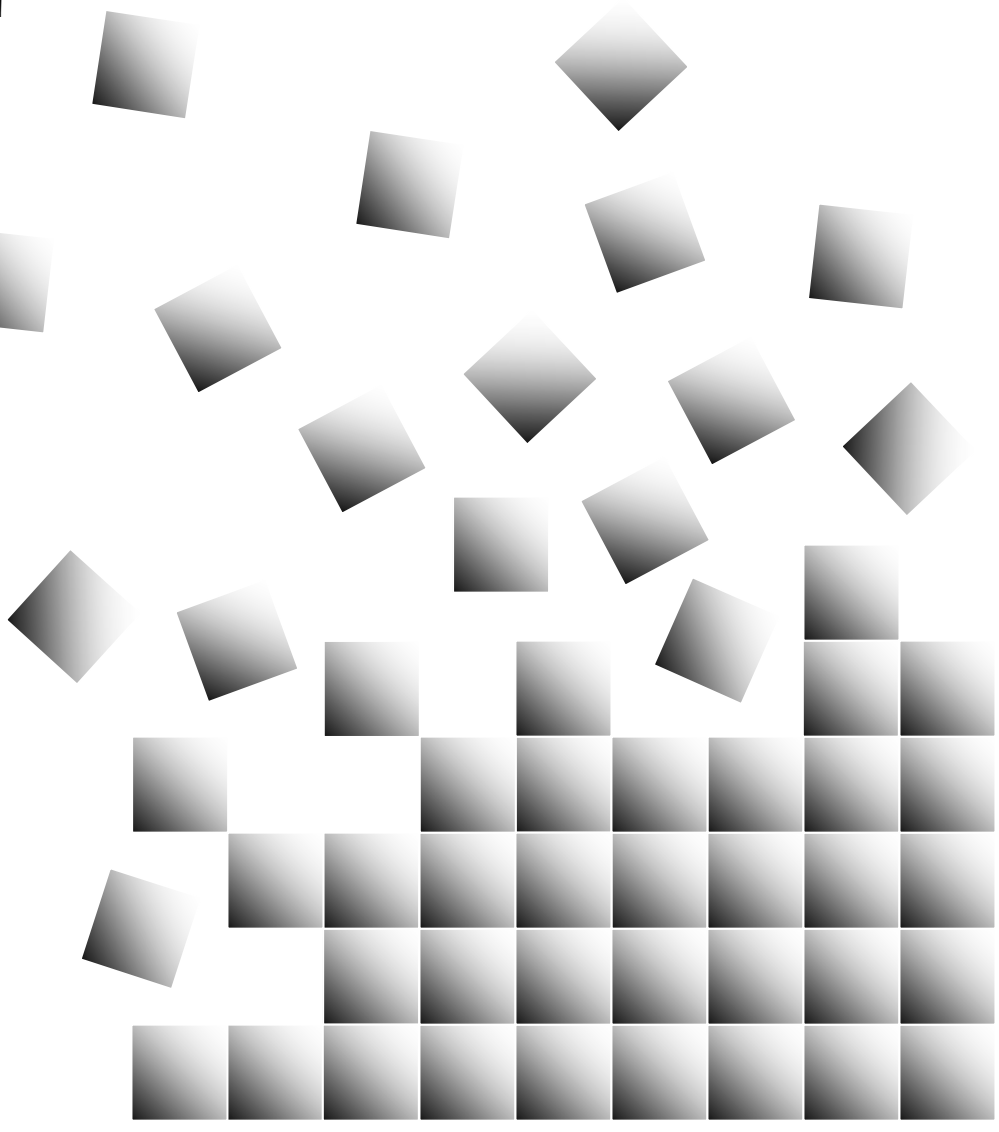
An Intuitive Approach to Database Design

An Introduction to Data Modeling

For Harvard CSCI S-T



PETER AVILA

An Intuitive Approach to Database Design:
An Introduction to Data Modeling

Peter Avila

P13

Table of Contents

Introduction

As is so often the case, courses in database design start out with some of the toughest concepts to grasp. Students are often asked to understand functional dependencies, determinants, normalization and other such concepts before they can truly appreciate their significance. The purpose of this text is to help you become familiar with the fundamental concepts behind Database Design theory by relying on your intuitive sense first. At the end of this article, we will have designed a database that is normalized to Boyce-Codd Normal Form without ever having discussed any of the formal concepts.[1] Instead, the design will draw on common sense. This, in turn, will help you more easily grasp the formal concepts we will discuss later in the course and that are indispensable if a comprehensive understanding of the subject of database design is to be achieved.

We will use a school enrollment system as a backdrop for our discussion. First, we will define the term "database" and look at its most fundamental building block, the table. We will then learn how to properly organize data into tables. Once we have determined the tables, we will need a way to represent the relationships among them, so that we can ask questions such as, "How many students are enrolled in a particular course?" (This question uses data from two *related* tables: the Student table and the Course table.) Against the backdrop of the school enrollment system, we will examine the types of relationships that can exist among tables, how to represent these relationships so that our database works properly, and other topics.

While the information in this article does not depend on any specific DBMS such as Oracle, Access, SQL Server, or others, the implementation of a design does. In other words, you can decide what tables to use, how they will be related, and so forth, regardless of your DBMS; on the other hand, how you actually create the tables and relate them *will* depend on your particular DBMS.

We start by exploring just what a database is.

[1] Normalization is a formal approach to database design, and it is a cornerstone of this course. However, it is not explicitly discussed in this article even though the principles are here in an intuitive sense.

For a formal treatment of database theory, see these sources:

- Date, C., "An Introduction to Database Systems," Addison-Wesley.
- Elmasri, R. and Navathe, S., "Fundamentals of Database Systems," Benjamin Cummings.

Databases: Definition and Purpose

We do many things with databases. We track aspects of an environment, such as the activities of a medical office or school enrollment system; we produce reports and charts from the data in the database; we look up data based on a value such as someone's last name, or based on criteria such as "everyone living in the state of California;" we anticipate changes in the real environment based on what we see going on in the database; and so on. All of these things have this in common: In each case we *ask and answer questions* about the data. To track data, we answer questions such as, "What are the courses in which a particular student has enrolled over the past year?" When we produce reports and charts, we answer questions such as, "What percentage of patients have improved from taking a particular medication?" When we anticipate changes, we answer questions such as, "How many students enrolled in each of the past years?" In database terms such questions are called *queries*. The process by which answers to queries are obtained is the process by which *data*—the raw facts represented in the database—are transformed into *information*—the answers to our queries.[2]

So, the database allows this transformation to occur, but it can only occur accurately if a) the data are properly organized in the database, and b) the data are related. The issue of how to properly organize a database is the central topic of discussion in this text, and we will get to it shortly. As for relationships among the data, it is important not to clutter the database with unrelated data, as this leads to confusion, and confusion can lead to poor transformations. For example, we would not store data about horse race results in a database designed to run a medical office, unless, for example, our medical office were investigating horse race results among betting patients who are depressed. In the latter case, horse race results *would* be related to our other medical records and could be included in the database.

Now, let's look at this last point from the other side of the coin. The real world, as we know, has a large set of characteristics and details. When designing a database, we are not typically interested in *all* of the characteristics and details of the real world; just those that lie in our sphere of interest. Database designers refer to this sphere of interest as the *mini-world* or *universe of discourse*. For example, a mini-world might be the aspects of a medical office or school enrollment system that interest us, such as the names and contact information of all patients, their medical records, schedule of appointments for each patient, and so on. Data in a database are related because they belong to the same mini-world.

We can say that a database is a *model* of a mini-world. If a model of an airplane is built properly, it will fly like the real thing. Similarly, if we build our database model properly, we can get accurate transformations (information) from it.

We now define a database:

> *A database is an organized collection of related data that models an aspect of an environment in which we have an interest with the purpose of giving us a means by which data can be transformed into information.*

[2] A quick note on the distinction between data and information: there is a certain amount of relativity here. One person's data may be another person's information. To one person, the fact that John's birthday is in June and Mary's is in August may be data, and, for such a person, the answer to the question, "How many people have birthdays in June?" may be information. To another person, the fact that John's birthday is in June may be the answer to a question, and so it would be information.

This text focuses on how to properly organize related data and build a model of the mini-world in which it exists. We will start by looking at the most fundamental organizational structure of a database: the table.

The Database Table

You may already be familiar with the concept of the database table; it is a two-dimensional object made up of a collection of rows and a collection of columns. In this section, we will take a look at a table in a way that may be new to you. We will consider each row in the table to represent an *entity*, and each column an *attribute.* To us, an entity is something that exists distinctly in the miniworld and that has characteristics (attributes) in which we are interested.

Most of the time we find many entities of the same type, and we can then group them together. For example, cars may have the following attributes: number of cylinders, number of doors, horsepower, and others. We can group all car entities together into a car *entity-type*. Another example of an entity is a game. Attributes of games might be team A, team B, final score, number of innings, and so on. (Notice that entities do not have to be physical things, such as cars, people, houses, books, or trees, but they can also be things like soccer games, weddings, recitals, campaigns, and others. Anything that has a distinct existence and that has attributes in which we are interested qualifies as an entity.)

In a database, the job of a table is to represent an entity-type. The rows of the table hold the individual entities that belong to that entity-type, and the columns of the table represent the attributes that all of the entities in that entity-type have in common. Figure 1 shows an example of a table of orders. Each order is an entity represented by a different row of the table, and each column of the table represents a different attribute of an order, such as order date, salesperson who placed the order, customer for whom the order was placed, terms of payment, and so on; all of which help to describe an order entity.

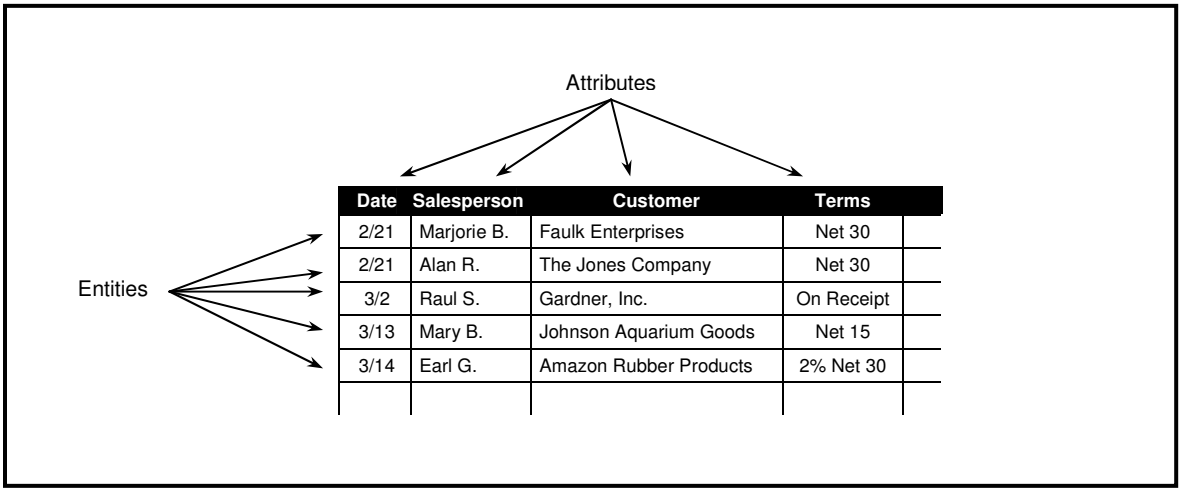


Figure 1 In a table of orders, each row represents a different order entity (or order), and each column represents an attribute of orders. Each attribute contributes to the description of the order. The table itself represents an entity-type.

Can we say that each table must represent only one entity-type? Not necessarily. A table can represent more than one entity-type only as long as entities in the table can (but may not necessarilty) belong to any of the entity types, and all of the entity-types share the same attributes. To illustrate, let's assume we have professors and advisors in our mini-world, and that advisors are all professors. As long as one does not have any attributes that are not needed by the other, we can represent both in the same table. Similarly, employees and supervisors can be represented in the same table, but homes and townhouses, for example, cannot, because townhouses have an attribute, number of homes, which is not shared by homes.

As simple a concept as a table may at first seem, tables are the single most common source of problems in a database. Poorly designed tables can result in *anomalies* in the data, and this can in turn result in unreliable query results. Let's see how this can happen.

Designing Tables that Prevent Anomalies

What exactly do we mean by a "well-designed" database? The problem with poorly-designed databases is that they can present what is referred to as *anomalies*. Although we will be discussing anomalies in some detail later, for now we can say that a well-designed database is one that prevents anomalies from occurring. In what follows, we will design a database together. In the process, we will see examples of the different anomalies and how to build a design in which they do not occur.

Since the database will be a model of the mini-world, as we discussed, we will start by describing the mini-world. In our school, students enroll in sections where a course is taught by an instructor at a certain time and location. We want to capture information about sections, courses, professors and students, as well as information about how they relate to each other (which students are enrolled in which sections and so on). You are probably familiar with such a mini-world.

How do we start? Most database design problems boil down to one deceptively simple problem: too few tables! This doesn't mean that more tables will make the design a better one; what it means is that a well designed database will typically have more tables in it than a poorly designed database of the same mini-world. To illustrate, let's start with just one table to exaggerate the problem. This will allow us to see the anomalies sooner.

Figure 2 shows what our table might look like after a few students have enrolled. Keep in mind that, for now, *all* of our enrollment data are kept in this one table.

Student Name	Student Address	Student Phone	(...)	Sctn Days	Sctn Time	Sctn Loc	Prof Name	Prof Address	Pro Phone	(...)	Crse Name	Crse Units	Crse Fee	
Marla Faulk	1 First St	934-5437		M,W	7PM	Rm A	Vickie P.	5 Fifth Ave.	243-7759		Intro to Jazz	4	500	
Tom Lee	2 Second St	395-2117		M,W	7PM	Rm A	Vickie P.	5 Fifth Ave.	243-7759		Intro to Jazz	4	500	
Tom Lee	2 Second St	395-2117		T,Th	8AM	Rm B	Mary F.	6 Sixth St.	399-2118		Databases	4	650	
Yoshi Ohta	3 Third Ave.	223-9849		M,W	9AM	Rm A	Vickie P.	5 Fifth Ave	243-7759		Photography	4	450	
Haydee Dias	4 Fourth St.	838-2322		T,Th	8AM	Rm B	Mary F.	6 Sixth St.	399-2118		Databases	4	650	
Marla Faulk	1 First St	934-5437		M,W	2PM	Rm C	Mary F.	6 Sixth St.	399-2118		Databases	4	650	
Yoshi Ohta	3 Third Ave.	223-9849		T,W	3PM	Rm A	Raul S.	7 7th Ave.	268-2194		Intro to Jazz	4	500	

Figure 2 A single table used to represent an entire mini-world. We build this table to illustrate anomalies and what leads to them.

If you look carefully at figure 2, you can see that there are a lot of redundant (duplicate) data in the table. Marla's data (address, phone, etc.) are stored every time she enrolls in a course. Likewise, every time a student enrolls in the same section, data about that section are stored (days, time, location, etc.). This is also the case with professors, courses, and any other entity that might be represented in this table.[3] Redundancy in data is a problem for several reasons:

1. **It is a waste of storage space**. It is unnecessary to represent the same fact more than once, e.g., that Marla Faulk's phone number is what it is, the fact that there is an Introduction to Jazz section taught by Vicke P. in room A on Mondays and Wednesdays at 7 PM, the fact that Database Systems is worth 4 units, and so on. Facts need only be represented once. Representing them more than once is a waste of space.
2. **It increases the chances for errors**. The more often we enter data into the system, the more likely it is that an error will be made when entering it.
3. **It duplicates effort**. It takes more work to enter data more than once than it does to enter it only once.

These are the obvious problems that arise when we duplicate data in the database as we have done in the table shown in figure 2, but there are other problems that, while not as obvious, are even more serious because they have the potential to defeat the integrity of the data. We refer to these as anomalies. Let's continue our list as we examine the three anomalies.

4. **UPDATE ANOMALY. Changing data in one location of a table causes inconsistencies in the data**. Let's suppose that Marla Faulk moves to another address. If we change her address in the first row of the table shown in figure 2 to the new address, that new value would not be consistent with the value shown a few rows below. This presents an anomaly. In this case, results of queries and reports may be inconsistent and the database becomes unreliable. In database design terms, this is referred to as an *update anomaly*. So, an update anomaly occurs when we must make changes in more than one location in the database when a single fact changes, such as someone's address.
5. **INSERTION ANOMALY: We must unnecessarily represent certain data to be able to represent other data.** With just the enrollments shown in figure 2, we do not have any representation of the fact that our school offers other courses, such as Political Economy or Molecular Physiology. We would have to wait for someone to enroll in these courses before we had any record of their existence, even though courses have existences that are distinct from enrollments. While there is a relationship between course entities and enrollment entities, the existence of a course does not depend on the existence of an enrollment in the mini-world. When a database shows an existence dependency such as this one that does not correspond to the realities of the mini-world, the database exhibits what database designers refer to as an *insertion anomaly*. So, an insertion anomaly occurs when the existence of an entity depends on the existence of another entity that belongs to another entity-type, and when such an existence dependency between the two entities does not occur in the mini-world.
6. **DELETION ANOMALY: We can lose more data than we should if we delete a row**. This is the opposite of the insertion anomaly. If we delete all enrollments for sections of Introduction to Jazz, for example, we also lose the representation of the fact that we offer that course. In database design terms, this is referred to as a *deletion anomaly*. A deletion anomaly occurs when an entity is lost as a result of the removal of another entity that

[3] Redundancy in this way does not qualify as a *backup*! When we capture Marla's address over and over again, we are not making a backup, but rather re-representing the data. If we were making a backup, we would be wiser to make a copy of the entire table.

belongs to another entity-type, and when such an existence dependency between the two entities does not exist in the mini-world.

Clearly, it is not a good idea to duplicate data. In particular, an update anomaly affects the reliability of the results of our queries, and insertion and deletion anomalies make some queries plainly impossible, e.g., "Which courses are offered by the school?"

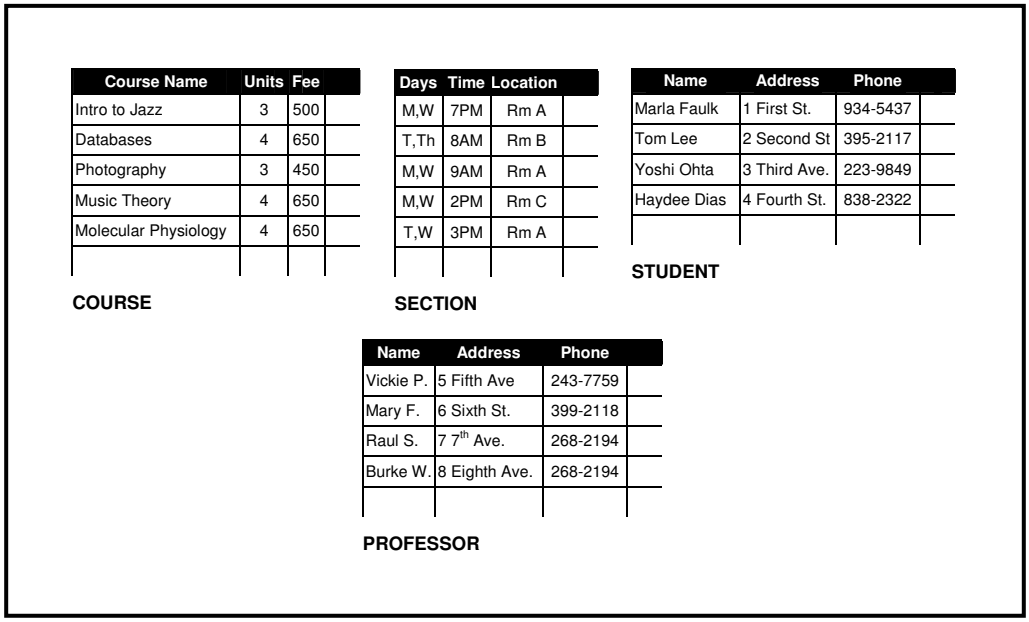
Before we look at how to fix these problems, let's take a moment to examine what is causing them to happen in the first place.

Identifying the Entity-Types: Eliminating the Cause of Anomalies

Now that we know all these things about tables, can we come up with some rules that can help us create tables that do not exhibit anomalies? As we will explore later in the course, there is a set of such rules, referred to as the *rules of normalization*. Normalization, however, depends on an understanding of functional dependencies and other concepts that you will learn about later in the course. The good news is that we don't need the rules of normalization right now; we can acquire an intuitive appreciation for what's involved. For now, a good rule of thumb is to verify that every column (attribute) in the table contributes to the description of what is represented in an *entire* row. For example, in a Student table, each column must contribute to the description of a student; Student Name contributes to the description of a student, as does Student Address, Student Phone, and so on. If the table contains more than one entity-type, then each column must contribute to the description of all of its entity types. For example, in a Professor table that also represents advisors, the Professor Name attribute contributes to the description of both the professor and the advisor, as does the Professor Address, Professor Phone, and so on. We can see that the table in figure 2 does not meet this requirement. Student Name and Course Name, for example, contribute to the descriptions of different *parts* of the row (entities)—namely, students and courses—and not the entire row. This means that while the table represents more than one entity-type, the different entity-types do not share the same attributes, nor can an entity in one entity-type belong to another entity-type. Because the entity-types are all grouped into the same table, the problems listed above occur. This is because each time we add data for an entity-type, we must also represent the data for any related entity-types, causing data duplication, which in turn leads to the anomalies as we have seen.

To fix the problem with the table in figure 2, we start by creating a table for each entity-type as shown in figure 3.[4] ("Section" in figure 3 does not refer to the term used here at Harvard for gatherings with teaching fellows outside of class. In figure 3, a section refers to occurrences of courses.)

[4] Note that the proper way to name tables is to use the singular, as you see in figure 3, because table names refer to an entity-type. If the table represents more than one entity-type, we name the table after only one of them, or we choose a name that refers to all of them collectively.



Course Name	Units	Fee	
Intro to Jazz	3	500	
Databases	4	650	
Photography	3	450	
Music Theory	4	650	
Molecular Physiology	4	650	

COURSE

Days	Time	Location	
M,W	7PM	Rm A	
T,Th	8AM	Rm B	
M,W	9AM	Rm A	
M,W	2PM	Rm C	
T,W	3PM	Rm A	

SECTION

Name	Address	Phone	
Marla Faulk	1 First St.	934-5437	
Tom Lee	2 Second St	395-2117	
Yoshi Ohta	3 Third Ave.	223-9849	
Haydee Dias	4 Fourth St.	838-2322	

STUDENT

Name	Address	Phone	
Vickie P.	5 Fifth Ave	243-7759	
Mary F.	6 Sixth St.	399-2118	
Raul S.	7 7th Ave.	268-2194	
Burke W.	8 Eighth Ave.	268-2194	

PROFESSOR

Figure 3 Our revised database design includes a table for each entity-type.

We seem to have resolved the insertion anomaly issue, because we can now represent the fact that we offer Molecular Physiology even though we may not be offering it in a particular term, and we can list Professor Burke W. even though he is not teaching any courses this term, and so on throughout. We seem to have also resolved the deletion anomaly issue, because we can remove a student from our Student table without having to remove course data from our Course table, and so on throughout. Also, if a professor's address changes, we need only change it in one row of the Professor table, and so on throughout, and so we seem to have resolved the update anomaly issue as well. Are we out of the woods? Alas, no.

In spite of these sound changes, consider how a user might respond to them. With all the problems exhibited in the table of figure 2, the user could still use that table to ask questions that spanned more than one entity-type. Referring to the table in figure 2, we can see that Professor Vickie P. teaches the Introduction to Jazz section on Monday and Wednesday mornings; that Introduction to Jazz is offered in two sections, one on Mondays and Wednesdays and the other on Tuesdays and Thursdays; that there are three students enrolled in Database Systems, and so on. After our changes, how will the user get the same information? Vickie P. is in the Professor table, the courses she teaches are somewhere in the Course table, and her sections are somewhere in the Section table! How will the user answer any question that requires data from more than one table? In particular, how will the user answer questions such as, "In which rooms (Section table) are courses with more than 3 units (Course table) being offered?" or "What are the names and addresses of all students (Student table) taking Introduction to Jazz (Course table)?"

It is clear from questions such as these that much of the information we expect to get from our database refers to *relationships* among the entities. Vickie P., a professor entity, obviously has a relationship with a course entity and a section entity as shown in the table of figure 2. We must give our database design a means by which to represent these relationships in a manner that does not cause the anomalies or other problems exhibited in the table of figure 2.

Identifying the Relationship Types

Between any two related tables,[5] one of the following three types of relationships exists:

- One-to-one (1:1)
- One-to-many (1:N)
- Many-to-many (N:M)

When relating two tables, it is important to identify which of the three types of relationships exists between them. This is because each type of relationship is represented in a different way. To discover the type of relationship between any two related tables, say Table A and Table B, we ask and answer two questions:

Question 1: **For each row in Table A, how many rows *can* there be in Table B, one or many?**

Question 2: **For each row in Table B, how many rows *can* there be in Table A, one or many?**

As the wording of the questions indicates, the answer to either question must be "one" or "many" (we use the word "many" to mean "more than one"). If the answer to both questions is "one," then the type of relationship is 1:1. If the answer to one question is "one" and to the other it is "many," then the type of relationship is 1:N. If the answer to both questions is "many," then the type of relationship is N:M.

The following points regarding the two questions are worth noting:

- We use the word "can" rather than "are." This is because we are not looking at only the particular rows currently in the table, also referred to as the current *state* of the table, but rather any state in which the table might be at any time. In other words, even though there might be only one student enrolled in a particular course (let's say one day after the course is first offered), there *can* be more than one; the possibility is acceptable.
- It is important to ask both questions. One of the most common mistakes made in determining the type of relationship is to neglect to ask the second question. This is probably because the two questions are so similar. If the second question is not answered, a 1:N relationship can appear to be a 1:1, and a N:M relationship can appear to be a 1:N. Errors of this kind have a profoundly negative impact on the ability of the database to function properly and are therefore at the same magnitude of undesirability as are the anomalies.
- The answer to either question will depend on an understanding of the mini-world. For example, in most schools, sections can have only one professor, and so the relationship between the two would be 1:N. On the other hand, it is possible to imagine a mini-world in which a section can have more than one (or "many") professor, as is the case when professors co-teach a course, and so the relationship between the two in this case would be N:M.
- With respect to the one-to-many type, there is no distinction here between a one-to-many and a so-called many-to-one. In other words, it does not matter to which question the answer "one" or "many" belongs. Some DBMS make the distinction, but this

[5] Relationships that span more than two tables are beyond the scope of this text; however, they are very similar to relationships between two tables, and understanding the latter can help understand the former.

is for purely practical purposes, such as determining which table "drives" the relationship.[6]

Let's apply our strategy to the revised database design in figure 3. We will start with Course and Section. These two tables clearly relate to each other since a section is an instance of a course. Let's ask the two questions:

Q1: For each course (row in the Course table), how many sections (rows in the Section table) can there be, one or many?

Each course can have many sections. As can be seen in the table of figure 2, Introduction to Jazz has sections on Mondays and Wednesdays and on Tuesdays and Thursdays. On the other hand, Music Theory has only one section, but it *could* have more than one, perhaps if another room or instructor were available, or if the demand were higher, and so on. The answer to Q1 is therefore *many*. Now let's look at the second question.

Q2: For each section, how many courses can there be, one or many?

It is possible to imagine a school in which a group of students assembled in a room with a particular professor might be studying both Nuclear Physics and Quantum Mechanics. In this case the answer to Q2 would be *many*. We know that the proper answer depends on an accurate understanding of the mini-world. In our mini-world, we postulate that each section will be dedicated to one course. So, in our case, the answer to Q2 is *one*.

We can conclude that the relationship between Course and Section is of the type 1:N. Using the same approach, we can see that the relationship between Section and Student is of the type N:M (for each section there can be *many* students, and for each student there can be *many* sections). The relationships between the selected tables in our example so far are of the types shown in figure 4. For now, we will leave the Professor table out of the picture. We will also leave the 1:1 relationship type for later.

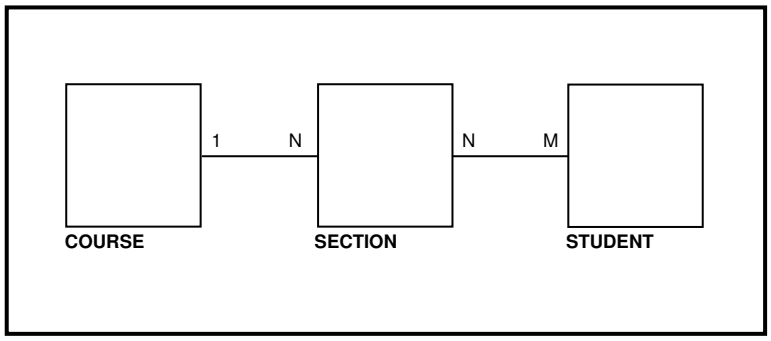


Figure 4 There is a 1:N relationship between Course and Section and a N:M relationship between Section and Student.

[6] Many DBMS allow the user to move in either the table on the one side of a 1:N relationship or in the table on the many side. Depending on the row selected by the user in the table the user is browsing, the DBMS displays the corresponding row(s) in the other table. The table in which the user is moving is often referred to as the *driving* or *parent* table. If the driving table is the one on the many side, the DBMS might refer to the relationship as a many-to-one. However, this does not change the fact that for each row in one of the tables, there can be many rows in the other, and that for each row in the other table, there can be only one row in the first. Which table is "driving" is irrelevant to the type of relationship that exists between the two!

It is natural to wonder at this point why we chose to relate the tables in the way we did. More specifically, why did we not relate Student to Course? Do students not take courses, and are these two entity-types not therefore related? Of course, they do and they are. We could relate Student to Course directly, and then Course to Section, but then we would loose some information. For a given student, we would know the courses that the student is taking, but we would not know the sections, since for each course there could be many sections. By relating the tables in the way we did, we have all the information we need. For each student, we know the section, and, since each section points to only one Course, we also know the course.

There is only one other topic we must consider before we are ready to discuss the issue of how to represent relationships in the database.

Unique Identifiers, Keys, and Primary Keys

As we have seen, each section in the Section table points to only one course in the Course table. That section has to be able to locate that one, particular, unique course in the Course table. It can only do so if each course is somehow uniquely identified. This is done by means of a *unique identifier* or *key* (thought the term *key* is synonymous with *unique identifier*, it is not synonymous with *primary key*, as we shall soon see). A unique identifier, or key, is a column in the table in which no two rows have the same value. Let's look at an example.

Imagine a table of products as shown in figure 5 with a number of attributes, one of which is Product Name and another of which is Product Category. Different products can belong to the same category, as is the case between Ka-Pow Tea and Meh-Low Tea. This means that there can be duplicate values in Product Category. On the other hand, no two products can be called Ka-Pow Tea! Product Name is a key in the Product table because no two products can have the same value in that column.

Product Name	Category	Unit Cost	
Jasmine Delight	Rice	1.50	
Ka-Pow Tea	Tea	2.45	
Morning Cream	Dairy	.60	
Meh-Low Tea	Tea	2.35	
Morning Mist	Toiletry	2.12	

PRODUCT

Figure 5 Product Name is a unique identifier (key) because no two rows in the table can ever have the same value in that column.

Tables may have more than one key. Imagine a table called Parking Lot, in which the entities are all the city parking lots, and in which the attributes are Lot Name, Address, and Capacity. In such a table, both Lot Name and Address would be unique attributes, or keys. When this happens, we call each key a *candidate key*.[7] But why are they called candidates? For what position are they being considered? They are being considered for the position of *primary key*. A primary key is a key that is *chosen* to act as the representative of the entities in the table. We could choose Product Name to uniquely identify each product, and we could choose either Lot Name or Address to uniquely identify each parking lot. So, should we use Product Name for the Products table, and, say, Lot Name for the Parking Lot table? Nope! In reality, they are all poor choices for two reasons:

1. **They are too big.** As will soon become clear, and as we have been implying all along, we will be duplicating the primary key values into what you will come to understand as a *foreign key* in another table. The larger the primary key, the more data we will be forced to duplicate.
2. **They can all change**. Remember that the primary key holds the identity of an entity. Now, suppose a new products manager decides to rename Ka-Pow Tea to Ker-Bam Tea. Should this change the identity of the product? In other words, is this now a new product? Of course not! Primary keys should not be tied to values of attributes, because we do not want an entity's identity to change simply because its attributes have changed. Another way to put this reason is that it is best to make a distinction between primary keys and data. The values in primary keys should not be considered data, because they do not describe the entities; they identify them. This distinction will become very useful later in the course when we discuss normalization.

Because of these considerations, the best primary keys are usually artificial values that have no inherent connection to the data in the other attributes (furthermore, it is best to use numbers, since most DBMS can handle automatic increments of numeric values for primary keys). An assigned number, such as our social security number, can be relatively small, and, because it is not inherently related to its entity, it will never need to change (the IRS can find you even if you change the color of your hair!).

Do all tables require primary keys? In principle, the answer is yes, since each entity has a distinct existence. A distinct existence implies a distinct identity. Identity is represented by means of a primary key, and so all tables should have a primary key. As we will see later in the course, *wea*k entity-types do not need primary keys because they are identified in another way. But, and again as we shall see, even weak entity-types can benefit from having primary keys.

Figure 6 shows our tables with primary keys added.

[7] Keys are sometimes referred to as candidate keys even if there are not multiple keys in a table. In this course, the term candidate key will refer only to keys in tables where there are other keys.

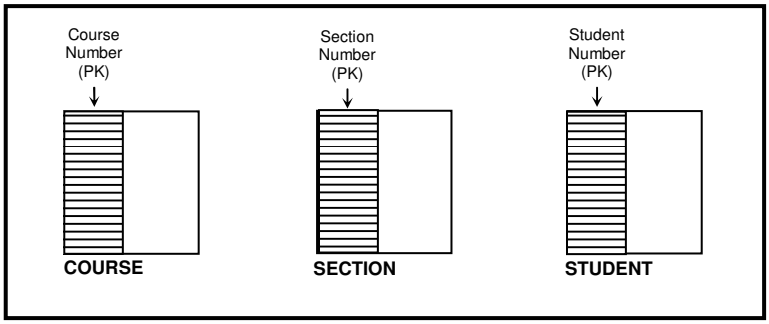


Figure 6 Because all entities have distinct existences, each one has a distinct identity. The primary key represents this identity in the database, and so all tables (except weak entity-types, as we will discuss later) should have a primary key. This figure shows our tables so far with primary keys.

Now that we have seen the types of relationships that exist among our tables and have explored the concept of the primary key that will allow us to look up and therefore connect entities from different entity-types, we are ready to turn to the issue of how to represent relationships in the database. As we have mentioned, each type of relationship is represented in a different way. Let's look at the 1:N relationship between Course and Section, first.

Representing a 1:N Relationship

To represent the 1:N relationship, we follow two steps, as listed in Table 1.

Step	Do this:
1	Create a *primary key* in the table on the one side of the relationship.
2	Create a corresponding *foreign key* in the table on the many side of the relationship.

Table 1 A 1:N relationship is represented by means of a primary key and a foreign key.

We satisfied the requirement of step 1 in the previous topic, in which we created primary keys for all of our tables. We created a Course Number attribute for the primary key in the Course table. To satisfy the requirement of step 2, we also include a column for Course Number in the Section table. We call this column the *foreign key*. In a 1:N relationship, a foreign key is a column in the table on the many side that only has values that can be found in the primary key in the table on the one side. This allows us to assign a record from the table on the one side to a record in the table on the many side, as shown in figure 7.

Figure 7 shows our Course and Section tables with primary and foreign keys. In figure 7, we can see more clearly how the primary/foreign key combination is used to relate the two tables. Let's suppose we want to know the number of units for the course taught in a particular section. The number of units is an attribute in the Course table. We need only take the course number that appears in the row of that section (foreign key) and look it up in the course number column in the Course table (primary key). Likewise, if we want to get a listing of all the sections offered

for a particular course (primary key), we need only filter the Section table to show only those rows with the proper course number (foreign key).

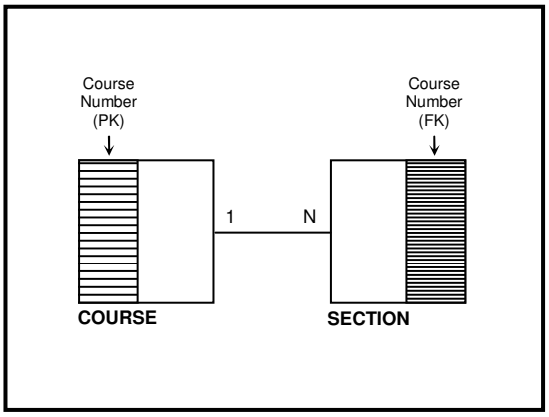


Figure 7 In a 1:N relationship, we create a primary key (PK) in the table on the one side of the relationship and a corresponding foreign key (FK) in the table on the many side.

Figure 8 shows an example of the two tables with values in both primary and foreign keys.

Course Number	Course Name	Units	Fee
MJ101	Intro to Jazz	3	500
CS303	Databases	4	650
FA569	Photography	3	450
MG521	Music Theory	4	650
LS709	Molecular Physiology	4	650

COURSE

Days	Time	Location	Course Number
M,W	7PM	Rm A	MJ101
T,Th	8AM	Rm B	CS303
M,W	9AM	Rm A	FA569
M,W	2PM	Rm C	CS303
T,W	3PM	Rm A	MJ101

SECTION

Figure 8 Our two tables, Course and Section, with primary and foreign keys (Course Number). To assign a section to a course, we enter the course number into the section table. Notice that a course number, while unique in the Course table, can appear many times in the Section table.

The following points concerning primary keys and foreign keys are worth noting:[8]

- Only the primary key is unique; the foreign key is not. In fact, that is the whole point! A value must be able to appear many times in the foreign key column to represent the many side of the relationship.
- Although we are using the same name (Course Number) for both primary key and foreign key, most DBMS allow different names to be used.[9] What is important is that they both contain the same data.
- Strictly speaking, the foreign key is not considered a "key." As we defined it above, key is another term for unique identifier. Thus, a foreign key is not a key; it is a set of values from the key of another (foreign) table.

Let's look at the N:M relationship, next.

Representing a N:M Relationship

The solution we developed for the 1:N relationship does not work for the N:M relationship. First of all, there is no table on the one side of the relationship, so we can not even get past step 1! But even if we were to force the issue, we would not have a good solution; let's say we create a primary key in each one of the tables with a corresponding foreign key in the other table. Because the relationship is of the type N:M, the primary keys in each table would need many corresponding foreign keys in the other table. As we will eventually see, this would amount to creating *multivalued attributes*, a violation of First Normal Form.[10] On an intuitive level for now, consider the complications of having to work with several foreign keys. How many should we create? How many students will we allow per section? Of course, this will depend on several factors, such as room size, number of teaching assistants and other resources available. If we create fewer foreign keys than we need, we may not be able to represent all the sections for a particular student, or all the students for a particular section. If we create more than we need, we will be wasting storage space; if we create fewer than we need, we will not be able to represent the relationship properly. Clearly, this approach is not a good one.

In order to properly represent a N:M relationship, we must break it down into two 1:N relationships by means of a third table. Table 2 lists the steps to take when we encounter a N:M relationship, and figure 9 shows the three tables involved in representing the relationship between Section and Student.

[8] There is much more to the topic of keys (primary keys, foreign keys, super keys, candidate keys) than can be treated in this article. For a deeper treatment of this topic, please refer to the references outlined under footnote 2.

[9] The name of an attribute of an entity-type (table column) is actually the *role* played by the domain of the attribute. *Domain* is the set of possible values that an attribute can assume. Suppose we are dealing with a database which models the activities of a large, interstate shipping company. Suppose further that such a database has a table of shipments. Since our shipping company can ship goods from any state to any state, there may be a column in the table of shipments called "Ship To State" and another called "Ship From State." Although both of these columns share the same domain (the set of all US states), each column represents a different role that the same domain plays. Similarly, a table of shippers might have a Shipper ID column as its primary key and a corresponding foreign key called "Ship Via" in the invoice table. Although the domains are the same, they play different roles in the different tables. In the shipper table, the role of the domain is one of identity, and in the invoice table it is the role of "the way the order is shipped."

[10] Please see footnote 2 for references that discuss Normal Forms.

Step	Do this:
1	Create a primary key in both tables.
2	Create a third table.
3	For each of the primary keys created in step 1, create a corresponding foreign key in the third table.

Table 2 Follow these three steps to represent a N:M relationship.

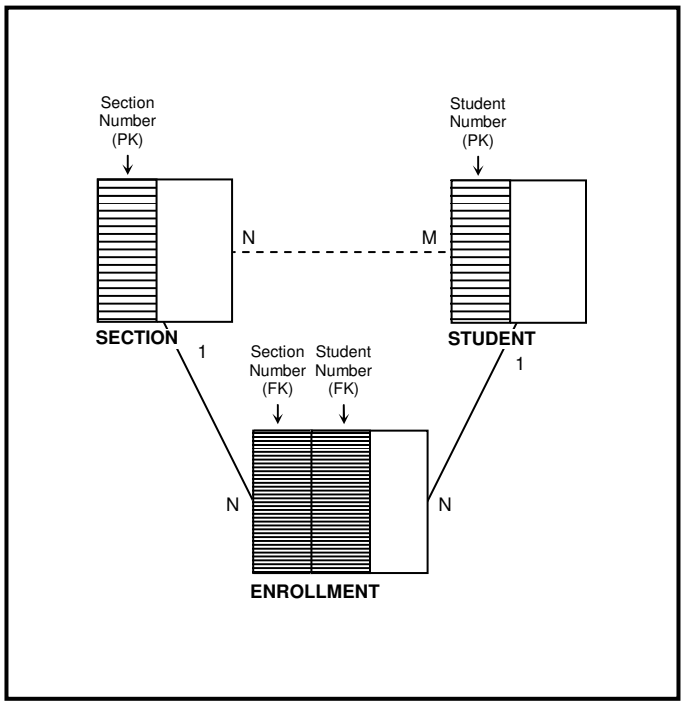


Figure 9 In a N:M relationship, we create a primary key in each of the tables and corresponding foreign keys in a third table.

The following points concerning the N:M relationship are worth noting:

- Several names are given to the third table, including but not limited to, *intersection table, intermediate table, combination table, relationship table* and even *join table*. The term *join table* is particularly ill-suited, because a join is an operation of a query. While a discussion of the operations of a query is beyond the scope of this text, suffice it to say that the term *join* is not synonymous with *relationship*. Because the so-called third table embodies a relationship, the term *join* is inappropriate. In this text, we will settle on the use of the term *intersection table*.

- Intersection tables are often referred to as relationships since they describe the relationship between two entity-types.
- "Resolving" a N:M relationship produces two 1:N relationships where the many sides of both the resulting relationships are on the intersection table, and the one sides are on the original tables.
- The primary key of an intersection table is usually, but not always, the combination (or *concatenation*) of both of its foreign keys.[11]
- Often, the intersection table will play a role that goes beyond simply providing a means by which to represent the relationship between two other tables. In our example, the intersection table holds a record of enrollments. It is often appropriate to include other columns in this table, as long as they contribute to the description of, in our case, an enrollment (see below). When this is the case, the database design benefits if the intersection table has its own distinct primary key attribute.[12]

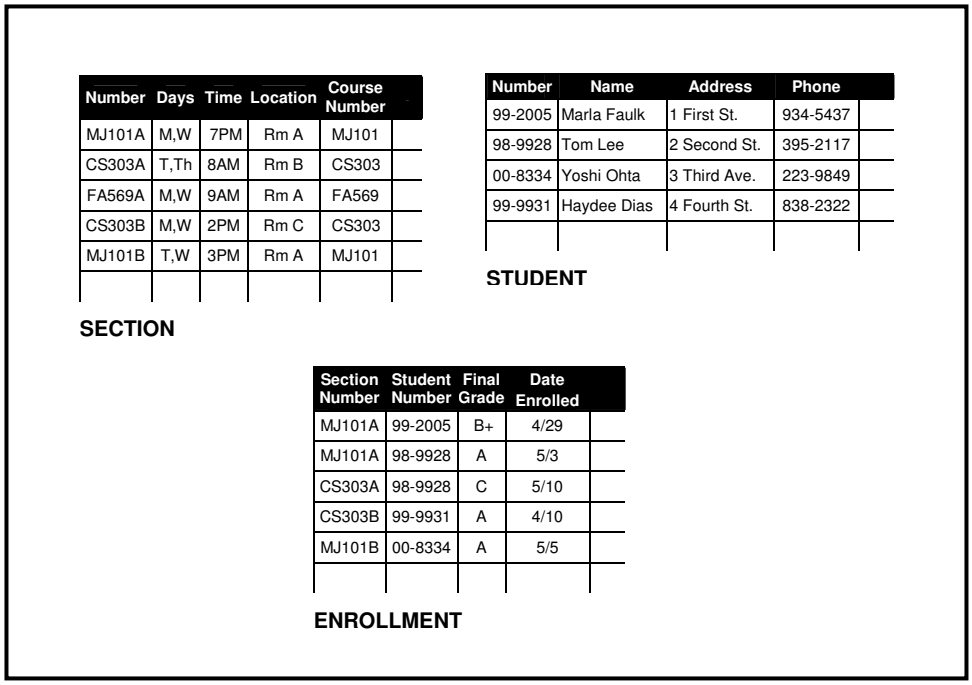
Figure 10 provides an actual sample of enrollments. Each time a student enrolls in a section, a row is added to the Enrollment intersection table. In the example provided in figure 10, the first two rows of the Enrollment table show that the same section can have many students. The second and third rows show that the same student can be enrolled in many sections.

Note that the final grade appears in the intersection table. This is because it does not describe a student or a section, but rather a student's performance (or enrollment result) in a section, and is therefore an attribute of the relationship between a student and a section.

We can see in figure 10 that it is possible to ask questions relating the data in Section and Student by using the Enrollment intersection table. As an example, suppose we would like a listing of the names of all students enrolled in a particular section. We need only filter the Enrollment table to show only the rows that have that particular section number in the Section Number column and then look up the corresponding student numbers in the Student table. In a similar fashion, we can find all classroom numbers in which a particular student attends class. We simply filter the Enrollment table to show only the rows that have that particular student number in the Student Number column and then look up the corresponding section numbers in the Section table.

[11] This works in our example if we assume that the same Section table is used throughout the life of our college and is not replaced with a new Section table for each term. If it were replaced each term, then the same section number could appear in the different Section tables. This could be resolved by also replacing the Enrollment table with a new one each term! To make our design clearer in this regard, we might add a "Term" column in the Section table. Even so, the primary keys of some intersection tables simply do not work well as concatenated primary keys. Such is the case with an Orders table that intersects Salesperson, Customer, and Shipper. Clearly, two orders can have the same of all three.

[12] When we design the database to reflect the realities of the mini-world like this, we make it a more faithful model, and thus one that more accaurately change as the mini-world changes, or that can grow as the mini-world expands to other areas of interest.



Number	Days	Time	Location	Course Number
MJ101A	M,W	7PM	Rm A	MJ101
CS303A	T,Th	8AM	Rm B	CS303
FA569A	M,W	9AM	Rm A	FA569
CS303B	M,W	2PM	Rm C	CS303
MJ101B	T,W	3PM	Rm A	MJ101

SECTION

Number	Name	Address	Phone
99-2005	Marla Faulk	1 First St.	934-5437
98-9928	Tom Lee	2 Second St.	395-2117
00-8334	Yoshi Ohta	3 Third Ave.	223-9849
99-9931	Haydee Dias	4 Fourth St.	838-2322

STUDENT

Section Number	Student Number	Final Grade	Date Enrolled
MJ101A	99-2005	B+	4/29
MJ101A	98-9928	A	5/3
CS303A	98-9928	C	5/10
CS303B	99-9931	A	4/10
MJ101B	00-8334	A	5/5

ENROLLMENT

Figure 10 The N:M relationship between Section and Student using the intersection table, Enrollment.

Figure 11 shows our database design so far. We can see in figure 11 that we can now ask questions that span several of our tables. Can you see how you might answer questions such as, "What are the names and addresses of all students taking courses with more than 3 units?" We would first filter the Course table to show only courses where the number in the Course Units column is greater than or equal to 3. Second, we would find these same course numbers in the Course Number column of the Section table and display the corresponding section numbers. Third, we would find these same section numbers in the Section Number column of the Enrollment table and display the corresponding student numbers. Finally, we would find these same student numbers in the Student table and display the corresponding names and addresses.

While this procedure might seem unreasonably complicated, the good news is that we do not have to perform it! It is the job of the DBMS to do this. All we have to do is set up our design properly, and the DBMS will do the rest. We will see this when we look at queries later in this course.

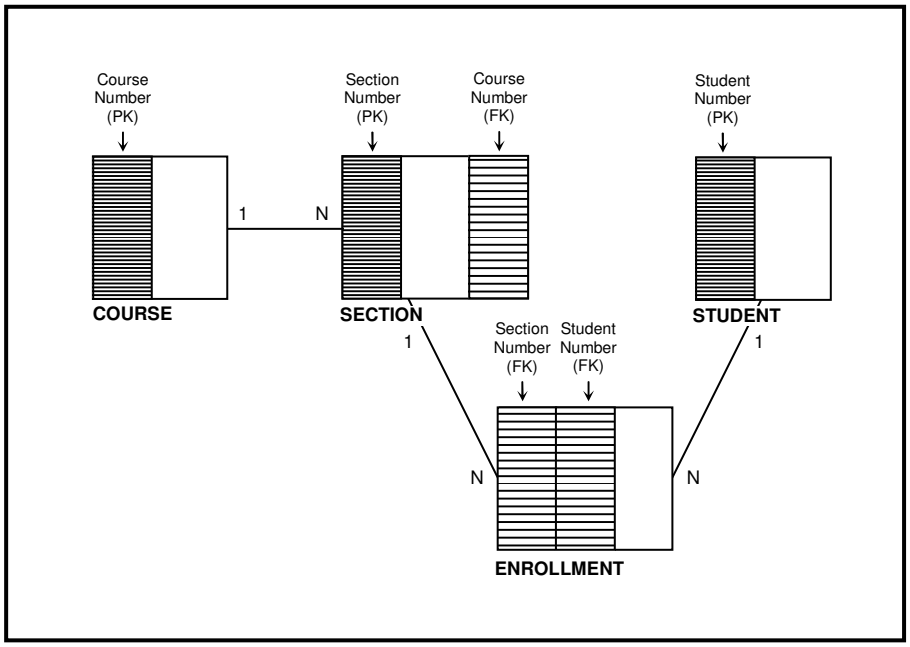


Figure 11 Our database design showing all tables and their relationships so far.

Finalizing the Design

What about the Professor table? How does it factor into our design? We will start by relating the Professor table to the Section table, since professors teach sections. A professor can teach many sections, and a section can be taught by only one professor, so the relationship between Professor and Section is of the type 1:N and, accordingly, we create primary and foreign keys as shown in figure 12.

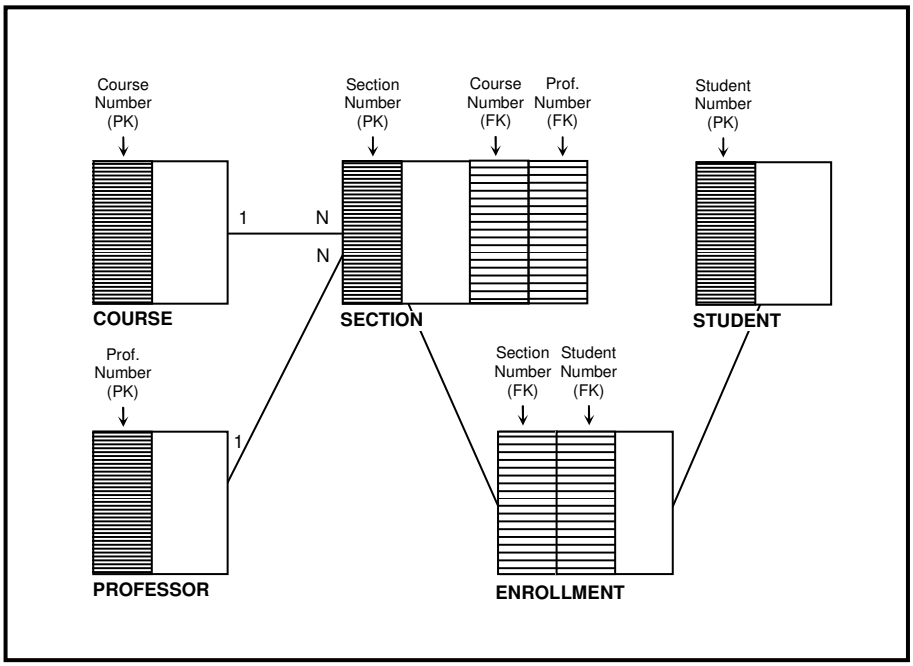


Figure 12 The Professor table is related to the Section table in a 1:N relationship.

Interestingly enough, this maneuver has resulted in two intersection tables in our database! The Enrollment table is one of them. Can you see the other?

We can see in figure 12 that the Section table also serves as an intersection table for a N:M relationship between the Course and Professor tables. There is a primary key in both the Course and Professor tables and corresponding foreign keys for each in the Section table. What does this mean intuitively? It means that a professor can be teaching many courses, and that a course can be "being taught" by many professors. We will explore this in greater detail in just a moment. For now, notice that in a well-designed database, relationships sometimes reveal themselves even if the database designer had not originally seen them!

Now, let's go back and explore the implications of our newly discovered N:M relationship. The new intersection table, Section, does not show which courses a professor is *qualified* to teach, only the courses a professor *is teaching*. How do we represent the fact that professors are qualified to teach many courses, and that courses can be taught by many professors? This seems to be another N:M relationship between the same two tables, Course and Professor! Two relationships between the same entities? Why not? Think of relationships in your own life. Have you ever played golf with your boss or used a book as a pillow? You may have a boss/subordinate relationship with your boss when at work, but on the golf course, your relationship is one of friends. In fact, there are often several relationships that we maintain with other people in our lives; there is nothing surprising about that! It seems that professors and courses are related in two N:M relationships: In one N:M relationship, a professor can *be qualified* to teach many courses, and a course can *be taught* by many qualified professors. In the second N:M relationship, a professor can *be teaching* many courses, and a course can be *being taught* by many professors. It seems we need a second intersection table between Course and Professor that shows us which professors are qualified to teach which courses. Our revised design is shown in figure 13.

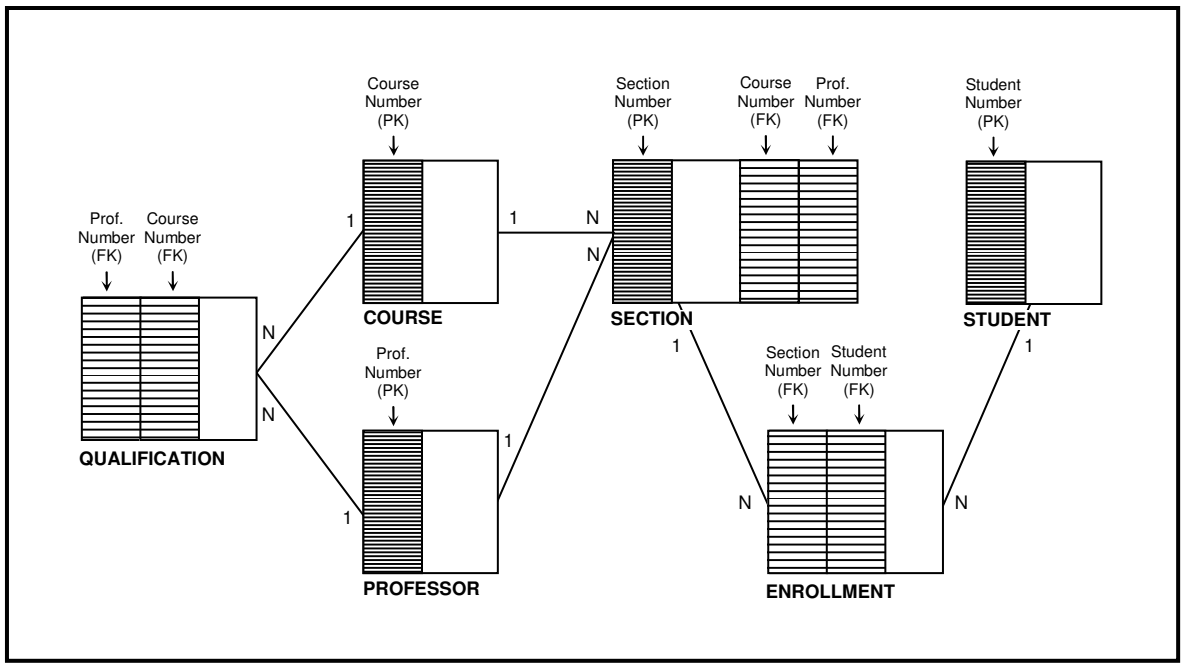


Figure 13 Our final design showing the two N:M relationships between Course and Professor.

Where Does Our Database Go from Here?

One of the characteristics of a well-designed database is that it allows room for growth. Consider how you might enhance our design to represent the following:

- Professors belong to one department.
- Courses belong to one department.
- Professors can be qualified to teach courses in different departments.
- Departments offer majors.
- Students are assigned one major.
- Certain Professors can be qualified to advise for certain majors.
- Students are assigned one advisor for their major.

The 1:1 Relationship

The 1:1 relationship is a special case of the 1:N relationship. To represent a 1:1 relationship, we create a primary key in at least one of the tables, and a corresponding foreign key in the other table. The foreign key is usually also the primary key of the second table, but it does not have to be.

To illustrate, let's suppose that our school wants to send a questionnaire to the students, and that we can make the following assumptions:

1. All the questions on the questionnaire are multiple-choice, and each question has only one possible answer.
2. Each student receives one questionnaire and can therefore submit only one completed questionnaire as a response.
3. Responding to questionnaires is optional, and as a result, we can expect a very low response rate. Let's say that for every 100 questionnaires that we send out, we expect to get five or ten back. We are expecting at most a 10% response rate.

These assumptions allow us to create a new table, Response, as shown in figure 14, where each column represents a question in the questionnaire, and each row the responses to all questions by a certain student. This table relates to Student in a relationship of type 1:1 (for each student there can be only one questionnaire response, and for each questionnaire response there can be only one student).

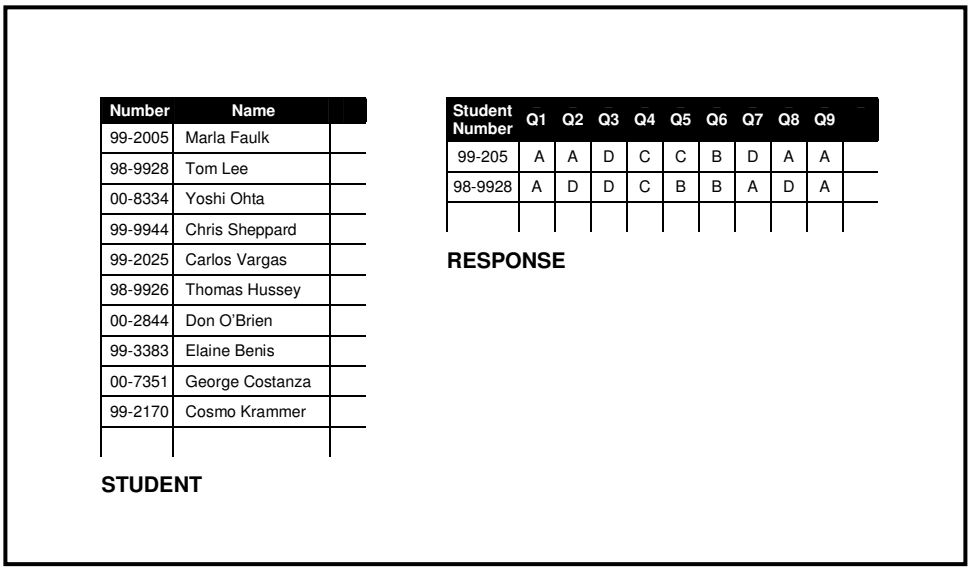


Figure 14 In a 1:1 relationship, we create a primary key in at least one of the tables and a corresponding foreign key in the other table. The foreign key usually "doubles-up" as a primary key for its table.

When we encounter a 1:1 relationship, we have to ask ourselves why not simply make one table out of the two? After all, this will not cause any data anomalies as we saw in the table of figure 2. The reasons for using 1:1 relationships are practical in nature and not because failing to do so would violate any of the rules of database design.[13]

In our particular example, assumptions 1 & 2 allow us to create a 1:1 relationship, but assumption 3 strongly suggests that we do. If we do not—if we simply expand the Student table instead, so that it includes columns for the questions—we are going to end up with lots of empty space in our combined table. According to our expectations for a 10% response rate, about 90% of

[13] Not using a 1:1 relationship where one is warranted, however, can result in a *poor* design.

the combined table will be empty space that is reserved for answers that we never expect to receive.

Other reasons for creating a 1:1 relationship include security and data distribution. While it is possible to secure certain columns in a table, it is often more practical to secure an entire table. In an environment where a single table may be distributed throughout various locations, such as an employee table in an organization which has its accounting department in one state and its benefits processing center in another, processing employee records may at times involve 1:1 relationships.

The steps for creating a 1:1 relationship are similar to those for a 1:N relationship. Although both tables may have the same primary key (as in the example above where Number and Student Number are the same values), the primary key in one of the tables acts as the foreign key in the relationship. Which one?

If you look at the relationship between Course and Section once again, you will notice that it is acceptable to have a course that does not have sections (such an event would mean simply that such a course was not being offered). We can say that not every row in the Course table participates in the relationship with the Section table. We say that the Course table has *partial participation* in the relationship with the Section table. The same cannot be said for the Section table. It is *not* OK to have a section that refers to none of the courses in the Course table. In other words, every value in the foreign key must match a value in the corresponding primary key. This means that every row in the Course table participates in the relationship with the Course table. We say that the Section table has *total* participation in the relationship with the Section table.

The situation is similar in the case of the 1:1 relationship. The table with total participation is the table with the foreign key. In the case of our example of the questionnaire, the primary key of the Response table acts as the foreign key in the relationship with Student, because the Response table has total participation in the relationship.

Summary

We saw that a database is, 1) an organized collection of related data, 2) a model of an aspect of an environment in which we have an interest, and 3) a tool that gives us a means by which we can transform data into information. We saw that the fundamental building-block of a database is the table, and that, in general, a table is required for each entity-type the database needs to represent. Exceptions are entity-types that share the same attributes and that can share entities. These can be represented in the same table. We further saw that in order to represent the relationships that exist between two entity-types, we must identify the type of relationship in question, because we use a different strategy to represent each relationship-type. We saw that there are three relationship-types: one-to-one, one-to-many, and many-to-many. We identify which type of relationship exists between two related tables by asking the two questions shown in table 1.

We discussed the concepts of unique identifier and keys, including candidate keys and primary keys, and we determined that artificial numeric values are the best choices for primary keys. This is because a primary key serves to represent the identify of an entity in the database, and using values that are inherently tied to the data values of an entity can cause problems. We also saw that in every relationship there is a primary key/foreign key pair, in which the foreign key is used to refer to a primary key value in its corresponding table.

When we encounter a 1:1 relationship between two tables, we use the primary key in the table with partial participation as the primary key in the relationship, and the primary key in the table with total participation as the foreign key in the relationship. When we encounter a 1:N relationship, we create a primary key in the table on the one side and a corresponding foreign key in the table on the many side. When we encounter a N:M relationship, we break it down into two 1:N relationships by means of a third table. We create primary keys in each of the two original tables, and corresponding foreign keys in the third table. The primary key in an intersection table is often the concatenation of the two foreign keys.

We saw that a table does not have to be dedicated to its duties as an intersection table. We saw that the Enrollment table, initially created as an intersection table between Section and Student, also represents an entity-type, Enrollment, which has its own characteristics. On the other hand, the Section table, initially created to represent the Section entity-type, turned out to also be an intersection table between Professor and Course.

We also saw that related tables can actually have more than one relationship between them, and that each relationship represents a different way that the two entity-types interact with each other. The example we provided for this was in how the Course and Professor tables relate to each other.

We pointed out that the intuitive approach described in this article works regardless of the DBMS being used, and that different DBMS have different ways of actually implementing multiple tables and their relationships.

By following the informal steps outlined in this article, it is possible to have a sound database design (normalized *at least* to Boyce-Codd Normal Form if not to Domain-Key; the final Normal Form!) that minimizes the chances of there being anomalies in the data, and allows the database to grow.