Associative (Composite) Entities

In the original ERM described by Chen, relationships do not contain attributes. You should recall from Chapter 3 that the relational model generally requires the use of 1:M relationships. (Also, recall that the 1:1 relationship has its place, but it should be used with caution and proper justification.) If M:N relationships are encountered, you must create a bridge between the entities that display such relationships. The associative entity is used to implement a M:N relationship between two or more entities. This associative entity (also known as a composite or bridge entity) is composed of the primary keys of each of the entities to be connected. An example of such a bridge is shown in Figure 23. The Crow's Foot notation does not identify the composite entity as such. Instead, the composite entity is identified by the solid relationship line between the parent and child entities, thereby indicating the presence of a strong (identifying) relationship.

Table name: STUDENT — Database name: Ch04_CollegeTry

STU_NUM	STU_LNAME
321452	Bowser
324257	Smithson

Table name: ENROLL

CLASS_CODE	STU_NUM	ENROLL_GRADE
10014	321452	C
10014	324257	B
10018	321452	A
10018	324257	B
10021	321452	C
10021	324257	C

Table name: CLASS

CLASS_CODE	CRS_CODE	CLASS_SECTION	CLASS_TIME	CLASS_ROOM	PROF_NUM
10014	ACCT-211	3	TTh 2:30-3:45 p.m.	BUS252	342
10018	CIS-220	2	MWF 9:00-9:50 a.m.	KLR211	114
10021	QM-261	1	MWF 8:00-8:50 a.m.	KLR200	114

Figure 23 Converting the M:N relationship two 1:M relationships

Note that the composite ENROLL entity in Figure 23 is existence-dependent on the other two entities; the composition of the ENROLL entity is based on the primary keys of the entities that are connected by the composite entity. The composite entity may also contain additional attributes that play no role in the connective process. For example, although the entity must be composed of at least the STUDENT and CLASS primary keys, it may also include such additional attributes as grades, absences, and other data uniquely identified by the student's performance in a specific class.

Finally, keep in mind that the ENROLL table's key (CLASS_CODE and STU_NUM) is composed entirely of the primary keys of the CLASS and STUDENT tables. Therefore, no null entries are possible in the ENROLL table's key attributes.

Implementing the small database shown in Figure 23 requires that you define the relationships clearly. Specifically, you must know the “1” and the “M” sides of each relationship, and you must know whether the relationships are mandatory or optional. For example, note the following points:

- A class may exist (at least at the start of registration) even though it contains no students. Therefore, if you examine Figure 24, an optional symbol should appear on the STUDENT side of the M:N relationship between STUDENT and CLASS.

You might argue that to be classified as a STUDENT, a person must be enrolled in at least one CLASS. Therefore, CLASS is mandatory to STUDENT from a purely conceptual point of view. However, when a student is admitted to college, that student has not (yet) signed up for any classes. Therefore, at least initially, CLASS is optional to STUDENT. Note that the practical considerations in the data environment help dictate the use of optionalities. If CLASS is not optional to STUDENT—from a database point of view—a class assignment must be made when the student is admitted. But that's not how the process actually works, and the database design must reflect this. In short, the optionality reflects practice.

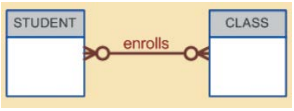

Figure 24 The M:N relationship between STUDENT and CLASS

Because the M: N relationship between STUDENT and CLASS is decomposed into two 1 : M relationships through ENROLL, the optionalities must be transferred to ENROLL. In other words, it now becomes possible for a class not to occur in ENROLL if no student has signed up for that class. Because a class need not occur in ENROLL, the ENROLL entity becomes optional to CLASS. And because the ENROLL entity is created before an y students have signed up for a class, the ENROLL entity is also optional to STUDENT, at least initially.

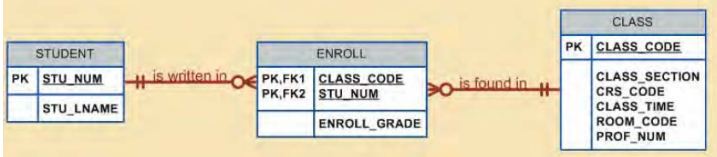

Figure 25 Acomposite entity in an ERD

- As students begin to sign up for their classes, they will be entered into the ENROLL entity. Naturally, if a student takes more than one class, that student will occur more than once in ENROLL. For example, note that in the ENROLL table in Figure 23, STU_NUM = 321452 occurs three times. On the other hand, each student occurs only once in the STUDENT entity. (Note that the STUDENT table in Figure 23 has only one STU_NUM = 321452 entry.) Therefore, in Figure 25, the relationship between STUDENT and ENROLL is shown to be 1:M, with the M on the ENROLL side.
- As you can see in Figure 23, a class can occur more than once in the ENROLL table. For example, CLASS_CODE = 10014 occurs twice. However, CLASS_CODE = 10014 occurs only once in the CLASS table to reflect that the relationship between CLASS and ENROLL is 1:M. Note that in Figure 25, the M is located on the ENROLL side, while the 1 is located on the CLASS side.

Developing an ER diagram

The process of database design is an iterative rather than a linear or sequential process. The verb iterate means “to do again or repeatedly.” An iterative process is, thus, one based on repetition of processes and procedures. Building an ERD usually involves the following activities:

- Create a detailed narrative of the organization's description of operations.
- Identify the business rules based on the description of operations.
- Identify the main entities and relationships from the business rules.
- Develop the initial ERD.
- Identify the attributes and primary keys that adequately describe the entities.
- Revise and review the ERD.

During the review process, it is likely that additional objects, attributes, and relationships will be uncovered. Therefore, the basic ERM will be modified to incorporate the newly discovered ER components. Subsequently, another round of reviews might yield additional components or clarification of the existing diagram. The process is repeated until the end users and designers agree that the ERD is a fair representation of the organization's activities and functions.

During the design process, the database designer does not depend simply on interviews to help define entities, attributes, and relationships. A surprising amount of information can be gathered by examining the business forms and reports that an organization uses in its daily operations.

To illustrate the use of the iterative process that ultimately yields a workable ERD, let's start with an initial interview with the Tiny College administrators. The interview process yields the following business rules:

1. 1 . Tiny College (TC) is divided into several schools: a school of business, a school of arts and sciences, a school of education, and a school of applied sciences. Each school is administered by a dean who is a professor. Each professor can be the dean of only one school, and a professor is not required to be the dean of any school. Therefore, a 1:1 relationship exists between PROFESSOR and SCHOOL. Note that the cardinality can be expressed by writing (1,1) next to the entity PROFESSOR and (0,1) next to the entity SCHOOL.
2. Each school comprises several departments. For example, the school of business has an accounting department, a management/marketing department, an economics/finance department, and a computer information systems department. Note again the cardinality rules: The smallest number of departments

operated by a school is one, and the largest number of departments is indeterminate (N). On the other hand, each department belongs to only a single school; thus, the cardinality is expressed by (1,1). That is, the minimum number of schools that a department belongs to is one, as is the maximum number. Figure 26 illustrates these first two business rules.

3. Each department may offer courses. For example, the management/marketing department offers courses such as Introduction to Management, Principles of Marketing, and Production Management. The ERD segment for this condition is shown in Figure 27. Note that this relationship is based on the way Tiny College operates. If, for example, Tiny College had some departments that were classified as "research only," those departments would not offer courses; therefore, the COURSE entity would be optional to the DEPARTMENT entity.
4. The relationship between COURSE and CLASS was illustrated in Figure 9. Nevertheless, it is worth repeating that a CLASS is a section of a COURSE. That is, a department may offer several sections (classes) of the same database course. Each of those classes is taught by a professor at a given time in a given place. In short, a 1:M relationship exists between COURSE and CLASS. However, because a course may exist in Tiny College's course catalog even when it is not offered as a class in a current class schedule, CLASS is optional to COURSE. Therefore, the relationship between COURSE and CLASS looks like Figure 28.

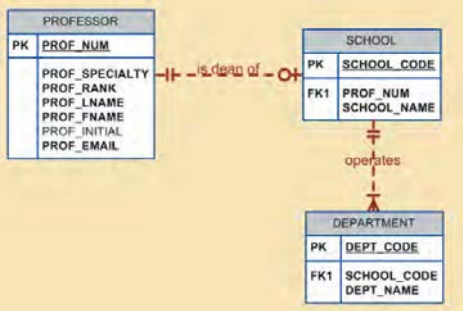

Figure 26 The first Tiny College ERD segment

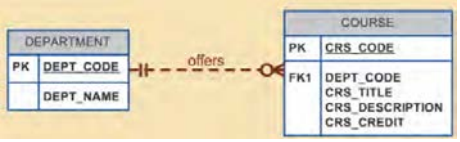

Figure 27 The second Tiny College ERD segment

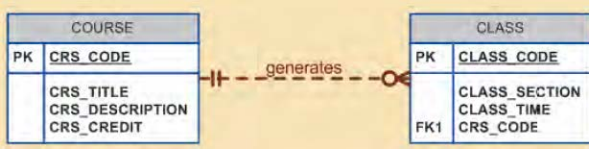

Figure 28 The third Tiny College ERD segment

5. Each department should have one or more professors assigned to it. One and only one of those professors chairs the department, and no professor is required to accept the chair position. Therefore, DEPARTMENT is optional to PROFESSOR in the "chairs" relationship. Those relationships are summarized in the ER segment shown in Figure 29.
6. Each professor may teach up to four classes; each class is a section of a course. A professor may also be on a research contract and teach no classes at all. The ERD segment in Figure 30 depicts those conditions.

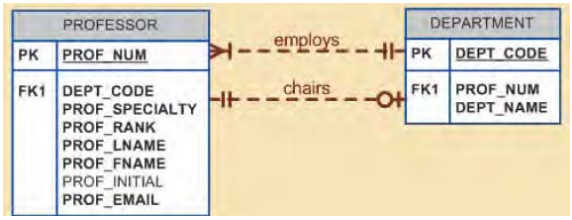

Figure 29 The forth Tiny College ERD segment

7. A student may enroll in several classes but takes each class only once during any given enrollment period. For example, during the current enrollment period, a student may decide to take five classes—Statistics, Accounting, English, Database, and History—but that student would not be enrolled in the same Statistics class five times during the enrollment period! Each student may enroll in up to six classes, and each class may have up to 35 students, thus creating an M:N relationship between STUDENT and CLASS. Because a CLASS can initially exist (at the start of the enrollment period) even though no students have enrolled in it, STUDENT is optional to CLASS in the M:N relationship. This M:N relationship must be divided into two 1:M relationships through the use of the ENROLL entity, shown in the ERD segment in Figure 31. But note that the optional symbol is shown next to ENROLL. If a class exists but has no students enrolled in it, that class doesn't occur in the ENROLL table. Note also that the ENROLL entity is weak: it is existence-dependent, and its (composite) PK is composed of the PKs of the STUDENT and CLASS entities. You can add the cardinalities (0,6) and (0,35) next to the ENROLL entity to reflect the business rule constraints, as shown in Figure 31. (Visio Professional does not automatically generate such cardinalities, but you can use a text box to accomplish that task.)

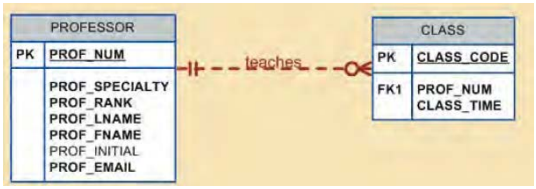

Figure 30 The fifth Tiny College ERD segment

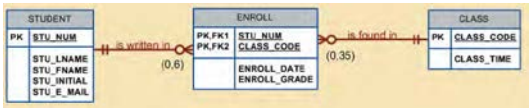

Figure 31 The sixth Tiny College ERD segment

8. Each department has several (or many) students whose major is offered by that department. However, each student has only a single major and is, therefore, associated with a single department. (See Figure 32.) However, in the Tiny College environment, it is possible—at least for a while—for a student not to declare a major field of study. Such a student would not be associated with a department; therefore, DEPARTMENT is optional to STUDENT. It is worth repeating that the relationships between entities and the entities themselves reflect the organization's operating environment. That is, the business rules define the ERD components.

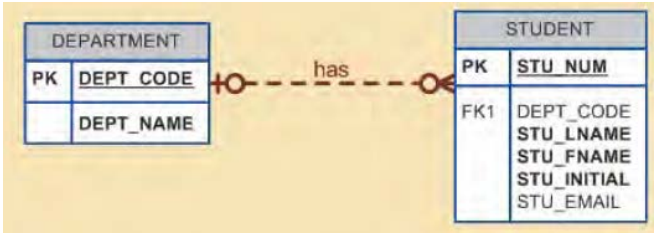

Figure 32 The seventh Tiny College ERD segment

9. Each student has an advisor in his or her department; each advisor counsels several students. An advisor is also a professor, but not all professors advise students. Therefore, STUDENT is optional to PROFESSOR in the "PROFESSOR advises STUDENT" relationship. (See Figure 33.)
10. As you can see in Figure 34, the CLASS entity contains a ROOM_CODE attribute. Given the naming conventions, it is clear that ROOM_CODE is an FK to another entity. Clearly, because a class is taught in a room, it is reasonable to assume that the ROOM_CODE in CLASS is the FK to an entity named ROOM. In turn, each room is located in a building. So the last Tiny College ERD is created by observing that a BUILDING can contain many ROOMs, but each ROOM is found in a single BUILDING. In this ERD segment, it is clear that some buildings do not contain (class) rooms. For example, a storage building might not contain any named rooms at all.

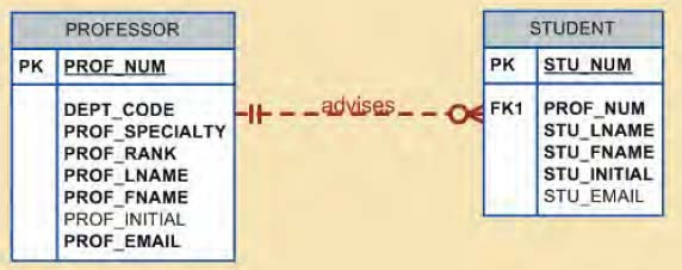

Figure 33 The eight Tiny College ERD segment

Using the preceding summary, you can identify the following entities:

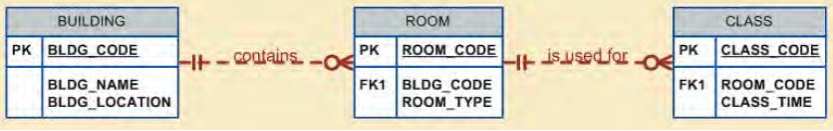

Figure 34 The ninth Tiny College ERD segment

Once you have discovered the relevant entities, you can define the initial set of relationships among them. Next, you describe the entity attributes. Identifying the attributes of the entities helps you to better understand the relationships among entities. Table 4 summarizes the ERM's components, and names the entities and their relations.

ENTITY	RELATIONSHIP	CONNECTIVITY	ENTITY
SCHOOL	operates	1:M	DEPARTMENT
DEPARTMENT	has	1:M	STUDENT
DEPARTMENT	employs	1:M	PROFESSOR
DEPARTMENT	offers	1:M	COURSE
COURSE	generates	1:M	CLASS
PROFESSOR	is dean of	1:1	SCHOOL
PROFESSOR	chairs	1:1	DEPARTMENT
PROFESSOR	teaches	1:M	CLASS
PROFESSOR	advises	1:M	STUDENT
STUDENT	enrolls in	M:N	CLASS
BUILDING	contains	1:M	ROOM
ROOM	is used for	1:M	CLASS
Note: ENROLL is the composite entity that implements the M:N relationship "STUDENT enrolls in CLASS."			

Table 4 Components of ERM

You must also define the connectivity and cardinality for the just-discovered relations based on the business rules. However, to avoid crowding the diagram, the cardinalities are not shown. Figure 35 shows the Crow's Foot ERD for Tiny College. Note that this is an implementation-ready model. Therefore it shows the ENROLL composite entity.

Figure 36 shows the conceptual UML class diagram for Tiny College. Note that this class diagram depicts the M:N relationship between STUDENT and CLASS. Figure 37 shows the implementation-ready UML class diagram for Tiny College (note that the ENROLL composite entity is shown in this class diagram.

DATABASE DESIGN CHALLENGES: CONFLICTING GOALS

Database designers often must make design compromises that are triggered by conflicting goals, such as adherence to design standards (design elegance), processing speed, and information requirements.

- Design standards. The database design must conform to design standards. Such standards have guided you in developing logical structures that minimize data redundancies, thereby minimizing the likelihood that destructive data anomalies will occur. You have also learned how standards prescribe avoiding nulls to the greatest extent possible. In fact, you have learned that design standards govern the presentation of all components within the database design. In short, design standards allow you to work with well-defined components and to evaluate the interaction of those components with some precision. Without design standards, it is nearly impossible to formulate a proper design process, to evaluate an existing design, or to trace the likely logical impact of changes in design.
- Processing speed. In many organizations, particularly those generating large numbers of transactions, high processing speeds are often a top priority in database design. High processing speed means minimal access time, which may be achieved by minimizing the number and complexity of logically desirable relationships. For example, a “perfect” design might use a 1:1 relationship to avoid nulls, while a higher transaction-speed design might combine the two tables to avoid the use of an additional relationship, using dummy entries to avoid the nulls. If the focus is on data-retrieval speed, you might also be forced to include derived attributes in the design.
- Information requirements. The quest for timely information might be the focus of database design. Complex information requirements may dictate data transformations, and they may expand the number of entities and attributes within the design. Therefore, the database may have to sacrifice some of its “clean” design structures and/or some of its high transaction speed to ensure maximum information generation. For example, suppose that a detailed sales report must be generated periodically. The sales report includes all invoice subtotals, taxes, and totals; even the invoice lines include subtotals. If the sales report includes hundreds of thousands (or even millions) of invoices, computing the totals, taxes, and subtotals is likely to take some time. If those computations had been made and the results had been stored as derived attributes in the INVOICE and LINE tables at the time of the transaction, the real-time transaction speed might have declined. But that loss of speed would only be noticeable if there were many simultaneous transactions. The cost of a slight loss of transaction speed at the front end and the addition of multiple derived attributes is likely to pay off when the sales reports are generated (not to mention the fact that it will be simpler to generate the queries).

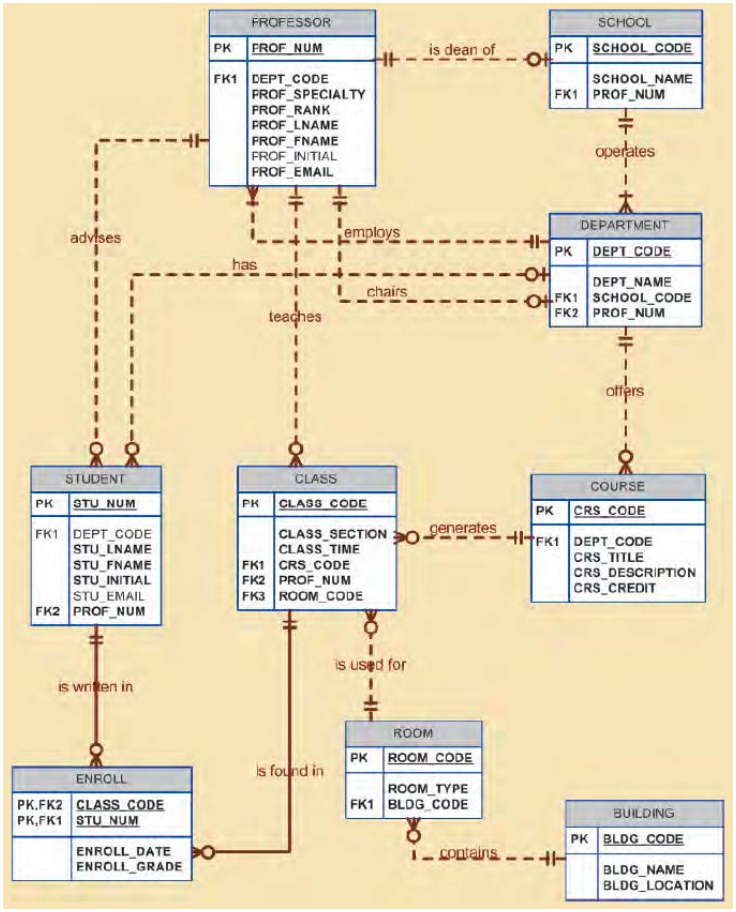

Figure 35 The completed Tiny College ERD

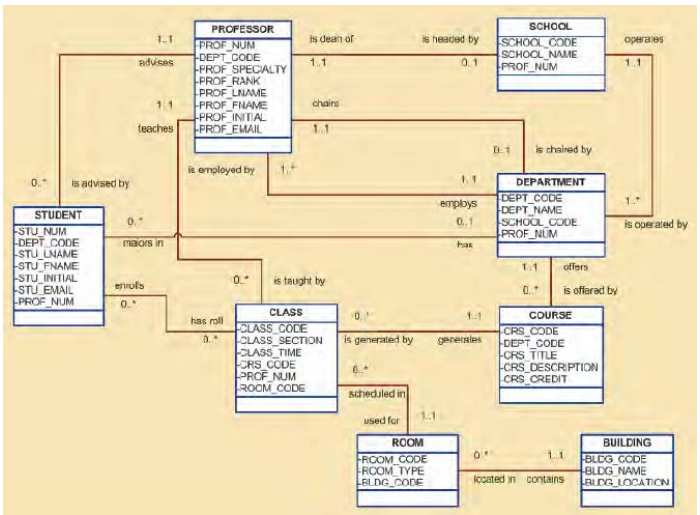

Figure 36 The conceptual UML class diagram for Tiny College

A design that meets all logical requirements and design conventions is an important goal. However, if this perfect design fails to meet the customer's transaction speed and/or information requirements, the designer will not have done a proper job from the end user's point of view. Compromises are a fact of life in the real world of database design.

Even while focusing on the entities, attributes, relationships, and constraints, the designer should begin thinking about end-user requirements such as performance, security, shared access, and data integrity. The designer must consider processing requirements and verify that all update, retrieval, and deletion options are available. Finally,

a design is of little value unless the end product is capable of delivering all specified query and reporting requirements.

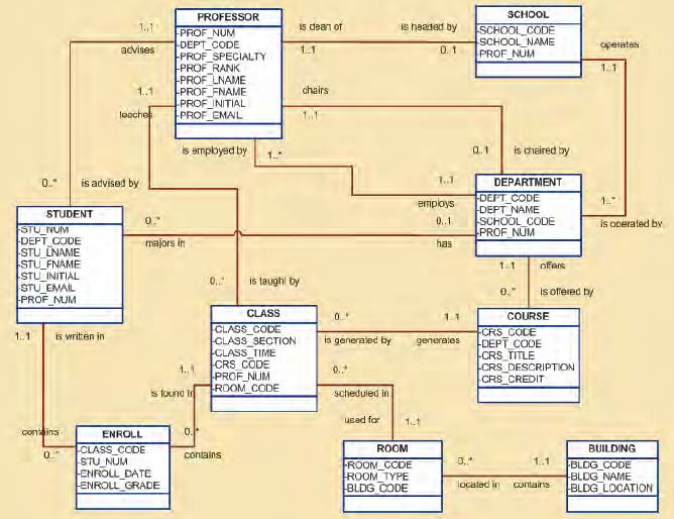

Figure 37 The implementation-redy UML class diagram for College

You are quite likely to discover that even the best design process produces an ERD that requires further changes mandated by operational requirements. Such changes should not discourage you from using the process. ER modeling is essential in the development of a sound design that is capable of meeting the demands of adjustment and growth. Using ERDs yields perhaps the richest bonus of all: a thorough understanding of how an organization really functions.

There are occasional design and implementation problems that do not yield “clean” implementation solutions. To get a sense of the design and implementation choices a database designer faces, let's revisit the 1:1 recursive relationship “EMPLOYEE is married to EMPLOYEE” first examined in Figure 18. Figure 38 shows three different ways of implementing such a relationship.

Note that the EMPLOYEE_V1 table in Figure 38 is likely to yield data anomalies. For example, if Anne Jones divorces Anton Shapiro, two records must be updated—by setting the respective EMP_SPOUSE values to null—to properly reflect that change. If only one record is updated, inconsistent data occur. The problem becomes even worse if several of the divorced employees then marry each other. In addition, that implementation also produces undesirable nulls for employees who are not married to other employees in the company.

Another approach would be to create a new entity shown as MARRIED_V1 in a 1:M relationship with EMPLOYEE. (See Figure 38.) This second implementation does eliminate the nulls for employees who are not married to somebody working for the same company. (Such employees would not be entered in the MARRIED_V1 table.) However, this approach still yields possible duplicate values. For example, the marriage between employees 345 and 347 may still appear twice, once as 345,347 and once as 347,345. (Since each of those permutations is unique the first time it appears, the creation of a unique index will not solve the problem.)

As you can see, the first two implementations yield several problems:

• Both solutions use synonyms. The EMPLOYEE_V1 table uses EMP_NUM and EMP_SPOUSE to refer to an employee. The MARRIED_V1 table uses the same synonyms.

• Both solutions are likely to produce inconsistent data. For example, it is possible to enter employee 345 as married to employee 347 and to enter employee 348 as married to employee 345.

• Both solutions allow data entries to show one employee married to several other employees. For example, it is possible to have data pairs such as 345,347 and 348,347 and 349,347, none of which will violate entity integrity requirements, because they are all unique.

A third approach would be to have two new entities, MARRIAGE and MARPART, in a 1: M relationship. MARPART contains the EMP_NUM foreign key to EMPLOYEE. But even this approach has issues. It requires

the collection of additional data regarding the employees' marriage—the marriage date. If the business users do not need this data, then requiring them to collect it would be inappropriate. To ensure that an employee occurs only once in any given marriage, you would have to create a unique index on the EMP_NUM attribute in the MARPART table. Another potential problem with this solution is that the database implementation will allow more than two employees to "participate" in the same marriage.

As you can see, a recursive 1 :1 relationship yields many different solutions with varying degrees of effectiveness and adherence to basic design principles. Any of the above solutions would likely involve the creation of program code to help ensure the integrity and consistency of the data. In a later chapter, we will examine the creation of database triggers that can do exactly that. Your job as a database designer is to use your professional judgment to yield a solution that meets the requirements imposed by business rules, processing requirements, and basic design principles.

Finally, document, document, and document! Put all design activities in writing. Then review what you've written. Documentation not only helps you stay on track during the design process, but also enables you (or those following you) to pick up the design thread when the time comes to modify the design. Although the need for documentation should be obvious, one of the most vexing problems in database and systems analysis work is that the "put it in writing" rule is often not observed in all of the design and implementation stages. The development of organizational documentation standards is a very important aspect of ensuring data compatibility and coherence.

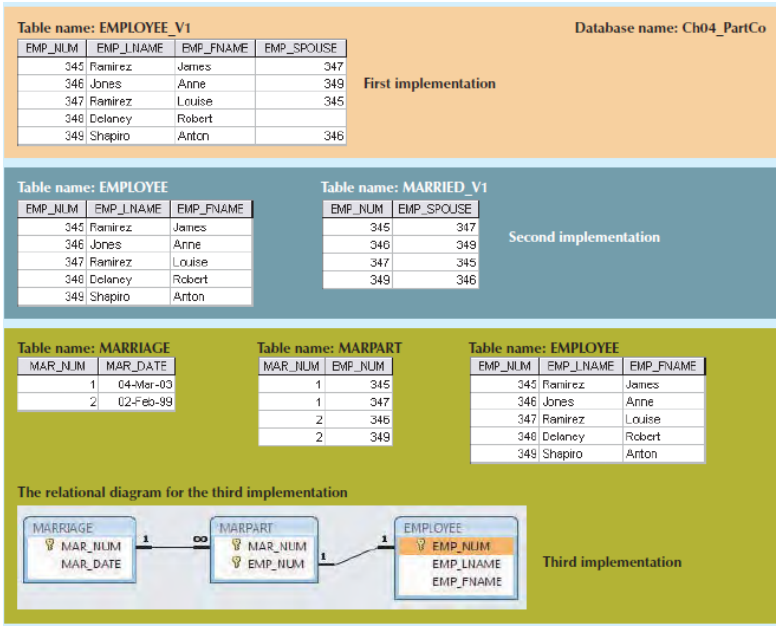

Figure 38 Various implementations of the 1:1 recursive relationship

The database design process

A well-structured database:

- Saves disk space by eliminating redundant data.
- Maintains data accuracy and integrity.
- Provides access to the data in useful ways.

Designing an efficient, useful database is a matter of following the proper process, including these phases:

1. Requirements analysis, or identifying the purpose of your database
2. Organizing data into tables
3. Specifying primary keys and analyzing relationships
4. Normalizing to standardize the tables

Let's take a closer look at each step. Note that this guide deals with Edgar Codd's relational database model as written in SQL (rather than the hierarchical, network, or object data models). To learn more about database models, read our guide here.

Requirements analysis: identifying the purpose of the database

Understanding the purpose of your database will inform your choices throughout the design process. Make sure you consider the database from every perspective. For instance, if you were making a database for a public library, you'd want to consider the ways in which both patrons and librarians would need to access the data.

Here are some ways to gather information before creating the database:

- Interview the people who will use it
- Analyze business forms, such as invoices, timesheets, surveys
- Comb through any existing data systems (including physical and digital files)

Start by gathering any existing data that will be included in the database. Then list the types of data you want to store and the entities, or people, things, locations, and events, that those data describe, like this:

Customers

- Name
- Address
- City, State, Zip
- Email address

Products

- Name
- Price
- Quantity in stock
- Quantity on order

Orders

- Order ID
- Sales representative
- Date
- Product(s)
- Quantity
- Price
- Total

This information will later become part of the data dictionary, which outlines the tables and fields within the database. Be sure to break down the information into the smallest useful pieces. For instance, consider separating the street address from the country so that you can later filter individuals by their country of residence. Also, avoid placing the same data point in more than one table, which adds unnecessary complexity.

Once you know what kinds of data the database will include, where that data comes from, and how it will be used, you're ready to start planning out the actual database.

Database structure: the building blocks of a database

The next step is to lay out a visual representation of your database. To do that, you need to understand exactly how relational databases are structured.

Within a database, related data are grouped into tables, each of which consists of rows (also called tuples) and columns, like a spreadsheet.

To convert your lists of data into tables, start by creating a table for each type of entity, such as products, sales, customers, and orders. Here's an example:

Each row of a table is called a record. Records include data about something or someone, such as a particular customer. By contrast, columns (also known as fields or attributes) contain a single type of information that appears in each record, such as the addresses of all the customers listed in the table.

First Name	Last Name	Age	ZIP Code
Roger	Williams	43	34760
Jerrica	Jorgensen	32	97453
Samantha	Hopkins	56	64829

To keep the data consistent from one record to the next, assign the appropriate data type to each column. Common data types include:

- CHAR - a specific length of text
- VARCHAR - text of variable lengths
- TEXT - large amounts of text
- INT - positive or negative whole number
- FLOAT, DOUBLE - can also store floating point numbers
- BLOB - binary data

Some database management systems also offer the Autonumber data type, which automatically generates a unique number in each row.

For the purposes of creating a visual overview of the database, known as an entity-relationship diagram, you won't include the actual tables. Instead, each table becomes a box in the diagram. The title of each box should indicate what the data in that table describes, while attributes are listed below, like this:

Finally, you should decide which attribute or attributes will serve as the primary key for each table, if any. A primary key (PK) is a unique identifier for a given entity, meaning that you could pick out an exact customer even if you only knew that value.

Attributes chosen as primary keys should be unique, unchanging, and always present (never NULL or empty). For this reason, order numbers and usernames make good primary keys, while telephone numbers or street addresses do not. You can also use multiple fields in conjunction as the primary key (this is known as a composite key).

When it comes time to create the actual database, you'll put both the logical data structure and the physical data structure into the data definition language supported by your database management system. At that point, you should also estimate the size of the database to be sure you can get the performance level and storage space it will require.

Creating relationships between entities

With your database tables now converted into tables, you're ready to analyze the relationships between those tables. Cardinality refers to the quantity of elements that interact between two related tables. Identifying the cardinality helps make sure you've divided the data into tables most efficiently.

Each entity can potentially have a relationship with every other one, but those relationships are typically one of three types:

One-to-one relationships

When there's only one instance of Entity A for every instance of Entity B, they are said to have a one-to-one relationship (often written 1:1). You can indicate this kind of relationship in an ER diagram with a line with a dash on each end:

Unless you have a good reason not to, a 1:1 relationship usually indicates that you'd be better off combining the two tables' data into a single table.

However, you might want to create tables with a 1:1 relationship under a particular set of circumstances. If you have a field with optional data, such as "description," that is blank for many of the records, you can move all of the descriptions into their own table, eliminating empty space and improving database performance.

To guarantee that the data matches up correctly, you'd then have to include at least one identical column in each table, most likely the primary key.

One-to-many relationships
These relationships occur when a record in one table is associated with multiple entries in another. For example, a single customer might have placed many orders, or a patron may have multiple books checked out from the library at once. One-to-many (1:M) relationships are indicated with what's called "Crow's foot notation," as in this example:

To implement a 1:M relationship as you set up a database, simply add the primary key from the "one" side of the relationship as an attribute in the other table. When a primary key is listed in another table in this manner, it's called a foreign key. The table on the "1" side of the relationship is a considered a parent table to the child table on the other side.

Many-to-many relationships
When multiple entities from a table can be associated with multiple entities in another table, they are said to have a many-to-many (M:N) relationship. This might happen in the case of students and classes, since a student can take many classes and a class can have many students.

In an ER diagram, these relationships are portrayed with these lines:

Unfortunately, it's not directly possible to implement this kind of relationship in a database. Instead, you have to break it up into two one-to-many relationships.

To do so, create a new entity between those two tables. If the M:N relationship exists between sales and products, you might call that new entity "sold_products," since it would show the contents of each sale. Both the sales and products tables would have a 1:M relationship with sold_products. This kind of go-between entity is called a link table, associative entity, or junction table in various models.

Each record in the link table would match together two of the entities in the neighboring tables (it may include supplemental information as well). For instance, a link table between students and classes might look like this:

Mandatory or not?
Another way to analyze relationships is to consider which side of the relationship has to exist for the other to exist. The non-mandatory side can be marked with a circle on the line where a dash would be. For instance, a country has to exist for it to have a representative in the United Nations, but the opposite is not true:

Two entities can be mutually dependent (one could not exist without the other).

Recursive relationships
Sometimes a table points back to itself. For example, a table of employees might have an attribute "manager" that refers to another individual in that same table. This is called a recursive relationship.

Redundant relationships
A redundant relationship is one that is expressed more than once. Typically, you can remove one of the relationships without losing any important information. For instance, if an entity "students" has a direct relationship with another called "teachers" but also has a relationship with teachers indirectly through "classes," you'd want to remove the relationship between "students" and "teachers." It's better to delete that relationship because the only way that students are assigned to teachers is through classes.

Database normalization

Once you have a preliminary design for your database, you can apply normalization rules to make sure the tables are structured correctly. Think of these rules as the industry standards.

That said, not all databases are good candidates for normalization. In general, online transaction processing (OLTP for short) databases, in which users are concerned with creating, reading, updating, and deleting records, should be normalized.

Online analytical processing (OLAP) databases which favor analysis and reporting might fare better with a degree of denormalization, since the emphasis is on speed of calculation. These include decision support applications in which data needs to be analyzed quickly but not changed.

Each form, or level of normalization, includes the rules associated with the lower forms.

First normal form

The first normal form (abbreviated as 1NF) specifies that each cell in the table can have only one value, never a list of values, so a table like this does not comply:

ProductID	Color	Price
1	brown, yellow	$15
2	red, green	$13
3	blue, orange	$11

You might be tempted to get around this by splitting that data into additional columns, but that's also against the rules: a table with groups of repeated or closely related attributes does not meet the first normal form. The table below, for example, fails to comply:

Instead, split the data into multiple tables or records until each cell holds only one value and there are no extra columns. At that point, the data is said to be atomic, or broken down to the smallest useful size. For the table above, you could create an additional table called "Sales details" that would match specific products with sales. "Sales" would then have a 1:M relationship with "Sales details."

Second normal form

The second normal form (2NF) mandates that each of the attributes should be fully dependent on the entire primary key. That means each attribute should depend directly on the primary key, rather than indirectly through some other attribute.

For instance, an attribute "age" that depends on "birthdate" which in turn depends on "studentID" is said to have a partial functional dependency, and a table containing these attributes would fail to meet the second normal form.

Furthermore, a table with a primary key made up of multiple fields violates the second normal form if one or more of the other fields do not depend on every part of the key.

Thus, a table with these fields wouldn't meet the second normal form, because the attribute "product name" depends on the product ID but not on the order number:

- Order number (primary key)
- Product ID (primary key)
- Product name

Third normal form

The third normal form (3NF) adds to these rules the requirement that every non-key column be independent of every other column. If changing a value in one non-key column causes another value to change, that table does not meet the third normal form.

This keeps you from storing any derived data in the table, such as the "tax" column below, which directly depends on the total price of the order:

Order	Price	Tax
14325	$40.99	$2.05
14326	$13.73	$.69
14327	$24.15	$1.21

Additional forms of normalization have been proposed, including the Boyce-Codd normal form, the fourth through sixth normal forms, and the domain-key normal form, but the first three are the most common.

While these forms explain the best practices to follow generally, the degree of normalization depends on the context of the database.

Diagramming is quick and easy with Lucidchart. Start a free trial today to start creating and collaborating.

Multidimensional data

Some users may want to access multiple dimensions of a single type of data, particularly in OLAP databases. For instance, they may want to know the sales by customer, state, and month. In this situation, it's best to create a central fact table that other customer, state, and month tables can refer to, like this:

Data integrity rules

You should also configure your database to validate the data according to the appropriate rules. Many database management systems, such as Microsoft Access, enforce some of these rules automatically.

The entity integrity rule says that the primary key can never be NULL. If the key is made up of multiple columns, none of them can be NULL. Otherwise, it could fail to uniquely identify the record.

The referential integrity rule requires each foreign key listed in one table to be matched with one primary key in the table it references. If the primary key changes or is deleted, those changes will need to be implemented wherever that key is referenced throughout the database.

Business logic integrity rules make sure that the data fits within certain logical parameters. For instance, an appointment time would have to fall within normal business hours.

Adding indexes and views

An index is essentially a sorted copy of one or more columns, with the values either in ascending or descending order. Adding an index allows users to find records more quickly. Instead of re-sorting for each query, the system can access records in the order specified by the index.

Although indexes speed up data retrieval, they can slow down inserting, updating, and deleting, since the index has to be rebuilt whenever a record is changed.

A view is simply a saved query on the data. They can usefully join data from multiple tables or else show part of a table.

Extended properties

Once you have the basic layout completed, you can refine the database with extended properties, such as instructional text, input masks, and formatting rules that apply to a particular schema, view, or column. The advantage is that, because these rules are stored in the database itself, the presentation of the data will be consistent across the multiple programs that access the data.

SQL and UML

The Unified Modeling Language (UML) is another visual way of expressing complex systems created in an object-oriented language. Several of the concepts mentioned in this guide are known in UML under different names. For instance, an entity is known as a class in UML.

UML is not used as frequently today as it once was. Today, it is often used academically and in communications between software designers and their clients.