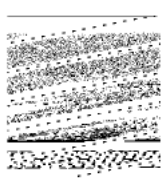
Database Modeling & Design: Logical Design

Fourth Edition

TOBY TEOREY
SAM LIGHTSTONE
TOM NADEAU



AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO
MORGAN KAUFMANN PUBLISHERS IS AN IMPRINT OF ELSEVIER

MORGAN KAUFMANN PUBLISHERS

Publishing Director	Michael Forster
Publisher	Diane Cerra
Publishing Services Manager	Simon Crump
Editorial Assistant	Asma Stephan
Cover Design	Yvo Riezebos Design
Cover Image	Getty Images
Composition	Multiscience Press, Inc.
Technical Illustration	Dartmouth Publishing, Inc.
Copyeditor	Multiscience Press, Inc.
Proofreader	Multiscience Press, Inc.
Indexer	Multiscience Press, Inc.
Interior printer	Maple-Vail Book Manufacturing Group
Cover printer	Phoenix Color

Morgan Kaufmann Publishers is an imprint of Elsevier.
500 Sansome Street, Suite 400, San Francisco, CA 94111

This book is printed on acid-free paper.

Library of Congress Cataloging-in-Publication Data
Application submitted.

ISBN 13: 978-0-12-685352-0
ISBN 10: 0-12-685352-5

For information on all Morgan Kaufmann publications,
visit our Web site at www.mkp.com or www.books.elsevier.com

Printed in the United States of America
05 06 07 08 09 5 4 3 2 1

Contents

Chapter 9

1 Introduction

Database technology has evolved rapidly in the three decades since the rise and eventual dominance of relational database systems. While many specialized database systems (object-oriented, spatial, multimedia, etc.) have found substantial user communities in the science and engineering fields, relational systems remain the dominant database technology for business enterprises.

Relational database design has evolved from an art to a science that has been made partially implementable as a set of software design aids. Many of these design aids have appeared as the database component of computer-aided software engineering (CASE) tools, and many of them offer interactive modeling capability using a simplified data modeling approach. Logical design—that is, the structure of basic data relationships and their definition in a particular database system—is largely the domain of application designers. These designers can work effectively with tools such as ERwin Data Modeler or Rational Rose with UML, as well as with a purely manual approach. Physical design, the creation of efficient data storage and retrieval mechanisms on the computing platform being used, is typically the domain of the database administrator (DBA). Today's DBAs have a variety of vendor-supplied tools available to help design the most efficient databases. This book is devoted to the logical design methodologies and tools most popular for relational databases today. Physical design methodologies and tools are covered in a separate book.

In this chapter, we review the basic concepts of database management and introduce the role of data modeling and database design in the database life cycle.

1.1 Data and Database Management

The basic component of a file in a file system is a *data item*, which is the smallest named unit of data that has meaning in the real world—for example, last name, first name, street address, ID number, or political party. A group of related data items treated as a single unit by an application is called a *record*. Examples of types of records are order, salesperson, customer, product, and department. A *file* is a collection of records of a single type. Database systems have built upon and expanded these definitions: In a relational database, a data item is called a *column* or *attribute*; a record is called a *row* or *tuple*; and a file is called a *table*.

A *database* is a more complex object; it is a collection of interrelated stored data that serves the needs of multiple users within one or more organizations, that is, interrelated collections of many different types of tables. The motivations for using databases rather than files include greater availability to a diverse set of users, integration of data for easier access to and updating of complex transactions, and less redundancy of data.

A *database management system* (DBMS) is a generalized software system for manipulating databases. A DBMS supports a logical view (schema, subschema); physical view (access methods, data clustering); data definition language; data manipulation language; and important utilities, such as transaction management and concurrency control, data integrity, crash recovery, and security. Relational database systems, the dominant type of systems for well-formatted business databases, also provide a greater degree of data independence than the earlier hierarchical and network (CODASYL) database management systems. *Data independence* is the ability to make changes in either the logical or physical structure of the database without requiring reprogramming of application programs. It also makes database conversion and reorganization much easier. Relational DBMSs provide a much higher degree of data independence than previous systems; they are the focus of our discussion on data modeling.

1.2 The Database Life Cycle

The database life cycle incorporates the basic steps involved in designing a global schema of the logical database, allocating data across a computer network, and defining local DBMS-specific schemas. Once the design is completed, the life cycle continues with database implementation and maintenance. This chapter contains an overview of the database life cycle, as shown in Figure 1.1. In succeeding chapters, we will focus on the database design process from the modeling of requirements through logical design (steps I and II below). The result of each step of the life cycle is illustrated with a series of diagrams in Figure 1.2. Each diagram shows a possible form of the output of each step, so the reader can see the progression of the design process from an idea to actual database implementation. These forms are discussed in much more detail in Chapters 2 through 6.

I. ***Requirements analysis***. The database requirements are determined by interviewing both the producers and users of data and using the information to produce a formal requirements specification. That specification includes the data required for processing, the natural data relationships, and the software platform for the database implementation. As an example, Figure 1.2 (step I) shows the concepts of products, customers, salespersons, and orders being formulated in the mind of the end user during the interview process.

II. ***Logical design.*** The *global schema*, a conceptual data model diagram that shows all the data and their relationships, is developed using techniques such as ER or UML. The data model constructs must ultimately be transformed into normalized (global) relations, or tables. The global schema development methodology is the same for either a distributed or centralized database.

   a. *Conceptual data modeling*. The data requirements are analyzed and modeled using an ER or UML diagram that includes, for example, semantics for optional relationships, ternary relationships, supertypes, and subtypes (categories). Processing requirements are typically specified using natural language

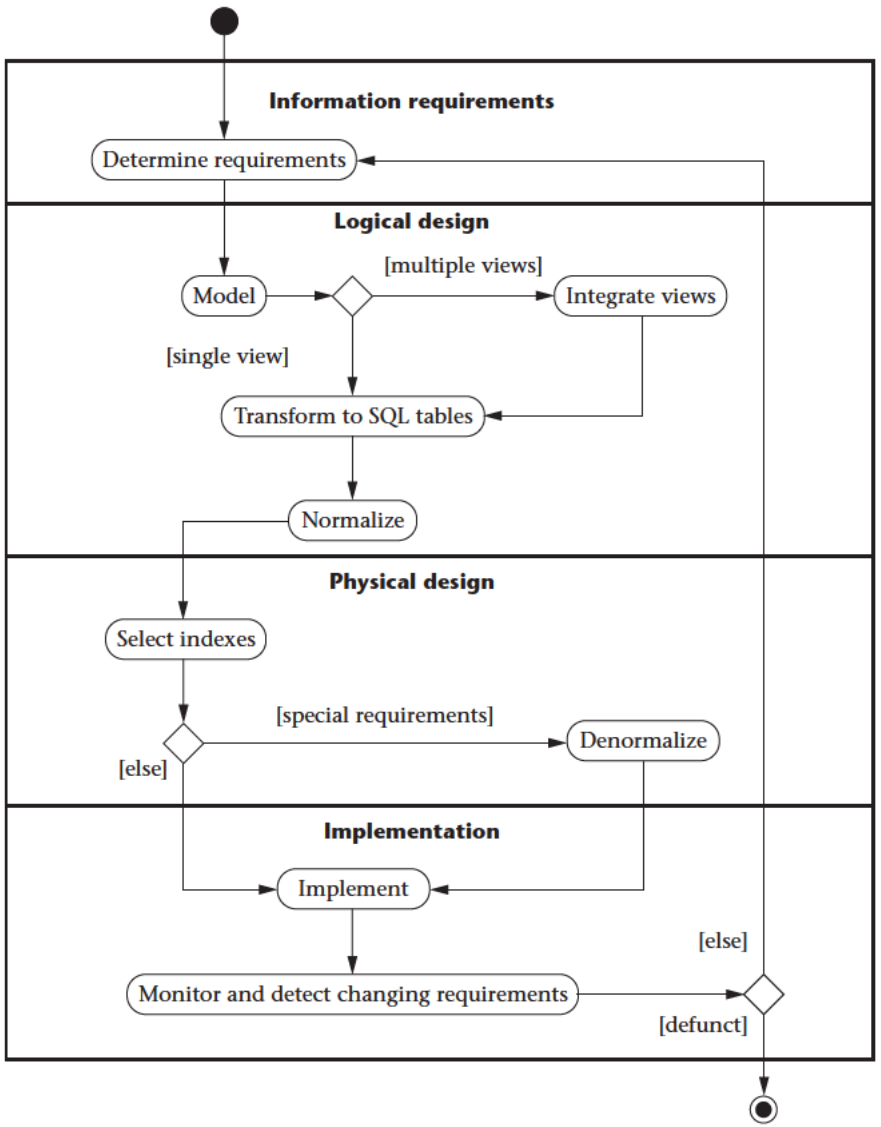


Figure 1.1 The database life cycle

expressions or SQL commands, along with the frequency of occurrence. Figure 1.2 [step II(a)] shows a possible ER model representation of the product/customer database in the mind of the end user.

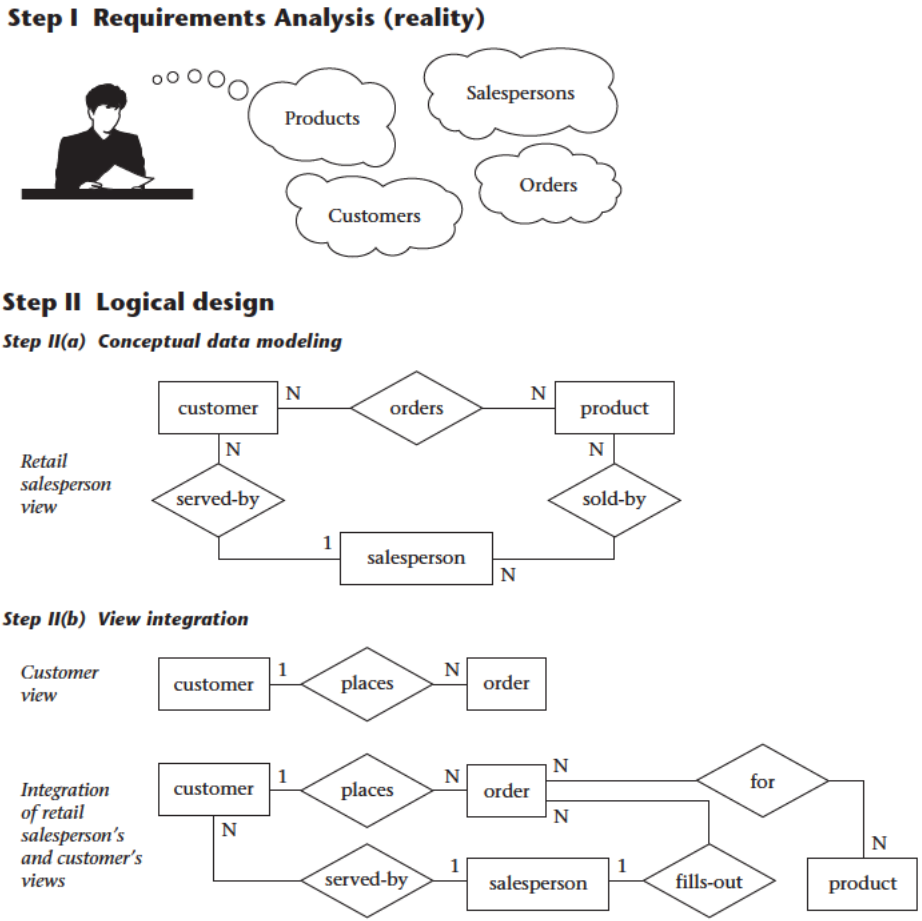


Figure 1.2 Life cycle results, step-by-step

b. *View integration.* Usually, when the design is large and more than one person is involved in requirements analysis, multiple views of data and relationships result. To eliminate redundancy and inconsistency from the model, these views must eventually be "rationalized" (resolving inconsistencies due to variance in taxonomy, context, or perception) and then consolidated into a single global view. View integration requires the use of ER semantic tools such as identification of synonyms, aggregation, and generalization. In Figure 1.2 [step

II(b)], two possible views of the product/customer database are merged into a single global view based on common data for customer and order. View integration is also important for application integration.

c. *Transformation of the conceptual data model to SQL tables.* Based on a categorization of data modeling constructs and a set of mapping rules, each relationship and its associated entities are transformed into a set of DBMS-specific candidate relational tables. We will show these transformations in standard SQL in Chapter 5. Redundant tables are eliminated as part of this process. In our example, the tables in step II(c) of Figure 1.2 are the result of transformation of the integrated ER model in step II(b).

d. *Normalization of tables.* Functional dependencies (FDs) are derived from the conceptual data model diagram and the semantics of data relationships in the requirements analysis. They represent the dependencies among data elements that are unique identifiers (keys) of entities. Additional FDs that represent the dependencies among key and nonkey attributes within entities can be derived from the requirements specification. Candidate relational tables associated with all derived FDs are normalized (i.e., modified by decomposing or splitting tables into smaller tables) using standard techniques. Finally, redundancies in the data in normalized candidate tables are analyzed further for possible elimination, with the constraint that data integrity must be preserved. An example of normalization of the **Salesperson** table into the new **Salesperson** and **SalesVacations** tables is shown in Figure 1.2 from step II(c) to step II(d).

We note here that database tool vendors tend to use the term *logical model* to refer to the conceptual data model, and they use the term *physical model* to refer to the DBMS-specific implementation model (e.g., SQL tables). Note also that many conceptual data models are obtained not from scratch, but from the process of *reverse engineering* from an existing DBMS-specific schema [Silberschatz, Korth, and Sudarshan, 2002].

III. ***Physical design.*** The physical design step involves the selection of indexes (access methods), partitioning, and clustering of data. The logical design methodology in step II simplifies the approach to designing large relational databases by reducing the

Step II(c) Transformation of the conceptual model to SQL tables

Customer

cust-no	cust-name	. . .

Product

prod-no	prod-name	qty-in-stock

Salesperson

sales-name	addr	dept	job-level	vacation-days

Order

order-no	sales-name	cust-no

Order-product

order-no	prod-no

```
create table customer
    (cust_no integer,
    cust_name char(15),
    cust_addr char(30),
    sales_name char(15),
    prod_no integer,
    primary key (cust_no),
    foreign key (sales_name)
        references salesperson
    foreign key (prod_no)
        references product);
```

Step II(d) Normalization of SQL tables

Decomposition of tables and removal of update anomalies

Salesperson

sales-name	addr	dept	job-level

Sales-vacations

job-level	vacation-days

Step III Physical design

Indexing
Clustering
Partitioning
Materialized views
Denormalization

Figure 1.2 *(continued)*

number of data dependencies that need to be analyzed. This is accomplished by inserting conceptual data modeling and integration steps [steps II(a) and II(b) of Figure 1.2] into the tradi-

tional relational design approach. The objective of these steps is an accurate representation of reality. Data integrity is preserved through normalization of the candidate tables created when the conceptual data model is transformed into a relational model. The purpose of physical design is to optimize performance as closely as possible.

As part of the physical design, the global schema can sometimes be refined in limited ways to reflect processing (query and transaction) requirements if there are obvious, large gains to be made in efficiency. This is called *denormalization*. It consists of selecting dominant processes on the basis of high frequency, high volume, or explicit priority; defining simple extensions to tables that will improve query performance; evaluating total cost for query, update, and storage; and considering the side effects, such as possible loss of integrity. This is particularly important for Online Analytical Processing (OLAP) applications.

IV. *Database implementation, monitoring, and modification.* Once the design is completed, the database can be created through implementation of the formal schema using the data definition language (DDL) of a DBMS. Then the data manipulation language (DML) can be used to query and update the database, as well as to set up indexes and establish constraints, such as referential integrity. The language SQL contains both DDL and DML constructs; for example, the *create table* command represents DDL, and the *select* command represents DML.

As the database begins operation, monitoring indicates whether performance requirements are being met. If they are not being satisfied, modifications should be made to improve performance. Other modifications may be necessary when requirements change or when the end users' expectations increase with good performance. Thus, the life cycle continues with monitoring, redesign, and modifications. In the next two chapters we look first at the basic data modeling concepts and then—starting in Chapter 4—we apply these concepts to the database design process.

1.3 Conceptual Data Modeling

Conceptual data modeling is the driving component of logical database design. Let us take a look at how this component came about, and why

it is important. Schema diagrams were formalized in the 1960s by Charles Bachman. He used rectangles to denote record types and directed arrows from one record type to another to denote a one-to-many relationship among instances of records of the two types. The *entity-relationship* (ER) approach for conceptual data modeling, one of the two approaches emphasized in this book and described in detail in Chapter 2, was first presented in 1976 by Peter Chen. The Chen form of the ER model uses rectangles to specify entities, which are somewhat analogous to records. It also uses diamond-shaped objects to represent the various types of relationships, which are differentiated by numbers or letters placed on the lines connecting the diamonds to the rectangles.

The Unified Modeling Language (UML) was introduced in 1997 by Grady Booch and James Rumbaugh and has become a standard graphical language for specifying and documenting large-scale software systems. The data modeling component of UML (now UML-2) has a great deal of similarity with the ER model and will be presented in detail in Chapter 3. We will use both the ER model and UML to illustrate the data modeling and logical database design examples throughout this book.

In conceptual data modeling, the overriding emphasis is on simplicity and readability. The goal of conceptual schema design, where the ER and UML approaches are most useful, is to capture real-world data requirements in a simple and meaningful way that is understandable by both the database designer and the end user. The end user is the person responsible for accessing the database and executing queries and updates through the use of DBMS software, and therefore has a vested interest in the database design process.

The ER model has two levels of definition—one that is quite simple and another that is considerably more complex. The simple level is the one used by most current design tools. It is quite helpful to the database designer who must communicate with end users about their data requirements. At this level you simply describe, in diagram form, the entities, attributes, and relationships that occur in the system to be conceptualized, using semantics that are definable in a data dictionary. Specialized constructs, such as "weak" entities or mandatory/optional existence notation, are also usually included in the simple form. But very little else is included, to avoid cluttering up the ER diagram while the designer's and end user's understandings of the model are being reconciled.

An example of a simple form of ER model using the Chen notation is shown in Figure 1.3. In this example, we want to keep track of videotapes and customers in a video store. Videos and customers are represented as entities Video and Customer, and the relationship "rents"

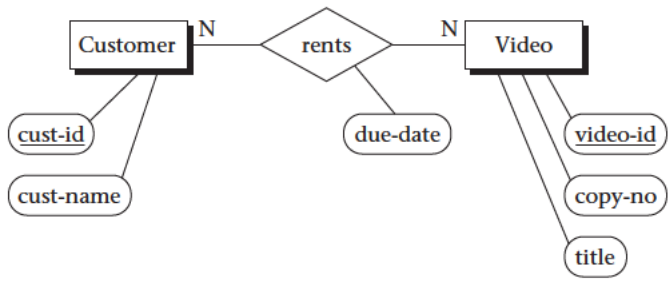


Figure 1.3 A simple form of ER model using the Chen notation

shows a many-to-many association between them. Both Video and Customer entities have a few attributes that describe their characteristics, and the relationship "rents" has an attribute due date that represents the date that a particular video rented by a specific customer must be returned.

From the database practitioner's standpoint, the simple form of the ER model (or UML) is the preferred form for both data modeling and end user verification. It is easy to learn and applicable to a wide variety of design problems that might be encountered in industry and small businesses. As we will demonstrate, the simple form can be easily translated into SQL data definitions, and thus it has an immediate use as an aid for database implementation.

The complex level of ER model definition includes concepts that go well beyond the simple model. It includes concepts from the semantic models of artificial intelligence and from competing conceptual data models. Data modeling at this level helps the database designer capture more semantics without having to resort to narrative explanations. It is also useful to the database application programmer, because certain integrity constraints defined in the ER model relate directly to code—code that checks range limits on data values and null values, for example. However, such detail in very large data model diagrams actually detracts from end user understanding. Therefore, the simple level is recommended as the basic communication tool for database design verification.

1.4 Summary

Knowledge of data modeling and database design techniques is important for database practitioners and application developers. The database life cycle shows the steps needed in a methodical approach to designing a database,, from logical design, which is independent of the system environment, to physical design, which is based on the details of the database management system chosen to implement the database. Among the variety of data modeling approaches, the ER and UML data models are arguably the most popular ones in use today, due to their simplicity and readability. A simple form of these models is used in most design tools; it is easy to learn and to apply to a variety of industrial and business applications. It is also a very useful tool for communicating with the end user about the conceptual model and for verifying the assumptions made in the modeling process. A more complex form, a superset of the simple form, is useful for the more experienced designer who wants to capture greater semantic detail in diagram form, while avoiding having to write long and tedious narrative to explain certain requirements and constraints.

1.5 Literature Summary

Much of the early data modeling work was done by Bachman [1969, 1972], Chen [1976], Senko et al. [1973], and others. Database design textbooks that adhere to a significant portion of the relational database life cycle described in this chapter are Teorey and Fry [1982], Muller [1999], Stephens and Plew [2000], Simsion and Witt [2001], and Hernandez and Getz [2003]. Temporal (time-varying) databases are defined and discussed in Jensen and Snodgrass [1996] and Snodgrass [2000]. Other well used approaches for conceptual data modeling include IDEF1X [Bruce, 1992; IDEF1X, 2005] and the data modeling component of the Zachmann Framework [Zachmann, 1987; Zachmann Institute for Framework Advancement, 2005]. Schema evolution during development, a frequently occurring problem, is addressed in Harriman, Hodgetts, and Leo [2004].

2 The Entity-Relationship Model

This chapter defines all the major entity-relationship (ER) concepts that can be applied to the conceptual data modeling phase of the database life cycle. In Section 2.1, we will look at the simple level of ER modeling described in the original work by Chen and extended by others. The simple form of the ER model is used as the basis for effective communication with the end user about the conceptual database. Section 2.2 presents the more advanced concepts; although they are less generally accepted, they are useful to describe certain semantics that cannot be constructed with the simple model.

2.1 Fundamental ER Constructs

2.1.1 Basic Objects: Entities, Relationships, Attributes

The basic ER model consists of three classes of objects: entities, relationships, and attributes.

Entities

Entities are the principal data objects about which information is to be collected; they usually denote a person, place, thing, or event of informational interest. A particular occurrence of an entity is called an *entity instance* or sometimes an *entity occurrence*. In our example, employee,

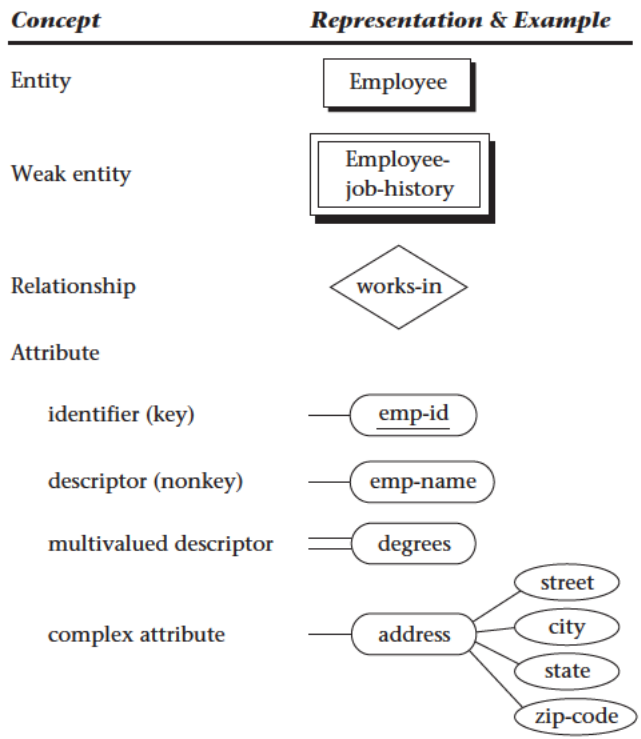


Figure 2.1 The basic ER model

department, division, project, skill, and location are all examples of entities. For easy reference, entity names will henceforth be capitalized throughout this text (e.g., Employee, Department, and so forth). The entity construct is a rectangle as depicted in Figure 2.1. The entity name is written inside the rectangle.

Relationships

Relationships represent real-world associations among one or more entities, and, as such, have no physical or conceptual existence other than that which depends upon their entity associations. Relationships are described in terms of degree, connectivity, and existence. These terms are defined in the sections that follow. The most common meaning associated with the term *relationship* is indicated by the connectivity between entity occurrences: one-to-one, one-to-many, and many-to-many. The relationship construct is a diamond that connects the associ-

ated entities, as shown in Figure 2.1. The relationship name can be written inside or just outside the diamond.

A *role* is the name of one end of a relationship when each end needs a distinct name for clarity of the relationship. In most of the examples given in Figure 2.2, role names are not required because the entity names combined with the relationship name clearly define the individual roles of each entity in the relationship. However, in some cases role names should be used to clarify ambiguities. For example, in the first case in Figure 2.2, the recursive binary relationship "manages" uses two roles, "manager" and "subordinate," to associate the proper connectivities with the two different roles of the single entity. Role names are typically nouns. In this diagram, one employee's role is to be the "manager" of up to *n* other employees. The other role is for particular "subordinates" to be managed by exactly one other employee.

Attributes

Attributes are characteristics of entities that provide descriptive detail about them. A particular occurrence of an attribute within an entity or relationship is called an attribute value. Attributes of an entity such as Employee may include emp-id, emp-name, emp-address, phone-no, fax-no, job-title, and so on. The attribute construct is an ellipse with the attribute name inside (or an oblong, as shown in Figure 2.1). The attribute is connected to the entity it characterizes.

There are two types of attributes: identifiers and descriptors. An identifier (or key) is used to uniquely determine an instance of an entity; a descriptor (or nonkey attribute) is used to specify a nonunique characteristic of a particular entity instance. Both identifiers and descriptors may consist of either a single attribute or some composite of attributes. For example, an identifier or key of Employee is emp-id, and a descriptor of Employee is emp-name or job-title. Key attributes are underlined in the ER diagram, as shown in Figure 2.1. We note, briefly, that you can have more than one identifier (key) for an entity, or you can have a set of attributes that compose a key (see Section 6.1.2).

Some attributes, such as specialty-area, may be multivalued. The notation for multivalued attributes is a double attachment line, as shown in Figure 2.1. Other attributes may be complex, such as an address that further subdivides into street, city, state, and zip code. Complex attributes are constructed to have attributes of their own; sometimes, however, the individual parts of a complex attribute are specified as individual attributes in the first place. Either form is reasonable in ER notation.

Entities have internal identifiers that uniquely determine the existence of entity instances, but weak entities derive their identity from the identifying attributes of one or more "parent" entities. Weak entities are often depicted with a double-bordered rectangle (see Figure 2.1), which denotes that all occurrences of that entity depend on an associated (strong) entity for their existence in the database. For example, in Figure 2.1, the weak entity Employee-job-history is related to the entity Employee and dependent upon Employee for its own existence.

2.1.2 Degree of a Relationship

The degree of a relationship is the number of entities associated in the relationship. Binary and ternary relationships are special cases where the degrees are 2 and 3, respectively. An *n*-ary relationship is the general form for any degree *n*. The notation for degree is illustrated in Figure 2.2. The binary relationship, an association between two entities, is by far the most common type in the natural world. In fact, many modeling systems use only this type. In Figure 2.2, we see many examples of the association of two entities in different ways: Department and Division, Department and Employee, Employee and Project, and so on. A binary recursive relationship (for example, "manages" in Figure 2.2) relates a particular Employee to another Employee by management. It is called recursive because the entity relates only to another instance of its own type. The binary recursive relationship construct is a diamond with both connections to the same entity.

A ternary relationship is an association among three entities. This type of relationship is required when binary relationships are not sufficient to accurately describe the semantics of the association. The ternary relationship construct is a single diamond connected to three entities, as shown in Figure 2.2. Sometimes a relationship is mistakenly modeled as ternary when it could be decomposed into two or three equivalent binary relationships. When this occurs, the ternary relationship should be eliminated to achieve both simplicity and semantic purity. Ternary relationships are discussed in greater detail in Section 2.2.3 and Chapter 6.

An entity may be involved in any number of relationships, and each relationship may be of any degree. Furthermore, two entities may have any number of binary relationships between them, and so on for any *n* entities (see *n*-ary relationships defined in Section 2.2.4).

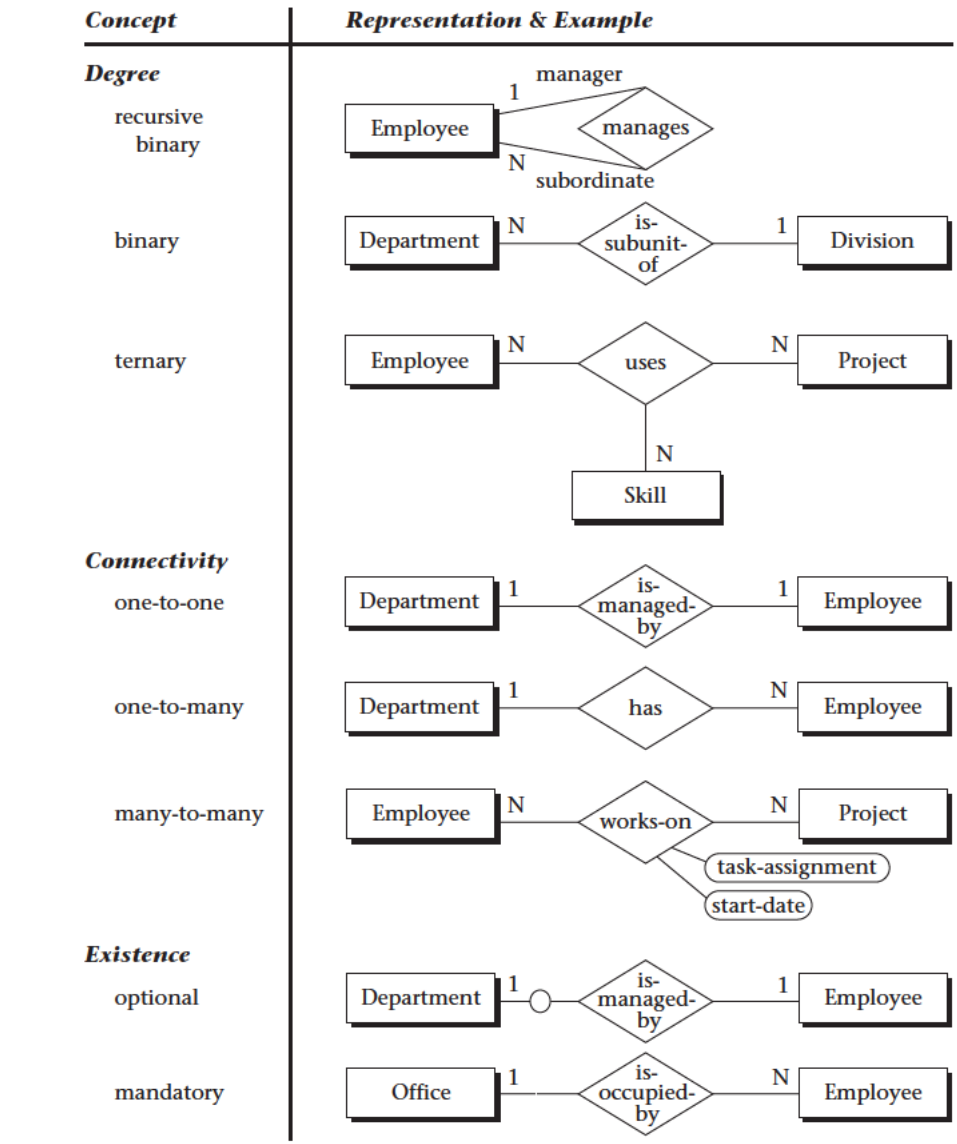


Figure 2.2 Degrees, connectivity, and attributes of a relationship

2.1.3 Connectivity of a Relationship

The *connectivity* of a relationship describes a constraint on the connection of the associated entity occurrences in the relationship. Values for connectivity are either "one" or "many." For a relationship between the entities Department and Employee, a connectivity of one for Department and many for Employee means that there is at most one entity occurrence of Department associated with many occurrences of Employee. The actual count of elements associated with the connectivity is called the *cardinality* of the relationship connectivity; it is used much less frequently than the connectivity constraint because the actual values are usually variable across instances of relationships. Note that there are no standard terms for the connectivity concept, so the reader is admonished to consider the definition of these terms carefully when using a particular database design methodology.

Figure 2.2 shows the basic constructs for connectivity for binary relationships: one-to-one, one-to-many, and many-to-many. On the "one" side, the number one is shown on the connection between the relationship and one of the entities, and on the "many" side, the letter *N* is used on the connection between the relationship and the entity to designate the concept of many.

In the one-to-one case, the entity Department is managed by exactly one Employee, and each Employee manages exactly one Department. Therefore, the minimum and maximum connectivities on the "is-managed-by" relationship are exactly one for both Department and Employee.

In the one-to-many case, the entity Department is associated with ("has") many Employees. The maximum connectivity is given on the Employee (many) side as the unknown value *N*, but the minimum connectivity is known as one. On the Department side the minimum and maximum connectivities are both one, that is, each Employee works within exactly one Department.

In the many-to-many case, a particular Employee may work on many Projects and each Project may have many Employees. We see that the maximum connectivity for Employee and Project is *N* in both directions, and the minimum connectivities are each defined (implied) as one.

Some situations, though rare, are such that the actual maximum connectivity is known. For example, a professional basketball team may be limited by conference rules to 12 players. In such a case, the number

12 could be placed next to an entity called "team members" on the many side of a relationship with an entity "team." Most situations, however, have variable connectivity on the many side, as shown in all the examples of Figure 2.2.

2.1.4 Attributes of a Relationship

Attributes can be assigned to certain types of relationships as well as to entities. An attribute of a many-to-many relationship, such as the "works-on" relationship between the entities Employee and Project (Figure 2.2), could be "task-assignment" or "start-date." In this case, a given task assignment or start date only has meaning when it is common to an instance of the assignment of a particular Employee to a particular Project via the relationship "works-on."

Attributes of relationships are typically assigned only to binary many-to-many relationships and to ternary relationships. They are not normally assigned to one-to-one or one-to-many relationships, because of potential ambiguities. For example, in the one-to-one binary relationship "is-managed-by" between Department and Employee, an attribute "start-date" could be applied to Department to designate the start date for that department. Alternatively, it could be applied to Employee as an attribute for each Employee instance, to designate the employee's start date as the manager of that department. If, instead, the relationship is many-to-many, so that an employee can manage many departments over time, then the attribute "start-date" must shift to the relationship, so each instance of the relationship that matches one employee with one department can have a unique start date for that employee as manager of that department.

2.1.5 Existence of an Entity in a Relationship

Existence of an entity occurrence in a relationship is defined as either mandatory or optional. If an occurrence of either the "one" or "many" side entity must always exist for the entity to be included in the relationship, then it is mandatory. When an occurrence of that entity need not always exist, it is considered optional. For example, in Figure 2.2 the entity Employee may or may not be the manager of any Department, thus making the entity Department in the "is-managed-by" relationship between Employee and Department optional.

Optional existence, defined by a zero on the connection line between an entity and a relationship, defines a minimum connectivity of zero. *Mandatory existence* defines a minimum connectivity of one. When existence is unknown, we assume the minimum connectivity is one (that is, mandatory).

Maximum connectivities are defined explicitly on the ER diagram as a constant (if a number is shown on the ER diagram next to an entity) or a variable (by default if no number is shown on the ER diagram next to an entity). For example, in Figure 2.2, the relationship "is-occupied-by" between the entity Office and Employee implies that an Office may house from zero to some variable maximum (*N*) number of Employees, but an Employee must be housed in exactly one Office, that is, mandatory.

Existence is often implicit in the real world. For example, an entity Employee associated with a dependent (weak) entity, Dependent, cannot be optional, but the weak entity is usually optional. Using the concept of optional existence, an entity instance may be able to exist in other relationships even though it is not participating in this particular relationship.

The term existence is also associated with identification of a data object. Many DBMSs provide unique identifiers for rows (Oracle ROWIDs, for example). Identifying an object such as a row can be done in an existence-based way. It can also be done in a value-based way by identifying the object (row) with the values of one or more attributes or columns of the table.

2.1.6 Alternative Conceptual Data Modeling Notations

At this point we need to digress briefly to look at other conceptual data modeling notations that are commonly used today and compare them with the Chen approach. A popular alternative form for one-to-many and many-to-many relationships uses "crow's-foot" notation for the "many" side (see Figure 2.3a). This form is used by some CASE tools, such as Knowledgeware's Information Engineering Workbench (IEW). Relationships have no explicit construct but are implied by the connection line between entities and a relationship name on the connection line. Minimum connectivity is specified by either a 0 (for zero) or perpendicular line (for one) on the connection lines between entities. The term *intersection entity* is used to designate a weak entity, especially an entity that is equivalent to a many-to-many relationship. Another popular form

used today is the IDEFIX notation [IDEF1X, 2005], conceived by Robert G. Brown [Bruce, 1992]. The similarities with the Chen notation are obvious in Figure 2.3b. Fortunately, any of these forms is reasonably easy to learn and read, and their equivalence with the basic ER concepts is obvious from the diagrams. Without a clear standard for the ER model, however, many other constructs are being used today in addition to the three types shown here.

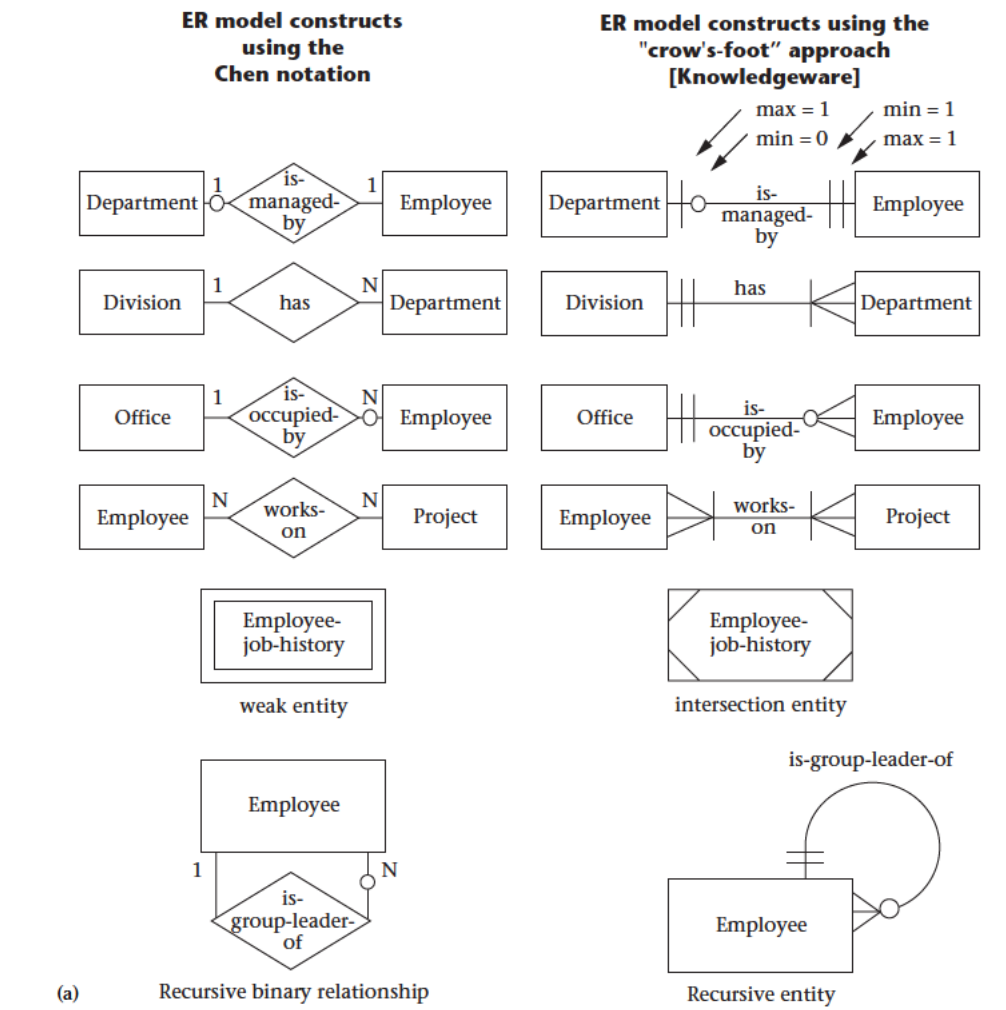


Figure 2.3 Conceptual data modeling notations

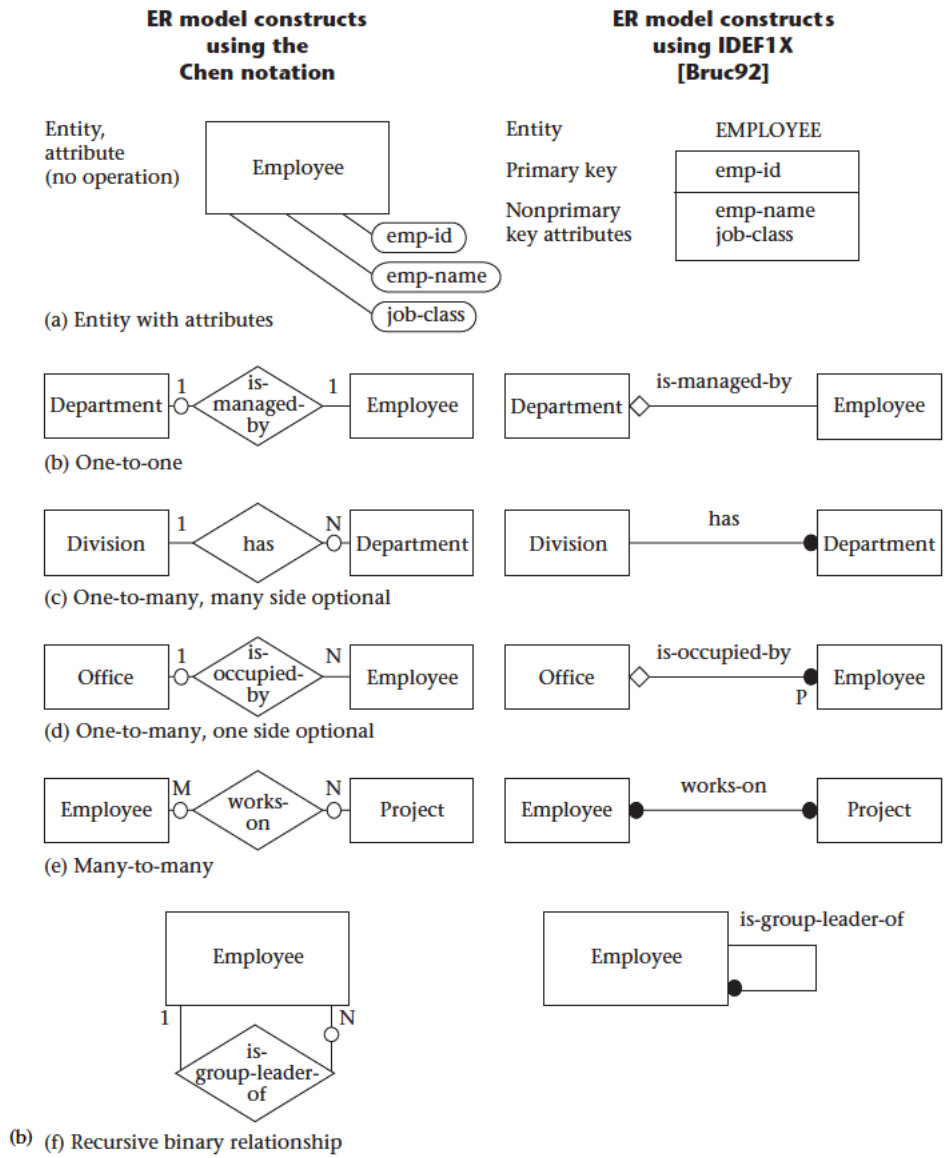
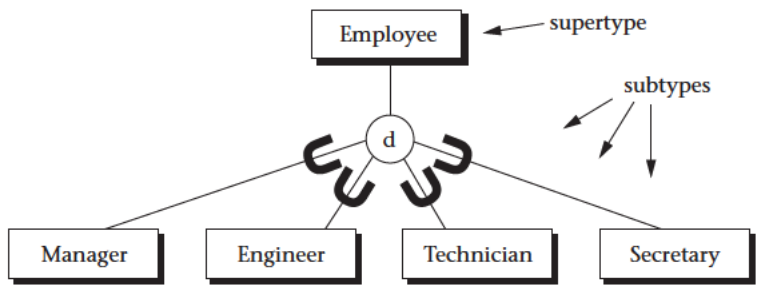


Figure 2.3 *(continued)*

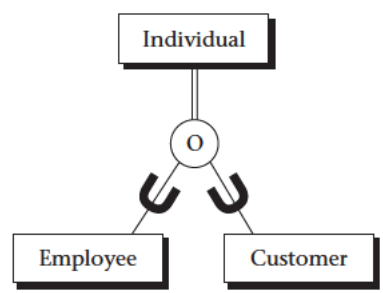
2.2 Advanced ER Constructs

2.2.1 Generalization: Supertypes and Subtypes

The original ER model has been effectively used for communicating fundamental data and relationship definitions with the end user for a long time. However, using it to develop and integrate conceptual models with different end user views was severely limited until it could be extended to include database abstraction concepts such as *generalization*. The generalization relationship specifies that several types of entities with certain common attributes can be generalized into a higher-level entity type—a generic or superclass entity, more commonly known as a *supertype* entity. The lower levels of entities—*subtypes* in a generalization hierarchy—can be either disjoint or overlapping subsets of the supertype entity. As an example, in Figure 2.4 the entity Employee is a higher-level



(a) Generalization with disjoint subtypes



(b) Generalization with overlapping subtypes and completeness constraint

Figure 2.4 Supertypes and subtypes

abstraction of Manager, Engineer, Technician, and Secretary—all of which are disjoint types of Employee. The ER model construct for the generalization abstraction is the connection of a supertype entity with its subtypes, using a circle and the subset symbol on the connecting lines from the circle to the subtype entities. The circle contains a letter specifying a disjointness constraint (see the following discussion). *Specialization*, the reverse of generalization, is an inversion of the same concept; it indicates that subtypes specialize the supertype.

A supertype entity in one relationship may be a subtype entity in another relationship. When a structure comprises a combination of supertype/subtype relationships, that structure is called a *supertype/subtype hierarchy* or *generalization hierarchy*. Generalization can also be described in terms of inheritance, which specifies that all the attributes of a supertype are propagated down the hierarchy to entities of a lower type. Generalization may occur when a generic entity, which we call the supertype entity, is partitioned by different values of a common attribute. For example, in Figure 2.4 the entity Employee is a generalization of Manager, Engineer, Technician, and Secretary over the attribute "job-title" in Employee.

Generalization can be further classified by two important constraints on the subtype entities: *disjointness* and *completeness*. The disjointness constraint requires the subtype entities to be mutually exclusive. We denote this type of constraint by the letter "d" written inside the generalization circle (Figure 2.4a). Subtypes that are not disjoint (i.e., that overlap) are designated by using the letter "o" inside the circle. As an example, the supertype entity Individual has two subtype entities, Employee and Customer; these subtypes could be described as overlapping, or not mutually exclusive (Figure 2.4b). Regardless of whether the subtypes are disjoint or overlapping, they may have additional special attributes in addition to the generic (inherited) attributes from the supertype.

The completeness constraint requires the subtypes to be all-inclusive of the supertype. Thus, subtypes can be defined as either total or partial coverage of the supertype. For example, in a generalization hierarchy with supertype Individual and subtypes Employee and Customer, the subtypes may be described as all-inclusive or total. We denote this type of constraint by a double line between the supertype entity and the circle. This is indicated in Figure 2.4b, which implies that the only types of individuals to be considered in the database are employees and customers.

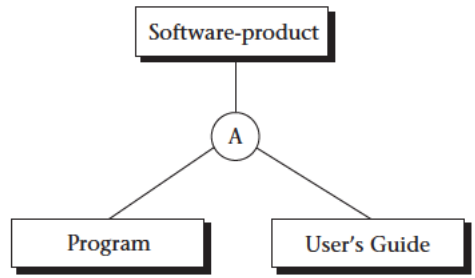


Figure 2.5 Aggregation

2.2.2 Aggregation

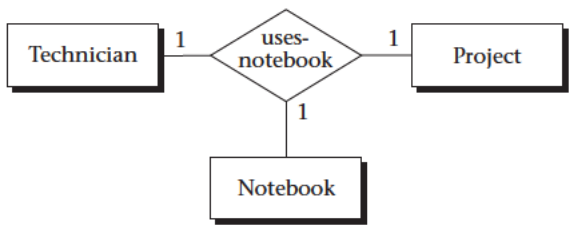
Aggregation is a form of abstraction between a supertype and subtype entity that is significantly different from the generalization abstraction. Generalization is often described in terms of an "is-a" relationship between the subtype and the supertype—for example, an Employee is an Individual. Aggregation, on the other hand, is the relationship between the whole and its parts and is described as a "part-of" relationship—for example, a report and a prototype software package are both parts of a deliverable for a contract. Thus, in Figure 2.5, the entity Software-product is seen to consist of component parts Program and User's Guide. The construct for aggregation is similar to generalization, in that the supertype entity is connected with the subtype entities with a circle; in this case, the letter "A" is shown in the circle. However, there are no subset symbols because the "part-of" relationship is not a subset. Furthermore, there are no inherited attributes in aggregation; each entity has its own unique set of attributes.

2.2.3 Ternary Relationships

Ternary relationships are required when binary relationships are not sufficient to accurately describe the semantics of an association among three entities. Ternary relationships are somewhat more complex than binary relationships, however. The ER notation for a ternary relationship is shown in Figure 2.6 with three entities attached to a single relationship diamond, and the connectivity of each entity is designated as either "one" or "many." An entity in a ternary relationship is considered to be "one" if only one instance of it can be associated with one instance of

each of the other two associated entities. It is "many" if more than one instance of it can be associated with one instance of each of the other two associated entities. In either case, it is assumed that one instance of each of the other entities is given.

As an example, the relationship "manages" in Figure 2.6c associates the entities Manager, Engineer, and Project. The entities Engineer and

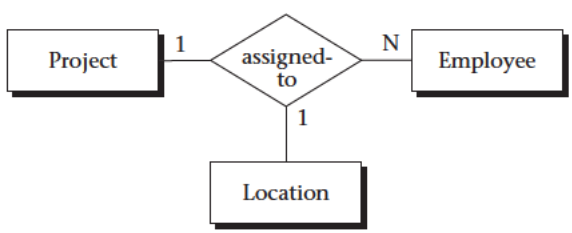


A technician uses exactly one notebook for each project. Each notebook belongs to one technician for each project. Note that a technician may still work on many projects and maintain different notebooks for different projects.

Functional dependencies

emp-id, project-name → notebook-no
emp-id, notebook-no → project-name
project-name, notebook-no → emp-id

(a) One-to-one-to-one ternary relationship



Each employee assigned to a project works at only one location for that project, but can be at different locations for different projects. At a particular location, an employee works on only one project. At a particular location, there can be many employees assigned to a given project.

Functional dependencies

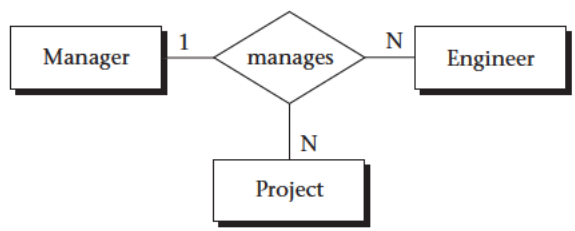
emp-id, loc-name → project-name
emp-id, project-name → loc-name

(b) One-to-one-to-many ternary relationship

Figure 2.6 Ternary relationships

Project are considered "many;" the entity Manager is considered "one." This is represented by the following assertions.

- **Assertion 1:** One engineer, working under one manager, could be working on many projects.
- **Assertion 2:** One project, under the direction of one manager, could have many engineers.
- **Assertion 3:** One engineer, working on one project, must have only a single manager

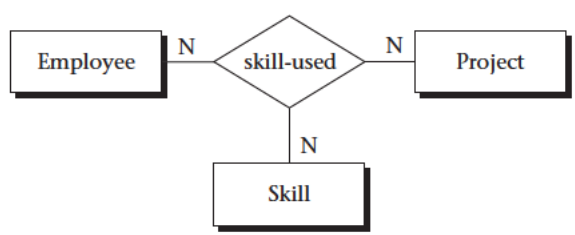


Each engineer working on a particular project has exactly one manager, but each manager of a project may manage many engineers, and each manager of an engineer may manage that engineer on many projects.

Functional dependency

project-name, emp-id → mgr-id

(c) One-to-many-to-many ternary relationship



Employees can use many skills on any one of many projects, and each project has many employees with various skills.

Functional dependencies

None

(d) Many-to-many-to-many ternary relationship

Figure 2.6 *(continued)*

Assertion 3 could also be written in another form, using an arrow (->) in a kind of shorthand called a *functional dependency*. For example:

```
emp-id, project-name -> mgr-id
```

where emp-id is the key (unique identifier) associated with the entity Engineer, project-name is the key associated with the entity Project, and mgr-id is the key of the entity Manager. In general, for an *n*-ary relationship, each entity considered to be a "one" has its key appearing on the right side of exactly one functional dependency (FD). No entity considered "many" ever has its key appear on the right side of an FD.

All four forms of ternary relationships are illustrated in Figure 2.6. In each case, the number of "one" entities implies the number of FDs used to define the relationship semantics, and the key of each "one" entity appears on the right side of exactly one FD for that relationship.

Ternary relationships can have attributes in the same way that many-to-many binary relationships can. The values of these attributes are uniquely determined by some combination of the keys of the entities associated with the relationship. For example, in Figure 2.6d the relationship "skill-used" might have the attribute "tool" associated with a given employee using a particular skill on a certain project, indicating that a value for tool is uniquely determined by the combination of employee, skill, and project.

2.2.4 General *n*-ary Relationships

Generalizing the ternary form to higher-degree relationships, an *n*-ary relationship that describes some association among *n* entities is represented by a single relationship diamond with *n* connections, one to each

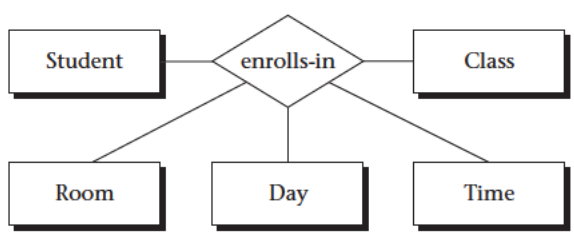


Figure 2.7 *n*-ary relationships.

entity (see Figure 2.7). The meaning of this form can best be described in terms of the functional dependencies among the keys of the n associated entities. There can be anywhere from zero to n FDs, depending on the number of "one" entities. The collection of FDs that describe an n-ary relationship must have n components: $n - 1$ on the left side (determinant) and 1 on the right side. A ternary relationship ($n = 3$), for example, has two components on the left and one on the right, as we saw in the example in Figure 2.6. In a more complex database, other types of FDs may also exist within an n-ary relationship. When this occurs, the ER model does not provide enough semantics on its own, and it must be supplemented with a narrative description of these dependencies.

2.2.5 Exclusion Constraint

The normal, or default, treatment of multiple relationships is the *inclusive OR*, which allows any or all of the entities to participate. In some situations, however, multiple relationships may be affected by the *exclusive OR* (exclusion) constraint, which allows at most one entity instance among several entity types to participate in the relationship with a single root entity. For example, in Figure 2.8, suppose the root entity Work-task has two associated entities, External-project and Internal-project. At most one of the associated entity instances could apply to an instance of Work-task.

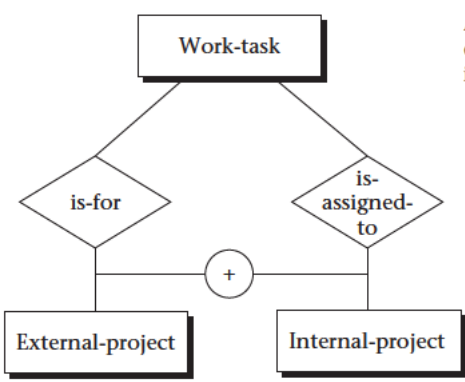


Figure 2.8 Exclusion constraint

2.2.6 Referential Integrity

We note that a foreign key is an attribute of a table (not necessarily a key of any kind) that relates to a key in another table. *Referential integrity* requires that for every foreign key instance that exists in a table, the row (and thus the key instance) of the parent table associated with that foreign key instance must also exist. The referential integrity constraint has become integral to relational database design and is usually implied as requirements for the resulting relational database implementation. (Chapter 5 discusses the SQL implementation of referential integrity constraints.)

2.3 Summary

The basic concepts of the ER model and their constructs are described in this chapter. An entity is a person, place, thing, or event of informational interest. Attributes are objects that provide descriptive information about entities. Attributes may be unique identifiers or nonunique descriptors. Relationships describe the connectivity between entity instances: one-to-one, one-to-many, or many-to-many. The degree of a relationship is the number of associated entities: two (binary), three (ternary), or any n (n-ary). The role (name), or relationship name, defines the function of an entity in a relationship.

The concept of existence in a relationship determines whether an entity instance must exist (mandatory) or not (optional). So, for example, the minimum connectivity of a binary relationship—that is, the number of entity instances on one side that are associated with one instance on the other side—can either be zero, if optional, or one, if mandatory. The concept of generalization allows for the implementation of supertype and subtype abstractions.

The more advanced constructs in ER diagrams are sporadically used and have no generally accepted form as yet. They include ternary relationships, which we define in terms of the FD concept of relational databases; constraints on exclusion; and the implicit constraints from the relational model, such as referential integrity.

2.4 Literature Summary

Most of the notation in this chapter is from Chen's original ER definition [Chen, 1976]. The concept of data abstraction was first proposed by Smith and Smith [1977] and applied to the ER model by Scheuermann, Scheffner, and Weber [1980], Elmasri and Navathe [2003], Bruce [1992], IDEF1X [2005], among others. The application of the semantic network model to conceptual schema design was shown by Bachman [1977], McKleod and King [1979], Hull and King [1987], and Peckham and Maryanski [1988].

3 The Unified Modeling Language (UML)

The Unified Modeling Language (UML) is a graphical language for communicating design specifications for software. The object-oriented software development community created UML to meet the special needs of describing object-oriented software design. UML has grown into a standard for the design of digital systems in general.

There are a number of different types of UML diagrams serving various purposes [Rumb05]. The *class* and *activity* diagram types are particularly useful for discussing database design issues. UML class diagrams capture the structural aspects found in database schemas. UML activity diagrams facilitate discussion on the dynamic processes involved in database design. This chapter is an overview of the syntax and semantics of the UML class and activity diagram constructs used in this book. These same concepts are useful for planning, documenting, discussing and implementing databases. We are using UML 2.0, although for the purposes of the class diagrams and activity diagrams shown in this book, if you are familiar with UML 1.4 or 1.5 you will probably not see any differences.

UML class diagrams and entity-relationship (ER) models [Chen, 1976; Chen, 1987] are similar in both form and semantics. The original creators of UML point out the influence of ER models on the origins of class diagrams [Rumbaugh, Jacobson, and Booch, 2005]. The influence of UML has in turn affected the database community. Class diagrams

now appear frequently in the database literature to describe database schemas.

UML activity diagrams are similar in purpose to flowcharts. Processes are partitioned into constituent activities along with control flow specifications.

This chapter is organized into three sections. Section 3.1 presents class diagram notation, along with examples. Section 3.2 covers activity diagram notation, along with illustrative examples. Section 3.3 concludes with a few rules of thumb for UML usage.

3.1 Class Diagrams

A *class* is a descriptor for a set of *objects* that share some attributes and/or operations. We conceptualize classes of objects in our everyday lives. For example, a car has attributes, such as vehicle identification number (VIN) and mileage. A car also has operations, such as accelerate and brake. All cars have these attributes and operations. Individual cars differ in the details. A given car has its own values for VIN and mileage. For example, a given car might have VIN 1NXBR32ES3Z126369 and a mileage of 22,137 miles. Individual cars are objects that are instances of the class "Car."

Classes and objects are a natural way of conceptualizing the world around us. The concepts of classes and objects are also the paradigms that form the foundation of object-oriented programming. The development of object-oriented programming led to the need for a language to describe object-oriented design, giving rise to UML.

There is a close correspondence between class diagrams in UML and ER diagrams. Classes are analogous to entities. Database schemas can be diagrammed using UML. It is possible to conceptualize a database table as a class. The columns in the table are the attributes, and the rows are objects of that class. For example, we could have a table named "Car" with columns named "vin" and "mileage." Each row in the table would have values for these columns, representing an individual car. A given car might be represented by a row with the value "1NXBR32ES3Z126369" in the vin column, and 22,137 in the mileage column.

The major difference between classes and entities is the lack of operations in entities. Note that the term *operation* is used here in the UML sense of the word. Stored procedures, functions, triggers, and constraints are forms of named behavior that can be defined in databases; however

these are not associated with the behavior of individual rows. The term *operations* in UML refers to the methods inherent in classes of objects. These behaviors are not stored in the definition of rows within the database. There are no operations named "accelerate" or "brake" associated with rows in our "Car" table. Classes can be shown with attributes and no operations in UML, which is the typical usage for database schemas.

3.1.1 Basic Class Diagram Notation

The top of Figure 3.1 illustrates the UML syntax for a class, showing both *attributes* and *operations*. It is also possible to include user-defined named compartments, such as responsibilities. We will focus on the class name, attributes, and operations compartments. The UML icon for a class is a rectangle. When the class is shown with attributes and operations, the rectangle is subdivided into three horizontal compartments. The top compartment contains the class name, centered in bold face, beginning with a capital letter. Typically, class names are nouns. The middle compartment contains attribute names, left-justified in regular face, beginning with a lowercase letter. The bottom compartment contains operation names, left-justified in regular face, beginning with a lowercase letter, ending with a parenthetical expression. The parentheses may contain arguments for the operation.

The class notation has some variations reflecting emphasis. Classes can be written without the attribute compartment and/or the operations compartment. Operations are important in software. If the software designer wishes to focus on the operations, the class can be shown with only the class name and operations compartments. Showing operations and hiding attributes is a very common syntax used by software designers. Database designers, on the other hand, do not generally deal with class operations; however, the attributes are of paramount importance. The needs of the database designer can be met by writing the class with only the class name and attribute compartments showing. Hiding operations and showing attributes is an uncommon syntax for a software designer, but it is common for database design. Lastly, in high-level diagrams, it is often desirable to illustrate the relationships of the classes without becoming entangled in the details of the attributes and operations. Classes can be written with just the class name compartment when simplicity is desired.

Various types of relationships may exist between classes. *Associations* are one type of relationship. The most generic form of association is drawn with a line connecting two classes. For example, in Figure 3.1

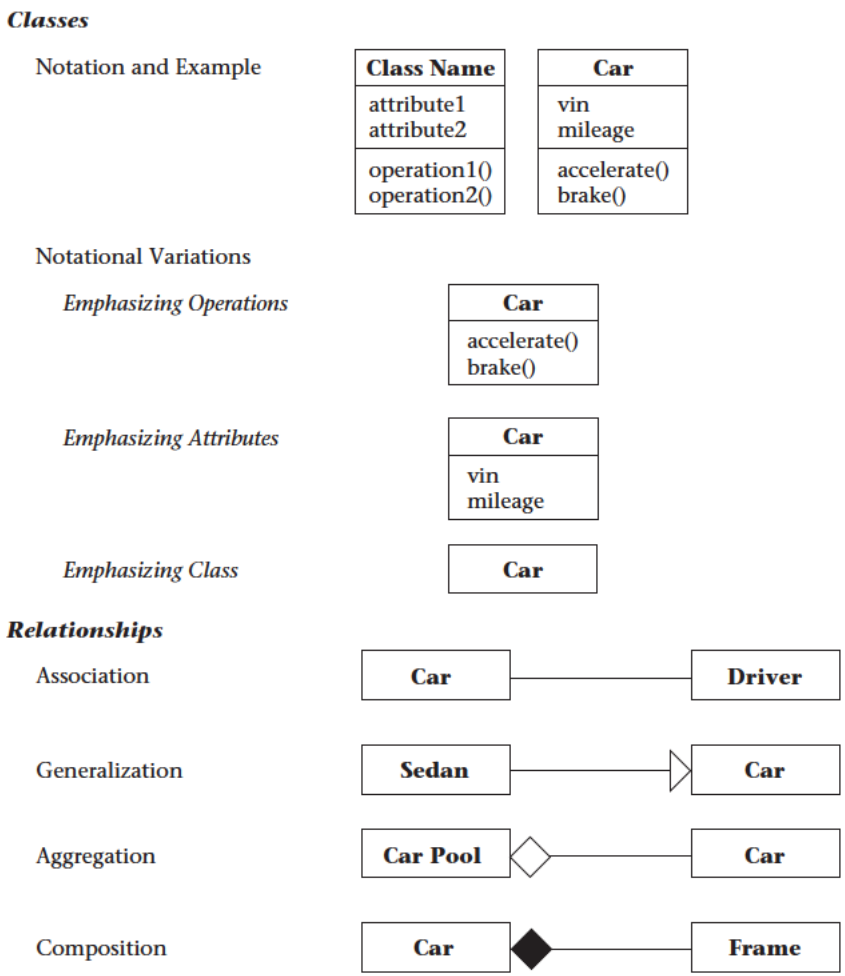


Figure 3.1 Basic UML class diagram constructs

there is an association between the class "Car" and the class named "Driver."

A few types of associations, such as *aggregation* and *composition*, are very common. UML has designated symbols for these associations. Aggregation indicates "part of" associations, where the parts have an independent existence. For example, a Car may be part of a Car Pool. The Car also exists on its own, independent of any Car Pool. Another distinguishing feature of aggregation is that the part may be shared

among multiple objects. For example, a Car may belong to more than one Car Pool. The aggregation association is indicated with a hollow diamond attached to the class that holds the parts. Figure 3.1 indicates that a Car Pool aggregates Cars.

Composition is another "part of" association in which the parts are strictly owned, not shared. For example, a Frame is part of a single Car. The notation for composition is an association adorned with a solid black diamond attached to the class that owns the parts. Figure 3.1 indicates that a Frame is part of the composition of a Car.

Generalization is another common relationship. For example, Sedan is a type of car. The "Car" class is more general than the "Sedan" class. Generalization is indicated by a solid line adorned with a hollow arrowhead pointing to the more general class. Figure 3.1 shows generalization from the Sedan class to the Car class.

3.1.2 Class Diagrams for Database Design

The reader may be interested in the similarities and differences between UML class diagrams and ER models. Figures 3.2 through 3.5 parallel some of the figures in Chapter 2, allowing for easy comparison. We then turn our attention to capturing primary key information in Figure 3.6. We conclude this section with an example database schema of the music industry, illustrated by Figures 3.7 through 3.10.

Figure 3.2 illustrates UML constructs for relationships with various degrees of association and multiplicities. These examples are parallel to the ER models shown in Figure 2.2. You may refer back to Figure 2.2 if you wish to contrast UML constructs with ER constructs.

Associations between classes may be reflexive, binary or *n*-ary. *Reflexive association* is a term we are carrying over from ER modeling. It is not a term defined in UML, although it is worth discussing. Reflexive association relates a class to itself. The reflexive association in Figure 3.2 means an Employee in the role of manager is associated with many managed Employees. The roles of classes in a relationship may be indicated at the ends of the relationship. The number of objects involved in the relationship, referred to as *multiplicity*, may also be specified at the ends of the relationship. An asterisk indicates that many objects take part in the association at that end of the relationship. The multiplicities of the reflexive association example in Figure 3.2 indicate that an Employee is associated with one manager, and a manager is associated with many managed Employees.

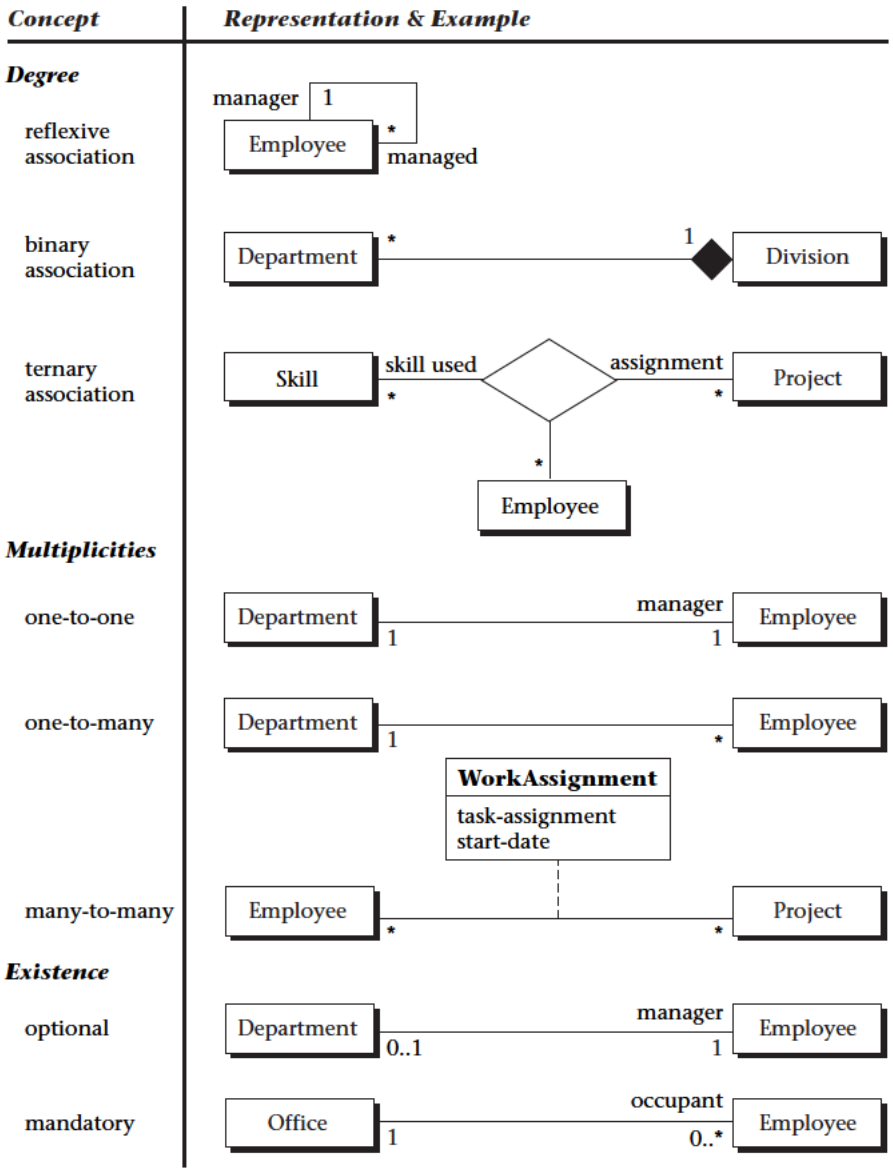


Figure 3.2 Selected UML relationship types (parallel to Figure 2.2)

A binary association is a relationship between two classes. For example, one Division has many Departments. Notice the solid black diamond at the Division end of the relationship. The solid diamond is an adorn-

ment to the associations that indicates composition. The Division is composed of Departments.

The ternary relationship in Figure 3.2 is an example of an *n*-ary association—an association that relates three or more classes. All classes partaking in the association are connected to a hollow diamond. Roles and/or multiplicities are optionally indicated at the ends of the *n*-ary association. Each end of the ternary association example in Figure 3.2 is marked with an asterisk, signifying many. The meaning of each multiplicity is isolated from the other multiplicities. Given a class, if you have exactly one object from every other class in the association, the multiplicity is the number of associated objects for the given class. One Employee working on one Project assignment uses many Skills. One Employee uses one Skill on many Project assignments. One Skill used on one Project is fulfilled by many Employees.

The next three class diagrams in Figure 3.2 show various combinations of multiplicities. The illustrated one-to-one association specifies that each Department is associated with exactly one Employee acting in the role of manager, and each manager is associated with exactly one Department. The diagram with the one-to-many association means that each Department has many Employees, and each Employee belongs to exactly one Department.

The many-to-many example in Figure 3.2 means each Employee associates with many Projects, and each Project associates with many Employees. This example also illustrates the use of an association class. If an association has attributes, these are written in a class that is attached to the association with a dashed line. The association class named "WorkAssignment" in Figure 3.2 contains two association attributes named "task-assignment" and "start-date." The association and the class together form an association class.

Multiplicity can be a range of integers, written with the minimum and maximum values separated by two periods. The asterisk by itself carries the same meaning as the range [0 .. *]. Also, if the minimum and maximum are the same number, then the multiplicity can be written as a single number. For example, [1 .. 1] means the same as [1]. Optional existence can be specified using a zero. The [0 .. 1] in the optional existence example of Figure 3.2 means an Employee in the role of manager is associated with either no Department (e.g., is upper management), or one Department.

Mandatory existence is specified whenever a multiplicity begins with a positive integer. The example of mandatory existence in Figure 3.2 means an Employee is an occupant of exactly one Office. One end of an

association can indicate mandatory existence, while the other end may use optional existence. This is the case in the example, where an Office may have any number of occupants, including zero.

Generalization is another type of relationship; a superclass is a generalization of a subclass. Specialization is the opposite of generalization; a subclass is a specialization of the superclass. The generalization relationship in UML is written with a hollow arrow pointing from the subclass to the generalized superclass. The top example in Figure 3.3 shows four subclasses: Manager, Engineer, Technician, and Secretary. These four subclasses are all specializations of the more general superclass Employee; that is, Managers, Engineers, Technicians, and Secretaries are types of Employees.

Notice the four relationships share a common arrowhead. Semantically, these are still four separate relationships. The sharing of the arrowhead is permissible in UML, to improve the clarity of the diagrams.

The bottom example in Figure 3.3 illustrates that a class can act as both a subclass in one relationship, and a superclass in another relation-

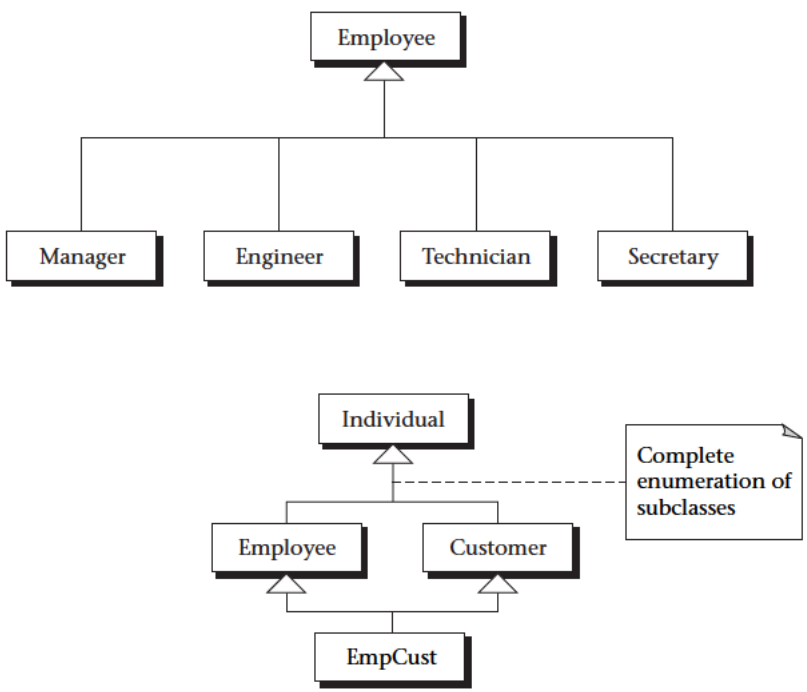


Figure 3.3 UML generalization constructs (parallel to Figure 2.4)

ship. The class named Individual is a generalization of the Employee and Customer classes. The Employee and Customer classes are in turn superclasses of the EmpCust class. A class can be a subclass in more than one generalization relationship. The meaning in the example is that an EmpCust object is both an Employee and a Customer.

You may occasionally find that UML doesn't supply a standard symbol for what you are attempting to communicate. UML incorporates some extensibility to accommodate user needs, such as a *note*. A note in UML is written as a rectangle with a dog-eared upper right corner. The note can attach to the pertinent element(s) with a dashed line(s). Write briefly in the note what you wish to convey. The bottom diagram in Figure 3.3 illustrates a note, which describes the Employee and Customer classes as the "Complete enumeration of subclasses."

The distinction between composition and aggregation is sometimes elusive for those new to UML. Figure 3.4 shows an example of each, to help clarify. The top diagram means that a Program and Electronic Documentation both contribute to the composition of a Software Product. The composition signifies that the parts do not exist without the Software Product (there is no software pirating in our ideal world). The bottom diagram specifies that a Teacher and a Textbook are aggregated by a course. The aggregation signifies that the Teacher and the Textbook are part of the Course, but they also exist separately. If a Course is canceled, the Teacher and the Textbook continue to exist.

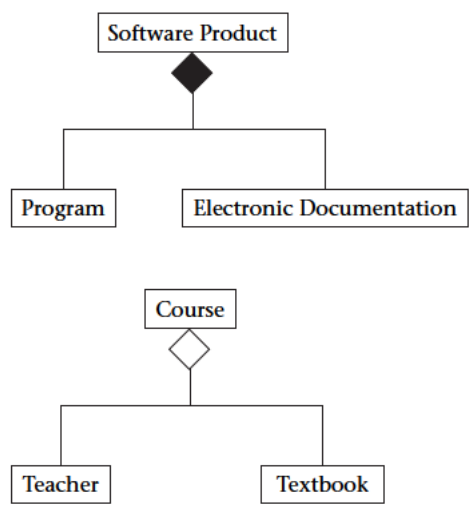


Figure 3.4 UML aggregation constructs (parallel to Figure 2.5)

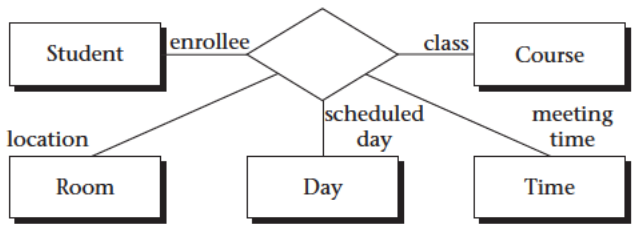


Figure 3.5 UML *n*-ary relationship (parallel to Figure 2.7)

Figure 3.5 illustrates another example of a *n*-ary relationship. The *n*-ary relationship may be clarified by specifying roles next to the participating classes. A Student is an enrollee in a class, associated with a given Room location, scheduled Day, and meeting Time.

The concept of a primary key arises in the context of database design. Often, each row of a table is uniquely identified by the values contained in one or more columns designated as the primary key. Objects in software are not typically identified in this fashion. As a result, UML does not have an icon representing a primary key. However, UML is extensible. The meaning of an element in UML may be extended

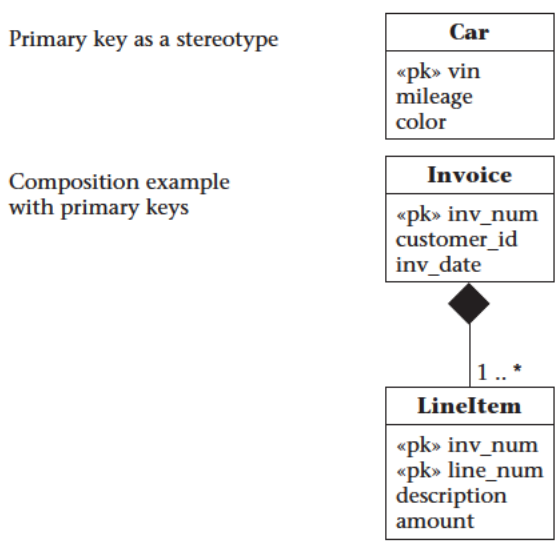


Figure 3.6 UML constructs illustrating primary keys

with a *stereotype*. Stereotypes are depicted with a short natural language word or phrase, enclosed in guillemets: « and ». We take advantage of this extensibility, using a stereotype «pk» to designate primary key attributes. Figure 3.6 illustrates the stereotype mechanism. The vin attribute is specified as the primary key for Cars. This means that a given vin identifies a specific Car. A noteworthy rule of thumb for primary keys: when a composition relationship exists, the primary key of the part includes the primary key of the owning object. The second diagram in Figure 3.6 illustrates this point.

3.1.3 Example from the Music Industry

Large database schemas may be introduced with high-level diagrams. Details can be broken out in additional diagrams. The overall goal is to present ideas in a clear, organized fashion. UML offers notational variations and organizational mechanism. You will sometimes find that there are multiple ways of representing the same material in UML. The decisions you make with regard to your representation depend in part on your purpose for a given diagram. Figures 3.7 through 3.10 illustrate some of the possibilities, with an example drawn from the music industry.

Packages may be used to organize classes into groups. Packages may themselves also be grouped into packages. The goal of using packages is to make the overall design of a system more comprehensible. One use for packages is to represent a schema. You can then show multiple schemas concisely. Another use for packages is to group related classes together within a schema, and present the schema clearly. Given a set of classes, different people may conceptualize different groupings. The division is a design decision, with no right or wrong answer. Whatever decisions are made, the result should enhance readability. The notation for a package is a folder icon, and the contents of a package can be optionally shown in the body of the folder. If the contents are shown, then the name of the package is placed in the tab. If the contents are elided, then the name of the package is placed in the body of the icon.

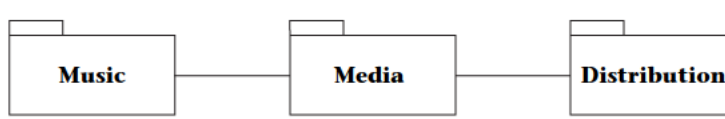


Figure 3.7 Example of related packages

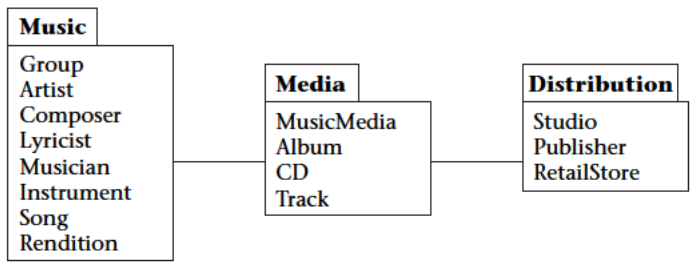


Figure 3.8 Example illustrating classes grouped into packages

If the purpose is to illustrate the relationships of the packages, and the classes are not important at the moment, then it is better to illustrate with the contents elided. Figure 3.7 illustrates the notation with the music industry example at a very high level. Music is created and placed on Media. The Media is then Distributed. There is an association between Music and Media, and between Media and Distribution.

Let us look at the organization of the classes. The music industry is illustrated in Figure 3.8 with the classes listed. The Music package contains classes that are responsible for creating the music. Examples of Groups are the Beatles and the Bangles. Sarah McLachlan and Sting are Artists. Groups and Artists are involved in creating the music. We will look shortly at the other classes and how they are related. The Media

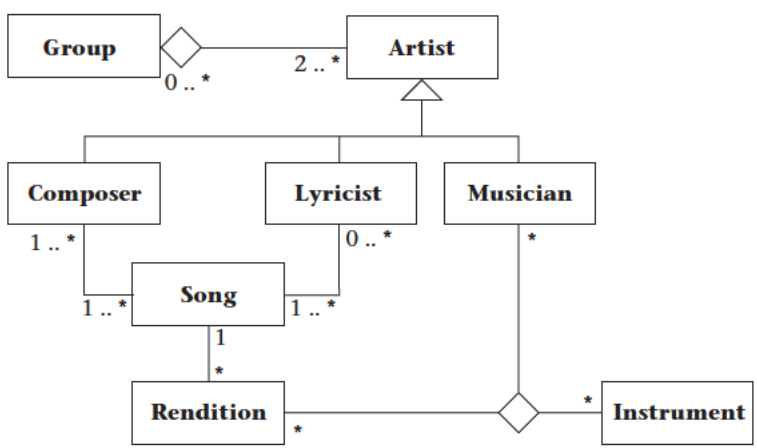


Figure 3.9 Relationships between classes in the music package

package contains classes that physically hold the recordings of the music. The Distribution package contains classes that bring the media to you.

The contents of a package can be expanded into greater detail. The relationships of the classes within the Music package are illustrated in Figure 3.9. A Group is an aggregation of two or more Artists. As indicated by the multiplicity between Artist and Group, [0 .. *], an Artist may or may not be in a Group, and may be in more than one Group. Composers, Lyricists, and Musicians are different types of Artists. A Song is associated with one or more Composers. A Song may not have any Lyricist, or any number of Lyricists. A Song may have any number of Renditions. A Rendition is associated with exactly one Song. A Rendition is associated with Musicians and Instruments. A given Musician-Instrument combination is associated with any number of Renditions. A specific Rendition-Musician combination may be associated with any number of Instruments. A given Rendition-Instrument combination is associated with any number of Musicians.

A system can be understood more easily by shifting focus to each package in turn. We turn our attention now to the classes and relationships in the Media package, shown in Figure 3.10. The associated classes from the Music and Distribution packages are also shown, detailing how the Media package is related to the other two packages. The Music Media

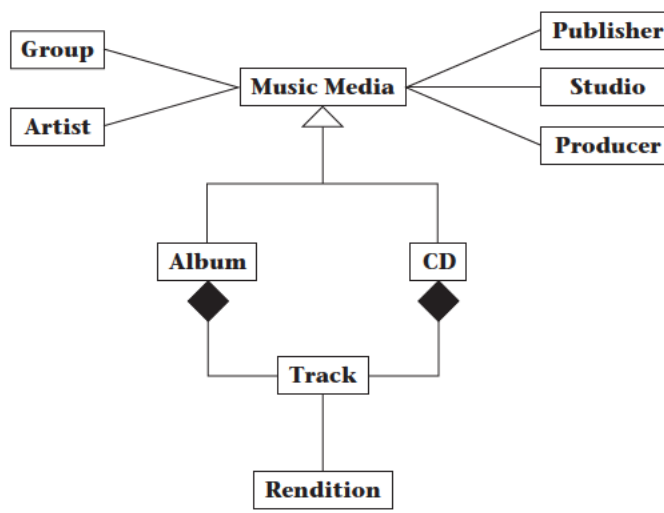


Figure 3.10 Classes of the media package and related classes

is associated with the Group and Artist classes, which are contained in the Music package shown in Figure 3.8. The Music Media is also associated with the Publisher, Studio, and Producer classes, which are contained in the Distribution package shown in Figure 3.8. Albums and CDs are types of Music Media. Albums and CDs are both composed of Tracks. Tracks are associated with Renditions.

3.2 Activity Diagrams

UML has a full suite of diagram types, each of which fulfills a need for describing a view of the design. UML *activity diagrams* are used to specify the activities and the flow of control in a process. The process may be a workflow followed by people, organizations, or other physical things. Alternatively, the process may be an algorithm implemented in software. The syntax and the semantics of the UML constructs are the same, regardless of the process described. Our examples draw from workflows that are followed by people and organizations, since these are more useful for the logical design of databases.

3.2.1 Activity Diagram Notation Description

Activity diagrams include notation for nodes, control flow, and organization. The icons we are describing here are outlined in Figure 3.11. The notation is further clarified by example in Section 3.2.2.

The nodes include *initial node*, *final node*, and *activity node*. Any process begins with control residing in the initial node, represented as a solid black circle. The process terminates when control reaches a final node, represented as a solid black circle surrounded by a concentric circle (i.e., a bull's-eye). Activity nodes are states where specified work is processed. For example, an activity might be named "Generate quote." The name of an activity is typically a descriptive verb or short verb phrase, written inside a lozenge shape. Control resides in an activity until that activity is completed. Then control follows the outgoing flow.

Control flow icons include *flows*, *decisions*, *forks*, and *joins*. A flow is drawn with an arrow. Control flows in the direction of the arrow. Decision nodes are drawn as a hollow diamond with multiple outgoing flows. Each flow from a decision node must have a *guard condition*. A guard condition is written in brackets next to the flow. Control flows in exactly one direction from a decision node, and only follows a flow if the guard condition is true. The guard conditions associated with a deci-

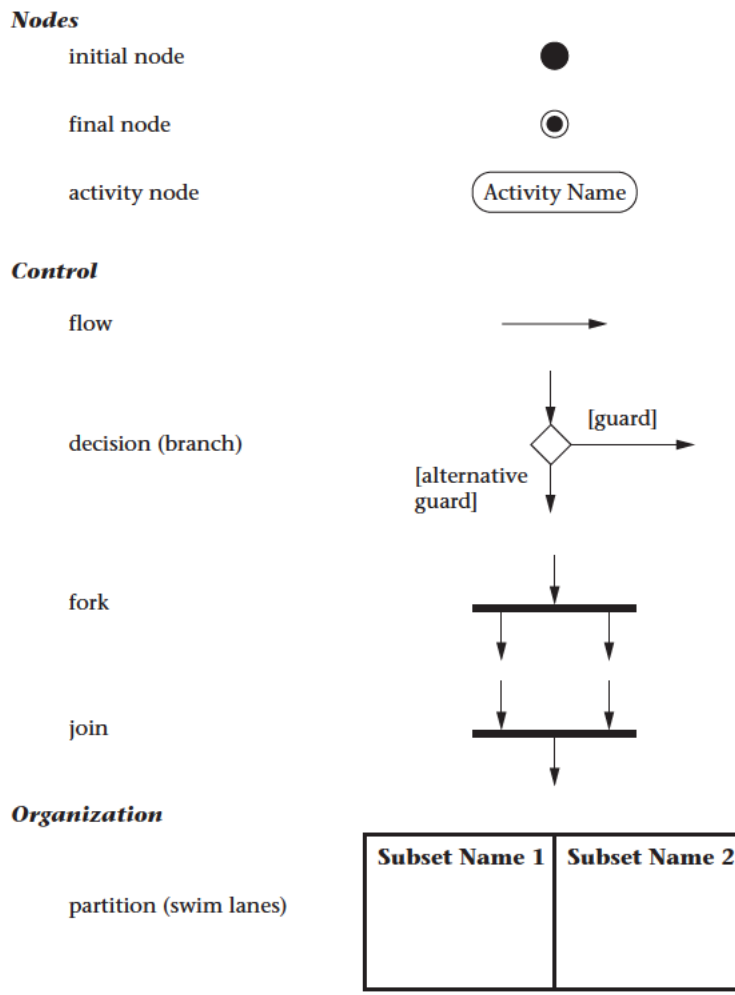


Figure 3.11 UML activity diagram constructs

sion node must be mutually exclusive, to avoid nondeterministic behavior. There can be no ambiguity as to which direction the control flows. The guards must cover all possible test conditions, so that control is not blocked at the decision node. One path may be guarded with [else]. If a path is guarded by [else], then control flows in that direction only if all the other guards fail. Forks and joins are both forms of synchronization written with a solid bar. The fork has one incoming flow and multiple outgoing flows. When control flows to a fork, the control concurrently

follows all the outgoing flows. These are referred to as concurrent threads. Joins are the opposite of forks; the join construct has multiple incoming flows and one outgoing flow. Control flows from a join only when control has reached the join from each of the incoming flows.

Activity diagrams may be further organized using partitions, also known as swim lanes. Partitions split activities into subsets, organized by responsible party. Each subset is named and enclosed with lines.

3.2.2 Activity Diagrams for Workflow

Figure 3.12 illustrates the UML activity diagram constructs used for the publication of this book. This diagram is partitioned into two subsets of activities, organized by responsible party: the left subset contains Customer activities, and the right subset contains Manufacturer activities. Activity partitions are sometimes called swim lanes, because of their typical appearance. Activity partitions may be arranged vertically, horizontally, or in a grid. Curved dividers may be used, although this is atypical. Activity diagrams can also be written without a partition. The construct is organizational, and doesn't carry inherent semantics. The meaning is suggested by your choice of subset names.

Control begins in the initial state, represented by the solid dot in the upper-left corner of Figure 3.12. Control flows to the first activity, where the customer requests a quote (Request quote). Control remains in an activity until that activity is completed; then the control follows the outgoing arrow. When the request for a quote is complete, the Manufacturer generates a quote (Generate quote). Then the Customer reviews the quote (Review quote).

The next construct is a branch, represented by a diamond. Each outgoing arrow from a branch has a guard. The guard represents a condition that must be true in order for control to flow along that path. Guards are written as short condition descriptions enclosed in brackets. After the Customer finishes reviewing the quote in Figure 3.12, if it is unacceptable the process reaches a final state and terminates. A final state is represented with a target (the bull's-eye). If the quote is acceptable, then the Customer places an order (Place order). The Manufacturer enters (Enter order), produces (Produce order), and ships the order (Ship order).

At a fork, control splits into multiple concurrent threads. The notation is a solid bar with one incoming arrow and multiple outgoing arrows. After the order ships in Figure 3.12, control reaches a fork and splits into two threads. The Customer receives the order (Receive order).

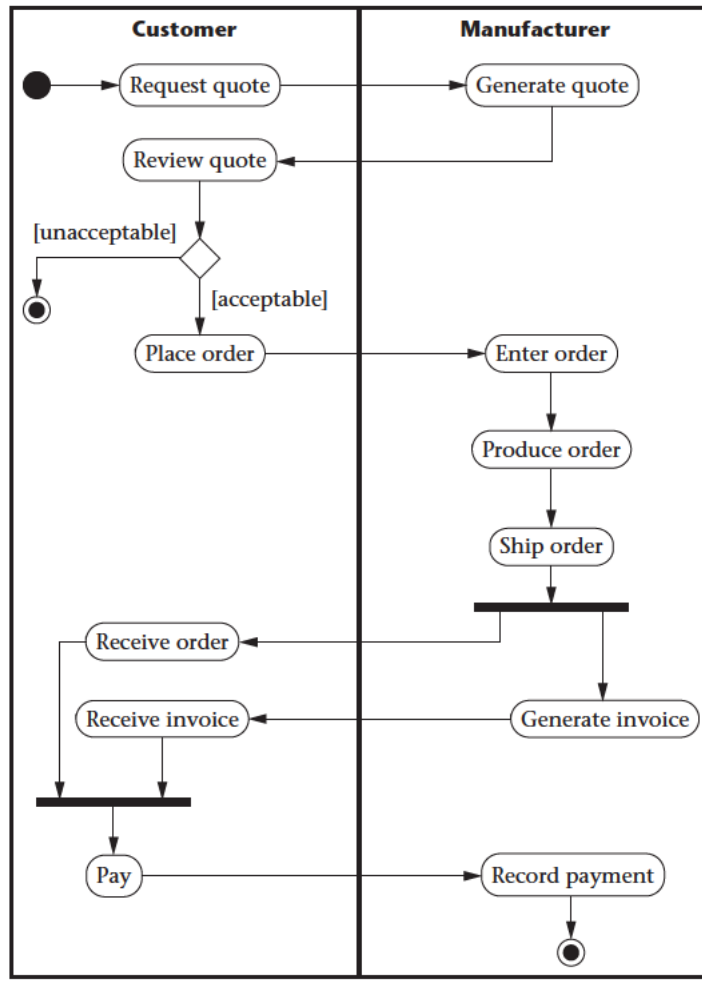


Figure 3.12 UML activity diagram, manufacturing example.

In parallel to the Customer receiving the order, the Manufacturer generates an invoice (Generate invoice), and then the Customer receives the invoice (Receive invoice). Order of activities between threads is not constrained. Thus, the Customer may receive the order before or after the manufacturer generates the invoice, or even after the Customer receives the invoice.

At a join, multiple threads merge into a single thread. The notation is a solid bar with multiple incoming arrows and one outgoing arrow. In

Figure 3.12, after the Customer receives the order and the invoice, then the Customer will pay (Pay). All incoming threads must complete before control continues along the outgoing arrow.

Finally, in Figure 3.12, the Customer pays, the Manufacturer records the payment (Record payment), and then a final state is reached. Notice that an activity diagram may have multiple final states. However, there can only be one initial state.

There are at least two uses for activity diagrams in the context of database design. Activity diagrams can specify the interactions of classes in a database schema. Class diagrams capture structure, activity diagrams capture behavior. The two types of diagrams can present complementary aspects of the same system. For example, one can easily imagine that Figure 3.12 illustrates the usage of classes named Quote, Order, Invoice, and Payment. Another use for activity diagrams in the context of database design is to illustrate processes surrounding the database. For example, database life cycles can be illustrated using activity diagrams.

3.3 Rules of Thumb for UML Usage

1. Decide what you wish to communicate first, and then focus your description. Illustrate the details that further your purpose, and elide the rest. UML is like any other language in that you can immerse yourself in excruciating detail and lose your main purpose. Be concise.
2. Keep each UML diagram to one page. Diagrams are easier to understand if they can be seen in one glance. This is not to say that you must restrict yourself; rather, you should divide and organize your content into reasonable, understandable portions. Use packages to organize your presentation. If you have many brilliant ideas to convey (of course you do!), begin with a high-level diagram that paints the broad picture. Then follow up with a diagram dedicated to each of your ideas.
3. Use UML when it is useful. Don't feel compelled to write a UML document just because you feel you need a UML document. UML is not an end in itself, but it is an excellent design tool for appropriate problems.
4. Accompany your diagrams with textual descriptions, thereby clarifying your intent. Additionally, remember that some people are

oriented verbally, others visually. Combining natural language with UML is effective.

5. Take care to clearly organize each diagram. Avoid crossing associations. Group elements together if there is a connection in your mind. Two UML diagrams can contain the exact same elements and associations, and one might be a jumbled mess, while the other is elegant and clear. Both convey the same meaning in the UML, but clearly the elegant version will be more successful at communicating design issues.

3.4 Summary

The Unified Modeling Language (UML) is a graphical language that is currently very popular for communicating design specifications for software and in particular for logical database designs via class diagrams. The similarity between UML and the ER model is shown through some common examples, including ternary relationships and generalization. UML activity diagrams are used to specify the activities and flow of control in processes. Use of UML in logical database design is summarized with five basic rules of thumb.

3.5 Literature Summary

The definitive reference manual for UML is Rumbaugh, Jacobson, and Booch [2005]. Use Mullins [1999] for more detailed UML database modeling. Other useful UML texts are Naiburg and Maksimchuk [2001], Quatrani [2002], and Rumbaugh, Jacobson, and Booch [2004].

4 Requirements Analysis and Conceptual Data Modeling

This chapter shows how the ER and UML approaches can be applied to the database life cycle, particularly in steps I through II(b) (as defined in Section 1.2), which include the requirements analysis and conceptual data modeling stages of logical database design. The example introduced in Chapter 2 is used again to illustrate the ER modeling principles developed in this chapter.

4.1 Introduction

Logical database design is accomplished with a variety of approaches, including the top-down, bottom-up, and combined methodologies. The traditional approach, particularly for relational databases, has been a low-level, bottom-up activity, synthesizing individual data elements into normalized tables after carefully analyzing the data element interdependencies defined during the requirements analysis. Although the traditional process has been somewhat successful for small- to medium-sized databases, when used for large databases its complexity can be overwhelming to the point where practicing designers do not bother to use it with any regularity. In practice, a combination of the top-down and bottom-up approaches is used; in most cases, tables can be defined directly from the requirements analysis.

The conceptual data model has been most successful as a tool for communication between the designer and the end user during the

requirements analysis and logical design phases. Its success is due to the fact that the model, using either ER or UML, is easy to understand and convenient to represent. Another reason for its effectiveness is that it is a top-down approach using the concept of abstraction. The number of entities in a database is typically far fewer than the number of individual data elements, because data elements usually represent the attributes. Therefore, using entities as an abstraction for data elements and focusing on the relationships between entities greatly reduces the number of objects under consideration and simplifies the analysis. Though it is still necessary to represent data elements by attributes of entities at the conceptual level, their dependencies are normally confined to the other attributes within the entity or, in some cases, to attributes associated with other entities with a direct relationship to their entity.

The major interattribute dependencies that occur in data models are the dependencies between the *entity keys*, the unique identifiers of different entities that are captured in the conceptual data modeling process. Special cases, such as dependencies among data elements of unrelated entities, can be handled when they are identified in the ensuing data analysis.

The logical database design approach defined here uses both the conceptual data model and the relational model in successive stages. It benefits from the simplicity and ease of use of the conceptual data model and the structure and associated formalism of the relational model. To facilitate this approach, it is necessary to build a framework for transforming the variety of conceptual data model constructs into tables that are already normalized or that can be normalized with a minimum of transformation. The beauty of this type of transformation is that it results in normalized or nearly normalized SQL tables from the start; frequently, further normalization is not necessary.

Before we do this, however, we need to first define the major steps of the relational logical design methodology in the context of the database life cycle.

4.2 Requirements Analysis

Step I, requirements analysis, is an extremely important step in the database life cycle and is typically the most labor intensive. The database designer must interview the end user population and determine exactly what the database is to be used for and what it must contain. The basic objectives of requirements analysis are:

- To delineate the data requirements of the enterprise in terms of basic data elements
- To describe the information about the data elements and the relationships among them needed to model these data requirements
- To determine the types of transactions that are intended to be executed on the database and the interaction between the transactions and the data elements
- To define any performance, integrity, security, or administrative constraints that must be imposed on the resulting database
- To specify any design and implementation constraints, such as specific technologies, hardware and software, programming languages, policies, standards, or external interfaces
- To thoroughly document all of the preceding in a detailed requirements specification. The data elements can also be defined in a data dictionary system, often provided as an integral part of the database management system

The conceptual data model helps designers accurately capture the real data requirements because it requires them to focus on semantic detail in the data relationships, which is greater than the detail that would be provided by FDs alone. The semantics of the ER model, for instance, allow for direct transformations of entities and relationships to at least first normal form (1NF) tables. They also provide clear guidelines for integrity constraints. In addition, abstraction techniques such as generalization provide useful tools for integrating end user views to define a global conceptual schema.

4.3 Conceptual Data Modeling

Let us now look more closely at the basic data elements and relationships that should be defined during requirements analysis and conceptual design. These two life cycle steps are often done simultaneously.

Consider the substeps in step II(a), conceptual data modeling, using the ER model:

- Classify entities and attributes (classify classes and attributes in UML)
- Identify the generalization hierarchies (for both the ER model and UML)

- Define relationships (define associations and association classes in UML)

The remainder of this section discusses the tasks involved in each substep.

4.3.1 Classify Entities and Attributes

Though it is easy to define entity, attribute, and relationship constructs, it is not as easy to distinguish their roles in modeling the database. What makes a data element an entity, an attribute, or even a relationship? For example, project headquarters are located in cities. Should "city" be an entity or an attribute? A vita is kept for each employee. Is "vita" an entity or a relationship?

The following guidelines for classifying entities and attributes will help the designer's thoughts converge to a normalized relational database design:

- Entities should contain descriptive information.
- Multivalued attributes should be classified as entities.
- Attributes should be attached to the entities they most directly describe.

Now we examine each guideline in turn.

Entity Contents

Entities should contain descriptive information. If there is descriptive information about a data element, the data element should be classified as an entity. If a data element requires only an identifier and does not have relationships, it should be classified as an attribute. With "city," for example, if there is some descriptive information such as "country" and "population" for cities, then "city" should be classified as an entity. If only the city name is needed to identify a city, then "city" should be classified as an attribute associated with some entity, such as Project. The exception to this rule is that if the identity of the value needs to be constrained by set membership, you should create it as an entity. For example, "State" is much the same as city, but you probably want to have a State entity that contains all the valid State instances. Examples of other data elements in the real world that are typically classified as entities

include Employee, Task, Project, Department, Company, Customer, and so on.

Multivalued Attributes

Classify multivalued attributes as entities. If more than one value of a descriptor attribute corresponds to one value of an identifier, the descriptor should be classified as an entity instead of an attribute, even though it does not have descriptors itself. A large company, for example, could have many divisions, some of them possibly in different cities. In that case, "division" could be classified as a multivalued attribute of "company," but it would be better classified as an entity, with "division-address" as its identifier. If attributes are restricted to be single valued only, the later design and implementation decisions will be simplified.

Attribute Attachment

Attach attributes to the entities they most directly describe. For example, "office-building-name" should normally be an attribute of the entity Department, rather than the entity Employee. The procedure of identifying entities and attaching attributes to entities is iterative. Classify some data elements as entities and attach identifiers and descriptors to them. If you find some violation of the preceding guidelines, change some data elements from entity to attribute (or from attribute to entity), attach attributes to the new entities, and so forth.

4.3.2 Identify the Generalization Hierarchies

If there is a generalization hierarchy among entities, then put the identifier and generic descriptors in the supertype entity and put the same identifier and specific descriptors in the subtype entities.

For example, suppose five entities were identified in the ER model shown in Figure 2.4a:

- Employee, with identifier empno and descriptors empname, address, and date-of-birth
- Manager, with identifier empno and descriptors empname and jobtitle
- Engineer, with identifier empno and descriptors empname, highest-degree and jobtitle

- Technician, with identifier empno, and descriptors empname and specialty
- Secretary, with identifier empno, and descriptors empname and best-skill

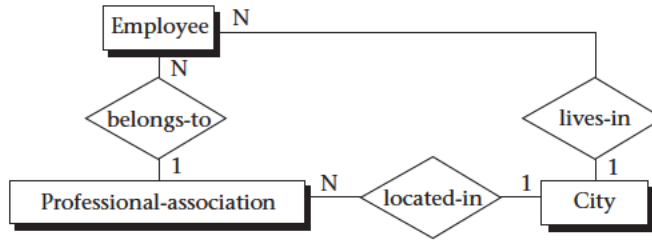
Let's say we determine, through our analysis, that the entity Employee could be created as a generalization of Manager, Engineer, Technician, and Secretary. Then we put identifier empno and generic descriptors empname, address, and date-of-birth in the supertype entity Employee; identifier empno and specific descriptor jobtitle in the subtype entity Manager; identifier empno and specific descriptor highest-degree and jobtitle in the subtype entity Engineer, etc. Later, if we decide to eliminate **Employee** as a table, the original identifiers and generic attributes can be redistributed to all the subtype tables. (Note that we put table names in boldface throughout the book for readability.)

4.3.3 Define Relationships

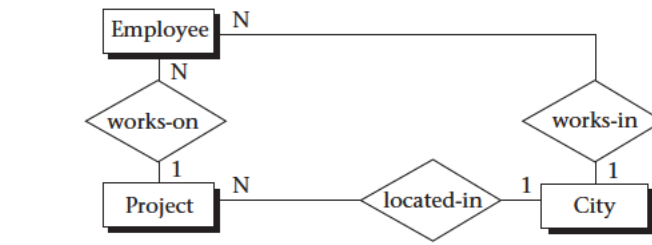
We now deal with data elements that represent associations among entities, which we call relationships. Examples of typical relationships are "works-in," "works-for," "purchases," "drives," or any verb that connects entities. For every relationship, the following should be specified: degree (binary, ternary, etc.); connectivity (one-to-many, etc.); optional or mandatory existence; and any attributes associated with the relationship and not the entities. The following are some guidelines for defining the more difficult types of relationships.

Redundant Relationships

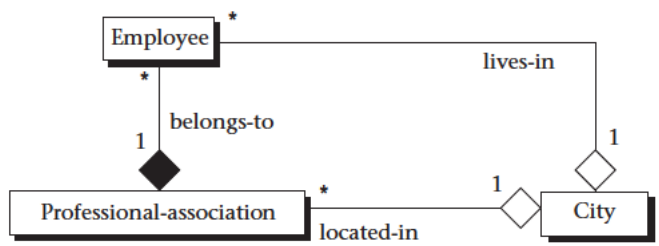
Analyze redundant relationships carefully. Two or more relationships that are used to represent the same concept are considered redundant. Redundant relationships are more likely to result in unnormalized tables when transforming the ER model into relational schemas. Note that two or more relationships are allowed between the same two entities, as long as those relationships have different meanings. In this case they are not considered redundant. One important case of nonredundancy is shown in Figure 4.1a for the ER model and Figure 4.1c for UML. If "belongs-to" is a one-to-many relationship between Employee and Professional-association, if "located-in" is a one-to-many relationship between Professional-association and City, and if "lives-in" is a one-to-many relation-



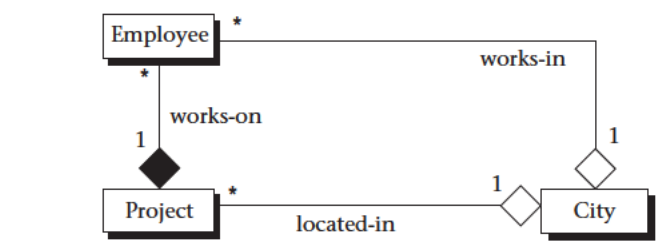
(a) Nonredundant relationships



(b) Redundant relationships using transitivity



(c) Nonredundant associations



(d) Redundant associations using transitivity

Figure 4.1 Redundant relationships

ship between Employee and City, then "lives-in" is not redundant, because the relationships are unrelated. However, consider the situation shown in Figure 4.1b for the ER model and Figure 4.1d for UML. The employee works on a project located in a city, so the "works-in" relationship between Employee and City is redundant and can be eliminated.

Ternary Relationships

Define ternary relationships carefully. We define a ternary relationship among three entities only when the concept cannot be represented by several binary relationships among those entities. For example, let us assume there is some association among entities Technician, Project, and Notebook. If each technician can be working on any of several projects and using the same notebooks on each project, then three many-to-many binary relationships can be defined (see Figure 4.2a for the ER model and Figure 4.2c for UML). If, however, each technician is

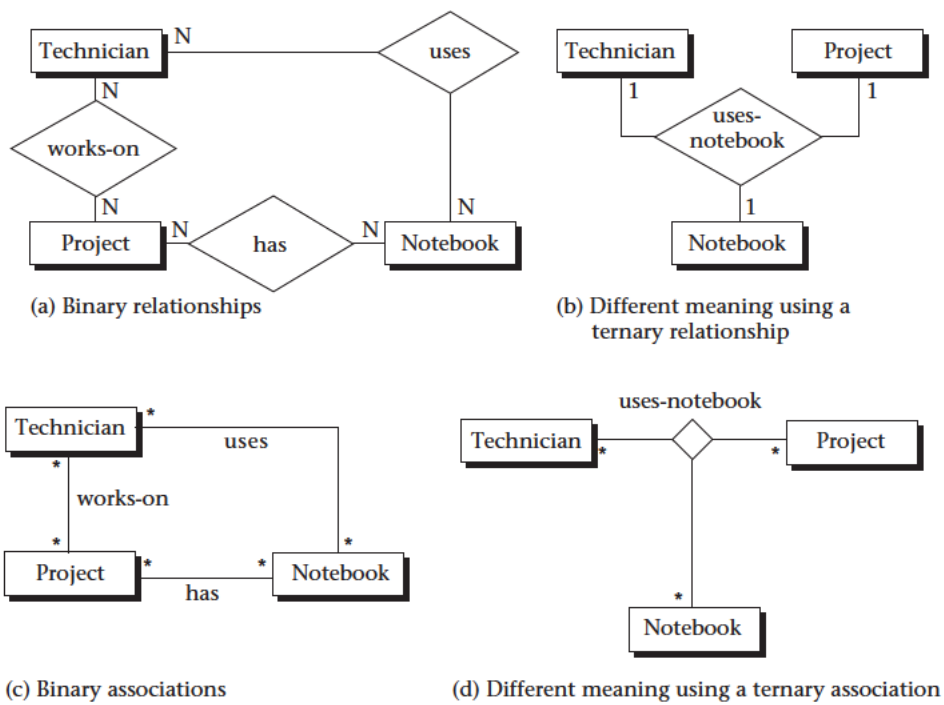


Figure 4.2 Ternary relationships

constrained to use exactly one notebook for each project and that notebook belongs to only one technician, then a one-to-one-to-one ternary relationship should be defined (see Figure 4.2b for the ER model and Figure 4.2d for UML). The approach to take in ER modeling is to first attempt to express the associations in terms of binary relationships; if this is impossible because of the constraints of the associations, try to express them in terms of a ternary.

The meaning of connectivity for ternary relationships is important. Figure 4.2b shows that for a given pair of instances of Technician and Project, there is only one corresponding instance of Notebook; for a given pair of instances of Technician and Notebook, there is only one corresponding instance of Project; and for a given pair of instances of Project and Notebook, there is only one instance of Technician. In general, we know by our definition of ternary relationships that if a relationship among three entities can only be expressed by a functional dependency involving the keys of all three entities, then it cannot be expressed using only binary relationships, which only apply to associations between two entities. Object-oriented design provides arguably a better way to model this situation [Muller, 1999].

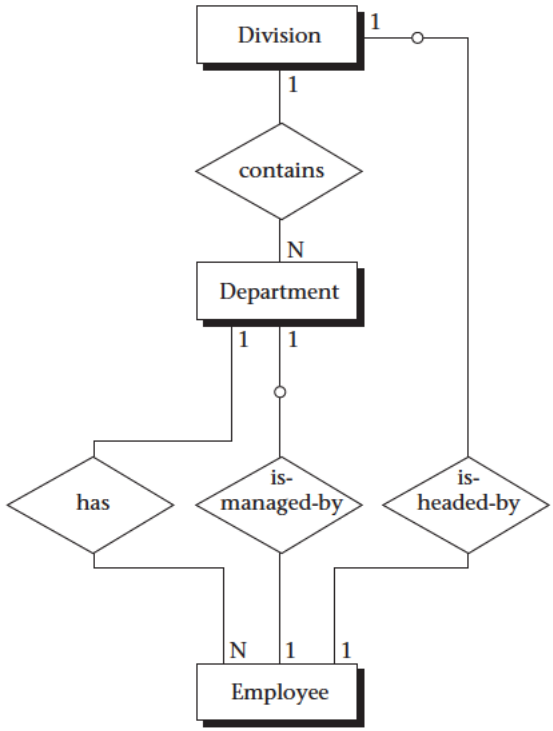
4.3.4 Example of Data Modeling: Company Personnel and Project Database

ER Modeling of Individual Views Based on Requirements

Let us suppose it is desirable to build a company-wide database for a large engineering firm that keeps track of all full-time personnel, their skills and projects assigned, the departments (and divisions) worked in, the engineer professional associations belonged to, and the engineer desktop computers allocated. During the requirements collection process—that is, interviewing the end users—we obtain three views of the database.

The first view, a management view, defines each employee as working in a single department, and defines a division as the basic unit in the company, consisting of many departments. Each division and department has a manager, and we want to keep track of each manager. The ER model for this view is shown in Figure 4.3a.

The second view defines each employee as having a job title: engineer, technician, secretary, manager, and so on. Engineers typically belong to professional associations and might be allocated an engineer-

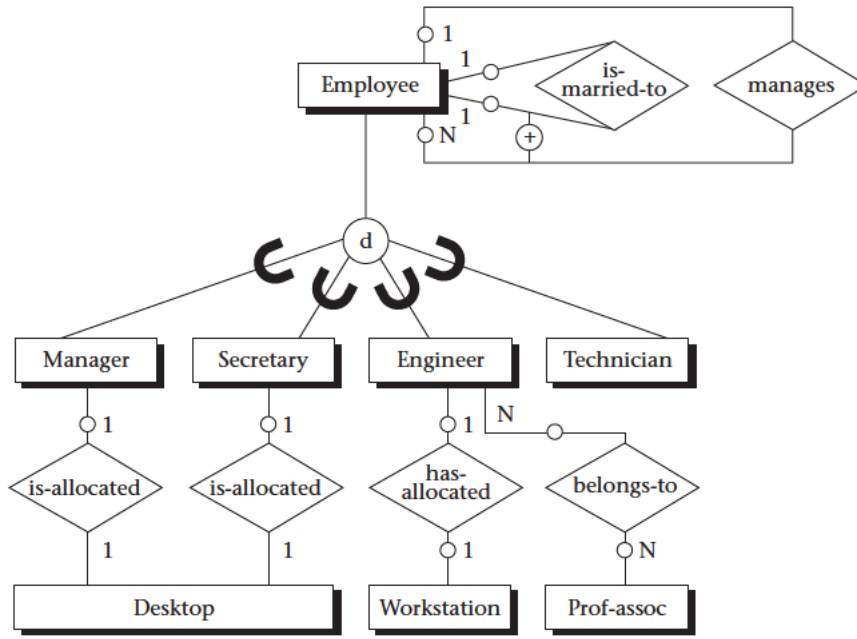


(a) Management view

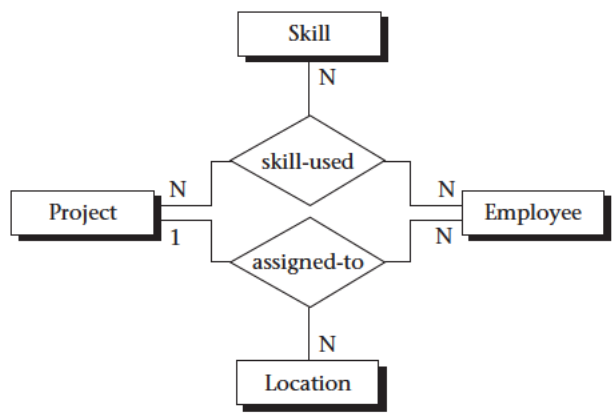
Figure 4.3 Example of data modeling

ing workstation (or computer). Secretaries and managers are each allocated a desktop computer. A pool of desktops and workstations is maintained for potential allocation to new employees and for loans while an employee's computer is being repaired. Any employee may be married to another employee, and we want to keep track of these relationships to avoid assigning an employee to be managed by his or her spouse. This view is illustrated in Figure 4.3b.

The third view, shown in Figure 4.3c, involves the assignment of employees, mainly engineers and technicians, to projects. Employees may work on several projects at one time, and each project could be headquartered at different locations (cities). However, each employee at a given location works on only one project at that location. Employee skills can be individually selected for a given project, but no individual has a monopoly on skills, projects, or locations.



(b) Employee view



(c) Employee assignment view

Figure 4.3 *(continued)*

Global ER Schema

A simple integration of the three views over the entity Employee defines results in the global ER schema (diagram) in Figure 4.3d, which becomes the basis for developing the normalized tables. Each relationship in the global schema is based upon a verifiable assertion about the actual data in the enterprise, and analysis of those assertions leads to the transformation of these ER constructs into candidate SQL tables, as Chapter 5 shows.

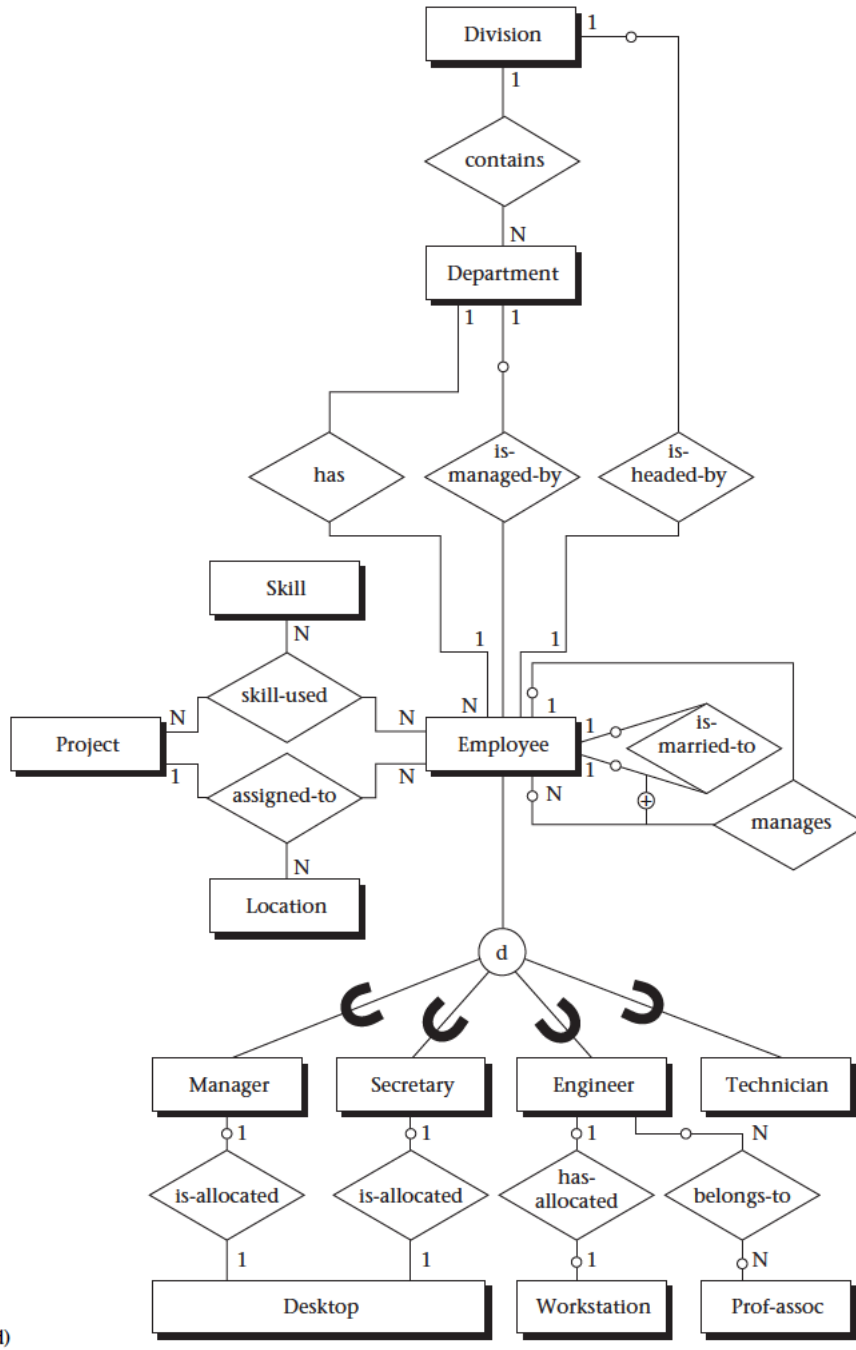
Note that equivalent views and integration could be done for a UML conceptual model over the class Employee. We will use the ER model for the examples in the rest of this chapter, however.

The diagram shows examples of binary, ternary, and binary recursive relationships; optional and mandatory existence in relationships; and generalization with the disjointness constraint. Ternary relationships "skill-used" and "assigned-to" are necessary, because binary relationships cannot be used for the equivalent notions. For example, one employee and one location determine exactly one project (a functional dependency). In the case of "skill-used," selective use of skills to projects cannot be represented with binary relationships (see Section 6.5).

The use of optional existence, for instance, between Employee and Division or between Employee and Department, is derived from our general knowledge that most employees will not be managers of any division or department. In another example of optional existence, we show that the allocation of a workstation to an engineer may not always occur, nor will all desktops or workstations necessarily be allocated to someone at all times. In general, all relationships, optional existence constraints, and generalization constructs need to be verified with the end user before the ER model is transformed to SQL tables.

In summary, the application of the ER model to relational database design offers the following benefits:

- Use of an ER approach focuses end users' discussions on important relationships between entities. Some applications are characterized by counterexamples affecting a small number of instances, and lengthy consideration of these instances can divert attention from basic relationships.
- A diagrammatic syntax conveys a great deal of information in a compact, readily understandable form.
- Extensions to the original ER model, such as optional and mandatory membership classes, are important in many relationships.



(d)

Figure 4.3 *(continued)*

Generalization allows entities to be grouped for one functional role or to be seen as separate subtypes when other constraints are imposed.

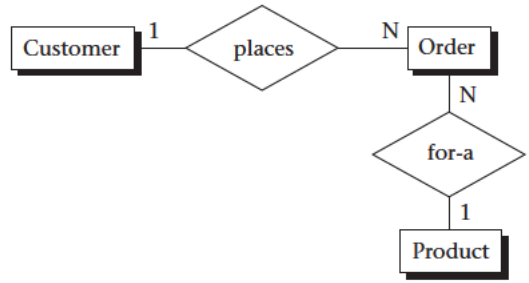
- A complete set of rules transforms ER constructs into mostly normalized SQL tables, which follow easily from real-world requirements.

4.4 View Integration

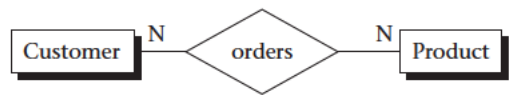
A critical part of the database design process is step II(b), the integration of different user views into a unified, nonredundant global schema. The individual end-user views are represented by conceptual data models, and the integrated conceptual schema results from sufficient analysis of the end-user views to resolve all differences in perspective and terminology. Experience has shown that nearly every situation can be resolved in a meaningful way through integration techniques.

Schema diversity occurs when different users or user groups develop their own unique perspectives of the world or, at least, of the enterprise to be represented in the database. For instance, the marketing division tends to have the whole product as a basic unit for sales, but the engineering division may concentrate on the individual parts of the whole product. In another case, one user may view a project in terms of its goals and progress toward meeting those goals over time, but another user may view a project in terms of the resources it needs and the personnel involved. Such differences cause the conceptual models to seem to have incompatible relationships and terminology. These differences show up in conceptual data models as different levels of abstraction, connectivity of relationships (one-to-many, many-to-many, and so on), or as the same concept being modeled as an entity, attribute, or relationship, depending on the user's perspective.

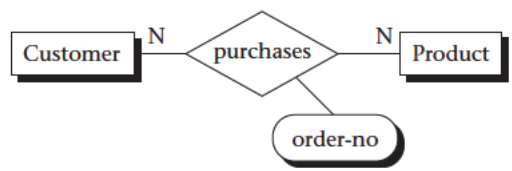
As an example of the latter case, in Figure 4.4 we see three different perspectives of the same real-life situation—the placement of an order for a certain product. The result is a variety of schemas. The first schema (Figure 4.4a) depicts Customer, Order, and Product as entities and "places" and "for-a" as relationships. The second schema (Figure 4.4b), however, defines "orders" as a relationship between Customer and Product and omits Order as an entity altogether. Finally, in the third case (Figure 4.4c), the relationship "orders" has been replaced by another relationship, "purchases"; "order-no," the identifier (key) of an order, is designated as an attribute of the relationship "purchases." In other



(a) The concept of order as an entity



(b) The concept of order as a relationship



(c) The concept of order as an attribute

Figure 4.4 Schemas: placement of an order

words, the concept of order has been variously represented as an entity, a relationship, and an attribute, depending on perspective.

There are four basic steps needed for conceptual schema integration:

1. Preintegration analysis
2. Comparison of schemas
3. Conformation of schemas
4. Merging and restructuring of schemas

4.4.1 Preintegration Analysis

The first step, preintegration analysis, involves choosing an integration strategy. Typically, the choice is between a binary approach with two

schemas merged at one time and an *n*-ary approach with *n* schemas merged at one time, where *n* is between 2 and the total number of schemas developed in the conceptual design. The binary approach is attractive because each merge involves a small number of data model constructs and is easier to conceptualize. The *n*-ary approach may require only one grand merge, but the number of constructs may be so large that it is not humanly possible to organize the transformations properly.

4.4.2 Comparison of Schemas

In the second step, comparison of schemas, the designer looks at how entities correspond and detects conflicts arising from schema diversity—that is, from user groups adopting different viewpoints in their respective schemas. Naming conflicts include synonyms and homonyms. Synonyms occur when different names are given for the same concept; these can be detected by scanning the data dictionary, if one has been established for the database. Homonyms occur when the same name is used for different concepts. These can only be detected by scanning the different schemas and looking for common names.

Structural conflicts occur in the schema structure itself. Type conflicts involve using different constructs to model the same concept. In Figure 4.4, for example, an entity, a relationship, or an attribute can be used to model the concept of order in a business database. Dependency conflicts result when users specify different levels of connectivity (one-to-many, etc.) for similar or even the same concepts. One way to resolve such conflicts might be to use only the most general connectivity—for example, many-to-many. If that is not semantically correct, change the names of entities so that each type of connectivity has a different set of entity names. Key conflicts occur when different keys are assigned to the same entity in different views. For example, a key conflict occurs if an employee's full name, employee ID number, and Social Security number are all assigned as keys.

4.4.3 Conformation of Schemas

The resolution of conflicts often requires user and designer interaction. The basic goal of the third step is to align or conform schemas to make them compatible for integration. The entities, as well as the key attributes, may need to be renamed. Conversion may be required so that concepts modeled as entities, attributes, or relationships are conformed to be only one of them. Relationships with equal degree, roles, and con-

nectivity constraints are easy to merge. Those with differing characteristics are more difficult and, in some cases, impossible to merge. In addition, relationships that are not consistent—for example, a relationship using generalization in one place and the exclusive OR in another—must be resolved. Finally, assertions may need to be modified so that integrity constraints remain consistent.

Techniques used for view integration include abstraction, such as generalization and aggregation to create new supertypes or subtypes, or even the introduction of new relationships. As an example, the generalization of Individual over different values of the descriptor attribute "jobtitle" could represent the consolidation of two views of the database—one based on an individual as the basic unit of personnel in the organization, and another based on the classification of individuals by job titles and special characteristics within those classifications.

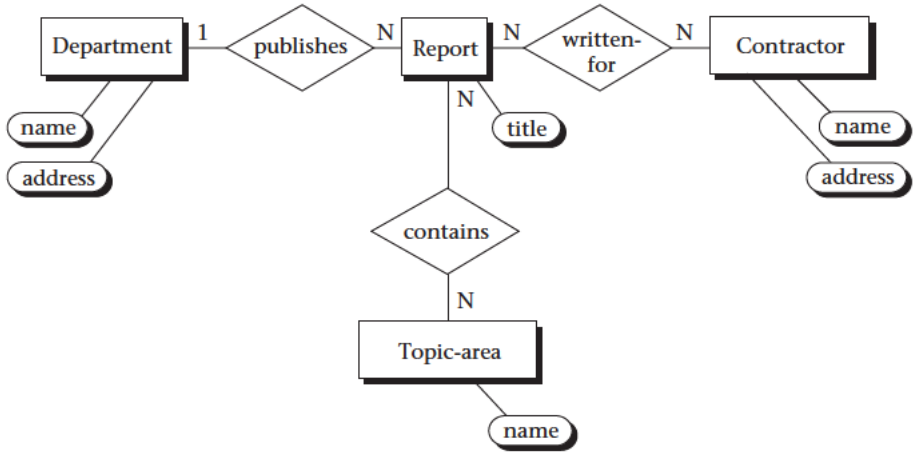
4.4.4 Merging and Restructuring of Schemas

The fourth step consists of the merging and restructuring of schemas. This step is driven by the goals of completeness, minimality, and understandability. Completeness requires all component concepts to appear semantically intact in the global schema. Minimality requires the designer to remove all redundant concepts in the global schema. Examples of redundant concepts are overlapping entities and truly semantically redundant relationships; for example, Ground-Vehicle and Automobile might be two overlapping entities. A redundant relationship might occur between Instructor and Student. The relationships "direct-research" and "advise" may or may not represent the same activity or relationship, so further investigation is required to determine whether they are redundant or not. Understandability requires that the global schema make sense to the user.

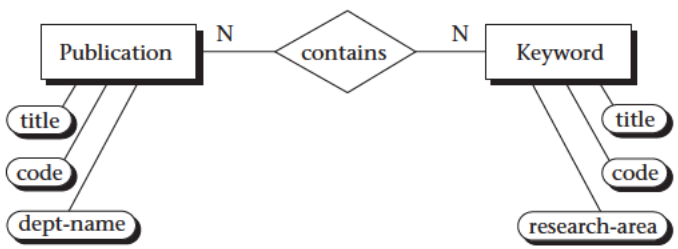
Component schemas are first merged by superimposing the same concepts and then restructuring the resulting integrated schema for understandability. For instance, if a supertype/subtype combination is defined as a result of the merging operation, the properties of the subtype can be dropped from the schema because they are automatically provided by the supertype entity.

4.4.5 Example of View Integration

Let us look at two different views of overlapping data. The views are based on two separate interviews of end users. We adapt the interesting



(a) Original schema 1, focused on reports



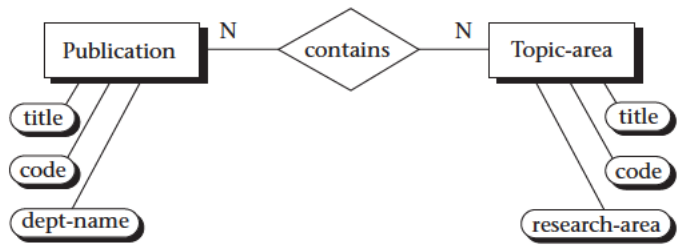
(b) Original schema 2, focused on publications

Figure 4.5 View integration: find meaningful ways to integrate

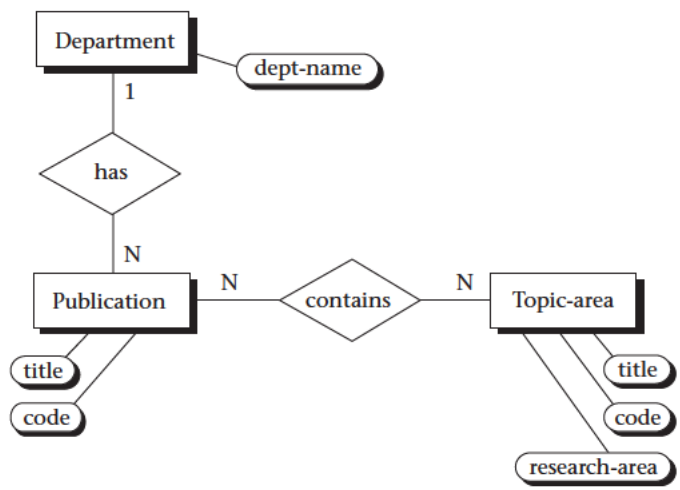
example cited by Batini, Lenzerini, and Navathe [1986] to a hypothetical situation related to our example.

In Figure 4.5a we have a view that focuses on reports and includes data on departments that publish the reports, topic areas in reports, and contractors for whom the reports are written. Figure 4.5b shows another view, with publications as the central focus and keywords on publication as the secondary data. Our objective is to find meaningful ways to integrate the two views and maintain completeness, minimality, and understandability.

We first look for synonyms and homonyms, particularly among the entities. Note that a synonym exists between the entities Topic-area in schema 1 and Keyword in schema 2, even though the attributes do not



(a) Schema 2.1, in which Keyword has changed to Topic-area

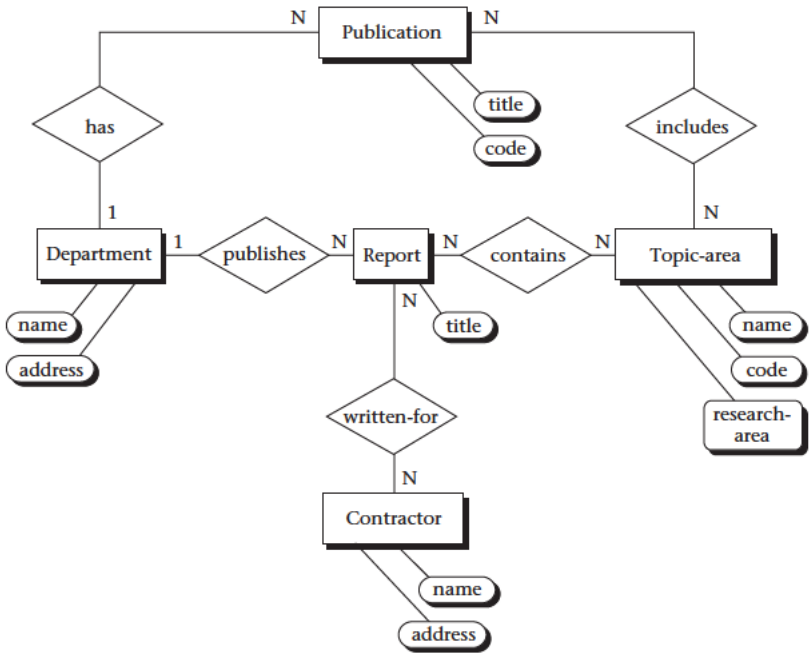


(b) Schema 2.2, in which the attribute dept-name has changed to an attribute and an entity

Figure 4.6 View integration: type conflict

match. However, we find that the attributes are compatible and can be consolidated. This is shown in Figure 4.6a, which presents a revised schema, schema 2.1. In schema 2.1 Keyword has been replaced by Topic-area.

Next we look for structural conflicts between schemas. A type conflict is found to exist between the entity Department in schema 1 and the attribute "dept-name" in schema 2.1. The conflict is resolved by keeping the stronger entity type, Department, and moving the attribute type "dept-name" under Publication in schema 2 to the new entity, Department, in schema 2.2 (see Figure 4.6b).

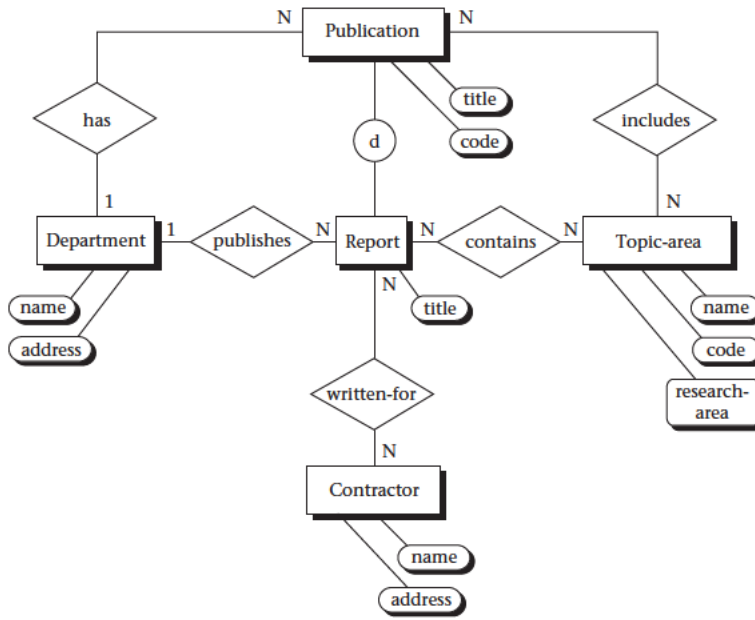


(a) Schema 3, the result of merging schema 1 and schema 2.2

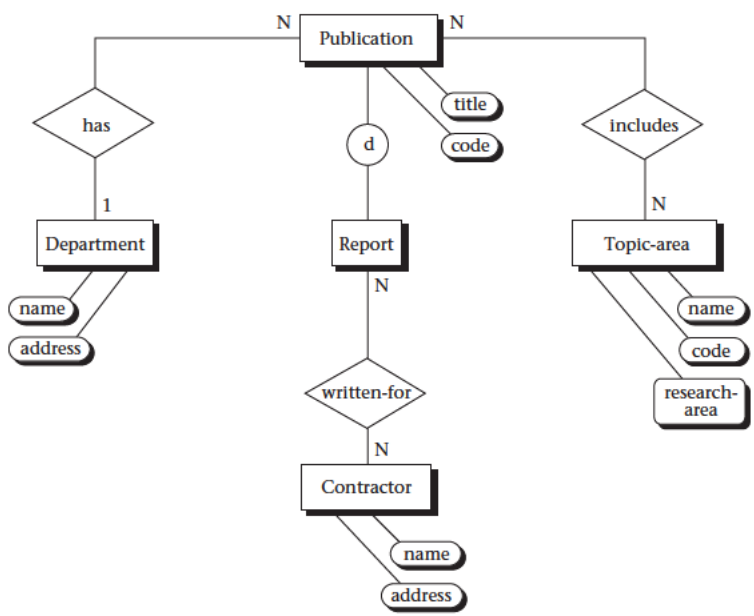
Figure 4.7 View integration: the merged schema

At this point we have sufficient commonality between schemas to attempt a merge. In schemas 1 and 2.2 we have two sets of common entities, Department and Topic-area. Other entities do not overlap and must appear intact in the superimposed, or merged, schema. The merged schema, schema 3, is shown in Figure 4.7a. Because the common entities are truly equivalent, there are no bad side effects of the merge due to existing relationships involving those entities in one schema and not in the other. (Such a relationship that remains intact exists in schema 1 between Topic-area and Report, for example.) If true equivalence cannot be established, the merge may not be possible in the existing form.

In Figure 4.7, there is some redundancy between Publication and Report in terms of the relationships with Department and Topic-area. Such a redundancy can be eliminated if there is a supertype/subtype relationship between Publication and Report, which does in fact occur in this case because Publication is a generalization of Report. In schema 4.1 (Figure 4.7b) we see the introduction of this generalization from Report to Publication. Then in schema 4.2 (Figure 4.7c) we see that the



(b) Schema 3.1, new generalization



(c) Schema 3.2, elimination of redundant relationships

Figure 4.7 *(continued)*

redundant relationships between Report and Department and Topic-area have been dropped. The attribute "title" has been eliminated as an attribute of Report in Figure 4.7c because "title" already appears as an attribute of Publication at a higher level of abstraction; "title" is inherited by the subtype Report.

The final schema, in Figure 4.7c, expresses completeness because all the original concepts (report, publication, topic area, department, and contractor) are kept intact. It expresses minimality because of the transformation of "dept-name" from attribute in schema 1 to entity and attribute in schema 2.2, and the merger between schema 1 and schema 2.2 to form schema 3, and because of the elimination of "title" as an attribute of Report and of Report relationships with Topic-area and Department. Finally, it expresses understandability in that the final schema actually has more meaning than the individual original schemas.

The view integration process is one of continual refinement and reevaluation. It should also be noted that minimality may not always be the most efficient way to proceed. If, for example, the elimination of the redundant relationships "publishes" and/or "contains" from schema 3.1 to 3.2 causes the time required to perform certain queries to be excessively long, it may be better from a performance viewpoint to leave them in. This decision could be made during the analysis of the transactions on the database or during the testing phase of the fully implemented database.

4.5 Entity Clustering for ER Models

This section presents the concept of entity clustering, which abstracts the ER schema to such a degree that the entire schema can appear on a single sheet of paper or a single computer screen. This has happy consequences for the end user and database designer in terms of developing a mutual understanding of the database contents and formally documenting the conceptual model.

An entity cluster is the result of a grouping operation on a collection of entities and relationships. Entity clustering is potentially useful for designing large databases. When the scale of a database or information structure is large and includes a large number of interconnections among its different components, it may be very difficult to understand the semantics of such a structure and to manage it, especially for the end users or managers. In an ER diagram with 1,000 entities, the overall

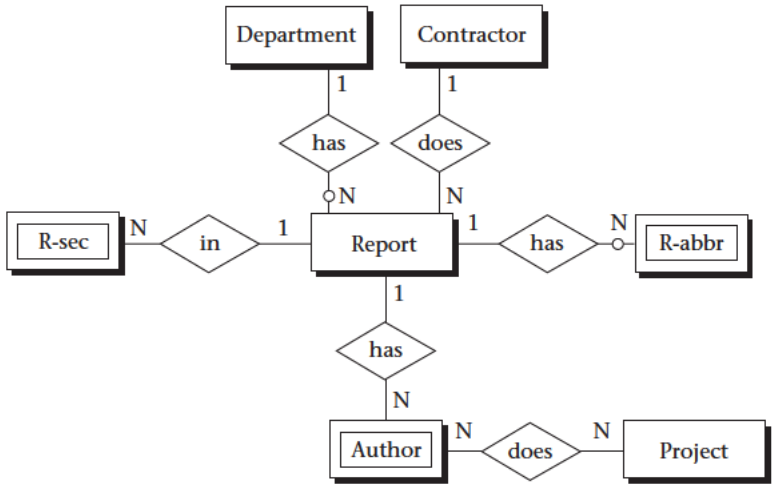
structure will probably not be very clear, even to a well-trained database analyst. Clustering is therefore important because it provides a method to organize a conceptual database schema into layers of abstraction, and it supports the different views of a variety of end users.

4.5.1 Clustering Concepts

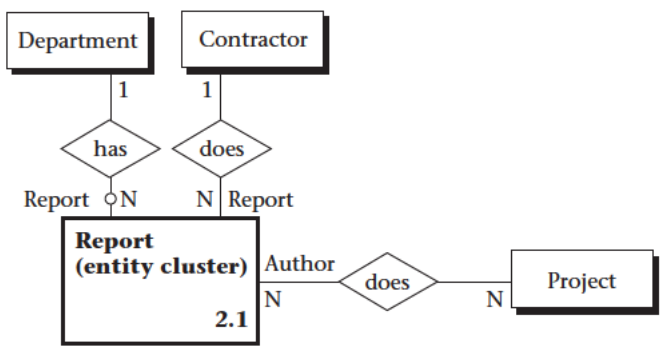
One should think of grouping as an operation that combines entities and their relationships to form a higher-level construct. The result of a grouping operation on simple entities is called an *entity cluster.* A grouping operation on entity clusters, or on combinations of elementary entities and entity clusters, results in a higher-level entity cluster. The highest-level entity cluster, representing the entire database conceptual schema, is called the *root entity cluster.*

Figure 4.8a illustrates the concept of entity clustering in a simple case where (elementary) entities R-sec (report section), R-abbr (report abbreviation), and Author are naturally bound to (dominated by) the entity Report; and entities Department, Contractor, and Project are not dominated. (Note that to avoid unnecessary detail, we do not include the attributes of entities in the diagrams.) In Figure 4.8b, the dark-bordered box around the entity Report and the entities it dominates defines the entity cluster Report. The dark-bordered box is called the EC box to represent the idea of an entity cluster. In general, the name of the entity cluster need not be the same as the name of any internal entity; however, when there is a single dominant entity, the names are often the same. The EC box number in the lower-right corner is a clustering-level number used to keep track of the sequence in which clustering is done. The number 2.1 signifies that the entity cluster Report is the first entity cluster at level 2. Note that all the original entities are considered to be at level 1.

The higher-level abstraction, the entity cluster, must maintain the same relationships between entities inside and outside the entity cluster as occur between the same entities in the lower-level diagram. Thus, the entity names inside the entity cluster should appear just outside the EC box along the path of their direct relationship to the appropriately related entities outside the box, maintaining consistent interfaces (relationships) as shown in Figure 4.8b. For simplicity, we modify this rule slightly: If the relationship is between an external entity and the dominant internal entity (for which the entity cluster is named), the entity cluster name need not be repeated outside the EC box. Thus, in Figure 4.8b, we could drop the name Report both places it occurs outside the



(a) ER model before clustering



(b) ER model after clustering

Figure 4.8 Entity clustering concepts

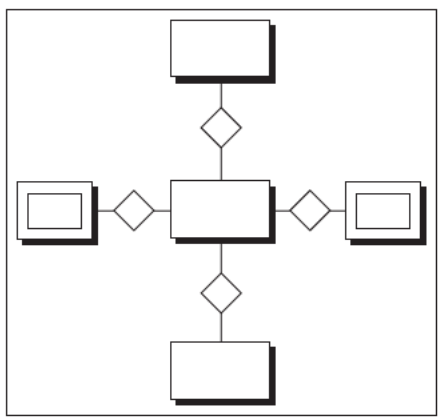
Report box, but we must retain the name Author, which is not the name of the entity cluster.

4.5.2 Grouping Operations

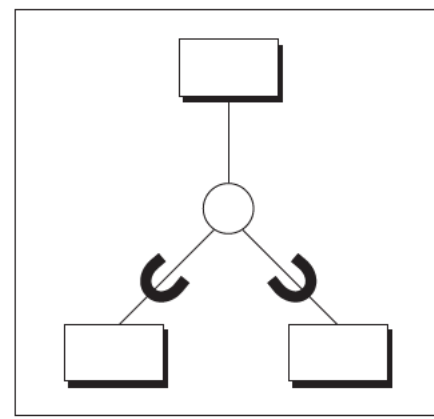
Grouping operations are the fundamental components of the entity clustering technique. They define what collections of entities and relationships comprise higher-level objects, the entity clusters. The operations are heuristic in nature and include (see Figure 4.9):

- Dominance grouping
- Abstraction grouping
- Constraint grouping
- Relationship grouping

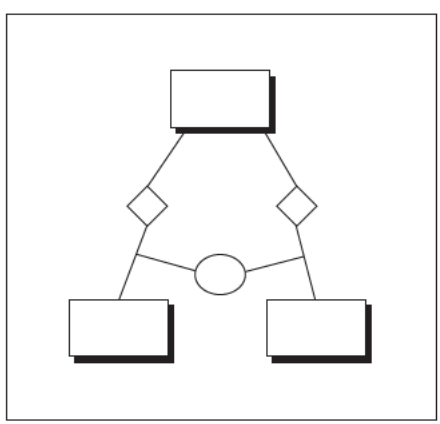
These grouping operations can be applied recursively or used in a variety of combinations to produce higher-level entity clusters, that is, clusters at any level of abstraction. An entity or entity cluster may be an



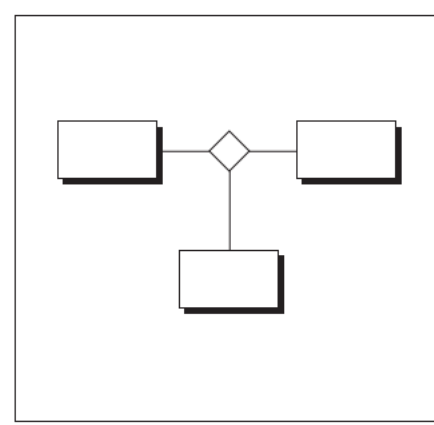
(a) Dominance grouping



(b) Abstraction grouping



(c) Constraint grouping



(d) Relationship grouping

Figure 4.9 Grouping operations

object that is subject to combinations with other objects to form the next higher level. That is, entity clusters have the properties of entities and can have relationships with any other objects at any equal or lower level. The original relationships among entities are preserved after all grouping operations, as illustrated in Figure 4.8.

Dominant objects or entities normally become obvious from the ER diagram or the relationship definitions. Each dominant object is grouped with all its related nondominant objects to form a cluster. Weak entities can be attached to an entity to make a cluster. Multilevel data objects using abstractions such as generalization and aggregation can be grouped into an entity cluster. The supertype or aggregate entity name is used as the entity cluster name. Constraint-related objects that extend the ER model to incorporate integrity constraints, such as the exclusive-OR can be grouped into an entity cluster. Additionally, ternary or higher degree relationships potentially can be grouped into an entity cluster. The cluster represents the relationship as a whole.

4.5.3 Clustering Technique

The grouping operations and their order of precedence determine the individual activities needed for clustering. We can now learn how to build a root entity cluster from the elementary entities and relationships defined in the ER modeling process. This technique assumes that a top-down analysis has been performed as part of the database requirement analysis and that the analysis has been documented so that the major functional areas and subareas are identified. Functional areas are often defined by an enterprise's important organizational units, business activities, or, possibly, by dominant applications for processing information. As an example, recall Figure 4.3 (reconstructed in Figure 4.10), which can be thought of as having three major functional areas: company organization (division, department), project management (project, skill, location, employee), and employee data (employee, manager, secretary, engineer, technician, prof-assoc, and desktop). Note that the functional areas are allowed to overlap. Figure 4.10 uses an ER diagram resulting from the database requirement analysis to show how clustering involves a series of bottom-up steps using the basic grouping operations. The following list explains these steps.

1. *Define points of grouping within functional areas*. Locate the dominant entities in a functional area through natural relationships, local *n*-ary relationships, integrity constraints, abstractions, or

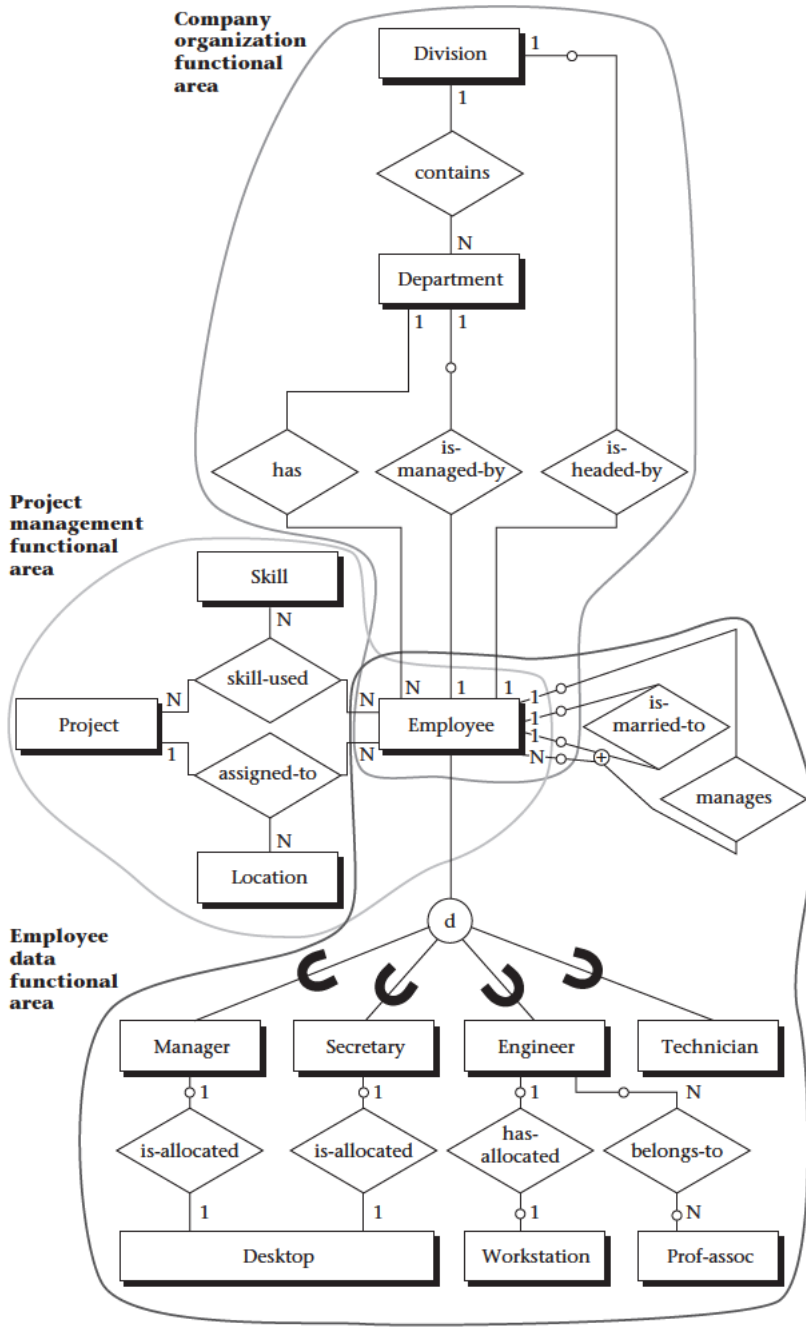


Figure 4.10 ER diagram: clustering technique

just the central focus of many simple relationships. If such points of grouping do not exist within an area, consider a functional grouping of a whole area.

2. *Form entity clusters*. Use the basic grouping operations on elementary entities and their relationships to form higher-level objects, or entity clusters. Because entities may belong to several potential clusters, we need to have a set of priorities for forming entity clusters. The following set of rules, listed in priority order, defines the set that is most likely to preserve the clarity of the conceptual model:
   a. Entities to be grouped into an entity cluster should exist within the same functional area; that is, the entire entity cluster should occur within the boundary of a functional area. For example, in Figure 4.10, the relationship between Department and Employee should not be clustered unless Employee is included in the company organization functional area with Department and Division. In another example, the relationship between the supertype Employee and its subtypes could be clustered within the employee data functional area.
   b. If a conflict in choice between two or more potential entity clusters cannot be resolved (e.g., between two constraint groupings at the same level of precedence), leave these entity clusters ungrouped within their functional area. If that functional area remains cluttered with unresolved choices, define functional subareas in which to group unresolved entities, entity clusters, and their relationships.
3. *Form higher-level entity clusters*. Apply the grouping operations recursively to any combination of elementary entities and entity clusters to form new levels of entity clusters (higher-level objects). Resolve conflicts using the same set of priority rules given in step 2. Continue the grouping operations until all the entity representations fit on a single page without undue complexity. The root entity cluster is then defined.
4. *Validate the cluster diagram*. Check for consistency of the interfaces (relationships) between objects at each level of the diagram. Verify the meaning of each level with the end users.

The result of one round of clustering is shown in Figure 4.11, where each of the clusters is shown at level 2.

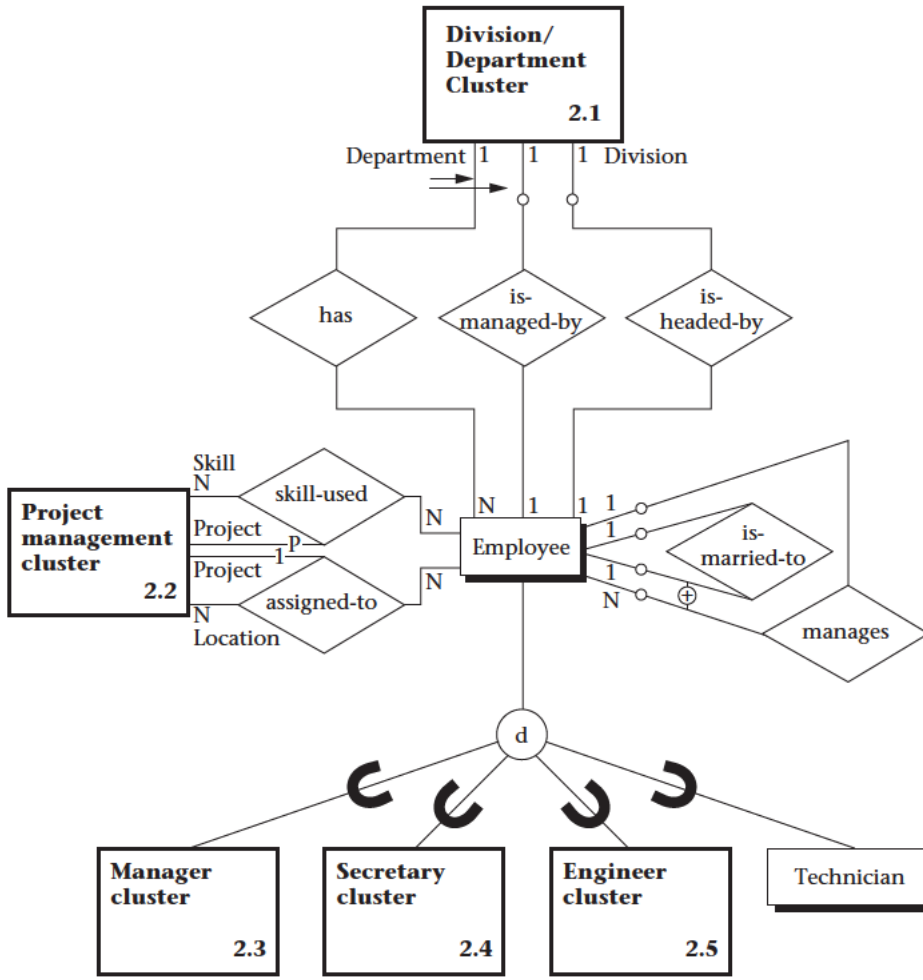


Figure 4.11 Clustering results

4.6 Summary

Conceptual data modeling, using either the ER or UML approach, is particularly useful in the early steps of the database life cycle, which involve requirements analysis and logical design. These two steps are often done simultaneously, particularly when requirements are determined from interviews with end users and modeled in terms of data-to-data relation-

ships and process-to-data relationships. The conceptual data modeling step (ER approach) involves the classification of entities and attributes first, then the identification of generalization hierarchies and other abstractions, and finally the definition of all relationships among entities. Relationships may be binary (the most common), ternary, and higher-level *n*-ary. Data modeling of individual requirements typically involves creating a different view for each end user's requirements. Then the designer must integrate those views into a global schema, so that the entire database is pictured as an integrated whole. This helps to eliminate needless redundancy—such elimination is particularly important in logical design. Controlled redundancy can be created later, at the physical design level, to enhance database performance. Finally, an entity cluster is a grouping of entities and their corresponding relationships into a higher-level abstract object. Clustering promotes the simplicity that is vital for fast end-user comprehension. In the next chapter we take the global schema produced from the conceptual data modeling and view integration steps, and we transform it into SQL tables. The SQL format is the end product of logical design, which is still independent of any particular database management system.

4.7 Literature Summary

Conceptual data modeling is defined in Tsichritzis and Lochovsky [1982], Brodie, Mylopoulos, and Schmidt [1984], Nijssen and Halpin [1989], Batini, Ceri, and Navathe [1992]. Discussion of the requirements data collection process can be found in Martin [1982], Teorey and Fry [1982], and Yao [1985]. View integration has progressed from a representation tool [Smith and Smith, 1977] to heuristic algorithms [Batini, Lenzerini, and Navathe, 1986; Elmasri and Navathe, 2003]. These algorithms are typically interactive, allowing the database designer to make decisions based on suggested alternative integration actions. A variety of entity clustering models have been defined that provide a useful foundation for the clustering technique shown here [Feldman and Miller, 1986; Dittrich, Gotthard, and Lockemann, 1986; Teorey et al., 1989].

5 Transforming the Conceptual Data Model to SQL

This chapter focuses on the database life cycle step that is of particular interest when designing relational databases: transformation of the conceptual data model to candidate tables and their definition in SQL [step II(c)]. There is a natural evolution from the ER and UML data models to a relational schema. The evolution is so natural, in fact, that it supports the contention that conceptual data modeling is an effective early step in relational database development. This contention has been proven to some extent by the widespread commercialization and use of software design tools that support not only conceptual data modeling but also the automatic conversion of these models to vendor-specific SQL table definitions and integrity constraints.

In this chapter we assume the applications to be Online Transaction Processing (OLTP). Note that Online Analytical Processing (OLAP) applications are the subject of Chapter 8.

5.1 Transformation Rules and SQL Constructs

Let's first look at the ER and UML modeling constructs in detail to see how the rules about transforming the conceptual data model to SQL tables are defined and applied. Our example is drawn from the company personnel and project conceptual schemas illustrated in Figure 4.3 (see Chapter 4).

The basic transformations can be described in terms of the three types of tables they produce:

- *SQL table with the same information content as the original entity from which it is derived.* This transformation always occurs for entities with binary relationships (associations) that are many-to-many, one-to-many on the "one" (parent) side, or one-to-one on either side; entities with binary recursive relationships that are many-to-many; and entities with any ternary or higher-degree relationship or a generalization hierarchy.
- *SQL table with the embedded foreign key of the parent entity.* This transformation always occurs for entities with binary relationships that are one-to-many for the entity on the "many" (child) side, for one-to-one relationships for one of the entities, and for each entity with a binary recursive relationship that is one-to-one or one-to-many. This is one of the two most common ways design tools handle relationships, by prompting the user to define a foreign key in the child table that matches a primary key in the parent table.
- *SQL table derived from a relationship, containing the foreign keys of all the entities in the relationship.* This transformation always occurs for relationships that are binary and many-to-many, relationships that are binary recursive and many-to-many, and all relationships that are of ternary or higher degree. This is the other most common way design tools handle relationships in the ER and UML models. A many-to-many relationship can only be defined in terms of a table that contains foreign keys that match the primary keys of the two associated entities. This new table may also contain attributes of the original relationship—for example, a relationship "enrolled-in" between two entities Student and Course might have the attributes "term" and "grade," which are associated with a particular enrollment of a student in a particular course.

The following rules apply to handling SQL null values in these transformations:

- Nulls are allowed in an SQL table for foreign keys of associated (referenced) optional entities.
- Nulls are not allowed in an SQL table for foreign keys of associated (referenced) mandatory entities.

- Nulls are not allowed for any key in an SQL table derived from a many-to-many relationship, because only complete row entries are meaningful in the table.

Figures 5.1 through 5.8 show the SQL create table statements that can be derived from each type of ER or UML model construct. Note that table names are shown in boldface for readability. Note also that in each SQL table definition, the term "primary key" represents the key of the table that is to be used for indexing and searching for data.

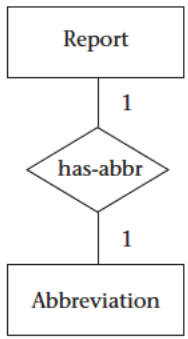
5.1.1 Binary Relationships

A one-to-one binary relationship between two entities is illustrated in Figure 5.1, parts a through c. Note that the UML equivalent binary association is given in Figure 5.2, parts a through c.

When both entities are mandatory (Figure 5.1a), each entity becomes a table, and the key of either entity can appear in the other entity's table as a foreign key. One of the entities in an optional relationship (see Department in Figure 5.1b) should contain the foreign key of the other entity in its transformed table. Employee, the other entity in Figure 5.1b, could also contain a foreign key (dept_no) with nulls allowed, but this would require more storage space because of the much greater number of Employee entity instances than Department instances. When both entities are optional (Figure 5.1c), either entity can contain the embedded foreign key of the other entity, with nulls allowed in the foreign keys.

The one-to-many relationship can be shown as either mandatory or optional on the "many" side, without affecting the transformation. On the "one" side it may be either mandatory (Figure 5.1d) or optional (Figure 5.1e). In all cases the foreign key must appear on the "many" side, which represents the child entity, with nulls allowed for foreign keys only in the optional "one" case. Foreign key constraints are set according to the specific meaning of the relationship and may vary from one relationship to another.

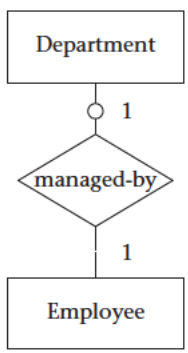
The many-to-many relationship, shown in Figure 5.1f as optional for both entities, requires a new table containing the primary keys of both entities. The same transformation applies to either the optional or mandatory case, including the fact that the not null clause must appear for the foreign keys in both cases. Note also that an optional entity means that the SQL table derived from it may have zero rows for that particular relationship. This does not affect "null" or "not null" in the table definition.



Every report has one abbreviation, and every abbreviation represents exactly one report.

```
create table report
        (report_no integer,
         report_name varchar(256),
         primary key(report_no);
create table abbreviation
        (abbr_no char(6),
         report_no integer not null unique,
         primary key (abbr_no),
         foreign key (report_no) references report
             on delete cascade on update cascade);
```

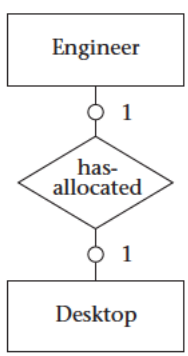
(a) One-to-one, both entities mandatory



Every department must have a manager, but an employee can be a manager of at most one department.

```
create table department
        (dept_no integer,
         dept_name char(20),
         mgr_id char(10) not null unique,
         primary key (dept_no)
         foreign key (mgr_id) references employee
             on delete set default on update cascade);
create table employee
        (emp_id char(10),
         emp_name char(20)
         primary key (emp_id));
```

(b) One-to-one, one entity optional, one mandatory

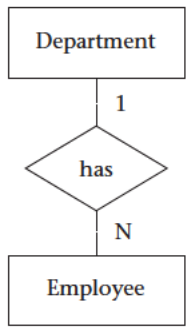


Some desktop computers are allocated to engineers, but not necessarily to all engineers.

```
create table engineer
        (emp_id char(10),
         desktop_no integer,
         primary key (emp_id));
create table desktop
        (desktop_no integer,
         emp_id char(10)
         primary key (desktop_no)
         foreign key (emp_id) references engineer
             on delete set null on update cascade);
```

(c) One-to-one, both entities optional

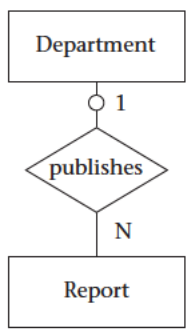
Figure 5.1 ER model: one-to-one binary relationship between two entities



Every employee works in exactly one department, and each department has at least one employee.

```
create table department
        (dept_no integer,
         dept_name char(20),
         primary key (dept_no));
create table employee
        (emp_id char(10),
         emp_name char(20),
         dept_no integer not null,
         primary key (emp_id),
         foreign key (dept_no) references department
            on delete set default on update cascade)
```

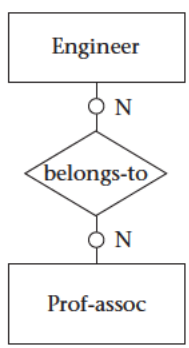
(d) One-to-many, both entities mandatory



Each department publishes one or more reports. A given report may not necessarily be published by a department.

```
create table department
        (dept_no integer,
         dept_name char(20),
         primary key (dept_no));
create table report
        (report_no integer,
         dept_no integer,
         primary key (report_no),
         foreign key (dept_no) references department
            on delete set null on update cascade);
```

(e) One-to-many, one entity optional, one mandatory

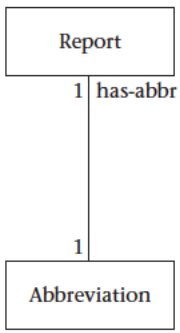


Every professional association could have none, one, or many engineer members. Each engineer could be a member of none, one, or many professional associations.

```
create table engineer
        (emp_id char(10),
         primary key (emp_id));
create table prof_assoc
        (assoc_name varchar(256),
         primary key (assoc_name));
create table belongs_to
        (emp_id char(10),
         assoc_name varchar(256),
         primary key (emp_id, assoc_name),
         foreign key (emp_id) references engineer
            on delete cascade on update cascade,
         foreign key (assoc_name) references prof_assoc
            on delete cascade on update cascade);
```

(f) Many-to-many, both entities optional

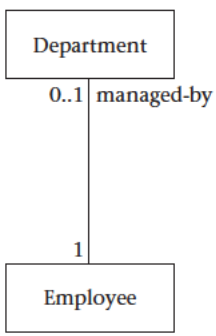
Figure 5.1 *(continued)*



Every report has one abbreviation, and every abbreviation represents exactly one report.

```
create table report
        (report_no integer,
         report_name varchar(256),
         primary key(report_no));
create table abbreviation
        (abbr_no char(6),
         report_no integer not null unique,
         primary key (abbr_no),
         foreign key (report_no) references report
             on delete cascade on update cascade);
```

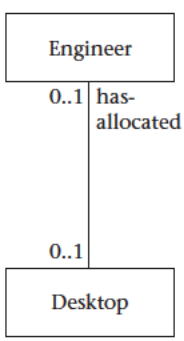
(a) one-to-one, both entities mandatory



Every department must have a manager, but an employee can be a manager of at most one department.

```
create table department
        (dept_no integer,
         dept_name char(20),
         mgr_id char(10) not null unique,
         primary key (dept_no),
         foreign key (mgr_id) references employee
             on delete set default on update cascade);
create table employee
        (emp_id char(10),
         emp_name char(20),
         primary key (emp_id));
```

(b) one-to-one, one entity optional, one mandatory

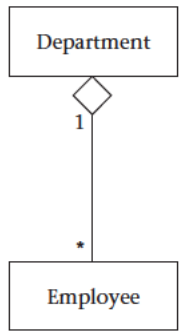


Some desktop computers are allocated to engineers, but not necessarily to all engineers.

```
create table engineer
        (emp_id char(10),
         desktop_no integer,
         primary key (emp_id));
create table desktop
        (desktop_no integer,
         emp_id char(10),
         primary key (desktop_no),
         foreign key (emp_id) references engineer
             on delete set null on update cascade);
```

(c) one-to-one, both entities optional

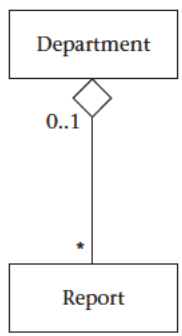
Figure 5.2 UML: one-to-one binary relationship between two entities



Every employee works in exactly one department, and each department has at least one employee.

```
create table department
        (dept_no integer,
         dept_name char(20),
         primary key (dept_no));
create table employee
        (emp_id char(10),
         emp_name char (20),
         dept_no integer not null,
         primary key (emp_id),
         foreign key (dept_no) references department
            on delete set default on update cascade);
```

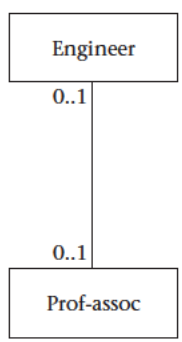
(d) one-to-many, both entities mandatory



Each department publishes one or more reports. A given report may not necessarily be published by a department.

```
create table department
        (dept_no integer,
         dept_name char(20),
         primary key (dept_no));
create table report
        (report_no integer,
         dept_no integer,
         primary key (report_no),
         foreign key (dept_no) references department
            on delete set null on update cascade);
```

(e) one-to-many, one entity optional, one mandatory



Every professional association could have none, one, or many engineer members. Each engineer could be a member of none, one, or many professional associations.

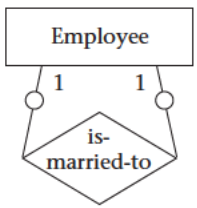
```
create table engineer
        (emp_id char(10),
         primary key (emp_id));
create table prof_assoc
        (assoc_name varchar(256),
         primary key (assoc_name));
create table belongs_to
        (emp_id char(10),
         assoc_name varchar(256),
         primary key (emp_id, assoc_name),
         foreign key (emp_id) references engineer
            on delete cascade on update cascade,
         foreign key (assoc_name) references prof_assoc
            on delete cascade on update cascade);
```

(f) many-to-many, both entities optional

Figure 5.2 *(continued)*

5.1.2 Binary Recursive Relationships

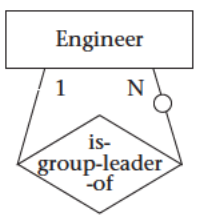
A single entity with a one-to-one relationship implies some form of entity occurrence pairing, as indicated by the relationship name. This pairing may be completely optional, completely mandatory, or neither. In all of these cases (Figure 5.3a for ER and Figure 5.4a for UML), the



Any employee is allowed to be married to another employee in this company.

```
create table employee
        (emp_id char(10),
         emp_name char(20),
         spouse_id char(10),
         primary key (emp_id),
         foreign key (spouse_id) references employee
            on delete set null on update cascade);
```

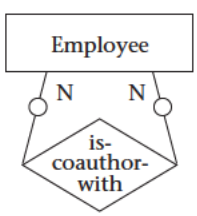
(a) One-to-one, both sides optional



Engineers are divided into groups for certain projects. Each group has a leader.

```
create table engineer
        (emp_id char(10),
         leader_id char(10) not null,
         primary key (emp_id),
         foreign key (leader_id) references engineer
            on delete set default on update cascade);
```

(b) One-to-many, one side mandatory, many side optional



Each employee has the opportunity to coauthor a report with one or more other employees, or to write the report alone.

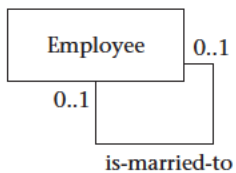
```
create table employee
        (emp_id char(10),
         emp_name char(20),
         primary key (emp_id));

create table coauthor
        (author_id char(10),
         coauthor_id char(10),
         primary key (author_id, coauthor_id),
         foreign key (author_id) references employee
            on delete cascade on update cascade,
         foreign key (coauthor_id) reference employee
            on delete cascade on update cascade);
```

(c) Many-to-many, both sides optional

Figure 5.3 ER model: binary recursive relationship

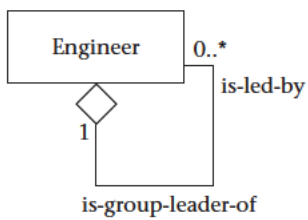
pairing entity key appears as a foreign key in the resulting table. The two key attributes are taken from the same domain but are given different names to designate their unique use. The one-to-many relationship requires a foreign key in the resulting table (Figure 5.3b). The foreign key constraints can vary with the particular relationship.



Any employee is allowed to be married to another employee in this company.

```
create table employee
        (emp_id char(10),
         emp_name char(20),
         spouse_id char(10),
         primary key (emp_id),
         foreign key (spouse_id) references employee
             on delete set null on update cascade);
```

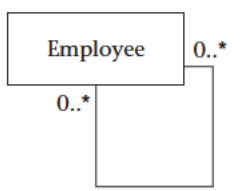
(a) one-to-one, both sides optional



Engineers are divided into groups for certain projects. Each group has a leader.

```
create table engineer
        (emp_id char(10),
         leader_id char(10) not null,
         primary key (emp_id),
         foreign key (leader_id) references engineer
             on delete set default on update cascade);
```

(b) one-to-many, one side mandatory, many side optional



Each employee has the opportunity to coauthor a report with one or more other employees, or to write the report alone.

```
create table employee
        (emp_id char(10),
         emp_name char(20),
         primary key (emp_id));
create table coauthor
        (author_id char(10),
         coauthor_id char(10),
         primary key (author_id, coauthor_id),
         foreign key (author_id) references employee
             on delete cascade on update cascade,
         foreign key (coauthor_id) references employee
             on delete cascade on update cascade);
```

(c) many-to-many, both sides optional

Figure 5.4 UML: binary recursive relationship

The many-to-many relationship is shown as optional (Figure 5.3c) and results in a new table; it could also be defined as mandatory (using the word "must" instead of "may"); both cases have the foreign keys defined as "not null." In many-to-many relationships, foreign key constraints on delete and update must always be cascade, because each entry in the SQL table depends on the current value or existence of the referenced primary key.

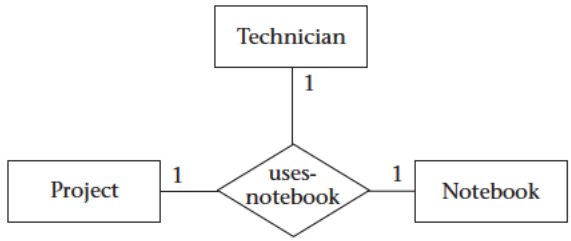
5.1.3 Ternary and n-ary Relationships

An n-ary relationship has $(n + 1)$ possible variations of connectivity: all n sides with connectivity "one;" $(n - 1)$ sides with connectivity "one," and one side with connectivity "many;" $(n - 2)$ sides with connectivity "one" and two sides with "many;" and so on until all sides are "many."

The four possible varieties of a ternary relationship are shown in Figure 5.5 for the ER model and Figure 5.6 for UML. All variations are transformed by creating an SQL table containing the primary keys of all entities; however, in each case the meaning of the keys is different. When all three relationships are "one" (Figure 5.5a), the resulting SQL table consists of three possible distinct keys. This arrangement represents the fact that three FDs are needed to describe this relationship. The optionality constraint is not used here because all n entities must participate in every instance of the relationship to satisfy the FD constraints. (See Chapter 6 for more discussion of functional dependencies.)

In general the number of entities with connectivity "one" determines the lower bound on the number of FDs. Thus, in Figure 5.3b, which is one-to-one-to-many, there are two FDs; in Figure 5.5c, which is one-to-many-to-many, there is only one FD. When all relationships are "many" (Figure 5.5d), the relationship table is all one composite key, unless the relationship has its own attributes. In that case the key is the composite of all three keys from the three associated entities.

Foreign key constraints on delete and update for ternary relationships transformed to SQL tables must always be cascade, because each entry in the SQL table depends on the current value of, or existence of, the referenced primary key.



A technician uses exactly one notebook for each project. Each notebook belongs to one technician for each project. Note that a technician may still work on many projects and maintain different notebooks for different projects.

```
create table technician (emp_id char(10),
                         primary key (emp_id));
create table project (project_name char(20),
                      primary key (project_name));
create table notebook (notebook_no integer,
                       primary key (notebook_no));
create table uses_notebook (emp_id char(10),
                            project_name char(20),
                            notebook_no integer not null,
                            primary key (emp_id, project_name),
                            foreign key (emp_id) references technician
                               on delete cascade on update cascade,
                            foreign key (project_name) references project
                               on delete cascade on update cascade,
                            foreign key (notebook_no) references notebook
                               on delete cascade on update cascade,
                            unique (emp_id, notebook_no),
                            unique (project_name, notebook_no));
```

uses_notebook

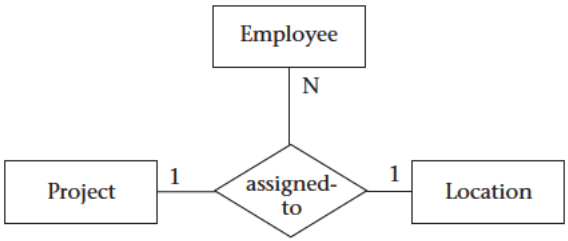
emp_id	project_name	notebook_no
35	alpha	5001
35	gamma	2008
42	delta	1004
42	epsilon	3005
81	gamma	1007
93	alpha	1009
93	beta	5001

Functional dependencies

emp_id, project_name → notebook_no
emp_id, notebook_no → project_name
project_name, notebook_no → emp_id

(a) One-to-one-to-one ternary relationship

Figure 5.5 ER model: ternary and *n*-ary relationships



Each employee assigned to a project works at only one location for that project, but can be at a different location for a different project. At a given location, an employee works on only one project. At a particular location, there can be many employees assigned to a given project.

```
create table employee (emp_id char(10),
                       emp_name char(20),
                       primary key (emp_id));
create table project (project_name char(20),
                      primary key (project_name));
create table location (loc_name char(15),
                       primary key (loc_name));
create table assigned_to (emp_id char(10),
                          project_name char(20),
                          loc_name char(15) not null,
                          primary key (emp_id, project_name),
                          foreign key (emp_id) references employee
                             on delete cascade on update cascade,
                          foreign key (project_name) references project
                             on delete cascade on update cascade,
                          foreign key (loc_name) references location
                             on delete cascade on update cascade,
                          unique (emp_id, loc_name));
```

assigned_to

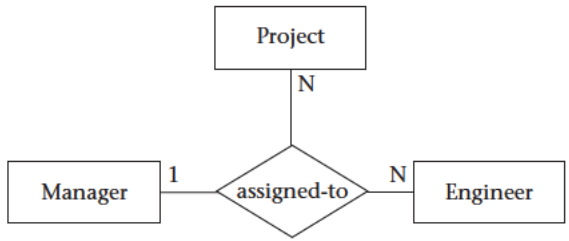
emp_id	project_name	loc_name
48101	forest	B66
48101	ocean	E71
20702	ocean	A12
20702	river	D54
51266	river	G14
51266	ocean	A12
76323	hills	B66

Functional dependencies

emp_id, loc_name → project_name
emp_id, project_name → loc_name

(b) One-to-one-to-many ternary relationships

Figure 5.5 *(continued)*



Each engineer working on a particular project has exactly one manager, but a project can have many managers and an engineer may have many managers and many projects. A manager may manage several projects.

```
create table project (project_name char(20),
                      primary key (project_name));
create table manager (mgr_id char(10),
                      primary key (mgr_id));
create table engineer (emp_id char(10),
                       primary key (emp_id));
create table manages (project_name char(20),
                      mgr_id char(10) not null,
                      emp_id char(10),
                      primary key (project_name, emp_id),
                      foreign key (project_name) references project
                         on delete cascade on update cascade,
                      foreign key (mgr_id) references manager
                         on delete cascade on update cascade,
                      foreign key (emp_id) references engineer
                         on delete cascade on update cascade);
```

manages

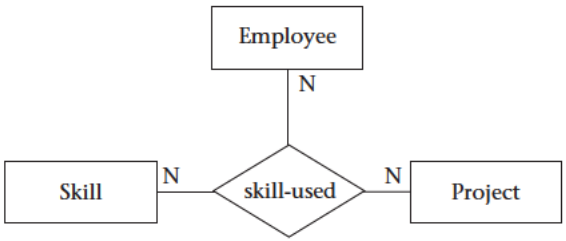
project_name	emp_id	mgr_id
alpha	4106	27
alpha	4200	27
beta	7033	32
beta	4200	14
gamma	4106	71
delta	7033	55
delta	4106	39
iota	4106	27

Functional dependency

project_name, emp_id → mgr_id

(c) One-to-many-to-many ternary relationships

Figure 5.5 *(continued)*



Employees can use different skills on any one of many projects, and each project has many employees with various skills.

```
create table employee (emp_id char(10),
                       emp_name char(20),
                       primary key (emp_id));
create table skill (skill_type char(15),
                    primary key (skill_type));
create table project (project_name char(20),
                      primary key (project_name));
create table skill_used (emp_id char(10),
                         skill_type char(15),
                         project_name char(20),
                         primary key (emp_id, skill_type, project_name),
                         foreign key (emp_id) references employee
                            on delete cascade on update cascade,
                         foreign key (skill_type) references skill
                            on delete cascade on update cascade,
                         foreign key (project_name) references project
                            on delete cascade on update cascade);
```

skill_used

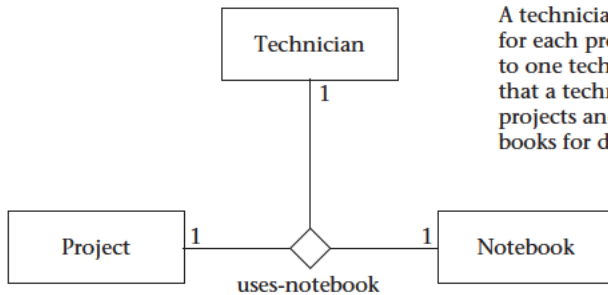
emp_id	skill_type	project_name
101	algebra	electronics
101	calculus	electronics
101	algebra	mechanics
101	geometry	mechanics
102	algebra	electronics
102	set theory	electronics
102	geometry	mechanics
105	topology	mechanics

Functional dependencies

None

(d) Many-to-many-to-many ternary relationships

Figure 5.5 *(continued)*



A technician uses exactly one notebook for each project. Each notebook belongs to one technician for each project. Note that a technician may still work on many projects and maintain different notebooks for different projects.

```
create table technician (emp_id char(10),
                         primary key (emp_id));
create table project (project_name char(20),
                      primary key (project_name));
create table notebook (notebook_no integer,
                       primary key (notebook_no));
create table uses_notebook (emp_id char(10),
                            project_name char(20),
                            notebook_no integer not null,
                            primary key (emp_id, project_name),
                            foreign key (emp_id) references technician
                               on delete cascade on update cascade,
                            foreign key (project_name) references project
                               on delete cascade on update cascade,
                            foreign key (notebook_no) references notebook
                               on delete cascade on update cascade,
                            unique (emp_id, notebook_no),
                            unique (project_name, notebook_no));
```

uses_notebook

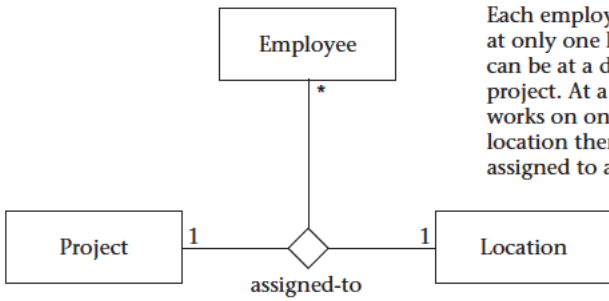
emp_id	project_name	notebook_no
35	alpha	5001
35	gamma	2008
42	delta	1004
42	epsilon	3005
81	gamma	1007
93	alpha	1009
93	beta	5001

Functional dependencies

emp_id, project_name → notebook_no
emp_id, notebook_no → project_name
project_name, notebook_no → emp_id

(a) one-to-one-to-one ternary association

Figure 5.6 UML: ternary and *n*-ary relationships



Each employee assigned to a project works at only one location for that project, but can be at a different location for a different project. At a given location, an employee works on only one project. At a particular location there can be many employees assigned to a given project.

```
create table employee (emp_id char(10),
                       emp_name char(20),
                       primary key (emp_id));
create table project (project_name char(20),
                      primary key (project_name));
create table location (loc_name char(15),
                       primary key (loc_name));
create table assigned_to (emp_id char(10),
                          project_name char(20),
                          loc_name char(15) not null,
                          primary key (emp_id, project_name),
                          foreign key (emp_id) references employee
                              on delete cascade on update cascade,
                          foreign key (project_name) references project
                              on delete cascade on update cascade,
                          foreign key (loc_name) references location
                              on delete cascade on update cascade,
                          unique (emp_id, loc_name));
```

assigned_to

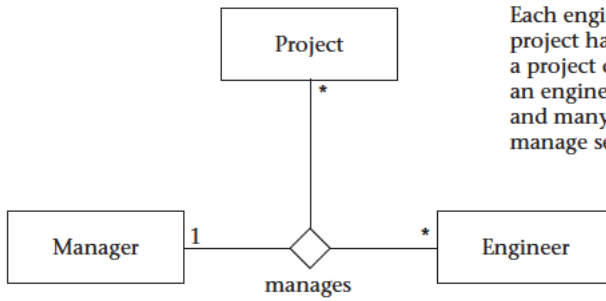
emp_id	project_name	loc_name
48101	forest	B66
48101	ocean	E71
20702	ocean	A12
20702	river	D54
51266	river	G14
51266	ocean	A12
76323	hills	B66

Functional dependencies

emp_id, loc_name → project_name
emp_id, project_name → loc_name

(b) one-to-one-to-many ternary associations

Figure 5.6 *(continued)*



Each engineer working on a particular project has exactly one manager, but a project can have many managers and an engineer may have many managers and many projects. A manager may manage several projects.

```
create table project (project_name char(20),
                      primary key (project_name));
create table manager (mgr_id char(10),
                      primary key (mgr_id));
create table engineer (emp_id char(10),
                       primary key (emp_id));
create table manages (project_name char(20),
                      mgr_id char(10) not null,
                      emp_id char(10),
                      primary key (project_name, emp_id),
                      foreign key (project_name) references project
                          on delete cascade on update cascade,
                      foreign key (mgr_id) references manager
                          on delete cascade on update cascade,
                      foreign key (emp_id) references engineer
                          on delete cascade on update cascade);
```

manages

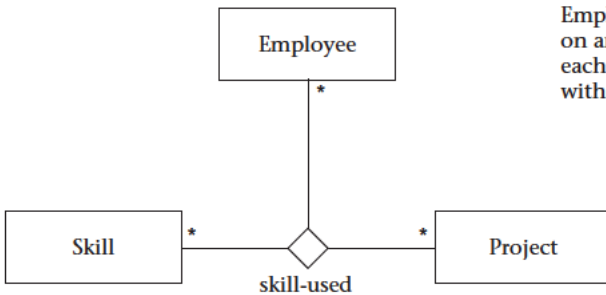
project_name	emp_id	mgr_id
alpha	4106	27
alpha	4200	27
beta	7033	32
beta	4200	14
gamma	4106	71
delta	7033	55
delta	4106	39
iota	4106	27

Functional dependency

project_name, emp_id → mgr_id

(c) one-to-many-to-many ternary association

Figure 5.6 *(continued)*



Employees can use different skills on any one of many projects, and each project has many employees with various skills.

```
create table employee (emp_id char(10),
                       emp_name char(20),
                       primary key (emp_id));
create table skill (skill_type char(15),
                    primary key (skill_type));
create table project (project_name char(20),
                      primary key (project_name));
create table skill_used (emp_id char(10),
                         skill_type char(15),
                         project_name char(20),
                         primary key (emp_id, skill_type, project_name),
                         foreign key (emp_id) references employee
                            on delete cascade on update cascade,
                         foreign key (skill_type) references skill
                            on delete cascade on update cascade,
                         foreign key (project_name) references project
                            on delete cascade on update cascade);
```

skill_used

emp_id	skill_type	project_name
101	algebra	electronics
101	calculus	electronics
101	algebra	mechanics
101	geometry	mechanics
102	algebra	electronics
102	set-theory	electronics
102	geometry	mechanics
105	topology	mechanics

Functional dependencies

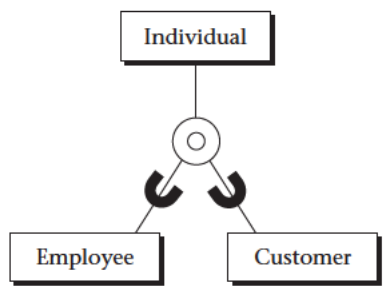
None

(d) many-to-many-to-many ternary association

Figure 5.6 *(continued)*

5.1.4 Generalization and Aggregation

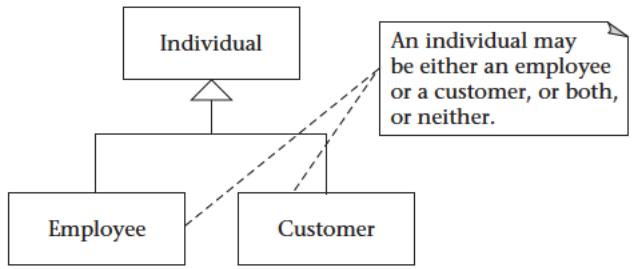
The transformation of a generalization abstraction can produce separate SQL tables for the generic or supertype entity and each of the subtypes (Figure 5.7 for the ER model and Figure 5.8 for UML). The table derived from the supertype entity contains the supertype entity key and all common attributes. Each table derived from subtype entities contains the supertype entity key and only the attributes that are specific to that subtype. Update integrity is maintained by requiring all insertions and deletions to occur in both the supertype table and relevant subtype table—that is, the foreign key constraint cascade must be used. If the update is to the primary key of the supertype table, then all subtype tables, as well as the supertype table, must be updated. An update to a nonkey attribute affects either the supertype or one subtype table, but not both. The



An individual may be either an employee or a customer, or both, or neither.

```
create table individual (indiv_id char(10),
                         indiv_name char(20),
                         indiv_addr char(20),
                         primary key (indiv_id));
create table employee (emp_id char(10),
                       job_title char(15),
                       primary key (emp_id),
                       foreign key (emp_id) references individual
                          on delete cascade on update cascade);
create table customer (cust_no char(10),
                       cust_credit char(12),
                       primary key (cust_no),
                       foreign key (cust_no) references individual
                          on delete cascade on update cascade);
```

Figure 5.7 ER model: generalization and aggregation




```
create table individual (indiv_id char(10),
                         indiv_name char(20),
                         indiv_addr char(20),
                         primary key (indiv_id));
create table employee (emp_id char(10),
                       job_title char(15),
                       primary key (emp_id),
                       foreign key (emp_id) references individual
                          on delete cascade on update cascade);
create table customer (cust_no char(10),
                       cust_credit char(12),
                       primary key (cust_no),
                       foreign key (cust_no) references individual
                          on delete cascade on update cascade);
```

Figure 5.8 UML: generalization and aggregation

transformation rules (and integrity rules) are the same for both the disjoint and overlapping subtype generalizations.

Another approach is to have a single table that includes all attributes from the supertype and subtypes (the whole hierarchy in one table), with nulls used when necessary. A third possibility is one table for each subtype, pushing down the common attributes into the specific subtypes. There are advantages and disadvantages to each of these three approaches. Several software tools now support all three options [Fowler 2003; Ambler, 2003].

Database practitioners often add a discriminator to the supertype when they implement generalization. The discriminator is an attribute that has a separate value for each subtype and indicates which subtype to use to get further information. This approach works, up to a point. However, there are situations requiring multiple levels of supertypes and subtypes, where more than one discriminator may be required.

The transformation of an aggregation abstraction also produces a separate table for the supertype entity and each subtype entity. However,

there are no common attributes and no integrity constraints to maintain. The main function of aggregation is to provide an abstraction to aid the view integration process. In UML, aggregation is a composition relationship, not a type relationship, which corresponds to a weak entity [Muller, 1999].

5.1.5 Multiple Relationships

Multiple relationships among *n* entities are always considered to be completely independent. One-to-one, one-to-many binary, or binary recursive relationships resulting in tables that are either equivalent or differ only in the addition of a foreign key can simply be merged into a single table containing all the foreign keys. Many-to-many or ternary relationships that result in SQL tables tend to be unique and cannot be merged.

5.1.6 Weak Entities

Weak entities differ from entities only in their need for keys from other entities to establish their uniqueness. Otherwise, they have the same transformation properties as entities, and no special rules are needed. When a weak entity is already derived from two or more entities in the ER diagram, it can be directly transformed into a table without further change.

5.2 Transformation Steps

The following list summarizes the basic transformation steps from an ER diagram to SQL tables:

- Transform each entity into a table containing the key and non-key attributes of the entity
- Transform every many-to-many binary or binary recursive relationship into a table with the keys of the entities and the attributes of the relationship
- Transform every ternary or higher-level *n*-ary relationship into a table

Now let us study each step in turn.

5.2.1 Entity Transformation

If there is a one-to-many relationship between two entities, add the key of the entity on the "one" side (the parent) into the child table as a foreign key. If there is a one-to-one relationship between one entity and another entity, add the key of one of the entities into the table for the other entity, thus changing it to a foreign key. The addition of a foreign key due to a one-to-one relationship can be made in either direction. One strategy is to maintain the most natural parent-child relationship by putting the parent key into the child table. Another strategy is based on efficiency: add the foreign key to the table with fewer rows.

Every entity in a generalization hierarchy is transformed into a table. Each of these tables contains the key of the supertype entity; in reality, the subtype primary keys are foreign keys as well. The supertype table also contains nonkey values that are common to all the relevant entities; the other tables contain nonkey values specific to each subtype entity.

SQL constructs for these transformations may include constraints for not null, unique, and foreign key. A primary key must be specified for each table, either explicitly from among the keys in the ER diagram or by taking the composite of all attributes as the default key. Note that the primary key designation implies that the attribute is not null or unique. It is important to note, however, that not all DBMSs follow the ANSI standard in this regard—it may be possible in some systems to create a primary key that can be null. We recommend that you specify "not null" explicitly for all key attributes.

5.2.2 Many-to-Many Binary Relationship Transformation

In this step, every many-to-many binary relationship is transformed into a table containing the keys of the entities and the attributes of the relationship. The resulting table will show the correspondence between specific instances of one entity and those of another entity. Any attribute of this correspondence, such as the elected office an engineer has in a professional association (Figure 5.1f), is considered intersection data and is added to the table as a nonkey attribute.

SQL constructs for this transformation may include constraints for not null. The unique constraint is not used here because all keys are composites of the participating primary keys of the associated entities in the relationship. The constraints for primary key and foreign key are required, because a table is defined as containing a composite of the primary keys of the associated entities.

5.2.3 Ternary Relationship Transformation

In this step, every ternary (or higher *n*-ary) relationship is transformed into a table. Ternary or higher *n*-ary relationships are defined as a collection of the *n* primary keys in the associated entities in that relationship, with possibly some nonkey attributes that are dependent on the key formed by the composite of those *n* primary keys.

SQL constructs for this transformation must include constraints for not null, since optionality is not allowed. The unique constraint is not used for individual attributes, because all keys are composites of the participating primary keys of the associated entities in the relationship. The constraints for primary key and foreign key are required because a table is defined as a composite of the primary keys of the associated entities. The unique clause must also be used to define alternate keys that often occur with ternary relationships. Note that a table derived from an *n*-ary relationship has *n* foreign keys.

5.2.4 Example of ER-to-SQL Transformation

ER diagrams for the company personnel and project database (Chapter 4) can be transformed to SQL tables. A summary of the transformation of entities and relationships to SQL tables is illustrated in the following list.

SQL tables derived directly from entities (see Figure 4.3d):

division	**secretary**	**project**
department	**engineer**	**location**
employee	**technician**	**prof_assoc**
manager	**skill**	**desktop**

SQL tables derived from many-to-many binary or many-to-many binary recursive relationships:

- **belongs_to**

SQL tables transformed from ternary relationships:

- **skill_used**
- **assigned_to**

5.3 Summary

Entities, attributes, and relationships in the ER model and classes, attributes, and associations in UML can be transformed directly into SQL (SQL-99) table definitions with some simple rules. Entities are transformed into tables, with all attributes mapped one-to-one to table attributes. Tables representing entities that are the child ("many" side) of a parent-child (one-to-many or one-to-one) relationship must also include, as a foreign key, the primary key of the parent entity. A many-to-many relationship is transformed into a table that contains the primary keys of the associated entities as its composite primary key; the components of that key are also designated as foreign keys in SQL. A ternary or higher-level *n*-ary relationship is transformed into a table that contains the primary keys of the associated entities; these keys are designated as foreign keys in SQL. A subset of those keys can be designated as the primary key, depending on the functional dependencies associated with the relationship. Rules for generalization require the inheritance of the primary key from the supertype to the subtype entities when transformed into SQL tables. Optionality constraints in the ER or UML diagrams translate into nulls allowed in the relational model when applied to the "one" side of a relationship. In SQL, the lack of an optionality constraint determines the not null designation in the create table definition.

5.4 Literature Summary

Definition of the basic transformations from the ER model to tables is covered in McGee [1974], Wong and Katz [1979], Sakai [1983], Martin [1983], Hawryszkiewyck [1984], Jajodia and Ng [1984], and for UML in Muller [1999].

6 Normalization

This chapter focuses on the fundamentals of normal forms for relational databases and the database design step that normalizes the candidate tables [step II(d) of the database design life cycle]. It also investigates the equivalence between the conceptual data model (e.g., the ER model) and normal forms for tables. As we go through the examples in this chapter, it should become obvious that good, thoughtful design of a conceptual model will result in databases that are either already normalized or can be easily normalized with minor changes. This truth illustrates the beauty of the conceptual modeling approach to database design, in that the experienced relational database designer will develop a natural gravitation toward a normalized model from the beginning.

For most database practitioners, Sections 6.1 through 6.4 cover the critical normalization needed for everyday use, through Boyce-Codd normal form (BCNF). Section 6.5 covers the higher normal forms of mostly theoretical interest; however, we do show the equivalency between higher normal forms and ternary relationships in the ER model and UML for the interested reader.

6.1 Fundamentals of Normalization

Relational database tables, whether they are derived from ER or UML models, sometimes suffer from some rather serious problems in terms of

Sales

product_name	order_no	cust_name	cust_addr	credit	date	sales_name
vacuum cleaner	1458	Dave Bachmann	Austin	6	1-3-03	Carl Bloch
computer	2730	Qiang Zhu	Plymouth	10	4-15-05	Ted Hanss
refrigerator	2460	Mike Stolarchuck	Ann Arbor	8	9-12-04	Dick Phillips
DVD player	519	Peter Honeyman	Detroit	3	12-5-04	Fred Remley
radio	1986	Charles Antonelli	Chicago	7	5-10-05	R. Metz
CD player	1817	C.V. Ravishankar	Mumbai	8	8-3-02	Paul Basile
vacuum cleaner	1865	Charles Antonelli	Chicago	7	10-1-04	Carl Bloch
vacuum cleaner	1885	Betsy Karmeisool	Detroit	8	4-19-99	Carl Bloch
refrigerator	1943	Dave Bachmann	Austin	6	1-4-04	Dick Phillips
television	2315	Sakti Pramanik	East Lansing	6	3-15-04	Fred Remley

Figure 6.1 Single table database

performance, integrity and maintainability. For example, when the entire database is defined as a single large table, it can result in a large amount of redundant data and lengthy searches for just a small number of target rows. It can also result in long and expensive updates, and deletions in particular can result in the elimination of useful data as an unwanted side effect.

Such a situation is shown in Figure 6.1, where products, salespersons, customers, and orders are all stored in a single table called Sales. In this table, we see that certain product and customer information is stored redundantly, wasting storage space. Certain queries, such as "Which customers ordered vacuum cleaners last month?" would require a search of the entire table. Also, updates such as changing the address of the customer Dave Bachmann would require changing many rows. Finally, deleting an order by a valued customer such as Qiang Zhu (who bought an expensive computer), if that is his only outstanding order, deletes the only copy of his address and credit rating as a side effect. Such information may be difficult (or sometimes impossible) to recover. These problems also occur for situations in which the database has already been set up as a collection of many tables, but some of the tables are still too large.

If we had a method of breaking up such a large table into smaller tables so that these types of problems would be eliminated, the database would be much more efficient and reliable. Classes of relational database schemes or table definitions, called normal forms, are commonly used to accomplish this goal. The creation of a normal form database table is called normalization. Normalization is accomplished by analyzing the

interdependencies among individual attributes associated with those tables and taking projections (subsets of columns) of larger tables to form smaller ones.

Let us first review the basic normal forms, which have been well established in the relational database literature and in practice.

6.1.1 First Normal Form

> ***Definition.*** A table is in *first normal form (1NF)* if and only if all columns contain only atomic values, that is, each column can have only one value for each row in the table.

Relational database tables, such as the **Sales** table illustrated in Figure 6.1, have only atomic values for each row and for each column. Such tables are considered to be in first normal form, the most basic level of normalized tables.

To better understand the definition for 1NF, it helps to know the difference between a domain, an attribute, and a column. A *domain* is the set of all possible values for a particular type of attribute, but may be used for more than one attribute. For example, the domain of people's names is the underlying set of all possible names that could be used for either customer-name or salesperson-name in the database table in Figure 6.1. Each column in a relational table represents a single attribute, but in some cases more than one column may refer to different attributes from the same domain. When this occurs, the table is still in 1NF because the values in the table are still atomic. In fact, standard SQL assumes only atomic values and a relational table is by default in 1NF. A nice explanation of this is given in Muller [1999].

6.1.2 Superkeys, Candidate Keys, and Primary Keys

A table in 1NF often suffers from data duplication, update performance, and update integrity problems, as noted above. To understand these issues better, however, we must define the concept of a key in the context of normalized tables. A *superkey* is a set of one or more attributes, which, when taken collectively, allows us to identify uniquely an entity or table. Any subset of the attributes of a superkey that is also a superkey, and not reducible to another superkey, is called a *candidate key*. A *primary* key is selected arbitrarily from the set of candidate keys to be used in an index for that table.

Report

report_no	editor	dept_no	dept_name	dept_addr	author_id	author_name	author_addr
4216	woolf	15	design	argus1	53	mantei	cs-tor
4216	woolf	15	design	argus1	44	bolton	mathrev
4216	woolf	15	design	argus1	71	koenig	mathrev
5789	koenig	27	analysis	argus2	26	fry	folkstone
5789	koenig	27	analysis	argus2	38	umar	prise
5789	koenig	27	analysis	argus2	71	koenig	mathrev

Figure 6.2 **Report** table

As an example, in Figure 6.2 a composite of all the attributes of the table forms a superkey because duplicate rows are not allowed in the relational model. Thus, a trivial superkey is formed from the composite of all attributes in a table. Assuming that each department address (dept_addr) in this table is single valued, we can conclude that the composite of all attributes except dept_addr is also a superkey. Looking at smaller and smaller composites of attributes and making realistic assumptions about which attributes are single valued, we find that the composite (report_no, author_id) uniquely determines all the other attributes in the table and is therefore a superkey. However, neither report_no nor author_id alone can determine a row uniquely, and the composite of these two attributes cannot be reduced and still be a superkey. Thus, the composite (report_no, author_id) becomes a candidate key. Since it is the only candidate key in this table, it also becomes the primary key.

A table can have more than one candidate key. If, for example, in Figure 6.2, we had an additional column for author_ssn, and the composite of report_no and author_ssn uniquely determine all the other attributes of the table, then both (report_no, author_id) and (report_no, author_ssn) would be candidate keys. The primary key would then be an arbitrary choice between these two candidate keys.

Other examples of multiple candidate keys can be seen in Figure 5.5 (see Chapter 5). In Figure 5.5a the table **uses_notebook** has three candidate keys: (emp_id, project_name), (emp_id, notebook_no), and (project_name, notebook_no); and in Figure 5.5b the table **assigned_to** has two candidate keys: (emp_id, loc_name) and (emp_id, project_name). Figures 5.5c and 5.5d each have only a single candidate key.

6.1.3 Second Normal Form

To explain the concept of second normal form (2NF) and higher, we introduce the concept of functional dependence, which was briefly described in Chapter 2. The property of one or more attributes that uniquely determine the value of one or more other attributes is called *functional dependence*. Given a table (R), a set of attributes (B) is functionally dependent on another set of attributes (A) if, at each instant of time, each A value is associated with only one B value. Such a functional dependence is denoted by A -> B. In the preceding example from Figure 6.2, let us assume we are given the following functional dependencies for the table **report**:

report: report_no -> editor, dept_no
dept_no -> dept_name, dept_addr
author_id -> author_name, author_addr

Definition. A table is in second normal form (2NF) if and only if it is in 1NF and every nonkey attribute is fully dependent on the primary key. An attribute is fully dependent on the primary key if it is on the right side of an FD for which the left side is either the primary key itself or something that can be derived from the primary key using the transitivity of FDs.

An example of a transitive FD in **report** is the following:

report_no -> dept_no
dept_no -> dept_name

Therefore we can derive the FD (report_no -> dept_name), since dept_name is transitively dependent on report_no.

Continuing our example, the composite key in Figure 6.2, (report_no, author_id), is the only candidate key and is therefore the primary key. However, there exists one FD (dept_no -> dept_name, dept_addr) that has no component of the primary key on the left side, and two FDs (report_no -> editor, dept_no and author_id -> author_name, author_addr) that contain one component of the primary key on the left side, but not both components. As such, **report** does not satisfy the condition for 2NF for any of the FDs.

Consider the disadvantages of 1NF in table **report**. Report_no, editor, and dept_no are duplicated for each author of the report. Therefore, if the editor of the report changes, for example, several rows must be updated. This is known as the *update anomaly*, and it represents a potential degradation of performance due to the redundant updating. If a new editor is to be added to the table, it can only be done if the new editor is editing a report: both the report number and editor number must be known to add a row to the table, because you cannot have a primary key with a null value in most relational databases. This is known as the *insert anomaly*. Finally, if a report is withdrawn, all rows associated with that report must be deleted. This has the side effect of deleting the information that associates an author_id with author_name and author_addr. Deletion side effects of this nature are known as *delete anomalies*. They represent a potential loss of integrity, because the only way the data can be restored is to find the data somewhere outside the database and insert it back into the database. All three of these anomalies represent problems to database designers, but the delete anomaly is by far the most serious because you might lose data that cannot be recovered.

These disadvantages can be overcome by transforming the 1NF table into two or more 2NF tables by using the projection operator on the subset of the attributes of the 1NF table. In this example we project **report** over report_no, editor, dept_no, dept_name, and dept_addr to form **report1**; and project **report** over author_id, author_name, and author_addr to form **report2**; and finally project **report** over report_no and author_id to form **report3**. The projection of **report** into three smaller tables has preserved the FDs and the association between report_no and author_no that was important in the original table. Data for the three tables is shown in Figure 6.3. The FDs for these 2NF tables are:

report1: report_no -> editor, dept_no
dept_no -> dept_name, dept_addr

report2: author_id -> author_name, author_addr

report3: report_no, author_id is a candidate key (no FDs)

We now have three tables that satisfy the conditions for 2NF, and we have eliminated the worst problems of 1NF, especially integrity (the delete anomaly). First, editor, dept_no, dept_name, and dept_addr are no longer duplicated for each author of a report. Second, an editor change results in only an update to one row for **report1**. And third, the most important, the deletion of the report does not have the side effect of deleting the author information.

Report 1

report_no	editor	dept_no	dept_name	dept_addr
4216	woolf	15	design	argus 1
5789	koenig	27	analysis	argus 2

Report 2

author_id	author_name	author_addr
53	mantei	cs-tor
44	bolton	mathrev
71	koenig	mathrev
26	fry	folkstone
38	umar	prise
71	koenig	mathrev

Report 3

report_no	author_id
4216	53
4216	44
4216	71
5789	26
5789	38
5789	71

Figure 6.3 2NF tables

Not all performance degradation is eliminated, however; report_no is still duplicated for each author, and deletion of a report requires updates to two tables (**report1** and **report3**) instead of one. However, these are minor problems compared to those in the 1NF table **report**.

Note that these three report tables in 2NF could have been generated directly from an ER (or UML) diagram that equivalently modeled this situation with entities Author and Report and a many-to-many relationship between them.

6.1.4 Third Normal Form

The 2NF tables we established in the previous section represent a significant improvement over 1NF tables. However, they still suffer from

Report 11

report_no	editor	dept_no
4216	woolf	15
5789	koenig	27

Report 12

dept_no	dept_name	dept_addr
15	design	argus 1
27	analysis	argus 2

Report 2

author_id	author_name	author_addr
53	mantei	cs-tor
44	bolton	mathrev
71	koenig	mathrev
26	fry	folkstone
38	umar	prise
71	koenig	mathrev

Report 3

report_no	author_id
4216	53
4216	44
4216	71
5789	26
5789	38
5789	71

Figure 6.4 3NF tables

the same types of anomalies as the 1NF tables although for different reasons associated with transitive dependencies. If a transitive (functional) dependency exists in a table, it means that two separate facts are represented in that table, one fact for each functional dependency involving a different left side. For example, if we delete a report from the database, which involves deleting the appropriate rows from **report1** and **report3** (see Figure 6.3), we have the side effect of deleting the association between dept_no, dept_name, and dept_addr as well. If we could project table **report1** over report_no, editor, and dept_no to form table **report11**, and project **report1** over dept_no, dept_name, and dept_addr to form table **report12**, we could eliminate this problem. Example tables for **report11** and **report12** are shown in Figure 6.4.

Definition. A table is in *third normal form (3NF)* if and only if for every nontrivial functional dependency X->A, where X and A are either simple or composite attributes, one of two conditions must hold. Either attribute X is a superkey, or attribute A is a member of a candidate key. If attribute A is a member of a candidate key, A is called a prime attribute. Note: a trivial FD is of the form YZ->Z.

In the preceding example, after projecting **report1** into **report11** and **report12** to eliminate the transitive dependency report_no -> dept_no -> dept_name, dept_addr, we have the following 3NF tables and their functional dependencies (and example data in Figure 6.4):

report11: report_no -> editor, dept_no
report12: dept_no -> dept_name, dept_addr
report2: author_id -> author_name, author_addr
report3: report_no, author_id is a candidate key (no FDs)

6.1.5 Boyce-Codd Normal Form

3NF, which eliminates most of the anomalies known in databases today, is the most common standard for normalization in commercial databases and CASE tools. The few remaining anomalies can be eliminated by the Boyce-Codd normal form (BCNF) and higher normal forms defined here and in Section 6.5. BCNF is considered to be a strong variation of 3NF.

Definition. A table **R** is in *Boyce-Codd normal form (BCNF)* if for every nontrivial FD X->A, X is a superkey.

BCNF is a stronger form of normalization than 3NF because it eliminates the second condition for 3NF, which allowed the right side of the FD to be a prime attribute. Thus, every left side of an FD in a table must be a superkey. Every table that is BCNF is also 3NF, 2NF, and 1NF, by the previous definitions.

The following example shows a 3NF table that is not BCNF. Such tables have delete anomalies similar to those in the lower normal forms.

Assertion 1. For a given team, each employee is directed by only one leader. A team may be directed by more than one leader.

emp_name, team_name -> leader_name

Assertion 2. Each leader directs only one team.

leader_name -> team_name

This table is 3NF with a composite candidate key emp_id, team_id:

team:	emp_name	team_name	leader_name
	Sutton	Hawks	Wei
	Sutton	Condors	Bachmann
	Niven	Hawks	Wei
	Niven	Eagles	Makowski
	Wilson	Eagles	DeSmith

The **team** table has the following delete anomaly: if Sutton drops out of the Condors team, then we have no record of Bachmann leading the Condors team. As shown by Date [1999], this type of anomaly cannot have a lossless decomposition and preserve all FDs. A lossless decomposition requires that when you decompose the table into two smaller tables by projecting the original table over two overlapping subsets of the scheme, the natural join of those subset tables must result in the original table without any extra unwanted rows. The simplest way to avoid the delete anomaly for this kind of situation is to create a separate table for each of the two assertions. These two tables are partially redundant, enough so to avoid the delete anomaly. This decomposition is lossless (trivially) and preserves functional dependencies, but it also degrades update performance due to redundancy, and necessitates additional storage space. The trade-off is often worth it because the delete anomaly is avoided.

6.2 The Design of Normalized Tables: A Simple Example

The example in this section is based on the ER diagram in Figure 6.5 and the FDs given below. In general, FDs can be given explicitly, derived from the ER diagram, or derived from intuition (that is, from experience with the problem domain).

1. emp_id, start_date -> job_title, end_date
2. emp_id -> emp_name, phone_no, office_no, proj_no, proj_name, dept_no
3. phone_no -> office_no

4. proj_no -> proj_name, proj_start_date, proj_end_date
5. dept_no -> dept_name, mgr_id
6. mgr_id -> dept_no

Our objective is to design a relational database schema that is normalized to at least 3NF and, if possible, minimize the number of tables required. Our approach is to apply the definition of third normal form (3NF) in Section 6.1.4 to the FDs given above, and create tables that satisfy the definition.

If we try to put FDs 1 through 6 into a single table with the composite candidate key (and primary key) (emp_id, start_date), we violate the 3NF definition, because FDs 2 through 6 involve left sides of FDs that are not superkeys. Consequently, we need to separate 1 from the rest of the FDs. If we then try to combine 2 through 6, we have many transitivities. Intuitively, we know that 2, 3, 4, and 5 must be separated into different tables because of transitive dependencies. We then must decide whether 5 and 6 can be combined without loss of 3NF; this can be done because mgr_id and dept_no are mutually dependent and both attributes are

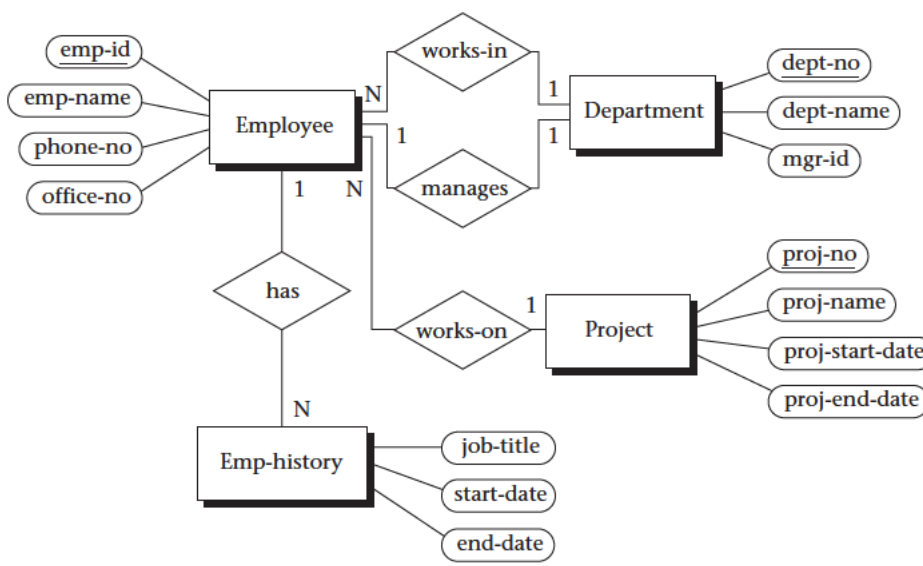


Figure 6.5 ER diagram for employee database

superkeys in a combined table. Thus, we can define the following tables by appropriate projections from 1 through 6.

emp_hist:	emp_id, start_date -> job_title, end_date
employee:	emp_id -> emp_name, phone_no, proj_no, dept_no
phone:	phone_no -> office_no
project:	proj_no -> proj_name, proj_start_date, proj_end_date
department:	dept_no -> dept_name, mgr_id mgr_id -> dept_no

This solution, which is BCNF as well as 3NF, maintains all the original FDs. It is also a minimum set of normalized tables. In Section 6.4, we will look at a formal method of determining a minimum set that we can apply to much more complex situations.

Alternative designs may involve splitting tables into partitions for volatile (frequently updated) and passive (rarely updated) data, consolidating tables to get better query performance, or duplicating data in different tables to get better query performance without losing integrity. In summary, the measures we use to assess the trade-offs in our design are:

- Query performance (time)
- Update performance (time)
- Storage performance (space)
- Integrity (avoidance of delete anomalies)

6.3 Normalization of Candidate Tables Derived from ER Diagrams

Normalization of candidate tables [step II(d) in the database life cycle] is accomplished by analyzing the FDs associated with those tables: explicit FDs from the database requirements analysis (Section 6.2), FDs derived from the ER diagram, and FDs derived from intuition.

Primary FDs represent the dependencies among the data elements that are keys of entities, that is, the interentity dependencies. *Secondary FDs*, on the other hand, represent dependencies among data elements that comprise a single entity, that is, the intraentity dependencies. Typically, primary FDs are derived from the ER diagram, and secondary FDs are obtained explicitly from the requirements analysis. If the ER constructs do

Table 6.1 Primary FDs Derivable from ER Relationship Constructs

Degree	*Connectivity*	*Primary FD*
Binary or	one-to-one	2 ways: key(one side) -> key(one side)
Binary	one-to-many	key(many side) -> key(one side)
Recursive	many-to-many	none (composite key from both sides)
Ternary	one-to-one-to-one	3 ways: key(one), key(one) -> key(one)
	one-to-one-to-many	2 ways: key(one), key(many) -> key(one)
	one-to-many-to-many	1 way: key(many), key(many) -> key(one)
	many-to-many-to-many	none (composite key from all 3 sides)
Generalization	none	none (secondary FD only)

not include nonkey attributes used in secondary FDs, the data requirements specification or data dictionary must be consulted. Table 6.1 shows the types of primary FDs derivable from each type of ER construct.

Each candidate table will typically have several primary and secondary FDs uniquely associated with it that determine the current degree of normalization of the table. Any of the well-known techniques for increasing the degree of normalization can be applied to each table to the desired degree stated in the requirements specification. Integrity is maintained by requiring the normalized table schema to include all data dependencies existing in the candidate table schema.

Any table **B** that is subsumed by another table **A** can potentially be eliminated. Table **B** is subsumed by another table **A** when all the attributes in **B** are also contained in **A**, and all data dependencies in **B** also occur in **A**. As a trivial case, any table containing only a composite key and no nonkey attributes is automatically subsumed by any other table containing the same key attributes, because the composite key is the weakest form of data dependency. If, however, tables **A** and **B** represent the supertype and subtype cases, respectively, of entities defined by the generalization abstraction, and **A** subsumes **B** because **B** has no additional specific attributes, the designer must collect and analyze additional information to decide whether or not to eliminate **B**.

A table can also be subsumed by the construction of a join of two other tables (a "join" table). When this occurs, the elimination of a sub-

sumed table may result in the loss of retrieval efficiency, although storage and update costs will tend to be decreased. This trade-off must be further analyzed during physical design with regard to processing requirements to determine whether elimination of the subsumed table is reasonable.

To continue our example company personnel and project database, we want to obtain the primary FDs by applying the rules in Table 6.1 to each relationship in the ER diagram in Figure 4.3. The results are shown in Table 6.2.

Next we want to determine the secondary FDs. Let us assume that the dependencies in Table 6.3 are derived from the requirements specification and intuition.

Normalization of the candidate tables is accomplished next. In Table 6.4 we bring together the primary and secondary FDs that apply to each candidate table. We note that for each table except **employee**, all attributes are functionally dependent on the primary key (denoted by the left side of the FDs) and are thus BCNF. In the case of table **employee**, we note that spouse_id determines emp_id and emp_id is the primary key; thus spouse_id can be shown to be a superkey (see Superkey Rule 2 in Section 6.4). Therefore, **employee** is found to be BCNF.

Table 6.2 Primary FDs Derived from the ER Diagram in Figure 4.3

dept_no -> div_no	in Department from relationship "contains"
emp_id -> dept_no	in Employee from relationship "has"
div_no -> emp_id	in Division from relationship "is-headed-by"
dept_no -> emp_id	from binary relationship "is-managed-by"
emp_id -> desktop_no	from binary relationship "has-allocated"
desktop_no -> emp_no	from binary relationship "has-allocated"
emp_id -> spouse_id	from binary recursive relationship "is-married-to"
spouse_id -> emp_id	from binary recursive relationship "is-married-to"
emp_id, loc_name -> project_name	from ternary relationship "assigned-to"

Table 6.3 Secondary FDs Derived from the Requirements Specification

div_no -> div_name, div_addr	from entity Division
dept_no -> dept_name, dept_addr, mgr_id	from entity Department
emp_id -> emp_name, emp_addr, office_no, phone_no	from entity Employee
skill_type -> skill_descrip	from entity Skill
project_name -> start_date, end_date, head_id	from entity Project
loc_name -> loc_county, loc_state, zip	from entity Location
mgr_id -> mgr_start_date, beeper_phone_no	from entity Manager
assoc_name -> assoc_addr, phone_no, start_date	from entity Prof-assoc
desktop_no -> computer_type, serial_no	from entity Desktop

In general, we observe that candidate tables, like the ones shown in Table 6.4, are fairly good indicators of the final schema and normally require very little refinement to get to 3NF or BCNF. This observation is important—good initial conceptual design usually results in tables that are already normalized or are very close to being normalized, and thus the normalization process is usually a simple task.

Table 6.4 Candidate Tables (and FDs) from ER Diagram Transformation

division	div_no -> div_name, div_addr
	div_no -> emp_id
department	dept_no -> dept_name, dept_addr, mgr_id dept_no -> div_no dept_no -> emp_id
employee	emp_id -> emp_name, emp_addr, office_no, phone_no emp_id -> dept_no emp_id -> spouse_id spouse_id -> emp_id
manager	mgr_id -> mgr_start_date, beeper_phone_no
secretary	none
engineer	emp_id -> desktop_no

Table 6.4 Candidate Tables (and FDs) from ER Diagram Transformation *(continued)*

technician	none
skill	skill_type -> skill_descrip
project	project_name -> start_date, end_date, head_id
location	loc_name -> loc_county, loc_state, zip
prof_assoc	assoc_name -> assoc_addr, phone_no, start_date
desktop	desktop_no -> computer_type, serial_no desktop_no -> emp_no
assigned_to	emp_id, loc_name -> project_name
skill_used	none

6.4 Determining the Minimum Set of 3NF Tables

A minimum set of 3NF tables can be obtained from a given set of FDs by using the well-known synthesis algorithm developed by Bernstein [1976]. This process is particularly useful when you are confronted with a list of hundreds or thousands of FDs that describe the semantics of a database. In practice, the ER modeling process automatically decomposes this problem into smaller subproblems: the attributes and FDs of interest are restricted to those attributes within an entity (and its equivalent table) and any foreign keys that might be imposed upon that table. Thus, the database designer will rarely have to deal with more than ten or twenty attributes at a time, and in fact most entities are initially defined in 3NF already. For those tables that are not yet in 3NF, only minor adjustments will be needed in most cases.

In the following text we briefly describe the synthesis algorithm for those situations where the ER model is not useful for the decomposition. In order to apply the algorithm, we make use of the well-known Armstrong axioms, which define the basic relationships among FDs.

Inference rules (Armstrong axioms)

Reflexivity	If Y is a subset of the attributes of X, then X -> Y (i.e., if X is ABCD and Y is ABC, then X -> Y. Trivially, X -> X)
Augmentation	If X -> Y and Z is a subset of table R (i.e., Z is any attribute in R), then XZ -> YZ.

Transitivity	If X->Y and Y->Z, then X->Z.
Pseudotransitivity	If X->Y and YW->Z, then XW->Z. (Transitivity is a special case of pseudotransitivity when W = null.)
Union	If X->Y and X->Z, then X->YZ (or equivalently, X->Y,Z).
Decomposition	If X->YZ, then X->Y and X->Z.

These axioms can be used to derive two practical rules of thumb for deriving superkeys of tables where at least one superkey is already known.

Superkey Rule 1. Any FD involving all attributes of a table defines a superkey as the left side of the FD.

Given: Any FD containing all attributes in the table **R**(W,X,Y,Z), i.e., XY -> WZ.

Proof:

1. XY -> WZ as given.
2. XY -> XY by applying the reflexivity axiom.
3. XY -> XYWZ by applying the union axiom.
4. XY uniquely determines every attribute in table **R**, as shown in 3.
5. XY uniquely defines table **R**, by the definition of a table as having no duplicate rows.
6. XY is therefore a superkey, by definition.

Superkey Rule 2. Any attribute that functionally determines a superkey of a table is also a superkey for that table.

Given: Attribute A is a superkey for table **R**(A,B,C,D,E), and E -> A.

Proof:

1. Attribute A uniquely defines each row in table **R**, by the definition of a superkey.
2. A -> ABCDE by applying the definition of a superkey and a relational table.
3. E -> A as given.
4. E -> ABCDE by applying the transitivity axiom.
5. E is a superkey for table **R**, by definition.

Before we can describe the synthesis algorithm, we must define some important concepts. Let H be a set of FDs that represents at least part of the known semantics of a database. The closure of H, specified by H+, is the set of all FDs derivable from H using the Armstrong axioms or inference rules. For example, we can apply the transitivity rule to the following FDs in set H:

A -> B, B -> C, A -> C, and C -> D

to derive the FDs A -> D and B -> D. All six FDs constitute the closure H+. A cover of H, called H', is any set of FDs from which H+ can be derived. Possible covers for this example are:

1. A->B, B->C, C->D, A->C, A->D, B->D (trivial case where H' and H+ are equal)
2. A->B, B->C, C->D, A->C, A->D
3. A->B, B->C, C->D, A->C (this is the original set H)
4. A->B, B->C, C->D

A nonredundant cover of H is a cover of H that contains no proper subset of FDs, which is also a cover. The synthesis algorithm requires nonredundant covers.

3NF Synthesis Algorithm

Given a set of FDs, H, we determine a minimum set of tables in 3NF.

H:	AB->C	DM->NP
	A->DEFG	D->M
	E->G	L->D
	F->DJ	PQR->ST
	G->DI	PR->S
	D->KL	

From this point the process of arriving at the minimum set of 3NF tables consists of five steps:

1. Eliminate extraneous attributes in the left sides of the FDs
2. Search for a nonredundant cover, G of H

3. Partition G into groups so that all FDs with the same left side are in one group
4. Merge equivalent keys
5. Define the minimum set of normalized tables

Now we discuss each step in turn, in terms of the preceding set of FDs, H.

Step 1. Elimination of Extraneous Attributes

The first task is to get rid of extraneous attributes in the left sides of the FDs.

The following two relationships (rules) among attributes on the left side of an FD provide the means to reduce the left side to fewer attributes.

Reduction Rule 1. XY->Z and X->Z => Y is extraneous on the left side (applying the reflexivity and transitivity axioms).

Reduction Rule 2. XY->Z and X->Y => Y is extraneous; therefore X->Z (applying the pseudotransitivity axiom).

Applying these **reduction rules** to the set of FDs in H, we get:

DM->NP and D->M => D->NP

PQR->ST and PR->S => PQR->T

Step 2. Search for a Nonredundant Cover

We must eliminate any FD derivable from others in H using the inference rules.

Transitive FDs to be eliminated:

A->E and E->G => eliminate A->G

A->F and F->D => eliminate A->D

Step 3. Partitioning of the Nonredundant Cover

To partition the nonredundant cover into groups so that all FDs with the same left side are in one group, we must separate the nonfully functional dependencies and transitive dependencies into separate tables. At

this point we have a feasible solution for 3NF tables, but it is not necessarily the minimum set.

These nonfully functional dependencies must be put into separate tables:

AB->C
A->EF

Groups with the same left side:

G1:	AB->C	G6:	D->KLMNP
G2:	A->EF	G7:	L->D
G3:	E->G	G8:	PQR->T
G4:	G->DI	G9:	PR->S
G5:	F->DJ		

Step 4. Merge of Equivalent Keys (Merge of Tables)

In this step we merge groups with left sides that are equivalent (for example, X->Y and Y->X imply that X and Y are equivalent). This step produces a minimum set.

1. Write out the closure of all left side attributes resulting from Step 3, based on transitivities.
2. Using the closures, find tables that are subsets of other groups and try to merge them. Use Superkey Rule 1 and Superkey Rule 2 to establish whether the merge will result in FDs with superkeys on the left side. If not, try using the axioms to modify the FDs to fit the definition of superkeys.
3. After the subsets are exhausted, look for any overlaps among tables and apply Superkey Rules 1 and 2 (and the axioms) again.

In this example, note that G7 (L->D) has a subset of the attributes of G6 (D->KLMNP). Therefore, we merge to a single table, R6, with FDs D->KLMNP and L->D, because it satisfies 3NF: D is a superkey by Superkey Rule 1, and L is a superkey by Superkey Rule 2.

Step 5. Definition of the Minimum Set of Normalized Tables

The minimum set of normalized tables has now been computed. We define them below in terms of the table name, the attributes in the table, the FDs in the table, and the candidate keys for that table:

R1: ABC (AB->C with key AB)	R5: DFJ (F->DJ with key F)
R2: AEF (A->EF with key A)	R6: DKLMNP (D->KLMNP, L->D, with keys D, L)
R3: EG (E->G with key E)	R7: PQRT (PQR->T with key PQR)
R4: DGI (G->DI with key G)	R8: PRS (PR->S with key PR)

Note that this result is not only 3NF, but also BCNF, which is very frequently the case. This fact suggests a practical algorithm for a (near) minimum set of BCNF tables: Use Bernstein's algorithm to attain a minimum set of 3NF tables, then inspect each table for further decomposition (or partial replication, as shown in Section 6.1.5) to BCNF.

6.5 Fourth and Fifth Normal Forms

Normal forms up to BCNF were defined solely on FDs, and, for most database practitioners, either 3NF or BCNF is a sufficient level of normalization. However, there are in fact two more normal forms that are needed to eliminate the rest of the currently known anomalies. In this section, we will look at different types of constraints on tables: multivalued dependencies and join dependencies. If these constraints do not exist in a table, which is the most common situation, then any table in BCNF is automatically in fourth normal form (4NF), and fifth normal form (5NF) as well. However, when these constraints do exist, there may be further update (especially delete) anomalies that need to be corrected. First, we must define the concept of multivalued dependency.

6.5.1 Multivalued Dependencies

Definition. In a *multivalued dependency (MVD)*, X->>Y holds on table **R** with table scheme RS if, whenever a valid instance of table **R**(X,Y,Z) contains a pair of rows that contain duplicate values of X,

then the instance also contains the pair of rows obtained by interchanging the Y values in the original pair. This includes situations where only pairs of rows exist. Note that X and Y may contain either single or composite attributes.

An MVD X ->> Y is trivial if Y is a subset of X, or if X union Y = RS. Finally, an FD implies an MVD, which implies that a single row with a given value of X is also an MVD, albeit a trivial form.

The following examples show where an MVD does and does not exist in a table. In **R1**, the first four rows satisfy all conditions for the MVDs X->>Y and X->>Z. Note that MVDs appear in pairs because of the cross-product type of relationship between Y and Z=RS-Y as the two right sides of the two MVDs. The fifth and sixth rows of **R1** (when the X value is 2) satisfy the row interchange conditions in the above definition. In both rows, the Y value is 2, so the interchanging of Y values is trivial. The seventh row (3,3,3) satisfies the definition trivially.

In table **R2**, however, the Y values in the fifth and sixth rows are different (1 and 2), and interchanging the 1 and 2 values for Y results in a row (2,2,2) that does not appear in the table. Thus, in **R2** there is no MVD between X and Y or between X and Z, even though the first four rows satisfy the MVD definition. Note that for the MVD to exist, all rows must satisfy the criterion for an MVD.

Table **R3** contains the first three rows that do not satisfy the criterion for an MVD, since changing Y from 1 to 2 in the second row results in a row that does not appear in the table. Similarly, changing Z from 1 to 2 in the third row results in a nonappearing row. Thus, **R3** does not have any MVDs between X and Y or between X and Z.

R1:

X	Y	Z
1	1	1
1	1	2
1	2	1
1	2	2
2	2	1
2	2	2
3	3	3

R2:

X	Y	Z
1	1	1
1	1	2
1	2	1
1	2	2
2	2	1
2	1	2

R3:

X	Y	Z
1	1	1
1	1	2
1	2	1
2	2	1
2	2	2

By the same argument, in table **R1** we have the MVDs Y->> X and Y->>Z, but none with Z on the left side. Tables **R2** and **R3** have no MVDs at all.

The following inference rules for MVDs are somewhat analogous to the inference rules for functional dependencies given in Section 6.4 [Beeri, Fagin, and Howard, 1977]. They are quite useful in the analysis and decomposition of tables into 4NF.

Multivalued Dependency Inference Rules

Reflexivity	X -->> X
Augmentation	If X -->> Y, then XZ -->> Y.
Transitivity	If X -->>Y and Y -->> Z, then X -->> (Z-Y).
Pseudotransitivity	If X -->> Y and YW -->> Z, then XW -->> (Z-YW). (Transitivity is a special case of pseudotransitivity when W is null.)
Union	If X -->> Y and X -->> Z, then X -->> YZ.
Decomposition	If X -->> Y and X -->> Z, then X -->> Y intersect Z and X -->> (Z-Y).
Complement	If X -->> Y and Z=R-X-Y, then X -->> Z.
FD Implies MVD	If X -> Y, then X -->> Y.
FD, MVD Mix	If X -->> Z and Y -->> Z′ (where Z′ is contained in Z, and Y and Z are disjoint), then X->Z′.

6.5.2 Fourth Normal Form

The goal of 4NF is to eliminate nontrivial MVDs from a table by projecting them onto separate smaller tables, and thus to eliminate the update anomalies associated with the MVDs. This type of normal form is reasonably easy to attain if you know where the MVDs are. In general, MVDs must be defined from the semantics of the database; they cannot be determined from just looking at the data. The current set of data can only verify whether your assumption about an MVD is currently true or not, but this may change each time the data is updated.

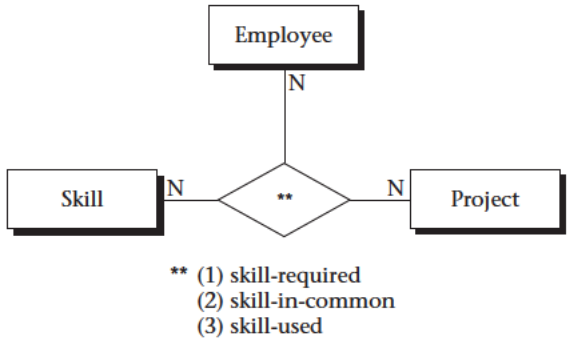


Figure 6.6 Ternary relationship with multiple interpretations

Definition. A table **R** is in *fourth normal form (4NF)* if and only if it is in BCNF and, whenever there exists an MVD in **R** (say X ->> Y), at least one of the following holds: the MVD is trivial, or X is a superkey for **R**.

Applying this definition to the three tables in the example in the previous section, we see that **R1** is not in 4NF because at least one nontrivial MVD exists and no single column is a superkey. In tables **R2** and **R3**, however, there are no MVDs. Thu,s these two tables are at least 4NF.

As an example of the transformation of a table that is not in 4NF to two tables that are in 4NF, we observe the ternary relationship skill-required, shown in Figure 6.6. The relationship skill-required is defined as follows: "An employee must have all the required skills needed for a project to work on that project." For example, in Table 6.5 the project with proj_no = 3 requires skill types A and B by all employees (see employees 101 and 102). The table **skill_required** has no FDs, but it does have several nontrivial MVDs, and is therefore only in BCNF. In such a case it can have a lossless decomposition into two many-to-many binary relationships between the entities Employee and Project, and Project and Skill. Each of these two new relationships represents a table in 4NF. It can also have a lossless decomposition resulting in a binary many-to-many relationship between the entities Employee and Skill, and Project and Skill.

A two-way lossless decomposition occurs when **skill_required** is projected over (emp_id, proj_no) to form skill_req1 and projected over (proj_no, skill_type) to form skill_req3. Projection over (emp_id,

Table 6.5 The Table **skill_required** and Its Three Projections

skill_required	emp_id	proj_no	skill_type	*MVDs(nontrivial)*
	101	3	A	proj_no ->> skill_type
	101	3	B	proj_no ->> emp_id
	101	4	A	
	101	4	C	
	102	3	A	
	102	3	B	
	103	5	D	

skill_req1		skill_req2		skill_req3	
emp_id	proj_no	emp_id	skill_type	proj_no	skill_type
101	3	101	A	3	A
101	4	101	B	3	B
102	3	101	C	4	A
103	5	102	A	4	C
		102	B	5	D
		103	D		

proj_no) to form skill_req1 and over (emp_id, skill_type) to form skill_req2, however, is not lossless. A three-way lossless decomposition occurs when **skill_required** is projected over (emp_id, proj_no), (emp_id, skill_type), and (proj_no, skill_type).

Tables in 4NF avoid certain update anomalies (or inefficiences). For instance, a delete anomaly exists when two independent facts get tied together unnaturally so that there may be bad side effects of certain deletes. For example, in **skill_required**, the last row of a skill_type may be lost if an employee is temporarily not working on any projects. An update inefficiency may occur when adding a new project in **skill_required**, which requires insertions for many rows to include all the required skills for that new project. Likewise, loss of a project requires many deletions. These inefficiencies are avoided when

skill_required is decomposed into **skill_req1** and **skill_req3**. In general (but not always), decomposition of a table into 4NF tables results in less data redundancy.

6.5.3 Decomposing Tables to 4NF

Algorithms to decompose tables into 4NF are difficult to develop. Let's look at some straightforward approaches to 4NF from BCNF and lower normal forms. First, if a table is BCNF, it either has no FDs, or each FD is characterized by its left side being a superkey. Thus, if the only MVDs in this table are derived from its FDs, they have only superkeys as their left sides, and the table is 4NF by definition. If, however, there are other nontrivial MVDs whose left sides are not superkeys, the table is only in BCNF and must be decomposed to achieve higher normalization.

The basic decomposition process from a BCNF table is defined by selecting the most important MVD (or if that is not possible, then by selecting one arbitrarily), defining its complement MVD, and decompose the table into two tables containing the attributes on the left and right sides of that MVD and its complement. This type of decomposition is lossless because each new table is based on the same attribute, which is the left side of both MVDs. The same MVDs in these new tables are now trivial because they contain every attribute in the table. However, other MVDs may be still present, and more decompositions by MVDs and their complements may be necessary. This process of arbitrary selection of MVDs for decomposition is continued until only trivial MVDs exist, leaving the final tables in 4NF.

As an example, let **R**(A,B,C,D,E,F) with no FDs, and with MVDs A ->> B and CD ->> EF. The first decomposition of **R** is into two tables **R1**(A,B) and **R2**(A,C,D,E,F) by applying the MVD A ->> B and its complement A ->> CDEF. Table **R1** is now 4NF, because A ->> B is trivial and is the only MVD in the table. Table **R2**, however, is still only BCNF, because of the nontrivial MVD CD ->> EF. We then decompose **R2** into **R21**(C,D,E,F) and **R22**(C,D,A) by applying the MVD CD ->> EF and its complement CD ->> A. Both **R21** and **R22** are now 4NF. If we had applied the MVD complement rule in the opposite order, using CD ->> EF and its complement CD ->> AB first, the same three 4NF tables would result from this method. However, this does not occur in all cases; it only occurs in those tables where the MVDs have no intersecting attributes.

This method, in general, has the unfortunate side effect of potentially losing some or all of the FDs and MVDs. Therefore, any decision to

transform tables from BCNF to 4NF must take into account the trade-off between normalization and the elimination of delete anomalies, and the preservation of FDs and possibly MVDs. It should also be noted that this approach derives a feasible, but not necessarily a minimum, set of 4NF tables.

A second approach to decomposing BCNF tables is to ignore the MVDs completely and split each BCNF table into a set of smaller tables, with the candidate key of each BCNF table being the candidate key of a new table and the nonkey attributes distributed among the new tables in some semantically meaningful way. This form of decomposing by candidate key (that is, superkey) is lossless because the candidate keys uniquely join; it usually results in the simplest form of 5NF tables, those with a candidate key and one nonkey attribute, and no MVDs. However, if one or more MVDs still exist, further decomposition must be done with the MVD/MVD-complement approach given above. The decomposition by candidate keys preserves FDs, but the MVD/MVD-complement approach does not preserve either FDs or MVDs.

Tables that are not yet in BCNF can also be directly decomposed into 4NF using the MVD/MVD-complement approach. Such tables can often be decomposed into smaller minimum sets than those derived from transforming into BCNF first and then 4NF, but with a greater cost of lost FDs. In most database design situations, it is preferable to develop BCNF tables first, then evaluate the need to normalize further while preserving the FDs.

6.5.4 Fifth Normal Form

> ***Definition.*** A table **R** is in *fifth normal form (5NF)* or project-join normal form (PJ/NF) if and only if every join dependency in **R** is implied by the keys of **R**.

As we recall, a lossless decomposition of a table implies that it can be decomposed by two or more projections, followed by a natural join of those projections (in any order) that results in the original table, without any spurious or missing rows. The general lossless decomposition constraint, involving any number of projections, is also known as a *join dependency* (JD). A join dependency is illustrated by the following example: in a table **R** with n arbitrary subsets of the set of attributes of **R**, **R** satisfies a join dependency over these n subsets if and only if **R** is equal to the natural join of its projections on them. A JD is trivial if one of the subsets is **R** itself.

5NF or PJ/NF requires satisfaction of the membership algorithm [Fagin, 1979], which determines whether a JD is a member of the set of logical consequences of (can be derived from) the set of key dependencies known for this table. In effect, for any 5NF table, every dependency (FD, MVD, JD) is determined by the keys. As a practical matter we note that because JDs are very difficult to determine in large databases with many attributes, 5NF tables are not easily derivable, and logical database design typically produces BCNF tables.

We should also note that by the preceding definitions, just because a table is decomposable does not necessarily mean it is not 5NF. For example, consider a simple table with four attributes (A,B,C,D), one FD (A->BCD), and no MVDs or JDs not implied by this FD. It could be decom-

Table 6.6 The Table **skill_in_common** and Its Three Projections

skill_in_common	emp_id	proj_no	skill_type
	101	3	A
	101	3	B
	101	4	A
	101	4	B
	102	3	A
	102	3	B
	103	3	A
	103	4	A
	103	5	A
	103	5	C

skill_in_com1		**skill_in_com2**		**skill_in_com3**	
emp_id	proj_no	emp_id	skill_type	proj_no	skill_type
101	3	101	A	3	A
101	4	101	B	3	B
102	3	102	A	4	A
103	3	102	B	4	B
103	4	103	A	5	A
103	5	103	C	5	C

posed into three tables, A->B, A->C, and A->D, all based on the same superkey A; however, it is already in 5NF without the decomposition. Thus, the decomposition is not required for normalization. On the other hand, decomposition can be a useful tool in some instances for performance improvement.

The following example demonstrates that a table representing a ternary relationship may not have any two-way lossless decompositions; however, it may have a three-way lossless decomposition, which is equivalent to three binary relationships, based on the three possible projections of this table. This situation occurs in the relationship skill-in-common (Figure 6.6), which is defined as "The employee must apply the intersection of his or her available skills with the skills needed to work on certain projects." In this example, skill-in-common is less restrictive than skill-required because it allows an employee to work on a project even if he or she does not have all the skills required for that project.

As Table 6.6 shows, the three projections of **skill_in_common** result in a three-way lossless decomposition. There are no two-way lossless decompositions and no MVDs; thus, the table **skill_in_common** is in 4NF.

The ternary relationship in Figure 6.6 can be interpreted yet another way. The meaning of the relationship skill-used is "We can selectively record different skills that each employee applies to working on individual projects." It is equivalent to a table in 5NF that cannot be decomposed into either two or three binary tables. Note by studying Table 6.7 that the associated table, **skill_used**, has no MVDs or JDs.

Table 6.7 The Table **skill_used**, Its Three Projections, and Natural Joins of Its Projections

skill_used	emp_id	proj_no	skill_type
	101	3	A
	101	3	B
	101	4	A
	101	4	C
	102	3	A
	102	3	B
	102	4	A
	102	4	B

Table 6.7 The Table **skill_used**, Its Three Projections, and Natural Joins of Its Projections *(continued)*

Three projections on **skill_used** result in:

skill_used1

emp_id	proj_no
101	3
101	4
102	3
102	4

skill_used2

proj_no	skill_type
3	A
3	B
4	A
4	B
4	C

skill_used3

emp_id	skill_type
101	A
101	B
101	C
102	A
102	B

join **skill_used1** with **skill_used2** to form:

skill_used_12

emp_id	proj_no	skill_type
101	3	A
101	3	B
101	4	A
101	4	B
101	4	C
102	3	A
102	3	B
102	4	A
102	4	B
102	4	C

join **skill_used12** with **skill_used3** to form:

skill_used_123

emp_id	proj_no	skill_type
101	3	A
101	3	B
101	4	A
101	4	B (spurious)
101	4	C
102	3	A
102	3	B
102	4	A
102	4	B

A table may have constraints that are FDs, MVDs, and JDs. An MVD is a special case of a JD. To determine the level of normalization of the table, analyze the FDs first to determine normalization through BCNF; then analyze the MVDs to determine which BCNF tables are also 4NF; then, finally, analyze the JDs to determine which 4NF tables are also 5NF.

Table 6.8 Summary of Higher Normal Forms

Table Name	*Normal Form*	*Two-way Lossless decomp/join?*	*Three-way Lossless decomp/join?*	*Nontrivial MVDs*
skill_required	BCNF	yes	yes	2
skill_in_common	4NF	no	yes	0
skill_used	5NF	no	no	0

A many-to-many-to-many ternary relationship is:

- BCNF if it can be replaced by two binary relationships
- 4NF if it can only be replaced by three binary relationships
- 5NF if it cannot be replaced in any way (and thus is a true ternary relationship)

We observe the equivalence between certain ternary relationships and the higher normal form tables transformed from those relationships. Ternary relationships that have at least one "one" entity cannot be decomposed (or broken down) into binary relationships, because that would destroy the one or more FDs required in the definition, as shown previously. A ternary relationship with all "many" entities, however, has no FDs, but in some cases may have MVDs, and thus have a lossless decomposition into equivalent binary relationships.

In summary, the three common cases that illustrate the correspondence between a lossless decomposition in a many-to-many-to-many ternary relationship table and higher normal forms in the relational model are shown in Table 6.8.

6.6 Summary

In this chapter, we defined the constraints imposed on tables—most commonly the functional dependencies or FDs. Based on these constraints, practical normal forms for database tables are defined: 1NF, 2NF, 3NF, and BCNF. All are based on the types of FDs present. In this chapter, a practical algorithm for finding the minimum set of 3NF tables is given.

The following statements summarize the functional equivalence between the ER model and normalized tables:

1. *Within an entity.* The level of normalization is totally dependent upon the interrelationships among the key and nonkey attributes. It could be any form from unnormalized to BCNF or higher.
2. *Binary (or binary recursive) one-to-one or one-to-many relationship.* Within the "child" entity, the foreign key (a replication of the primary key of the "parent") is functionally dependent upon the child's primary key. This is at least BCNF, assuming that the entity by itself, without the foreign key, is already BCNF.
3. *Binary (or binary recursive) many-to-many relationship.* The intersection table has a composite key and possibly some nonkey attributes functionally dependent upon it. This is at least BCNF.
4. *Ternary relationship.*
   a. one-to-one-to-one => three overlapping composite keys, at least BCNF
   b. one-to-one-to-many => two overlapping composite keys, at least BCNF
   c. one-to-many-to-many => one composite key, at least BCNF
   d. many-to-many-to-many => one composite key with three attributes, at least BCNF; in some cases it can be also 4NF, or even 5NF

In summary, we observed that a good, methodical conceptual design procedure often results in database tables that are either normalized (BCNF) already, or can be normalized with very minor changes.

6.7 Literature Summary

Good summaries of normal forms can be found in Date [1999], Kent [1983], Dutka and Hanson [1989], and Smith [1985]. Algorithms for normal form decomposition and synthesis techniques are given in Bernstein [1976], Fagin [1977], and Maier [1983]. The earliest work in normal forms was done by Codd [1970, 1974].

7 An Example of Logical Database Design

The following example illustrates how to proceed through the requirements analysis and logical design steps of the database life cycle, in a practical way, for a relational database.

7.1 Requirements Specification

The management of a large retail store would like a database to keep track of sales activities. The requirements analysis for this database led to the six entities and their unique identifiers shown in Table 7.1.

The following assertions describe the data relationships:

- Each customer has one job title, but different customers may have the same job title.
- Each customer may place many orders, but only one customer may place a particular order.
- Each department has many salespeople, but each salesperson must work in only one department.
- Each department has many items for sale, but each item is sold in only one department. ("Item" means item type, like IBM PC).

- For each order, items ordered in different departments must involve different salespeople, but all items ordered within one department must be handled by exactly one salesperson. In other words, for each order, each item has exactly one salesperson; and for each order, each department has exactly one salesperson.

For physical design (e.g., access methods, etc.) it is necessary to determine what kind of processing needs to be done on the data; that is, what are the queries and updates needed to satisfy the user requirements, and what are their frequencies? In addition, the requirements analysis should determine whether there will be substantial database growth (i.e., volumetrics); in what time frame that growth will take place; and whether the frequency and type of queries and updates will change, as well. Decay as well as growth should be estimated, as each will have significant effect on the later stages of database design.

7.1.1 Design Problems

1. Using the information given and, in particular, the five assertions, derive a conceptual data model and a set of functional dependencies (FDs) that represent all the known data relationships.
2. Transform the conceptual data model into a set of candidate SQL tables. List the tables, their primary keys, and other attributes.
3. Find the minimum set of normalized (BCNF) tables that are functionally equivalent to the candidate tables.

Table 7.1 Requirements Analysis Results

Entity	*Entity key* in characters	*Key length(max)*	*Number of* occurrences
Customer	cust-no	6	80,000
Job	job-no	24	80
Order	order-no	9	200,000
Salesperson	sales-id	20	150
Department	dept-no	2	10
Item	item-no	6	5,000

7.2 Logical Design

Our first step is to develop a conceptual data model diagram and a set of FDs to correspond to each of the assertions given. Figure 7.1 presents the diagram for the ER model, and Figure 7.2 shows the equivalent diagram for UML. Normally, the conceptual data model is developed without knowing all the FDs, but in this example the nonkey attributes are omitted so that the entire database can be represented with only a few statements and FDs. The result of this analysis, relative to each of the assertions given, follows in Table 7.2.

The candidate tables required to represent the semantics of this problem can be derived easily from the constructs for entities and relationships. Primary keys and foreign keys are explicitly defined.

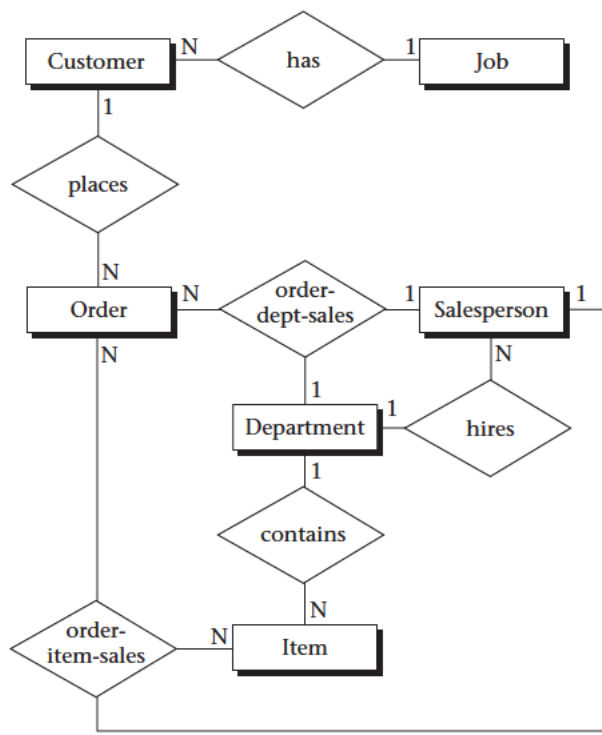


Figure 7.1 Conceptual data model diagram for the ER model

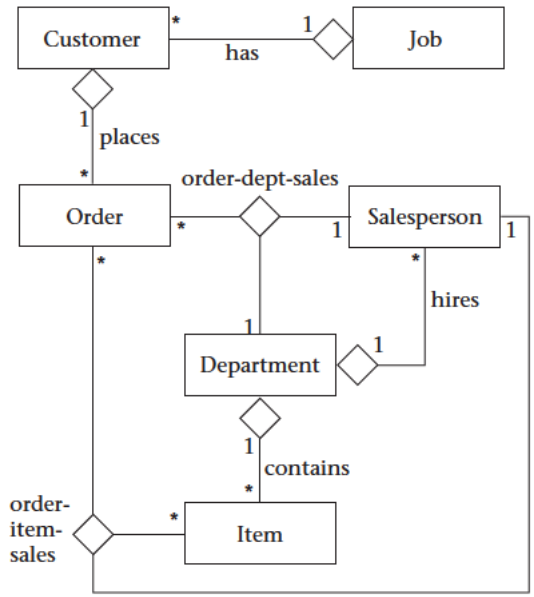


Figure 7.2 Conceptual data model diagram for UML

Table 7.2 Results of the Analysis of the Conceptual Data Model

ER Construct	*FDs*
Customer(many): Job(one)	cust-no -> job-title
Order(many): Customer(one)	order-no -> cust-no
Salesperson(many): Department(one)	sales-id -> dept-no
Item(many): Department(one)	item-no -> dept-no
Order(many): Item(many): Salesperson(one)	order-no,item-no->sales-id
Order(many): Department(many): Salesperson(one)	order-no,dept-no-> sales-id

```
create table customer
   (cust_no        char(6),
   job_title       varchar(256),
   primary key (cust_no),
   foreign key (job_title) references job
       on delete set null on update cascade);

create table job
   (job_no         char(6),
   job_title       varchar(256),
   primary key (job_no));

create table order
   (order_no       char(9),
   cust_no         char(6) not null,
   primary key (order_no),
   foreign key (cust_no) references customer
       on delete set null on update cascade);

create table salesperson
   (sales_id       char(10)
   sales_name      varchar(256),
   dept_no         char(2),
   primary key (sales_id),
   foreign key (dept_no) references department
       on delete set null on update cascade);

create table department
   (dept_no        char(2),
   dept_name       varchar(256),
   manager_name    varchar(256),
   primary key (dept_no));

create table item
   (item_no        char(6),
   dept_no         char(2),
   primary key (item_no),
   foreign key (dept_no) references department
       on delete set null on update cascade);
```

```
create table order_item_sales
   (order_no        char(9),
   item_no          char(6),
   sales_id         varchar(256) not null,
   primary key (order_no, item_no),
   foreign key (order_no) references order
       on delete cascade on update cascade,
   foreign key (item_no) references item
       on delete cascade on update cascade,
   foreign key (sales_id) references salesperson
       on delete cascade on update cascade);

create table order_dept_sales
   (order_no        char(9),
   dept_no          char(2),
   sales_id         varchar(256) not null,
   primary key (order_no, dept_no),
   foreign key (order_no) references order
       on delete cascade on update cascade,
   foreign key (dept_no) references department
       on delete cascade on update cascade,
   foreign key (sales_id) references salesperson
       on delete cascade on update cascade);
```

Note that it is often better to put foreign key definitions in separate (alter) statements. This prevents the possibility of getting circular definitions with very large schemas.

This process of decomposition and reduction of tables moves us closer to a minimum set of normalized (BCNF) tables, as shown in Table 7.3.

The reductions shown in this section have decreased storage space and update cost and have maintained the normalization of BCNF (and thus 3NF). On the other hand, we have potentially higher retrieval cost—given the transaction "list all job_titles," for example—and have increased the potential for loss of integrity because we have eliminated simple tables with only key attributes. Resolution of these trade-offs depends on your priorities for your database.

The details of indexing will be covered in the companion book, *Physical Database Design*. However, during the logical design phase of defining SQL tables, it makes sense to start considering where to create indexes. At a minimum, all primary keys and all foreign keys should be

Table 7.3 Decomposition and Reduction of Tables

Table	*Primary key*	*Likely non-keys*
customer	cust_no	job_title, cust_name, cust_address
order	order_no	cust_no, item_no, date_of_purchase, price
salesperson	sales_id	dept_no, sales_name, phone_no
item	item_no	dept_no, color, model_no
order_item_sales	order_no,item_no	sales_id
order_dept_sales	order_no,dept_no	sales_id

indexed. Indexes are relatively easy to implement and store, and make a significant difference in reducing the access time to stored data.

7.3 Summary

In this chapter, we developed a global conceptual schema and a set of SQL tables for a relational database, given the requirements specification for a retail store database. The example illustrates the database life cycle steps of conceptual data modeling, global schema design, transformation to SQL tables, and normalization of those tables. It summarizes the techniques presented in Chapters 1 through 6.

8 Business Intelligence

Business intelligence has become a buzzword in recent years. The database tools found under the heading of *business intelligence* include data warehousing, online analytical processing (OLAP), and data mining. The functionalities of these tools are complementary and interrelated. *Data warehousing* provides for the efficient storage, maintenance, and retrieval of historical data. *OLAP* is a service that provides quick answers to *ad hoc* queries against the data warehouse. *Data mining* algorithms find patterns in the data and report models back to the user. All three tools are related to the way data in a data warehouse are logically organized, and performance is highly sensitive to the database design techniques used [Barquin and Edelstein, 1997]. The encompassing goal for business intelligence technologies is to provide useful information for decision support.

Each of the major DBMS vendors is marketing the tools for data warehousing, OLAP, and data mining as business intelligence. This chapter covers each of these technologies in turn. We take a close look at the requirements for a data warehouse; its basic components and principles of operation; the critical issues in its design; and the important logical database design elements in its environment. We then investigate the basic elements of OLAP and data mining as special query techniques applied to data warehousing. We cover data warehousing in Section 8.1, OLAP in Section 8.2, and data mining in Section 8.3.

8.1 Data Warehousing

A data warehouse is a large repository of historical data that can be integrated for decision support. The use of a data warehouse is markedly different from the use of operational systems. Operational systems contain the data required for the day-to-day operations of an organization. This operational data tends to change quickly and constantly. The table sizes in operational systems are kept manageably small by periodically purging old data. The data warehouse, by contrast, periodically receives historical data in batches, and grows over time. The vast size of data warehouses can run to hundreds of gigabytes, or even terabytes. The problem that drives data warehouse design is the need for quick results to queries posed against huge amounts of data. The contrasting aspects of data warehouses and operational systems result in a distinctive design approach for data warehousing.

8.1.1 Overview of Data Warehousing

A data warehouse contains a collection of tools for decision support associated with very large historical databases, which enables the end user to make quick and sound decisions. Data warehousing grew out of the technology for decision support systems (DSS) and executive information systems (EIS). DSSs are used to analyze data from commonly available databases with multiple sources, and to create reports. The report data is not time critical in the sense that a real-time system is, but it must be timely for decision making. EISs are like DSSs, but more powerful, easier to use, and more business specific. EISs were designed to provide an alternative to the classical online transaction processing (OLTP) systems common to most commercially available database systems. OLTP systems are often used to create common applications, including those with mission-critical deadlines or response times. Table 8.1 summarizes the basic differences between OLTP and data warehouse systems.

The basic architecture for a data warehouse environment is shown in Figure 8.1. The diagram shows that the data warehouse is stocked by a variety of source databases from possibly different geographical locations. Each source database serves its own applications, and the data warehouse serves a DSS/EIS with its informational requests. Each feeder system database must be reconciled with the data warehouse data model; this is accomplished during the process of extracting the required data from the feeder database system, transforming the data

Table 8.1 Comparison between OLTP and Data Warehouse Databases

OLTP	*Data Warehouse*
Transaction oriented	Business process oriented
Thousands of users	Few users (typically under 100)
Generally small (MB up to several GB)	Large (from hundreds of GB to several TB)
Current data	Historical data
Normalized data (many tables, few columns per table)	Denormalized data (few tables, many columns per table)
Continuous updates	Batch updates*
Simple to complex queries	Usually very complex queries

* There is currently a push in the industry towards "active warehousing," in which the warehouse receives data in continuous updates. See Section 8.2.5 for further discussion.

from the feeder system to the data warehouse, and loading the data into the data warehouse [Cataldo, 1997].

Core Requirements for Data Warehousing

Let us now take a look at the core requirements and principles that guide the design of data warehouses (DWs) [Simon, 1995; Barquin and Edelstein, 1997; Chaudhuri and Dayal, 1997; Gray and Watson, 1998]:

1. DWs are organized around subject areas. Subject areas are analogous to the concept of functional areas, such as sales, project management, or employees, as discussed in the context of ER diagram clustering in Section 4.5. Each subject area has its own conceptual schema and can be represented using one or more entities in the ER data model or by one or more object classes in the object-oriented data model. Subject areas are typically independent of individual transactions involving data creation or manipulation. Metadata repositories are needed to describe source databases, DW objects, and ways of transforming data from the sources to the DW.
2. DWs should have some integration capability. A common data representation should be designed so that all the different individual representations can be mapped to it. This is particularly

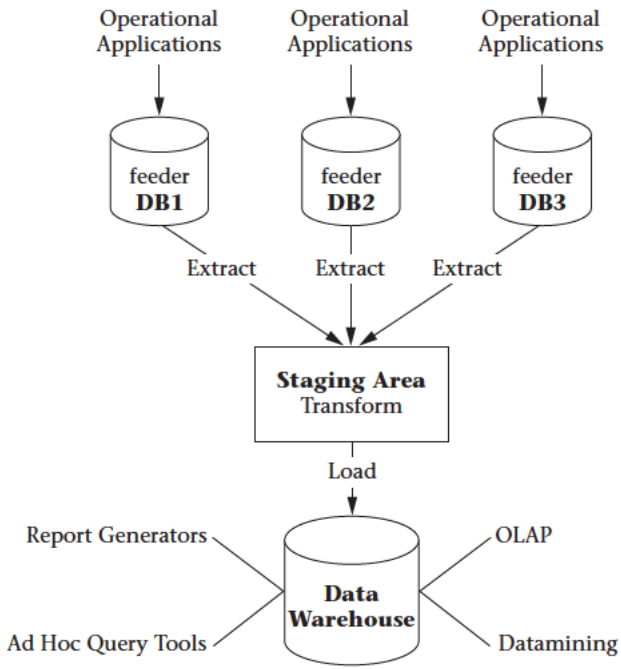


Figure 8.1 Basic data warehouse architecture

useful if the warehouse is implemented as a multidatabase or federated database.

3. The data is considered to be nonvolatile and should be mass loaded. Data extraction from current databases to the DW requires that a decision should be made whether to extract the data using standard relational database (RDB) techniques at the row or column level or specialized techniques for mass extraction. Data cleaning tools are required to maintain data quality—for example, to detect missing data, inconsistent data, homonyms, synonyms, and data with different units. Data migration, data scrubbing, and data auditing tools handle specialized problems in data cleaning and transformation. Such tools are similar to those used for conventional relational database schema (view) integration. Load utilities take cleaned data and load it into the DW, using batch processing techniques. Refresh techniques propagate updates on the source data to base data and derived data in the DW. The decision of when and how to refresh is made by the DW

administrator and depends on user needs (e.g., OLAP needs) and existing traffic to the DW.

4. Data tends to exist at multiple levels of granularity. Most important, the data tends to be of a historical nature, with potentially high time variance. In general, however, granularity can vary according to many different dimensions, not only by time frame but also by geographic region, type of product manufactured or sold, type of store, and so on. The sheer size of the databases is a major problem in the design and implementation of DWs, especially for certain queries, updates, and sequential backups. This necessitates a critical decision between using a relational database (RDB) or a multidimensional database (MDD) for the implementation of a DW.
5. The DW should be flexible enough to meet changing requirements rapidly. Data definitions (schemas) must be broad enough to anticipate the addition of new types of data. For rapidly changing data retrieval requirements, the types of data and levels of granularity actually implemented must be chosen carefully.
6. The DW should have a capability for rewriting history, that is, allowing for "what-if" analysis. The DW should allow the administrator to update historical data temporarily for the purpose of "what-if" analysis. Once the analysis is completed, the data must be correctly rolled back. This condition assumes that the data are at the proper level of granularity in the first place.
7. A usable DW user interface should be selected. The leading choices today are SQL, multidimensional views of relational data, or a special-purpose user interface. The user interface language must have tools for retrieving, formatting, and analyzing data.
8. Data should be either centralized or distributed physically. The DW should have the capability to handle distributed data over a network. This requirement will become more critical as the use of DWs grows and the sources of data expand.

The Life Cycle of Data Warehouses

Entire books have been written about select portions of the data warehouse life cycle. Our purpose in this section is to present some of the basics and give the flavor of data warehousing. We strongly encourage those who wish to pursue data warehousing to continue learning through other books dedicated to data warehousing. Kimball and Ross

[1998, 2002] have a series of excellent books covering the details of data warehousing activities.

Figure 8.2 outlines the activities of the data warehouse life cycle, based heavily on Kimball and Ross's Figure 16.1 [2002]. The life cycle begins with a dialog to determine the project plan and the business requirements. When the plan and the requirements are aligned, design and implementation can proceed. The process forks into three threads that follow independent timelines, meeting up before deployment (see Figure 8.2). Platform issues are covered in one thread, including technical architectural design, followed by product selection and installation. Data issues are covered in a second thread, including dimensional modeling and then physical design, followed by data staging design and development. The special analytical needs of the users are pursued in the third thread, including analytic application specification followed by analytic application development. These three threads join before deployment. Deployment is followed by maintenance and growth, and changes in the requirements must be detected. If adjustments are needed, the cycle repeats. If the system becomes defunct, then the life cycle terminates.

The remainder of our data warehouse section focuses on the dimensional modeling activity. More comprehensive material can be found in Kimball and Ross [1998, 2002] and Kimball and Caserta [2004].

8.1.2 Logical Design

We discuss the logical design of data warehouses in this section; the physical design issues are covered in volume two. The logical design of data warehouses is defined by the dimensional data modeling approach. We cover the schema types typically encountered in dimensional modeling, including the star schema and the snowflake schema. We outline the dimensional design process, adhering to the methodology described by Kimball and Ross [2002]. Then we walk through an example, covering some of the crucial concepts of dimensional data modeling.

Dimensional Data Modeling

The dimensional modeling approach is quite different from the normalization approach typically followed when designing a database for daily operations. The context of data warehousing compels a different approach to meeting the needs of the user. The need for dimensional modeling will be

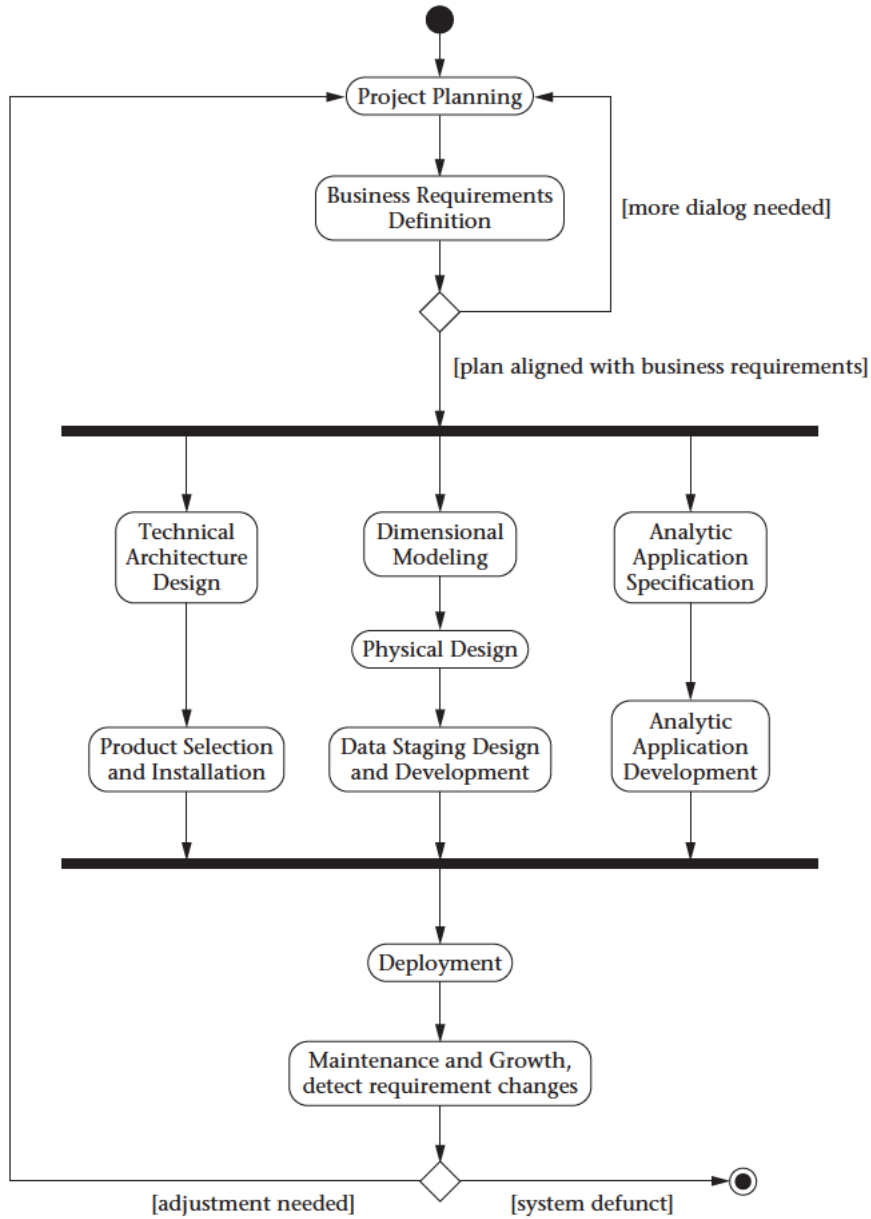


Figure 8.2 Data warehouse life cycle (based heavily on Kimball and Ross [2002], Figure 16.1)

discussed further as we proceed. If you haven't been exposed to data warehousing before, be prepared for some new paradigms.

The Star Schema

Data warehouses are commonly organized with one large central *fact table*, and many smaller *dimension tables*. This configuration is termed a *star schema*; an example is shown in Figure 8.3. The fact table is composed of two types of attributes: *dimension attributes* and *measures*. The dimension attributes in Figure 8.3 are CustID, ShipDateID, BindID, and JobId. Most dimension attributes have foreign key/primary key relationships with dimension tables. The dimension tables in Figure 8.3 are Customer, Ship Calendar, and Bind Style. Occasionally, a dimension attribute exists without a related dimension table. Kimball and Ross refer to these as *degenerate dimensions*. The JobId attribute in Figure 8.3 is a degenerate dimension (more on this shortly). We indicate the dimension attributes that act as foreign keys using the stereotype «fk». The primary keys of the dimension tables are indicated with the stereotype «pk». Any degenerate dimensions in the fact table are indicated with the stereotype «dd». The fact table also contains measures, which contain values to be aggregated when queries group rows together. The measures in Figure 8.3 are Cost and Sell.

Queries against the star schema typically use attributes in the dimension tables to select the pertinent rows from the fact table. For example, the user may want to see cost and sell for all jobs where the Ship Month

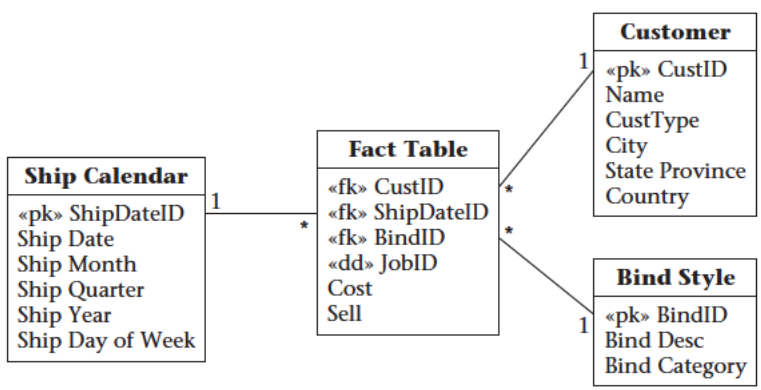


Figure 8.3 Example of a star schema for a data warehouse

is January 2005. The dimension table attributes are also typically used to group the rows in useful ways when exploring summary information. For example, the user may wish to see the total cost and sell for each Ship Month in the Ship Year 2005. Notice that dimension tables can allow different *levels* of detail the user can examine. For example, the Figure 8.3 schema allows the fact table rows to be grouped by Ship Date, Month, Quarter or Year. These dimension levels form a *hierachy*. There is also a second hierarchy in the Ship Calendar dimension that allows the user to group fact table rows by the day of the week. The user can move up or down a hierarchy when exploring the data. Moving down a hierarchy to examine more detailed data is a *drill-down* operation. Moving up a hierarchy to summarize details is a *roll-up* operation.

Together, the dimension attributes compose a candidate key of the fact table. The level of detail defined by the dimension attributes is the *granularity* of the fact table. When designing a fact table, the granularity should be the most detailed level available that any user would wish to examine. This requirement sometimes means that a degenerate dimension, such as JobId in Figure 8.3, must be included. The JobId in this star schema is not used to select or group rows, so there is no related dimension table. The purpose of the JobId attribute is to distinguish rows at the correct level of granularity. Without the JobId attribute, the fact table would group together similar jobs, prohibiting the user from examining the cost and sell values of individual jobs.

Normalization is not the guiding principle in data warehouse design. The purpose of data warehousing is to provide quick answers to queries against a large set of historical data. Star schema organization facilitates quick response to queries in the context of the data warehouse. The core detailed data are centralized in the fact table. Dimensional information and hierarchies are kept in dimension tables, a single join away from the fact table. The hierarchical levels of data contained in the dimension tables of Figure 8.3 violate 3NF, but these violations to the principles of normalization are justified. The normalization process would break each dimension table in Figure 8.3 into multiple tables. The resulting normalized schema would require more join processing for most queries. The dimension tables are small in comparison to the fact table, and typically slow changing. The bulk of operations in the data warehouse are read operations. The benefits of normalization are low when most operations are read only. The benefits of minimizing join operations overwhelm the benefits of normalization in the context of data warehousing. The marked differences between the data warehouse environment and the

operational system environment lead to distinct design approaches. Dimensional modeling is the guiding principle in data warehouse design.

Snowflake Schema

The data warehouse literature often refers to a variation of the star schema known as the *snowflake schema*. Normalizing the dimension tables in a star schema leads to a snowflake schema. Figure 8.4 shows the snowflake schema analogous to the star schema of Figure 8.3. Notice that each hierarchical level becomes its own table. The snowflake schema is generally losing favor. Kimball and Ross strongly prefer the star schema, due to its speed and simplicity. Not only does the star schema yield quicker query response, it is also easier for the user to understand when building queries. We include the snowflake schema here for completeness.

Dimensional Design Process

We adhere to the four-step dimensional design process promoted by Kimball and Ross. Figure 8.5 outlines the activities in the four-step process.

Dimensional Modeling Example

Congratulations, you are now the owner of the ACME Data Mart Company! Your company builds data warehouses. You consult with other companies, design and deploy data warehouses to meet their needs, and support them in their efforts.

Your first customer is XYZ Widget, Inc. XYZ Widget is a manufacturing company with information systems in place. These are operational systems that track the current and recent state of the various business processes. Older records that are no longer needed for operating the plant are purged. This keeps the operational systems running efficiently.

XYZ Widget is now ten years old, and growing fast. The management realizes that information is valuable. The CIO has been saving data before they are purged from the operational system. There are tens of millions of historical records, but there is no easy way to access the data in a meaningful way. ACME Data Mart has been called in to design and build a DSS to access the historical data.

Discussions with XYZ Widget commence. There are many questions they want to have answered by analyzing the historical data. You begin by making a list of what XYZ wants to know.

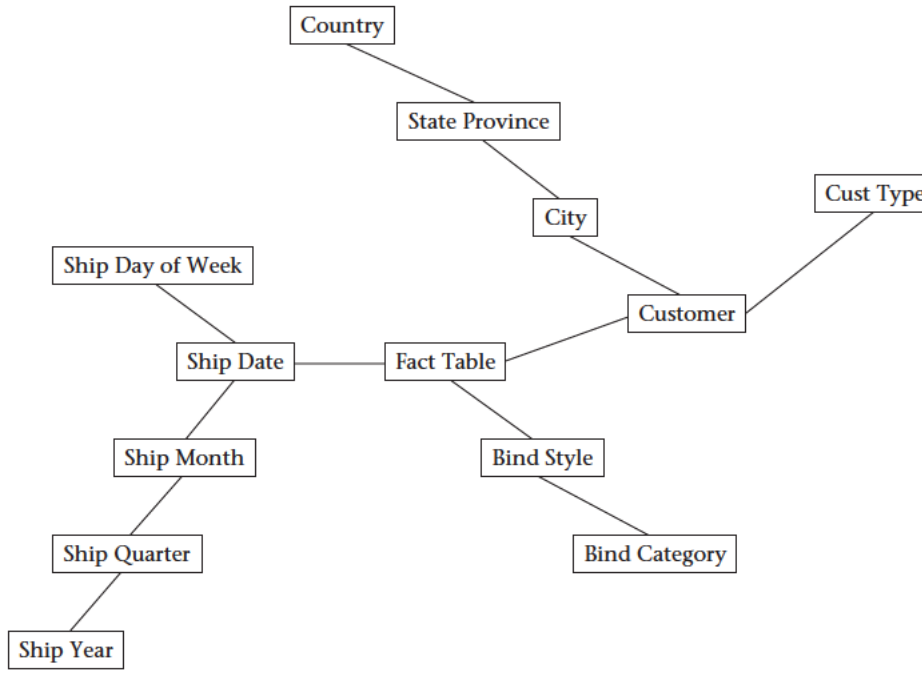


Figure 8.4 Example of a snowflake schema for a data warehouse

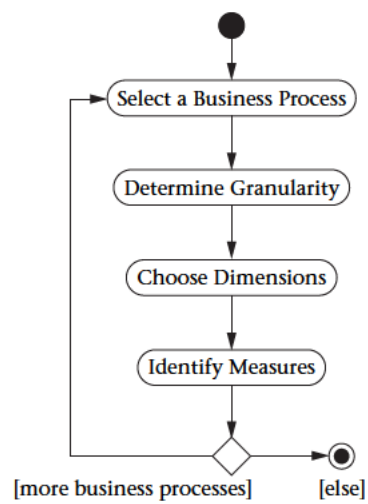


Figure 8.5 Four step dimensional design process [Kimball and Ross, 2002]

XYZ Widget Company Wish List

1. What are the trends of our various products in terms of sales dollars, unit volume, and profit margin?
2. For those products that are not profitable, can we drill down and determine why they are not profitable?
3. How accurately do our estimated costs match our actual costs?
4. When we change our estimating calculations, how are sales and profitability affected?
5. What are the trends in the percentage of jobs that ship on time?
6. What are the trends in productivity by department, for each machine, and for each employee?
7. What are the trends in meeting the scheduled dates for each department, and for each machine?
8. How effective was the upgrade on machine 123?
9. Which customers bring the most profitable jobs?
10. How do our promotional bulk discounts affect sales and profitability?

Looking over the wish list, you begin picking out the business processes involved. The following list is sufficient to satisfy the items on the wish list.

Business Processes

1. Estimating
2. Scheduling
3. Productivity Tracking
4. Job Costing

These four business processes are interlinked in the XYZ Widget Company. Let's briefly walk through the business processes and the organization of information in the operational systems, so we have an idea what information is available for analysis. For each business process, we'll design a star schema for storing the data.

The estimating process begins by entering widget specifications. The type of widget determines which machines are used to manufacture the widget. The estimating software then calculates estimated time on each

machine used to produce that particular type of widget. Each machine is modeled with a standard setup time and running speed. If a particular type of widget is difficult to process on a particular machine, the times are adjusted accordingly. Each machine has an hourly rate. The estimated time is multiplied by the rate to give labor cost. Each estimate stores widget specifications, a breakdown of the manufacturing costs, the markup and discount applied (if any), and the price. The quote is sent to the customer. If the customer accepts the quote, then the quote is associated with a job number, the specifications are printed as a job ticket, and the job ticket moves to scheduling.

We need to determine the grain before designing a schema for the estimating data mart. The grain should be at the most detailed level, giving the greatest flexibility for drill-down operations when users are exploring the data. The most granular level in the estimating process is the estimating detail. Each estimating detail record specifies information for an individual cost center for a given estimate. This is the finest granularity of estimating data in the operational system, and this level of detail is also potentially valuable for the data warehouse users.

The next design step is to determine the dimensions. Looking at the estimating detail, we see that the associated attributes are the job specifications, the estimate number and date, the job number and win date if the estimate becomes a job, the customer, the promotion, the cost center, the widget quantity, estimated hours, hourly rate, estimated cost, markup, discount, and price. Dimensions are those attributes that the users want to group by when exploring the data. The users are interested in grouping by the various job specifications and by the cost center. The users also need to be able to group by date ranges. The estimate date and the win date are both of interest. Grouping by customer and promotion are also of interest to the users. These become the dimensions of the star schema for the estimating process.

Next, we identify the measures. Measures are the columns that contain values to be aggregated when rows are grouped together. The measures in the estimating process are estimated hours, hourly rate, estimated cost, markup, discount, and price.

The star schema resulting from the analysis of the estimating process is shown in Figure 8.6. There are five widget qualities of interest: shape, color, texture, density, and size. For example, a given widget might be a medium round red fuzzy fluffy widget. The estimate and job numbers are included as degenerate dimensions. The rest of the dimensions and measures are as outlined in the previous two paragraphs.

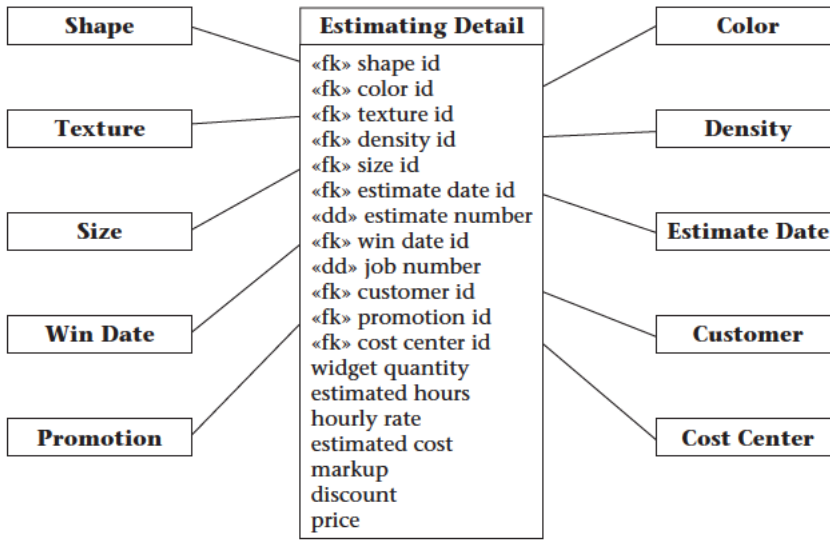


Figure 8.6 Star schema for estimating process

Dimension values are categorical in nature. For example, a given widget might have a density of fluffy or heavy. The values for the size dimension include small, medium, and large. Measures tend to be numeric, since they are typically aggregated using functions such as sum or average.

The dimension tables should include any hierarchies that may be useful for analysis. For example, widgets are offered in many colors. The colors fall into categories by hue (e.g., pink, blue) and intensity (e.g., pastel, hot). Some even glow in the dark! The user may wish to examine all the pastel widgets as a group, or compare pink versus blue widgets. Including these attributes in the dimension table as shown in Figure 8.7 can accommodate this need.

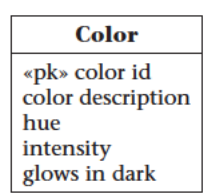


Figure 8.7 Color dimension showing attributes

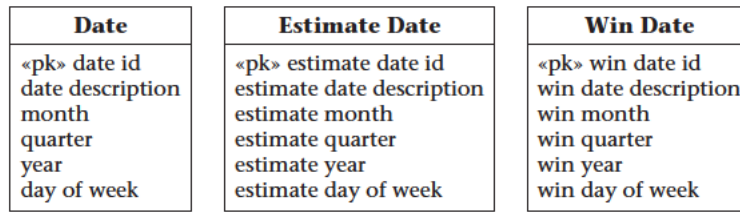


Figure 8.8 Date dimensions showing attributes

Dates can also form hierarchies. For example, the user may wish to group by month, quarter, year or the day of the week. Date dimensions are very common. The estimating process has two date dimensions: the estimate date and the win date. Typically, the date dimensions have analogous attributes. There is an advantage in standardizing the date dimensions across the company. Kimball and Ross [2002] recommend establishing a single standard date dimension, and then creating views of the date dimension for use in multiple dimensions. The use of views provides for standardization, while at the same time allowing the attributes to be named with aliases for intuitive use when multiple date dimensions are present. Figure 8.8 illustrates this concept with a date dimension and two views named Estimate Date and Win Date.

Let's move on to the scheduling process. Scheduling uses the times calculated by the estimating process to plan the workload on each required machine. Target dates are assigned to each manufacturing step. The job ticket moves into production after the scheduling process completes.

XYZ Widget, Inc. has a shop floor automatic data collection (ADC) system. Each job ticket has a bar code for the assigned job number. Each machine has a sheet with bar codes representing the various operations of that machine. Each employee has a badge with a bar code representing that employee. When an employee starts an operation, the job bar code is scanned, the operation bar code is scanned, and the employee bar code is scanned. The computer pulls in the current system time as the start time. When one operation starts, the previous operation for that employee is automatically stopped (an employee is unable do more than one operation at once). When the work on the widget job is complete on that machine, the employee marks the job complete via the ADC system. The information gathered through the ADC system is used to update scheduling, track the employee's work hours and productivity, and also track the machine's productivity.

The design of a star schema for the scheduling process begins by determining the granularity. The most detailed scheduling table in the operational system has a record for each cost center applicable to manufacturing each job. The users in the scheduling department are interested in drilling down to this level of detail in the data warehouse. The proper level of granularity in the star schema for scheduling is determined by the job number and the cost center.

Next we determine the dimensions in the star schema for the scheduling process. The operational scheduling system tracks the scheduled start and finish date and times, as well as the actual start and finish date and times. The estimated and actual hours are also stored in the operational scheduling details table, along with a flag indicating whether the operation completed on time. The scheduling team must have the ability to group records by the scheduled and actual start and finish times. Also critical is the ability to group by cost center. The dimensions of the star schema for scheduling are the scheduled and actual start and finish date and times, and the cost center. The job number must also be included as a degenerate dimension to maintain the proper granularity in the fact table. Figure 8.9 reflects the decisions on the dimensions appropriate for the scheduling process.

The scheduling team is interested in aggregating the estimated hours and, also, the actual hours. They are also very interested in examining trends in on-time performance. The appropriate measures for the scheduling star schema include the estimated and actual hours and a flag indicating whether the operation was finished on time. The appropriate measures for scheduling are reflected in Figure 8.9.

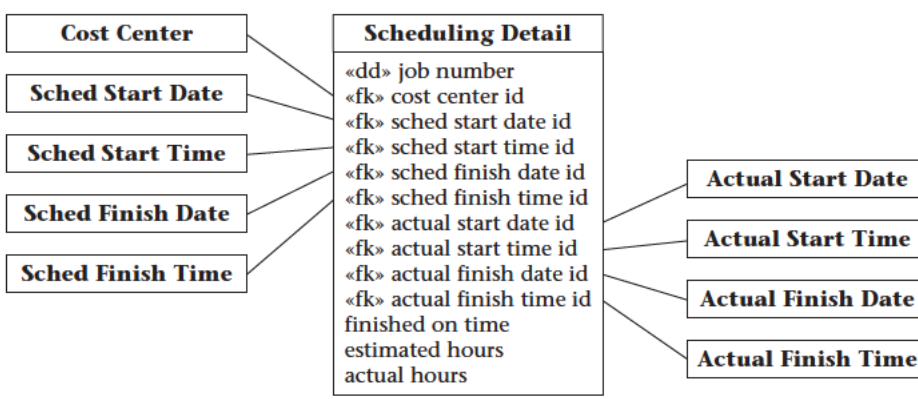


Figure 8.9 Star schema for the scheduling process

There are several standardization principles in play in Figure 8.9. Note that there are multiple time dimensions. These should be standardized with a single time dimension, along with views filling the different roles, similar to the approach used for the date dimensions. Also, notice the Cost Center dimension is present both in the estimating and the scheduling processes. These are actually the same, and should be designed as a single dimension. Dimensions can be shared between multiple star schemas. One last point: the estimated hours are carried from estimating into scheduling in the operational systems. These numbers feed into the star schemas for both the estimating and the scheduling processes. The meaning is the same between the two attributes; therefore, they are both named "estimated hours." The rule of thumb is that if two attributes carry the same meaning, they should be named the same, and if two attributes are named the same, they carry the same meaning. This consistency allows discussion and comparison of information between business processes across the company.

The next process we examine is productivity tracking. The granularity is determined by the level of detail available in the ADC system. The detail includes the job number, cost center, employee number, and the start and finish date and time. The department managers need to be able to group rows by cost center, employee, and start and finish date and times. These attributes therefore become the dimensions of the star schema for the productivity process, shown in Figure 8.10. The managers are interested in aggregating productivity numbers, including the widget quantity produced, the percentage finished on time and the estimated and actual hours. Since these attributes are to be aggregated, they become the measures shown in Figure 8.10.

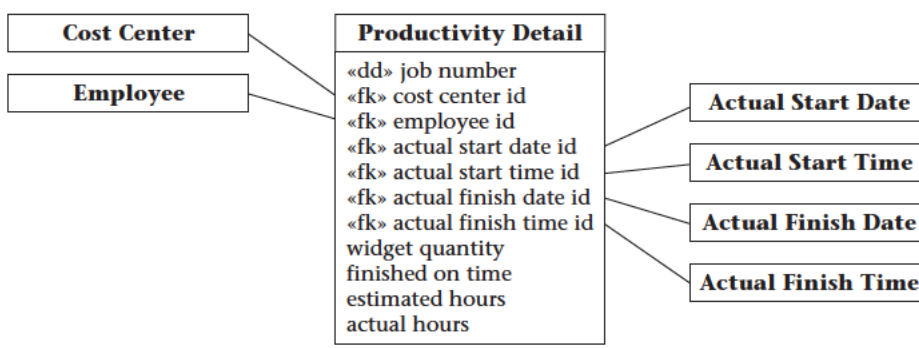


Figure 8.10 Star schema for the productivity tracking process

There are often dimensions in common between star schemas in a data warehouse, because business processes are usually interlinked. A useful tool for tracking the commonality and differences of dimensions across multiple business processes is the data warehouse bus [Kimball and Ross, 2002]. Table 8.2 shows a data warehouse bus for the four business processes in our dimensional design example. Each row represents a business process. Each column represents a dimension. Each X in the body of the table represents the use of the given dimension in the given business process. The data warehouse bus is a handy means of presenting the organization of a data warehouse at a high level. The dimensions common between multiple business processes need to be standardized or "conformed" in Kimball and Ross's terminology. A dimension is conformed if there exists a most detailed version of that dimension, and all other uses of that dimension utilize a subset of the attributes and a subset of the rows from that most detailed version. Conforming dimensions ensures that whenever data are related or compared across business processes, the result is meaningful.

The data warehouse bus also makes some design decisions more obvious. We have taken the liberty of choosing the dimensions for the job-costing process. Table 8.2 includes a row for the job-costing process. When you compare the rows for estimating and job costing, it quickly becomes clear that the two processes have most of the same dimensions. It probably makes sense to combine these two processes into one star

Table 8.2 Data Warehouse Bus for Widget Example

	Shape	*Color*	*Texture*	*Density*	*Size*	*Estimate Date*	*Win Date*	*Customer*	*Promotion*	*Cost Center*	*Sched Start Date*	*Sched Start Time*	*Sched Finish Date*	*Sched Finish Time*	*Actual Start Date*	*Actual Start Time*	*Actual Finish Date*	*Actual Finish Time*	*Employee*	*Invoice Date*
Estimating	X	X	X	X	X	X	X	X	X	X										
Scheduling										X	X	X	X	X	X	X	X	X		
Productivity Tracking										X					X	X	X	X	X	
Job Costing	X	X	X	X	X			X	X	X										X

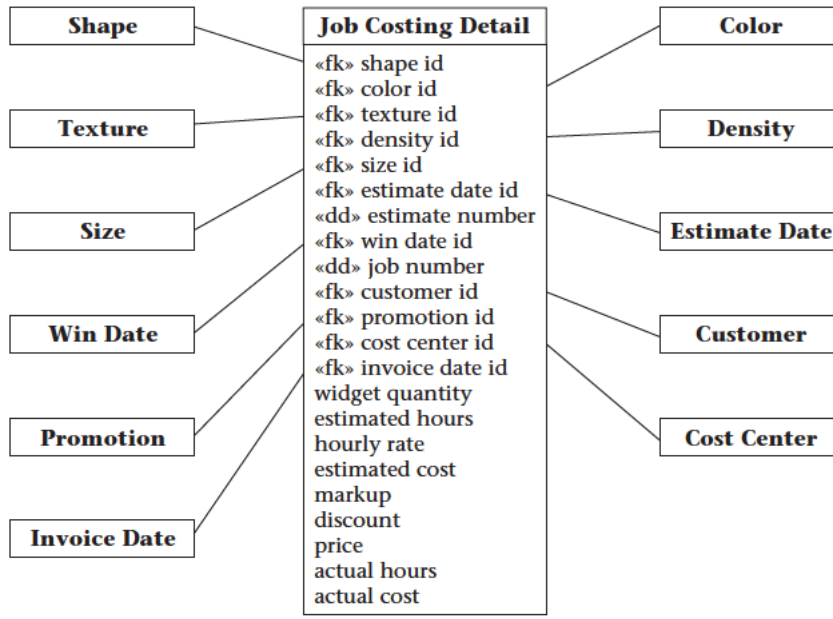


Figure 8.11 Star schema for the job costing process

schema. This is especially true since job-costing analysis requires comparing estimated and actual values. Figure 8.11 is the result of combining the estimating and job costing processes into one star schema.

Summarizing Data

The star schemas we have covered so far are excellent for capturing the pertinent details. Having fine granularity available in the fact table allows the users to examine data down to that level of granularity. However, the users will often want summaries. For example, the managers may often query for a daily snapshot of the job-costing data. Every query the user may wish to pose against a given star schema can be answered from the detailed fact table. The summary could be aggregated on the fly from the fact table. There is an obvious drawback to this strategy. The fact table contains many millions of rows, due to the detailed nature of the data. Producing a summary on the fly can be expensive in terms of computer resources, resulting in a very slow response. If a summary table were available to answer the queries for the job costing daily snapshot, then the answer could be presented to the user blazingly fast.

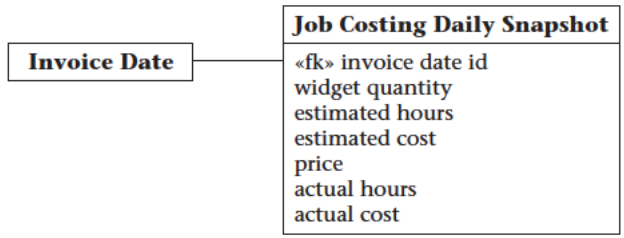


Figure 8.12 Schema for the job costing daily snapshot

The schema for the job costing daily snapshot is shown in Figure 8.12. Notice that most of the dimensions used in the job-costing detail are not used in the snapshot. Summarizing the data has eliminated the need for most dimensions in this context. The daily snapshot contains one row for each day that jobs have been invoiced. The number of rows in the snapshot would be in the thousands. The small size of the snapshot allows very quick response when a user requests the job costing daily snapshot. When there are a small number of summary queries that occur frequently, it is a good strategy to materialize the summary data needed to answer the queries quickly.

The daily snapshot schema in Figure 8.12 also allows the user to group by month, quarter, or year. Materializing summary data is useful for quick response to any query that can be answered by aggregating the data further.

8.2 Online Analytical Processing (OLAP)

Designing and implementing strategic summary tables is a good approach when there is a small set of frequent queries for summary data. However, there may be a need for some users to explore the data in an *ad hoc* fashion. For example, a user who is looking for types of jobs that have not been profitable needs to be able to roll up and drill down various dimensions of the data. The *ad hoc* nature of the process makes predicting the queries impossible. Designing a strategic set of summary tables to answer these *ad hoc* explorations of the data is a daunting task. OLAP provides an alternative. OLAP is a service that overlays the data warehouse. The OLAP system automatically selects a strategic set of summary views, and saves the automatic summary tables (AST) to disk as materialized views. The OLAP system also maintains these views, keep-

ing them in step with the fact tables as new data arrives. When a user requests summary data, the OLAP system figures out which AST can be used for a quick response to the given query. OLAP systems are a good solution when there is a need for *ad hoc* exploration of summary information based on large amounts of data residing in a data warehouse.

OLAP systems automatically select, maintain, and use the ASTs. Thus, an OLAP system effectively does some of the design work automatically. This section covers some of the issues that arise in building an OLAP engine, and some of the possible solutions. If you use an OLAP system, the vendor delivers the OLAP engine to you. The issues and solutions discussed here are not items that you need to resolve. Our goal here is to remove some of the mystery about what an OLAP system is and how it works.

8.2.1 The Exponential Explosion of Views

Materialized views aggregated from a fact table can be uniquely identified by the aggregation level for each dimension. Given a hierarchy along a dimension, let 0 represent no aggregation, 1 represent the first level of aggregation, and so on. For example, if the Invoice Date dimension has a hierarchy consisting of date id, month, quarter, year and "all" (i.e., complete aggregation), then date id is level 0, month is level 1, quarter is level 2, year is level 3, and "all" is level 4. If a dimension does not explicitly have a hierarchy, then level 0 is no aggregation, and level 1 is "all." The scales so defined along each dimension define a coordinate system for uniquely identifying each view in a product graph. Figure 8.13 illustrates a product graph in two dimensions. Product graphs are a generalization of the hypercube lattice structure introduced by Harinarayan, Rajaraman, and Ullman [1996], where dimensions may have associated hierarchies. The top node, labeled (0, 0) in Figure 8.13, represents the fact table. Each node represents a view with aggregation levels as indicated by the coordinate. The relationships descending the product graph indicate aggregation relationships. The five shaded nodes indicate that these views have been materialized. A view can be aggregated from any materialized ancestor view. For example, if a user issues a query for rows grouped by year and state, that query would naturally be answered by the view labeled (3, 2). View (3, 2) is not materialized, but the query can be answered from the materialized view (2, 1) since (2, 1) is an ancestor of (3, 2). Quarters can be aggregated into years, and cities can be aggregated into states.

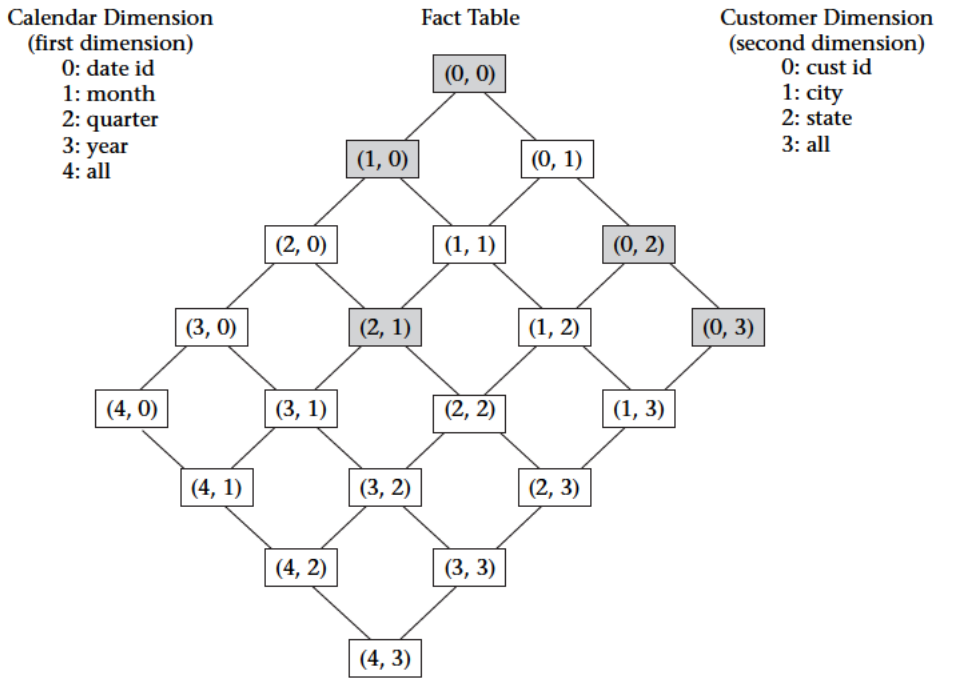


Figure 8.13 Product graph labeled with aggregation level coordinates

The central issue challenging the design of OLAP systems is the exponential explosion of possible views as the number of dimensions increases. The Calendar dimension in Figure 8.13 has five levels of hierarchy, and the Customer dimension has four levels of hierarchy. The user may choose any level of aggregation along each dimension. The number of possible views is the product of the number of hierarchical levels along each dimension. The number of possible views for the example in Figure 8.13 is $5 \times 4 = 20$. Let d be the number of dimensions in a data warehouse. Let h_i be the number of hierarchical levels in dimension i. The general equation for calculating the number of possible views is given by Equation 8.1.

$$\text{Possible views} = \prod_{i=1}^{d} h_i \qquad 8.1$$

If we express Equation 8.1 in different terms, the problem of exponential explosion becomes more apparent. Let g be the geometric mean

of the number of hierarchical levels in the dimensions. Then Equation 8.1 becomes Equation 8.2.

$$\text{Possible views} = g^d \qquad 8.2$$

As dimensionality increases linearly, the number of possible views explodes exponentially. If $g = 5$ and $d = 5$, there are $5^5 = 3{,}125$ possible views. Thus if $d = 10$, then there are $5^{10} = 9{,}765{,}625$ possible views. OLAP administrators need the freedom to scale up the dimensionality of their data warehouses. Clearly the OLAP system cannot create and maintain all possible views as dimensionality increases. The design of OLAP systems must deliver quick response while maintaining a system within the resource limitations. Typically, a strategic subset of views must be selected for materialization.

8.2.2 Overview of OLAP

There are many approaches to implementing OLAP systems presented in the literature. Figure 8.14 maps out one possible approach, which will serve for discussion. The larger problem of OLAP optimization is broken into four subproblems: view size estimation, materialized view selection, materialized view maintenance, and query optimization with materialized views. This division is generally true of the OLAP literature, and is reflected in the OLAP system plan shown in Figure 8.14.

We describe how the OLAP processes interact in Figure 8.14, and then explore each process in greater detail. The plan for OLAP optimization shows *Sample Data* moving from the *Fact Table* into *View Size Estimation*. *View Selection* makes an *Estimate Request* for the view size of each view it considers for materialization. *View Size Estimation* queries the *Sample Data*, examines it, and models the distribution. The distribution observed in the sample is used to estimate the expected number of rows in the view for the full dataset. The *Estimated View Size* is passed to *View Selection*, which uses the estimates to evaluate the relative benefits of materializing the various views under consideration. *View Selection* picks *Strategically Selected Views* for materialization with the goal of minimizing total query costs. *View Maintenance* builds the original views from the *Initial Data* from the *Fact Table*, and maintains the views as *Incremental Data* arrives from *Updates*. *View Maintenance* sends statistics on *View Costs* back to *View Selection*, allowing costly views to be discarded dynamically. *View Maintenance* offers *Current Views* for use by *Query Optimization*. *Query Optimization* must consider which of the *Current Views*

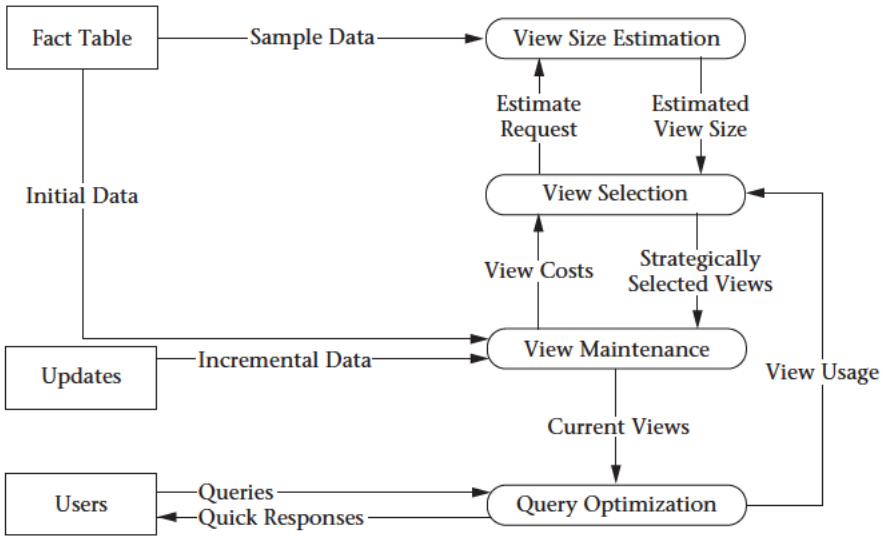


Figure 8.14 A plan for OLAP optimization

can be utilized to most efficiently answer *Queries* from *Users*, giving *Quick Responses* to the *Users*. *View Usage* feeds back into *View Selection*, allowing the system to dynamically adapt to changes in query workloads.

8.2.3 View Size Estimation

OLAP systems selectively materialize strategic views with high benefits to achieve quick response to queries, while remaining within the resource limits of the computer system. The size of a view affects how much disk space is required to store the view. More importantly, the size of the view determines in part how much disk input/output will be consumed when querying and maintaining the view. Calculating the exact size of a given view requires calculating the view from the base data. Reading the base data and calculating the view is the majority of the work necessary to materialize the view. Since the objective of view materialization is to conserve resources, it becomes necessary to estimate the size of the views under consideration for materialization.

Cardenas' formula [Cardenas, 1975] is a simple equation (Equation 8.3) that is applicable to estimating the number of rows in a view:

Let n be the number of rows in the fact table.

Let v be the number of possible keys in the data space of the view.

$$\text{Expected distinct values} = v(1 - (1 - 1/v)^n) \tag{8.3}$$

Cardenas' formula assumes a uniform data distribution. However, many data distributions exist. The data distribution in the fact table affects the number of rows in a view. Cardenas' formula is very quick, but the assumption of a uniform data distribution leads to gross overestimates of the view size when the data is actually clustered. Other methods have been developed to model the effect of data distribution on the number of rows in a view.

Faloutsos, Matias, and Silberschatz [1996] present a sampling approach based on the binomial multifractal distribution. Parameters of the distribution are estimated from a sample. The number of rows in the aggregated view for the full data set is then estimated using the parameter values determined from the sample. Equations 8.4 and 8.5 [Faloutsos, Matias, and Silberschatz, 1996] are presented for this purpose.

$$\text{Expected distinct values} = \sum_{a=0}^{k} C_a^k (1 - (1 - P_a)^n) \tag{8.4}$$

$$P_a = P^{k-a}(1 - P)^a \tag{8.5}$$

Figure 8.15 illustrates an example. Order k is the decision tree depth. C_a^k is the number of bins in the set reachable by taking some combination of a left hand edges and $k - a$ right hand edges in the decision tree. P_a is the probability of reaching a given bin whose path contains a left hand edges. n is the number of rows in the data set. Bias P is the probability of selecting the right hand edge at a choice point in the tree.

The calculations of Equation 8.4 are illustrated with a small example. An actual database would yield much larger numbers, but the concepts and the equations are the same. These calculations can be done with logarithms, resulting in very good scalability. Based on Figure 8.15, given five rows, calculate the expected distinct values using Equation 8.4:

$$\begin{aligned}&\text{Expected distinct values} = \\ &1 \cdot (1 - (1 - 0.729)^5) + 3 \cdot (1 - (1 - 0.081)^5) + \\ &3 \cdot (1 - (1 - 0.009)^5) + 1 \cdot (1 - (1 - 0.001)^5) \approx 1.965\end{aligned} \tag{8.6}$$

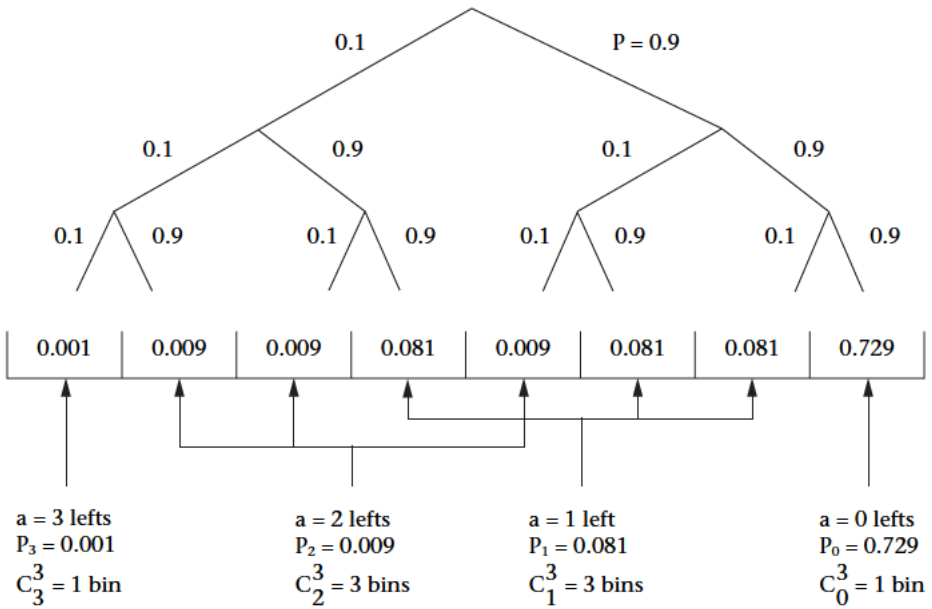


Figure 8.15 Example of a binomial multifractal distribution tree

The values of P and k can be estimated based on sample data. The algorithm used in [Faloutsos, Matias, and Silberschatz, 1996] has three inputs: the number of rows in the sample, the frequency of the most commonly occurring value, and the number of distinct aggregate rows in the sample. The value of P is calculated based on the frequency of the most commonly occurring value. They begin with:

$$k = \lceil \text{Log}_2(\text{Distinct rows in sample}) \rceil \qquad 8.7$$

and then adjust k upwards, recalculating P until a good fit to the number of distinct rows in the sample is found.

Other distribution models can be utilized to predict the size of a view based on sample data. For example, the use of the Pareto distribution model has been explored [Nadeau and Teorey, 2003]. Another possibility is to find the best fit to the sample data for multiple distribution models, calculate which model is most likely to produce the given sample data, and then use that model to predict the number of rows for the full data set. This would require calculation for each distribution model considered, but should generally result in more accurate estimates.

8.2.4 Selection of Materialized Views

Most of the published works on the problem of materialized view selection are based on the hypercube lattice structure [Harinarayan, Rajaraman, and Ullman, 1996]. The hypercube lattice structure is a special case of the product graph structure, where the number of hierarchical levels for each dimension is two. Each dimension can either be included or excluded from a given view. Thus, the nodes in a hypercube lattice structure represent the power set of the dimensions.

Figure 8.16 illustrates the hypercube lattice structure with an example [Harinarayan, Rajaraman, and Ullman, 1996]. Each node of the lattice structure represents a possible view. Each node is labeled with the set of dimensions in the "group by" list for that view. The numbers associated with the nodes represent the number of rows in the view. These numbers are normally derived from a view size estimation algorithm, as discussed in Section 8.2.3. However, the numbers in Figure 8.16 follow the example as given by Harinarayan et al. [1996]. The relationships between nodes indicate which views can be aggregated from other views. A given view can be calculated from any materialized ancestor view.

We refer to the algorithm for selecting materialized views introduced by Harinarayan et al. [1996] as HRU. The initial state for HRU has only the fact table materialized. HRU calculates the benefit of each possible view during each iteration, and selects the most beneficial view for materialization. Processing continues until a predetermined number of materialized views is reached.

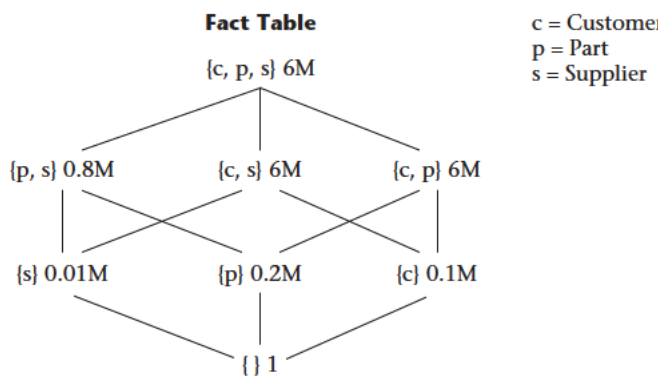


Figure 8.16 Example of a hypercube lattice structure [Harinarayan et al. 1996]

Table 8.3 Two Iterations of HRU, Based on Figure 8.16

	Iteration 1 Benefit	*Iteration 2 Benefit*
{p, s}	**5.2M × 4 = 20.8M**	
{c, s}	0 × 4 = 0	0 × 2 = 0
{c, p}	0 × 4 = 0	0 × 2 = 0
{s}	5.99M × 2 = 11.98M	0.79M × 2 = 1.58M
{p}	5.8M × 2 = 11.6M	0.6M × 2 = 1.2M
{c}	5.9M × 2 = 11.8M	**5.9M × 2 = 11.8M**
{}	6M – 1	0.8M – 1

Table 8.3 shows the calculations for the first two iterations of HRU. Materializing {*p*, *s*} saves 6M – 0.8M = 5.2M rows for each of four views: {*p*, *s*} and its three descendants: {p}, {s}, and {}. The view {*c*, *s*} yields no benefit materialized, since any query that can be answered by reading 6M rows from {*c*, *s*} can also be answered by reading 6M rows from the fact table {*c*, *p*, *s*}. HRU calculates the benefits of each possible view materialization. The view {*p*, *s*} is selected for materialization in the first iteration. The view {*c*} is selected in the second iteration.

HRU is a greedy algorithm that does not guarantee an optimal solution, although testing has shown that it usually produces a good solution. Further research has built upon HRU, accounting for the presence of index structures, update costs, and query frequencies.

HRU evaluates every unselected node during each iteration, and each evaluation considers the effect on every descendant. The algorithm consumes $O(kn^2)$ time, where k = |views to select| and n = |nodes|. This order of complexity looks very good; it is polynomial time. However, the result is misleading. The nodes of the hypercube lattice structure constitute a power set. The number of possible views is therefore 2^d where d = |dimensions|. Thus, $n = 2^d$, and the time complexity of HRU is $O(k2^{2d})$. HRU runs in time exponentially relative to the number of dimensions in the database.

The Polynomial Greedy Algorithm (PGA) [Nadeau and Teorey, 2002] offers a more scalable alternative to HRU. PGA, like HRU, also selects one view for materialization with each iteration. However, PGA divides each iteration into a nomination phase and a selection phase. The first phase nominates promising views into a candidate set. The second phase estimates the benefits of materializing each candidate, and selects the view with the highest evaluation for materialization.

Table 8.4 First Iteration of PGA, Based on Figure 8.16

Candidates	*Iteration 1 Benefit*
{p, s}	**5.2M × 4 = 20.8M**
{s}	5.99M × 2 = 11.98M
{}	6M – 1

The nomination phase begins at the top of the lattice; in Figure 8.16, this is the node {*c*, *p*, *s*}. PGA nominates the smallest node from amongst the children. The candidate set is now {{*p*, *s*}}. PGA then examines the children of {*p*, *s*} and nominates the smallest child, {*s*}. The process repeats until the bottom of the lattice is reached. The candidate set is then {{*p*, *s*}, {*s*}, {}}. Once a path of candidate views has been nominated, the algorithm enters the selection phase. The resulting calculations are shown in Tables 8.4 and 8.5.

Compare Tables 8.4 and 8.5 with Table 8.3. Notice PGA does fewer calculations than HRU, and yet in this example reaches the same decisions as HRU. PGA usually picks a set of views nearly as beneficial as those chosen by HRU, and yet PGA is able to function when HRU fails due to the exponential complexity. PGA is polynomial relative to the number of dimensions. When HRU fails, PGA extends the usefulness of the OLAP system.

The materialized view selection algorithms discussed so far are static; that is, the views are picked once and then materialized. An entirely different approach to the selection of materialized views is to treat the problem similar to memory management [Kotidis and Roussopoulos, 1999]. The materialized views constitute a view pool. Metadata is tracked on usage of the views. The system monitors both space and update window constraints. The contents of the view pool are adjusted dynamically. As queries are posed, views are added appropriately. Whenever a constraint is violated, the system selects a view for eviction. Thus the

Table 8.5 Second Iteration of PGA, Based on Figure 8.16

Candidates	*Iteration 2 Benefit*
{c, s}	0 × 2 = 0
{s}	0.79M × 2 = 1.58M
{c}	**5.9M × 2 = 11.8M**
{}	6M – 1

view pool can improve as more usage statistics are gathered. This is a self-tuning system that adjusts to changing query patterns.

The static and dynamic approaches complement each other and should be integrated. Static approaches run fast from the beginning, but do not adapt. Dynamic view selection begins with an empty view pool, and therefore yields slow response times when a data warehouse is first loaded; however, it is adaptable and improves over time. The complementary nature of these two approaches has influenced our design plan in Figure 8.14, as indicated by *Queries* feeding back into *View Selection*.

8.2.5 View Maintenance

Once a view is selected for materialization, it must be computed and stored. When the base data is updated, the aggregated view must also be updated to maintain consistency between views. The original view materialization and the incremental updates are both considered as view maintenance in Figure 8.14. The efficiency of view maintenance is greatly affected by the data structures implementing the view. OLAP systems are multidimensional, and fact tables contain large numbers of rows. The access methods implementing the OLAP system must meet the challenges of high dimensionality in combination with large row counts. The physical structures used are deferred to volume two, which covers physical design.

Most of the research papers in the area of view maintenance assume that new data is periodically loaded with incremental data during designated update windows. Typically, the OLAP system is made unavailable to the users while the incremental data is loaded in bulk, taking advantage of the efficiencies of bulk operations. There is a down side to deferring the loading of incremental data until the next update window. If the data warehouse receives incremental data once a day, then there is a one-day latency period.

There is currently a push in the industry to accommodate data updates close to real time, keeping the data warehouse in step with the operational systems. This is sometimes referred to as "active warehousing" and "real-time analytics." The need for data latency of only a few minutes presents new problems. How can very large data structures be maintained efficiently with a trickle feed? One solution is to have a second set of data structures with the same schema as the data warehouse. This second set of data structures acts as a holding tank for incremental data, and is referred to as a delta cube in OLAP terminology. The operational systems feed into the delta cube, which is small and efficient for

quick incremental changes. The data cube is updated periodically from the delta cube, taking advantage of bulk operation efficiencies. When the user queries the OLAP system, the query can be issued against both the data cube and the delta cube to obtain an up-to-date result. The delta cube is hidden from the user. What the user sees is an OLAP system that is nearly current with the operational systems.

8.2.6 Query Optimization

When a query is posed to an OLAP system, there may be multiple materialized views available that could be used to compute the result. For example, if we have the situation represented in Figure 8.13, and a user issues a query to group rows by month and state, that query is naturally answered from the view labeled (1, 2). However, since (1, 2) is not materialized, we need to find a materialized ancestor to obtain the data. There are three such nodes in the product graph of Figure 8.13. The query can be answered from nodes (0, 0), (1, 0), or (0, 2). With the possibility of answering queries from alternative sources, the optimization issue arises as to which source is the most efficient for the given query. Most existing research focuses on syntactic approaches. The possible query translations are carried out, alternative query costs are estimated, and what appears to be the best plan is executed. Another approach is to query a metadata table containing information on the materialized views to determine the best view to query against, and then translate the original SQL query to use the best view.

Database systems contain metadata tables that hold data about the tables and other structures used by the system. The metadata tables facilitate the system in its operations. Here's an example where a metadata

Table 8.6 Example of Materialized View Metadata

Dimensions			
Calendar	*Customer*	*Blocks*	*ViewID*
0	0	10,000,000	1
0	2	50,000	3
0	3	1,000	5
1	0	300,000	2
2	1	10,000	4

table can facilitate the process of finding the best view to answer a query in an OLAP system. The coordinate system defined by the aggregation levels forms the basis for organizing the metadata for tracking the materialized views. Table 8.6 displays the metadata for the materialized views shaded in Figure 8.13. The two dimensions labeled *Calendar* and *Customer* form the composite key. The *Blocks* column tracks the actual number of blocks in each materialized view. The *ViewID* column is used to identify the associated materialized view. The implementation stores materialized views as tables where the value of the *ViewID* forms part of the table name. For example, the row with *ViewID* = 3 contains information on the aggregated view that is materialized as table **AST3** (short for automatic summary table 3).

Observe the general pattern in the coordinates of the views in the product graph with regard to ancestor relationships. Let Value(V, d) represent a function that returns the aggregation level for view V along dimension d. For any two views V_i and V_j where $V_i \neq V_j$, V_i is an ancestor of V_j if and only if for every dimension d of the composite key, Value(V_i, d) $\leq$ Value(V_j, d). This pattern in the keys can be utilized to identify ancestors of a given view by querying the metadata. The semantics of the product graph are captured by the metadata, permitting the OLAP system to search semantically for the best materialized ancestor view by querying the metadata table. After the best materialized view is determined, the OLAP system can rewrite the original query to utilize the best materialized view, and proceed.

8.3 Data Mining

Two general approaches are used to extract knowledge from a database. First, a user may have a hypothesis to verify or disprove. This type of analysis is done with standard database queries and statistical analysis. The second approach to extracting knowledge is to have the computer search for correlations in the data, and present promising hypotheses to the user for consideration. The methods included here are data mining techniques developed in the fields of Machine Learning and Knowledge Discovery.

Data mining algorithms attempt to solve a number of common problems. One general problem is categorization: given a set of cases with known values for some parameters, classify the cases. For example, given observations of patients, suggest a diagnosis. Another general problem type is clustering: given a set of cases, find natural groupings of the cases. Clustering is useful, for example, in identifying market seg-

ments. Association rules, also known as market basket analyses, are another common problem. Businesses sometimes want to know what items are frequently purchased together. This knowledge is useful, for example, when decisions are made about how to lay out a grocery store. There are many types of data mining available. Han and Kamber [2001] cover data mining in the context of data warehouses and OLAP systems. Mitchell [1997] is a rich resource, written from the machine learning perspective. Witten and Frank [2000] give a survey of data mining, along with freeware written in Java available from the Weka Web site [http://www.cs.waikato.ac.nz/ml/weka]. The Weka Web site is a good option for those who wish to experiment with and modify existing algorithms. The major database vendors also offer data mining packages that function with their databases.

Due to the large scope of data mining, we focus on two forms of data mining: forecasting and text mining.

8.3.1 Forecasting

Forecasting is a form of data mining in which trends are modeled over time using known data, and future trends are predicted based on the model. There are many different prediction models with varying levels of sophistication. Perhaps the simplest model is the least squares line model. The best fit line is calculated from known data points using the method of least squares. The line is projected into the future to determine predictions. Figure 8.17 shows a least squares line for an actual data set. The crossed (jagged) points represent actual known data. The circular (dots) points represent the least squares line. When the least squares line projects beyond the known points, this region represents predictions. The intervals associated with the predictions in our figures represent a 90% prediction interval. That is, given an interval, there is a 90% probability that the actual value, when known, will lie in that interval.

The least squares line approach weights each known data point equally when building the model. The predicted upward trend in Figure 8.17 does not give any special consideration to the recent downturn.

Exponential smoothing is an approach that weights recent history more heavily than distant history. Double exponential smoothing models two components: level and trend (hence "double" exponential smoothing). As the known values change in level and trend, the model adapts. Figure 8.18 shows the predictions made using double exponen-

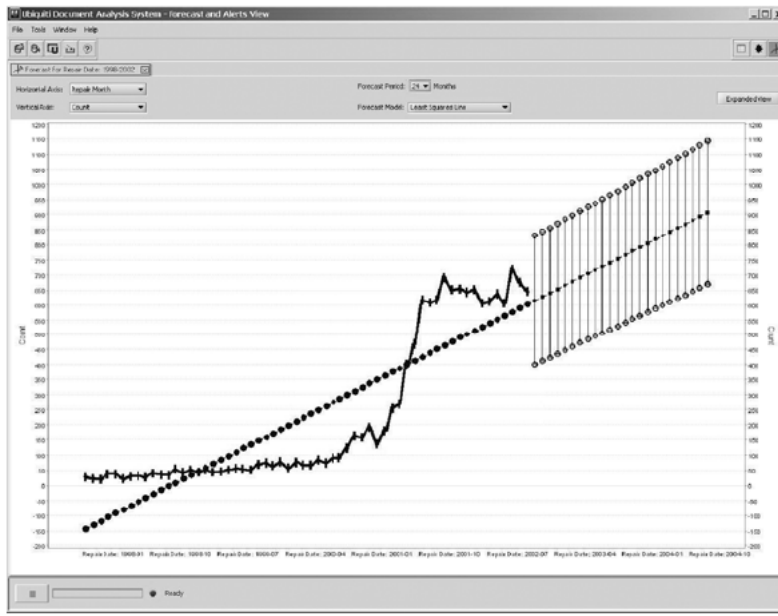


Figure 8.17 Least squares line (courtesy of Ubiquiti, Inc.)

tial smoothing, based on the same data set used to compute Figure 8.17. Notice the prediction is now more tightly bound to recent history.

Triple exponential smoothing models three components: level, trend, and seasonality. This is more sophisticated than double exponential smoothing, and gives better predictions when the data does indeed exhibit seasonal behavior. Figure 8.19 shows the predictions made by triple exponential smoothing, based on the same data used to compute Figures 8.17 and 8.18. Notice the prediction intervals are tighter than in Figures 8.17 and 8.18. This is a sign that the data varies seasonally; triple exponential smoothing is a good model for the given type of data.

Exactly how reliable are these predictions? If we revisit the predictions after time has passed and compare the predictions with the actual values, are they accurate? Figure 8.20 shows the actual data overlaid with the predictions made in Figure 8.19. Most of the actual data points do indeed lie within the prediction intervals. The prediction intervals look very reasonable. Why don't we use these forecast models to make our millions on Wall Street? Take a look at Figure 8.21, a cautionary tale. Figure 8.21 is also based on the triple exponential smoothing model, using four years of known data for training, compared with five years of data used in constructing the model for Figure 8.20. The resulting pre-

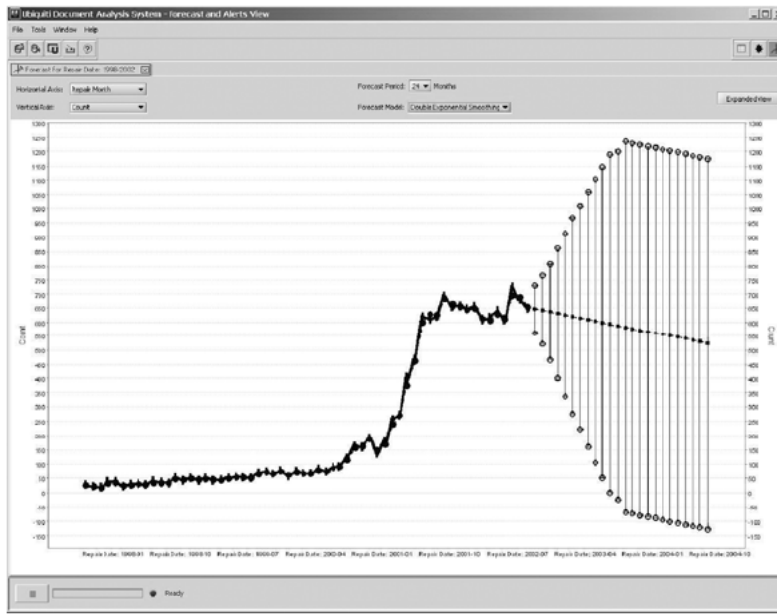


Figure 8.18 Double exponential smoothing (courtesy of Ubiquiti, Inc.)

dictions match for four months, and then diverge greatly from reality. The problem is that forecast models are built on known data, with the assumption that known data forms a good basis for predicting the future. This may be true most of the time; however, forecast models can be unreliable when the market is changing or about to change drastically. Forecasting can be a useful tool, but the predictions must be taken only as indicators.

The details of the forecast models discussed here, as well as many others, can be found in Makridakis et al. [1998].

8.3.2 Text Mining

Most of the work on data processing over the past few decades has used structured data. The vast majority of systems in use today read and store data in relational databases. The schemas are organized neatly in rows and columns. However, there are large amounts of data that reside in freeform text. Descriptions of warranty claims are written in text. Medical records are written in text. Text is everywhere. Only recently has the work in text analysis made significant headway. Companies are now marketing products that focus on text analysis.

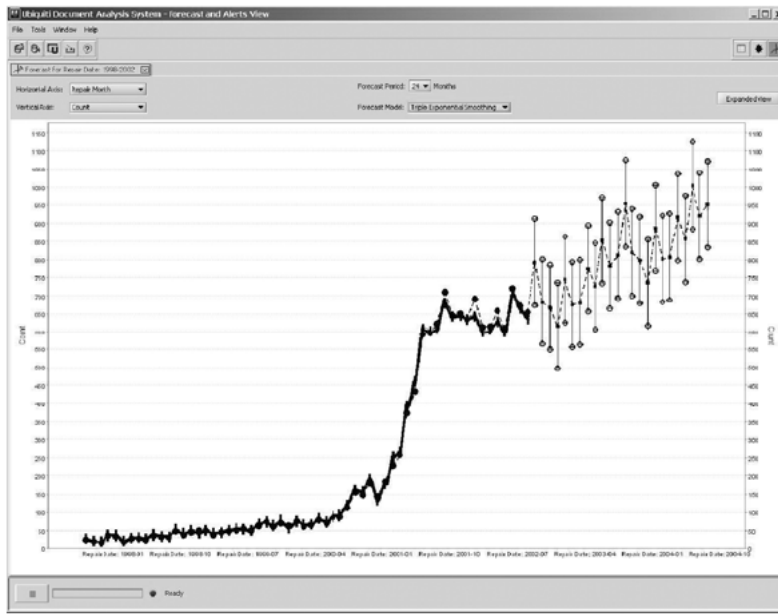


Figure 8.19 Triple exponential smoothing (courtesy of Ubiquiti, Inc.)

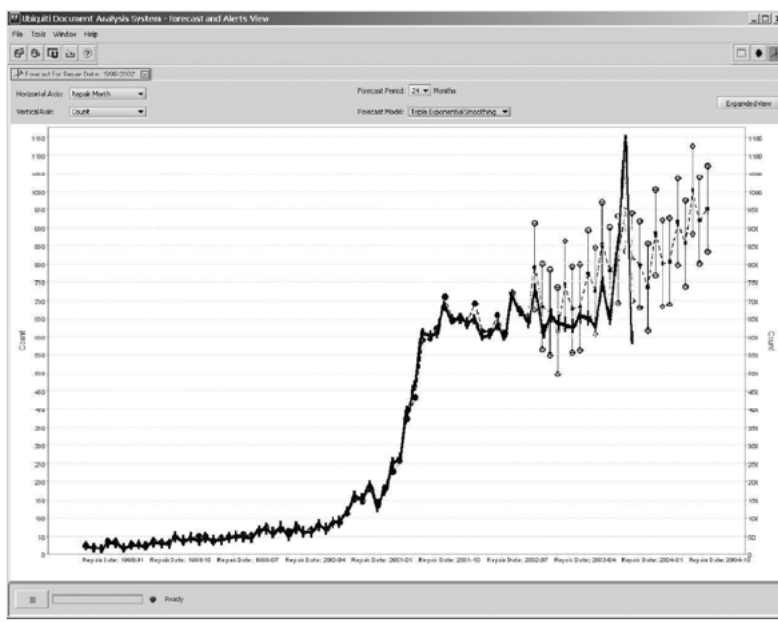


Figure 8.20 Triple exponential smoothing with actual values overlaying forecast values, based on five years of training data (courtesy of Ubiquiti, Inc.)

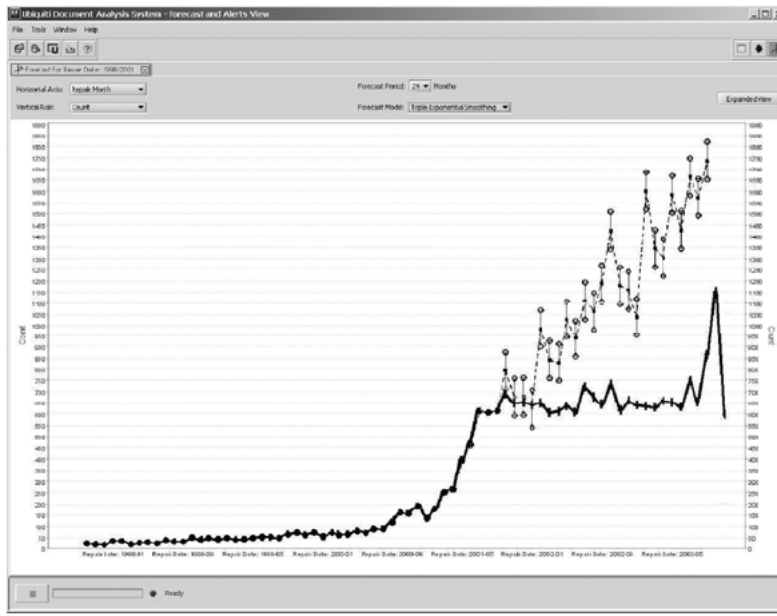


Figure 8.21 Triple exponential smoothing with actual values overlaying forecast values, based on four years of training data (courtesy of Ubiquiti, Inc.)

Let's look at a few of the possibilities for analyzing text and their potential impact. We'll take the area of automotive warranty claims as an example. When something goes wrong with your car, you bring it to an automotive shop for repairs. You describe to a shop representative what you've observed going wrong with your car. Your description is typed into a computer. A mechanic works on your car, and then types in observations about your car and the actions taken to remedy the problem. This is valuable information for the automotive companies and the parts manufacturers. If the information can be analyzed, they can catch problems early and build better cars. They can reduce breakdowns, saving themselves money, and saving their customers frustration.

The data typed into the computer is often entered in a hurry. The language includes abbreviations, jargon, misspelled words, and incorrect grammar. Figure 8.22 shows an example entry from an actual warranty claim database.

As you can see, the raw information entered on the shop floor is barely English. Figure 8.23 shows a cleaned up version of the same text.

7 DD40 BASC 54566 CK OUT AC INOP PREFORM PID CK CK PCM PID ACC CK OK OPERATING ON AND OFF PREFORM POWER AND GRONED CK AT COMPRESOR FONED NO GRONED PREFORM PINPONT DIAG AND TRACE GRONED FONED BAD CO NECTION AT S778 REPAIR AND RETEST OK CK AC OPERATION

Figure 8.22 Example of a verbatim description in a warranty claim (courtesy of Ubiquiti, Inc.)

7 DD40 Basic 54566 Check Out Air Conditioning Inoperable Perform PID Check Check Power Control Module PID Accessory Check OK Operating On And Off Perform Power And Ground Check At Compressor Found No Ground Perform Pinpoint Diagnosis And Trace Ground Found Bad Connection At Splice 778 Repair And Retest OK Check Air Conditioning Operation.

Figure 8.23 Cleaned up version of description in warranty claim (courtesy of Ubiquiti, Inc.)

Even the cleaned up version is difficult to read. The companies paying out warranty claims want each claim categorized in various ways, to track what problems are occurring. One option is to hire many people to read the claims and determine how each claim should be categorized. Categorizing the claims manually is tedious work. A more viable option, developed in the last few years, is to apply a software solution. Figure 8.24 shows some of the information that can be gleaned automatically from the text in Figure 8.22.

The software processes the text and determines the concepts likely represented in the text. This is not a simple word search. Synonyms map

Automated Coding	Confidence
Primary Group: Electrical	90 %
Subgroup: Climate Control	85 %
Part: Connector 1008	93 %
Problem: Bad Connection	72 %
Repair: Reconnect	75 %
Location: Engin. Cmprt.	90 %

Figure 8.24 Useful information extracted from verbatim description in warranty claim (courtesy of Ubiquiti, Inc.)

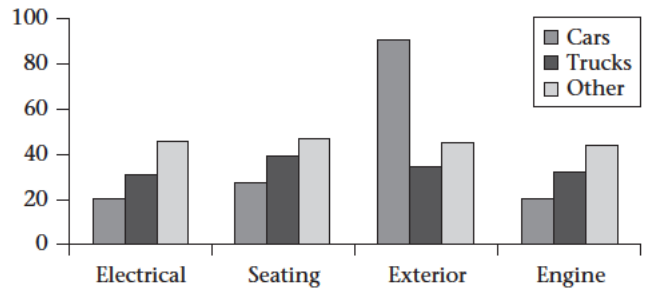


Figure 8.25 Aggregate data from warranty claims (courtesy of Ubiquiti, Inc.)

to the same concept. Some words map to different concepts depending on the context. The software uses an ontology that relates words and concepts to each other. After each warranty is categorized in various ways, it becomes possible to obtain useful aggregate information, as shown in Figure 8.25.

8.4 Summary

Data warehousing, OLAP, and data mining are three areas of computer science that are tightly interlinked and marketed under the heading of business intelligence. The functionalities of these three areas complement each other. Data warehousing provides an infrastructure for storing and accessing large amounts of data in an efficient and user-friendly manner. Dimensional data modeling is the approach best suited for designing data warehouses. OLAP is a service that overlays the data warehouse. The purpose of OLAP is to provide quick response to *ad hoc* queries, typically involving grouping rows and aggregating values. Roll-up and drill-down operations are typical. OLAP systems automatically perform some design tasks, such as selecting which views to materialize in order to provide quick response times. OLAP is a good tool for exploring the data in a human-driven fashion, when the person has a clear question in mind. Data mining is usually computer driven, involving analysis of the data to create likely hypotheses that might be of interest to users. Data mining can bring to the forefront valuable and interesting structure in the data that would otherwise have gone unnoticed.

8.5 Literature Summary

The evolution and principles of data warehouses can be found in Barquin and Edelstein [1997], Cataldo [1997], Chaudhuri and Dayal [1997], Gray and Watson [1998], Kimball and Ross [1998, 2002], and Kimball and Caserta [2004]. OLAP is discussed in Barquin and Edelstein [1997], Faloutsos, Matia, and Silberschatz [1996], Harinarayan, Rajaraman, and Ullman [1996], Kotidis and Roussopoulos [1999], Nadeau and Teorey [2002 2003], Thomsen [1997], and data mining principles and tools can be found in Han and Kamber [2001], Makridakis, Wheelwright, and Hyndman [1998], Mitchell [1997], The University of Waikato [2005], Witten and Frank [2000], among many others.

9 CASE Tools for Logical Database Design

Database design is just one part of the analysis and design phase of creating effective business application software (see Figure 9.1), but it is often the part that is the most challenging and the most critical to performance. In the previous chapters, we explored the classic ways of creating efficient and effective database designs, including ER modeling and the transformation of the ER models into constructs by transformation rules. We also examined normalization, normal forms and denormalization, and specific topologies used in warehousing, such as star schema. All this information may leave your head spinning!

This chapter focuses on commercially available tools to simplify these design processes. These computer-aided system engineering, or CASE, tools provide functions that assist in system design. CASE tools are widely used in numerous industries and domains, such as circuit design, manufacturing, and architecture. Logical database design is another area where CASE tools have proven effective. This chapter explores the offerings of the major vendors in this space: IBM, Computer Associates, and Sybase. Each of these companies offers powerful, feature-rich technology for developing logical database designs and transitioning them into physical databases you can use.

Although it is impossible to present information on software products without some subjectivity and comment, we have sincerely attempted to discuss the capabilities of these products with minimal product bias or critique. Also, it is impossible to describe the features of

Figure 9.1 Business system life cycle (courtesy IBM Corp.)

these products in great detail in a chapter of this sort (a user manual of many hundred pages could be written describing each), so we have set the bar slightly lower, with the aim of surveying these products to give the reader a taste for the capabilities they provide. Further details can be obtained from the manufacturer's Web sites, which are listed in the Literature Summary at the end of the chapter.

9.1 Introduction to the CASE Tools

In this chapter, we will introduce some of the most popular and powerful products available for helping with logical database design: IBM's Rational Data Architect, Computer Associate's AllFusion ERwin Data Modeler, and Sybase's PowerDesigner. These CASE tools help the designer develop a well-designed database by walking through a process of conceptual design, logical design and physical creation, as shown in Figure 9.2.

Computer Associates' AllFusion ERwin Data Modeler has been around the longest. A stand-alone product, AllFusion ERwin's strengths stem from relatively strong support of physical data modeling, the

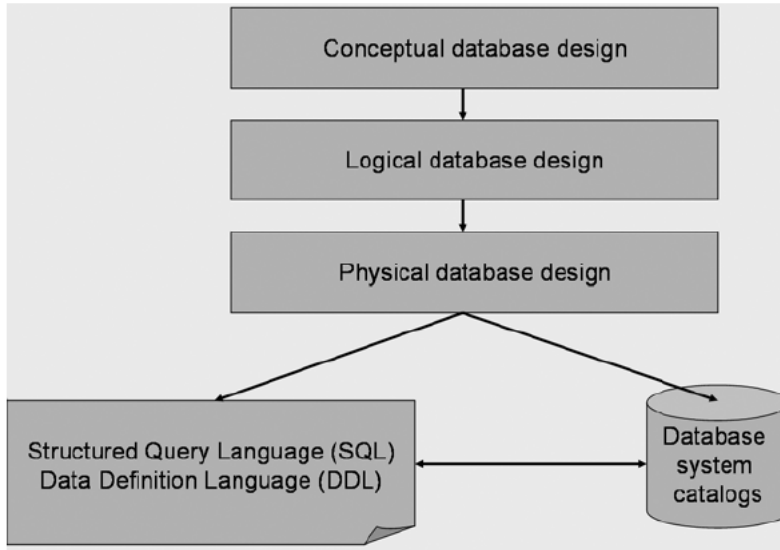


Figure 9.2 Database design process

broadest set of technology partners, and third-party training. What it does it does well, but in recent years it has lagged in some advanced features. Sybase's PowerDesigner has come on strong in the past few years, challenging AllFusion ERwin. It has some advantages in reporting and some advanced features that will be described later in this chapter. IBM's Rational Data Architect is a new product that supplants IBM's previous product, Rational Rose Data Modeler. Its strength lies in strong design checking; rich integration with IBM's broad software development platform, including products from their Rational, Information Management, and Tivoli divisions; and advanced features that will be described below.

In previous chapters, we discussed the aspects of logical database design that CASE tools help design, annotate, apply, and modify. These include, for example, ER, and UML modeling, and how this modeling can be used to develop a logical database design. Within the ER relationship design, there are several types of entity definitions and relationship modeling (unrelated, one-to-many, and many-to-many). These relationships are combined and denormalized into schema patterns known as normal forms (e.g., 3NF, star schema, snowflake schema). An effective design requires the clear definition of keys, such as the primary key, the foreign key, and unique keys within relationships. The addition of constraints to limit the usage (and abuses) of the system within reasonable bounds or business rules is also critical. The effective logical design of

the database will have a profound impact on the performance of the system, as well as the ease with which the database system can be maintained and extended.

There are several other CASE products that we will not discuss in this book. A few additional products worth investigating include Datanamic's DeZign for Databases, QDesigner by Quest Software, Visible Analyst by Standard, and Embarcadero ER/Studio. The Visual Studio .NET Enterprise Architect edition includes a version of Visio with some database design stencils that can be used to create ER models. The cost and function of these tools vary wildly, from open source products up through enterprise software that costs thousands of dollars per license.

The full development cycle includes an iterative cycle of understanding business requirements; defining product requirements; analysis and design; implementation; testing (component, integration, and system); deployment; administration and optimization; and change management. No single product currently covers that entire scope. Instead, product vendors provide, to varying degrees, suites of products that focus on portions of that cycle. CASE tools for database design largely focus on the analysis and design, and to a lesser degree testing, of the database model and creation as illustrated in Figure 9.2.

CASE tools provide software that simplifies or automates some of the steps described in Figure 9.2. Conceptual design includes steps such as describing the business entities and functional requirements of the database; logical design includes definition of entity relationships and normal forms; physical database design helps transform the logical design into actual database objects, such as tables, indexes, and constraints. The software tools provide significant value to database designers by:

1. Dramatically reducing the complexity of conceptual and logical design, both of which can be rather difficult to do well. This reduced complexity results in better database design in less time and with less skill requirements for the user.
2. Automating transformation of the logical design to the physical design (at least the basic physical design). This not only reduces time and skill requirements for the designer, but significantly removes chances of manual error in performing the conversion from the logical model to the physical data definition language (DDL), which the database server will "consume" (i.e., as input) to create the physical database.
3. Providing the reporting, round trip engineering, and reverse engineering that make such tools invaluable in maintaining systems

over a long period of time. System design can and does evolve over time due to changing and expanding business needs. Also, the people who design the system (sometimes teams of people) may not be the same as those charged with maintaining the system. The complexity of large systems combined with the need for continuous adaptability, virtually necessitates the use of CASE tools to help visualize, reverse engineer, and track the system design over time.

You can find a broader list of available database design tools at the Web site "Database Answers" (http://www.databaseanswers.com/modelling_tools.htm), maintained by David Alex Lamb at Queen's University in Kingston, Canada.

9.2 Key Capabilities to Watch For

Design tools should be able to help you with both data modeling and logical database design. Both processes are important. A good distinction between these appears on the "Database Answers" Web site, cited above.

For data modeling, the question you are asking is: What does the world being modeled look like? In particular, you are looking for similarities between things. Then you identify a "supertype" of thing which may have subtypes. For example, Corporate Customers and Personal Customers. If, for example, supplier contacts are conceptually different things from customer contacts, then the answer is that they should be modeled separately. On the other hand, if they are merely subsets of the same thing, then treat them as the same thing.

For database design, you are answering a different question: How can I efficiently design a database that will support the functions of a proposed application or Web site? The key task here is to identify similarities between entities so that you can integrate them into the same table, usually with a "Type" indicator. For example, a Customer table, which combines all attributes of both Corporate and Personal Customers. As a result, it is possible to spend a great deal of time breaking things out when creating a Data Model, and then collapsing them back together when designing the corresponding database.

Support for programmable and physical design attributes with a design tool can also expand the value a tool provides. In database terms, aspects to watch for will include support for indexes, uniqueness, triggers, and stored procedures.

The low-end tools (selling for less than US$100 or available as open source) provide the most basic functionality for ER modeling. The higher end products provide the kinds of support needed for serious project design, such as:

- Complete round trip engineering
- UML design
- Schema evolution; change management
- Reverse engineering of existing systems
- Team support, allowing multiple people to work on the same project concurrently
- Integration with Eclipse and .NET and other tooling products
- Component and convention reuse (being able to reuse naming standard, domain, and logical models over multiple design projects)
- Reusable assets (e.g., extensibility, template)
- Reporting

9.3 The Basics

All of the products in question provide strong, easy to use functions for both data modeling and database design. All of these products provide the ability to graphically represent ER relationships. These tools also provide transformation processes to map from an ER model into an SQL design (DDL), using the transformation types described earlier in Chapter 5:

- Transform each entity into a table containing the key and nonkey attributes of the entity
- Transform every many-to-many binary or binary recursive relationship into a relationship table with the keys of the entities and the attributes of the relationship
- Transform every ternary or higher-level *n*-ary relationship into a relationship table

Similarly these tools produce the transformation table types described in Chapter 5:

- An entity table with the same information content as the original entity
- An entity table with the embedded foreign key of the parent entity
- A relationship table with the foreign keys of all the entities in the relationship

Chapter 5 also described rules for null transformations that must apply, and the CASE tools typically enforce these.

These CASE tools also help with the modeling of normal forms and denormalization to develop a true physical schema for your database, as described in Chapter 5. The tools provide graphical interfaces for physical database design as well as basic modeling of uniqueness, constraints, and indexes. Figure 9.3 shows an example of the IBM Rational Data Architect's GUI for modeling ERs. Figure 9.4 shows a similar snapshot of the interface for Computer Associate's AllFusion ERwin Data Modeler.

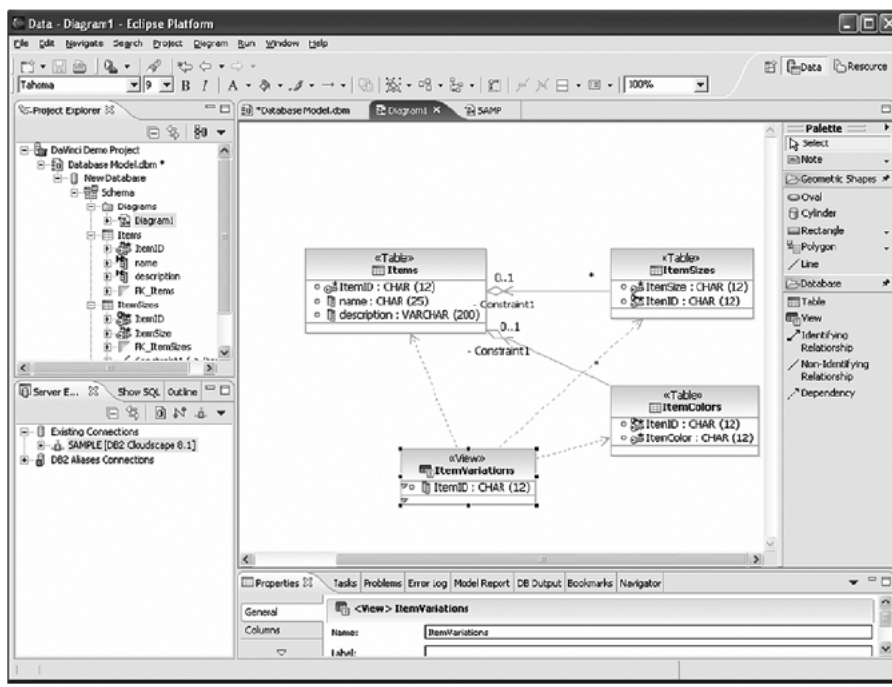


Figure 9.3 Rational Data Architect ER modeling (courtesy IBM Rational Division)

After creating an ER model, the CASE tools enable easy modification of the model and its attributes through graphical interfaces. An example is shown below in Figure 9.5 with IBM's Rational Data Architect, illustrating attribute editing. Of these CASE tools, Rational Data Architect has perhaps the most useful UML modeling function for data modeling and design. Its predecessor, Rational Rose Data Modeler, was the industry's first UML-based data modeler, and IBM has continued its leadership in this area with Rational Data Architect. UML provides a somewhat richer notation than information engineering (IE) entity-relationship diagram (ERD) notation, particularly for conceptual and logical data modeling. However, the IE-ERD notation is older and more commonly used. One of the nice aspects of Rational Data Architect is the ability to work with either UML or IE notation.

Figure 9.6 shows the AllFusion ERwin screen for defining the cardinality of entity relationships. It is worth noting that many relationships do not need to enter this dialog at all.

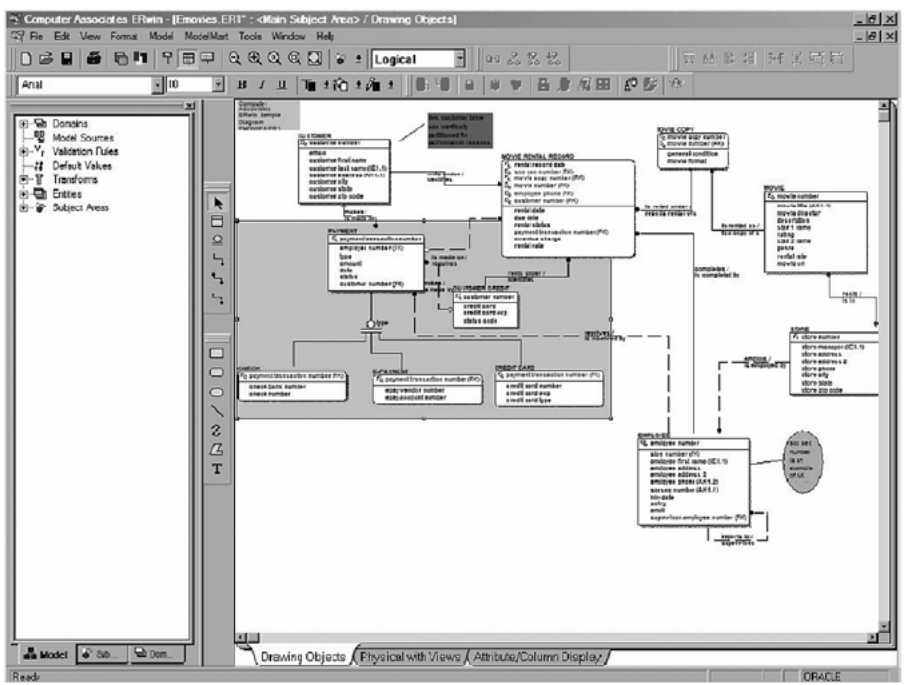


Figure 9.4 AllFusion ERwin Data Modeler ER modeling (picture from Computer Associates, http://agendas.ca.com/Agendas/Presentations/AFM54PN.pdf)

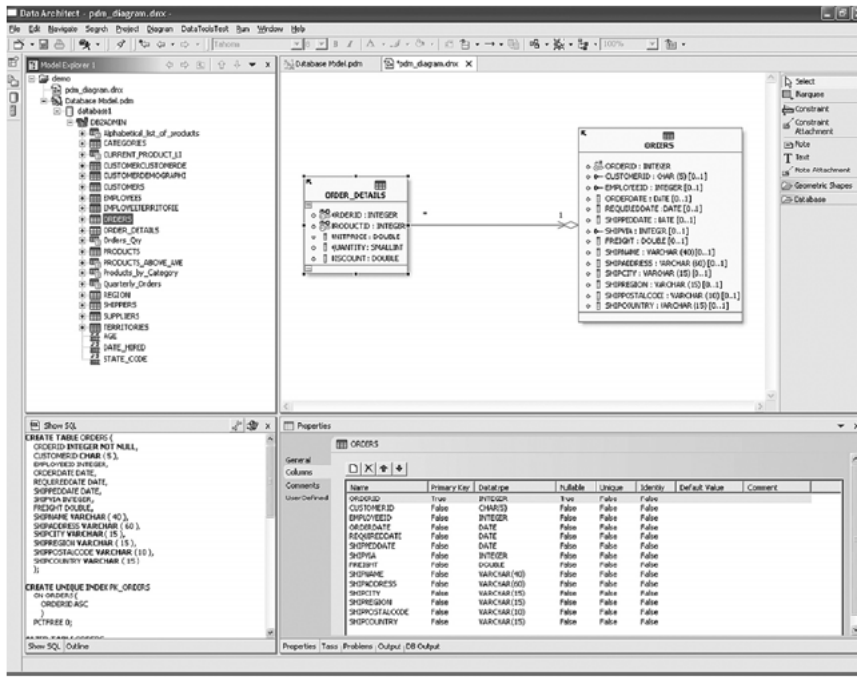


Figure 9.5 Property editing with IBM Rational Data Architect (courtesy IBM Rational Division)

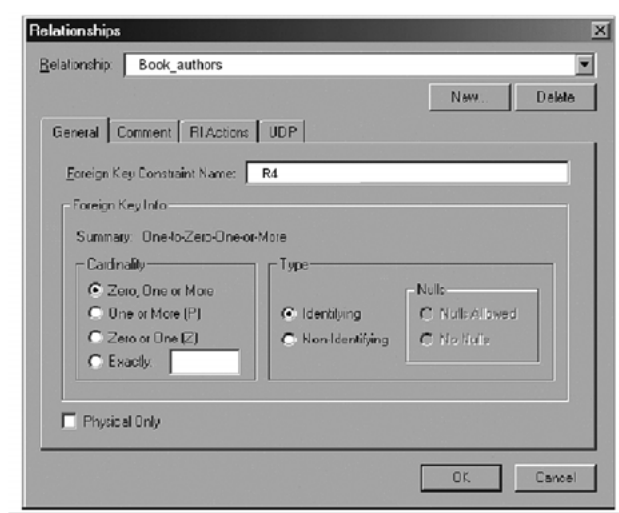


Figure 9.6 Specifying one-to-many relationships with ERwin (courtesy Computer Associates)

Table 9.1 ER Model Containing Sales Relationships

ER Construct	*FDs*
Customer(many): Job(one)	cust-no -> job-title
Order(many): Customer(one)	order-no -> cust-no
Salesperson(many): Department(one)	sales-name -> dept-no
Item(many): Department(one)	item-no -> dept-no
Order(many): Item(many): Salesperson(one)	order-no,item-no->sales-name
Order(many): Department(many): Salesperson(one)	order-no,dept-no-> sales-name

9.4 Generating a Database from a Design

To really take your design to the next level (i.e., a practical level) you will need a good design tool that can interact with your specific database product to actually create the Data Definition Language (DDL) and associated scripts or commands to create and modify the basic database for you. For instance, using the example of Chapter 7, we have modeled an ER model containing the sales relationships shown in Table 9.1.

The CASE tools will automatically generate the required scripts, including the DDL specification to create the actual database, and will provide you with an option to apply those changes to an actual database, as follows:

```
create table customer (cust_no   char(6),
     job_title    varchar(256),
     primary key (cust_no),
     foreign key (job_title) references job
     on delete set null on update cascade);

create table job (job_title   varchar(256),
     primary key (job_title));

create table order (order_no  char(9),
     cust_no      char(6) not null,
     primary key (order_no),
     foreign key (cust_no) references customer
     on delete cascade on update cascade);
```

```
create table salesperson (sales_name varchar(256),
     dept_no      char(2),
     primary key (sales_name),
     foreign key (dept_no) references department
     on delete set null on update cascade);

create table department (dept_no char(2),
     primary key (dept_no));

create table item (item_no   char(6),
     dept_no      char(2),
     primary key (item_no),
     foreign key (dept_no) references department
     on delete set null on update cascade);

create table order_item_sales (order_no   char(9),
     item_no      char(6),
     sales_name   varchar(256) not null,
     primary key (order_no, item_no),
     foreign key (order_no) references order
     on delete cascade on update cascade,
     foreign key (item_no) references item
     on delete cascade on update cascade,
     foreign key (sales_name) references salesperson
     on delete cascade on update cascade);

create table order_dept_sales (order_no   char(9),
     dept_no      char(2),
     sales_name   varchar(256) not null,
     primary key (order_no, dept_no),
     foreign key (order_no) references order
     on delete cascade on update cascade,
     foreign key (dept_no) references department
     on delete cascade on update cascade,
     foreign key (sales_name) references salesperson
     on delete cascade on update cascade);
```

It is worth noting that with all the CASE tools we discuss here, the conversion of the logical design to the physical design is quite rudimentary. These tools help create basic database objects, such as tables and, in some cases, indexes. However, the advanced features of the database server are often not supported—where they are supported, the CASE tool is usually behind by two or three software releases. Developing advanced

physical design features, such as multidimensional clustering or materialized views, is far beyond the capabilities of the logical design tools we are discussing. Advanced physical database design is often highly dependant on data density and data access patterns. One feature of Rational Data Architect that stands out is that it provides linkages with the automatic computing (self-managing) capabilities within DB2 to provide semi-automated selection of advanced physical design attributes.

Figure 9.7 shows an example with ERwin schema generation, generating the DB2 DDL directly from the ER model designed within ERwin.

Other very important capabilities shared by these tools include the ability to reverse-engineer existing databases (for which you may not have an existing ER or physical UML model), and the ability to automatically materialize the physical model or incremental changes of a model onto a real database. This capability enables you to synchronize your database design with a real database as you make changes. This capacity

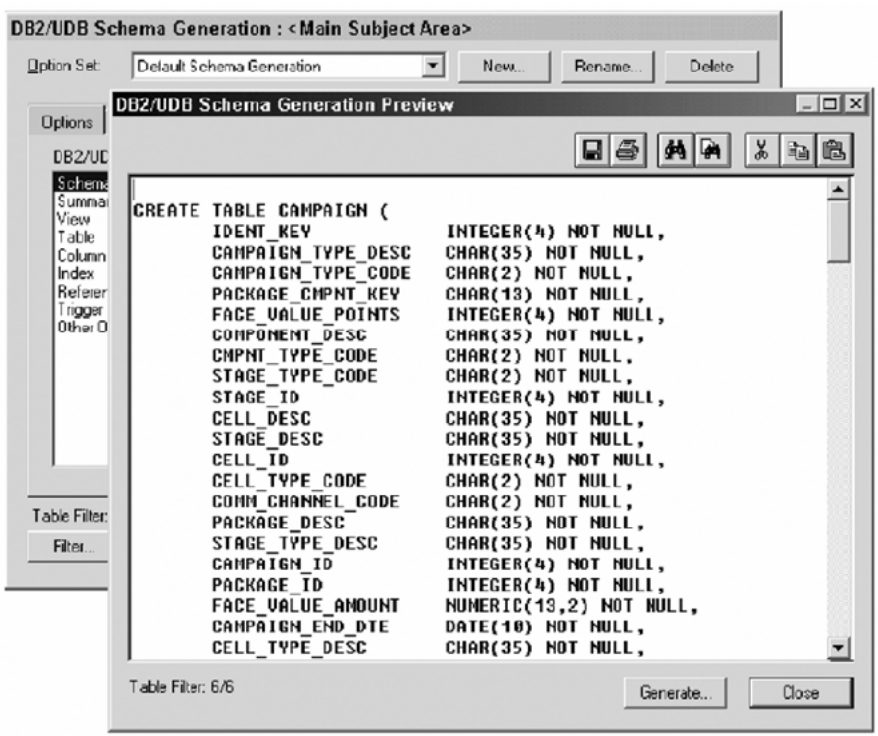


Figure 9.7 ERwin schema generation for a DB2 database (picture from IBM: http://www.redbooks.ibm.com/abstracts/redp3714.html?Open)

is massively useful for incremental application and database development, as well as for incremental maintenance.

9.5 Database Support

All of these products support a large array of database types. Certainly, all of the major database vendors are supported by each of these products (i.e., DB2 UDB, DB2 zOS, Informix IDS, Oracle, SQL Server), and a much larger set is supported through ODBC. However, what really matters most to the application developer is whether the database he or she is programming toward is directly supported by the CASE design tool. Database support is not equal between these products. Also, very significantly, each database product will have unique features of its own (such as reverse scan indexes, informational constraints, and so forth) which are not standard. One of the qualitative attributes of a good database design tool is whether it distinguishes and supports the unique extensions of individual database products. Each of the products has a strong history for doing so: in descending order, AllFusion ERwin Data Modeler, Sybase PowerDesigner, and Rational Data Architect. Notably, IBM's Rational Data Architect has a somewhat smaller range of supported databases than the other products, though it does support all the major database platforms. However, Rational Data Architect can be expected over time to have the tightest integration with the DB2 and Informix families, since all of these products are developed by IBM. Database designers

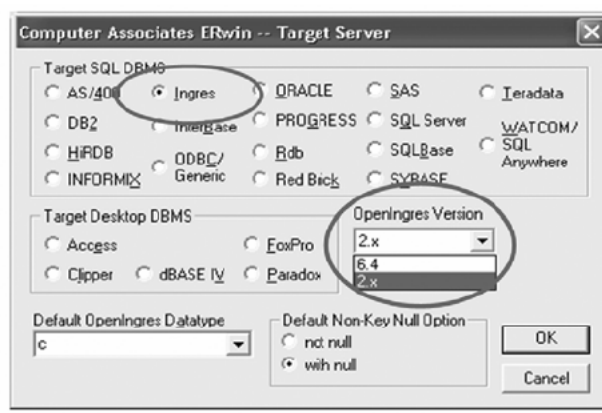


Figure 9.8 DBMS selection in AllFusion ERwin Data Modeler (picture from Computer Associates: http://iua.org.uk/conference/Autumn%202003/Ruth%20Wunderle.ppt#9)

are advised to investigate the level of support provided by a CASE tool for the database being developed toward, to ensure the level of support is adequate. Figure 9.8 shows an example of database server selection with AllFusion ERwin Data Modeler.

9.6 Collaborative Support

All three of these products are designed for collaborative development, so that multiple developers can work together to design portions of a database design, either supporting different applications or collaborating on the same portions. These collaboration features fall into two domains:

1. **Concurrency control**. This form of collaboration ensures that multiple designers do not modify the same component of the database design at the same time. This is comparable in software development terms to a source code control system.
2. **Merge and collaboration capabilities**. This form of collaboration enables designers to combine designs or merge their latest changes into a larger design project. These merging capabilities compare components between what is already logged into the

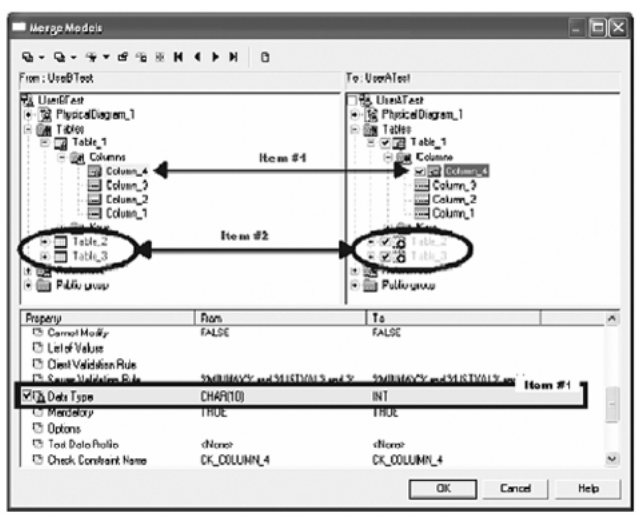


Figure 9.9 Merge process with PowerDesigner (courtesy of Sybase)

project system and what a designer wishes to add or modify. The CASE tools locate the conflicting changes and visually identify them for the designer, who can decide which changes should be kept, and which discarded in favor of the model currently defined in the project.

Figure 9.9 shows the Sybase PowerDesigner merge GUI, which identifies significant changes between the existing schema and the new schema being merged. In particular, notice how the merge tool has identified a change in **Table_1** Column_1, which has changed base types. The tool also found that **Table_2** and **Table_3**, which exist in the merging design, were not present in the base design. AllFusion ERwin Data Modeler and Rational Data Architect have similar capabilities for merging design changes.

9.7 Distributed Development

Distributed development has become a fact of life for large enterprise development teams, in which groups of developers collaborate from geographically diverse locations to develop a project. The phenomenon is not only true across floors of a building, or between sites in a city, but now across states, provinces, and even countries. In fact, outsourcing of software development has become a *tour de force*, with many analysts projecting that the average enterprise will ultimately outsource 60% of application work, shifting aspects of project development to locations with cheaper labor. As the META Group said in its September 16, 2004 Offshore Market Milieu report, *"With global resources costing one-third to one-fifth that of American employees—without accounting for hidden costs and having higher process discipline, offshore strategies now pervade North American IT organizations."*

Therefore, developers of database software working in a distributed collaborative environment need to consider the collaborative and distributed qualities of CASE tools for database design. The trend towards collaborative development and distributed development shows no sign of slowing; rather, it is on the increase. In the existing space, IBM's Rational MultiSite software, shown in Figure 9.10, allows the best administration across geographically diverse locations for replicating project software and data and subsequently merging the results. Rational MultiSite is a technology layered on top of Rational ClearCase and Rational Clear-

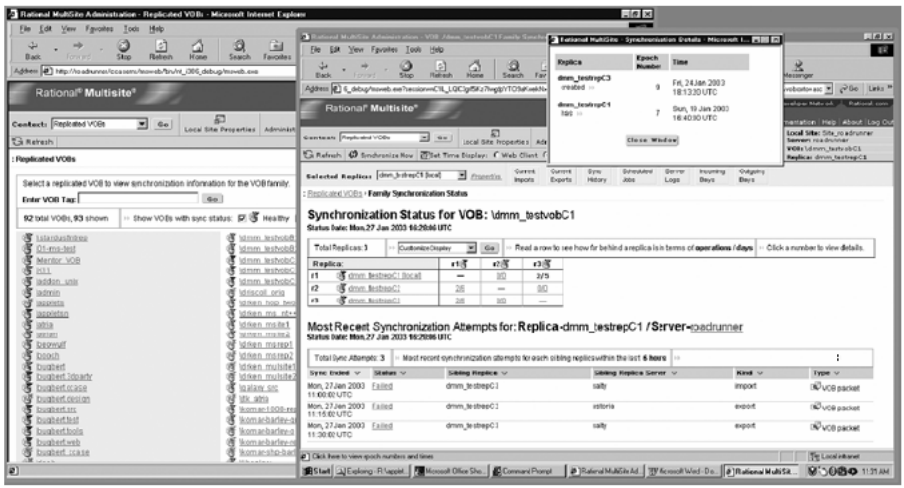


Figure 9.10 IBM's Rational MultiSite software for massively distributed software management (courtesy IBM Rational Division)

Quest (development and source code control products) to allow local replicas of Rational ClearCase and Rational ClearQuest repositories. Rational MultiSite also handles the automatic synchronization of the replicas. This is particularly useful for companies with distributed development who wish to have fast response times for their developers via access to local information, and such replication is often an absolute requirement for geographically distributed teams.

9.8 Application Life Cycle Tooling Integration

The best CASE tools for database design are integrated with a complete suite of application development tools that cover the software development life cycle. This allows the entire development team to work from an integrated tool platform, rather than the data modelers being off in their own world. Only the largest vendors offer this, and in fact true tooling integration across the development life cycle is somewhat rare. This solution is, in a very real way, the philosopher's stone of development infrastructure vendors. All the vendors who produce software development platforms have been working to develop this breadth during the past two decades. The challenge is elusive simply because it is hard to do well. The three companies we are discussing here all have

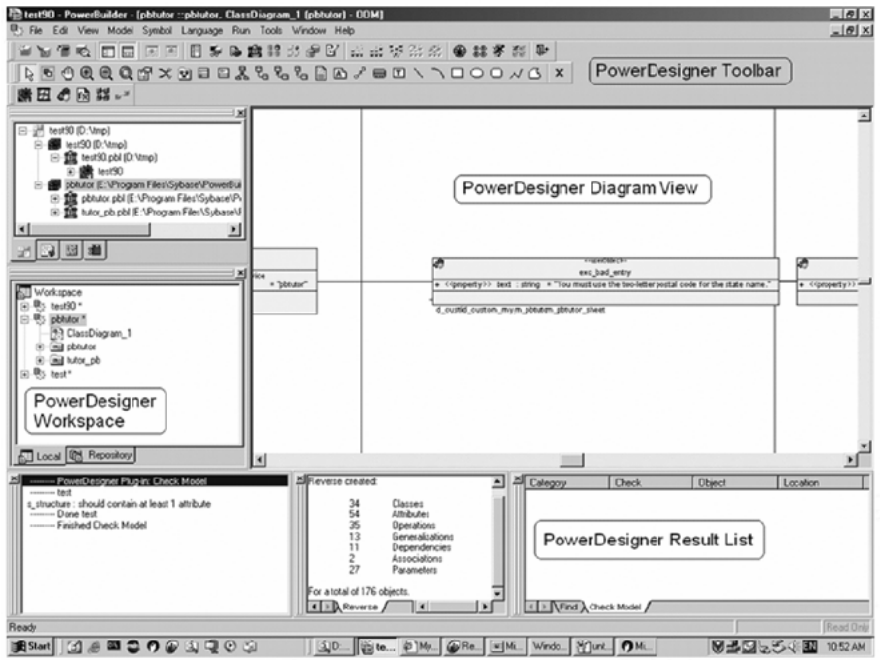


Figure 9.11 Sybase PowerDesigner plug-in to Sybase PowerBuilder (picture from http://www.pbugg.de/docs/1, Berndt Hambock)

broad offerings, and provide some degree of integration with their database design CASE technology.

For Computer Associates, the AllFusion brand is a family of development life cycles tools. It is intended to cover designing, building, deploying and managing eBusiness applications. Sybase also has a broad product suite, and their strength is in collaborative technology. From a design perspective, the ability to plug Sybase PowerDesigner into their popular Sybase Power Builder application development tooling is a very nice touch, as seen in Figure 9.11. The IBM tooling is clearly the broadest based, and their new IBM Software Development Platform, which is built heavily but not exclusively from their Rational products, covers everything from requirements building to portfolio management, source code control, architectural design constraints, automated testing, performance analysis, and cross site development. A representation of the IBM Software Development Platform is shown in Figure 9.12.

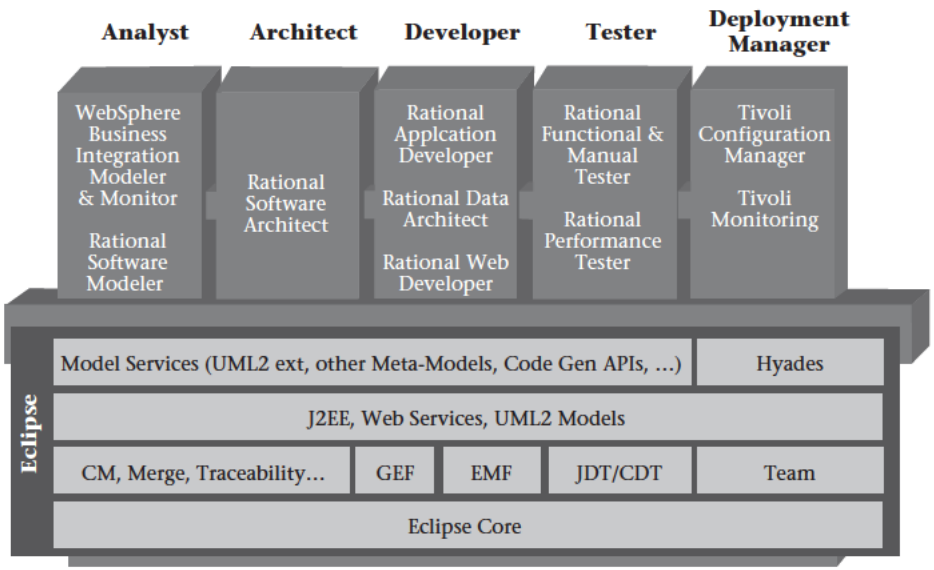


Figure 9.12 IBM Software Development Platform (courtesy IBM Rational Division)

9.9 Design Compliance Checking

With all complex designs, and particularly when multiple designers are involved, it can be very hard to maintain the integrity of the system design. The best software architects and designers grapple with this by defining design guidelines and rules. These are sometimes called "design patterns" and "anti-patterns." A design pattern is a design principle that is expected to be generally adhered to within the system design. Conversely, an anti-pattern is precisely the opposite. It represents flaws in the system design that can occur either through violation of the design patterns or through explicit definition of an anti-pattern. The enforcement of design patterns and anti-patterns is an emerging attribute of the best CASE tools for systems design in general, and database design in particular. Figure 9.13 shows an example of the interface used in the Rational Data Architect for compliance checking, which scans the system to enforce design patterns and check for anti-patterns. Some degree of support for design pattern and anti-pattern checking exists in AllFusion ERwin Data Modeler and Sybase PowerDesigner, as well. The compliance checking in IBM's Rational products is the most mature in

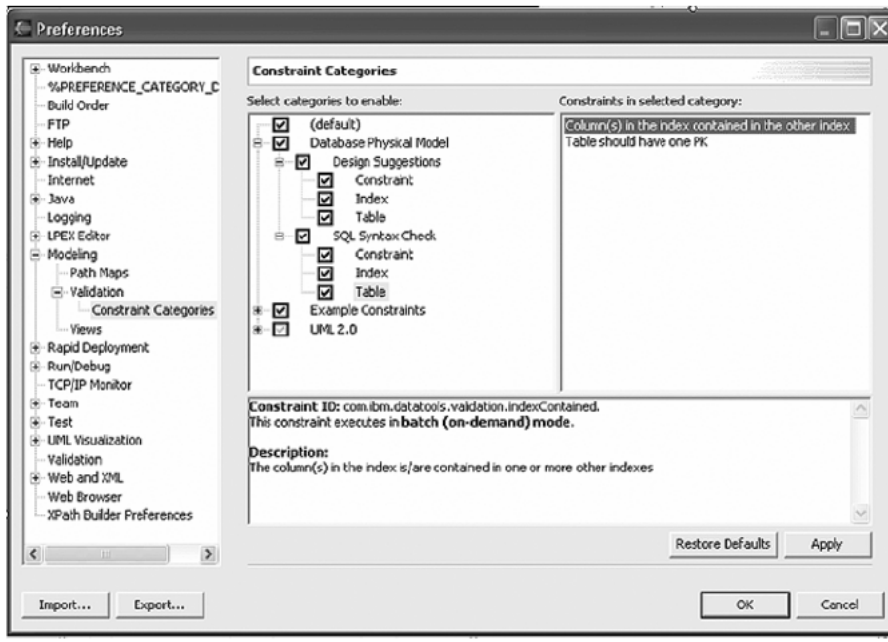


Figure 9.13 Modeling and database compliance checking (courtesy IBM Rational Division)

general, with the notion of design patterns and anti-patterns being a key philosophical point for the IBM's Rational family of products. Some examples of things these compliance checkers will scan for include:

- Complete round trip engineering
- Design and normalization
  - — Discover 1st, 2nd, and 3rd normalization
- Index and storage
  - — Check for excessive indexing
- Naming standards
- Security compliance
- Sarbanes-Oxley compliance
  - — Check for valid data model and rules
- Model syntax checks

9.10 Reporting

Reporting capabilities are a very important augmentation of the design capabilities in CASE tools for database design. These reports allow you to examine your ER and UML modeling and database schema in both graphical and textual formats. The Sybase products have a superb reputation for reporting features; their products enable you to generate reports in common formats like Microsoft Word. Reporting can include both the modeling and the annotations that the designer has added. It can cover physical data models, conceptual data models, object-oriented

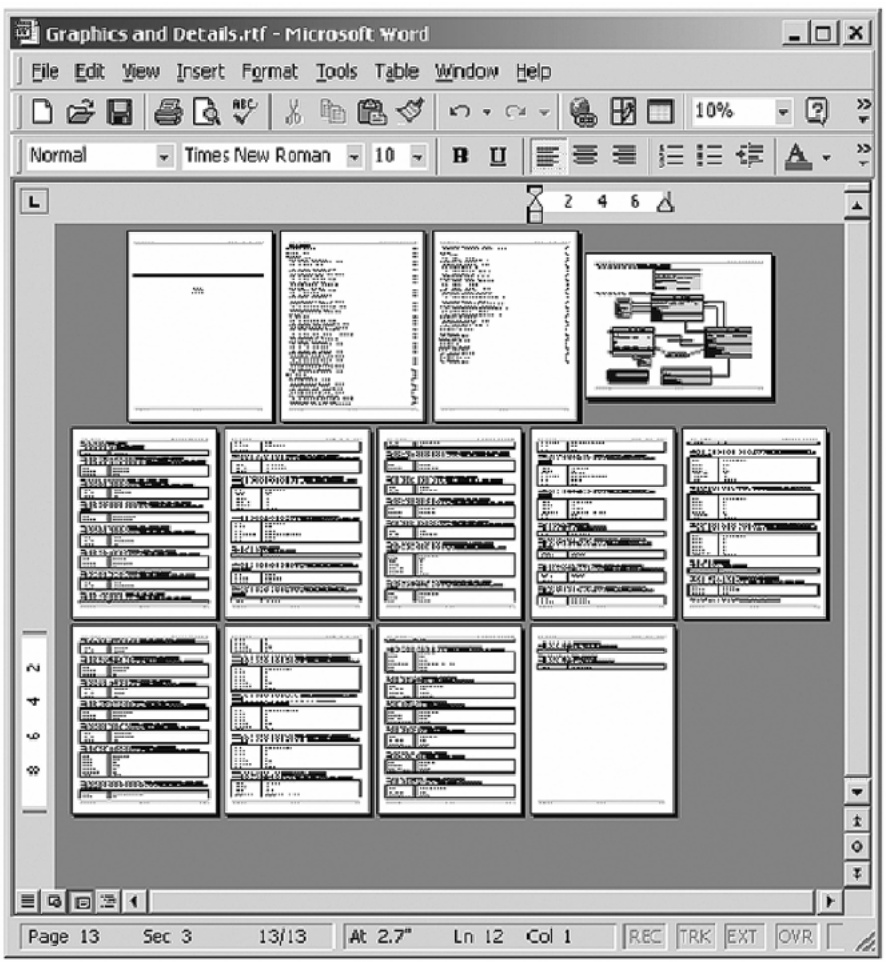


Figure 9.14 Reporting features with Sybase PowerDesigner (courtesy Sybase)

models (UML), business process models, and semi-structured data using eXtensible Markup Language (XML). Notice in Figure 9.14 how the fourth page, which contains graphics, can be oriented in landscape mode, while the remaining pages are kept in portrait mode. AllFusion ERwin and Rational Data Architect also provide rich reporting features, though Sybase PowerDesigner has the richest capabilities.

9.11 Modeling a Data Warehouse

In Chapter 8 we discussed the unique design considerations required for data warehousing and decision support. Typically, warehouses are designed to support complex queries that provide analytic analysis of your data. As such, they exploit different schema topology models, such as star schema and horizontal partitioning. They typically exploit data views and materialized data views, data aggregation, and multidimensional modeling far more extensively than other operational and transactional databases.

Traditionally, warehouses have been populated with data that is extracted and transformed from other operational databases. However, more and more companies are moving to consolidate system resources and provide real-time analytics by either feeding warehouses data in near-real-time (i.e., with a few minutes latency) or entirely merging their transactional data stores with their analytic warehouses into a single server or cluster. These trends are known as "active data warehousing," and pose even more complex design challenges. There is a vast need for CASE tooling in this space.

Sybase offers a CASE tool known as Sybase Industry Warehouse Studio (IWS). Sybase IWS is really a set of industry-specific, prepackaged warehouses that require some limited customization. Sybase IWS tooling provides a set of wizards for designing star schemas, dimensional tables, denormalization, summarization, and partitioning; as usual, the Sybase tools are strong on reporting facilities.

The industry domains covered by ISW are fairly reasonable—they include IWS for Media, IWS for Healthcare, IWS for Banking, IWS for Capital Markets, IWS for Life Insurance, IWS for Telco, IWS for Credit Cards, IWS fro P&C Insurance, and IWS for CRA.

IBM's DB2 Cube Views (shown in Figure 9.15) provides OLAP and multidimensional modeling. DB2 Cube Views allows you to create metadata objects to dimensionally model OLAP structures and relational data. The graphical interface allows you to create, manipulate, import, or export cube models, cubes, and other metadata objects.

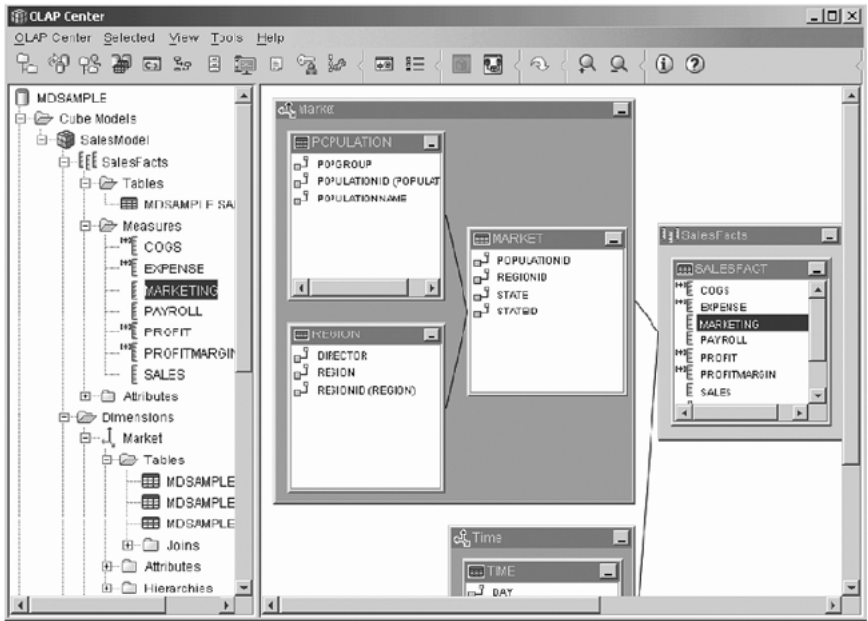


Figure 9.15 DB2 Cube Views interface (courtesy IBM Rational Division)

Sybase IWS uses standard database design constructs that port to many database systems, such as DB2 UDB, Oracle, Microsoft SQL Server, Sybase Adaptive Server Enterprise, and Sybase IQ. In contrast, IBM's DB2 Cube Views is designed specifically to exploit DB2 UDB. The advantage of DB2 Cube View is that it can exploit product-specific capabilities in the DB2 database that may not be generally available in other databases. Some examples of this include materialized query tables (precomputed aggregates and cubes), multidimensional clustering, triggers, functional dependencies, shared-nothing partitioning, and replicated MQTs. Sybase IWS dependence on the lowest common denominator database feature provides flexibility when selecting the database server but may prove extremely limiting for even moderately sized marts and warehouses (i.e., larger than 100 GB), where advanced access and design features become critical.

To summarize and contrast, Sybase offers portable warehouse designs that require minimal customization and are useful for smaller systems, and DB2 Cube View provides significantly richer and more powerful capabilities, which fit larger systems, require more customization, and necessitate DB2 UDB as the database server.

AllFusion ERwin Data Modeler has basic support to model OLAP and multidimensional databases, but does not have the same richness of tooling and wizards that the other companies offer to actually substantially simplify the design process of these complex systems.

9.12 Semi-Structured Data, XML

XML (eXtensible Markup Language) is a data model consisting of nodes of several types linked together with ordered parent/child relationships to form a hierarchy. One representation of that data model is textual—there are others that are not text! XML has increasingly become a data format of choice for data sharing between systems. As a result, increasing volumes of XML data are being generated.

While XML data has some structure it is not a fully-structured format, such as the table definitions that come from a fully-structured modeling using ER with IE or UML. XML is known in the industry as a semi-structured format: It lacks the strict adherence of schema that structured data schemas have, yet it has some degree of structure which distinguishes it from completely unstructured data, such as image and video data.

Standards are forming around XML to allow it to be used for database style design and query access. The dominant standards are XML Schema and XML Query (also known as XQuery). Also worth noting is OMG XMI standard, which defines a standard protocol for defining a structured format for XML interchange, based on an object model. Primarily for interfacing reasons, UML tools such as MagicDraw have taken XMI seriously and have therefore become the preferred alternatives in the open source space.

XML data is text-based, and self-describing (meaning that XML described the type of each data point, and defines its own schema). XML

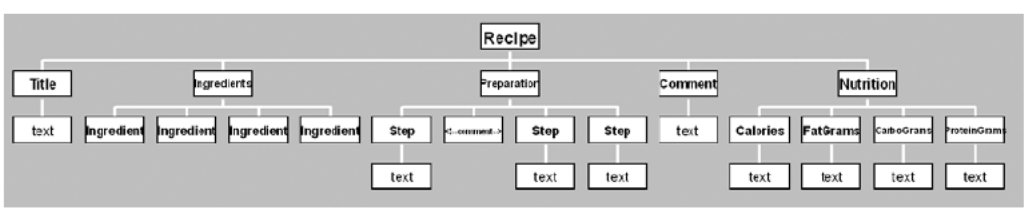


Figure 9.16 An XML schema for a recipe

has become popular for Internet-based data exchange based on these qualities as well as being "well-formed." Well-formed is a computer science term, implying XML's grammar is unambiguous through the use of mandated structure that guarantees terms are explicitly prefixed and closed. Figure 9.16 shows the conceptual design of a semi-structured document type named "recipe." Figure 9.17 shows an XML document for a hot dog recipe. Notice that the file is completely textual.

IBM Rational Data Architect and Sybase PowerDesigner have taken the lead in being early adopters of XML data modeling CASE tools. Both products support the modeling of semi-structured data through XML and provide graphical tooling for modeling XML hierarchies.

```
<?xml version="1.0" encoding="UTF-8" ?>
<Recipe TimeToPrepare="5" CookMethod="Grill" Difficulty="Easy" Serves="1"
Category="Entrees">
 <Title>Hot Dog with Onions</Title>
 <Ingredients>
  <Ingredient Name="Hot Dog" Amount="1" />
  <Ingredient Name="Hot Dog Bun" Amount="1" />
  <Ingredient Name="Sliced Onion" Amount="1" Unit="tablespoon" />
  <Ingredient Name="Mustard" Amount="2" Unit="teaspoon" />
  <Ingredient Name="Relish" Amount="2" Unit="teaspoon" />
 </Ingredients>
 <Preparation>
  <Step>Preheat Grill to 350&#176; F. </Step>
  <!-- &#176; is the degree symbol -->
  <Step>Grill Hot dog for 5 Minutes, Turning frequently to avoid burning</Step>
  <Step>Put Hot Dog on bun, add mustard, onion, and relish</Step>
 </Preparation>
 <Comment>Some like to use spicy mustard,or replace mustard with
catsup</Comment>
 <Nutrition>
  <Calories>300</Calories>
  <FatGrams>18.5</FatGrams>
  <CarboGrams>12</CarboGrams>
  <ProteinGrams>9.5</ProteinGrams>
 </Nutrition>
</Recipe>
```

Figure 9.17 An XML document for a hot dog

9.13 Summary

There are several good CASE tools available for computer-assisted database design. This chapter has touched on some of the features for three of the leading products: IBM Rational Data Architect, Computer Associates AllFusion ERwin Data Modeler, and Sybase PowerDesigner. Each provides powerful capabilities to assist in developing ER models and transforming those models to logical database designs and physical implementations. All of these products support a wide range of database vendors, including DB2 UDB, DB2 zOS, Informix Data Server (IDS), Oracle, SQL Server, and many others through ODBC support. Each product has different advantages and strengths. The drawbacks a product may have now are certain to be improved over time, so discussing the relative merits of each product in a book can be somewhat of an injustice to a product that will deliver improved capabilities in the near future.

At the time of authoring this text, Computer Associate's AllFusion ERwin Data Modeler had advantages as a mature product with vast database support. The AllFusion products don't have the advanced complex feature support for XML and warehousing/analytics, but what they do support they do well. Sybase PowerDesigner sets itself apart for superior reporting capabilities. IBM's Rational Data Architect has the best integration with a broad software application development suite of tooling, and the most mature use of UML. Both the Sybase and IBM tools are blazing new ground in their early support for XML semi structured data and for CASE tools for warehousing and OLAP. The best products provide the highest level of integration into a larger software development environment for large-scale collaborative, and possible geographically diverse, development. These CASE tools can dramatically reduce the time, cost, and complexity of developing, deploying, and maintaining a database design.

9.14 Literature Summary

Current logical database design tools can be found in manufacturer Web sites [Database Answers, IBM Rational Software, Computer Associates, Sybase PowerDesigner, Directroy of Data Modeling Resources, Objects by Design, Understanding relational databases: referential integrity, and Widom].

Appendix
The Basics of SQL

Structured Query Language, or SQL, is the ISO-ANSI standard data definition and manipulation language for relational database management systems. Individual relational database systems use slightly different dialects of SQL syntax and naming rules, and these differences can be seen in the SQL user guides for those systems. In this text, as we explore each step of the logical and physical design portion of the database life cycle, many examples of database table creation and manipulation make use of SQL syntax.

Basic SQL use can be learned quickly and easily by reading this appendix. The more advanced features, such as statistical analysis and presentation of data, require additional study and are beyond the reach of the typical nonprogrammer. However, the DBA can create SQL views to help nonprogrammers set up repetitive queries; other languages, such as forms, are being commercially sold for nonprogrammers. For the advanced database programmer, embedded SQL (in C programs, for instance) is widely available for the most complex database applications, which need the power of procedural languages.

This appendix introduces the reader to the basic constructs for the SQL-99 (and SQL-92) database definition, queries, and updates through a sequence of examples with some explanatory text. We start with a definition of SQL terminology for data types and operators. This is followed by an explanation of the data definition language (DDL) constructs using the "create table" commands, including a definition of the various

types of integrity constraints, such as foreign keys and referential integrity. Finally, we take a detailed look at the SQL-99 data manipulation language (DML) features through a series of simple and then more complex practical examples of database queries and updates.

The specific features of SQL, as implemented by the major vendors IBM, Oracle, and Microsoft, can be found in the references at the end of this appendix.

A.1 SQL Names and Operators

This section gives the basic rules for SQL-99 (and SQL-92) data types and operators.

- *SQL names:* Although these have no particular restrictions, some vendor-specific versions of SQL do have some restrictions. For example, in Oracle, names of tables and columns (attributes) can be up to 30 characters long, must begin with a letter, and can include the symbols (a-z, 0-9,_,$,#). Names should not duplicate reserved words or names for other objects (attributes, tables, views, indexes) in the database.
- *Data types for attributes:* character, character varying, numeric, decimal, integer, smallint, float, double precision, real, bit, bit varying, date, time, timestamp, interval.
- *Logical operators:* and, or, not, ().
- *Comparison operators:* =, <>, <, <=, >, >=, (), in, any, some, all, between, not between, is null, is not null, like.
- *Set operators:*
  - union: combines queries to display any row in each subquery
  - intersect: combines queries to display distinct rows common to all subqueries
  - except: combines queries to return all distinct rows returned by the first query, but not the second (this is "minus" or "difference" in some versions of SQL)
- *Set functions:* count, sum, min, max, avg.
- *Advanced value expressions:* CASE, CAST, row value constructors. CASE is similar to CASE expressions in programming languages, in which a select command needs to produce different results when there are different values of the search condition. The CAST

expression allows you to convert data of one type to a different type, subject to some restrictions. Row value constructors allow you to set up multiple column value comparisons with a much simpler expression than is normally required in SQL (see Melton and Simon [1993] for detailed examples).

A.2 Data Definition Language (DDL)

The basic definitions for SQL objects (tables and views) are:

- *create table:* defines a table and all its attributes
- *alter table:* add new columns, drop columns, or modifies existing columns in a table
- *drop table:* deletes an existing table
- *create view, drop view:* defines/deletes a database view (see Section A.3.4)

Some versions of SQL also have *create index/drop index*, which defines/deletes an index on a particular attribute or composite of several attributes in a particular table.

The following table creation examples are based on a simple database of three tables: **customer**, **item**, and **order**. (Note that we put table names in boldface throughout the book for readability.)

```
create table customer
       (cust_num       numeric,
       cust_name       char(20),
       address         varchar(256),
       credit_level    numeric,
       check (credit_level >= 1000),
       primary key (cust_num));
```

Note that the attribute cust_num could be defined as "numeric not null unique" instead of explicity defined as the primary key, since they both have the same meaning. However, it would be redundant to have both forms in the same table definition. The check rule is an important integrity constraint that tells SQL to automatically test each insertion of credit_level value for something greater than or equal to 1000. If not, an error message should be displayed.

```
create table item
       (item_num      numeric,
       item_name      char(20),
       price          numeric,
       weight         numeric,
       primary key (item_num));

create table order
       (ord_num       char(15),
       cust_num       numeric  not null,
       item_num       numeric  not null,
       quantity       numeric,
       total_cost     numeric,
       primary key (ord_num),
       foreign key (cust_num) references customer
            on delete no action on update cascade,
       foreign key (item_num) references item
            on delete no action on update cascade);
```

SQL, while allowing the above format for primary key and foreign key, recommends a more detailed format, shown below, for table **order**:

```
constraint pk_constr primary key (ord_num),
constraint fk_constr1 foreign key
      (cust_num) references customer
      (cust_num) on delete no action on update cascade,
constraint fk_constr2 foreign key
      (item_num) references item (item_num)
      on delete no action on update cascade);
```

in which pk_constr is a primary key constraint name, and fk_constr1 and fk_constr2 are foreign key constraint names. The word "constraint" is a keyword, and the object in parentheses after the table name is the name of the primary key in that table referenced by the foreign key.

The following constraints are common for attributes defined in the SQL *create table* commands:

- *Not null:* a constraint that specifies that an attribute must have a nonnull value.
- *Unique:* a constraint that specifies that the attribute is a candidate key; that is, that it has a unique value for every row in the table. Every attribute that is a candidate key must also have the constraint not null. The constraint unique is also used as a clause to

designate composite candidate keys that are not the primary key. This is particularly useful when transforming ternary relationships to SQL.

- *Primary key:* the primary key is a set of one or more attributes that, when taken collectively, enables us to uniquely identify an entity or table. The set of attributes should not be reducible (see Section 6.1.2). The designation primary key for an attribute implies that the attribute must be not null and unique, but the SQL keywords NOT NULL and UNIQUE are redundant for any attribute that is part of a primary key, and need not be specified in the create table command.
- *Foreign key:* the referential integrity constraint specifies that a foreign key in a referencing table column must match an existing primary key in the referenced table. The references clause specifies the name of the referenced table. An attribute may be both a primary key and a foreign key, particularly in relationship tables formed from many-to-many binary relationships or from *n*-ary relationships.

Foreign key constraints are defined for deleting a row on the referenced table and for updating the primary key of the referenced table. The referential trigger actions for delete and update are similar:

- *on delete cascade:* the delete operation on the referenced table "cascades" to all matching foreign keys.
- *on delete set null:* foreign keys are set to null when they match the primary key of a deleted row in the referenced table. Each foreign key must be able to accept null values for this operation to apply.
- *on delete set default:* foreign keys are set to a default value when they match the primary key of the deleted row(s) in the reference table. Legal default values include a literal value, "user," "system user," or "no action."
- *on update cascade:* the update operation on the primary key(s) in the referenced table "cascades" to all matching foreign keys.
- *on update set null:* foreign keys are set to null when they match the old primary key value of an updated row in the referenced table. Each foreign key must be able to accept null values for this operation to apply.
- *on update set default:* foreign keys are set to a default value when they match the primary key of an updated row in the reference

table. Legal default values include a literal value, "user," "system user," or "no action."

The *cascade* option is generally applicable when either the mandatory existence constraint or the ID dependency constraint is specified in the ER diagram for the referenced table, and either *set null* or *set default* is applicable when optional existence is specified in the ER diagram for the referenced table (see Chapters 2 and 5).

Some systems, such as DB2, have an additional option on delete or update, called *restricted*. Delete restricted means that the referenced table rows are deleted only if there are no matching foreign key values in the referencing table. Similarly, *update restricted* means that the referenced table rows (primary keys) are updated only if there are no matching foreign key values in the referencing table.

Various column and table constraints can be specified as *deferrable* (the default is *not deferrable*), which means that the DBMS will defer checking this constraint until you commit the transaction. Often this is required for mutual constraint checking.

The following examples illustrate the *alter table* and *drop table* commands. The first *alter table* command modifies the cust_name data type from char(20) in the original definition to varchar(256). The second and third *alter table* commands add and drop a column, respectively. The *add column* option specifies the data type of the new column.

```
alter table customer
modify (cust_name varchar(256));

alter table customer
add column cust_credit_limit numeric;

alter table customer
drop column credit_level;

drop table customer;
```

A.3 Data Manipulation Language (DML)

Data manipulation language commands are used for queries, updates, and the definition of views. These concepts are presented through a series of annotated examples, from simple to moderately complex.

A.3.1 SQL Select Command

The SQL select command is the basis for all database queries. The following series of examples illustrates the syntax and semantics of the select command for the most frequent types of queries in everyday business applications. To illustrate each of the commands, we assume the following set of data in the database tables:

customer table

cust num	cust name	address	credit level
001	Kirk	Enterprise	10
002	Spock	Enterprise	9
003	Scotty	Enterprise	8
004	Bones	Enterprise	8
005	Gorn	PlanetoidArena	1
006	Khan	CetiAlphaFive	2
007	Uhura	Enterprise	7
008	Chekov	Enterprise	6
009	Sulu	Enterprise	6

item table

item num	item name	price	weight
125	phaser	350	2
137	beam	1500	250
143	shield	4500	3000
175	fusionMissile	2750	500
211	captainsLog	50	2
234	starShip	25000	15000
356	sensor	245	15
368	intercom	1200	75
399	medicalKit	75	3

order table

ord num	cust num	item num	quantity	total cost
10012	005	125	2	700
10023	006	175	20	55000
10042	003	137	3	4500
10058	001	211	1	50
10232	007	368	1	1200
10266	002	356	50	12250
10371	004	399	10	750
11070	009	143	1	4500
11593	008	125	2	700
11775	006	125	3	1050
12001	001	234	1	25000

Basic Commands

1. Display the entire **customer** table. The asterisk (*) denotes that all records from this table are to be read and displayed.

```
select *
from customer;
```

   This results in a display of the complete **customer** table (as shown above).

2. Display customer name, customer number, and credit level for all customers on the Enterprise who have a credit level greater than 7. Order by ascending sequence of customer name (the order-by options are asc, desc). Note that the first selection condition is specified in the *where* clause and succeeding selection conditions are specified by *and* clauses. Character type data and other non-numeric data are placed inside single quotes, but numeric data is given without quotes. Note that useful column names can be created by using formatting commands (which are not shown here).

```
select cust_name, cust_num, credit_level
from customer
where address = 'Enterprise'
and credit_level > 7
order by cust_name asc;

customer name    customer number    credit level
---------------------------------------------
Bones                 004                 8
Kirk                  001                10
Scotty                003                 8
Spock                 002                 9
```

3. Display all customer and order item information (all columns), but omit customers with a credit level greater than 6. In this query, the *from* clause shows the definition of abbreviations **c** and **o** for tables **customer** and **order**, respectively. The abbreviations can be used anywhere in the query to denote their respective table names. This example also illustrates a *join* between tables **customer** and **order**, using the common attribute name cust_num as shown in the *where* clause. The join finds matching cust_num values from the two tables and displays all the data

from the matching rows, except where the credit number is 7 or above, and ordered by customer number.

```
select c.*, o.*
from customer as c, order as o
where c.cust_num = o.cust_num
and c.credit_level < 7
order by cust_no asc;
```

```
cust. no. cust. name   address         credit level  order no.  item no.  qty   total  cost
-------------------------------------------------------------------------------------------
005       Gorn         PlanetoidArena  1             10012      125       2     700
006       Khan         CetiAlphaFive   2             11775      125       3     1050
006       Khan         CetiAlphaFive   2             10023      175       20    55000
008       Chekov       Enterprise      6             11593      125       2     700
009       Sulu         Enterprise      6             11070      143       1     4500
```

Union and Intersection Commands

1. Which items were ordered by customer 002 or customer 007? This query can be answered in two ways, one with a set operator (union) and the other with a logical operator (or).

```
select item_num, item_name, cust_num, cust_name
from order
where cust_num = 002
union
select item_num, item_name, cust_num, cust_name
from order
where cust_num = 007;

select item_num, item_name, cust_num, cust_name
from order
where (cust_num = 002 or cust_num = 007);
```

```
item number   item name   customer no.   customer name
-----------------------------------------------------
356           sensor      002            Spock
368           intercom    007            Uhura
```

2. Which items are ordered by both customers 005 and 006? All the rows in table **order** that have customer 005 are selected and compared to the rows in **order** that have customer 006. Rows from each set are compared with all rows from the other set, and those

that have matching item numbers have the item numbers displayed.

```
select item_num, item_name, cust_num, cust_name
from order
where cust_num = 005
intersect
select item_num, item_name, cust_num, cust_name
from order
where cust_num = 006;

item number   item name   customer no   customer name
-----------------------------------------------------
125           phaser      005           Gorn
125           phaser      006           Khan
```

Aggregate Functions

1. Display the total number of orders. This query uses the SQL function *count* to count the number of rows in table **order**.

```
select count(*)
from order;

count(order)
--------------------
11
```

2. Display the total number of customers actually placing orders for items. This is a variation of the count function and specifies that only the distinct number of customers is to be counted. The *distinct* modifier is required because duplicate values of customer numbers are likely to be found, since a customer can order many items and will appear in many rows of table order.

```
select count (distinct cust_num)
from order;

distinct count(order)
-------------------------
9
```

3. Display the maximum quantity of an order of item number 125. The SQL *maximum* function is used to search the table **order**, select rows where the item number is 125, and display the maximum value of quantity from the rows selected.

```
select max (quantity)
from order
where item_num = 125;

max(quantity)
-----------------
3
```

4. For each type of item ordered, display the item number and total order quantity. Note that item_num and item_name in the select line must be in a *group by* clause. In SQL, any attribute to be displayed in the result of the select command must be included in a *group by* clause when the result of an SQL function is also to be displayed. The *group by* clause results in a display of the aggregate sum of quantity values for each value of item_num and item_name. The aggregate sums will be taken over all rows with the same value of item_num.

```
select item_num, item_name, sum(quantity)
from order
group by item_num, item_name;

item number    item name     sum(quantity)
-----------------------------------------
125            phaser          7
137            beam            3
143            shield          1
175            fusionMissile   20
211            captainsLog     1
234            starShip        1
356            sensor          50
368            intercom        1
399            medicalKit      10
```

5. Display item numbers for all items ordered more than once. This query requires the use of the *group by* and *having* clauses to display data that is based on a count of rows from table **order** having the same value for attribute item_num.

```
select item_num, item_name
from order
group by item_num, item_name
having count(*) >1;

item number     item name
----------------------------
125             phaser
```

Joins and Subqueries

1. Display customer names of the customers who order item number 125. This query requires a *join* (*equijoin*) of tables **customer** and **order** to match customer names with item number 125. Including the item_num column in the output verifies that you have selected the item number you want. Note that the default ordering of output is typically ascending by the first column.

```
select c.cust_name, o.item_num
from customer as c, order as o
where c.cust_num = o.cust_num
and item_num = 125;


customer name       item number
----------------    ---------------
Chekov              125
Gorn                125
Khan                125
```

This query can be equivalently performed with a *subquery* (sometimes called a nested subquery) with the following format. The *select* command inside the parentheses is a nested subquery and is executed first, resulting in a set of values for customer number (cust_num) selected from the **order** table. Each of those values is compared with cust_num values from the **customer** table, and matching values result in the display of customer name from the matching row in the **customer** table. This is effectively a join between tables **customer** and **order** with the selection condition of item number 125.

```
select cust_name, order_num
from customer
where cust_num in
      (select cust_num
      from order
      where item_num = 125);
```

2. Display names of customers who order at least one item priced over 1000. This query requires a three-level nested subquery format. Note that the phrases *in*, = *some*, and = *any* in the *where* clauses are often used as equivalent comparison operators; see Melton and Simon [1993].

```
select c.cust_name
from customer as c
where c.cust_num in
     (select o.cust_num
     from order as o
     where o.item_num = any
     (select i.item_num
     from item as i
     where i.price > 1000));
```

```
customer name
------------------
Khan
Kirk
Scotty
Sulu
Uhura
```

3. Which customers have not ordered any item priced over 100? Note that one can equivalently use *not in* instead of *not any*. The query first selects the customer numbers from all rows from the join of tables **order** and **item** where the item price is over 100. Then it selects rows from table **customer** where the customer number does not match any of the customers selected in the subquery, and displays the customer names.

```
select c.cust_name
from customer as c
where c.cust_num not any
```

```
      (select o.cust_num
      from order as o, item as i
      where o.item_num = i.item_num
      and i.price > 100);

customer name
------------------
Bones
```

4. Which customers have only ordered items weighing more than 1000? This is an example of the universal quantifier *all*. First the subquery selects all rows from table **item** where the item weight is over 1000. Then it selects rows from table **order** where all rows with a given item number match at least one row in the set selected in the subquery. Any rows in **order** satisfying this condition are joined with the **customer** table, and the customer name is displayed as the final result.

```
select c.cust_name
from customer as c, order as o
where c.cust_num = o.cust_num
and o.item_num = all
      (select i.item_num
      from item as i
      where i.weight > 1000);

customer name
------------------
Sulu
```

Note that Kirk has ordered one item weighing over 1000 (starShip), but he has also ordered an item weighing under 1000 (captainsLog), so his name does not get displayed.

A.3.2 SQL Update Commands

The following SQL update commands relate to our continuing example and illustrate typical usage of insertion, deletion, and update of selected rows in tables.

This command adds one more customer (klingon) to the **customer** table:

```
insert into customer
values (010,'klingon','rogueShip',4);
```

This command deletes all customers with credit levels less than 2:

```
delete from customer
where credit_level < 2;
```

This command modifies the credit level of any customer with level 6 to level 7:

```
update customer
set credit_level = 7
where credit_level = 6;
```

A.3.3 Referential Integrity

The following update to the **item** table resets the value of item_num for a particular item, but because item_num is a foreign key in the **order** table, SQL must maintain referential integrity by triggering the execution sequence named by the foreign key constraint *on update cascade* in the definition of the **order** table (see Section A2). This means that, in addition to updating a row in the **item** table, SQL will search the **order** table for values of item_num equal to 368 and reset each item_num value to 370.

```
update item
set item_num = 370
where item_num = 368;
```

If this update had been a *delete* instead, such as the following:

```
delete from item
where item_num = 368;
```

then the referential integrity trigger would have caused the additional execution of the foreign key constraint *on delete set default* in order (as defined in Section A.2), which finds every row in order with item_num equal to 368 and takes the action set up in the default. A typical action for this type of database might be to set item_num to either null or a predefined literal value to denote that the particular item has been

deleted; this would then be a signal to the system that the customer needs to be contacted to change the order. Of course, the system would have to be set up in advance to check for these values periodically.

A.3.4 SQL Views

A view in SQL is a named, derived (virtual) table that derives its data from base tables, the actual tables defined by the *create table* command. View definitions can be stored in the database, but the views (derived tables) themselves are not stored; they are derived at execution time when the view is invoked as a query using the SQL *select* command. The person who queries the view treats the view as if it were an actual (stored) table, unaware of the difference between the view and the base table.

Views are useful in several ways. First, they allow complex queries to be set up in advance in a view, and the novice SQL user is only required to make a simple query on the view. This simple query invokes the more complex query defined by the view. Thus, nonprogrammers are allowed to use the full power of SQL without having to create complex queries. Second, views provide greater security for a database, because the DBA can assign different views of the data to different users and control what any individual user sees in the database. Third, views provide a greater sense of data independence; that is, even though the base tables may be altered by adding, deleting, or modifying columns, the view query may not need to be changed. While view definition may need to be changed, that is the job of the DBA, not the person querying the view.

Views may be defined hierarchically; that is, a view definition may contain another view name as well as base table names. This enables some views to become quite complex.

In the following example, we create a view called "orders" that shows which items have been ordered by each customer and how many. The first line of the view definition specifies the view name and (in parentheses) lists the attributes of that view. The view attributes must correlate exactly with the attributes defined in the select statement in the second line of the view definition:

```
create view orders (customer_name, item_name,
     quantity) as
          select c.cust_name, i.item_name, o.quantity
          from customer as c, item as i, order as o
          where c.cust_num = o.cust_num
          and o.item_num = i.item_num;
```

The *create view* command creates the view definition, which defines two joins among three base tables **customer**, **item**, and **order**; and SQL stores the definition to be executed later when invoked by a query. The following query selects all the data from the view "orders." This query causes SQL to execute the select command given in the preceding view definition, producing a tabular result with the column headings for customer_name, item_name, and quantity.

```
select *
from orders;
```

Usually, views are not allowed to be updated, because the updates would have to be made to the base tables that make up the definition of the view. When a view is created from a single table, the view update is usually unambiguous, but when a view is created from the joins of multiple tables, the base table updates are very often ambiguous and may have undesirable side effects. Each relational system has its own rules about when views can and cannot be updated.

A.4 References

Bulger B., Greenspan, J., and Wall, D. *MySQL/PHP Database Applications*, 2nd ed., Wiley, 2004.

Gennick, J. *Oracle SQL*Plus: The Definitive Guide*, O'Reilly, 1999.

Gennick, J. *SQL Pocket Guide*, O'Reilly, 2004.

Melton, J., and Simon, A. R. *Understanding The New SQL: A Complete Guide*, Morgan Kaufmann, 1993.

Mullins, C. S. *DB2 Developer's Guide*, 5th ed., Sams Publishing, 2004.

Neilson, P. *Microsoft SQL Server Bible*, Wiley, 2003.

van der Lans, R. *Introduction to SQL: Mastering the Relational Database Lannguage*, 3rd ed., Addison-Wesley, 2000.

Glossary

activity diagram (UML)—A process workflow model (diagram) showing the flow from one activity to the next.

aggregation—A special type of abstraction relationship that defines a higher-level entity that is an aggregate of several lower-level entities; a "part-of" type relationship. For example, a bicycle entity would be an aggregate of wheel, handlebar, and seat entities.

association—A relationship between classes (in UML); associations can be binary, *n*-ary, reflexive, or qualified.

attribute—A primitive data element that provides descriptive detail about an entity; a data field or data item in a record. For example, lastname would be an attribute for the entity customer. Attributes may also be used as descriptive elements for certain relationships among entities.

automatic summary table (AST)—Materialized (summary) views or aggregates of data saved by OLAP for future use to reduce query time.

binary recursive relationship—A relationship between one occurrence of an entity with another occurrence of the same entity.

binary relationship—A relationship between occurrences of two entities.

Boyce Codd normal form (BCNF)—A table is in Boyce Codd normal form if and only if for every functional dependency X->A, where

X and A are either simple or composite attributes (data items), X must be a superkey in that table. This is a strong form of 3NF and is the basis for most practical normalization methodologies.

candidate key—Any subset of the attributes (data items) in a superkey that is also a superkey and is not reducible to another superkey.

CASE tool—Computer-aided software engineering tool, a software design tool to assist in the logical design of large or complex databases. examples include ERwin Data Modeler and Rational Rose using UML.

class—A concept in a real-world system, represented by a noun in UML; similar to an entity in the ER model.

class diagram (UML)—A conceptual data model; a model of the static relationships between data elements of a system (similar to an ER diagram).

completeness constraint—Double-line symbol connecting a supertype entity with the subtypes to designate that the listed subtype entities represent a complete set of possible subtypes.

composition—A relationship between one class and a group of other classes in UML; the class at the diamond (aggregate) end of the relationship is composed of the class(es) at the small (component) end; similar to aggregation in the ER model.

conceptual data model—An organization of data that describes the relationships among the primitive data elements. For example, in the ER model, it is a diagram of the entities, their relationships, and their attributes.

connectivity of a relationship—A constraint on the count of the number of associated entity occurrences in a relationship, either one or many.

data item—The basic component of a data record in a file or database table; the smallest unit of information that has meaning in the real world. Examples include customer lastname, address, and identification number.

data model—An organization of data that describes the relationships among primitive and composite data elements.

data warehouse—A large repository of historical data that can be integrated for decision support.

database—A collection of interrelated stored data that serves the needs of multiple users; a collection of tables in the relational model.

database administrator (DBA)—The person in a software organization who is in charge of designing, creating, and maintaining the databases of an enterprise. The DBA makes use of a variety of software tools provided by a DBMS.

database life cycle—An enumeration and definition of the basic steps in the requirements analysis, design, creation, and maintenance of a database as it evolves over time.

database management system (DBMS)—A generalized software system for storing and manipulating databases. Examples include Oracle, IBM's DB2, Microsoft SQL Server, or Access.

data mining—A way of extracting knowledge from a database by searching for correlations in the data and presenting promising hypotheses to the user for analysis and consideration.

DBA—See database administrator.

degree of a relationship—The number of entities associated in the relationship: recursive binary (1 entity), binary (2 entities), ternary (3 entities), *n*-ary (*n* entities).

denormalization—The consolidation of database tables to increase performance in data retrieval (query), despite the potential loss of data integrity. Decisions on when to denormalize tables are based on cost/benefit analysis by the DBA.

deployment diagram (UML)—Shows the physical nodes on which a system executes. This is more closely associated with physical database design.

dimension table—The smaller tables used in a data warehouse to denote the attributes of a particular dimension, such as time, location, customer characteristics, product characteristics, etc.

disjointness constraint (d)—A symbol in an ER diagram to designate that the lower-level entities in a generalization relationship have nonoverlapping (disjoint) occurrences. If the occurrences overlap, then use the designation (o) in the ER diagram.

entity—A data object that represents a person, place, thing, or event if informational interest; it corresponds to a record in a file when stored. For example, you could define employee, customer, project, team, and department as entities.

entity cluster—The result of a grouping operation on a collection of entities and relationships in an ER model to form a higher-level abstraction, which can be used to more easily keep track of the major components of a large-scale global schema.

entity instance (or occurrence)—A particular occurrence of an entity. For example, an instance of the entity actor would be Johnny Depp.

entity-relationship (ER) diagram—A diagram (or graph) of entities and their relationships, and possibly the attributes of those entities.

entity-relationship (ER) model—A conceptual data model involving entities, relationships among entities, and attributes of those entities.

exclusion constraint—A symbol (+) between two relationships in the ER model with a common entity that implies that either one relationship must hold at a given point in time, or the other must hold, but not both.

existence dependency—A dependency between two entities such that one is dependent upon the other for its existence, and cannot exist alone. For example, an employee work-history entity cannot exist without the corresponding employee entity. Also refers to the connectivity between two entities as being mandatory or optional.

fact table—The dominating table in a data warehouse and its star schema, containing dimension attributes and data measures at the individual data level.

fifth normal form (5NF)—A table is in fifth normal form (5NF) if and only if there are no possible lossless decompositions into any subset of tables; in other words, if there is no possible lossless decomposition, then the table is in 5NF (see Section 6.5).

file—A collection of records of the same type. For example, an employee file is a collection of employee records.

first normal form (1NF)—A table is in first normal form (1NF) if and only if there are no repeating columns of data taken from the same domain and having the same meaning.

foreign key—Any attribute in a SQL table (key or nonkey) that is taken from the same domain of values as the primary key in another SQL table and can be used to join the two tables (without loss of data integrity) as part of a SQL query.

fourth normal form (4NF)—A table is in fourth normal form (4NF) if and only if it is at least in BCNF and if whenever there exists a nontrivial multivalued dependency of the form X->>Y, then X must be a superkey in the table.

functional dependency (FD)—The property of one or more attributes (data items) that uniquely determines the value of one or more other attributes (data items). Given a table R, a set of attributes B is functionally dependent on another set of attributes A if, at each instant of time, each A value is associated with only one B value.

generalization—A special type of abstraction relationship that specifies that several types of entities with certain common attributes can be generalized (or abstractly defined) with a higher-level entity type, a supertype entity; an "is-a" type relationship. For example, employee is a generalization of engineer, manager, and administrative assistant, based on the common attribute job-title. A tool often used to make view integration possible.

global schema—A conceptual data model that shows all the data and their relationships in the context of an entire database.

key—A generic term for a set of one or more attributes (data items) that, taken collectively, enables one to identify uniquely an entity or a record in a SQL table; a superkey.

logical design—The steps in the database life cycle involved with the design of the conceptual data model (schema), schema integration, transformation to SQL tables, and table normalization; the design of a database in terms of how the data is related, but without regard to how it will be stored.

lossless decomposition—A decomposition of a SQL table into two or more smaller tables is lossless if and only if the cycle of table decomposition (normalization) and the recomposition (joining the tables back through common attributes) can be done without loss of data integrity.

mandatory existence—A connectivity between two entities that has a lower bound of one. One example is the "works-in" relationship between an employee and a department; every department has at least one employee at any given time. Note: if this is not true, then the existence is optional.

multiplicity—In UML, the multiplicity of a class is an integer that indicates how many instances of that class are allowed to exist.

multivalued dependency (MVD)—The property of a pair of rows in a SQL table such that if the two rows contain duplicate values of attribute(s) X, then there is also a pair of rows obtained by interchanging the values of Y in the original pair. This is a multivalued

dependency of X->>Y. For example, if two rows have the attribute values ABC and ADE, where X=A and Y has the values C and E, then the rows ABE and ADC must also exist for an MVD to occur. A trivial MVD occurs when Y is a subset of X or X union Y is the entire set of attributes in the table.

normalization—The process of breaking up a table into smaller tables to eliminate problems with unwanted loss of data (the egregious side effects of losing data integrity) from the deletion of records and inefficiencies associated with multiple data updates.

online analytical processing (OLAP)—A query service that overlays a data warehouse by creating and maintaining a set of summary views (automatic summary tables, or ASTs) to enable quick access to summary data.

optional existence—A connectivity between two entities that has a lower bound of zero. For example, for the "occupies" relationship between an employee and an office, there may exist some offices that are not currently occupied.

package—In UML, a package is a graphical mechanism used to organize classes into groups for better readability.

physical design—The step in the database life cycle involved with the physical structure of the data; that is, how it will be stored, retrieved, and updated efficiently. In particular, it is concerned with issues of table indexing and data clustering on secondary storage devises (disk).

primary key—A key that is selected from among the candidate keys for a SQL table to be used to create an index for that table.

qualified association—In UML, an association between classes may have constraints specified in the class diagram.

record—A group of data items treated as a unit by an application; a row in a database table.

referential integrity—A constraint in a SQL database that requires, for every foreign key instance that exists in a table, that the row (and thus the primary key instance) of the parent table associated with that foreign key instance must also exist in the database.

reflexive association—In UML, a reflexive association relates a class to itself.

relationship—A real-world association among one or more entities. For example, "purchased" could be a relationship between customer and product.

requirements specification—A formal document that defines the requirements for a database in terms of the data needed, the major users and their applications, the physical platform and software system, and any special constraints on performance, security, and data integrity.

row—A group of data items treated as a unit by an application; a record; a tuple in relational database terminology.

schema—A conceptual data model that shows all the relationships among the data elements under consideration in a given context; the collection of table definitions in a relational database.

second normal form (2NF)—A table is in second normal form (2NF) if and only if each nonkey attribute (data item) is fully dependent on the primary key, that is either the left side of every functional dependency (FD) is a primary key or can be derived from a primary key.

star schema—The basic form of data organization for a data warehouse, consisting of a single large fact table and many smaller dimension tables.

subtype entity—The higher-level abstract entity in a generalization relationship.

superkey—A set of one or more attributes (data items) that, taken collectively, allows the unique identification of an entity or a record in a relational table.

supertype entity—The lower-level entity in a generalization relationship.

table—In a relational database, the collection of rows (or records) of a single type (similar to a file).

ternary relationship—A relationship that can only be defined among occurrences of three entities.

third normal form (3NF)—A table is in third normal form (3NF) if and only if for every functional dependency X->A, where X and A are either simple or composite attributes (data items), either X must be a superkey or A must be a member attribute of a candidate key in that table.

UML—Unified Modeling Language; a popular form of diagramming tools used to define data models and processing steps in a software application.

view integration—A step in the logical design part of the database life cycle that collects individual conceptual data models (views) into a single unified global schema. Techniques such as generalization are used to consolidate the individual data models.

References

Ambler, S. Agile Database Techniques: Effective Strategies for the Agile Software Developer, Wiley, 2003.

Bachman, C. W. "Data Structure Diagrams," *Database* 1,2 (1969), pp. 4–10.

Bachman, C. W. "The Evolution of Storage Structures," *Comm. ACM* 15,7 (July 1972), pp. 628–634.

Bachman, C. W. "The Role Concept in Data Models," *Proc. 3rd Intl. Conf. on Very Large Data Bases*, Oct. 6, 1977, IEEE, New York, pp. 464–476.

Barquin, R., and Edelstein, H. (editors). *Planning and Designing the Data Warehouse*, Prentice-Hall, Upper Saddle River, NJ, 1997, Chapter 10 (OLAP).

Batini, C., Lenzerini, M., and Navathe, S. B. "A Comparative Analysis of Methodologies for Database Schema Integration," *ACM Computing Surveys* 18,204 (Dec. 1986), pp. 323–364

Batini, C., Ceri, S., and Navathe, S. *Conceptual Database Design: An Entity-Relationship Approach*, Benjamin/Cummings, 1992.

Beeri, C., Fagin, R., and Howard, J. H. "A Complete Axiomization Functional and Multivalued Dependencies in Database Relations," *Proc. 1977 ACM SIGMOD Int'l. Conf. on Management of Data*, Toronto, 1977, pp. 47–61.

Beeri, C., Bernstein, P., and Goodman, N. "A Sophisticated Introduction to Database Normalization Theory," *Proc. 4th Intl. Conf. on Very Large Data Bases*, Sept. 13–15, 1978, IEEE, New York, pp. 113–124.

Bernstein, P. "Synthesizing 3NF Tables from Functional Dependencies," *ACM Trans. Database Systems* 1,4(1976), pp. 272–298.

Briand, H., Habrias, H., Hue, J., and Simon, Y. "Expert System for Translating E-R Diagram into Databases," *Proc. 4th Intl. Conf. on Entity-Relationship Approach*, IEEE Computer Society Press, 1985, pp. 199–206.

Brodie, M. L., Mylopoulos, J., and Schmidt, J. (editors). *On Conceptual Modeling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*, Springer-Verlag, New York, 1984.

Bruce, T. A. *Designing Quality Databases with IDEF1X Information Models*, Dorset House, New York, 1992.

Bulger B., Greenspan, J., and Wall, D. *MySQL/PHP Database Applications,* 2nd ed., Wiley, 2004.

Burleson, D. K. *Physical Database Design Using Oracle*, CRC Press, 2004.

Cardenas, A. F. "Analysis and Performance of Inverted Database Structures," Communications of the ACM, vol. 18, no. 5, pp. 253–264, May 1975.

Cataldo, J. "Care and Feeding of the Data Warehouse," *Database Prog. & Design* 10, 12 (Dec. 1997), pp. 36–42.

Chaudhuri, S., and Dayal, U. "An Overview of Data Warehousing and OLAP Technology," *SIGMOD Record* 26, 1 (March 1997), pp. 65–74.

Chen, P. P. "The Entity-Relationship Model—Toward a Unified View of Data," *ACM Trans. Database Systems* 1,1 (March 1976), pp. 9–36.

Chen and Associates, Inc. *ER Designer* (user manual), 1987.

Cobb, R. E., Fry, J. P., and Teorey, T. J. "The Database Designer's Workbench," *Information Sciences* 32,1 (Feb. 1984), pp. 33–45.

Codd, E. F. "A Relational Model for Large Shared Data Banks," *Comm. ACM* 13,6 (June 1970), pp. 377–387.

Codd, E. F. "Recent Investigations into Relational Data Base Systems," *Proc. IFIP Congress*, North-Holland, Amsterdam, 1974.

Codd, E. F. "Extending the Database Relational Model to Capture More Meaning," *ACM TODS* 4,4 (Dec. 1979), pp. 397–434.

Codd, E. F. *The Relational Model for Database Management, Version 2*, Addison-Wesley, 1990.

Computer Associates, AllFusion ERwin Data Modeler, http://www3.ca.com/Solutions/Product.asp?ID=260

Database Answers, Modeling tools, http://www.databaseanswers.com/modelling_tools.htm

Date, C. J. *An Introduction to Database Systems*, Vol. 1, 7th ed., Addison-Wesley, 1999.

Directory of Data Modeling Resources, http://www.infogoal.com/dmc/dmcdmd.htm

Dittrich, K. R., Gotthard, W., and Lockemann, P. C. "Complex Entities for Engineering Applications," *Proc. 5th ER Conf.*, North-Holland, Amsterdam, 1986.

Dutka, A. F., and Hanson, H. H. *Fundamentals of Data Normalization*, Addison-Wesley, Reading, MA, 1989.

Elmasri, R. and Navathe, S. B. *Fundamentals of Database Systems*, 4th ed., Addison-Wesley, 2003.

Fagin, R. "Multivalued Dependencies and a New Normal Form for Relational Databases," *ACM Trans. Database Systems* 2,3 (1977), pp. 262–278.

Fagin, R. "Normal Forms and Relational Database Operators," *Proc. 1979 ACM SIGMOD Conf. on Mgmt. of Data*, pp. 153–160.

Faloutsos, C., Matias, Y., and Silberschatz, A. "Modeling Skewed Distributions Using Multifractal and the '80–20 Law'," in *Proceedings of the 22nd VLDB Conference*, pp. 307–317, 1996.

Feldman, P., and Miller, D. "Entity Model Clustering: Structuring a Data Model by Abstraction," *Computer Journal* 29,204 (Aug. 1986), pp. 348–360.

Fowler, M. *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2002.

Gennick, J. *Oracle SQL*Plus: The Definitive Guide*, O'Reilly, 1999.

Gennick, J. *SQL Pocket Guide*, O'Reilly, 2004.

Gray, P., and Watson, H. J. *Decision Support in the Data Warehouse*, Prentice-Hall, Upper Saddle River, NJ, 1998.

Halpin, T. *Information Modeling and Relational Databases: From Conceptual Analysis to Logical Design*, Morgan Kaufmann, 2001.

Hammer, M., and McLeod, D. "Database Description with SDM: A Semantic Database Model," *ACM Trans. Database Systems* 6,3 (Sept. 1982), pp. 351–386.

Han, J., and Kamber, M., *Data Mining: Concepts and Techniques*, Morgan Kaufmann Publishers, San Francisco, 2001.

Harinarayan, V., Rajaraman, A., and Ullman, J. D., "Implementing Data Cubes Efficiently," in *Proceedings of the 1996 ACM-SIGMOD Conference*, pp. 205–216, 1996.

Harriman, A., Hodgetts, P., and Leo, M. "Emergent Database Design: Liberating Database Development with Agile Practices," Agile Development Conference, Salt Lake City, June 22–26, 2004. http://www.agiledevelopmentconference.com/files/XR2-2.pdf

Hawryszkiewycz, I. *Database Analysis and Design*, SRA, Chicago, 1984.

Hernandez, M. J. and Getz, K. *Database Design for Mere Mortals: A Hands-On Guide for Relational Databases*, 2nd ed., Addison-Wesley, 2003.

Hull, R., and King, R. "Semantic Database Modeling: Survey, Applications, and Research Issues," *ACM Computing Surveys* 19, 3 (Sept. 1987), pp. 201–260.

IBM Rational Software, http://www-306.ibm.com/software/rational/

IDEF1X, http://www.idef.com

Jajodia, S., and Ng, P. "Translation of Entity-Relationship Diagrams into Relational Structures," *J. Systems and Software* 4,2 (1984), pp. 123–133.

Jensen, C. S. and Snodgrass, R. T. "Semantics of Time-Varying Information," *Information Systems* 21,4 (1996), pp. 311–352.

Kent, W. "A Simple Guide to Five Normal Forms in Relational Database Theory," *Comm. ACM* 26,2 (Feb. 1983), pp. 120–125.

Kent, W. "Fact-Based Data Analysis and Design," *J. Systems and Software* 4 (1984), pp. 99–121.

Kimball, R., and Caserta, J. *The Data Warehouse ETL Toolkit*, 2nd ed., Wiley, 2004.

Kimball, R., and Ross, M. *The Data Warehouse Lifecycle Toolkit*, 2nd ed., Wiley, 1998.

Kimball, R., and Ross, M. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling*, 2nd ed., Wiley, 2002.

Kotidis, Y., and Roussopoulos, N. "DynaMat: A Dynamic View Management System for Data Warehouses," in *Proceedings SIGMOD '99*, pp. 371–382, 1999.

Maier, D. *Theory of Relational Databases*, Computer Science Press, Rockville, MD, 1983.

Makridakis, S., Wheelwright, C., and Hyndman, R. J. *Forecasting Methods and Applications*, 3rd ed., John Wiley & Sons, 1998.

Martin, J. *Strategic Data-Planning Methodologies*, Prentice-Hall, 1982.

Martin, J. *Managing the Data-Base Environment*, Prentice-Hall, 1983.

McGee, W. "A Contribution to the Study of Data Equivalence," *Data Base Management*, J. W. Klimbie and K. L. Koffeman (eds.), North-Holland, Amsterdam, 1974, pp. 123–148.

McLeod, D., and King, R. "Applying a Semantic Database Model," *Proc. 1st Intl. Conf. on the Entity-Relationship Approach to Systems Analysis and Design*, North-Holland, Amsterdam, 1979, pp. 193–210.

Melton, J., and Simon, A. R. *Understanding The New SQL: A Complete Guide*, Morgan Kaufmann Pub., 1993.

Mitchell, T. M. *Machine Learning*, WCB/McGraw-Hill, Boston, 1997.

Muller, R. *Database Design for Smarties: Using UML for Data Modeling*, Morgan Kaufmann Pub., 1999.

Mullins, C. S. *DB2 Developer's Guide*, 5th ed., Sams Publishing, 2004.

Nadeau, T. P., and Teorey, T. J. "Achieving Scalability in OLAP Materialized View Selection," In *Proceedings of DOLAP '02*, pp 28–34, 2002.

Nadeau, T. P., and Teorey, T. J. "A Pareto Model for OLAP View Size Estimation," *Information Systems Frontiers*, vol. 5, no. 2, pp. 137–147, Kluwer Academic Publishers, 2003.

Naiburg, E. J., and Maksimchuk, R. A. *UML for Database Design*, Addison-Wesley, 2001.

Neilson, P. *Microsoft SQL Server Bible*, Wiley, 2003.

Nijssen, G. M., and Halpin, T. A. *Conceptual Schema and Relational Database Design: A Fact Oriented Approach*, Prentice-Hall, 1989.

Objects by Design: UML modeling tools, http://www.objectsbydesign.com/tools/umltools_byCompany.html

Peckham, J., and Maryanski, F. "Semantic Data Models," *ACM Computing Surveys* 20, 3 (Sept. 1988), pp. 153–190.

Quatrani, T. *Visual Modeling with Rational Rose 2002 and UML*, 3rd ed., Addison-Wesley, 2003.

Ramakrishnan, R., and Gehrke, J. *Database Management Systems*, 3rd ed., McGraw Hill, 2004, Chapter 20.

Reiner, D., Brown, G., Friedell, M., Kramlich, D., Lehman, J., McKee, R., Rheingans, P., and Rosenthal, A. "A Database Designer's Workbench," *Proc. 5th ER Conference*, North-Holland, Amsterdam, 1986, pp. 347–360.

Rumbaugh, J., Jacobson, I., and Booch, G. *The Unified Modeling Language User Guide*, 2nd ed., Addison-Wesley, 2004.

Rumbaugh, J., Jacobson, I., and Booch, G. *The Unified Modeling Language Reference Manual*, 2nd ed., Addison-Wesley, 2005.

Sakai, H. "Entity-Relationship Approach to Logical Database Design," *Entity-Relationship Approach to Software Engineering*, C. G. Davis, S. Jajodia, P.A. Ng, and R. T. Yeh (editors), Elsevier, North-Holland, Amsterdam, 1983, pp. 155–187.

Scheuermann, P., Scheffner, G., and Weber, H. "Abstraction Capabilities and Invariant Properties Modelling within the Entity-Relationship Approach," *Entity-Relationship Approach to Systems Analysis and Design*, P. Chen (editor), Elsevier, North-Holland, Amsterdam, 1980, pp. 121–140.

Senko, M. et al. "Data Structures and Accessing in Database Systems," *IBM Syst. J.* 12,1 (1973), pp. 30–93.

Silberschatz, A., Korth, H. F., and Sudarshan, S. *Database System Concepts* 4th ed., McGraw-Hill, 2002.

Simon, A. R. *Strategic Database Technology: Management for the Year 2000*, Morgan Kaufmann, San Francisco, 1995.

Simsion, G. C., and Witt, G. C. *Data Modeling Essentials: Analysis, Design, and Innovation*, 2nd ed., Coriolis, 2001.

Smith, H. "Database Design: Composing Fully Normalized Tables from a Rigorous Dependency Diagram," *Comm. ACM* 28,8 (1985), pp. 826–838.

Smith, J., and Smith, D. "Database Abstractions: Aggregation and Generalization," *ACM Trans. Database Systems* 2,2 (June 1977), pp. 105–133.

Snodgrass, R. T. *Developing Time-Oriented Database Applications in SQL*, Morgan Kaufmann Publishers, 2000.

Stephens, R., and Plew, R. *Database Design*, Sams, 2000.

Sybase PowerDesigner, http://www.sybase.com/products/developmentintegration/powerdesigner

Teichroew, D., and Hershey, E. A. "PSL/PSA: A Computer Aided Technique for Structured Documentation and Analysis of Information Processing Systems," *IEEE Trans. Software Engr.* SE-3,1 (1977), pp. 41–48.

Teorey, T., and Fry, J. *Design of Database Structures*, Prentice-Hall, 1982.

Teorey, T. J., Wei, G., Bolton, D. L., and Koenig, J. A. "ER Model Clustering as an Aid for User Communication and Documentation in Database Design," *Comm. ACM* 32,8 (Aug. 1989), pp. 975–987.

Teorey, T. J., Yang, D., and Fry, J. P. "A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model," *ACM Computing Surveys* 18,2 (June 1986), pp. 197–222.

Thomsen, E. *OLAP Solutions*, Wiley, 1997.

Tsichritzis, D., and Lochovsky, F. *Data Models*, Prentice-Hall, 1982.

Ubiquiti Inc. Web site: http://www.ubiquiti.com/

UML Overview, from Developer.com, http://www.developer.com/design/article.php/1553851

Understanding relational databases: referential integrity http://www.miswebdesign.com/resources/articles/wrox-beginning-php-4-chapter-3-5.html

The University of Waikato, Weka 3—Data Mining with Open Source Machine Learning Software in Java. http://www.cs.waikato.ac.nz/ml/weka.

van der Lans, R. *Introduction to SQL: Mastering the Relational Database Language*, 3rd ed., Addison-Wesley, 2000.

Widom, J. Data Management for XML. http://www-db.stanford.edu/~widom/xml-whitepaper.html

Wilmot, R. "Foreign Keys Decrease Adaptability of Database Designs," *Comm. ACM* 27,12 (Dec. 1984), pp. 1237–1243.

Witten, I., and Frank, E., *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*, Morgan Kaufmann Publishers, San Francisco, 2000.

Wong, E., and Katz, R. "Logical Design and Schema Conversion for Relational and DBTG Databases," *Proc. Intl. Conf. on the Entity-Relationship Approach*, 1979, pp. 311–322.

Yao, S. B. (editor). *Principles of Database Design*, Prentice-Hall, 1985.

Zachmann, J. A. "A Framework for Information Systems Architecture," *IBM Systems Journal* 26,3 (1987), IBM Pub. G321-5298.

Zachmann Institute for Framework Advancement, http://www.zifa.com.

Exercises

ER and UML Conceptual Data Modeling

Problem 2-1

Draw a detailed ER diagram for an car rental agency database (e.g., Hertz), keeping track of the current rental location of each car, its current condition and history of repairs, customer information for a local office, expected return date, return location, car status (ready, being-repaired, currently-rented, being-cleaned). Select attributes from your intuition about the situation and list them separately from the diagram, but associated with a particular entity or relationship in the ER model.

Problem 2-2

Given the following assertions for a relational database that represents the current term enrollment at a large university, draw an ER diagram for this schema that takes into account all the assertions given. There are 2,000 instructors, 4,000 courses, and 30,000 students. Use as many ER constructs as you can to represent the true semantics of the problem.

Assertions:

- An instructor may teach one or more courses in a given term (average is 2.0 courses).
- An instructor must direct the research of at least one student (average = 2.5 students).
- A course may have none, one, or two prerequisites (average = 1.5 prerequisites).
- A course may exist even if no students are currently enrolled.
- Each course is taught by exactly one instructor.
- The average enrollment in a course is 30 students.
- A student must select at least one course per term (average = 4.0 course selections).

Problem 3-1

Draw UML class diagrams for an car rental agency database (e.g., Hertz), keeping track of the current rental location of each car, its current condition and history of repairs, customer information for a local office, expected return date, return location, car status (ready, being-repaired, currently-rented, being-cleaned). Select attributes from your intuition about the situation.

Draw one diagram showing the relationships of the classes without the attributes listed, then show each class individually with the attributes listed.

Problem 3-2

Given the following assertions for a relational database that represents the current term enrollment at a large university, draw an UML diagram for this schema that takes into account all the assertions given. There are 2,000 instructors, 4,000 courses, and 30,000 students. Use as many UML constructs as you can to represent the true semantics of the problem.

Assertions:

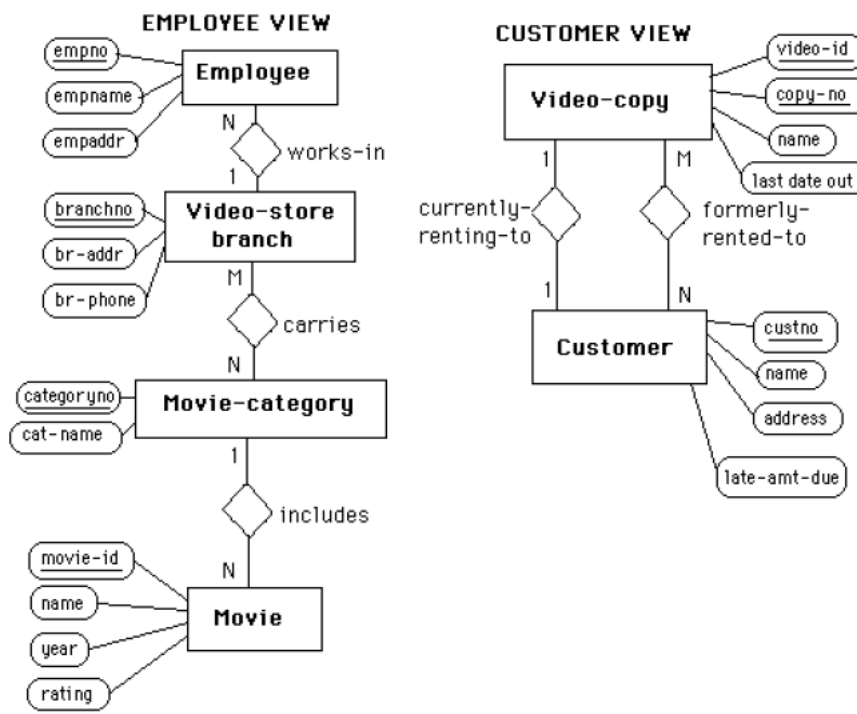
- An instructor may teach none, one, or more courses in a given term (average is 2.0 courses).

- An instructor must direct the research of at least one student (average = 2.5 students).
- A course may have none, one, or two prerequisites (average = 1.5 prerequisites).
- A course may exist even if no students are currently enrolled.
- Each course is taught by exactly one instructor.
- The average enrollment in a course is 30 students.
- A student must select at least one course per term (average = 4.0 course selections).

Conceptual Data Modeling and Integration

Problem 4-1

The following ER diagrams represent two views of a video store database as described to a database designer. Show how the two views can be integrated in the simplest and most useful way by making all necessary changes on the two diagrams. State any assumptions you need to make.



Transformation of the Conceptual Model to SQL

Problem 5-1

1. Transform your integrated ER diagram from Problem 4-1 into a SQL database with five to ten rows per table of data you make up to fit the database schema.
2. Demonstrate your database by displaying all the queries below:
   a. Which video store branches have *Shrek* in stock (available) now?
   b. In what section of the store (film category) can you find *Terminator*?
   c. For customer "Anika Sorenstam," what titles are currently being rented and what are the overdue charges, if any?
   d. Any query of your choice. (Show me what your system can really do!)

Normalization and Minimum Set of Tables

Problem 6-1

Given the table R1(A, B, C) with FDs A -> B and B -> C:

1. Is A a superkey for this table? _____
2. Is B a superkey for this table? _____
3. Is this table in 3NF, BCNF, or neither? _____

Problem 6-2

Given the table R(A,B,C,D) with FDs AB->C and BD->A:

1. What are all the superkeys of this table? _____
2. What are all the candidate keys for this table? _____
3. Is this table in 3NF, BCNF, or neither? _____

Problem 6-3

1. From the ER diagram given below and the resulting tables and candidate key attributes defined, what are all the functional dependencies (FDs) you can derive just by looking at the diagram and list of candidate keys?

Table	Candidate Key(s)
customer	cid
order	orderno
department	deptno
salesperson	sid
item	itemno
order-dept-sales	orderno,sid AND orderno,deptno
order-item-sales	orderno, itemno

Table	FDs
customer	
order	
department	
salesperson	
item	
order-dept-sales	
order-item-sales	

2. What is the level of normalization for each of these tables, based on the information given?

Problem 6-4

The following functional dependencies (FDs) represent a set of airline reservation system database constraints. Design a minimum set of BCNF tables, preserving all FDs, and express your solution in terms of the code letters given below (a time-saving device for your analysis). Is the set of tables you derived also BCNF?

reservation_no -> agent_no, agent_name, airline_name, flight_no, passenger_name

reservation_no -> aircraft_type, departure_date, arrival_date, departure_time, arrival_time

reservation_no -> departure_city, arrival_city, type_of_payment, seating_class, seat_no
airline_name, flight_no -> aircraft_type, departure_time, arrival_time
airline_name, flight_no -> departure_city, arrival_city, meal_type
airline_name, flight_no, aircraft_type -> meal_type
passenger_name -> home_address, home_phone, company_name
aircraft_type, seat_no -> seating_class
company_name -> company_address, company_phone
company_phone -> company_name

A: reservation_no	L: departure_city
B: agent_no	M: arrival_city
C: agent_name	N: type_of_payment
D: airline_name	P: seating_class
E: flight_no	Q: seat_no
F: passenger_name	R: meal_type
G: aircraft_type	S: home_address
H: departure_date	T: home_phone
I: arrival_date	U: company_name
J: departure_time	V: company_address
K: arrival_time	W: company_phone

Problem 6-5

Given the following set of FDs, find the minimum set of 3NF tables. Designate the candidate key attributes of these tables. Is the set of tables you derived also BCNF?

1. J -> KLMNP
2. JKL -> MNP
3. K -> MQ
4. KL -> MNP
5. KM -> NP
6. N -> KP

Problem 6-6

Using only the given set of functional dependencies (FDs), find the minimum set of BCNF tables. Show the results of each step of Bernstein's algorithm separately. What are the candidate keys for each table?

1. ABC -> DEFG
2. B -> CH
3. A -> HK
4. CD -> GKM
5. D -> CP
6. E -> FG
7. G -> CDM

Problem 6-7

Given the FDs listed below, determine the minimum set of 3NF tables, preserving all FDs. Define the candidate keys of each table and determine which tables are also BCNF.

1. ABC -> H
2. AC -> BDEFG
3. ACDF -> BG
4. AW->BG
5. B -> ACF
6. H -> AXY
7. M -> NZ
8. MN -> HPT
9. XY -> MNP

Problem 6-8

Given the following FDs, determine the minimum set of 3NF tables. Make sure that all FDs are preserved. Specify the candidate keys of each table. Note that each letter represents a separate data element (attribute). Is the set of tables you derived also BCNF?

1. ABCD -> EFGHIJK
2. ACD -> JKLMN
3. A -> BH
4. B -> JKL
5. BH -> PQR
6. BL -> PS
7. EF -> ABCDH
8. JK -> B
9. MN -> ACD
10. L -> JK
11. PQ -> S
12. PS -> JKQ
13. PSR -> QT

Problem 6-9

Given the following FDs, determine the minimum set of 3NF tables. Make sure that all FDs are preserved. Specify the candidate keys of each table. Note that each letter represents a separate data element (attribute). Is the set of tables you derived also BCNF?

1. A -> BGHJ
2. AG -> HK
3. B -> K
4. EA -> F
5. EB -> AF
6. EF -> A
7. H -> J
8. J -> AB

Problem 6-10

FDs and MVDs

Answer each question yes or no, and justify each answer. In most cases, you will be given a table R with a list of attributes, with at most one candidate key (the candidate key may be either a single attribute or composite attribute key, shown underlined).

Given table R(A,B,C,D) and the functional dependency AB->C:

1. Is R in 3NF?
2. Is R in BCNF?
3. Does the multivalued dependency AB ->>C hold?
4. Does the set {R1(A,B,C), R2(A,B,D)} satisfy the lossless join property?

Given table R(A,B,C) and the set {R1(A,B), R2(B,C)} satisfies the lossless decomposition property:

1. Does the multivalued dependency B->>C hold?
2. Is B a candidate key?
3. Is R in 4NF?

Given a table "skills_available" with attributes empno, project, and skill, in which the semantics of "skills_available" state that every skill an employee has must be used on every project that employee works on:

1. Is the level of normalization of "skills_available" at least 4NF?
2. Given table R(A,B,C) with actual data shown below:
   a. Does the multivalued dependency B->>C hold?
   b. Is R in 5NF?

R:

A	B	C
w	x	p
w	x	q
z	x	p
z	x	q
w	y	q
z	y	p

Logical Database Design (Generic Problem)

Problem 7-1

Design and implement a small database that will be useful to your company or student organization.

1. State the purpose of the database in a few sentences.
2. Construct an ER or UML class diagram for the database.
3. Transform your ER or UML diagram into a working database with five to ten rows per table of data you can make up to fit the data-

base schema. You should have at least four tables, enough to have some interesting queries. Use Oracle, DB2, SQL Server, Access, or any other database system.

4. Show that your database is normalized (BCNF) using FDs derived from your ER diagram and from personal knowledge of the data. Analyze the FDs for each table separately (this simplifies the process).
5. Demonstrate your working database by displaying the results of four queries. Pick interesting and complex queries (impress us!).

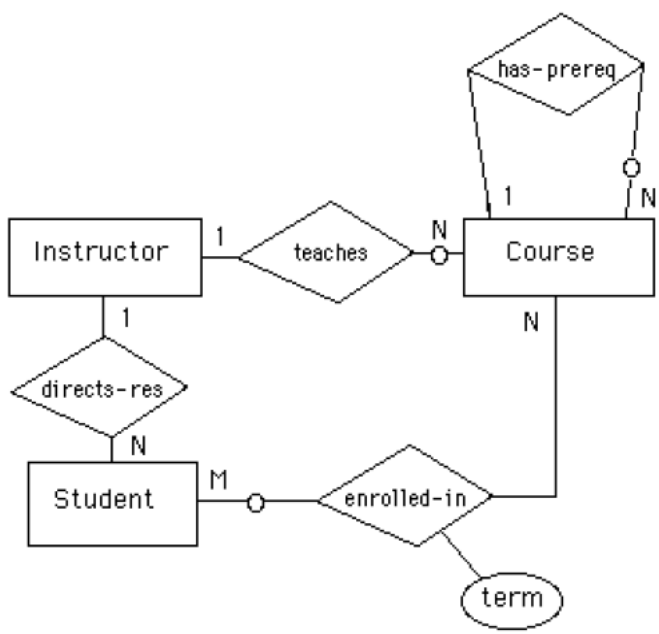
OLAP

Problem 8-1

As mentioned in Chapter 8, hypercube lattice structures are a specialization of product graphs. Figure 8.16 shows an example of a three-dimensional hypercube lattice structure. Figure 8.13 shows an example of a two-dimensional product graph. Notice that the two figures are written using different notations. Write the hypercube lattice structure in Figure 8.16 using the product graph notation introduced with Figure 8.13. Keep the same dimension order. Don't worry about carrying over the view sizes. Assume the Customer, Part, and Supplier dimensions are keyed by "customer id," "part id," and "supplier id," respectively. Shade the nodes representing the fact table and the views selected for materialization as indicated in Section 8.2.4.

Solutions to Selected Exercises

Problem 2-2



Problem 4-1

Connect Movie to Video-copy as a 1-to-*N* relationship (Video-copy at the *N* side); or, use a generalization from Movie to Video-copy, with Movie being the supertype and Video-copy as the subtype.

Problem 6-1

Given the table R1(A, B, C) with FDs A -> B and B -> C:

1. Is A a superkey for this table? *Yes*
2. Is B a superkey for this table? *No*
3. Is this table in 3NF, BCNF, or neither? *Neither 3NF nor BCNF*

Problem 6-3

Table	*FDs*	*Level of Normalization*
customer	cid -> cname, caddress	BCNF
order	orderno -> cid	BCNF
department	NONE	BCNF
salesperson	sid -> deptno	BCNF
item	itemno -> deptno, itemname, size	BCNF
order-dept-sales	orderno, sid -> deptno	BCNF
	orderno, deptno -> sid	BCNF
order-item-sales	orderno, itemno -> sid	BCNF

Problem 6-5

Given these FDs, begin Step 1 (LHS reduction):

1. J -> KLMNP
2. JKL -> MNP *First, eliminate K and L since J -> KL in (1); merge with (1)*

3. K -> MQ
4. KL -> MNP — *Third, eliminate L since K -> MNP from merged (3), (4) is redundant*
5. KM -> NP — *Second, eliminate M since K -> M in (3); merge with (3)*
6. N -> KP

End of Step 1, begin Step 2 (RHS reduction for transitivities):

1. J -> KLMNP — *First, reduce by eliminating MNP since K -> MNP*
2. K -> MQNP — *Second, reduce by eliminating P since N -> P*
3. N -> KP

End of Step 2 and consolidation in Step 3:

1. J -> KL
2. K -> MNQ (or K -> MNPQ) — *First, merge (2) and (3) for superkey rules 1 and 2*
3. N -> KP (or N -> K)

Steps 4 and 5:

1. J -> KL — *Candidate key is J (BCNF)*
2. K -> MNPQ and N -> K — *Candidate keys are K and N (BCNF)*

Problem 6-7

Given these FDs:

1. ABC -> H — *Step 1: B is extraneous due to AC -> B in (3)*
2. ACDF -> BG — *Step 1: EF are extraneous due to AC -> DF in (3)*
3. AC -> BDEFG — *Step 2: eliminate F from RHS due to B -> F from (5)*
4. AW-> BG — *Step 2: eliminate G from RHS due to B -> AC -> G from (3) and (5)*
5. B -> ACF

6. H -> AXY
7. M -> NZ
8. MN -> HPT *Step 1: N is extraneous due to M -> N in (7)*
9. XY -> MNP *Step 2: eliminate NP from the RHS due to M -> NP from (7,8)*

After Step 3, using the union axiom:

1. AC -> BDEGH *Combining (1), (2), and (3)*
2. AW->B
3. B -> ACF
4. H -> AXY
5. M -> HNPTZ *Combining (7) and (8)*
6. XY -> M

Step 4, merging:

1. AC -> BDEFGH, B -> AC, H -> A using Rule 1 for AC being a superkey, Rule 2 for B being a superkey, the definition of 3NF, and A being a prime attribute. 3NF only.
2. AW->BG using Rule 1 for AW to be a superkey. BCNF.
3. H -> XY, XY -> M, and M -> HNPTZ using Rule 1 for H being a superkey (after taking H to its closure H -> XYMNPTZ), using Rule 2 for M being a superkey from M -> H, and Rule 2 for XY being a superkey from XY -> M. BCNF. Note: H->AXY is also possible here.

Step 5, minimum set of normalized tables:

Table 1: ABCDEFGH with superkeys AC, B (3NF only)
Table 2: ABGW with superkey AW (3NF and BCNF)
Table 3: HMNPTXYZ with superkeys H, XY, M (3NF and BCNF)

Index