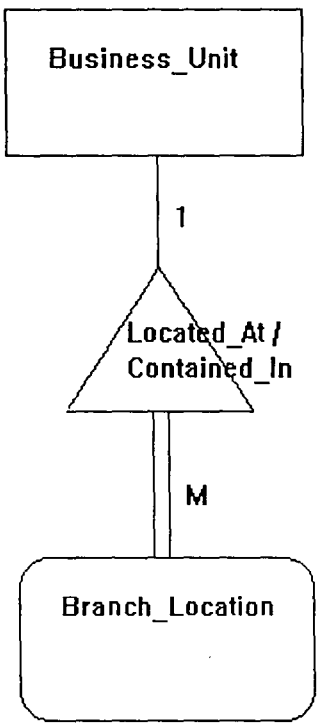


Figure 2-4: Example Recursive Relationship

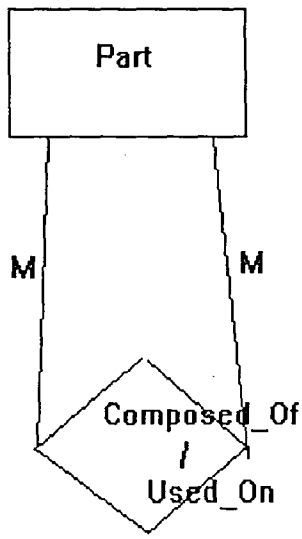
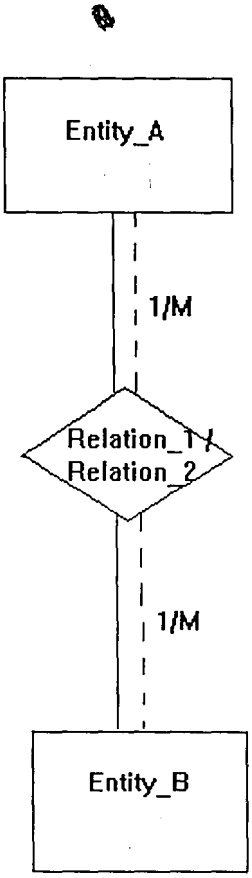


Figure 2-5: General Relationship Representation



2.4.3 Attributes

The specific information that can be stored for each entity type or relationship type is referred to as its attributes. Attributes provide the details that describe the significant aspects of an entity or relationship. [11] Attributes identify, classify, quantify or describe the state of an entity or relationship. Additionally, attributes may be key or non-key elements. An attribute that is a key element can serve as the unique identifier for the entity type. Non-key attributes do not uniquely identify an entity instance or relationship. Attributes may also be designated as optional or mandatory. The value of a mandatory attribute must be known when the entity or relationship instance is first created. And, although the values of mandatory elements can be modified, they cannot simply be removed. Only optional attributes may ever be assigned an "unknown" or "null" value. Finally, it is important to note that a relationship type may be defined without associating any attributes with it.

Diagrammatically, attributes are often shown as named ellipses attached to the rectangles/softboxes of entity types or the diamonds/triangles of relationship types. Alternately, attributes may be represented by placing their names in lower case within the graphic symbol of the owning object. For purposes of this thesis, attributes will normally not be listed or will be listed in a supplemental

document to the ER diagram. When listed, attributes that serve as the primary key will have a "#" prefix, and mandatory attributes will have a "*" prefix.

Figure 2-6 details attribute information that could be associated with the simple conceptual schema of figure 2-2. Of special interest is the attribute information associated with the relationship Quote. Note that it contains the primary keys of the related entity types, that is, Part_Number and Vendor_Number of entities Part and Vendor respectively.

Figure 2-6: Attribute Information for Figure 2-2

Entity: Part

Attributes: # * Part_Number
* Part_Name
Raw_Material_Spec
ECN_Number
Quality_Control_Level

Entity: Vendor

Attributes: # * Vendor_Number
* Vendor_Name
* Quality_Rating
Address
Standard_Terms

Relationship: Quote

Attributes: # * Quotation_Number
* Part_Number
* Vendor_Number
* Price
Special_Terms

2.4.4 Supertypes and Subtypes

A given entity type may have instances that are recognizable as distinct groups within the larger entity type. The larger entity type is referred to as a supertype, and the subset groups are referred to as subtypes. For example, a Person entity type may be divided into two subtypes: Applicant and Employee. Subtypes must be mutually exclusive.

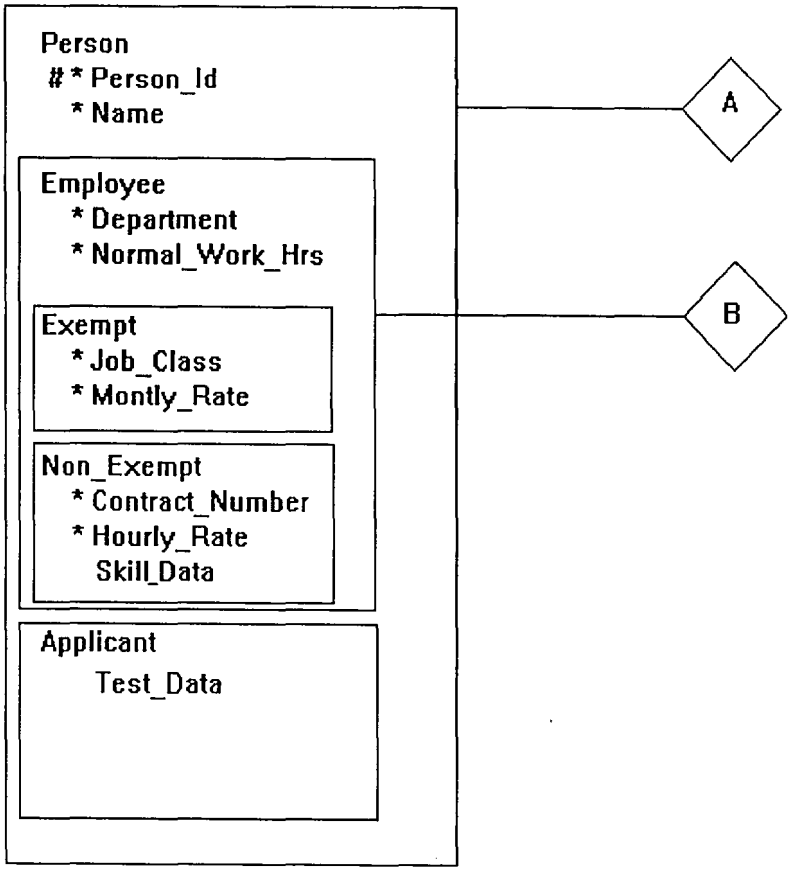
Each supertype and subtype may have associated attributes and relationship types. The attributes and relationship types of the supertype are inherited by its subtypes. That is, each subtype logically contains its supertype's attributes and logically participates in all of the relationships in which the supertype participates. Note that the converse is not true - the attributes and the relationship types of the subtype do not apply to the supertype. Note also that each entity subtype is a legitimate entity type, which permits it to be the supertype over other subordinate subtypes.

Diagrammatically, subtypes are represented as inner boxes (either rectangles or softboxes) contained within an outer box representing the supertype entity.

Figure 2-7 illustrates several supertypes and subtypes. The primary supertype is labeled Person. It has two mandatory attributes of which Person_Id is the primary key. All Person instances may participate in the relationship labeled

A. The supertype Person has subtypes, Employee and Applicant. As subtypes of the Person entity, they share its attributes (and unique identifier) and its relationship A. The subtype Employee also functions as a supertype, and has two subtypes of its own, labeled Exempt and Non_Exempt. The attributes of Employee are logically contained in its subtypes. Additionally, the relationship in which all Employee instances may participate, labeled B, is logically associated with its subtypes. That is, both Exempt and Non_Exempt subtype entities may participate in relationship B. However, the Applicant subtype (the other subtype entity of Person) cannot participate in relationship B, as that relationship is only associated with the Employee subtype, as shown by the relationship's direct connection to the Employee entity.

Figure 2-7: Supertypes and Subtypes



2.4.5 Primary Keys and Foreign Keys

Earlier, I referred to attributes as either key or non-key elements. In this section, I'll look more closely at keys.

The primary key of an entity type or relationship type is that attribute (or those attributes) that uniquely identifies each instance of the entity or relationship. [12] The attribute(s) chosen to be the primary key must uniquely identify every instance of the entity or relationship type. Additionally, if the primary key is a composite key, it should be minimal. That is, the primary key should be reduced to the set of attributes such that if any one attribute were removed, the uniqueness quality would be disrupted.

It is possible that a given entity type or relationship type may have more than one unique identifier (composed of one or more attributes). These are termed candidate keys. When multiple candidate keys exist, one (normally the best known, most commonly used, or easiest to determine) is chosen as the primary key. The other candidate keys then remain as alternate keys. That is, they function as unique identifiers, but not as "the" unique identifier for this entity or relationship type that is known within the remainder of the conceptual model.

While primary keys are normally formed from attributes associated with the entity or relationship in question, they

may also be formed by inheriting attributes from other entities or relationships. For instance, if a specific relationship type does not possess its own internal unique identifier, a unique identifier may be constructed by concatenating the primary keys that identify the two entity types in the relationship. Or, a weak entity may inherit part of its primary key from the superior entity upon which it depends, via the relationship between the two entities.

The concept of a foreign key was mentioned earlier. A foreign key can be defined as the inclusion of attribute(s) in one object that fully identify the primary key of another object. [13] For purposes of this semantic modeling technique (i.e., the ERM technique that I am describing), I will temporarily restrict the definition of foreign keys by requiring that they occur only in relationship types, and that the foreign keys only identify the primary keys of entity types. In this way, a reference from one entity type to another must be implemented through an intervening relationship type. Therefore, a (binary) relationship will always contain two foreign keys which reference the primary keys of the two entity types participating in the relationship. (As an extension, a ternary relationship will contain three foreign key definitions.) The definition of the foreign keys in a relationship must exactly correspond to the definitions of the primary keys in the referenced enti-

ties. It is this correspondence of foreign keys to primary keys (and vice versa) that provides the referential power of the resulting database.

Please note that the implementation of an actual relational database would allow the direct referencing of one entity type by another entity type, i.e., without the specification of an intervening relationship. While this is permissible, it requires one of two special relationship cases to exist between the two entities. These special cases will be discussed more fully in chapter 4. For now, the restriction I noted in the previous paragraph allows me to more easily describe the modeling technique.

2.4.6 Integrity Rules

Two categories of integrity rules exist. [14] First, the Entity Integrity rule states that the attribute (or attributes) defined as the primary key for an entity or relationship type must not accept null values. That is, it may never have an unknown value. Because this rule applies to all entity and relationship types, in all cases, it is unnecessary to represent this rule in the conceptual model. It simply applies in all cases.

The second integrity rule is the Referential Integrity rule. It states that the instances of objects within a database must never contain an unmatched foreign key value. A foreign key value is unmatched when there does not exist

an instance of the target entity with a matching primary key value. This integrity rule is primarily enforced whenever attributes identified as foreign key identifiers are modified within the database, either as new object instances are created or as current ones are modified. The general intent of this rule is to control the modification of the database in such a way as to always ensure the full referential integrity of the database.

To achieve the full intent of the general rule, two additional detail questions must be evaluated for every foreign key defined within the conceptual schema. The answers to these questions supplement the diagrams and attribute lists, discussed above, to yield a full conceptual schema. The answers are used to provide direction to database modification routines in such a way as to guarantee that the final result of any modification activity against a "referenced" entity is a referentially consistent database. This is achieved by prefixing audit routines and appending supplemental modification operations to each root database modification request.

The first question asks what is the appropriate action when the target instance of a foreign key, contained in an entity, is identified for deletion. That is, for example, what should be done if the particular Course instance, identified by the value of the foreign key, Course_Id, in a

particular Student_Registration relationship instance, is targeted for deletion? Two basic options exist:

a) "Restricted": In this case, the deletion of the entity instance is only permitted if no foreign key references, specified with the Restricted option, contain the value specified for deletion.

b) "Cascades": In this case, the deletion of the specified instance will be permitted, but its deletion will cause the deletion of all instances of the referencing relationships which contain a matching foreign key reference to the target instance, and have specified the Cascades option.

The description, above, is only partially complete. In reality, the "tests", associated with the potential deletion of an entity instance where its primary key is referenced as a foreign key in a relationship, must be more comprehensive. In one case, a single entity may be referenced by many relationship types, each with their own particular answer to the question. Therefore, the deletion of the target instance can occur only if the rules of all referencing relationships can be simultaneously and consistently satisfied, else, the root deletion operation must fail. In a second case, additional referential integrity logic must be invoked because the root deletion request for one entity instance might generate additional entity instance deletions in the other entity type participating in the relationship when its

participation is mandatory. Thus, through the specification of the Cascades option, an additional set of referential integrity checks must be invoked, each assuming the other entity instances identified through the application of the Cascade option as a new deletion target. Therefore, this integrity rule should be considered to apply recursively.

The second question, asks what is the appropriate action when the target instance of a foreign key, contained in an entity or relationship, is designated for update. That is, for example, what should be done if the particular Course instance - currently Id number 100, and specified as a Course_Id in a Student_Registration instance - is identified to have its Id number altered to 101? Again, as above, two basic options exist:

a) "Restricted": In this case, the update of the specified primary key is only permitted if no foreign key references, specified with the Restricted option, contain the value that will be replaced by the update operation.

b) "Cascades": In this case, the update of the specified primary key will be permitted, but the update effect will cascade to all instances of the relationship which contain a matching foreign key reference to the target instance, and have specified the Cascades option. In this way, the prior references remain intact and referentially consistent.

The description, above, is only partially complete. In reality, the "tests", associated with the potential update of an instance where its primary key is referenced as a foreign key, must be more comprehensive. That is, a single entity may be referenced by many relationship types, each with their own particular answer to the question. Therefore, the update of the target instance can occur only if the rules of all referencing relationships can be simultaneously and consistently satisfied, else, the root update operation must fail. Often the update of a foreign key value, caused by the application of a Cascades option, will not have a rippling effect beyond the first level of reference in the associated relationship types. But, a true rippling effect does occur when the updated foreign key participates as a component in the referencing instance's composite primary key. (In this case, the update of a primary key value may cascade through several levels of "reference".) Therefore, this integrity rule should be considered to apply recursively.

The answers to the prior questions (each associated with a foreign key definition) establish the necessary integrity rules to control the update or deletion of a referenced primary key value. These rules can be labeled as a Delete rule and an Update rule. Inserts to the referenced entity type need no special integrity checks, that is, they cannot create a referentially inconsistent database.

Additionally, inserts, deletes and updates to the instances of the referencing relationship type (containing the foreign key) need no special referential integrity checks beyond the basic enforcement of the Referential Integrity rule. They do, however, require that the optionality properties of the entity types participating in the relationship under modification not be disrupted.

To conclude, each foreign key defined within the conceptual schema must include selections, either Restricts or Cascades, to establish the Delete and Update rules.

Chapter 3 Project Management - Extended ER Model Example

Figures 3-1 and 3-2 provide a complete model of a simple Project Management system. The system's basic objectives are: 1) to provide a definition of a project, in terms of its tasks, 2) to detail the assignment of persons and material resources to a project, and 3) to enable the billing of clients for work or material expended on their projects.

The diagram in figure 3-1 shows the entity and relationship types of the system. The following description expresses the semantic content of the diagram. Projects are Composed_Of Tasks (and, Tasks can exist only in a relationship with a Project). Projects are Owned_By Clients (and again, as above, Projects are existence dependent on Clients). Persons exist as a supertype, with Employee and Contractor subtypes. But only an Employee Manages a Project, Contractors cannot. Any Person, both Employees and Contractors, can be assigned (through a Person_Assignment relationship) to a Task. The supertype Resource has two subtypes, Durable and Consumable, of which only a Durable Resource is worthy or capable of assignment (through a Durable_Assignment relationship). The final entity to be discussed is the weak entity, Actual_Charge. It is existence dependent, via the Billed_To relationship, on the entity Task - meaning that within this database it is not possible to record charge data without immediately identify-

ing the Task to be charged. Additionally, when an Actual_Charge is created, it must immediately participate in one of two mandatory relationships, either the Person_Charge relationship or the Resource_Charge relationship. (The exclusive nature of these relationships is recorded on the diagram by the single hash mark crossing the line segments that connect the Actual_Charge entity to the two relationships. If a single entity participated in multiple exclusive relationships, they would be differentiated by paired hash marks, e.g., the first exclusive relationships could be denoted with a single hash mark, the second set of exclusive relationships could be denoted with a double hash mark, etc.)

Figure 3-2 contains the supplemental information associated with the Extended ER diagram of figure 3-1. Specifically, it contains Attribute information (Names, key and mandatory characteristics, and format data) for all Entity and Relationship types, and the associated Integrity constraints.

Figure 3-1: Project Management Extended ER Diagram

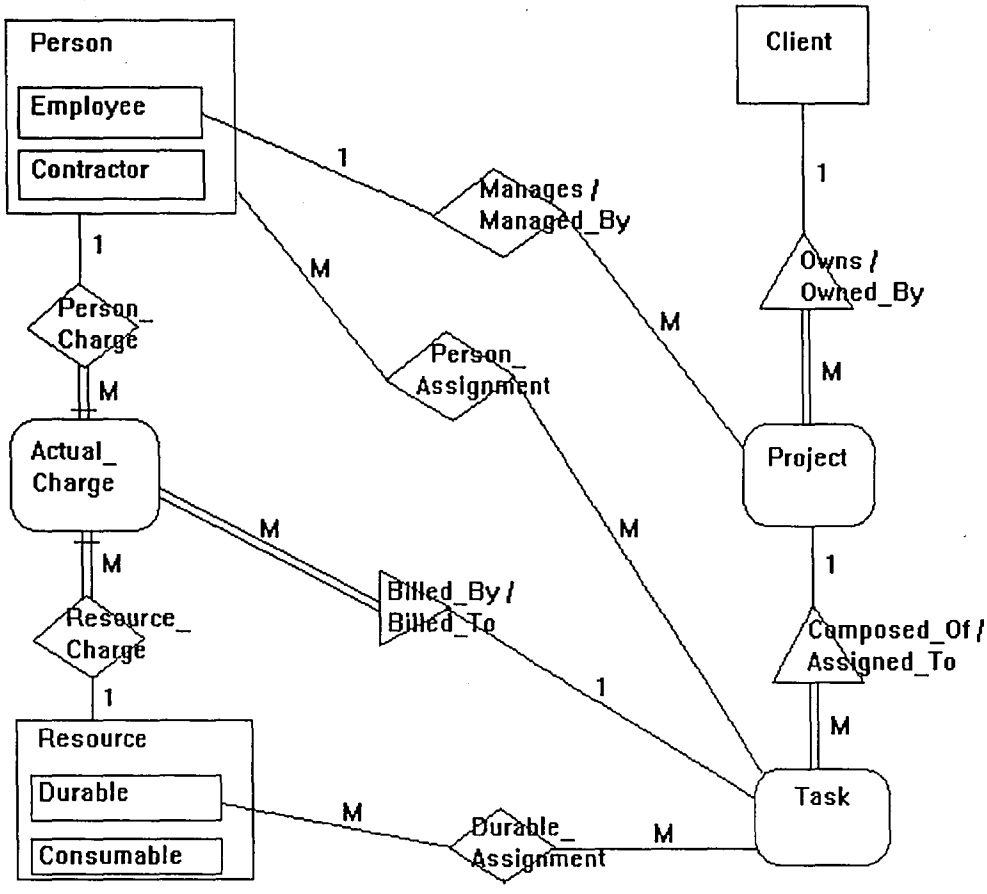


Figure 3-2: Project Management Extended ER Supplemental Data

Entity: Person

Attributes: # * Person_Id Numeric (6)

* Name Alpha (40)

Charge_Rate_Wk Numeric (4,2)

Entity: Employee

Attributes: * Social_Sec_Nbr Numeric (9)

* Hire_Date Date

* Pay_Rate_Wk Numeric (4,2)

Entity: Contractor

Attributes: * Contractor_Id Numeric (9)

* Fed_Emplr_Id Numeric (9)

Contract_Start Date

Contract_Stop Date

Contract_Rate_Mo Numeric (5,2)

Entity: Resource

Attributes: # * Resource_Id Numeric (4)

* Resource_Title Alpha (40)

Charge_Rate_Unit_Of_Meas Alpha (4)

Resource_Class Alpha (4)

Entity: Durable

Attributes: * Quantity Numeric (5)

Use_Restrictions Alphanum (60)

Entity: Consumable

Attributes: * Replenish_Days Numeric (3)

Entity: Actual_Charge

Attributes: # * Actual_Charge_Id Numeric (7)

* Actual_Charge_Date Date

* Actual_Charge_Rate Numeric (5,2)

* Actual_Charge_Unit_Of_Meas Alpha (4)

* Actual_Charge_Quantity Numeric (3,2)

Actual_Charge_Start_Date Date

Actual_Charge_Stop_Date Date

Entity: Client

Attributes:

#	*	Client_Id	Numeric (9)
	*	Client_Name	Alpha (40)
		Contact_Name	Alpha (40)
		Contact_Title	Alpha (50)
		Contact_Addr_Line_1	Alphanum (50)
		Contact_Addr_Line_2	Alphanum (50)
		Contact_Phone	Numeric (10)
		Client_Rating	Alphanum (2)

Entity: Project

Attributes:

#	*	Project_Id	Numeric (5)
	*	Project_Name	Alpha (40)
		Start_Date	Date
		Want_Date	Date
		Promise_Date	Date

Entity: Task

Attributes:

#	*	Project_Id	Numeric (5)
#	*	Task_Id	Numeric (3)
	*	Task_Name	Alpha (40)
		Start_Date	Date
		Stop_Date	Date

Relationship: Manages / Managed_By

Attributes: # * Person_Id Numeric (6)

* Project_Id Numeric (5)

Integrity Rules: Foreign Key Person_Id

References Employee.Person_Id

Delete of Employee Cascades

Update of Employee.Person_Id

Cascades

Foreign Key Project_Id

References Project.Project_Id

Delete of Project Cascades

Update Of Project.Project_Id

Cascades

Note: Because this relationship is a one-to-many relationship, it could be implemented as a foreign key residing directly in the entity referenced by the "Many" foreign key, i.e., the Project entity type. More will be said about this in the next chapter of the paper.

Relationship: Person_Assignment

Attributes: # * Person_Id Numeric (6)

* Project_Id Numeric (5)

* Task_Id Numeric (3)

Start_Date Date

Stop_Date Date

Integrity Rules: Foreign Key Person_Id

References Person.Person_Id

Delete of Person Cascades

Update of Person.Person_Id Cascades

Foreign Key Project_Id, Task_Id

References Task.Project_Id,

Task.Task_Id

Delete of Task Restricted

Update Of Task.Project_Id,

Task.Task_Id Cascades

Relationship: Billed_To / Billed_By

Attributes: # * Actual_Charge_Id Numeric (7)

* Project_Id Numeric (5)

* Task_Id Numeric (3)

Integrity Rules: Foreign Key Actual_Charge_Id

References Actual_Charge.

Actual_Charge_Id

Delete of Actual_Charge Cascades

Update of Actual_Charge.

Actual_Charge_Id Cascades

Foreign Key Project_Id, Task_Id

References Task.Project_Id,

Task.Task_Id

Delete of Task Restricted

Update Of Task.Project_Id,

Task.Task_Id Cascades

Relationship: Durable_Assignment

Attributes: # * Resource_Id Numeric (4)

* Project_Id Numeric (5)

* Task_Id Numeric (3)

Start_Date Date

Stop_Date Date

Integrity Rules: Foreign Key Resource_Id

References Durable.Resource_Id

Delete of Durable Cascades

Update of Durable.Resource_Id

Cascades

Foreign Key Project_Id, Task_Id

References Task.Project_Id,

Task.Task_Id

Delete of Task Restricted

Update Of Task.Project_Id,

Task.Task_Id Cascades

Relationship: Owns / Owned_By

Attributes: # * Client_Id Numeric (9)

* Project_Id Numeric (5)

Integrity Rules: Foreign Key Client_Id

References Client.Client_Id

Delete of Client Restricted

Update of Client.Client_Id Cascades

Foreign Key Project_Id

References Project.Project_Id

Delete of Project Restricted

Update Of Project.Project_Id

Cascades

Relationship: Composed_Of / Assigned_To

Attributes: # * Project_Id Numeric (5)

* Task_Id Numeric (3)

Integrity Rules: Foreign Key Project_Id

References Project.Project_Id

Delete of Project Restricted

Update of Project.Project_Id

Cascades

Foreign Key Project_Id, Task_Id

References Task.Project_Id,

Task.Task_Id

Delete of Task Restricted

Update of Task.Project_Id,

Task.Task_Id Cascades

Relationship: Person_Charge

Attributes: # * Person_Id Numeric (6)

* Actual_Charge_Id Numeric (7)

Integrity Rules: Foreign Key Person_Id

References Person.Person_Id

Delete of Person Restricted

Update of Person.Person_Id Cascades

Foreign Key Actual_Charge_Id

References Actual_Charge.

Actual_Charge_Id

Delete of Actual_Charge Cascades

Update Of Actual_Charge.

Actual_Charge_Id Cascades

Relationship: Resource_Charge

Attributes: # * Resource_Id Numeric (4)

* Actual_Charge_Id Numeric (7)

Integrity Rules: Foreign Key Resource_Id

References Resource.Resource_Id

Delete of Resource Restricted

Update of Resource.Resource_Id

Cascades

Foreign Key Actual_Charge_Id

References Actual_Charge.

Actual_Charge_Id

Delete of Actual_Charge Cascades

Update Of Actual_Charge.

Actual_Charge_Id Cascades

Chapter 4 Systematically Deriving a Relational DB Design from an Extended ER Model

4.1 Introduction

A primary benefit of using the Extended ER model as the conceptual specification for an information or application domain is that it can be used as the basis for an automatically generated default relational database design, that is, a logical relational specification of a database. The objective of this process is to produce a set of relation definitions that are capable of satisfying the information requirements defined by the Extended ER model. [15] The automatic generation of a logical design facilitates the rapid creation of a prototype database. The prototype can be used to verify the logical design of the Extended ER model. With a small populated prototype system, sample processes, corresponding to the principal retrieval and update actions of the system can be constructed and tested. As a side-effect, this process will develop early performance data about the base design. Eventually, alternatives to the conceptual design can be considered, or performance-enhancing modifications to the relational design can be implemented.

4.2 The Derivation Process

This section of the paper provides the details of the process that systematically derives a default relational

database design from an Extended ER model. To clarify, the process uses the detailed information provided in the Extended ER model (as described by the Diagram and the Supplemental Data) as its source data, and transforms it into a relational database schema, including definitions of relations (for the purpose of clarity, I will refer to relations as tables in the following discussions), views, indexes, columns and integrity rules. [16]

The Derivation Process:

- Create the Modified Extended ER Model
    - Identify a Subtype Attribute for All Complex Entity Types
    - Transform One-to-One and One-to-Many Relationship Types into Components of the Participating Entities
- Create the Relational Database Schema
    - Transform Simple Entity Types into Tables
    - Transform Complex Entity Types (Supertypes and Subtypes) into Tables and Views
    - Transform Many-to-Many Relationship Types into Tables
    - Transform Attributes into Columns
    - Dèfine Default Indexes
    - Define Default Integrity Constraints

To facilitate the explanation of the derivation process, I have included a complete Example Extended ER Model. It is described in figures 4-1 and 4-2. Its corresponding Modified Extended ER Model is included as figures 4-3 and 4-4. And, finally, the results of the derivation process are included as figure 4-5, Example Derived Relational DB Design.

4.2.1 Create the Modified Extended ER Model

4.2.1.1 Identify a Subtype Attribute for All Complex Entity Types

For a purpose that will become clearer in a later section of this chapter, it is necessary to identify a single attribute within each highest level supertype entity that can be used to distinguish each subtype entity. If such a single attribute does not exist in each of the highest level supertypes, then a new attribute can be invented to satisfy this requirement. This attribute will simply function as a definitive indicator of the lowest level subtype to which each instance of the complex entity may be categorized. This attribute must be designated as mandatory.

4.2.1.2 Transform One-to-One and One-to-Many Relationship Types into Components of the Participating

Entities

Relationship types can be categorized by their cardinality: one-to-one, one-to-many or many-to-many. In this step of the derivation process, all one-to-one and one-to-many relationship types are transformed into attributes and integrity rules associated with one of the entity types in the relationship. This transformation is legitimate because it is possible to guarantee a one-to-one correspondence between each instance of the identified relationship type and each instance of one of the participating entities. That is, because all instances of one of the participating entities will correspond to at most one instance of the relationship type, the values of that relationship type can be transferred to the entity type.

In a one-to-many relationship, the attributes and the integrity rules of the relationship type are transferred to the entity type designated as the "many" participant. In the Example ER model described in figures 4-1 and 4-2, the components of relationship type B are transferred to entity type C and the components of relationship type I are transferred to entity type G. (Note that the attributes of these relationship types that can be equated to the primary key of the "many" participant, simply vanish. That is, it is unnecessary to repeat this value because it already exists in the entity as the primary key attribute(s). It is equal-

ly unnecessary to transfer the foreign key integrity rule for these attribute(s) as this rule is redundant with the constraints standardly associated with a primary key.) The components of a one-to-one relationship type may be transferred to either participating entity type. In general, it is preferable to merge the relationship information into the entity type which has a mandatory membership in the relationship, or with the entity type that has the fewest anticipated occurrences.

When relationship attributes are transferred to an entity, the names of these attributes are normally prefixed with the name of the relationship type. This permits the unique identification of the attributes associated with each particular relationship type when multiple relationships exist between the same two entity types. In some cases, the attribute to be integrated into the entity will already be present. This is the case with relationship type B. An A_Key attribute already exists within entity type C as part of its concatenated primary key. It is, therefore, necessary only to transfer the integrity rule for A_Key to entity type C.

When the attributes of a relationship type are appended to those of an entity type, they are initially declared as optional attributes. Subsequent to this, all attributes associated with the root relationship type are declared as a unit. This unit identification is necessary so that the

relational database management system (RDBMS) can control the existence and modification of these attributes as a group. For example, the integrity of the database would be compromised if the non-foreign key attributes were added to the database in the absence of the associated foreign key value. Within this unit declaration, the optional / mandatory nature of all non-foreign key attributes are specified. This specification applies only when the described relationship actually exists for an instance of the entity type. That is, a mandatory attribute will only be required when the relationship actually exists, and it will be required to be null (i.e., not exist) when its associated relationship does not exist.

When the integrity rule of a relationship type's foreign key is appended to the supplemental data for an entity type, it must be modified and reviewed in the following manner. First, a new clause must be specified to describe the participation of the entity type in the relationship. "Nulls Allowed" is specified if the entity type's participation is optional, and "Nulls Not Allowed" is specified if the participation is mandatory. This specification defines whether or not the foreign key attribute(s) contained within the unit may ever contain nulls. And second, the value of the Delete clause within the integrity rule must be audited. If the value specified for the Delete clause is "Cascades",

it must be set to "Nullifies". "Cascades" instructs the RDBMS to remove the instance of a relationship type when the referenced entity has been identified for deletion. Altering this value to "Nullifies" causes the RDBMS to logically remove this instance of the relationship when the referenced entity has been identified for deletion by "nullifying" all of the attributes in the unit representing the relationship, this accomplishes the directed deletion. A Delete clause value of "Restricted" should remain unaltered.

The components of relationship types with a many-to-many cardinality cannot be transferred to a participating entity type. They must stand on their own, and remain designated as a relationship type. Relationship type H is such a relationship. These relationships will be considered in section 4.2.2.3.

4.2.1.3 Conclusion

Figure 4-3 shows the Example Modified Extended ER Diagram associated with figure 4-1. They are identical except that two relationship types, B and I, in figure 4-3 are shown with "dashed" symbols. This notes that their components have been integrated into one of the relationship's participating entities. Figure 4-4 details the Example Modified Extended ER Supplemental Data. The major changes from figure 4-2 include the integration of relationship types B and I components into entity types C and G, and

the addition of a new mandatory attribute in entity D, D_Subtype.

4.2.2 Create the Relational Database Schema

4.2.2.1 Transform Simple Entity Types into Tables

Each simple entity type can be translated into a single table. By simple entity type, I mean any entity that is not a supertype and/or subtype (these are complex entities and require further examination). A simple entity type, whether regular or weak, can be transformed in this direct manner. In figure 4-3, regular entity type A and weak entity type C are eligible for this transformation.

4.2.2.2 Transform Complex Entity Types (Supertypes and Subtypes) into Tables and Views

Two basic options exist when defining a default transformation process for complex entity types. The choice of one option over the other is driven by many concerns, primarily, maintaining the basic relational integrity of the database and retaining the semantic clarity and simplicity of the original logical model.

The first option would be to define a separate table for each of the lowest level subtypes, and define views for all higher level supertypes which union the contents of the underlying tables corresponding to the subtypes. In the

example modified diagram shown in figure 4-3, subtypes F, G and J would be transformed into tables, while supertypes D and E would be implemented as views. View E would union the contents of tables F and G, and view D would union the contents of view E and table J.

Certain concerns arise with this implementation method. First, and primary, maintaining the exclusive feature of the subtypes becomes difficult (i.e., I would not expect the RDBMS to standardly implement this type of control). By "exclusive feature of the subtypes", I am referring to the requirement that any value of the unique identifier used for each instance of all subtypes should occur only once in all of the subtypes, that is, the identifier remains unique at the highest supertype level. Because the values would be stored in multiple tables, the implementation of this integrity requirement would be cumbersome if attempted at the application level, and still my strong preference would be to have this very basic control handled directly by the RDBMS. A second problem also surfaces when we consider the relationship types in which the supertypes and subtypes participate. In a later section, we will see that the remaining (many-to-many) relationship types in a model will be transformed into tables. This process should be straightforward. However, when the subject relationship type has a participating member entity which is a supertype, the transformation process becomes quite difficult under

this transformation scheme. In figure 4-3, relationship type H, between entity type C and supertype E, is an excellent example. Using this first transformation process option, relationship type H becomes a relationship between C and F, and between C and G. Should relationship type H be implemented as two tables? Or, should it be implemented as one table with multiple foreign keys referencing two entity types. Under either approach, the clarity of the original model suffers significantly. Due to these concerns, I reject the first option.

The second option would be to define a single table for the highest level supertype, and define views for all subtypes. In the example diagram shown in figure 4-3, supertype D would be transformed into a table and subtypes E, F, G and J would be implemented as views. In this approach, the views would identify subsets of the larger underlying table (as opposed to performing union operations as described in option one, above). For example, view E would contain only the attributes associated directly with its logical specification and those inherited from supertype D, and the instances selected into view E would be governed by a "Where" clause which identifies the instances that make up the distinct subtype. (The subtype attribute identified for each complex entity type would be used to easily segregate the instances of the complex entity into their appropriate

subsets.) In this way, each subtype can be logically implemented as a view of the larger table. Additionally, a view corresponding to the highest level supertype, but containing only those attributes common to it, should also be defined.

This approach allows the RDBMS to overcome my primary concern associated with option one. Because all instances of this complex entity type are contained in a single table, the exclusive feature, or uniqueness requirement for all instances within the complex entity, can be automatically enforced by the RDBMS in an elegant fashion. The specification of a unique primary key for this table achieves the end.

As alluded to earlier, each view will reveal only those attributes, and participate in only those relationship types consistent with the supertype or subtype being described. The use of these views allows us to overcome the relationship problem specified above under option one. When relationships with complex entities must be described, they will, first, be defined within the base (highest level supertype) table as a "restricted" relationship based on the subtype column, and then noted as valid columns within the proper view. In our example, then, relationship type H can be implemented as a relationship between table C and view E of table D. The view E actually restricts the instances of table D to the appropriate subset that can legitimately participate in the relationship. This preserves the sim-

plicity and clarity of the model.

Unfortunately, a new problem arises when this transformation option is selected. Under this option, the enforcement of the mandatory / optional characteristic associated with each attribute becomes more difficult. The transformation of the attributes listed in the model will be more fully discussed in a later section. However, at this point we should just note that the enforcement is defined in the underlying table (in our example, table D) and invoked through the usage of the views. When the highest level supertype is transformed into a table, it must accommodate all subtypes. So, even though there are attributes that are mandatory for one subtype, within the table they must be specified as optional - because they will not occur for all of the other subtypes. Only mandatory attributes of the highest level supertype can have this characteristic enforced in the derived table. The specification of subtype data "units" define the true mandatory / optional characteristics for each attribute and are invoked and enforced when the subtype characteristic of each table instance is assigned or modified. The generated views, then, specify the enforcement of these characteristics through the inclusion of subtype data units, as shown in figure 4-5, Views E, F, G and J.

In summary, the transformation of complex entity types

into a single table with views (option two) is effective in maintaining the integrity of the application system, provides easy access to supertypes and subtypes, and retains the clarity of the model.

4.2.2.3 Transform Many-to-Many Relationship Types into Tables

Each of the remaining relationship types, all of them representing a many-to-many relationship, can be translated into a single table. In figure 4-3, relationship type H would yield a corresponding table. The attributes and integrity rules associated with each of these relationship types will be transferred to the derived table, as shown in figure 4-5.

4.2.2.4 Transform Attributes into Columns

Each attribute associated with an entity type or eligible relationship type is translated into a column. Optional attributes are specified as "null"(able) columns, and mandatory attributes are specified as "not null"(able) columns. (The specification of the "null" / "not null" clause is massaged to accurately reflect the constraints associated with subtype unit attributes and relationship unit attributes.) This specification allows the RDBMS to enforce this constraint. Additionally, other special characteristics such as the identification of the primary key or the

specification of "With Check Option" constraints associated with an attribute are also transferred to the column definitions.

It is important to note that columns are specified for both derived tables and views. Again, see figure 4-5 for an example of this transformation.

4.2.2.5 Define Default Indexes

Three types of default indexes can be generated in the process of transforming the modified model into a relational database schema. The indexes that are generated only apply to tables. Any derived views do not have associated indexes, as they inherit the use of the indexes associated with the underlying (or referenced) table. Remember, also, that these indexes may be based on concatenated columns.

First, the column(s) representing the primary key of each derived table receives an index. The importance of this index is two-fold. First, it facilitates the rapid and direct access of each instance within the table by the RDBMS. This enhances the general performance of the designed database. And, second (if the RDBMS does not do this as a result of identifying the primary key), the index can be specified as "unique" in order to direct the RDBMS to enforce the basic relational integrity of the system, i.e., disallowing the creation of duplicate rows or tuples.

Second, each table derived from a complex entity receives an additional index. This index would identify a concatenated key consisting of the table's subtype-identifying column followed by the column(s) representing the primary key. This default index is included to enhance the general performance of the database. It permits the RDBMS to rapidly access the appropriate subsets of instances that correspond to each derived view.

Third, all foreign keys specified within the tables, derived from the relationship types of the original model, would receive indexes as well. Foreign keys derived from one-to-one relationship types would receive a "unique" specification. This would enforce the one-to-one cardinality of the relationship. Foreign keys derived from one-to-many and many-to-many relationship types would not receive a "unique" specification.

In the example of figures 4-3 and 4-4, tables A, C, D, and H would each receive a unique index over the primary key. Table D would receive two additional indexes. The first index would be based on the subtype-identifying column, D_Subtype, and the second would be based on the foreign key implemented to effect relationship type I. Table H would receive an additional pair of indexes over its two foreign keys. Note that table C would not receive an additional index over the foreign key contained within its concatenated primary key. The index over the primary key

can function as the index over this foreign key.

4.2.2.6 Define Default Integrity Constraints

In the course of executing the transformation processes described above, additional default integrity constraints may be derived. Within the scope of this discussion, an additional constraint would be derived for each subtype-identifying column, limiting it to values which would have a one-to-one correspondence with each of the lowest level subtypes contained within the complex entity type.

4.3 Conclusion and Pseudo-SQL Syntax

The earlier sections of this chapter describe a systematic process for deriving a complete relational database design from an Extended Entity Relationship model. The result of this derivation process is a set of relational database object definitions (such as tables, views, and indexes) that can be processed by the RDBMS to actually implement the database, thus, achieving the objective and benefits of moving directly from the model to a relational specification. In figure 4-5, I have described the results of the process as applied to the Example Extended ER model of figures 4-1 and 4-2. The results utilize a pseudo Structured Query Language (SQL) as the basis for the syntax. This definition syntax has been abbreviated, ignoring any

parameters typically required by an RDBMS that are associated with each object's logical and physical characteristics. (In an actual implementation of this process, additional data such as "estimated instances of the entity/relationship" would have been captured in the model. This, augmented with some default location data, such as a database or tablespace specification, would permit the automatic generation of all characteristics associated with the objects yielding a complete relational definition.)

The basic syntax for the three main database objects is as follows:

```
Create Table <table_name>
 Columns: <column_name> <column_constraint> , ...
 [Integrity Rules: <rule_name> <table_constraint>
          , ...]

Create View <view_name>
 As (<query>)   {i.e., Select ...}

Create [Unique] Index <index_name> On <table_name>
 (<column_name(s)> [ASC|DESC] , ...)
```

Figure 4-1: Example Extended ER Diagram

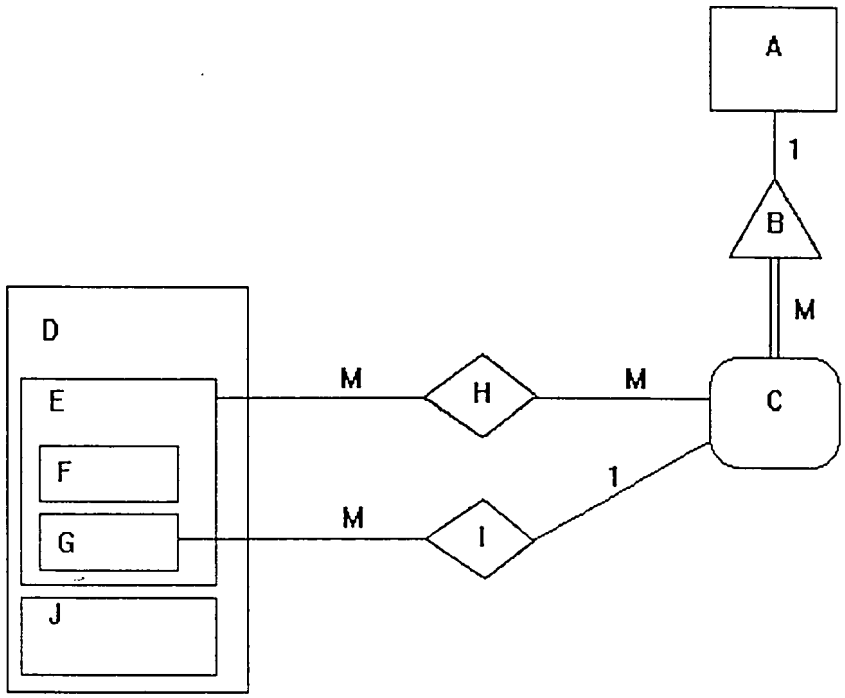


Figure 4-2 Example Extended ER Supplemental Data

```
Entity:  A
     Attributes:  # * A_Key   Numeric (5)
                  * A_Data_1  Alpha (30)
                    A_Data_2  Numeric (7)

Entity:  C
     Attributes:  # * A_Key   Numeric (5)
                  # * C_Key   Numeric (2)
                      C_Data_1  Alpha (20)

Entity:  D
     Attributes:  # * D_Key   Numeric (4)
                  * D_Data_1  Alpha (40)

Entity:  E
     Attributes:    * E_Data_1  Numeric (7)
                    E_Data_2  Alpha (20)

Entity:  F
     Attributes:    * F_Data_1  Alpha (10)
                    F_Data_2  Alpha (30)
```

Entity: G

Attributes:	*	G_Data_1	Numeric (10)
		G_Data_2	Alpha (30)

Entity: J

Attributes:	*	J-Data_1	Alpha (15)
		J_Data_2	Alpha (25)

Relationship: B

Attributes: # * A_Key Numeric (5)

* C_Key Numeric (2)

Integrity Rules: Foreign Key A_Key

References A.A_Key

Delete of A Restricted

Update of A.A_Key Cascades

Foreign Key A_Key, C_Key

References C.A_Key, C.C_Key

Delete of C Cascades

Update of C.A_Key, C.C_Key Cascades

Relationship: H

Attributes: # * H_Key Numeric (6)

* D_Key Numeric (4)

* A_Key Numeric (5)

* C_Key Numeric (2)

* H_Data_1 Alpha (4)

Integrity Rules: Foreign Key D_Key

References E.D_Key

Delete of E Cascades

Update of E.D_Key Cascades

A_Key, C_Key

References C.A_Key, C.C_Key

Delete of C Cascades

Update of C.A_Key, C.C_Key Cascades

Relationship: I

Attributes: # * D_Key Numeric (4)

* A_Key Numeric (5)

* C_Key Numeric (2)

* I_Data_1 Alpha (10)

Foreign Keys: D_Key

References G.D_Key

Delete of G Cascades

Update of G.D_Key Cascades

A_Key, C_Key

References C.A_Key, C.C_Key

Delete of C Cascades

Update of C.A_Key, C.C_Key Cascades

Figure 4-3: Example Modified ER Diagram

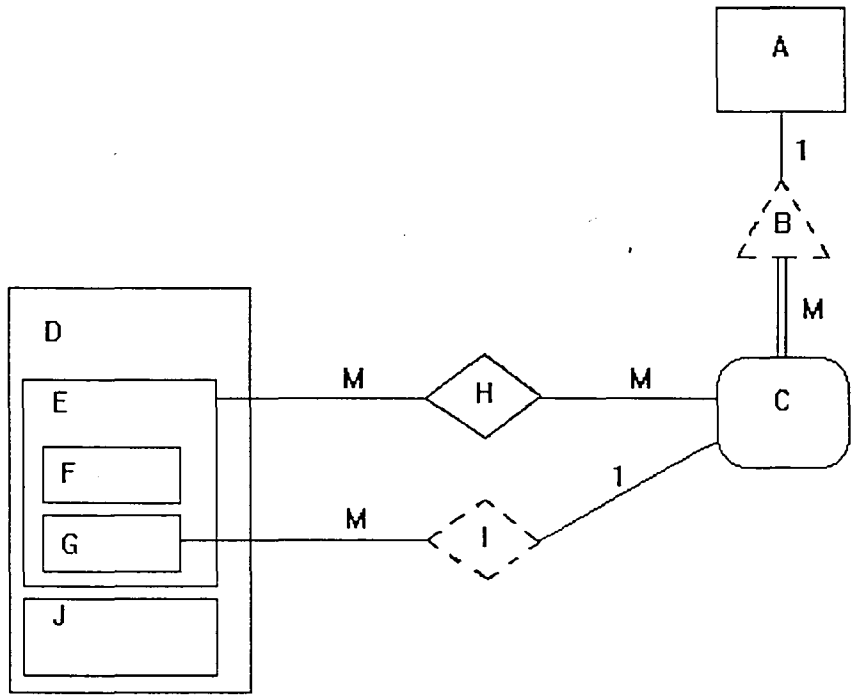


Figure 4-4 Example Modified Extended ER Supplemental Data

```
Entity:  A
     Attributes:  # * A_Key  Numeric (5)
                  * A_Data_1  Alpha (30)
                    A_Data_2  Numeric (7)

Entity:  C
     Attributes:  # * A_Key  Numeric (5)
                  # * C_Key  Numeric (2)
                    C_Data_1  Alpha (20)
     Relationship B Attributes:
                  * A_Key
                    Foreign Key A_Key
     Integrity Rules: Foreign Key A_Key
                      Nulls Not Allowed
                      References A.A_Key
                      Delete of A Restricted
                      Update of A.A_Key Cascades

Entity:  D
     Attributes:  # * D_Key  Numeric (4)
                  * D_Data_1  Alpha (40)
                  * D_Subtype Alpha (1)
```

Entity: E

Attributes: * E_Data_1 Numeric (7)

E_Data_2 Alpha (20)

Entity: F

Attributes: * F_Data_1 Alpha (10)

F_Data_2 Alpha (30)

Entity: G

Attributes: * G_Data_1 Numeric (10)

G_Data_2 Alpha (30)

Rel_I_A_Key Numeric (5)

Rel_I_C_Key Numeric (2)

Rel_I_Data_1 Alpha (10)

Relationship I Attributes:

* Rel_I_A_Key

* Rel_I_C_Key

* Rel_I_Data_1

Foreign Key Rel_I_A_Key, Rel_I_C_Key

Integrity Rules: Foreign Key Rel_I_A_Key, Rel_I_C_Key

Nulls Allowed

References C.A_Key, C.C_Key

Delete of C Nullifies

Update of C.A_Key, C.C_Key Cascades

Entity: J

Attributes: * J-Data_1 Alpha (15)

J_Data_2 Alpha (25)

Relationship: H

Attributes: # * H_Key Numeric (6)

* D_Key Numeric (4)

* A_Key Numeric (5)

* C_Key Numeric (2)

* H_Data_1 Alpha (4)

Integrity Rules: Foreign Key D_Key

References E.D_Key

Delete of E Cascades

Update of E.D_Key Cascades

A_Key, C_Key

References C.A_Key, C.C_Key

Delete of C Cascades

Update of C.A_Key, C.C_Key Cascades

Figure 4-5 Example Derived Relational DB Design

```
Create Table A
  Columns:  A_Key   Numeric (5)   Not Null
            A_Data_1  Alpha (30)   Not Null
            A_Data_2  Numeric (7)   Null
  Integrity Rules:
            Primary_Key = (A_Key)

Create Unique Index A_Primary_Key_Index on A
   (A_Key  ASC)
```

```
Create Table C
  Columns:  A_Key   Numeric (5)   Not Null
            C_Key   Numeric (2)   Not Null
            C_Data_1  Alpha (20)  Null
  Integrity Rules:
            Primary Key = (A_key, C_Key)
            Rel_B Unit = (A_Key  Not Null)  {see note below}
            Rel_B Foreign Key = (A_Key
                  Nulls Not Allowed {see note below}
                  References A.A_Key
                  Delete of A Restricted
                  Update of A.A_Key Cascades)
```

Note: The "Not Null" specification on A_Key within Rel_B Unit signifies that when the unit exists, the column A_key must not be null. The "Nulls Not Allowed" specification within the Rel_B Foreign Key clause specifies that the relationship is mandatory. (Both of these specifications are, in fact, redundant because A_Key is a component of table C's primary key, implying a "not nulls" specification.)

```
Create Unique Index C_Primary_Key_Index on C
  (A_Key, C_Key  ASC)
```

```
Create Table D
  Columns:  D_Key  Numeric (4)  Not Null
            D_Data_1  Alpha (40)  Not Null
            D_Subtype Alpha (1)  Not Null
              With Check Option (D_Subtype = 'F' or 'G' or
                                 'J')
            E_Data_1  Numeric (7)  Null
            E_Data_2  Alpha (20)  Null
            F_Data_1  Alpha (10)  Null
            F_Data_2  Alpha (30)  Null
            G_Data_1  Numeric (10)  Null
            G_Data_2  Alpha (30)  Null
            J-Data_1  Alpha (15)  Null
            J_Data_2  Alpha (25)  Null
            Rel_I_A_Key  Numeric (5)  Null
            Rel_I_C_Key  Numeric (2)  Null
            Rel_I_Data_1  Alpha (10)  Null
  Integrity Rules:
            Primary Key = (D_Key)
            Rel_I Unit = ((Rel_I_A_Key  Not Null
                           Rel_I_C_Key  Not Null
                           Rel_I_Data_1  Not Null)
                           With Check Option = (D_Subtype =
                             'G'))
            Rel_I Foreign Key = ((Rel_I_A_Key, Rel_I_C_Key)
                  Nulls Allowed
```

```
                References (C.A_Key, C.C_Key)
                Delete of C Nullifies Rel_I Unit
                Update of (C.A_Key, C.C_Key) Cascades)
        Subtype_E Unit = ((E_Data_1  Not Null
                           E_Data_2  Null)
                With Check Option = (D_Subtype = 'F' or
                           'G'))
        Subtype_F Unit = ((Subtype_E Unit
                           F_Data_1  Not Null
                           F_Data_2  Null)
                With Check Option = (D_Subtype = 'F'))
        Subtype_G Unit = ((Subtype_E Unit
                           G_Data_1  Not Null
                           G_Data_2  Null)
                With Check Option = (D_Subtype = 'G'))
        Subtype_J Unit = ((J_Data_1  Not Null
                           J_Data_2  Null)
                With Check Option = (D_Subtype = 'J'))

Create Unique Index D_Primary_Key_Index On D
  (D_Key  ASC)

Create Index D_Subtype_Index On D
  (D_Subtype, D_Key  ASC)
```

```
Create Index D_Rel_I_Index On D
  (Rel_I_A_Key, Rel_I_C_Key  ASC)

Create View D_V
  As (Select D_Key
             D_Data_1
             D_Subtype
        From D) (see note below)
```

Note: This view, like other views, inherits all integrity rules associated with the underlying base table. Therefore, while this view can be used to maintain some columns within the table, an attempt to change an instance's subtype from "F" to "E" will be unsuccessful because this view does not allow the specification of the "Rel_I" prefixed columns. Such an update action would generally have to be performed against the underlying table, such as table D, where all columns and integrity rules are available.

```
Create View E
  As (Select D_Key
             D_Data_1
             D_Subtype
             Subtype_E Unit
        From D Where D_Subtype = 'F' Or 'G')

Create View F
  As (Select D_Key
             D_Data_1
             D_Subtype
             Subtype_E Unit
             Subtype_F Unit
        From D Where D_Subtype = 'F')

Create View G
  As (Select D_Key
             D_Data_1
             D_Subtype
             Subtype_E Unit
             Subtype_G Unit
             Rel_I Unit
        From D Where D_Subtype = 'G')
```

```
Create View J
  As (Select D_Key
             D_Data_1
             D_Subtype
             Subtype_J Unit
        From D Where D_Subtype = 'J')
```

```
Create Table H
  Columns:  H_Key    Numeric (6)   Not Null
            D_Key    Numeric (4)   Not Null
            A_Key    Numeric (5)   Not Null
            C_Key    Numeric (2)   Not Null
            H_Data_1 Alpha   (4)   Not Null
  Integrity Rules:
            Primary Key = (H_Key)
            Foreign Key = (D_Key
                  Nulls Not Allowed  (see note 1 below)
                  References E.D_Key  (see note 2 below)
                  Delete of E Cascades
                  Update of E.D_Key Cascades)
            Foreign Key = ((A_Key, C_Key)
                  Nulls Not Allowed (see note 1 below)
                  References (C.A_Key, C.C_Key)
                  Delete of C Cascades
                  Update of (C.A_Key, C.C_Key) Cascades)
```

Note 1: These clauses are derived from the mandatory characteristic associated with the foreign key columns.

Note 2: This foreign key references view E, thus limiting the instances of table D to the appropriate subset.

```
Create Unique Index H_Primary_Key_Index On H
  (H_Key  ASC)

Create Index H_Rel_1_Index On H
  (D_Key  ASC)

Create Index H_Rel_2_Index On H
  (A_Key, C_Key  ASC)
```

Chapter 5 Derived Relational DB Design for the Project Management Example

This chapter presents the results of the derivation process described in chapter 4 when it is applied against a larger model. The source Conceptual model is the one described in chapter 3, figures 3-1 and 3-2.

In the first phase of the derivation process, a Modified Extended ER model is created. Figures 5-1 and 5-2 describe the Modified model for the Project Management example. It differs from the source Conceptual model in two ways. First, new subtype attributes have been added to the Person and Resource supertypes. This permits the easy identification of the subtypes. Second, six of the eight relationship types have been transformed into components of the participating entity types. Relationship types Owns / Owned_By and Manages / Managed_By have been folded into entity type Project. Relationship type Composed_Of / Assigned_To has been added to entity type Task. And, three relationship types: Billed_By / Billed_To, Person_Charge and Resource_Charge, have been defined as attributes within the Actual_Charge entity type. An additional integrity rule (see the note under entity type Actual_Charge within figure 5-2) has been created to enforce the exclusive nature of two of the relationship types.

In the second phase of the derivation process, the Modified Extended ER model is translated into a Relational

Database Schema. Figure 5-3 details the complete relational database design for the Project Management model. Simple entity types: Client, Project, Task and Actual_Charge are transformed into tables. Complex entity types Person and Resource are also transformed into tables. Table Person receives three views: Person_V, which is a view of the supertype, and two views corresponding to the valid subtypes, Employee and Contractor. Table Resource receives three views as well: Resource_V, which is a view of the supertype, and two views corresponding to the valid subtypes, Durable and Consumable. Many-To-Many relationship types, Person_Assignment and Durable_Assignment, are also transformed into tables. This yields a total of eight tables and six views comprising the schema. The required default indexes are defined for each table. First, a unique index on the primary key of each table is specified. Second, each table representing a complex entity type receives an additional index to aid in the access of the identified subsets. In this model, table Person receives the Person_Subtype_Index and table Resource receives the Resource_Subtype_Index. And, finally, each foreign key contained within the tables receives an index to aid in the enforcement of the integrity rules. In the model under examination, table Actual_Charge receives three of these indexes, table Project receives two, and tables

Person_Assignment and Durable_Assignment each receive one. In the case of these last two tables, they each have another foreign key which is a candidate for an index. But, they do not receive an index because these foreign keys, Person_Id and Resource_Id respectively, appear as the first component within each table's primary key, allowing the primary key index to function as the foreign key index as well. (Table Task has a similar situation. Its single foreign key is the first component of its primary key, so a specific foreign key index is not required also.) In all, seventeen default indexes are defined. Regarding integrity rules, all of the rules that were specified within the Conceptual and Modified models are transferred to the tables of the relational database schema. Additionally, I have included a "With Check Option" rule on each complex entity's subtype-identifying column to limit its values to codes that correspond to the defined subtypes.

In summary, figure 5-3 contains the complete set of database objects required to define the Project Management relational database.

Figure 5-1: Project Management Modified Extended ER Diagram

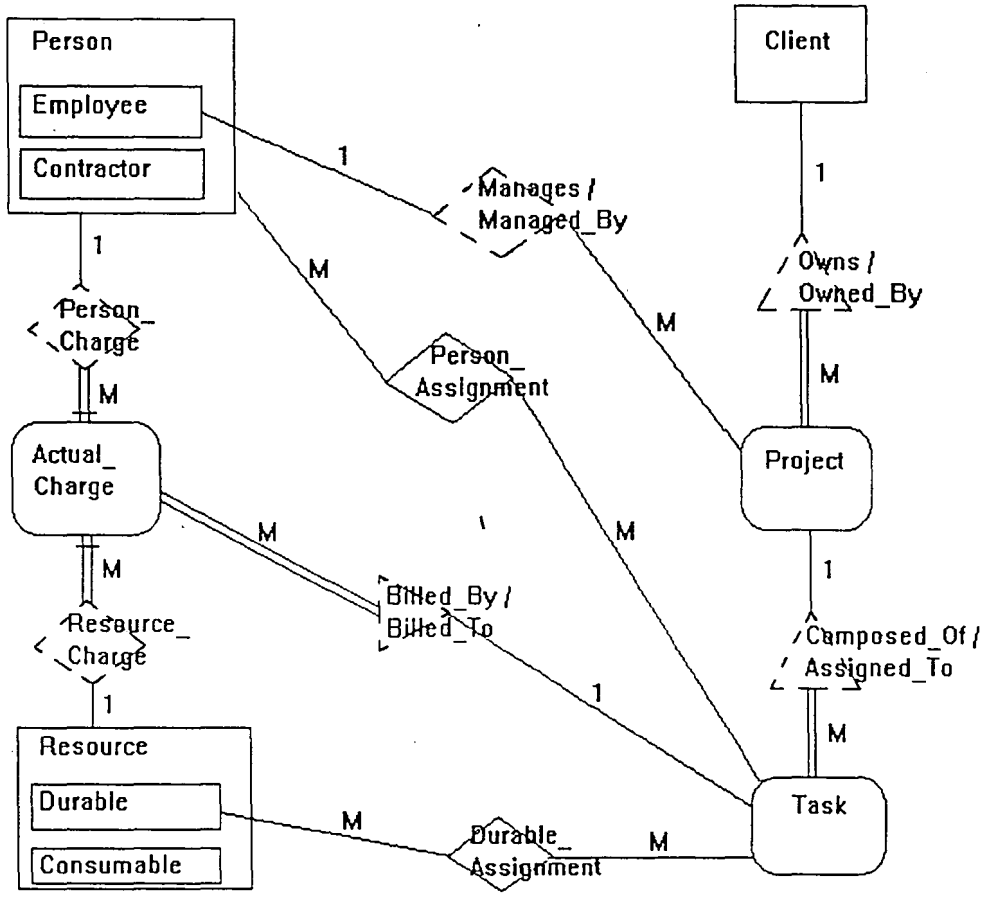


Figure 5-2: Project Management Modified Extended ER Supplemental Data

Entity: Person

Attributes:	#	*	Person_Id	Numeric (6)
		*	Name	Alpha (40)
			Charge_Rate_Wk	Numeric (4,2)
		*	Person_Subtype	Alpha (1)

Entity: Employee

Attributes:		*	Social_Sec_Nbr	Numeric (9)
		*	Hire_Date	Date
		*	Pay_Rate_Wk	Numeric (4,2)

Entity: Contractor

Attributes:		*	Contractor_Id	Numeric (9)
		*	Fed_Emplr_Id	Numeric (9)
			Contract_Start	Date
			Contract_Stop	Date
			Contract_Rate_Mo	Numeric (5,2)

Entity: Resource

Attributes: # * Resource_Id Numeric (4)

* Resource_Title Alpha (40)

Charge_Rate_Unit_Of_Meas Alpha (4)

Resource_Class Alpha (4)

* Resource_Subtype Alpha (1)

Entity: Durable

Attributes: * Quantity Numeric (5)

Use_Restrictions Alphanum (60)

Entity: Consumable

Attributes: * Replenish_Days Numeric (3)

Entity: Actual_Charge

Attributes: # * Actual_Charge_Id Numeric (7)

* Actual_Charge_Date Date

* Actual_Charge_Rate Numeric (5,2)

* Actual_Charge_Unit_Of_Meas Alpha (4)

* Actual_Charge_Quantity Numeric (3,2)

Actual_Charge_Start_Date Date

Actual_Charge_Stop_Date Date

Billed_To_Project_Id Numeric (5)

Billed_To_Task_Id Numeric (3)

Person_Charge_Person_Id Numeric (6)

Resource_Charge_Resource_Id

Numeric (4)

Relationship Billed_To Attributes:

* Billed_To_Project_Id

* Billed_To_Task_Id

Foreign Key Billed_To_Project_Id,

Billed_To_Task_Id

Relationship Person_Charge Attributes:

* Person_Charge_Person_Id

Foreign Key Person_Charge_Person_Id

Relationship Resource_Charge Attributes:

* Resource_Charge_Resource_Id

Foreign Key

Resource_Charge_Resource_Id

Integrity Rules: Foreign Key Billed_To_Project_Id,
Billed_To_Task_Id
Nulls Not Allowed
References Task.Project_Id,
Task.Task_Id
Delete of Task Restricted
Update of Task.Project_Id,
Task.Task_Id Cascades
Foreign Key Person_Charge_Person_Id
Nulls Allowed (see note below)
References Person.Person_Id
Delete of Person Restricted
Update of Person.Person_Id Cascades
Foreign Key
Resource_Charge_Resource_Id
Nulls Allowed (see note below)
References Resource.Resource_Id
Delete of Resource Restricted
Update of Resource.Resource_Id
Cascades
((Foreign Key Person_Charge_Person_Id
Must Not Be Null and
Foreign Key
Resource_Charge_Resource_Id
Must Be Null) or
(Foreign Key Person_Charge_Person_Id

Must Be Null and

Foreign Key

Resource_Charge_Resource_Id

Must Not Be Null))

{see note below}

Note: The combination of the "Nulls Allowed" specification on the foreign keys of the second and third integrity rules and the fourth rule fully enforces the mandatory participation of each Actual_Charge instance in either a Person_Charge or Resource_Charge relationship, but not both.

Entity: Client

Attributes: # * Client_Id Numeric (9)

* Client_Name Alpha (40)

Contact_Name Alpha (40)

Contact_Title Alpha (50)

Contact_Addr_Line_1 Alphanum (50)

Contact_Addr_Line_2 Alphanum (50)

Contact_Phone Numeric (10)

Client_Rating Alphanum (2)

Entity: Project

Attributes: # * Project_Id Numeric (5)

* Project_Name Alpha (40)

Start_Date Date

Want_Date Date

Promise_Date Date

Owned_By_Client_Id Numeric (9)

Managed_By_Person_Id Numeric (6)

Relationship Owned_By Attributes:

* Owned_By_Client_Id

Foreign Key Owned_By_Client_Id

Relationship Managed_By Attributes:

* Managed_By_Person_Id

Foreign Key Managed_By_Person_Id

Integrity Rules: Foreign Key Owned_By_Client_Id

Nulls Not Allowed

References Client.Client_Id

Delete of Client Restricted

Update of Client.Client_Id Cascades

Foreign Key Managed_By_Person_Id

Nulls Allowed

References Employee.Person_Id

Delete of Employee Nullifies

Update of Employee.Person_Id

Cascades

Entity: Task

Attributes: # * Project_Id Numeric (5)

* Task_Id Numeric (3)

* Task_Name Alpha (40)

Start_Date Date

Stop_Date Date

Relationship Assigned_To Attributes:

* Project_Id

Foreign Key Project_Id

Integrity Rules: Foreign Key Project_Id

Nulls Not Allowed

References Project.Project_Id

Delete of Project Restricted

Update of Project.Project_Id

Cascades

Relationship: Person_Assignment

Attributes: # * Person_Id Numeric (6)

* Project_Id Numeric (5)

* Task_Id Numeric (3)

Start_Date Date

Stop_Date Date

Integrity Rules: Foreign Key Person_Id

References Person.Person_Id

Delete of Person Cascades

Update of Person.Person_Id Cascades

Foreign Key Project_Id, Task_Id

References Task.Project_Id,

Task.Task_Id

Delete of Task Restricted

Update Of Task.Project_Id,

Task.Task_Id Cascades

Relationship: Durable_Assignment

Attributes: # * Resource_Id Numeric (4)

* Project_Id Numeric (5)

* Task_Id Numeric (3)

Start_Date Date

Stop_Date Date

Integrity Rules: Foreign Key Resource_Id

References Durable.Resource_Id

Delete of Durable Cascades

Update of Durable.Resource_Id

Cascades

Foreign Key Project_Id, Task_Id

References Task.Project_Id,

Task.Task_Id

Delete of Task Restricted

Update Of Task.Project_Id,

Task.Task_Id Cascades

Figure 5-3: Project Management Derived Relational DB Design

```
Create Table Person
  Columns:  Person_Id  Numeric (6)  Not Null
            Name  Alpha (40)  Not Null
            Charge_Rate_Wk  Numeric (4,2)  Null
            Person_Subtype  Alpha (1)  Not Null
              With Check Option (Person_Subtype = 'E' or
                'C')
            Employee_Social_Sec_Nbr  Numeric (9)  Null
            Employee_Hire_Date  Date  Null
            Employee_Pay_Rate_Wk  Numeric (4,2)  Null
            Contractor_Contractor_Id  Numeric (9)  Null
            Contractor_Fed_Emplr_Id  Numeric (9)  Null
            Contractor_Contract_Start  Date  Null
            Contractor_Contract_Stop  Date  Null
            Contractor_Contract_Rate_Mo  Numeric (5,2)  Null
  Integrity Rules:
            Primary Key = (Person_Id)
            Subtype_Employee Unit =
             ((Employee_Social_Sec_Nbr  Not Null
               Employee_Hire_Date  Not Null
               Employee_Pay_Rate_Wk  Not Null)
               With Check Option = (Person_Subtype = 'E'))
            Subtype_Contractor Unit =
             ((Contractor_Contractor_Id  Not Null
```

```
                Contractor_Fed_Emplr_Id  Not Null
                Contractor_Contract_Start  Null
                Contractor_Contract_Stop  Null
                Contractor_Contract_Rate_Mo  Null)
                With Check Option = (Person_Subtype = 'C'))

Create Unique Index Person_Primary_Key_Index On Person
  (Person_Id  ASC)

Create Index Person_Subtype_Index On Person
  (Person_Subtype, Person_Id  ASC)
```

```
Create View Person_V
  As (Select Person_Id
             Name
             Charge_Rate_Wk
             Person_Subtype
        From Person)

Create View Employee
  As (Select Person_Id
             Name
             Charge_Rate_Wk
             Person_Subtype
             Subtype_Employee Unit
        From Person Where Person_Subtype = 'E')

Create View Contractor
  As (Select Person_Id
             Name
             Charge_Rate_Wk
             Person_Subtype
             Subtype_Contractor Unit
        From Person Where Person_Subtype = 'C')
```

```
Create Table Resource
  Columns:  Resource_Id  Numeric (4)  Not Null
            Resource_Title  Alpha (40)  Not Null
            Charge_Rate_Unit_Of_Meas  Alpha (4)  Null
            Resource_Class  Alpha (4)  Null
            Resource_Subtype  Alpha (1)  Not Null
              With Check Option = (Resource_Subtype = 'D' or
                'C')
            Durable_Quantity  Numeric (5)  Null
            Durable_Use_Restrictions  Alphanum (60)  Null
            Consumable_Replenish_Days  Numeric (3)  Null
  Integrity Rules:
            Primary Key = (Resource_Id)
            Subtype_Durable Unit =
             ((Durable_Quantity  Not Null
               Durable_Use_Restrictions  Null)
               With Check Option = (Resource_Subtype = 'D'))
            Subtype_Consumable Unit =
             ((Consumable_Replenish_Days  Not Null)
               With Check Option = (Resource_Subtype = 'C'))
```

```
Create Unique Index Resource_Primary_Key_Index On Resource
  (Resource_Id  ASC)

Create Index Resource_Subtype_Index On Resource
  (Resource_Subtype, Resource_Id  ASC)
```

```
Create View Resource_V
  As (Select Resource_Id
             Resource_Title
             Charge_Rate_Unit_Of_Meas
             Resource_Class
             Resource_Subtype
        From Resource)

Create View Durable
  As (Select Resource_Id
             Resource_Title
             Charge_Rate_Unit_Of_Meas
             Resource_Class
             Resource_Subtype
             Subtype_Durable Unit
        From Resource Where Resource_Subtype = 'D')

Create View Consumable
  As (Select Resource_Id
             Resource_Title
             Charge_Rate_Unit_Of_Meas
             Resource_Class
             Resource_Subtype
             Subtype_Consumable Unit
        From Resource Where Resource_Subtype = 'C')
```

```
Create Table Actual_Charge
  Columns:  Actual_Charge_Id  Numeric (7)  Not Null
            Actual_Charge_Date  Date  Not Null
            Actual_Charge_Rate  Numeric (5,2)  Not Null
            Actual_Charge_Unit_Of_Meas  Alpha (4)  Not Null
            Actual_Charge_Quantity  Numeric (5,2)  Not Null
            Actual_Charge_Start_Date  Date  Null
            Actual_Charge_Stop_Date  Date  Null
            Billed_To_Project_Id  Numeric (5)  Null
            Billed_To_Task_Id  Numeric (3)  Null
            Person_Charge_Person_Id  Numeric (6)  Null
            Resource_Charge_Resource_Id  Numeric (4)  Null
  Integrity Rules:
            Primary Key = (Actual_Charge_Id)
            Billed_To Unit = (Billed_To_Project_Id  Not Null
                              Billed_To_Task_Id  Not Null)
            Billed_To Foreign Key = ((Billed_To_Project_Id,
                                      Billed_To_Task_Id)
              Nulls Not Allowed
              References (Task.Project_Id,
                          Task.Task_Id)
              Delete of Task Restricted
              Update of (Task.Project_Id,
                         Task.Task_Id) Cascades)
            Person_Charge Unit =
              (Person_Charge_Person_Id  Not Null)
```

```
Person_Charge Foreign Key =
  (Person_Charge_Person_Id
  Nulls Allowed
  References Person.Person_Id
  Delete of Person Restricted
  Update of Person.Person_Id Cascades)
Resource_Charge Unit =
  (Resource_Charge_Resource_Id  Not Null)
Resource_Charge Foreign Key =
  (Resource_Charge_Resource_Id
  Nulls Allowed
  References Resource.Resource_Id
  Delete of Resource Restricted
  Update of Resource.Resource_Id Cascades)
Person_Resource_Charge Rule =
  ((Person_Charge_Person_Id Must Not Be Null and
    Resource_Charge_Resource_Id Must Be Null) or
   (Person_Charge_Person_Id Must Be Null and
    Resource_Charge_Resource_Id
      Must Not Be Null))
```

```
Create Unique Index Actual_Charge_Primary_Key_Index On
  Actual_Charge  (Actual_Charge_Id  ASC)

Create Index Actual_Charge_Rel_Billed_To_Index On
  Actual_Charge
  (Billed_To_Project_Id, Billed_To_Task_Id  ASC)

Create Index Actual_Charge_Rel_Person_Index On
  Actual_Charge  (Person_Charge_Person_Id  ASC)

Create Index Actual_Charge_Rel_Resource_Index On
  Actual_Charge  (Resource_Charge_Resource_Id  ASC)
```

```
Create Table Client

  Columns:  Client_Id  Numeric (9)  Not Null

            Client_Name  Alpha (40)  Not Null

            Contact_Name  Alpha (40)  Null

            Contact_Title  Alpha (50)  Null

            Contact_Addr_Line_1  Alphanum (50)  Null

            Contact_Addr_Line_2  Alphanum (50)  Null

            Contact_Phone  Numeric (10)  Null

            Client_Rating  Alphanum (2)  Null

  Integrity Rules:

            Primary Key = (Client_Id)


Create Unique Index Client_Primary_Key_Index On Client

  (Client_Id  ASC)
```

```
Create Table Project
  Columns:  Project_Id  Numeric (5)  Not Null
            Project_Name  Alpha (40)  Not Null
            Start_Date  Date  Null
            Want_Date  Date  Null
            Promise_Date  Date  Null
            Owned_By_Client_Id  Numeric (9)  Null
            Managed_By_Person_Id  Numeric (6)  Null
  Integrity Rules:
            Primary Key = (Project_Id)
            Owned_By Unit = (Owned_By_Client_Id  Not Null)
            Owned_By Foreign Key = (Owned_By_Client_Id
              Nulls Not Allowed
              References Client.Client_Id
              Delete of Client Restricted
              Update of Client.Client_Id Cascades)
            Managed_By Unit =
              (Managed_By_Person_Id Not Null)
            Managed_By Foreign Key = (Managed_By_Person_Id
              Nulls Allowed
              References Employee.Person_Id
              Delete of Employee Nullifies Managed_By Unit
              Update of Employee.Person_Id Cascades)
```

```
Create Unique Index Project_Primary_Key_Index On Project
  (Project_Id  ASC)

Create Index Project_Rel_Owned_By_Index On Project
  (Owned_By_Client_Id  ASC)

Create Index Project_Rel_Managed_By_Index On Project
  (Managed_By_Person_Id  ASC)
```

```
Create Table Task
  Columns:  Project_Id  Numeric (5)  Not Null
            Task_Id  Numeric (3)  Not Null
            Task_Name  Alpha (40)  Not Null
            Start_Date  Date  Null
            Stop_Date  Date  Null
  Integrity Rules:
            Primary Key = (Project_Id, Task_Id)
            Assigned_To Unit = (Project_Id  Not Null)
            Assigned_To Foreign Key = (Project_Id
              Nulls Not Allowed
              References Project.Project_Id
              Delete of Project Restricted
              Update of Project.Project_Id Cascades)

Create Unique Index Task_Primary_Key_Index On Task
  (Project_Id, Task_Id  ASC)
```

```
Create Table Person_Assignment
  Columns:  Person_Id  Numeric (6)  Not Null
            Project_Id  Numeric (5)  Not Null
            Task_Id  Numeric (3)  Not Null
            Start_Date  Date  Null
            Stop_Date  Date Null
  Integrity Rules:
            Primary Key = (Person_Id, Project_Id, Task_Id)
            Foreign Key = (Person_Id
              Nulls Not Allowed
              References Person.Person_Id
              Delete of Person Cascades
              Update of Person.Person_Id Cascades)
            Foreign Key = ((Project_Id, Task_Id)
              Nulls Not Allowed
              References (Task.Project_Id, Task.Task_Id)
              Delete of Task Restricted
              Update of (Task.Project_Id, Task.Task_Id)
                Cascades)

Create Unique Index Person_Assignment_Primary_Key_Index On
  Person_Assignment  (Person_Id, Project_Id, Task_Id  ASC)

Create Index Person_Assignment_Rel_Project_Task_Index On
  Person_Assignment  (Project_Id, Task_Id  ASC)
```

```
Create Table Durable_Assignment
  Columns:  Resource_Id  Numeric (4)  Not Null
            Project_Id  Numeric (5)  Not Null
            Task_Id  Numeric (3)  Not Null
            Start_Date  Date  Null
            Stop_Date  Date  Null
  Integrity Rules:
            Primary Key = (Resource_Id, Project_Id, Task_Id)
            Foreign Key = (Resource_Id
              Nulls Not Allowed
              References Durable.Resource_Id
              Delete of Durable Cascades
              Update of Durable.Resource_Id  Cascades)
            Foreign Key = ((Project_Id, Task_Id)
              Nulls Not Allowed
              References (Task.Project_Id, Task.Task_Id)
              Delete of Task Restricted
              Update of (Task.Project_Id, Task.Task_Id)
                Cascades)
```

```
Create Unique Index Durable_Assignment_Primary_Key_Index On
  Durable_Assignment
  (Resource_Id, Project_Id, Task_Id  ASC)

Create Index Durable_Assignment_Rel_Project_Task_Index On
  Durable_Assignment  (Project_Id, Task_Id  ASC)
```

Chapter 6 Design Issues Raised by the Use of Extended ER Modeling

6.1 Introduction

The primary objective of utilizing an Extended ER Modeling technique is to facilitate the identification and organization of the semantic concepts that are essential to the achievement of an application system's goals. These semantic concepts can then be used as the basis to define the RDBMS structures and rules necessary to represent the target application domain. These derived structures and rules comprise the actual relational database design. In this chapter, I will discuss possible modifications to, or compromises on, the default database design. A constant underlying aspect of these discussions will be the impact of any contemplated design modifications on the quality features of the database, such as the operational feature of normalization and the usability features of flexibility, clarity, efficiency and semantic integrity. Within certain discussions, I will use the function definition requirements associated with each application domain (created in parallel with the Extended ER Model, see chapter 2) as a source of additional criteria to evaluate the overall quality of the default database design. These functions provide a useful reference point from which to examine and judge the overall effectiveness of the default design. It is likely that a large number of functions will be associated with each

application system. From these, a representative set of functions must be chosen and measured against the database design to determine its strengths and to uncover its weaknesses. The representative set of functions should include: all application performance-critical functions, a sampling of the most complex functions, and some randomly selected functions. [17] Through this analysis, we can assure the acceptability of the default database design, or identify points of improvement to achieve the required level of quality.

The following sections discuss permissible alterations to the default design that can be used to improve the quality of the design without compromising the business effectiveness of the model.

6.2 Denormalization

Strict data normalization can easily lead to the specification of a large number of tables. [18] Normalization seeks to reduce the complexity of maintaining data and enforcing its integrity by removing duplicate data. In general, normalized structures offer greater data control and update efficiency at the expense of function and reporting efficiency. This is because the body of normalized data must be manipulated, such as combined or summarized, in order to be useful in the accomplishment of functions or the

production of reports.

Denormalization techniques seek to improve the performance of vital functions by [19]:

- reducing the number of tables accessed per each function, primarily by reducing the need to join tables
- reducing the absolute number of rows in specific tables, those most often accessed by the significant functions
- reducing the number of real-time calculations necessary to accomplish the significant functions

In essence, denormalization techniques aim to improve a database design by adding controlled redundancy or by partitioning the target data. A discussion of the basic techniques follows. [20]

The first technique is to choose table structures that represent joined data, in violation of the normal forms definitions. An example might be to include reference data (such as, data that describes or translates code values) directly in the base table. Under a normalized structure, we would want to keep these descriptions separate from the base tables so that if a description would change, it would only need to be updated in one place, the reference data table. But, because these reference descriptions rarely change (and when they do, we are only adding new descriptions, not changing current ones) and because they are often

accessed in a set of critical functions and reports, we can choose to store the information in a joined format. This format reduces the I/O operations for each access (due to the fact that only a single table is accessed) and creates a simpler structure for users to examine. The reference data table remains, to support the validation of new data inserted into the base table, but not to support the utility of the data once it is inserted.

The second technique includes a series of possible design modifications that pursue function and reporting efficiency gains by storing redundant data in forms that are more easily and directly used by (that is, specifically designed to achieve or meet the needs of) critical functions and reports. These modifications seek to improve efficiency by storing artificially generated data, normally in the form of calculated values. These values may include appended calculated columns, where the values of many columns within a single row are used as inputs to a function to yield a calculated value, or summary rows in public summary tables that are calculated from detail data in operational tables. The first of these, calculated columns, make these values easily and quickly accessible to all processes that require them, without the need to explicitly invoke or perfectly replicate the function's logic. Of course, the disadvantage of utilizing calculated columns is that whenever a detail

data input within the row is modified, we must be certain that the RDBMS or the application software refreshes the calculated column. A lapse, here, would disrupt the basic integrity of the database. Calculated columns are best employed when the data, upon which the derived value is based, rarely changes. Regarding public summary data, this class of calculated values is less sensitive to database changes because it is usually historical. For example, the monthly payroll activity for each department could be summarized into a single row of data. New values would be calculated at the end of each month, but all data related to the previous months would remain intact. These values need to be calculated only once, yet they are continuously available to all processes that require the data while allowing each process to avoid the effort necessary to recalculate the values by passing through all of the detail operational values. An interesting refinement of this approach would be to retain the aggregated data in a Rolling Summarization format. By rolling summarization, I mean that the granularity of the summarized data is increased over time. In the payroll activity example mentioned earlier, the initial granularity of the summarized data might be weekly. And, then, as the data ages, the granularity might be elevated to monthly and even annual values. Again, the intent of retaining these summarized values is to achieve efficiency by allowing certain processes and reports to avoid a level of

detail data that is unimportant to their objectives.

A final technique seeks to increase performance through the artificial partitioning of base table data, i.e., reducing the number of rows searched, or the width of each row searched, to satisfy a function or reporting module. Horizontal partitioning can be applied when a normalized table contains some columns that are both broad (i.e., lengthy) and seldom referenced. For example, if a table contains required quality specifications for parts, as well as extended comments detailing the development of these specifications, it would be legitimate to divide this table into two when the vast majority of the processes accessing the table do not require the extended comment data. Each of those processes, then, enjoy faster access to the desired data because it is unnecessary to read and discard the comments. Processes that do require the extended comments may access that data directly from the second table or they may join it to the contents of the original base table. Vertical partitioning seeks to achieve efficiency by segregating a table into logical subsets that correspond in content to the subsets of data accessed by the most frequently performed or time-critical processes. This segregation is normally based on static values within each row (i.e., we do not want a situation where individual detail rows are frequently migrating between the many tables that

represent each logical subset - this would significantly complicate the database maintenance routines). Segregation value types might include currency measurements, where, for example, three tables are kept, one for products under development, one for current products and one for discontinued products. Segregation may also be employed where well-defined categories exist, so that separate sales data for each business sector or sales district may be stored in distinct tables. If the majority of the functions desire to retrieve and analyze the data as these subsets are constructed, then this can be a useful technique. However, whenever the data required for a function or process overlaps several subsets, this structure of several tables can become cumbersome to access and analyze.

6.3 Optional Indexes

Indexes, beyond those identified in the default database design derivation process, can substantially improve a design's quality by speeding query execution. [21]

To determine where indexes on non-key columns can be most effectively employed, the representative group of application functions, selected earlier, must be closely examined. Columns that are referenced in the "Where" clauses of critical functions, or are simply referenced in the "Where" clauses of many typical functions, should receive an index. Additionally, columns that are referenced

in SQL group functions, such as MIN and MAX, are also good candidates for indexing. These optional indexes can reduce database I/O at query execution time by permitting the RDBMS' query optimization routines to select a direct path, via an index, to the desired rows within a table without expending the effort to access and examine all rows contained within the target table. Again, these additional indexes should only be specified if there are particular functions that can benefit from their existence. The presence or absence of an index will not enable or prohibit the achievement of any function within the application system, it will only affect the efficiency of the database's response. Occasionally, the indexes suggested by some functions would be inappropriate. For example, if the selectivity (range) of the values contained within a column is small, an index may actually degrade the database's performance. Such a case would exist when a column contains a relatively even number of two or three possible values. If every physical unit of the database storage for the table contains some desired rows, then the use of an index will actually slow the execution of the query by causing the RDBMS to access index storage pages in addition to all of the associated table data storage pages. If the index does not exist, a simple full table scan (which is the default search procedure to satisfy a query) will be used to achieve

the same end. But, the full table scan will not expend any effort tracing through the index. In general, if the use of an index will always select 15% or more of the rows within a table, it should not be defined. In another case, any indexes identified for columns in small tables (250 rows or less) should not be defined. This is because the lengthened access path to the data, i.e., by going through the index first and then, second, to the data storage pages, may require more time than just performing a simple full table scan on the small number of actual rows.

Indexes can substantially improve the performance of queries against a database, yet indexes do have a cost. Each time an INSERT, UPDATE or DELETE SQL statement is performed on a table, not only must the table data be modified, but all indexes must be updated to correspond to the new table data. Therefore, while the extensive use of indexes may improve query performance, their use may make the performance of database update operations unacceptable. Database query and update performance objectives must be reconciled and balanced, leading to a practical limit on the number of indexes placed on each table. The timing of database update functions can be used to determine an acceptable upper limit on the number of indexes for each table. Occasionally, the absolute number of indexes is not a constraining factor, but rather the change frequency of the values within the candidate column. If the values

change too frequently, the cost of maintaining the index may be much greater than its benefit to the group of functions that could utilize it. In this case, it is best to leave the index undefined.

To this point, I have spoken strictly of indexes that can be applied to base tables. An alternative object, an index table, could be defined to meet the needs of a few critical functions. An index table simply contains rows of primary keys that identify and correspond to rows in some base or reference table. An index table is manually maintained by the application software. It is best employed when a huge table (with hundreds of thousands or millions of rows) has a very small subset of rows that is required for a particular set of functions. In this case, the application software maintains a separate table of primary key values that correspond to the desired subset. Then, when that subset needs to be accessed, it is identified by the rows in the index table. Other column values are "joined" to this index table from the base table as required to satisfy the requirements of the functions. This approach can be much more efficient than the normal approach of maintaining an index directly over the base table. The index of the base table would track all rows in the base table, not just the few that the set of functions require. Unfortunately, having the application software maintain the index table, rather

than having the RDBMS maintain a normal index over the base table, introduces some new integrity concerns. So, this approach must only be initiated when the demands of the application require it.

6.4 Retaining One-to-One and One-to-Many Relationship Types As Distinct Tables

For purposes of flexibility and clarity, it may be prudent to retain the distinct definition of one-to-one and one-to-many relationship types, rather than transforming them into components of the participating entities as described in chapter 4. [22] Implementing these relationship types as tables allows the database design to easily evolve. With a separate table for each relationship type, the cardinality of the relationship can easily move from one-to-one or one-to-many to the more complex situation of a many-to-many relationship, without causing a redesign of the database. This is an important point. The example Project Management application, discussed in chapters 3 and 5, displays a potential weakness associated with the implementation of the Manages / Managed_By relationship type. As it is implemented, the design will fail as soon as the user of the application wishes to assign co-managers to a project (i.e., altering the relationship type from a one-to-many to a many-to-many cardinality, requiring a distinct table to correctly represent the data). Whenever there is a legiti-

mate possibility that a current relationship type's cardinality could evolve into a many-to-many state, then the relationship type should be transformed into a distinct table rather than integrated with a participating entity. (The preceding discussion can be extended to include the design weakness resulting from choosing the "wrong" participating entity type as the recipient of a foreign key associated with a one-to-one relationship type.)

The one-to-one and one-to-many relationship types can be transformed into tables following the same procedure used to transform the many-to-many relationship types. As this alternate derivation process is followed, each foreign key within these new tables will receive a default index. When the foreign key represents a "one" membership condition, the index would receive a "unique" specification. And, when the foreign key represents a "many" membership condition, the index would not receive a "unique" specification. If the relationship's type cardinality does change from a "one" to a "many" condition, the index's "unique" specification constraint can then be relaxed (i.e., removed).

Of course, the added flexibility of implementing each relationship type as a distinct table does have a cost. First, there is a slightly larger database maintenance effort, in terms of a larger number of tables to be maintained and indexed. And, second, certain database query

operations will be slower because the path between the entities in these relationships will now be longer. That is, rather than directly "joining" the two tables representing the entities in the relationship, a third intermediate table representing the relationship type must now be "joined" to each of the members. These costs must be measured against the benefits of a more flexible design implementation.

Footnotes

1. The majority of these features are discussed in Fidel, chapter 8.

2. Fidel, chapter 7.

3. Normalization theory is discussed in Rodgers, chapter 4, and Bisland, p 361 - 369.

4. Ullman, chapter 7.

5. See Rodgers, chapter 6, for an interesting "Real Time vs Real Quick vs Human Time" discussion.

6. Date, An Introduction to Database Systems, chapter 22.

7. Barker, CASE Method: Tasks and Deliverables, p 4-13 and G-2.

8. Oracle Corporation, SQL Language Reference Manual, p 1-2.

9. My primary sources are Date, An Introduction to Database Systems, and Barker, CASE Method: Entity Relationship Modelling.

10. Barker, CASE Method: Entity Relationship Modelling, chapter 3 and p 5-2, plus Rodgers, p 38.

11. Barker, CASE Method: Entity Relationship Modelling, chapter 3.

12. Date, Relational Database Writings, p 116 - 117.

13. Date, An Introduction to Database Systems, p 281.

14. Date, Relational Database Writings, p 118 - 121.

15. Whittington, chapter 10, and Barker, CASE Method: Entity Relationship Modelling, appendix F.

16. Bisland, chapter 15, and Barker, CASE Method: Entity Relationship Modelling, appendix F.

17. Barker, CASE Method: Tasks and Deliverables, p 5-12.

18. Malamud, p 272.

19. Rodgers, p 94.

20. This discussion draws on three sources: Inmon, p 177 - 188, Malamud, p 272 - 275, and Barker, CASE Method: Entity Relationship Modelling, p F-13.

21. This discussion draws on three sources: Oracle Corporation, Oracle Database Administrator's Guide, p 5-11 - 5-15, Oracle Corporation, Oracle RDBMS Tuning Guide, p 2-7 - 2-14, and Barker, CASE Method: Entity Relationship Modelling, p F-3.

22. Date, An Introduction to Database Systems, p 586.

Bibliography

Barker, Richard. 1990. CASE Method: Entity Relationship Modelling. Reading, Massachusetts: Addison-Wesley Publishing Company.

Barker, Richard. 1990. CASE Method: Tasks and Deliverables. Reading, Massachusetts: Addison-Wesley Publishing Company.

Bisland, Ralph B. 1989. Database Management: Developing Application Systems Using Oracle. Englewood Cliffs, New Jersey: Prentice-Hall.

Date, C. J. 1990. An Introduction to Database Management Systems, Volume I, Fifth Edition. Reading, Massachusetts: Addison-Wesley Publishing Company.

Date, C. J. 1990. Relational Database Writings, 1985 - 1989. Reading, Massachusetts: Addison-Wesley Publishing Company.

Fidel, Raya. 1987. Database Design for Information Retrieval. New York: John Wiley & Sons.

Inmon, W. H. 1990. Using Oracle to Build Decision Support Systems. Wellesley, Massachusetts: QED Information Sciences.

Malamud, Carl. 1989. Ingres: Tools for Building an Information Architecture. New York: Van Nostrand Reinhold.

Oracle Corporation. 1989. Oracle Database Administrator's Guide, Version 6.0. Redwood Shores, California: Oracle Corporation.

Oracle Corporation. 1990. Oracle RDBMS Tuning Guide, Version 6.0. Redwood Shores, California: Oracle Corporation.

Oracle Corporation. 1990. SQL Language Reference Manual, Version 6.0. Redwood Shores, California: Oracle Corporation.

Rodgers, Ulka. 1991. Oracle: A Database Developer's Guide. Englewood Cliffs, New Jersey: Prentice-Hall.

Ullman, Jeffrey D. 1988. Principles of Database and Knowledge-Based Systems. Rockville, Maryland: Computer Science Press.

Whittington, R. P. 1988. Database Systems Engineering. New York: Oxford University Press.

Vita

Randall Wambold was born on December 13, 1954. He received an Associate in Applied Science degree from Northampton Community College in 1974 and a Bachelor of Arts degree in Business and Economics from Lafayette College in 1982. He is currently the Systems and Programming Manager of the Administrative Systems Office at Lehigh University.

END

OF

TITLE