DATABASE MODELS

The development of *database management systems (DBMSs)* over the last 40 years has had a significant impact on the way in which we work. Whether at home, in the office, in the classroom, or on the factory floor, we are faced with the need to store and retrieve data that we use in the day-to-day operation of our personal lives and commercial endeavors. This fact is especially evident with the widespread use of personal computers, laptops, and networking facilities such as the Internet. Data are truly at our fingertips.

To use data effectively, data must be organized and data must be shared. At a very basic level, data can be stored within files and managed using the file system facilities provided by the computer's operating system. In the early years of computing, files provided the primary facility for the storage and management of data. Several basic problems exist, however, with the use of traditional file processing. First of all, it is generally the user's responsibility to understand how to interpret the contents of a file or to understand how the contents of several different files may be related. Furthermore, files can be difficult to share when concurrent access is required. This difficulty can often lead to the redundant storage of data as users create their own individual files of information. A *database* provides a way of collecting together related data that are 1) described and accessed using a common language, where all users have a consistent interpretation of the data; and 2) stored in a form where multiple users can share access to the same data. As described by Elmasri and Navathe (1), a DBMS is a "general-purpose software system that facilitates the processes of defining, constructing, and manipulating databases for various applications."

This article focuses on the process of defining a database through the use of *database models*. A database model provides a way to conceptually describe the data that are to be stored in a database. A database model thus provides an abstract, conceptual layer on top of the actual database that protects users of the database from the need to be concerned with low-level, implementation details. The conceptual description that is created through the use of a specific model is developed by modeling a real-world enterprise that will ultimately provide the data that are to be stored in the database.

Several different types of database models have been developed since the 1960s. Some database models, such as the *network*, *hierarchical*, and *relational* data models, are closely tied to specific types of DBMSs. Other database models, known as *semantic* or *conceptual* data models, provide a DBMS-independent way of describing data that can then be translated to the database model of a specific DBMS implementation. All of these models generally provide a way to describe data in terms of the objects that are to be stored, the relevant characteristics of the objects, and the relationships that exist between objects. More recent *object-based* data models, including object-oriented and object-relational database models, provide a way to describe the behavioral characteristics of data through the specification of operations on objects.

A database model also provides the basis for the expression of *semantic integrity constraints*. Semantic integrity constraints are concerned with describing the validity of the data within a database according to restrictions that exist in the real-world enterprise that the database represents. At any given time, the data in the database must accurately reflect the constraints associated with the database; otherwise, the data may not be a true representation of the world it is intended to model. Some integrity constraints are a natural part of the structural constructs supported by the database model; other integrity constraints can be expressed in separate constraint languages that enhance the functionality of the database model. Still other constraints may be specified and enforced through the use of operations on objects. The different database models that have been introduced over the years provide varying degrees of support for semantic integrity constraints. Database modeling and the specification of integrity constraints has generally progressed from DBMS-dependent models (with limited support for the specification of integrity constraints), to conceptual data models (with greater support for the specification of integrity constraints), to object-oriented data models (with operations that encapsulate the specification and enforcement of constraints).

The following pages present the fundamental concepts associated with conceptual and relational database models. The article begins with a description of basic data modeling concepts, establishing the terminology and definitions that are relevant throughout the rest of this article. After presenting a brief history of the development of database models, several representative database models are described. Conceptual modeling concepts are introduced using the *Entity-Relationship Model* (2). More advanced conceptual modeling concepts are presented using the *Enhanced Entity-Relationship Model* (1). The *Relational Data Model* (3) is then presented as a DBMS-dependent model that can be used to implement database applications that are initially specified using a conceptual model. Throughout the article, the role of integrity constraints in the different types of data models is emphasized. Mapping procedures from conceptual models to the relational model are also addressed. This article ends by describing the object-oriented data model, the integration of object-oriented and relational concepts in the object-relational data model, and new directions for data modeling in the context of advanced applications, distributed computing, and the Internet.

FUNDAMENTAL DATA MODELING CONCEPTS

To develop a database for a specific application, a database model is used to develop a *schema* for the application. A schema is developed using a *data definition language (DDL)* that provides a means for describing the *entities* or *objects* that will be stored in the database and the *relationships* that exist between the objects. The DDL also provides a means for specifying the data types of the *attributes* that characterize each object and for specifying several of the

integrity constraints of the application. Some DDLs are textual in nature, whereas others provide graphical languages for specifying a schema.

A sample portion of a schema for a relational database application describing movie stars and film projects is shown in Fig. 1. The schema in Fig. 1 describes the data to be stored in terms of *tables* and the type of information to be stored in each table. The MOVIE-STAR table describes the information to be stored about movie star entities, whereas the FILM-PROJECT table describes the information associated with film project entities. The ACTS-IN table describes the information that will be needed to capture the film projects in which a star works and the amount of income that a star receives from each project. A schema such as the one shown in Fig. 1 is also referred to as the *intension* of the database.

The schema of an application is compiled to create the actual physical data structures that will be used to support the database. Figure 2 presents an example of the data that might be stored according to the schema in Fig. 1. The data are referred to as the *extension* of the database. The individual rows of each table are referred to as *records*. The *columns* of each row define the *data elements* of each record. The data elements must conform to the attribute types define in Fig. 1. Attribute types are referred to as *domains*. For example, the domain of a movie star's name is defined to be a character string of length 30.

The extension can alternatively be referred to as the *instance* of the database. At any given time, the database instance must satisfy the integrity constraints of the application. An example of an integrity constraint from the schema of Fig. 1 is the PRIMARY KEY statement of the MOVIE-STAR table definition. The PRIMARY KEY statement defines that the value of the STAR-ID attribute for each movie star entity must be unique (i.e., there must never be two stars that have the same identifier). Similar PRIMARY KEY statements are defined for the FILM-PROJECT and ACTS-IN tables. The FOREIGN KEY statements also define constraints on the types of values that can appear in the ACTS-IN table. The first FOREIGN KEY statement defines that values in the STAR-ID column of the ACTS-IN table must be valid values from the STAR-ID column of the MOVIE-STAR table. The second FOREIGN KEY statement defines a similar constraint on the FILM-ID attribute of the ACTS-IN table.

In addition to a DDL, a DBMS also provides a *data manipulation language (DML)*. A DML is used to insert data into a database, to delete data from a database, and to modify the database contents. A DML also includes a *query language* for retrieving data from the database. For example, a query may be needed over the database of Fig. 2 to retrieve the movie stars that are making more than $5,000,000 in a film project. Another example of a query might be a request to display the names of the stars that are currently working on a specific film project.

Query languages can be *procedural* in nature, where the query is expressed in the style of imperative programming languages. Procedural query languages are *navigational*, retrieving data one record at a time. Query languages can also be *declarative*, specifying *what* data are to be retrieved rather than *how* data are to be retrieved. Declarative query languages provide a *set-oriented* style of retrieval, where several records can be retrieved at once. SQL, which stands for *S*tructured English *Q*uery *L*anguage, is the declarative query language of the relational data model (4). A query specification expressed using a language such as SQL can be used to define *views* over a database. For example, a specific user may not need to see the entire database of Fig. 2 but may only need to see the stars working on the *Louisiana Saturday Night* film project. The corresponding view created using an SQL query is expressed as follows:

```
CREATE VIEW LSN-STARS AS
SELECT STAR-ID, NAME
FROM MOVIE-STAR, FILM-PROJECT, ACTS-IN
WHERE FILM-PROJECT.TITLE = "Louisiana Saturday Night" and
ACTS-IN.STAR-ID = MOVIE-STAR.STAR-ID and
ACTS-IN.FILM-ID = FILM-PROJECT.FILM-ID;
```

The schema in Fig. 1 is DBMS-dependent because it is specifically associated with the relational database approach to describing data. Figure 3 provides an example of the same schema described in a DBMS-independent manner. The specific graphical notation used is that of the Entity-Relationship model, which is explained in greater detail later in this article. The Entity-Relationship model is a type of conceptual data model that is used to describe database applications in a manner that is independent of the type of database that will eventually be used for the implementation. For complex applications, it is often easier to first describe the application using a model such as the Entity-Relationship model and then to map the description to an implementation-oriented model, such as the relational model.

To place the concepts described above into perspective, Fig. 4 presents a diagram of the three-schema database architecture that was originally developed by the ANSI/SPARC committee (5). As shown in Fig. 4, a database can be viewed as consisting of an *internal* level, a *conceptual* level, and an *external* level. The internal level describes the physical structures that are used to store and retrieve the extension of the database, such as the data in the tables in Fig. 2. The conceptual level describes the data at a more abstract level, such as the description in Fig. 1 or in Fig. 3. The external level is defined over the conceptual level using queries to create user-specific views, such as the LSN-STARS view defined above.

Mappings exist between each level to create the notion of *data independence*. For example, the mapping between the conceptual level and the internal level represents *physical data independence*. If a database environment supports physical data independence, then it should be possible to modify the underlying physical structure without affecting the conceptual description of the data. In other words, the user should not be concerned with how a database, such as the one in Fig. 2, is implemented. If the physical implementation details change, the user's conceptual view should remain the same. The mapping between the external and the conceptual level represents *logical data independence*. Logical data independence defines the ability to modify the conceptual schema without affecting the external views. Logical data independence is more difficult to achieve because the specification of queries that

```
CREATE TABLE MOVIE-STAR
        (STAR-ID        VARCHAR(9)      NOT NULL,
         NAME           VARCHAR(30)     NOT NULL,
         DBATE          DATE,
PRIMARY KEY (STAR-ID));

CREATE TABLE FILM-PROJECT
        (FILM-ID        VARCHAR(5)      NOT NULL,
         TITLE          VARCHAR(50)     NOT NULL,
         BUDGET         INT,
PRIMARY KEY (FILM-ID));

CREATE TABLE ACTS-IN
        (STAR-ID        VARCHAR(9)      NOT NULL,
         FILM-ID        VARCHAR(5)      NOT NULL,
         INCOME         INT,
PRIMARY KEY (STAR-ID, FILM-ID),
FOREIGN KEY (STAR-ID) REFERENCES MOVIE-STAR(STAR-ID),
FOREIGN KEY (FILM-ID) REFERENCES FILM-PROJECT(FILM-ID));
```

Figure 1. A relational database schema expressed using a textual data definition language.

MOVIE-STAR:

STAR-ID	NAME	BDATE
123456789	George Johns	06/16/55
222334444	Murray Hill	09/02/51
333445555	Ann Dugan	10/31/65
444556666	Elizabeth Jillian	08/23/87
555667777	Joseph Andrew	02/18/82
666778888	Joseph Edward	09/02/51
777889999	Christine Darling	02/15/22
888990101	William Edward	04/15/21

FILM-PROJECT:

FILM-ID	TITLE	BUDGET
11111	California Dreaming	15,000,000
22222	Louisiana Saturday Night	10,000,000
33333	Valley of the Sun	20,000,000

ACTS-IN:

ACTOR-ID	FILM-ID	INCOME
123456789	22222	2,000,000
222334444	33333	1,000,000
333445555	11111	2,000,000
444556666	22222	3,000,000
555667777	33333	2,000,000
666778888	11111	3,000,000
666778888	22222	3,000,000
777889999	33333	2,000,000
777889999	11111	2,000,000
888990101	22222	3,000,000

Figure 2. Three tables illustrating an extension of the relational schema in Fig. 1

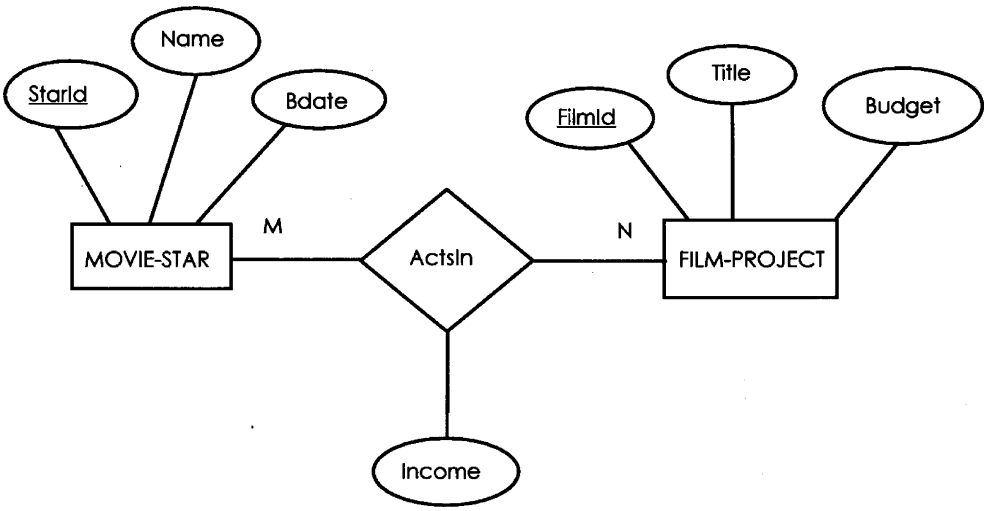


Figure 3. An Example of an ER diagram illustrating the ActsIn relationship between MOVIE-STAR entities and FILM-PROJECT entities.

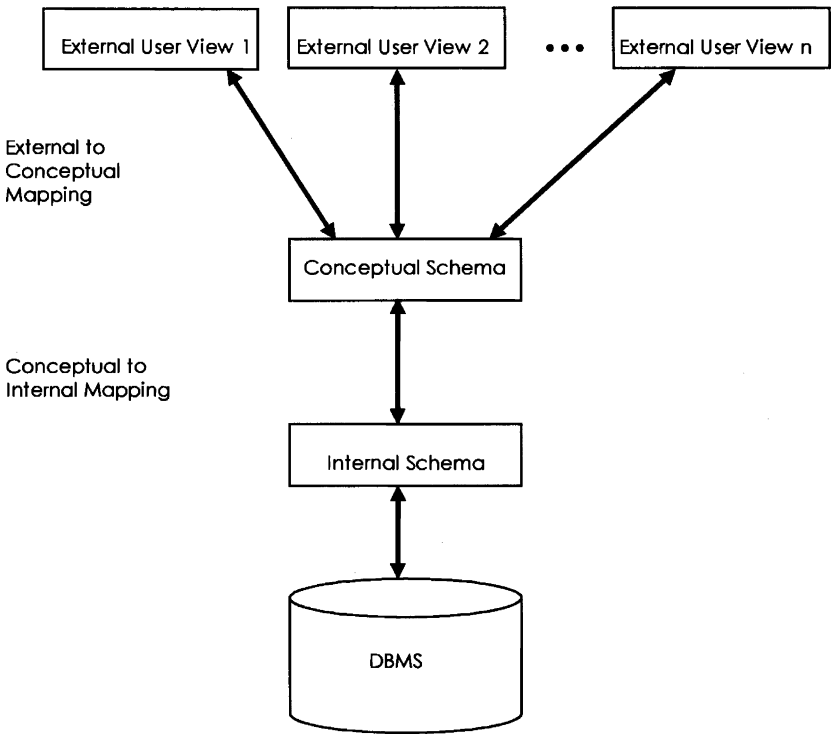


Figure 4. The ANSI/SPARC three-schema architecture. External views are queries defined on conceptual schemas such as relational or entity-relationship schemas. Conceptual schemas are implemented using the physical, internal schema of a specific DBMS.

define external views may be affected by changes to the conceptual schema.

DEVELOPMENT OF DATABASE MODELS

Figure 5 presents a hierarchical view of the major database models that have been developed since the 1960s. As mentioned in the previous section, database models can be broadly divided into implementation models that are associated with specific types of DBMSs and conceptual data models that provide an abstract, DBMS-independent way of modeling database applications. The oldest database models, which are now referred to as legacy data models, are the Network Data Model and the Hierarchical Data Model. The Network Data Model was developed by Bachmann (6) as part of General Electric's Integrated Data Store (IDS) product. Standards associated with the Network Data Model were developed by the Conference on Data Systems Languages Database Task Group (CODASYL) (7). Around the same time, the Hierarchical Data Model was developed as part of IBM's Information Management System (IMS) (8). The network data model provides a graph-based approach to the description of data, whereas the hierarchical data model provides a tree-based structure for organizing data. Both data models provide procedural DMLs for the retrieval and modification of data.

In 1970, Ted Codd published the first description of the Relational Data Model (3). The relational data model provided a different approach to the description of database applications that is based on the simple concept of tables. In addition to its simplicity, the model also provided a formal basis for the description of data using concepts from set theory. Perhaps the most important aspect of the relational model was the definition of the relational algebra for the set-oriented retrieval of data, providing a significant departure from the record-at-at-time retrieval approach provided by network and hierarchical models. By the later part of the 1970s, several relational database research projects had developed, the most notable being System R at IBM (9) and Ingres at the University of California, Berkeley (10) under the direction of Michael Stonebraker. By the early 1980s, commercial relational database products began to appear on the market with SQL as the standard query language interface.

Around the same time that relational database research projects were gaining strength, interest began to develop in describing data in a more abstract way than that provided by the network, hierarchical, and relational database models. In 1976, Peter Chen published his description of the Entity-Relationship model (2). The Entity-Relationship model was not based on any particular DBMS implementation, providing a more conceptual approach to the description of database entities, their attributes, and their relationships. The model was presented as a database design tool that could be used to characterize the data needed by a database application before developing a specific implementation using one of the three major types of database systems.

At about the same time that the Entity-Relationship model was developed, Smith and Smith (11) defined the concepts of *aggregation*, *generalization*, and *specialization* as data modeling abstractions that provided an even more

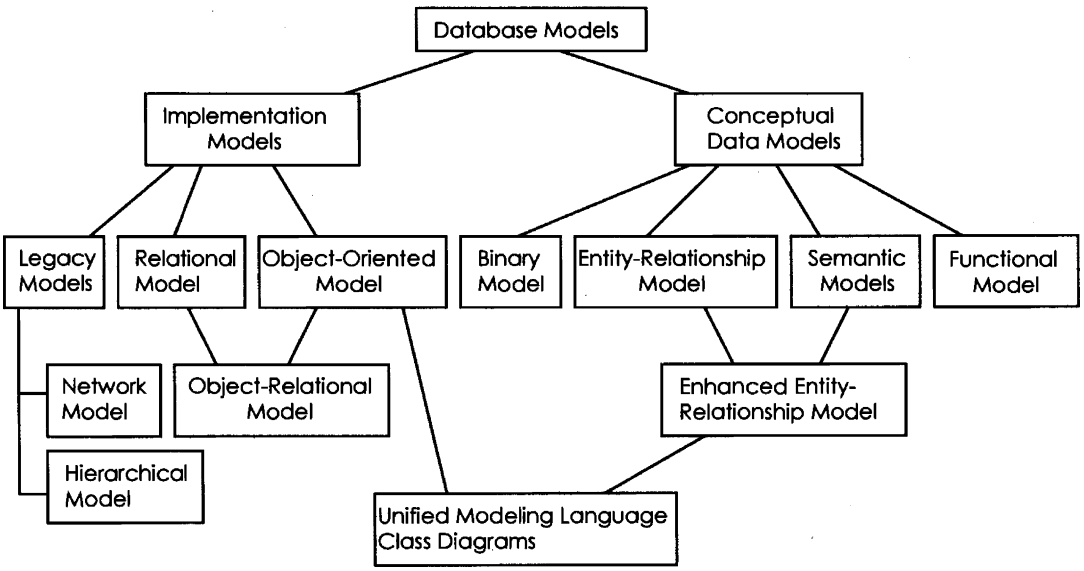


Figure 5. A hierarchical view of database models.

semantic approach to the description of data than that provided by the Entity-Relationship model. The data abstractions described by Smith and Smith were based on concepts from knowledge representation as used in the area of artificial intelligence. The introduction of these data abstractions to the database community invoked the development of several semantic data models that were developed in the late 1970s and early 1980s (12–15), although credit for the first semantic data model is generally given to Abrial who developed the Binary Data Model in 1974 (16). The Binary Data Model established a conceptual modeling approach that used binary relationships for the description of database application semantics. In general, semantic data models provide features that allow the database designer to incorporate a higher degree of semantic integrity constraints directly into the description of the data. Semantic data models also provide a more object-oriented approach to the description of data than that found in the network, hierarchical, and relational data models. Excellent surveys of semantic data modeling can be found in References 17 and 18.

After the definition of semantic data modeling concepts, extensions were made to the Entity-Relationship model to create the Enhanced Entity-Relationship model (1), which incorporated additional modeling features typically found in semantic data models. Several variations of extensions to the Entity-Relationship model have been developed (19–23). The Functional Data model also represents another type of conceptual modeling tool that was developed by Sibley and Kerschberg in 1977 (24). DAPLEX is perhaps one of the most well-known research projects involving the use of the functional data model (25). Functional data models are similar to semantic data models but use concepts from mathematical functions as the primary modeling construct.

The 1980s brought forth the development of object-oriented database management systems (OODBMSs), together with object-oriented data modeling concepts. Unlike the relational model, in which a complete formal description of the model appeared before the development of research prototypes and commercial systems, commercial object-oriented database systems began to appear before the database community fully agreed on any common, formally defined description of an object-oriented database model. OODBMSs are different from previous database systems in that they incorporate behavior into the database definition through the use of encapsulated objects. Data are defined not only in terms of objects and relationships but also in terms of the operations that manipulate the data. The concepts in object-oriented data modeling parallel the concepts found in object-oriented programming languages. As a result, a "marriage" of object-oriented database and programming language concepts occurred, providing a more seamless approach to the manipulation of database objects through procedural programming languages. Querying in the OODBMS paradigm returned to the navigational programming style of the network and hierarchical data models, although object algebras (26,27) have been defined. The Object Data Management Group (ODMG) ad hoc standards committee has defined standards for an object model, an object query language based on SQL, and programming language bindings (28).

Another milestone in the development of database models has been the integration of relational and object-oriented data modeling concepts to create object-relational database systems. Several relational database researchers published the Third Generation Database System Manifesto (29) in response to the Object-Oriented Database Systems Manifesto (30) published by the OODBMS community. Whereas the Object-Oriented Database Systems Manifesto defines the characteristics of object-oriented database systems, the third-generation document describes the manner in which relational technology can be extended with object-oriented concepts as well as other advanced features such as triggers and rules and still retain the data independence and query language advan-

tages of the relational model. The Postgres research prototype (31), an object-relational version of Ingres, is generally recognized as the seminal work on the definition of object-relational database modeling concepts, most of which are documented in Reference 32. Today, object-relational features are supported by several commercial database products, such as Oracle and IBM DB2.

Finally, Fig. 5 also includes the *Unified Modeling Language (UML) Class Diagrams* (33) as an intersection of concepts from object-oriented models and the EER model. UML is a standard for object-oriented modeling that is maintained by the Object Management Group (34), providing a collection of modeling techniques for describing the structural and dynamic aspects of software application design, including the specification of use-cases and code module interaction. Class diagrams, also known as *static structural diagrams*, are a subcomponent of UML that provide graphical techniques for object-relationship modeling together with the specification of operations that define the behavioral characteristics of objects. UML class diagrams, therefore, provide an ideal modeling approach for object-oriented and object-relational database designs. Dietrich and Urban (35) provide a comparison of UML class diagrams to EER modeling and describe techniques for mapping class diagrams to relational, object-oriented, and object-relational database implementations.

CONCEPTUAL DATA MODELING

No matter what type of DBMS will be used to implement a database application, database development begins with the design of a conceptual view of the application. Since Entity-Relationship modeling is one of the most well-known techniques for conceptual modeling, the following description of conceptual modeling begins by presenting the fundamental concepts of the Entity-Relationship model. More advanced semantic data modeling concepts are then presented through the use of the Enhanced Entity-Relationship model.

Entity-Relationship Model

The Entity-Relationship (ER) approach to conceptual modeling is a graphical approach that is used to describe the entities of an application, the attributes of entities, and the relationships that exist between entities. There have been many different graphical notations developed for the ER model. Although some notations presented in the literature vary, the underlying concepts remain the same. The specific notation that will be followed in this article is the notation as used in Reference 1.

Figure 6 presents an example of an ER schema that will be used throughout this section to illustrate the fundamental ER modeling concepts. The specific application involves the modeling of movie stars, the film projects they work in, the studios that produce film projects, and the shooting schedule for each film project.

Entities and Attributes. The most fundamental modeling component of the ER model is an *entity*. An entity represents an object that exists in the world that is being modeled. In some cases, an entity represents an object that you can actually see and touch in the real world. In other cases, an entity may represent a more abstract concept. In either case, entities generally have *attributes* that are used to characterize the entity.

Graphically, entities are depicted using rectangles, whereas attributes are depicted using ovals that are attached to entities. In Fig. 6, for example, four entities are displayed. The movie star entity and the studio entity are examples of physical entities, whereas the film project and the shooting schedule entities are abstract entities. Each entity is characterized as having several attributes. For example, an actor has a StarId, a Name, a Bdate (or birthdate), and an Age. An entity together with its attributes constitute the definition of an *entity type*. An entity type is used to collect together all of the entities that can be characterized in the same way. The collection of all entities of an entity type is referred to as an *entity set*. Any given entity in the entity set is referred to as an *instance* of the entity type. Using the database modeling terminology introduced earlier, an entity type is part of the intension of the database, whereas the entity set represents the actual extension of the database.

The attributes that are used to describe entities can be of several different types. The most common type of attribute is a *single-valued* attribute. A single-valued attribute is denoted by an oval drawn with a single line, such as the StarId attribute of the movie star entity. If an attribute is single-valued, then an entity instance can only have one value for such an attribute. Each star, therefore, can only be assigned one value to serve as a StarId. If an attribute is denoted by a double oval, such as the PhoneNumber attribute of the studio entity, then entity instances are allowed to have more than one value for such an attribute. The schema in Fig. 6 thus defines that a studio entity can have more than one PhoneNumber. An attribute such as PhoneNumber is referred to as a *multi-valued* attribute.

The Name attribute of the movie star entity is an example of a *composite* attribute. A composite attribute such as Name can be broken down into its constituent parts, such as FirstName, MiddleInit (for middle initial), and LastName. A composite attribute allows the value of the attribute to be globally viewed as one combined attribute value, where the individual subcomponents are concatenated to create the combined attribute value. Alternatively, a composite attribute can be accessed in terms of the individual components that make up the composite value.

Another distinction that can be made between attributes is whether they are *stored* or *derived*. The value of a stored attribute is to be physically stored within the database that will be constructed to represent the application described by the ER schema. A derived attribute, on the other hand, is an attribute having a value that can be derived from other stored or derived attribute values. As a result, derived attributes do not require any physical space within the database, but a procedure must be developed to calculate the value when it is needed. Derived attributes are indicated through the use of an oval with a dashed border. For example, the Age attribute of the movie star entity is a value that can be derived using a procedure that takes as input the star's Bdate value.

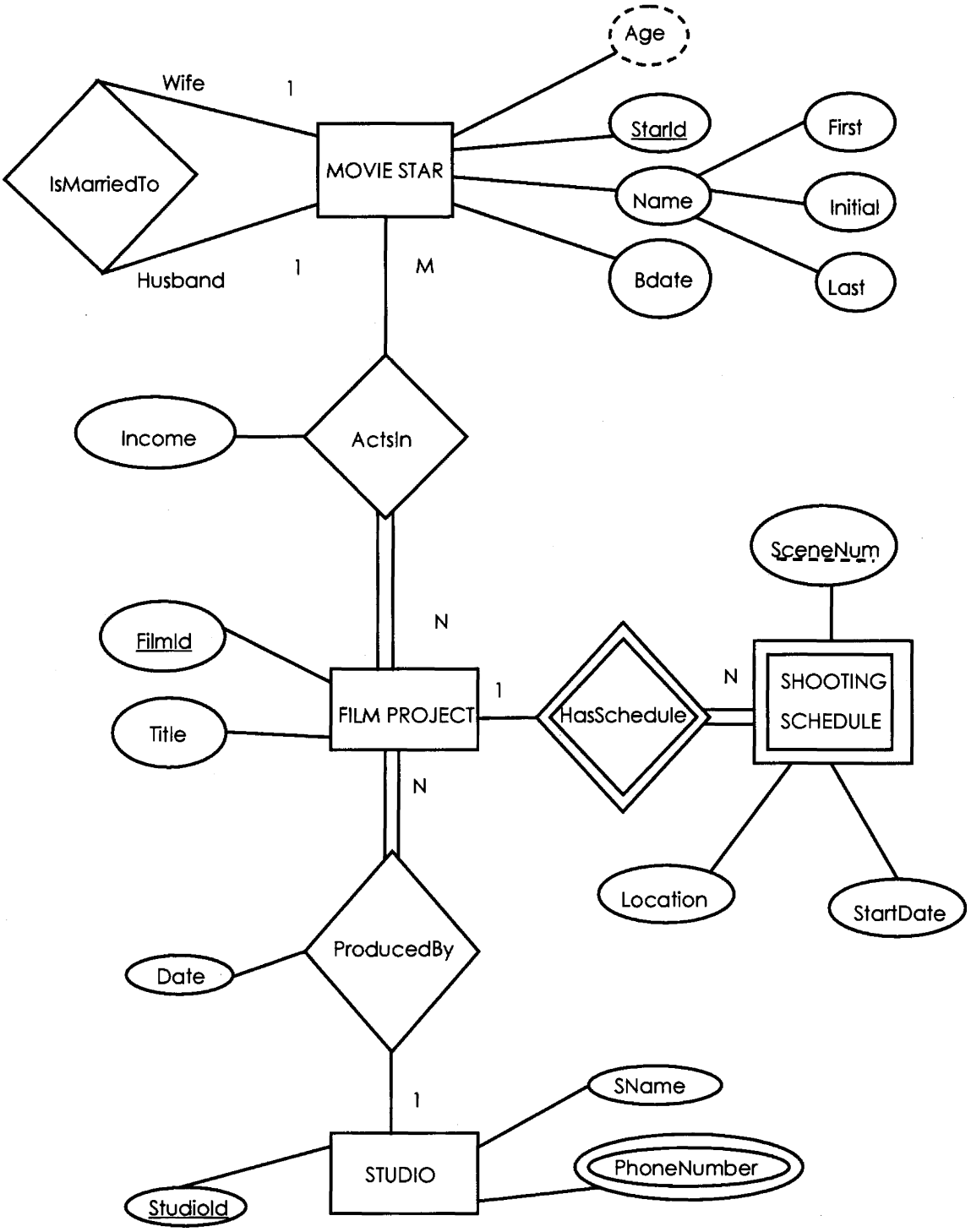


Figure 6. A graphical schema illustrating fundamental ER modeling concepts.

In Fig. 6, attributes that are underlined, such as StarId, FilmId, and StudioId, are referred to as *keys*. Keys provide a way to uniquely identify the instances of an entity set, thus establishing a *uniqueness* constraint that must be enforced by the database. If an attribute of an entity type is defined as a key, then it is assumed that no two entities of that type can have the same value. Some attributes of an entity are not appropriate to use as keys. For example, more than one movie star can have the same Name or Bdate value. Star's can, however, be assigned unique values for StarId.

By defining an attribute as a key, it is also implied that every entity must have a value for the key attribute, thus establishing a *required value* constraint. A key such as StarId is never allowed to be *null,* whereas a movie star entity could have a null value for the Bdate attribute if the

star's birth date is not known. A null value is a special value that is used in the database to indicate that an attribute value is either unknown (the entity has such as value, but the value is not known to the database), or that an attribute value is inapplicable (the entity does not have a value for the attribute). A movie star, for example, may not have a value for the Initial attribute, or as in the case of the singer Madonna, the Last attribute of Name may be null.

One aspect of entity definitions that is not graphically depicted in the notation as shown in Fig. 6 is the fact that each attribute is associated with a specific *domain*. A domain defines the valid set of values that may be assigned to an attribute. The StarId attribute, for example, could be defined to be from the domain of character strings that are nine characters long. Numeric domains, such as integers and real numbers, can be constrained with specific range values. For example, the Age attribute of a star can be constrained to be from the set of integers between 5 and 100.

Relationships Between Entities. Relationships are used to describe the interaction that can exist between the entities in the real world that the schema is intended to represent. As an example, Fig. 6 shows that a movie star ActsIn a film project. Structurally, a relationship is depicted through the use of a diamond that is connected by single or double lines to the entities that are involved in the relationship. A relationship together with the entities of the relationship form a *relationship type*. Attributes can be attached to a relationship type, as shown by the Income attribute of the ActsIn relationship. Relationships can also be recursive, in which an entity type participates in a relationship with entities of the same type. The IsMarriedTo relationship in Fig. 6 is an example of a recursive relationship. The actual occurrence of a relationship between the entities involved is referred to as a *relationship instance*. Each relationship in Fig. 6 is referred to as a *binary relationship* because each relationship describes the interaction between two entity types.

Relationships can be enhanced through the use of *structural constraints*. In particular, a relationship can be described using *cardinality constraints* and *participation constraints*. Cardinality constraints describe the number of relationship instances that can be formed between the entities of the relationship. The three main types of cardinality constraints for binary relationships are 1:1 (one-to-one), 1:N (one-to-many), and M:N (many-to-many). Graphically, cardinality constraints are depicted by placing the specific cardinalities on the lines that connect entities to the relationship. Participation constraints describe whether an entity's participation in a relationship is *total* (required) or *partial* (optional). Partial participation is shown graphically through the use of a single line to connect an entity to a relationship. Total participation is shown using a double line. Total participation defines an *existence dependency*, in which the entity cannot exist without being involved in a relationship instance.

Figure 7 shows a relationship type together with an example of a relationship instance for the IsMarriedTo relationship from Fig. 6. IsMarriedTo is a 1:1, recursive relationship, which indicates that an instance of a movie star entity type can only be married to one other instance of the movie star entity type. The lines in the relationship are labeled to distinguish the different *roles* that an entity can play in the relationship. For example, in any IsMarriedTo relationship instance, one star will play the *wife* role and the other star will play the *husband* role. Since the relationship type is described using single lines extending from the IsMarriedTo relationship diamond to the movie star entity type, the relationship is a partial relationship. As a result, a star is not required to be married to another star. Movie star instance m1, for example, is not connected to any other star through the IsMarriedTo relationship.

Figure 8 illustrates the ProducedBy 1:N relationship type together with an example of a relationship instance. The cardinality constraints indicate that a film project is produced by one studio. A studio, on the other hand, can produce many film projects. As the line connecting the film project entity type to the ProducedBy relationship diamond is a double line, a film project instance is required to participate in the relationship. Every instance of the film project type must therefore be connected to a studio instance. In the other direction, the relationship is partial, leading to studio instances that do not participate in any relationship with film project entities.

An example of an M:N relationship is the ActsIn relationship type and relationship instance example in Fig. 9. In this example, a movie star can act in any number of film projects and a film project can have several movie stars. Furthermore, a film project is required to have at least one star involved. A movie star is not required to participate in any relationship with a film project.

Weak Entities. Recall that entities can have attributes that serve as keys for the purpose of uniquely identifying an entity. Some entities, however, may not have keys and can only be identified through their relationships with other entities. These entities are referred to as *weak entities* and are graphically illustrated in an ER schema through the use of a double rectangle. The shooting schedule entity in Fig. 6 is an example of a weak entity. Weak entities always participate in a total relationship with an *identifying entity*. The identifying entity is said to *own* the relationship. The relationship between the weak entity and its identifying entity is referred to as the *identifying relationship* and is graphically indicated using a double diamond. The HasSchedule relationship is the identifying relationship for the shooting schedule entity, with film project serving as the owner of the relationship.

A weak entity typically has a *partial key*. The partial key must be used together with the key of the identifying entity to form the unique identification for the weak entity. For example, with the shooting schedule entity, SceneNum is not unique enough to be used as the identifying number for all shooting schedule entities; every film will have a scene one. SceneNum together with the FilmId, however, forms a unique key for shooting schedule entities. A partial key is always shown in an ER diagram with a dashed line underlining the partial key name. A weak entity type can be owned by more than one identifying entity.

N-ary Relationships. All relationships that have been discussed so far are binary relationships, involving relationships between two entity types. In general, relation-

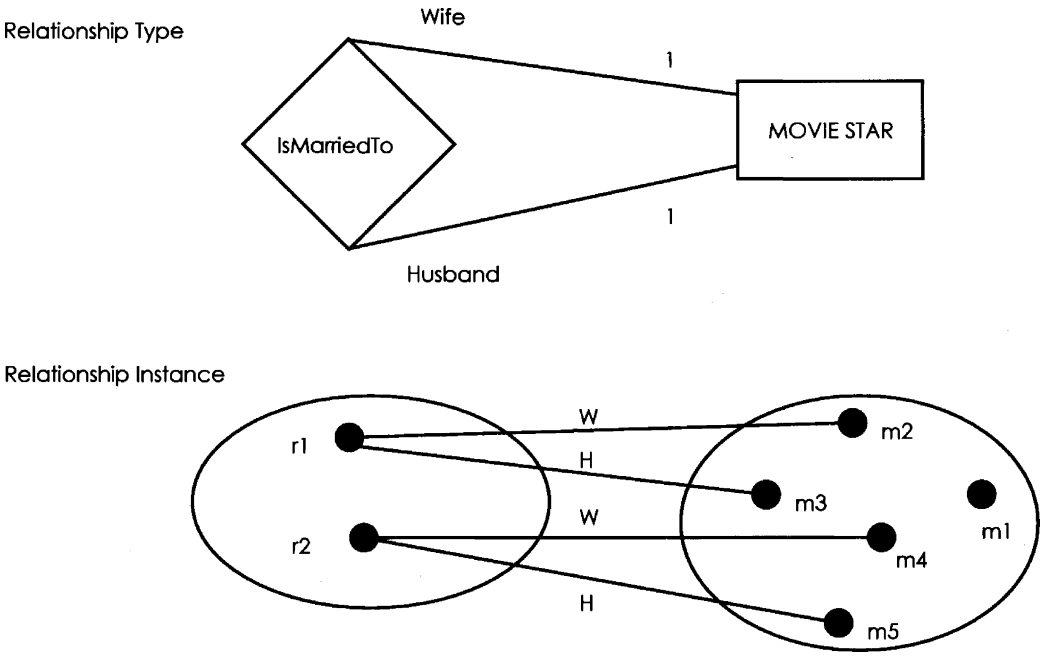


Figure 7. A 1:1, recursive relationship type and instance for the IsMarriedTo relationship.

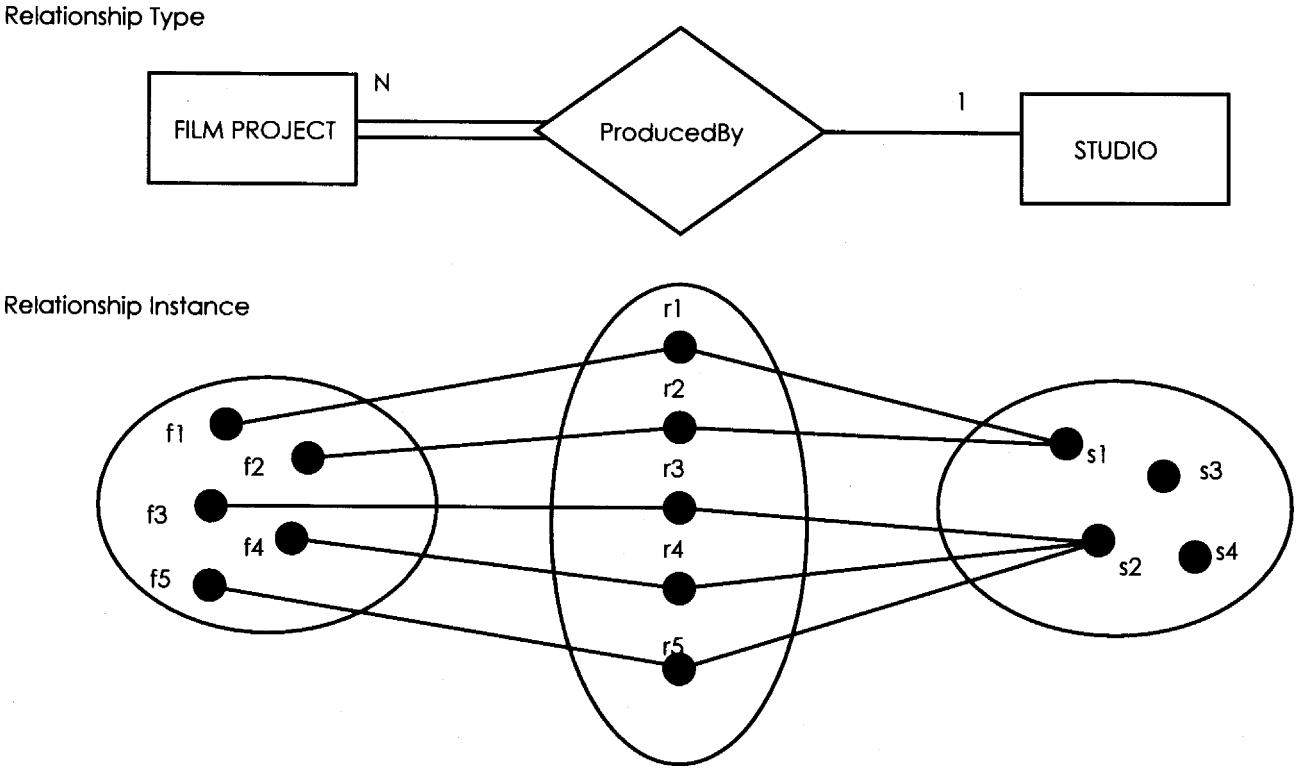


Figure 8. A 1:N relationship type and instance for the ProducedBy relationship.

ships can be N-ary, representing relationships between three or more entity types. Figure 10 presents a modeling scheme for describing the ProducedBy relationship, which illustrates ProducedBy as a three-way, or *ternary*, relationship among a film project, a studio, and a director. Decisions about whether to use an N-ary relationship or several binary relationships within a schema depend on the semantics of the application.

Enhanced Entity-Relationship Model

The EER model represents a semantically enhanced version of the ER model that was developed as a result of the object-oriented modeling and knowledge representation concepts that were used to develop semantic data mod-

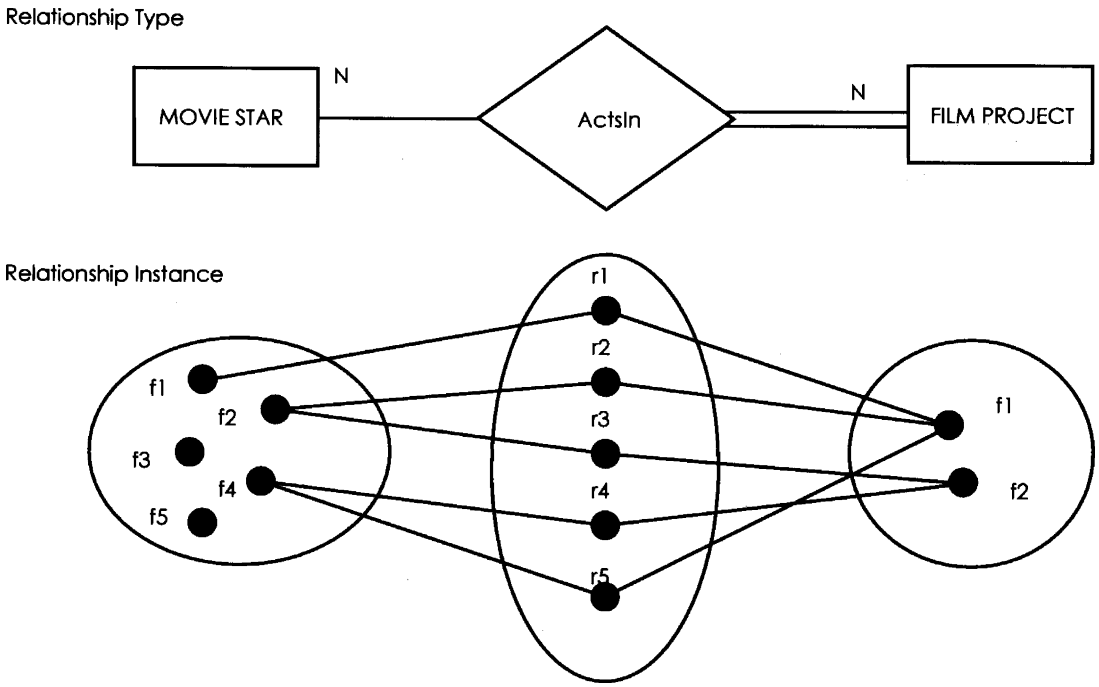


Figure 9. A M:N relationship type and instance for the ActsIn relationship.

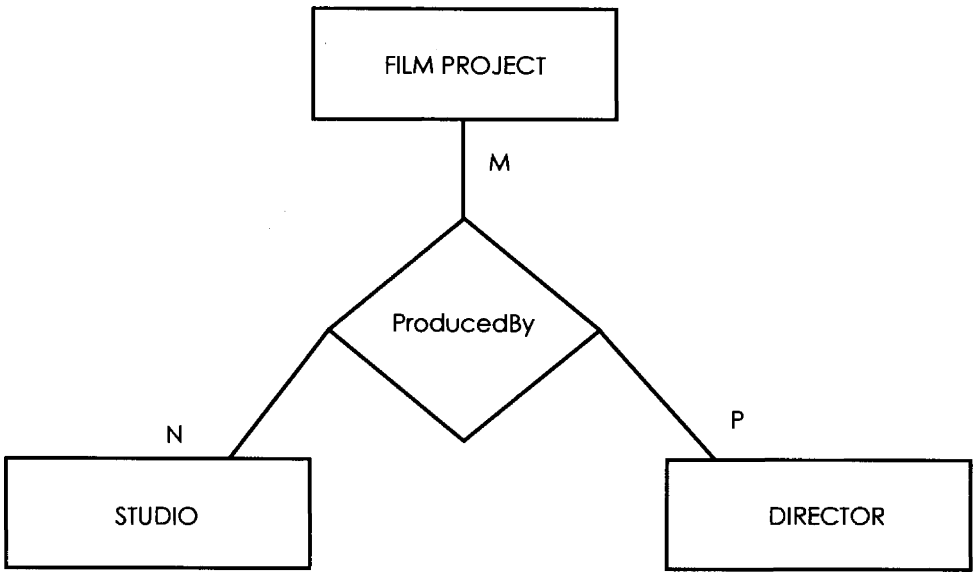


Figure 10. A ternary relationship among a studio, a film project, and a director.

els in the late 1970s and early 1980s. Semantic data models are based on three main forms of data abstraction: *classification*, *aggregation*, and *generalization / specialization* (17). Classification is a form of data abstraction in which objects of the same type are collected together into *classes*. Objects are assumed to have unique, internally assigned *object identifiers* that are not visible to the outside world. Classification of objects forms the *instance-of* relationship, in which an object is considered to be an instance of a specific class definition. Entity types in the ER and EER models support the notion of classification.

Aggregation is a form of abstraction in which a higher level object is created from relationships between lower level and/or other higher level objects. In its most basic form, the collection of simple types, such as those used to represent a social security number, a name, and an address, are combined to create an object, such as a movie star object. In its more complex form, higher level objects, such as movie stars and acting classes, can be combined to create objects that represent the enrollment of stars in acting classes. The enrollment relationship between stars and classes can then be viewed as an abstract object that can participate in relationships with other objects. The ER and the EER model only support the basic form of aggregation that is used to create entity types. Relationships in the ER approach cannot be used as objects to form addi-

tional relationships, which is generally regarded as one of the major weaknesses of the ER/EER models.

One of the most significant forms of data abstraction from semantic data modeling that has been incorporated into the EER model are the dual abstraction concepts of generalization and specialization. Generalization and specialization provide a way to form class (or entity type) definitions into *superclass/subclass hierarchies*, which are also known as *ISA hierarchies*. ISA hierarchies are a fundamental modeling concept from knowledge representation in which object classes can be formed into tree and/or graph structures that allow objects to be viewed at different levels of abstraction. Objects at lower levels of abstraction, known as subclasses, inherit the characteristics of objects at higher levels of abstraction, known as superclasses. Moving from the bottom of the tree to the top of the tree represents the notion of generalization, in which the object is viewed in its more general form. Moving from the top of the tree to the bottom of the tree represents the notion of specialization, in which an object is viewed in its more specific form. The discussion in this particular section on the EER model primarily focuses on the way in which generalization and specialization concepts have been incorporated into the EER model.

In addition to incorporating abstraction concepts from semantic data models, the EER model has also introduced an additional form of modeling abstraction known as *categorization* (23). A *category* in the EER model provides a way to define objects that represent heterogeneous collections of other object types, similar to the use of union types in C++ (36). The following subsections describe the use of generalization, specialization, and categories in further detail.

Generalization and Specialization in the EER Model. Figure 11 provides an example of an ISA hierarchy in the EER Model. This particular example illustrates the manner in which a person object can be viewed at different levels of abstraction. At the top of the hierarchy is the person class (we will use the terms class and entity type interchangeably). Every person object has an Ssn, PName, BDate, and Gender. The next level of the tree illustrates that a person object can be specialized into a person that is a movie professional or a person that is a celebrity. A movie professional has attributes, such as OfficePhone and OfficeAddress, that a celebrity does not have. Movie professionals and celebrities, however, all have an Ssn, a PName, a BDate, and a Gender. These attributes are automatically inherited from the person class because the movie professional class and the celebrity class are both defined to be subclasses of the person class. The person class is considered to be a superclass of movie professional and celebrity.

The ISA hierarchy in Fig. 11 is further specialized by defining the critic and agent classes to be subclasses of the movie professional class. As a critic is a movie professional, a critic inherits the attributes defined at the movie professional level. Furthermore, as a movie professional is a person and a person has an Ssn, a critic also has an Ssn as well as all of the other attributes that are defined at the person level. In a similar manner, movie star and model are defined to be subclasses of the celebrity class, thus defining different types of celebrities, each of which has different attributes. As with the critic and agent classes, the movie star and model classes also inherit the attributes of the person class as a movie star (or model) is a celebrity, and a celebrity is a person. Notice that by moving from the lower levels, such as movie star and critic, to the person level, the concept of generalization is applied; every object in the tree can be viewed in its more general form as a person. Moving from the person class down to more specific classes allows us to make distinctions between different types of person objects and to view objects in a more specific form.

The notation in Fig. 11 illustrates additional constraints that are placed on the classes involved in the ISA hierarchy. In particular, the circles in Fig. 11 that connect superclasses and subclasses contain either a "d" or an "o". A "d" indicates a *disjoint constraint* between the instances of the subclasses. A disjoint constraint defines that the intersection between instances of movie professional and instances of celebrity must be the empty set. In other words, it is not possible in this particular application for an object to be both a movie professional and a celebrity at the same time. The same is true for the critic and agent instances. The "o" connecting the celebrity class with the movie star and model classes defines that the instances of movie star and model can be *overlapping*. At any given point in time, it is therefore possible for a celebrity object to be an instance of the movie star class *and* an instance of the model class.

In addition to disjoint and overlapping specifications, the ISA hierarchy in Fig. 11 also specifies *total* and *partial* constraints on the specialization relationships that exist between superclasses and subclasses. The double line leading from the celebrity class to the movie star and model classes indicates a *total specialization* in which a celebrity object is required to exist as an instance in one of its subclasses. It is not possible, therefore, for an object to exist as an instance of the celebrity class and not also participate as an instance of a class at a lower level. In contrast, the single line that connects the person class to the disjoint specification for its subclasses indicates a *partial specialization*. In a partial specialization, an instance of a superclass, such as person, is not required to be an instance of any of its subclasses. It is not possible, therefore, for an object to exist as an instance of the person class and not be an instance of either the movie professional class or the celebrity class. Partial specialization emphasizes the following important property of ISA hierarchies. Objects at lower levels of a hierarchy always inherit attributes at higher levels because a lower level object "ISA" higher level object. Objects at higher levels, however, do not inherit attributes from objects at lower levels because an instance of a superclass is not necessarily an instance of its subclasses.

In the ISA hierarchy presented in Fig. 11, membership in each subclass is *user-defined*. For example, we know that a celebrity may be a movie star and/or a model, but it is left to the user of the application to determine the subclasses in which the celebrity object belongs. Nothing exists within the schema to determine whether a celebrity object should be a movie star or a model. Membership in subclasses can also be determined through the use of attribute values at the superclass level, as illustrated in Fig. 12. In this case, projects are specialized into film projects

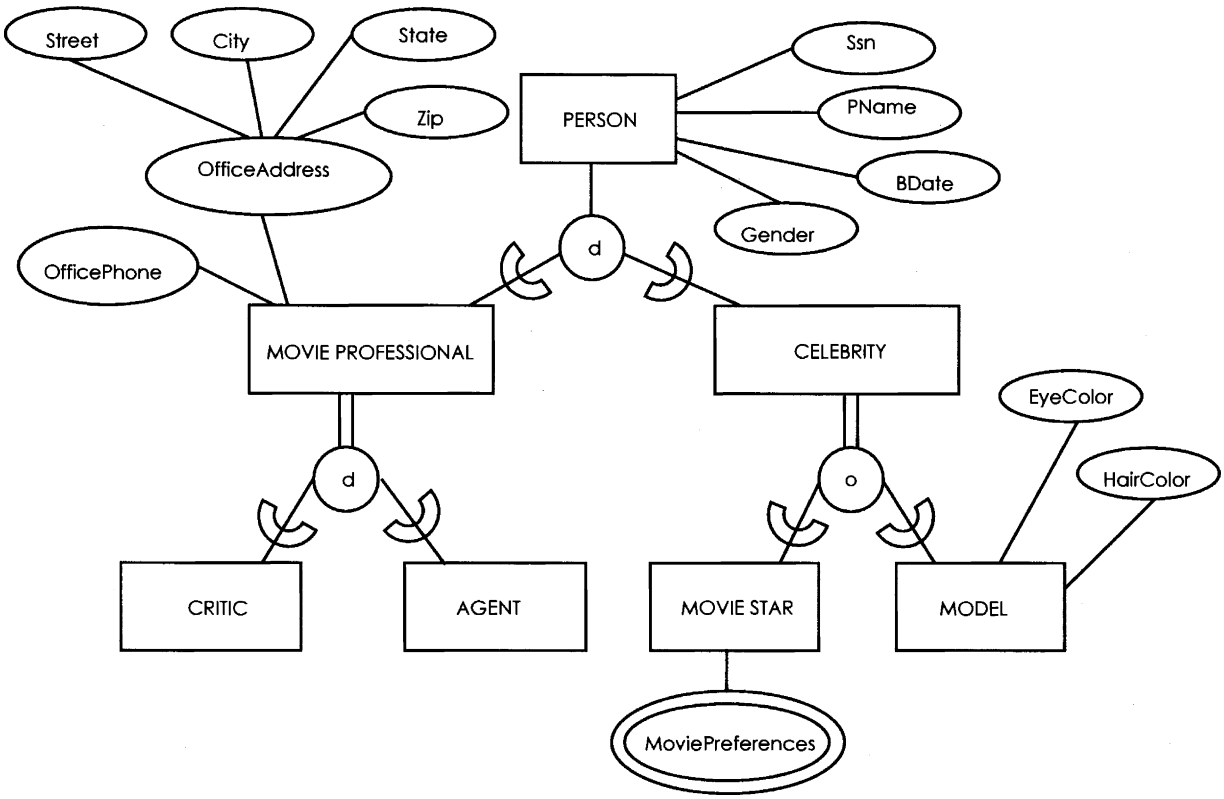


Figure 11. A superclass/subclass hierarchy in the EER model demonstrating disjoint and overlapping constraints as well as total and patial participation constraints.

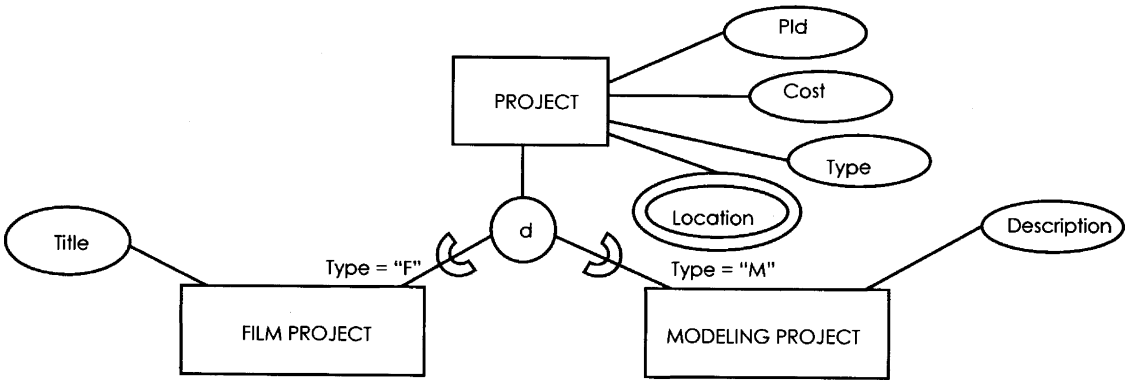


Figure 12. A superclass/subclass hierarchy in the EER model demonstrating attribute-defined subclasses.

and modeling projects, but the specialization is based on the value of the Type attribute defined within the project class. As the specialization is total, an instance of project must be an instance of one of its subclasses. If Type = "F", then a project instance can be automatically placed in the film projects class; if Type = "M", then a project instance is also an instance of modeling projects. This form of specialization is known as *predicate-defined* specialization and is indicated by placing the predicate on the appropriate path leading from the superclass to the subclass. If the attribute that is used to define the specialization is single valued, then membership at the subclass level will always be disjoint.

Multiple Inheritance. The ISA hierarchy examples presented so far illustrate the case of a subclass that inherits from only one superclass. In some cases, a subclass may need to inherit from more than one superclass. *Multiple inheritance* is often used to represent this situation. Figure 13 provides an example of modeling the star-model class as a subclass of the movie star and the model classes. The star-model class, therefore, represents the intersection of its superclasses, containing instances that are both movie stars and models. The star-model class is referred to as a *shared subclass*. As multiple inheritance represents an intersection, a common root to the hierarchy must exist, such as the celebrity class. A subclass that is defined using multiple inheritance inherits attributes along all paths that

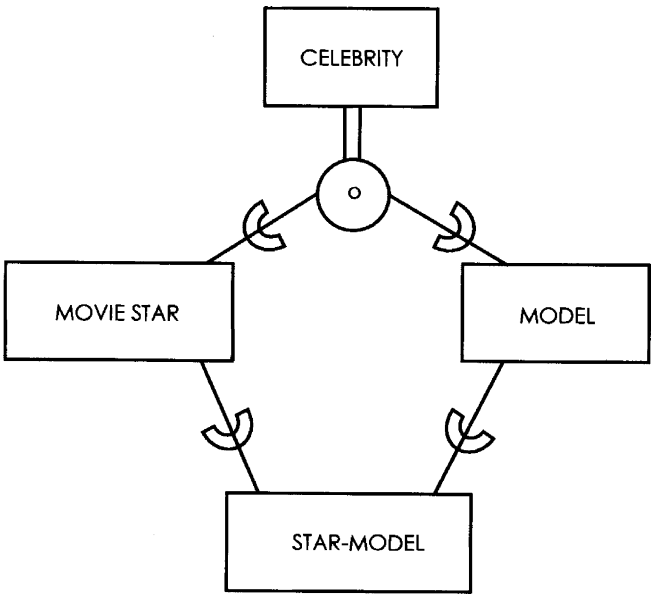


Figure 13. A superclass/subclass hierarchy in the EER model demonstrating multiple inheritance.

lead back to the root of the hierarchy.

Modeling with Categories in the EER Model. Figure 14 illustrates a modeling feature of the EER model known as categorization. Unlike multiple inheritance, categorization provides a way of creating union types, where a class represents a union of different classes. Using categorization, we can model the case where a class inherits from several superclasses, where each superclass is of a different entity type. Graphically, categorization is shown by placing a "u" in the circle that connects the superclasses to the subclass. The subclass is referred to as a *category*.

In Fig. 14, sponsor is a category that inherits from the person and company superclasses. As a modeling project can be sponsored by either a person or a company, the sponsor category provides a convenient way to model the SponsoredBy relationship. Instances of the sponsor class inherit from either person or company, depending on the actual type of the instance. Membership in the sponsor class can also be *total* or *partial*. Partial membership defines that a category is a subset of the union of its superclasses. The schema in Fig. 14 represents partial membership, which is indicated by a single line leading from the circle to the category. When a category such as sponsor has partial membership, then not every person and company instance is required to be a sponsor. Total membership defines that a category is exactly the union of its superclasses; in which case, every instance of a superclass must also be an instance of the category. Total membership is specified by placing a double line between the circle containing the "u" and the entity type that represents the category.

A Complete EER Schema Example. A complete schema example illustrating the features of the EER model is shown in Fig. 15. To simplify presentation of the schema, the only attributes shown are those on relationships as well as those needed for predicate-defined subclasses. The schema includes the person and project ISA hierarchies that have been discussed above as well as the sponsor category. In addition, the schema indicates that an agent serves as an AgentFor celebrities, where every celebrity is required to have an agent. A movie star ActsIn film projects, whereas a model ModelsIn modeling projects. The schema also captures the amount of money that each celebrity makes for the projects in which they participate. Modeling projects are SponsoredBy sponsors, where a sponsor can be either a person or a company. Each film project must be ProducedBy a studio. Studios, on the other hand, can produce many film projects. The schema also shows that a critic Critiques film projects, recording the rating and comments associated with each critique.

THE RELATIONAL DATA MODEL

Unlike the ER model, which is a DBMS-independent model, the relational model is associated with relational database management systems, one of the most widely used database software systems currently used for database implementations. Generally, a database implementation begins by describing the application using a conceptual tool such as an ER diagram. The conceptual model is then mapped to the relational model to support the actual implementation of the database. This section describes the fundamental concepts of the relational model as a DBMS implementation model. Basic concepts associated with mapping conceptual schemas to relational schemas are also presented, together with other issues related to relational schema design.

Fundamental Concepts of the Relational Model

The relational model is known for its simplicity because the *relation*, otherwise known as a *table*, is the fundamental modeling concept. A table is a two-dimensional structure consisting of *rows* and *columns*. Rows in a relation are referred to as *tuples*, whereas columns are referred to

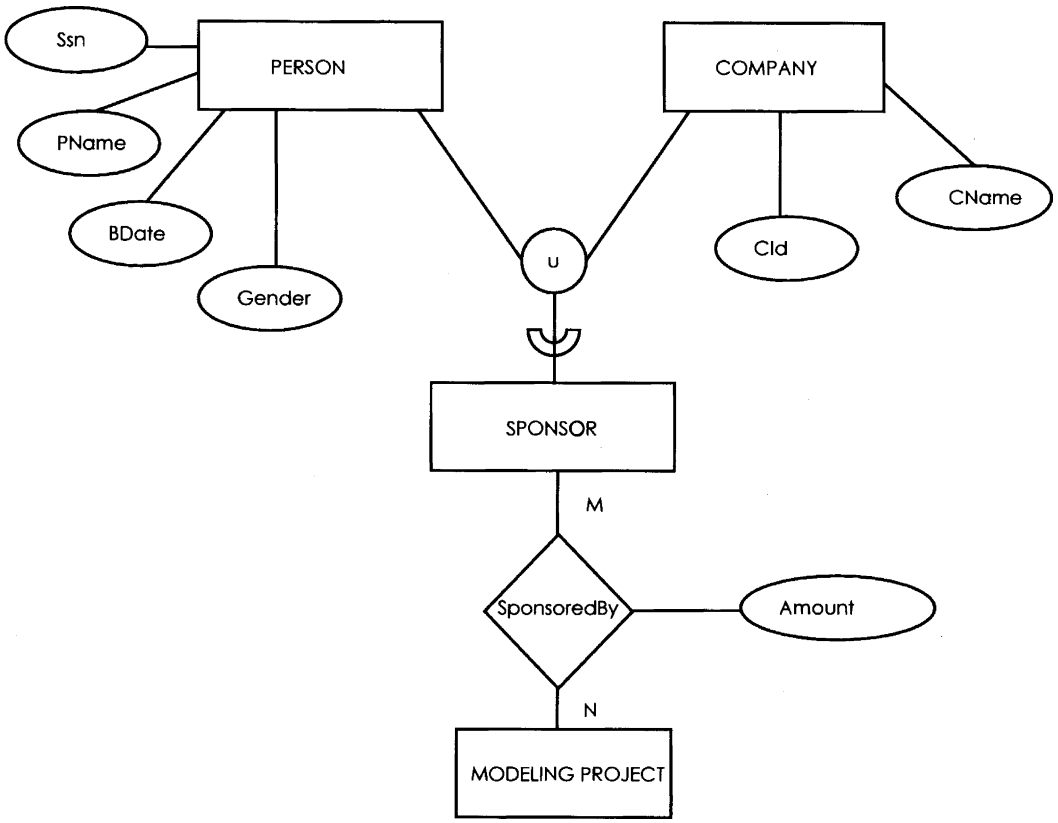


Figure 14. A category definition in the EER model, where the Sponsor entity inherits from heterogeneous entities with no common root.

as *attributes*. Each attribute can contain a value that is associated with a specific *domain*. Figure 2 has already been presented as an example of three relations associated with the relational schema shown in Fig. 1. In Fig. 2, the MOVIE-STAR relation is illustrated as a table containing three columns that represent the STAR-ID, NAME, and BDATE attributes of a movie star. Each row in the table is a tuple containing specific values for each attribute. The type of each attribute, as specified in Fig. 1, defines a domain. For example, the domain of STAR-ID is the domain of strings of length 9. The domain of BDATE, on the other hand, is a system-defined DATE type. The definition of each relation in Fig. 1 is a *relation scheme*, also known as the intension of the relation. The actual relation is the set of tuples that define the *relation instance*, also known as the extension of the relation.

More formally, the intension of a relation is $R(A_1, A_2, \ldots, A_n)$, where R is the name of the relation and each A_i is an attribute defined over a domain D. The domain D of an attribute A_i is denoted $dom(A_i)$. The *degree of a relation* is the number of attributes defined in the intension of the relation. A relation r of the intensional definition $R(A_1, A_2, \ldots, A_n)$, denoted as r(R), is the set of *n-tuples* $r = t_1, t_2, \ldots, t_n$, where an n-tuple is an ordered list of values $t = <v_1, v_2, \ldots, v_n>$. Each v_i is either a *null* value or a value from $dom(A_i)$. A relation can also be viewed as a subset of the Cartesian product of $dom(A_i)$, for all A_i that define the relation:

$$r(R) \subseteq (dom(A_1) X dom(A_2) X \ldots X dom(A_n))$$

Note that as a relation is a set, the tuples of a relation do not have any specific order.

Several types of constraints are associated with the definition of relational database schemas: *domain constraints*, *key constraints*, *entity integrity constraints*, and *referential integrity constraints*. Domain constraints are those that appear in a schema definition such as the one in Fig. 1, constraining the type of an attribute to be a value from a specified domain. Key constraints are those constraints that define unique identifiers for the tuples in a relation. By definition, the tuples of a relation must be distinct because a relation is a set. Hence, some attribute or set of attributes must serve as the key of each relation. Any set of attributes that can be used as a key of a relation is referred to as a *superkey*. In Fig. 2, for example, STAR-ID, NAME, and BDATE together can be used as a superkey to uniquely identify tuples in the MOVIE-STAR relation. Such a key is also referred to as a *composite key* because it is composed of more than one attribute.

A *key* of a relation is a superkey where removing any attribute from the set of attributes that compose the key will no longer form a superkey of the relation. A key is therefore a *minimal superkey*. STAR-ID is a key of the MOVIE-STAR relation, assuming that a person's social security number is used as the value for STAR-ID. Note that a relation can have more than one key. In this case, each key is referred to as a *candidate key*. One candidate key, however, is typically selected to be used as the *primary key* (i.e., the key that will generally be used to identify tuples in the relation). Pri-

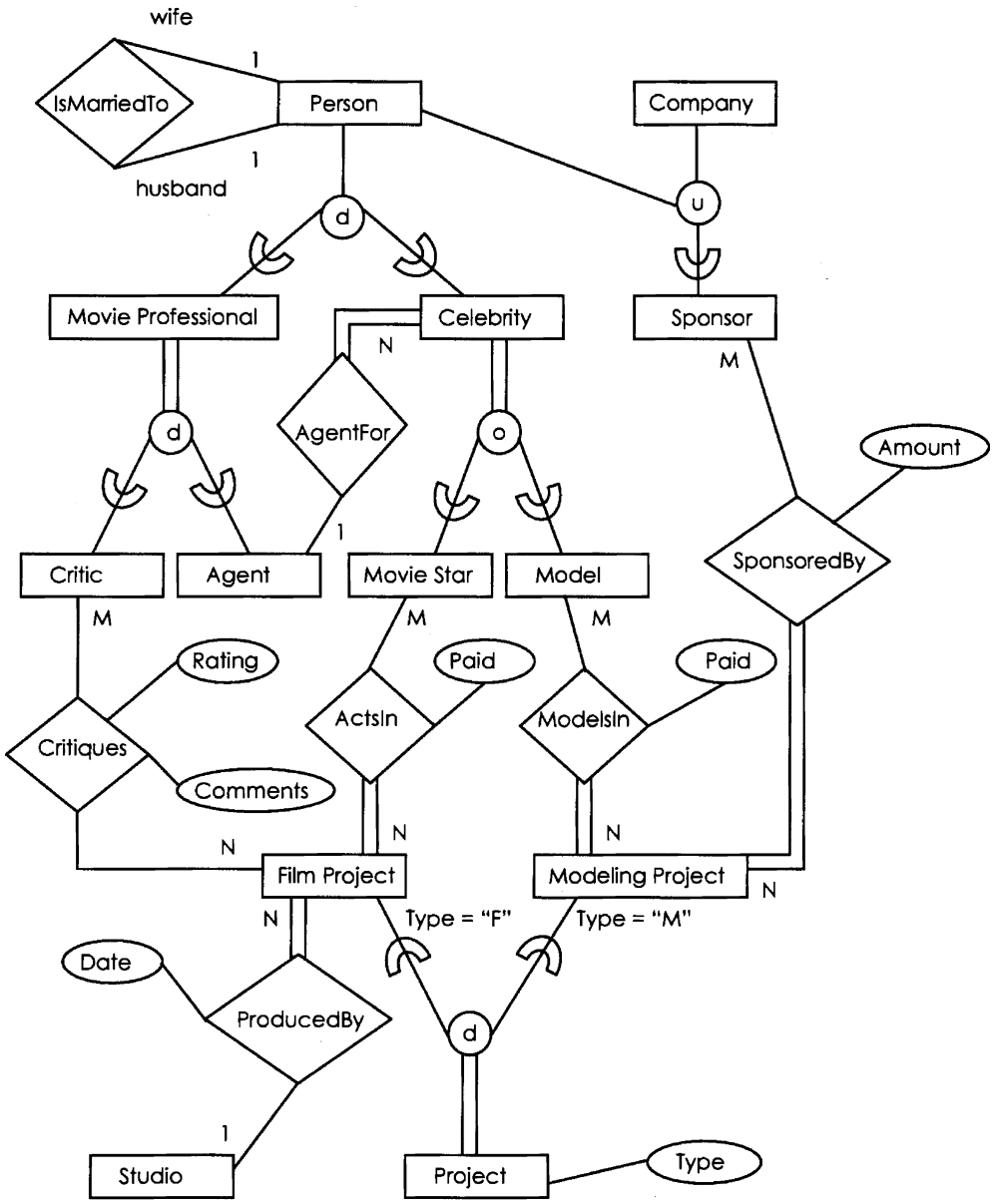


Figure 15. An EER schema illustrating the use of ISA hierarchies and a category in relationships with other entities.

mary keys are always identified by underlining the names of the attributes forming the primary key. Associated with the key constraint is the entity integrity constraint, which states that no primary key value can be null.

Referring again to Fig. 2, we can see that the relation ACTS-IN is composed of three attributes: STAR-ID, FILM-ID, and INCOME. The key of the relation is the composite key composed of STAR-ID and FILM-ID. The domains of STAR-ID and FILM-ID in the ACTS-IN relation are the same as the domains of STAR-ID and FILM-ID in the MOVIE-STAR and FILM-PROJECT relations, respectively. Furthermore, the STAR-ID and FILM-ID attributes in ACTS-IN represent the same concept as they do in STAR and FILM-PROJECT. The referential integrity constraint defines that values such as those for STAR-ID and FILM-ID in the ACTS-IN relation *must* refer to tuples that already exist in MOVIE-STAR and FILM-PROJECT. In other words, the MOVIE-STAR and FILM-PROJECT relations serve as the source for defining valid MOVIE-STAR and FILM-PROJECT tuples. Any other relation that needs to refer to a MOVIE-STAR or a FILM-PROJECT must refer to a valid value from one of these relations.

Attributes such as those in ACTS-IN that refer to values from another relation are referred to as *foreign keys*. In general, the value of a foreign key can be null, unless the foreign key is also serving as a candidate key as in

the ACTS-IN relation. Foreign keys are used to represent relationships between objects.

Mapping From Conceptual Models to the Relational Model

One way of generating a relational schema is to first develop a conceptual model using an entity-relationship approach. The conceptual schema can then be mapped to a relational schema. This section describes the fundamental concepts of mapping from ER and EER schemas to relational schemas. Enforcement of constraints from a conceptual schema in a relational design is also addressed.

Basic ER Mapping Concepts. To illustrate mapping procedures for the ER model, Fig. 16 presents a relational schema that was generated from the ER diagram in Fig. 6. Figure 17 presents an example of a database instance of the schema in Fig. 16.

As illustrated in Fig. 16, entities are mapped to relations by establishing a relation for every entity that exists in an ER diagram. All of the simple attributes (i.e., non-multivalued attributes) of the entity are included as attributes in the relation. A composite attribute is represented in terms of its components because of the restriction that attributes must be atomic. The key of the entity is identified as the key of the relation. In Fig. 16, for example, relations are generated for the movie star, film project, and studio entities. The MOVIE-STAR relation includes the STAR-ID, FIRST, INITIAL, LAST, and BDATE attributes. Notice that the Name attribute from the ER schema is represented in terms of its subcomponents in the MOVIE-STAR relation. The Age attribute is not included as an attribute in the relation because Age is defined to be a derived attribute. Weak entity types, such as shooting schedule, are mapped in a similar manner except that the key of a weak entity is formed by combining the primary key of the identifying owner with the partial key of the weak entity. As a result, FILM-ID and SCENE-NUM form a composite primary key for the SHOOTING-SCHEDULE relation.

To map 1:1 binary relationships, first find the relations of the entity types involved in the relationship from the ER diagram. One relation will be identified for receiving the key of the other relation as a foreign key, which thus establishesa relationship between the two entities. In Fig. 6, the only 1:1 relationship is the IsMarriedTo recursive relationship between movie stars. In this case, the MOVIE-STAR relation includes the IS-MARRIED-TO attribute to store the key of another tuple in the MOVIE-STAR relation, which thus identifies the wife or husband of the star, who also happens to be a movie star. STAR-ID and IS-MARRIED-TO should be defined as the same domains. IS-MARRIED-TO is a foreign key that must contain valid values that appear in the STAR-ID column of the relation.

In general, 1:1 relations can exist between two separate relations as shown by the ER diagram in Fig. 18. In this example, a studio is managed by one manager and a manager can manage only one studio. In this mapping, the key of the MANAGER relation is included as a foreign key in the STUDIO relation. As an alternative mapping, the key of the STUDIO relation could have been included in the MANAGER relation. The mapping shown in Fig. 18 is better, however, because the ER diagram states that every studio must have a manager. If one entity in the relationship has total participation, then the corresponding relation should be chosen as the relation to receive the foreign key.

There are two alternative mappings for binary 1:N relationships exist. One mapping approach is illustrated in the FILM-PROJECT relation of Fig. 16 in which the key of the relation from the "1" side of the relationship is included in the relation from the "N" side of the relationship. In this case, a film project is produced by one studio; a studio produces many film projects. The SID from STUDIO is included as a foreign key in FILM-PROJECT to capture the relationship. Notice that the DATE attribute of the relationship is also included.

The alternative mapping approach for 1:N relationships is to create a separate relation to represent the relationship. The relation should include the keys of the relations for each entity involved as well as any attributes of the relationship. The key of the relation from the "N" side of the relationship is used as the key of the new relation. Using this approach, the FILM-PROJECT and STUDIO relations in Fig. 16 can be replaced with the following relations:

```
FILM-PROJECT: FILM-ID, TITLE
STUDIO: SID, SNAME
PRODUCED-BY: FILM-ID, SID, DATE
```

For M:N binary relationships, the only option is to create a separate relation to represent the relationship. The new relation includes the primary keys of the relations that represent the entity types involved as well as any attributes of the relationship. The primary keys of each relation are combined to create the key for the new relation. Figure 16 uses the ACTS-IN relation to represent the M:N relationship between movie stars and film projects shown in the ER diagram of Fig. 6. The composite key of the relation is the concatenation of STAR-ID and FILM-ID.

Multi-valued attributes such as the PhoneNumber attribute of the studio entity in Fig. 6 are mapped in a manner similar to M:N relationships. As illustrated in Fig. 16, the new relation STUDIO-PHONE is created. The primary key of the relation is the composite key formed through the concatenation of the SID primary key of STUDIO and the PHONE-NUMBER attribute. If a studio has three different phone numbers, there will be three different tuples in the relation to record each phone number. Figure 17 shows that Hollywood Studios has three phone numbers, whereas Star Studios has two phone numbers.

The final mapping feature to consider for basic ER concepts involves N-ary relationships. N-ary relationships are generally mapped in the same manner as M:N relationships. Create a relation to represent the relationship and include the primary keys of the entities involved. The primary key of the new relation is the composite key formed by all primary keys. If any entity has a "1" as a cardinality for participation in the relationship, then the key of such an entity can be used as the primary key for the N-ary relationship.

MOVIE-STAR: STAR-ID, FIRST, INITIAL, LAST, BDATE, IS-MARRIED-TO

FILM-PROJECT: FILM-ID, TITLE, SID, DATE

ACTS-IN: STAR-ID, FILM_ID, INCOME

STUDIO: SID, SNAME

STUDIO-PHONE: SID, PHONE-NUMBER

SHOOTING-SCHEDULE: FILM-ID, SCENE-NUM, LOCATION, START-DATE

Figure 16. Relational schema generated from the ER schema in Fig. 6.

MOVIE-STAR:

STAR-ID	FIRST	INITIAL	LAST	BDATE	IS-MARRIED-TO
123456789	George	M	Johns	06/16/55	333445555
222334444	Murray	H	Hill	09/02/51	
333445555	Ann	T	Dugan	10/31/65	123456789
444556666	Elizabeth	U	Jillian	08/23/87	
555667777	Joseph	J	Andrew	02/18/82	
666778888	Joseph	P	Edward	09/02/51	
777889999	Christine	S	Darling	02/15/22	888990101
888990101	William	A	Edward	04/15/21	777889999

FILM-PROJECT:

FILM-ID	TITLE	SID	DATE
11111	California Dreaming	123	1998
22222	Louisiana Saturday Night	124	2000
33333	Valley of the Sun	123	1999

ACTS-IN:

ACTOR-ID	FILM_ID	INCOME
123456789	22222	2,000,000
222334444	33333	1,000,000
333445555	11111	2,000,000
444556666	22222	3,000,000
555667777	33333	2,000,000
666778888	11111	3,000,000
666778888	22222	3,000,000
777889999	33333	2,000,000
777889999	11111	2,000,000
888990101	22222	3,000,000

STUDIO:

SID	SNAME
123	Star Studios
124	Hollywood Studios

STUDIO-PHONE:

SID	PHONE-NUMBER
123	444-220-5555
123	444-220-5556
123	444-220-5557
124	443-321-1234
124	443-321-1235

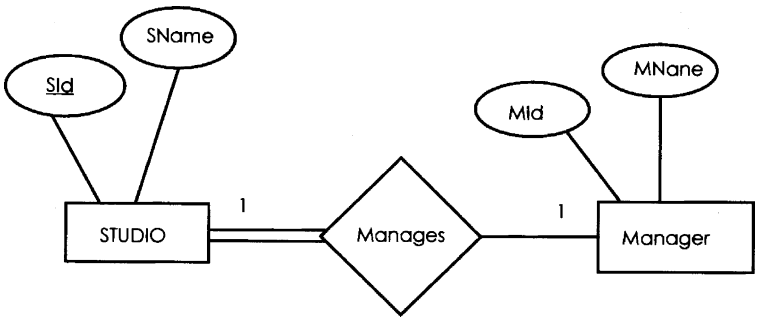
SHOOTING-SCHEDULE:

FILM-ID	SCENE-NUM	LOCATION	START-DATE
1111	1	Los Angeles	01/23/1997
1111	2	San Diego	02/15/1997
1111	3	San Jose	03/01/1997
2222	1	Lafayette	05/16/1998
2222	2	New Iberia	06/01/1998
2222	3	Opelousas	07/04/1998
3333	1	Phoenix	01/15/1998
3333	2	Flagstaff	02/01/1998
3333	3	Grand Canyon	02/15/1998

Figure 17. Tables illustrating the extension of the relational schema in Fig. 16.

Advanced Mapping Concepts for Enhanced Entity-Relationship Schemas. Mapping procedures for the EER model are the same as for the ER model with the addition of new procedures for mapping ISA hierarchies and categories to relations.

Three different approaches for mapping ISA hierarchies are illustrated in Fig. 19 using the ISA hierarchy from Fig. 12. The most straightforward approach is to create a separate relation for each entity in the hierarchy as shown in mapping 1. Each relation must include the key of the relation from the top of the hierarchy. Mapping 2 illustrates that a separate relation is created for the entities at the bottom of the hierarchy only, where each relation includes the attributes of the entity at the bottom of the hierarchy



STUDIO: SID, SNAME, MID

MANAGER: MID, MNAME

Figure 18. An example of a 1:1 mapping for the Manages relationships.

Mapping 1: A Separate Relation for Each Entity
PROJECT: PID, COST, TYPE
FILM-PROJECT: PID, TITLE
MODELING-PROJECT: PID, DESCRIPTION

Mapping 2: Relations for Subclasses Only
FILM-PROJECT: PID, COST, TYPE, TITLE
MODELING-PROJECT: PID, COSE, TYPE, DESCRIPTION

Mapping 3: One Relation for the Flattened Hierarchy
PROJECT: PID, COST, TYPE, TITLE, DESCRIPTION, IS-FILM-PROJECT, IS-MODELING-PROJECT

Figure 19. Three different mapping techniques for mapping an ISA-hierarchy in an EER diagram to a relational schema.

and the attributes of all superclasses. This approach produces duplicate representation of inherited attributes. In the case of disjoint classes as in Fig. 12, the duplication does not cause a problem because an entity is either a film project or a modeling project, but not both. This approach would not be as useful if film project and modeling project were overlapping. Also, no explicit representation exists of the entities in the project class with this approach. The final mapping approach is mapping 3 in which all attributes of the hierarchy are flattened into one relation. Additional Boolean-valued attributes, represented by the IS-FILM-PROJECT and IS-MODELING-PROJECT attributes, are added to the relation to indicate the subclasses in which the objects participate. This approach is useful for overlapping subclasses but will waste storage space for disjoint subclasses. In general, a relational mapping for a complex ISA hierarchy can involve any combination of the above techniques.

A category such as the one in Fig. 14 is mapped by creating the relation SPONSOR to represent the category and defining the SPONSOR-ID attribute as an identifier for tuples in the SPONSOR relation. The SPONSOR-ID attribute must also be added to the PERSON and COMPANY relations as a foreign key. An example of the mapping is shown in Fig. 20.

Enforcing Constraints of the Conceptual Schema. As conceptual modeling techniques such as the EER model provide a more expressive way to describe applications than that provided by a relational schema, the semantics of the original schema are often lost in the translation process. The application developer for the relational implementation must therefore be aware of the semantics of the conceptual schema so that the appropriate constraints can be enforced in the code that accesses the relational database. Constraints from a conceptual schema that directly translate to relational schemas are key constraints, some domain constraints, and referential integrity constraints. Other constraints such as total participation constraints, total specialization constraints, disjoint constraints, predicate-defined subclasses, and the semantics of weak entities and categories must be implemented by the relational database developer in the application code that access the database to enforce the semantics of the conceptual schema.

OBJECT-BASED MODELS

The previous sections have presented the fundamental concepts associated with conceptual data models and relational database modeling techniques. Database technology, however, has continually adapted to the needs of more complex, data-centric applications. This continual evolution, and sometimes revolution, in database systems has resulted in the development of database models that have incorporated object-oriented concepts into the modeling and implementation framework. This section provides an overview of object-based data models, including the object-oriented data model and the object-relational data model. Most data modeling concepts described in this section are associated with specific types of database management systems that are described in other articles that appear in this volume. Readers should refer to the appropriate articles for more in-depth coverage.

PERSON: SSN, PNAME, BDATE, GENDER, SPONSOR-ID

COMPANY: CID, CNAME, SPONSOR-ID

SPONSOR: SPONSOR-ID

Figure 20. Mapping from a category in the EER model to a relational schema.

The Object-Oriented Data Model

Object-oriented database management systems were developed as an alternative to relational database systems for the support of applications that required the storage of large objects, the definition of user-defined data types, and the representation of complex relationships between objects (37). Database technology was traditionally applied to business-oriented applications that were well suited to the table representation provided by relational database systems. As the use of database technology expanded to applications such as software engineering environments, mechanical and electrical engineering design, cartography, manufacturing, and medical applications, to name a few, researchers began to look for more sophisticated ways to model and store the data needed by such applications. Normalized relational tables were deemed to be an inadequate modeling and storage approach.

In the development of OODBMSs, researchers were also in search of a solution to the *impedance mismatch* problem associated with relational database systems. Impedance mismatch refers to the data structure differences that exist between database systems and programming languages such as C and C++. In particular, the access of data from a relational database application requires that data be retrieved from relations, the primary data structure of relational database systems, and transformed into the appropriate data structures of the language used to access the database. In the opposite direction, data from programming language data structures must be transformed into relations for storage in a relational database. In short, relational databases and programming languages view data in different ways. OODBMSs provide a tighter integration between programming languages and database concepts, which thus eliminates the impedance mismatch problem and enhances database technology with a computationally complete programming language.

Unlike the relational model, commercial OODBMSs began to appear on the market before any formal, standardized object-oriented model had been defined. Documents such as the Object-Oriented Database System Manifesto (30) have helped to shape general agreement about what constitutes an object-oriented data model. Efforts such as that of the Object Data Management Group (ODMG) (28) have defined a de facto standard among OODBMS vendors in cooperation with the Object Management Group (OMG) standard for interoperability (34). The purpose of the standard is to promote software portability so that applications can easily run on different OODBMS products. The standard includes:

1. An Object Model based on the object model of the OMG standard.
2. The Object Definition Language (ODL) for the specification of object schemas.
3. The Object Query Language (OQL), which is an object-oriented extension of SQL for querying object databases, and
4. The language bindings for C++, Smalltalk, and Java, defining language-specific issues for the implementation of portable, object-oriented database applications.

Similar to concepts that have already been presented for conceptual data models, the object model of the ODMG standard views the world in terms of *objects*. More complex objects are constructed from simpler objects by using *constructors* such as tuples, sets, lists, bags, and arrays. Constructors can be applied to any type of object, even those objects associated with *user-defined types*. Objects are assumed to have *object identifiers* (OIDs). An OID is a system-generated, internal identifier that is used to represent the existence of an object. No two objects can have the same OID. Furthermore, OIDs are used to establish relationships between objects rather than using external key values as in the relational model. *Literals* are printable values that do not have OIDs.

In the ODMG standard, an object is an instance of a specific *class*. The term "class" is used in different ways in different OODBMS products. In some systems, a class is used to refer to a type as in the traditional programming language interpretation of a type. In this case, the type defines the structure and behavior of the object, but it is the user's responsibility to manage objects within variables of that type. In other systems, a class is viewed in the traditional database sense. In this case, a class not only defines the structure and behavior of objects, but also provides an automatic collection of all instances of the type. In ODMG terminology, this collection is referred to as an *extent*. This view of a class as the database extension is better suited to the support of object-oriented query languages. The ODMG standard allows a class to be defined with or without an extent.

ODL is used to specify class definitions, where in addition to specifying a possible extent for a class, each class also defines the state of an object in terms of *properties*. Each property can either be an *attribute* that describe an object or a *relationship* with other objects. Classes are also used to define the *operations* that can be performed on objects. As with the conceptual models described earlier in this article, classes can be organized into superclass/subclass hierarchies, where a subclass inherits the properties *and* operations defined for its superclasses. A class corresponds to the notion of an abstract data type, which defines an *interface* that is visible to users of the type and defines an *implementation* that is only visible to database designers. The definition of operations on objects is a significant departure from the modeling approaches presented earlier in this article. The object-oriented model therefore emphasizes the structure *and* behavior of data,

whereas traditional database modeling approaches emphasize structural features only.

As an example, consider the studio class, the project class, the film project class, and the ProducedBy relationship in Fig. 15. The corresponding ODL class definitions are shown in Fig. 21. Each class specifies a class name and the name of an extent that will be used as the collection of all instances of the class. As a film project is a subclass of project, the film project class is also defined to *extend* a project. As a result, every film project object will inherit properties and operations defined at the project level. Furthermore, each instance of film project is also considered to be an instance of project.

Attribute specifications define the state of each class. For example, a project has a projectID, a type (film project or modeling project), and a location; a film project has a title; and a studio has a studioID, a studioName, and a set of phoneNumbers. An attribute can also be identified as a key of a class, where a key has the same meaning as in the relational model. Even though the object-oriented model uses internal object identifiers for each object instance, the use of keys is still an important application design feature.

In addition to attribute definitions, film project and studio have inverse relationships that define the producedBy 1:N relationship. As a studio is related to many film projects, the relationship in studio is defined as a set of film project objects. In the opposite direction, a film project defines the relationship to contain only one studio object. Notice that the date on the ProducedBy relationship is included in the list of attributes for film project. This representation is consistent with the 1:N mapping procedure described for the relational model (i.e., the object on the 1 side and any attributes of the relationship are included in the definition of the object on the N side of the relationship). Furthermore, each relationship definition is specified to be the inverse of the other. In the ODMG model, the specification of an inverse on a relationship definition is required. In many OODBMS implementations, inverse definitions are automatically maintained. As a result, assigning a studio to a film project will automatically invoke the inverse relationship assignment of a film project to the set of projects produced by a studio.

The film project class in Fig. 21 also provides an example of the assignStudio operation to assign a studio, together with an assignment date to a film project. In general, operations can be defined to create and delete objects, to assign values to attributes and relationships, or to perform application-specific functions and enforce application constraints.

For additional details on mapping conceptual models to the ODMG model as well as detailed coverage of the Object Query Language, see Reference 35.

The Object-Relational Model

In response to the development of object-oriented database systems, the relational database community has been involved with incorporating object-oriented extensions into relational database technology, creating object-relational database systems. Seminal work in the area of object-relational models was described by Michael Stonebraker in association with the Postgres data model (31), which is an extended version of the Ingres relational database system. Stonebraker also organized a group of relational database researchers in the preparation of The Third Generation Database System Manifesto (29). This document was prepared in response to (30) to describe the manner in which relational systems can be extended with features typically associated with object-oriented database systems and still retain the advantages of relational technology. Today, several relational database vendors provide object-relational features as part of their relational database products, including Oracle and IBM's DB2 system. The SQL standard has also been updated to include the definition of object-relational features. This section provides a brief overview of object-relational modeling concepts using the SQL standard.

The SQL object-relational model provides a means to create user-defined types (UDTs). A UDT is the same as an abstract data type, defining the internal structure of a data type together with operations that define the behavior for access and use of the type. A UDT can be used in one of two ways. In one approach, tables of UDTs, known as typed tables, can be created to simulate the notion of a class in an object-oriented model. Objects are then created as instances of typed tables, where objects have object identifiers. In the other approach, UDTs are used as structured types, where the type definition is used directly as the type of an attribute in an object table or in a traditional relational table. In this case, objects do not have object identifiers but are stored directly inside of other objects or tuples.

Figure 22 provides an example of UDTs and their corresponding typed table definitions for the project, film project, and studio classes from Fig. 15. The locationUDT type provides an example of a structured type, defining the city, state, and country of a project location. The locationUDT type is used as the type of the location column in the projectUDT type definition. An instance of locationUDT is a structured object, without an object identifier, that is embedded inside of each instance of a projectUDT type value.

In contrast, the projectUDT type is a user-defined type that is defined specifically for the purpose of creating a typed table. The projectUDT type defines columns for the projectID, type, and location of a project. The type definition concludes with the statement of three clauses. The instantiable clause defines that a constructor function is available for the type. A type can also be specified as not instantiable. The not instantiable clause would be used in the case of supertype and subtype definitions, where the type is to be instantiated at the subtype level (i.e., instantiable) and not at the supertype level (i.e., not instantiable). The second clause used in the projectUDT definition is the not final clause. This clause defines that the type is capable of having a subtype definition. Finally, the ref is system-generated clause indicates that, if the type is used to define a typed table, then the database system will automatically generate object identifiers for each object that is created as an instance of the typed table. Other options exist for the creation of object identifiers, where the value of the object identifier is user defined or based on a key value.

```
class Project
(       extent projects
        key projectID)
{
  attribute string projectID;
  attribute string type;
  attribute string location;

};

class FilmProject extends Project
(       extent filmProjects)
{
  attribute string title;
  attribute string dateProduced;

  relationship Studio producedBy
        inverse Studio::hasFilmProjects;

  void assignStudio(in Studio studioObj, in string assignDate);
};

class Studio
(       extent studios
        key studioID)
{
  attribute string studioID;
  attribute string studioName;
  attribute set<string> phoneNumbers;

  relationship set<FilmProjects> hasFilmProjects
        inverse FilmProject::producedBy;
};
```

Figure 21. An Object Definition Language example for a portion of Fig. 15.

```
create type locationUDT as
( city        varchar(30),
  state       varchar(2),
  country     varchar(5))
  not final;

create type projectUDT as
( projectID   varchar(6),
  type        varchar(2),
  location    locationUDT)
  instantiable not final ref is system generated;

create table project of projectUDT
( primary key (projectID),
  ref is projOID system generated);

create type filmProjectUDT under projectUDT
( title                 varchar(30),
  dateProduced          date,
  producedBy            ref (studioUDT))
  not final
  method assignStudio(studioObj studioUDT,  assignDate date);

create table  filmProject of filmProjectUDT under project;

create type studioUDT
( studioID              varchar(6),
  studioName            varchar(30),
  phoneNumbers          varchar(12) array(3),
  hasFilmProjects       ref (filmProjectUDT) array(1000))
  instantiable not final ref is system generated;

create table studio of studioUDT
( primary key (studioID),
  ref is system generated);
```

Figure 22. An object-relational schema example for a portion of Fig. 15 using language features of the SQL standard.

The projectUDT type definition is followed by a table definition for project, where project is defined to be a typed table containing objects of type projectUDT. The project table will have a column for every attribute defined in the projectUDT type. The definition of the project typed table also includes the definition of projectID to be a key for the table. In addition, the table will have an automatically generated column for the object identifier of each row in the table. The name projOID is used to refer to the value of the object identifier.

Similar UDT and typed table definitions exist in Fig. 22 for the film project typed table and the studio typed table. Notice that filmProjectUDT is defined as "under projectUDT," which defines the filmProjectUDT to be a subtype of projectUDT. In a similar manner, the filmProject table is defined to be "under project." These definitions collectively define an inheritance hierarchy, with project as the superclass and filmProject as the subclass. Inheritance hierarchies in the object-relational model follow similar semantics as in the object-oriented model, where classes at the subclass level inherit attributes and operations from the superclass level. The filmProjectUDT also illustrates the assignStudio method specification, which indicates that user-defined types in the object-relational model have the same capabilities as in the object-oriented model for defining the behavior of an object.

To represent relationships, an object in one typed table can refer to an object in another typed table (or the same typed table) through the use of an object reference (or ref). In this way, relations are extended with the capability to store a pointer to an object as an attribute value, rather than storing an external key value as a foreign key. The use of a ref simplifies the expression of relational queries because path expressions can be used to traverse object references instead of explicitly specifying join conditions. Figure 22 illustrates the use of refs to define the producedBy relationship between the studio and filmProject typed tables. The filmProjectUDT defines producedBy to be a ref to an object of type studioUDT. In the inverse direction, the studioUDT defines hasFilmProjects to be a ref to a collection of objects of type filmProjectUDT. The collection is indicated by following the ref definition with an array specification, which thus provides a means to define a column that contains multiple values. Arrays of refs are useful for directly defining 1:N and M:N relationships between classes in a conceptual model such as the EER model.

As relational database systems currently hold a greater share of the database market than object-oriented database systems, the evolutionary approach offered by object-relational technology has an advantage over pure object-oriented database technology. However, more sophisticated object modeling tools are required that support the definition of user-defined types and operations in the modeling process, with mapping procedures that capitalize on object-relational features. The work in Reference 35 provides more detailed coverage of the object-relational features of the SQL standard and provides case studies in Oracle.

SUMMARY

This article has presented fundamental concepts of database modeling using conceptual models such as the entity-relationship model and the enhanced entity-relationship model, as well as implementation models, such as the relational model, the object-oriented database model, and the object-relational database model. One of the more recent directions for data modeling has been introduced as a result of the widespread use of the World Wide Web. The Web provides individuals with access to a multitude of information, some of which is stored in traditional database systems and some of which is stored in the form of XML documents. XML has become a standard on the Web for representing data together with its structural description (38). XML documents provide an excellent tool for data exchange, but they are also being used as a storage alternative to traditional database systems, with a significant amount of research focused on querying XML files. XMLSchema provides a way to define the structure, content, and semantics of XML documents (39), with many of the modeling features of XMLSchema based on the modeling concepts presented in this article. In related efforts, researchers are defining languages such as RDF (40), RDF-Schema (41), and the Web Ontology Language (OWL) (42) to express ontologies for domain-specific vocabularies that can be used to enhance the understanding of database schemas and XML files. Readers should refer to the appropriate articles in this volume for more specific details about developing work in this area.

BIBLIOGRAPHY

1. R., Elmasri; S. B., Navathe *Fundamentals of Database Systems*, 5th ed.; Benjamin Cummings: Redwood City, 2006.
2. P., Chen The Entity-Relationship Model: Toward a Unified View of Data. *ACM Trans. Database Syst.* 1976, **1**(1).
3. E. F., Codd A Relational Model for Large Shared Data Banks. *Commun. ACM* 1970, **13**(6).
4. J., Melton; A. R., Simon *SQL:1999: Understanding Relational Language Components*; Morgan Kaufmann: San Francisco, CA, 2001.
5. D., Tsichritzis; A., Klug *The ANSI/X3/SPARC DBMS Framework*; AFIPS Press: Arlington, VA, 1978.
6. C., Bachmann; S., Williams A General Purpose Programming System for Random Access Memories; *Proc. of the Fall Joint Computer Conference*; 1964.
7. *Data Description Language Journal of Development*; Canadian Government Publishing Centre: Ottawa, Ontario, Canada, 1978.
8. W., McGee The Information System Management System IMS/VS, Part I: General Structure and Operation. *IBM Syst. J.* 1977, **16**(2).
9. D., Chamberlin, *et al*., A History and Evaluation of System R. *Commun. ACM* 1981, **24**(10).
10. M., Stonebraker *The Ingres Papers*; Addison-Wesley: Reading, MA, 1986.
11. J. M., Smith; D. C., Smith Database Abstractions: Aggregation and Generalization. *ACM Trans. Database Syst.* 1977, **2**(2), pp 105–133.

12. M., Hammer; D., McLeod Database Description with SDM: A Semantic Data Model. *ACM Trans. Database Syst.* 1980, **6**(3).
13. S., Abiteboul; R., Hull IFO: A Formal Semantic Database Model. *ACM Trans. Database Syst.* 1987, **12**(4), pp 525–565.
14. S., Su A Semantic Association Model for Corporate and Scientific-Statistical Databases. *Inform. Sci.* 1983, **29**.
15. M. L., Brodie; D., Ridjanovic On the Design and Specification of Database Transactions. In *On Conceptual Modeling*; Springer-Verlag: New York, 1984.
16. J. R., Abrial Data semantics. In *Data Base Management*; North Holland: Amsterdam, 1974; pp 1–59.
17. R., Hull; R., King Semantic Database Modelling: Survey, Applications, and Research Issues. *ACM Comput. Surv.* 1987, **19**(2), pp 201–260.
18. J., Peckham; F., Maryanski Semantic Data Models. *ACM Comput. Surv.* 1988, **20**(3).
19. P., Scheuermann; G., Schiffner; H., Weber Abstraction Capabilities and Invariant Properties Modeling with the Entity-Relationship Approach; *Proc. of the Entity-Relationship Conference*; 1979.
20. C., Dos Santos; E., Neuhold; A., Furtado A Data Type Approach to the Entity-Relationship Model; *Proc. of the Entity-Relationship Conference*; 1979.
21. T., Teorey; D., Yang; J., Fry A Logical Design Methodology for Relational Databases using the Extended Entity-Relationship Model. *ACM Comput. Surv.* 1986, **18**(2).
22. M., Gogolla; U., Hohenstein Towards a Semantic View of an Extended Entity-Relationship Model. *Trans. Database Syst.* 1991, **16**(3).
23. R., Elmasri; J., Weeldreyer; A., Hevner The Category Concept: An Extension to the Entity-Relationship Model. *Int. J. Data Knowl. Eng.* 1985, **1**(1).
24. H., Sibley; L., Kerschberg Data Architecture and Data Model Considerations; *Proc. of the National Computer Conference*. 785–96, 1977.
25. D., Shipman The Functional Data Model and the Data Language DAPLEX. *ACM Trans. Database Syst.* 1981, **6**(1), pp 140–173.
26. L., Fegaras; D., Maier Towards an Effective Calculus for Object Query Languages; *ACM SIGMOD Conference*; 1995, 47–58.
27. T., Leung, *et al*. The Aqua Data Model and Algebra; *Proc. of the Fourth International Workshop on Database Programming Languages*, 157–175, 1993.
28. R., Cattell, *et al*. *The Object Database Standard: ODMG 3.0*; Morgan Kaufmann: San Francisco, CA, 2000.
29. M., Stonebraker, *et al*. Third-Generation Database System Manifesto. *ACM SIGMOD Record* 1990, **19**(3).
30. M., Atkinson, *et al*. The Object-Oriented Database System Manifesto; *First International Conference on Deductive and Object-Oriented Databases*; Elsevier: New York, 1989.
31. M., Stonebraker; L., Rowe The Design of Postgres; *Proc. of the ACM SIGMOD Conference*; 1986.
32. M., Stonebraker; D., Moore *Object Relational DBMSs: The Next Great Wave*; Morgan Kaufmann: San Francisco, CA, 1995.
33. J., Rumbaugh; I., Jacobson; G., Booch *The Unified Modeling Language Reference Manual*; Addison-Wesley: Reading, MA, 1999.
34. Object Management Group, Home Web Site, 2007 http://www.omg.org.
35. S., Dietrich; S., Urban *An Advanced Course in Database Systems: Beyond Relational Databases*; Prentice Hall, Englewood Cliffs, NJ, 2005.
36. B., Stroudstrup *The C++ Programming Language, Second Edition*; Addison-Wesley: Reading, MA, 1992.
37. R. G. G., Cattell *Object Data Management: Object-Oriented and Extended Relational Systems*; Addison-Wesley: Menlo Park, CA, 1994.
38. Extensible Markup Language (XML), 2007 http://www.w3.org/XML.
39. XMLSchema, 2007 http://www/w3/org/XML/Schema.
40. Resource Description Framework (RDF), 2007 http://www/w3/org/RDF.
41. RDF Schema, 2007 http://www.w3.org/TR/rdf-schema.
42. OWL Web Ontology Language, 2007 http://www.w3.org/TR/owl-features.

SUSAN D. URBAN
Arizona State University