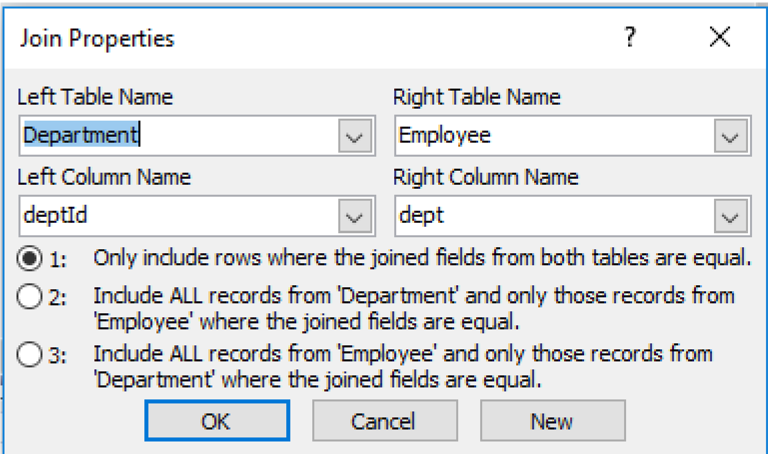


Figure 6.37: Default property is option 1

By choosing option 2 your query will include all departments, whether or not the department can join to an employee. If there is no employee for a department to join to, then the row is joined to a row of nulls. When you do this notice the change in the relationship line – it is now a directed line; this is how MS Access illustrates outer joins:

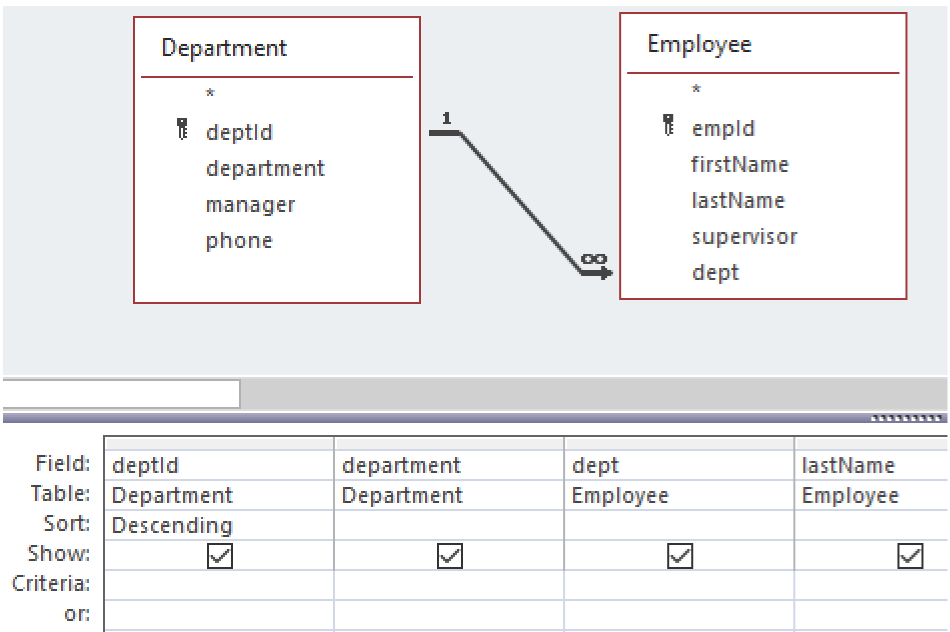


Figure 6.38: Outer join – all rows of Department

For our purposes here, we added the Special Operations department to Department. And now the first few rows of the result are:

deptId	department	dept	lastName
4	Special Operations		
3	Sales	3	Floyd
3	Sales	3	Mckay
3	Sales	3	Compton
3	Sales	3	Ford

Figure 6.39: Query result

Notice that the Special Operations department joined to a null row.

Exercises

1) Consider the Company database and list each department and the number of employees in the department.

2) Consider the Orders database:

- Create a query to list each customer and their orders (order id and order date). Are there any customers who have not placed an order?
- Modify the above query to list each customer and the number of orders they have placed (include all customers).

3) Consider the library database:

- Create a query that will list every book and the date it was borrowed. Include all books in your result.
- Create a query to list every library member and the dates they borrowed books. Include all members

6.8.3: Cartesian Product

Suppose you create a query, but without join criteria. This is easily done by clicking on the relationship line and deleting it. When criteria for matching rows is not present, then each row of one table will join to each row of the other table.

This type of join is called a Cartesian Product, and these can easily have very large result sets. If Department has 4 rows and Employee has 100 rows, then the Cartesian Product has (4×100=) 400 rows. Databases used in practice have hundreds, thousands, even millions of rows; a Cartesian Product may take a long, long time to run.

Exercises

1) Consider the Sales database and its Store and Product tables. Construct a query to list the storeID and the productID. When you add Store and Product to the relationships area there is no line joining the two tables. Run the query. Notice how many rows there are; the number of rows in the result set is the number of stores times the number of products.

2) Consider the Sales database and its Store, Product, and Sales tables. Suppose we want to obtain a list that shows for each store and product the total quantity sold. Note that the end user wants to see every store and product combination.

> Hint: An approach you can use with MS Access is to create two queries. The first of these performs a cross product of store and product (call this CP).

The second query is developed as a join between the query CP and the table Sales. CP is outer-joined to Sales in order that every combination of Store and Product is in the final result.

6.8.4: Self-Join

A self-join, also called a recursive join, is a case where a table is joined to itself.

Consider the Company database and suppose we must obtain a list of employees who report to another employee named Raphael Delaney (i.e., List the employees Raphael Delaney supervises). To do this we need to find the row in Employee for Raphael Delaney and then join that row to other rows of Employee where the supervisor field is equal to the empId field for Raphael. When we build the query in MS Access we simply add the Employee table to the relationships area twice. One copy of Employee will be named Employee_1. Consider the following query:

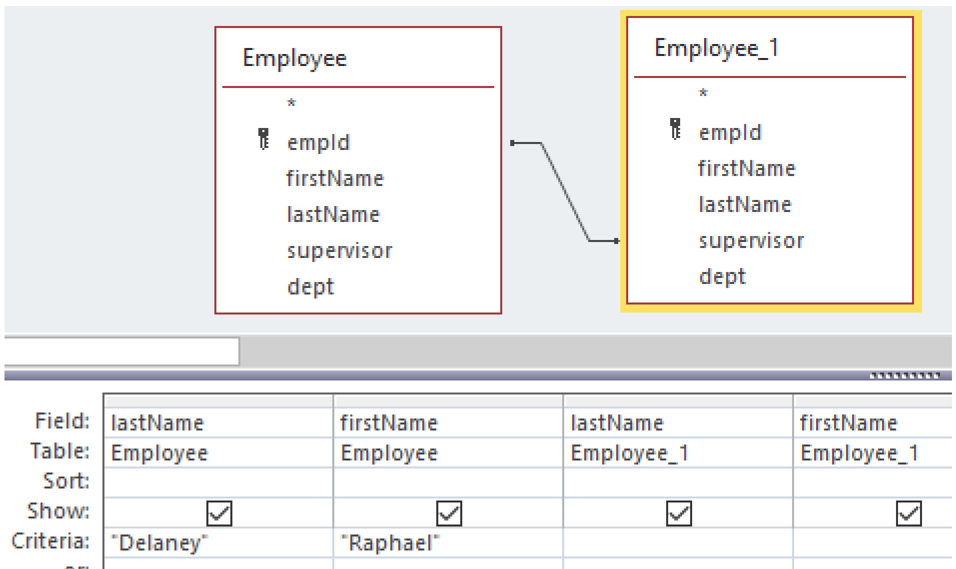


Figure 6.40: Self-join

Note the following:

- the criteria specifies the row in Employee will be that of Raphael Delaney
- the join line connects supervisor to empId and so rows of Employee_1 will be employees who report to Raphael.

Exercises

1) Consider the Genealogy database and develop queries to obtain:

- the father of Peter Chan.
- the mother of Peter Chan.
- the father and mother of Peter Chan.

2) Consider the Orders database and the Employee table:

- Write a query to list the employee who does not report to anyone.
- Write a query to list each employee and the number of employees they supervise.

6.8.5: Anti-Join

Suppose we need to list persons in our Company database that are not supervising anyone. One way of looking at this problem is to say we need to find those people that do not join to someone else based on the *supervises* relationship. That is, we need to find those employees whose employee id does not appear in the supervisor field of any employee.

To do this with MS Access, we can construct a query that uses an outer join to connect an employee to another employee based on employeeID equaling supervisor, but where the supervisor value is null. That is, we are looking for an employee who, in an outer join, does not join to another employee. See the query below:

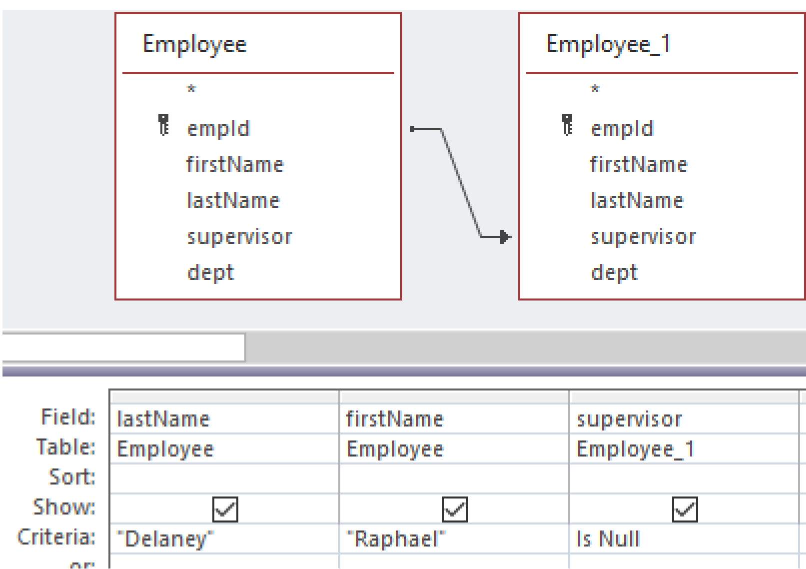


Figure 6.41: Anti-join

This query involves a join, specifically an outer join, and because it retrieves that rows that do not join, it is sometimes referred to as a special case – an *anti-join*.

Exercises

1) Consider the Company database and develop a query to count the number of employees who do not supervise anyone.

2) Consider the Library database. Create a query to list books that no one borrowed.

3) Consider the Library database. Create a query to list members that have not borrowed a book.

6.8.6: Non-Equi Join

A non-equi join is any join where the join criteria does not specify equals, "=".

Suppose we wish to list all persons in the Genealogy database who are younger than, say, Peter Chan. One approach to getting the results is to join the row for Peter Chan to another row in Person where the birthdate of Peter Chan is greater than the birthdate of the other person. This type of join would be a "greater than" join as opposed to an equi- join. Proceed in the following way:

- Add Person to the relationships area twice so there is a Person table and a Person_1 table. If there are any relationship lines delete them.
- In the criteria line for Person fields: for firstName type "Peter" and for LastName type "Chan".
- In the criteria line for birthDate in Person_1 type "> [Person].[birthDate]"

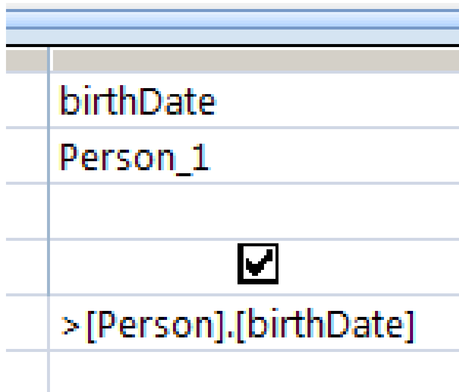


Figure 6.42: Non-equi join

In this way you are creating a "greater than" join.

- Include attributes from Person_1 to display these younger people.
- Run your query.

Exercises

1) Consider the Company database and create a query to list anyone younger than Tanya Dickson.

2) Modify the example in this section to list those people who are older than Peter Chan.

6.9: SQL Select Statement

SQL is the standard language for relational database systems. There are variations of SQL that appear in Object-oriented database systems, and elsewhere. The study of SQL is very important, and the knowledge gained here is useful in other database environments.

We will examine one SQL statement, the Select statement, used to retrieve data from a relational database. Other common data manipulation statements are the Insert, Update, and Delete used to modify or add data. Select, Insert, Update, and Delete all belong to the Data Manipulation Language (DML) subset of SQL. Another group of statements belong to the Data Definition Language (DDL) subset of SQL – these statements are used to create tables, indexes, and other structures and are discussed in a later section.

The general SQL **Select** statement syntax:

- **Select** list of attributes or calculated results[(1)]
- **From** list of tables with/without join condition[(2)]
- **Where** criteria rows must meet beyond the join specifications[(3)]
- **Group by** list of attributes for creating groups[(4)]
- **Order by** list of attributes for ordering the results[(5)]
- **Having** criteria groups must meet[(6)]

Each clause of the SQL statement has its counterpart in the Design View used by Access:

1. Attribute/calculated values are those for which Show is specified. If grouping is used, these must evaluate to a single value (group functions; grouping attribute) per group.
2. Tables that appear in the From clause are shown in the Relationships Area.
3. Specifications for the Where clause are found in the Criteria and Or rows.
4. Specifications for the Group By clause are made in the Totals row.
5. Specifications for sorting are made in the Sort row.
6. A Having clause specifies criteria that a group must meet to be included in the result. This clause is generated when you use an aggregate function with a criteria.

When you design a query you can switch between various views including SQL view. You can easily confirm through examples how the SQL statement is generated from Design View. For example, consider the following query and its SQL expression below. Note how MS Access has used names with dot-notation to fully specify fields and how MS Access has placed one criteria rows must meet in a Having clause.

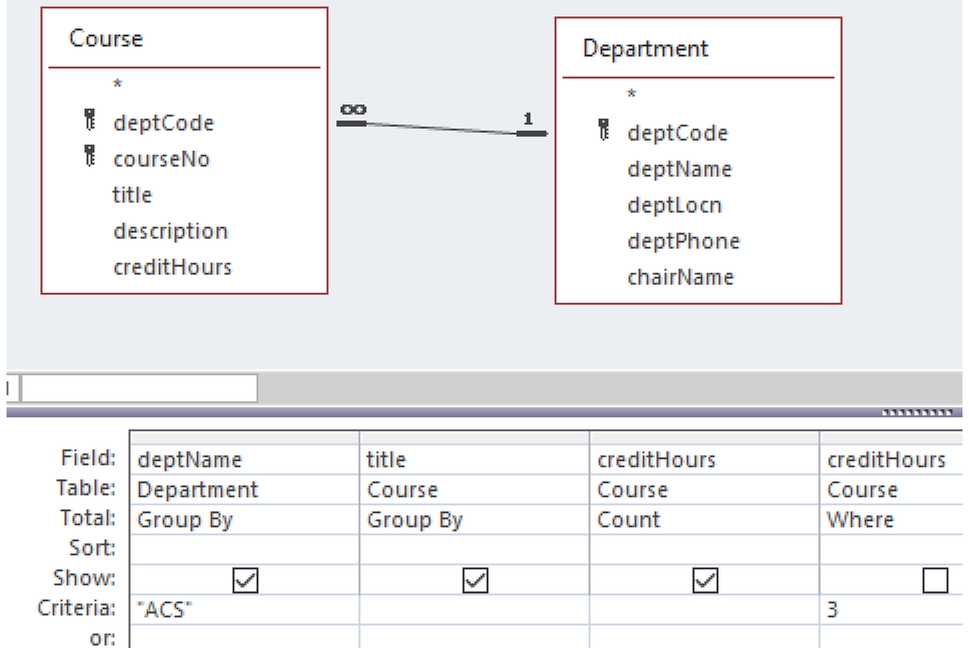


Figure 6.43: QBE and SQL Select

- SELECT Department.deptName, Course.title, Count(Course.creditHours) AS CountOfcreditHours
- FROM Department INNER JOIN Course ON Department.deptCode = Course.deptCode
- WHERE (((Course.creditHours)=3))
- GROUP BY Department.deptName, Course.title
- HAVING (((Department.deptName)="ACS"));

Exercises

Consider the following SQL statements for the Sales database and show how each statement would appear in Design View. You can confirm your result if you create a query, switch to SQL View, type the query statement and then switch to Design View.

1) SELECT Product.ProductID, Product.ProductName

- FROM Product;

2) SELECT Category.CategoryName, Product.ProductName

- FROM Category INNER JOIN Product
- ON Category.CategoryID = Product.CategoryID;

3) For the Sales database: List products in Condiments with a unit price over $4.

- SELECT Product.ProductID, Product.ProductName
- FROM Category INNER JOIN Product
- ON Category.CategoryID = Product.CategoryID
- WHERE (((Category.CategoryName)="Condiments") AND ((Product.UnitPrice)>4));

4) For the Sales database: List the number of products in each category.

- SELECT Category.CategoryID, Category.CategoryName, Count(Product.ProductID)
- FROM Category INNER JOIN Product
- ON Category.CategoryID = Product.CategoryID
- GROUP BY Category.CategoryID, Category.CategoryName;

5) For the Sales database: List products for which the total quantity sold is more than 10.

- SELECT Product.ProductID, Product.ProductName
- FROM Product INNER JOIN Sales
- ON Product.ProductID = Sales.productId
- GROUP BY Product.ProductID, Product.ProductName
- HAVING (((Sum(Sales.quantitySold))>10));

6.10: SQL Union and Union All

The Union and Union All operators merge the results of two or more queries that are given as SQL Select statements. With MS Access you must switch to SQL View to use Union/Union All

- UNION removes duplicates and sorts the results
- UNION ALL returns all values (includes duplicates) without sorting
- The output fields must be identical (number and type) for each SELECT.

The syntax for UNION of two Select's:

Union	**Union all**
SQL Select Statement$_1$	SQL Select Statement$_1$
UNION	**UNION ALL**
SQL Select Statement$_2$;	SQL Select Statement$_2$;

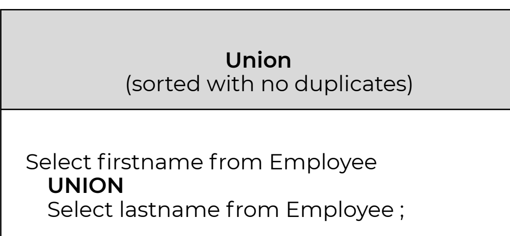
Figure 6.44: Union and Union All syntax

Any number of Select statements can be UNIONed. A requirement for using UNION is that the queries are union-compatible – the queries must retrieve the same number of fields, and fields in the same position across the multiple Select clauses must be of matching types.

Example

Consider the Employee table in the Company database. To list all names (first and last) in a single column construct two queries: one to list the first names of employees and one to list the last names of employees.

These two queries can be combined to produce a single list of names. Now, in SQL view type and run:



Union (sorted with no duplicates)
Select firstname from Employee **UNION** Select lastname from Employee ;

Figure 6.45: Union

Exercises

1) Create a query to produce a list of cities in the Orders database. The Employees table and the Customers tables have a city field.

2) Modify the example in Figure 6.45 so that duplicates are eliminated.

7. ENTITY RELATIONSHIP MODELLING

When designing a database, it is common practice for a database designer to develop an Entity Relationship model and to represent that model in a drawing, the entity relationship diagram (ERD). In this chapter we discuss the concepts required to develop an ERD and the Peter Chen notation.

Peter Chen introduced entity relationship modeling in his paper *The Entity-Relationship Model–Toward a Unified View of Data* (ACM Transactions on Database Systems, Vol. 1, No. 1, 1976). This paper can be found at http://csc.lsu.edu/news/erd.pdf. It is one of the most cited papers in the computer field and has been considered one of the most influential papers in computer science. Another later paper published in Software Pioneers: Contributions to Software Engineering (2002) is *Entity-Relationship Modeling: Historical Events, Future Trends, and Lessons Learned* and can be found at http://bit.csc.lsu.edu/~chen/pdf/Chen_Pioneers.pdf.

Entity Relationship modeling is a process used to help us understand and document the informational requirements of a system as a logical or conceptual data model. When the model is complete, we then create a physical model in some database management system (DBMS), typically a relational DBMS, or RDBMS.

7.1: Introduction

In the entity relationship approach to modeling, we analyze system requirements and classify our knowledge in terms of entities, relationships, and attributes.

Entities

Entities are the things we decide to keep track of. For example, if one considers a system to support an educational environment, one is likely to decide that we need to keep track of students, instructors, courses, etc. Typically, entities are the people, places, things, and events that we need to remember something about.

Suppose we know of four student entities and two course entities. For example, consider four students (say John, Amelia, Lee, and April) and two courses (Introduction to Art and Introduction to History). We can illustrate these in a number of ways:

As tables of information:

Students		
Name	Id Number	Phone
John	184	283-4984
Amelia	337	838-3737
Lee	876	933-2211
April	901	644-3838

Courses		
Title	Course Number	Department
Introduction to Art	661	Art
Introduction to History	765	History

Figure 7.1: Entities shown as rows in table

As sets of entities:

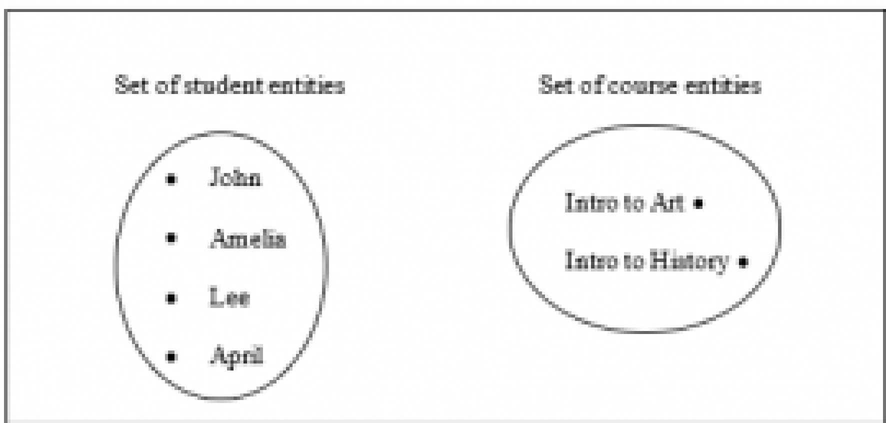


Figure 7.2: Entities shown as sets

Relationships

Entities can be related to one another and so we use *relationships* to describe how entities relate. Continuing with our educational example we know that students enroll in courses, and so this is one of the relationships we should know about. Suppose we have the two courses and four students listed previously. Suppose also that

- John and Amelia are enrolled in Introduction to Art
- John and Lee are enrolled in Introduction to History
- April is not enrolled in any course.

Below, we depict four instances of the *enroll-in* relationship by drawing a line from a student to a course. Each relationship pairs one student with one course.

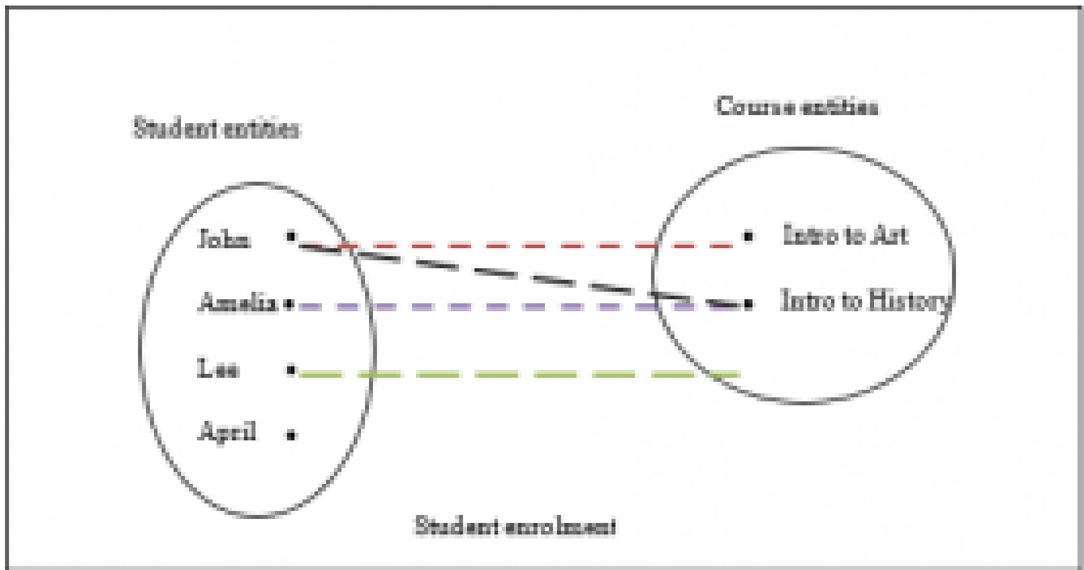


Figure 7.3: Relationships shown as lines connecting entities

Attributes

Entities and relationships have characteristics that describe them. For instance, the students in our example are described by the values for their name, id number, and phone number. As we look back at previous figures, we can see there is a student named *John* whose id number is *184* and his phone number is *283-4984*.

Courses are shown with a course title, a course number, and to belong to a department. There is a course numbered *661* that is offered by the *Art* department and it is titled *Introduction to Art*.

If we consider the enroll-in relationship we know there is a date when the student enrolled in the course and a final grade that was awarded to the student when the course was completed. For instance, we could have that *John* enrolled in *Introduction to Art* on *July 1, 2010* and was awarded an *A+* on completion of that course.

These characteristics that serve to describe entities and relationships are called attributes. We will be examining attributes in some detail. As we will see some attributes, such as student number,

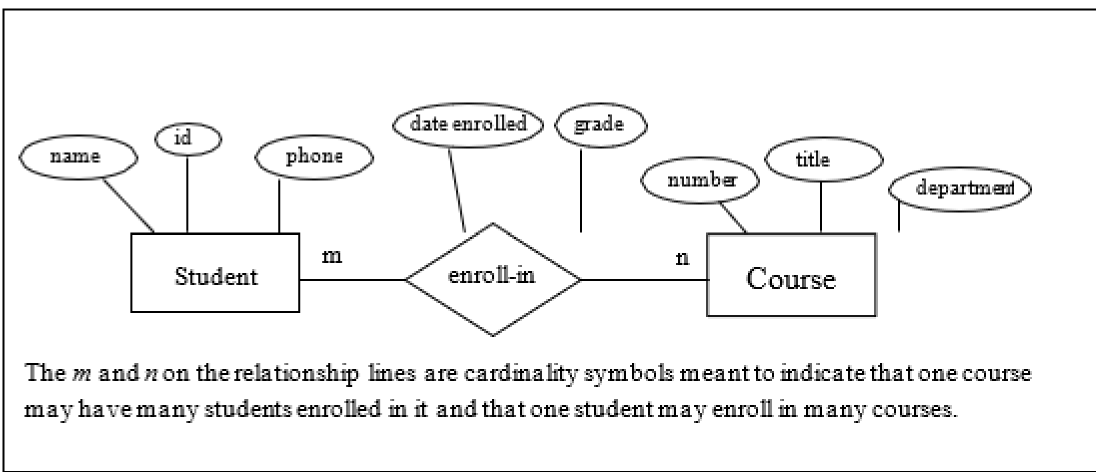
serve to distinguish one instance from another – each student has a student number distinct from any other student. Other attributes we consider to be purely descriptive, such as the name of a student – many students could have the same name.

Notation

There are many notations in use today that illustrate database designs. In this text, as is done in many database textbooks, the Peter Chen notation is used; other popular notations include IDEF1X, IE and UML. There are many similarities, and so once you master the Peter Chen notation it is not difficult to adapt to a different notation.

The following is an example of an ERD drawn using the Peter Chen notation. Note the following:

- Entity types are represented using rectangular shapes.
- Relationship types are shown with a diamond shape. Lines connect the relationship type to its related entity types with cardinality symbols (*m* and *n*).
- Attributes are shown as ovals with a line connecting it to the pertinent entity type or relationship type.



The *m* and *n* on the relationship lines are cardinality symbols meant to indicate that one course may have many students enrolled in it and that one student may enroll in many courses.

Figure 7.4: An ERD in Chen notation

The various symbols we use with the Peter Chen notation:

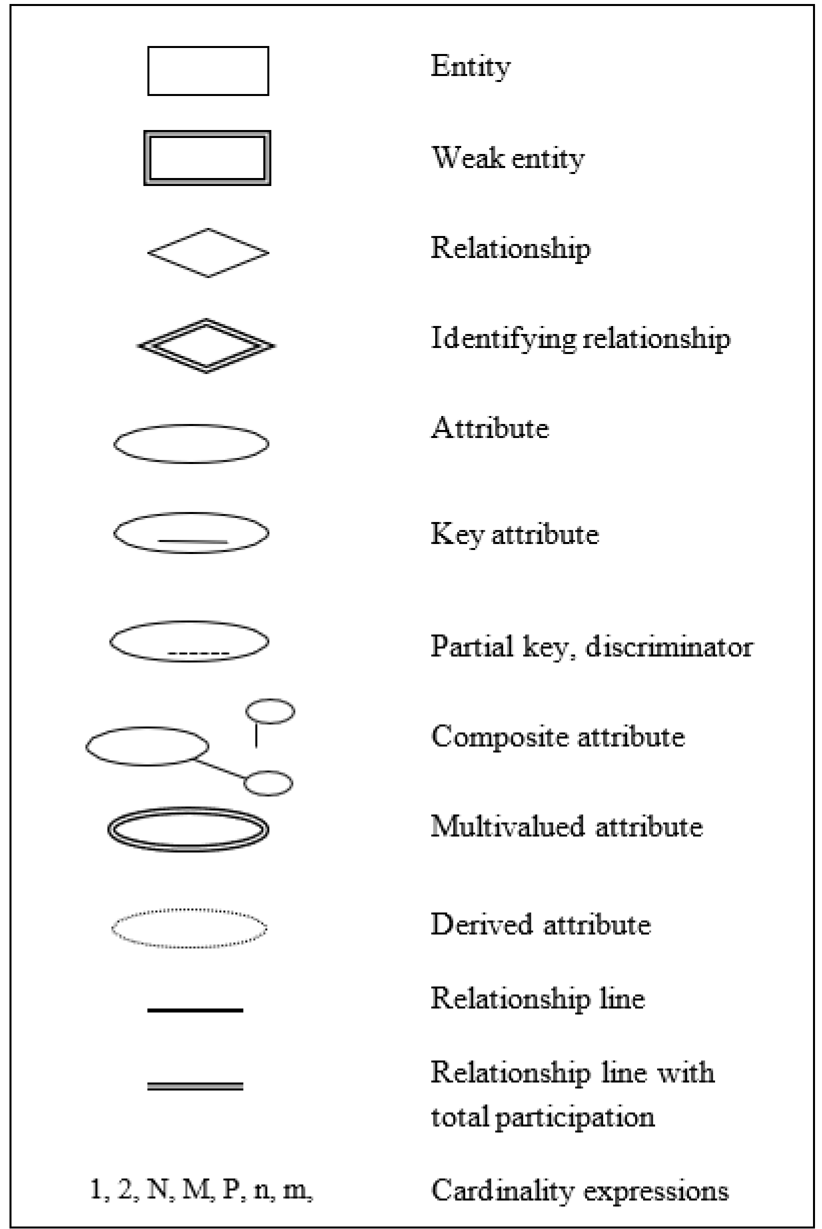


Figure 7.5: Symbols used in the Chen notation

7.2: Entities

Entities are the people, places, things, or events that are of interest for a system that we are planning to build. In the previous section we considered there were several entities: four students and two courses.

In general, we find examples of entities when we think of people, places, things, or events in our area of interest:

- People: student, customer, employee
- Places: resort, city, country
- Things: restaurant, product, invoice, movie, painting, book, building, contract
- Events: registration, election, presentation, earthquake, hurricane

Entity sets are named collections of related entities. From our example we have two entity sets:

- The Student entity set comprises at least the 4 student entities: John, Amelia, Lee, and April.
- The Course entity set comprises at least the 2 course entities: Introduction to Art and Introduction to History.

Entity sets are the collections of entities of one type. We consider an *Entity Type* to be the definition of the entities in such a set. A common convention is to name entity types as singular nouns and that, at least, the first letter is capitalized.

In an ERD entity types are shown as named rectangular shapes. For example:

The Student and Department entity types shown above are drawn with a simple single-line border. This means that they are regular (or strong) entity types that are not existence- dependent on other entity types (see the next section).

Exercises

1) Consider your educational institution. Your educational institution needs to keep track of its

students. How many student entities does the institution have? You have provided the institution with information about you. In your opinion, what attributes describe these entities?

2) Consider your place of work. The Human Resources department in your company needs to manage information about its employees. How many employee entities are there? What attributes describe these entities?

3) Consider your educational institution or place of work.

- What are some of the entity types that would be useful?
- What relationships exist that relate entity types to one another?
- What attributes would be useful to describe entities and relationships?
- Draw an ERD.

7.2.1: Weak Entities

Sometimes we know certain entities only exist in relationship to others. For example, a typical educational institution comprises a number of departments that offer courses. So, we could have a History department, an Art department and so on. These departments would design and deliver courses that students would register for. In this framework the courses exist in the context of a department, and the identifier for a course is typically a department code and course number combination. So, the history course, Introduction to History, belongs to the History department and it could be known by the identifier HIST- 765. HIST is a code representing the History department and 765 is a number assigned to the course; other departments could have a course with that same number, 765.

In these situations where the existence of an entity depends on the existence of another entity, we say the entity is a *weak* entity, and the corresponding entity type is a weak entity type. Weak entities often have identifiers that comprise multiple parts (such as department code and course number). Later we will see other aspects of an ERD that relate to weak entity types. At this time, we should be aware that weak entity types are illustrated in an ERD with a double-lined rectangle:

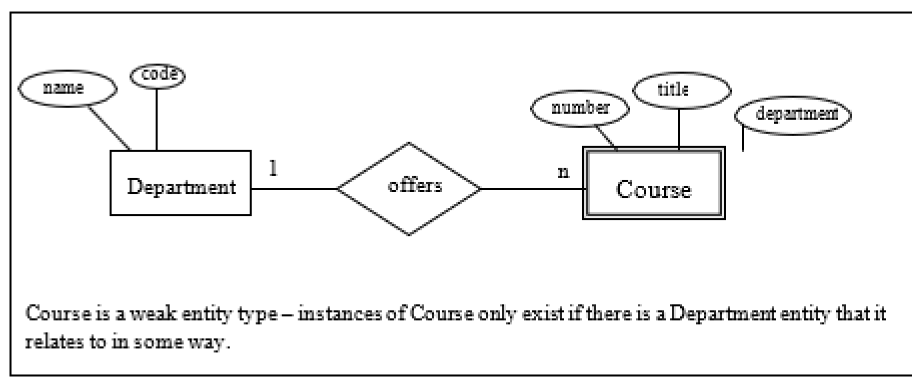


Figure 7.6: Course as a weak entity type

Often when we purchase things the vendor provides an invoice giving details of each item that is purchased (see the sample invoice below). Appearing on the invoice are detail lines specifying the product, the quantity and price. Invoice lines are things that exist only in

the context of an invoice and so each invoice line is a ***weak*** entity; the invoice lines are existence-dependent on an invoice:

Winnipeg Retail Co **INVOICE**

124 Any Street, Winnipeg
Manitoba, R3E 2E9

SOLD TO:
John Smith
124 AnyStreet
Winnipeg, MB, R3B 2E9

INVOICE NUMBER	192837
INVOICE DATE	February 20, 2013
OUR ORDER NO.	2314
YOUR ORDER NO.	7654
TERMS	Net 30
SALES REP	Jim Jones

SHIPPED TO:
same as above

QUANTITY	DESCRIPTION	UNIT PRICE	AMOUNT
10	pens	1.50	$15.00
15	pencils	2.50	37.50
		Total less taxes	52.50
		PST	3.68
		GST	2.63
			$58.80

Figure 7.7: Sample Invoice

The following includes a few attributes to show how Invoice and Invoice Line could appear in an ERD:

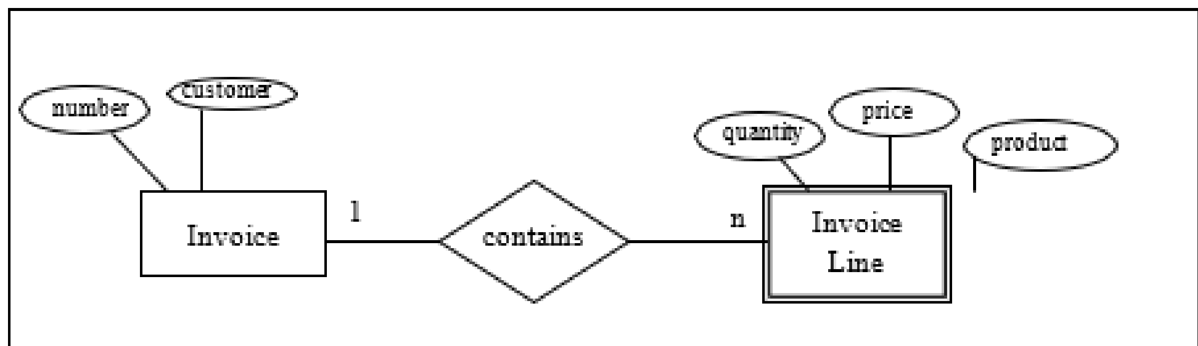


Figure 7.8: Invoices (regular entity type) and Invoice Lines (weak entity type)

Exercises

1) Consider a requirement having to do with benefits that may be given to employees of a company. Suppose employees work in a department and that each employee may have several dependents (spouse, child). Draw an ERD that includes Department, Employee, and Dependent in your design. Include attributes for your entity types.

2) When you buy items in a store you often get a cash register receipt that details the items you have purchased. Develop an ERD that includes Store, Customer, Receipt, and Detail Lines.

7.3: Attributes

Attributes are the characteristics that describe entities and relationships. For example, a Student entity may be described by attributes including:

- student number
- name
- first name
- last name
- address
- date of birth
- gender

An Invoice entity may be described by attributes including:

- invoice number
- invoice date
- invoice total

A common convention for naming attributes is to use singular nouns. Further, a naming convention may require one of:

- All characters are in upper case.
- All characters are in lower case.
- Only the first character is in upper case.
- All characters are lower case, but each subsequent part of a multipart name has the first character capitalized

Using the last convention mentioned, some examples of attribute names:

- lastName *for* last name
- empLastName *for* employee last name
- deptCode *for* department code
- prodCode *for* product code
- invNum *for* invoice number

In practice a naming convention is important, and you should expect the organization you are working for to have a standard approach for naming things appearing in a model. A substantial data model will have tens, if not into the hundreds, of entity types, many more attributes and relationships. It becomes important to easily understand the concept underlying a specific name; a naming convention can be helpful.

There are many ways we can look at attributes including whether they are atomic, composite, single-valued, etc. We consider these next.

7.3.1: Atomic Attributes

A simple, or **atomic**, attribute is one that cannot be decomposed into meaningful components. For example, consider an attribute for gender – such an attribute will assume values such as Male or Female. Gender cannot be meaningfully decomposed into other smaller components.

As another example consider an attribute for product price. A sample value for product price is $21.03. Of course, one could decompose this into two attributes where one attribute represents the dollar component (21), and the other attribute represents the cents component (03), but our assumption here is that such decompositions are not meaningful to the intended application or system. So, we would consider product price to be atomic because it cannot be usefully decomposed into meaningful components.

Similarly, an attribute for the employee's last name cannot be decomposed, because you cannot subdivide last name into a finer set of meaningful attributes.

Exercises

1) Consider that a Human Resources system must keep track of employees. If we are only including atomic attributes, what attributes would you include for the employee's name? Some possibilities are first name, last name, middle name, full name.

2) What simple attributes are used to describe courses at your institution?

3) In some large organizations where there are several buildings and floors we see room numbers that encode information about the building, floor, and room number. For example, in case the room 3C13 stands for room 13 on the third floor of the Centennial building. Suppose we need to include Room in an ERD. How would you represent the room number given that you must include atomic attributes only?

7.3.2: Composite Attributes

Consider an attribute such as full name which is to represent an employee's complete name. For example, suppose an employee's name is John McKenzie; the first name is John and the last name is McKenzie. It is easy to appreciate that one user may only need employee last names, and another user may need to display the first name followed by the last name, and yet another user may display the last name, a comma, and then the first name. If it's reasonable for one to refer to the complete concept of employee name and to its component parts, first name and last name, then we can use a *composite* attribute**.** An attribute is *composite* if it comprises other attributes. To show that an attribute is composite and contains other attributes we show the components as attribute ovals connected to the composite as in:

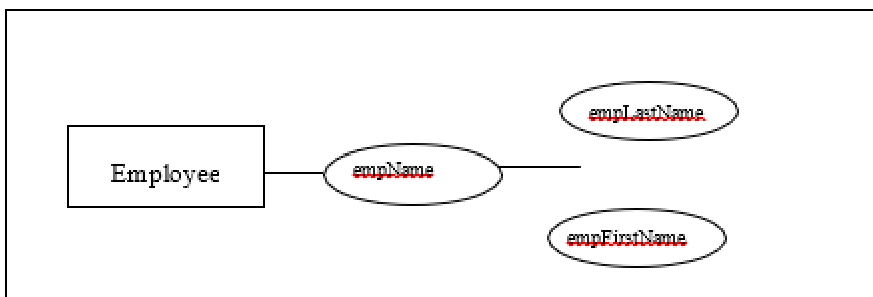


Figure 7.9: Composite attribute

Attributes can be composite and some of its component attributes may be composite as well (see exercise 3).

Exercises

1) How would you use a composite attribute to model a phone number.

2) Consider the previous exercise set. Show how we can include room number as a composite attribute that has multiple components.

3) Consider an address attribute. Show that this can be modeled as a multi-level composite attribute where the component attributes include street, city, province, country and where street includes apartment number, street number, street name.

7.3.3: Single-Valued Attributes

We characterize an attribute as being single-valued if there is only one value at a given time for the attribute.

Consider the Employee entity type for a typical business application where we need to include a gender attribute. Each employee is either male or female, and so there is only one value to store per employee. In this case, we have an attribute that is single-valued for each employee. Single-valued attributes are shown with a simple oval as in all diagrams up to this point. In all of our examples so far, we have assumed that each attribute was single-valued.

Exercises

1) Consider a marriage entity type and attributes marriage date, marriage location, husband, wife. Each marriage will only have one value for each of these attributes. Illustrate the marriage entity and its single-valued attributes in an ERD.

2) A college or university will keep track of several addresses for a student, but each of these can be named differently: for example, consider that a student has a mailing address and a home address. Create an ERD for a student entity type with two composite attributes for student addresses where each comprises several single-valued attributes.

7.3.4: Multi-Valued Attributes

Now, suppose someone proposes to track each employee's university degrees with an attribute named empDegree. Certainly, many employees could have several degrees and so there are multiple values to be stored at one time. Consider the following sample data for three employees: each employee has a single employee number and phone number, but they have varying numbers of degrees.

empNum	empPhone	empDegree
123	233-9876	
333	233-1231	BA, BSc, PhD
679	233-1231	BSc, MSc

Figure 7.10: Employees – number, phone, degrees

For a given employee and point in time, empDegree could have multiple values as is the case for the last two employees listed above. In this case we say the attribute is *multi- valued*.

Multi-valued attributes are illustrated in an ERD with a double-lined oval.

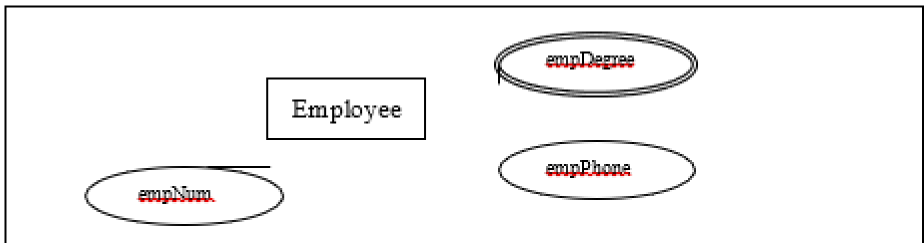


Figure 7.11: Employee degrees shown as multi-valued

We can use multi-valued attributes to (at least) document a requirement, and at a later time, refine the model replacing the multi-valued attribute with a more detailed representation. The presence of a multi-valued attribute indicates an area that may require more analysis; multi-valued attributes are discussed again in Chapter 10.

Exercises

1) Consider the employee entity type.

2) Suppose the company needs to track the names of dependents for each employee. Show the dependent name as a multi-valued attribute.

3) Modify your ERD to show empDependentName as a composite multi-valued attribute comprising first and last names and middle initials.

4) Create an ERD that avoids the multi-valued attribute empDegrees in the previous example. Hint: Consider including another entity type and a relationship for keeping track of degrees.

7.3.5: Derived Attributes

If an attribute's value can be derived from the values of other attributes, then the attribute is derivable, and is said to be a *derived* attribute. For example, if we have an attribute for birth date then age is derivable. Derived attributes are shown with a dotted lined oval.

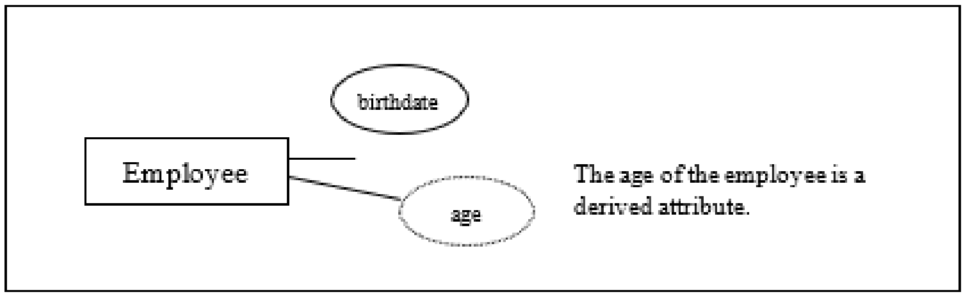


Figure 7.12: Age as a derived attribute

Sometimes an attribute of one entity type is derived from attributes from other entity types. Consider the attribute for the total of an Invoice. A value of InvTotal is derivable; it can be computed from invoice lines. Someone who implements a database and applications that access the database would need to decide whether the value of a derivable attribute should be computed when the entity is stored or updated versus computing the value (on-the-fly) when it is needed.

Exercises

1) Consider an educational environment where the institution tracks the performance of each student. Often this is called the students overall average, or overall grade point average. Is such an attribute a derived attribute? How is its value determined?

2) Consider a library application that needs to keep track of books that have been borrowed. Suppose there is an entity type Loan that has attributes bookID, memberID, dateBorrowed and dateDue. Suppose the due date is always 2 weeks after the borrowed date. Show Loan and its attributes in an ERD.

7.3.6: Key Attributes

Some attributes, or combinations of attributes, serve to identify individual entities. For instance, suppose an educational institution assigns each student a student number that is different from all other student numbers. We say the student number attribute is a *key* attribute; student numbers are *unique* and distinguish students.

In an ERD, keys are shown underlined:

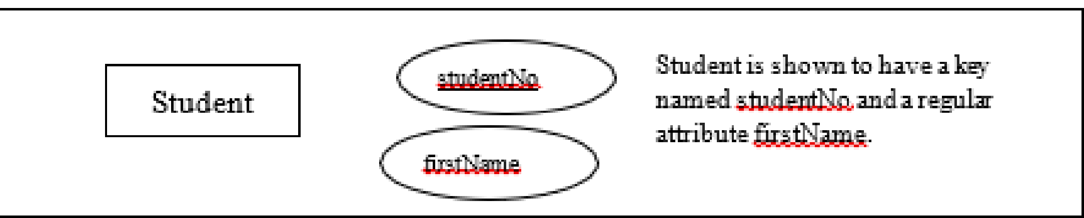


Figure 7.13: Key attribute is underlined

We define a *key* to be a minimal set of attributes that uniquely identify entities in an entity type. By minimal we mean that all of the attributes are required – none can be omitted.

For instance, a typical key for an invoice line entity type would be the combination of invoice number and invoice line number. Both attributes are required to identify a particular invoice line.

It is not unusual for an entity type to have several keys. For instance, suppose an educational institution has many departments such as Mathematics, Physics, and Computer Science. Each department is given a unique name and as well the institution assigns each one a unique code: MATH, PHYS, and CS. Both attributes would be underlined to show this in the ERD:

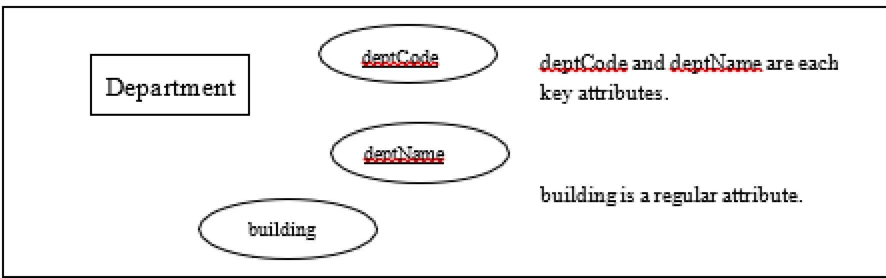


Figure 7.14: Multiple key attributes

Exercises

1) Suppose a company that sells products has a product entity type with the following attributes: prodNum, prodDesc, prodPrice. Suppose all three attributes are single- valued and that prodNum is a key attribute – each product has a different product number. Illustrate this information in an ERD.

2) Suppose a company assigns each employee a staff number and each employee has a government assigned identification number. For each employee the company records their first name, last name, and birth date. Illustrate this information in an ERD.

3) Consider a banking application where each account is identified first by an account number and then by its type (Savings, Chequing, and Loan). This scheme allows the customer to remember just one number instead of three, and then to pick a specific account by its type. Other attributes to be considered are the date the account was opened and the account's current balance. Draw an ERD for the entity type Account with the attributes account number, account type, date opened, current balance. What is the key for the entity type? Is there an attribute that is likely a derived attribute? Show these attributes appropriately in the ERD.

7.3.7: Partial Key

Sometimes we have attributes that distinguish entities of an entity type from other entities of the same type, but only relative to some other related entity. This situation arises naturally when we model things like invoices and invoice lines. If invoice lines are assigned line numbers (1, 2, 3, etc.), these line numbers distinguish lines on a single invoice from other lines of the same invoice. However, for any given line number value, there could be many invoice lines (from separate invoices) with that same line number.

A *partial key* (also called a *discriminator***)** is an attribute that distinguishes instances of a weak entity type relative to a strong entity. Invoice line number is a partial key for invoice lines; each line on one invoice will have different line numbers. Using the Peter Chen notation, the discriminator attribute is underlined with a dashed line:

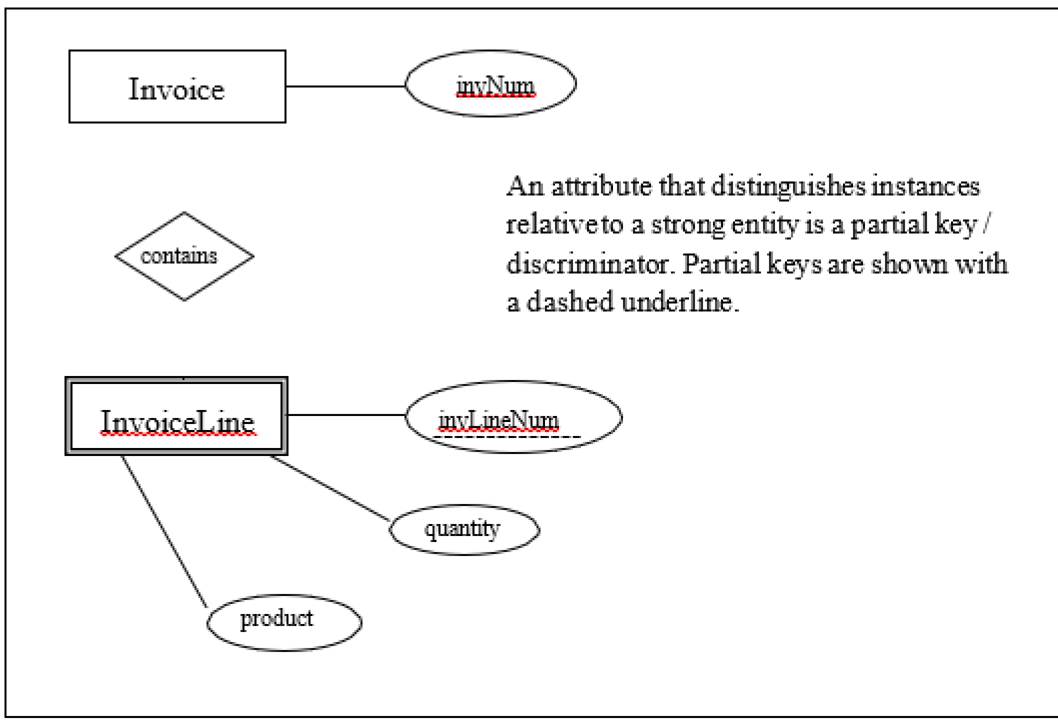


Figure 7.15: Line number distinguishes lines on the same invoice

Later when relationships are covered it will be clearer that attributes for relationships can be discriminators too. Consider that a library has books that members will borrow. Any book could be borrowed many times and even by the same member. However, when a member borrows the same book more than once the date/time will distinguish those events. Consider the following ERD for this case:

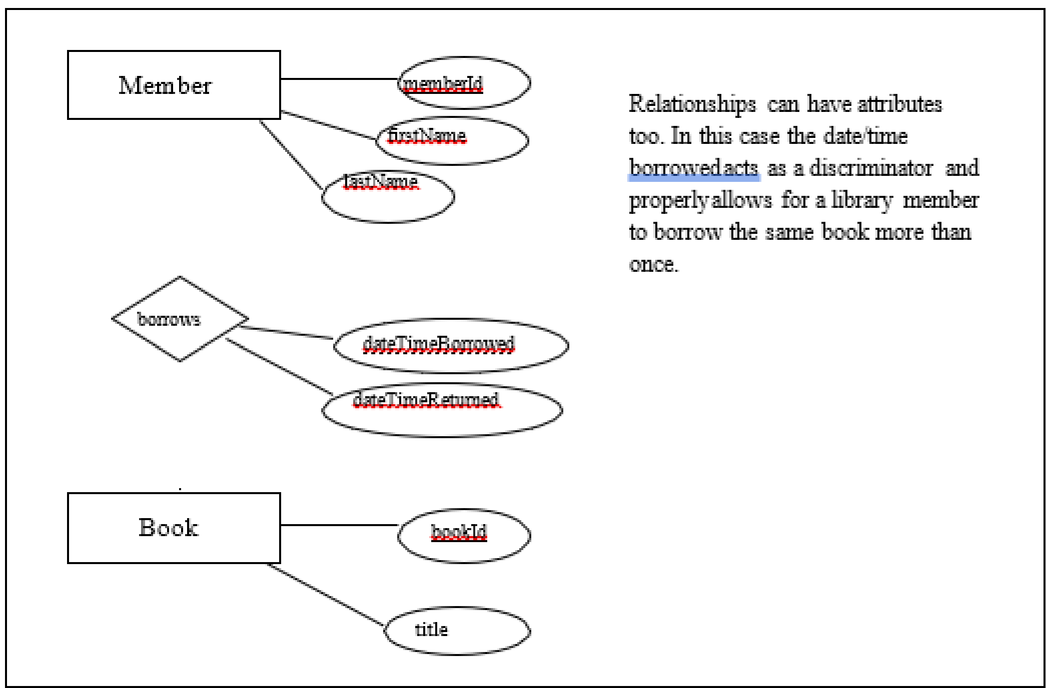


Figure 7.16: Relationship attribute as a discriminator

Exercises

1) Consider an educational institution that has departments and where each department offers courses. Suppose departments are assigned unique identifiers and so deptCode is a key for the department entity type. Courses are identified within a department by a course number; course numbers are unique within a department but not across departments. So, History may have a course numbered 215, and English could have a course numbered 215 too. In order to identify a particular course, we need to know the department and we need to know the course number. Illustrate an ERD including department and course entity types. Include attributes for Department (department code and department name), and for Course (course number, title, and description).

2) Consider a company that owns and operates parking lots. Develop an ERD with two entity types Parking Lot and Space and where:

- The address of a parking lot serves to identify the lot.
- Each space within a lot is rented at the same monthly rental charge.
- Each parking space is known by its number within the lot (within a lot these always start at 1).

- Each parking space is rented out to at most one vehicle. The vehicle's identifier must be recorded. The identifier comprises a province code and license plate number.

7.3.8: Surrogate Key

When a key specified for an entity is meaningless to the entity and to end-users (it doesn't describe any characteristic of an entity), the key is referred to as a *surrogate* key. A key that is not a surrogate key is often referred to as a *natural* key. Often a surrogate key is just a simple integer value assigned by the database system.

When database designs are implemented, surrogate keys can be useful to simplify references from one table to another (referential integrity) and the associated joins when tables are referenced in queries.

Exercises

1) Assuming you have experience with some database system, what data type would you use for surrogate keys?

7.3.9: Non-Key Attributes

Non-key attributes are attributes that are not part of any key. Generally, most attributes are simply descriptive, and fall into this category. Determining key and non-key attributes is an important modeling exercise, one that requires careful consideration. Consider an Employee entity type that has attributes for first name, last name, birth date; these attributes would serve to describe an employee but would not serve to uniquely identify employees.

People may join an organization and their name is not likely unique for the organization; we expect many people in a large organization to have the same first name, same last name, and even the same combination of first and last name. Names cannot usually be used as a key.

However, names chosen for entities such as departments in an organization could be keys because of the way the company would choose department names – they wouldn't give two different departments the same name.

Exercises

1) Consider a library and the fact that books are loaned out to library members. Dates could be used several times: for the date a book was borrowed, the date the book was returned, and the due date for a book. Consider an entity type Loan that has attributes book identifier, member identifier, date borrowed, date due, date returned. What combination of attributes would be a key? Which attributes are key attributes? Which attributes are non-key attributes

2) A birthdate attribute would appear for many entity types – for example students, employees, children. What is a birthdate likely to be: key or non-key?

7.3.10: Nulls

When a database design is implemented, one of the important things to know for each attribute of an entity type is whether or not that attribute must have a value. For example, when a book is borrowed from a library the date the book is borrowed is known, but the returned date is not known. Sometimes you will not know the value of an attribute until a certain event occurs.

Consider an educational environment and when a student registers for a course. The date the student registers would be known, but the grade is yet to be determined.

When an entity is created but some attribute does not have a value, we say it is *null*. Null represents the absence of a value; null is different from zero or from blank.

7.3.11: Domains

To complete the analysis for a database design it is necessary to determine what constitutes a valid value for an attribute. A *domain* for an attribute is its set of valid values which includes a choice of datatype, but a full specification of domain is typically more than that.

For instance, analysis for student identifiers may lead one to state that a student identifier is a positive whole number of exactly 7 digits with no leading zeros. The analysis of requirements for person names may lead one to state that the values stored in a database for a first name, last name, or middle name will not be more than 50 characters in length, and that names will not have any spaces at the beginning or end.

For each attribute one must determine its domain. More than one attribute can share the same domain. Knowing the underlying domains in your model is important. They help to complete your analysis, they are indispensable for coding programs, and they are useful for defining meaningful error messages.

Attribute domains are not usually shown in an ERD. Rather, domains are included in accompanying documentation which can be referred to when the database is being implemented.

7.4: Relationships

Up to this point we have made several references to the concept of relationship. Now, we will make our understanding of this concept more complete. A **relationship** is an association amongst entities. Relationships will have justification in business rules, in the way an enterprise manages its business.

There are several ways of classifying relationships, according to *degree*, *participation*, *cardinality*, whether *recursion* is involved, and whether a relationship is *identifying* or not.

7.4.1: Degree

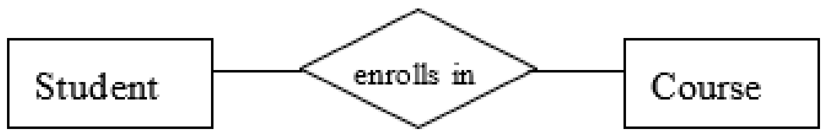


Figure 7.17: Binary relationship involves two entity types

We consider the *degree* as the number of entities that participate in the relationship. When we speak of a student enrolling in a course, we are considering a relationship (say, the *enroll in* relationship) where two entity types (Student and Course) are involved. This relationship is of degree 2 because each instance of the relationship will always involve one student entity and one course entity.

With binary relationships there must be two defining statements we can express, one from the perspective of each entity type. In this case our statements are:

- A student may enroll in any number of courses.
- A course may have any number of students enrolled.

Many database modeling tools only support binary relationships. However, there are situations where relationships of higher degree are useful. A relationship involving 3 entity types is called *ternary*; more generally we refer to relationships with *n* entity types as *n-ary*. Our primary focus in this text is on binary relationships.

7.4.2: Participation

Suppose we are designing a database for a company that has several departments and employees. Suppose further that each employee must be assigned to work in one department. We can define a *works in* relationship involving Department and Employee. Employees must participate in the relationship, and we show this using a double line joining the diamond symbol to the Employee entity type.

Figure 7.18: Employee must work in a department

The double line stands for *total* or *mandatory* participation which means that instances of the adjacent entity type must participate in the relationship – in the case above, all instances of Employee must be assigned to some department. Any time we show a single line we are stating participation is *optional*; for the above we are saying that a department will have zero or more employees who work there.

7.4.3: Cardinality

Cardinality is a constraint on a relationship specifying the number of entity instances that a specific entity may be related to via the relationship. Suppose we have the following rules for departments and employees:

- A department can have several employees that work in the department
- An employee is assigned to work in one department.

From these rules we know the cardinalities for the *works in* relationship and we express them with the cardinality symbols *1* and *n* below.

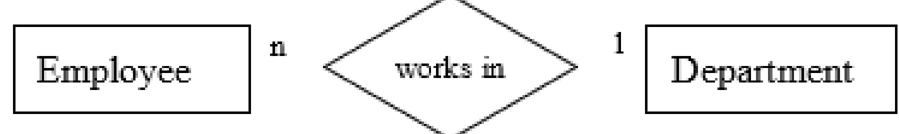


Figure 7.19: One-to-many relationships are most common

The *n* represents an *arbitrary number of instances*, and the *1* represents *at most one instance*. For the above works in relationship we have

- a specific employee works in at most only one department, and
- a specific department may have many (zero or more) employees who work there.

n, *m*, *N*, and *M* are common symbols used in ER diagrams for representing an arbitrary number of occurrences; however, any alphabetic character will suffice.

Based on cardinality there are three types of binary relationships: *one-to-one*, *one-to- many*, and *many-to-many*.

One-to-One

One-to-one relationships have *1* specified for both cardinalities. Suppose we have two entity types Driver and Vehicle. Assume that we are only concerned with the current driver of a vehicle, and that we are only concerned with the current vehicle that a driver is operating. Our two rules associate an instance of one entity type with at most one instance of the other entity type:

- a driver operates at most one vehicle, and
- a vehicle is operated by at most one driver.

And so, the relationship is one-to-one.

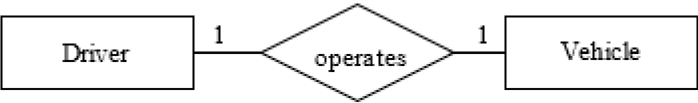


Figure 7.20: One-to-one relationship

One-to-Many

One-to-many relationships are the most common ones in database designs. Suppose we have customer entities and invoice entities and:

- an invoice is for exactly one customer, and
- a customer could have any number (zero or more) of invoices at any point in time.

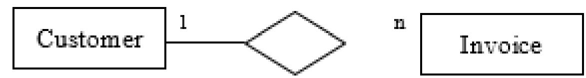


Figure 7.21: One-to-many relationship

Because one instance of an Invoice can only be associated with a single instance of Customer, and because one instance of Customer can be associated with any number of Invoice instances, this is a one-to-many relationship:

Many-to-Many

Suppose we are interested in courses and students and the fact that students register for courses. Our two rule statements are:

- any student may enroll in several courses,
- a course may be taken by several students.

This situation is represented as a many-to-many relationship between Course and Student:

Figure 7.22: Many-to-many relationship

As will be discussed again later, a many-to-many relationship is implemented in a relational database in a separate relation. In a relational database for the above, there would be three relations: one for Student, one for Course, and one for the many-to-many. (Sometimes this 3rd relation is called an intersection table, a composite table, a bridge table.)

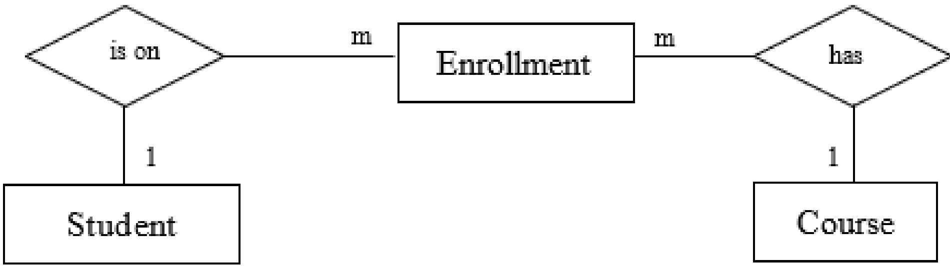


Figure 7.23: Many-to-many becomes two one-to-many relationships

Partly because of the need for a separate structure when the database is implemented, many modellers will 'resolve' a many-to-many relationship into two one-to-many relationships as they are modelling. We can restructure the above many-to-many as two one-to-many relationships where we have 'invented' a new entity type called Enrollment:

> A student can have many enrollments, and each course may have many enrollments.
> An enrollment entity is related to one student entity and to one course entity.

7.4.4: Recursive Relationships

A relationship is *recursive* if the same entity type appears more than once. A typical business example is a rule such as "an employee *supervises* other employees". The *supervises* relationship is recursive; each instance of *supervises* will specify two employees, one of which is considered a *supervisor* and the other the *supervised*. In the following diagram the relationship symbol joins to the Employee entity type twice by two separate lines. Note the relationship is one-to-many: an employee may supervise many employees, and an employee may be supervised by at most one other employee.

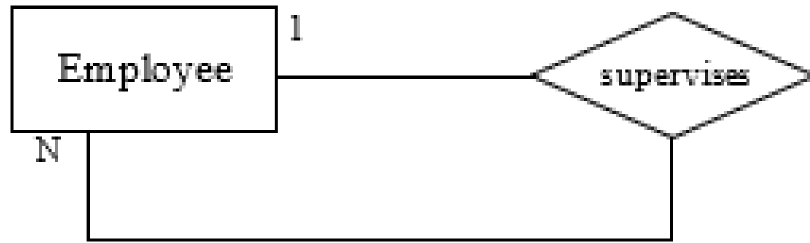


Figure 7.24: Recursive relationship involving Employee twice

With recursive relationships it is appropriate to name the roles each entity type plays. Suppose we have an instance of the relationship:

John *supervises* Terry

Then with respect to this instance, John is the *supervisor* employee and Terry is the *supervised* employee. We can show these two roles that entity types play in a relationship by placing labels on the relationship line:

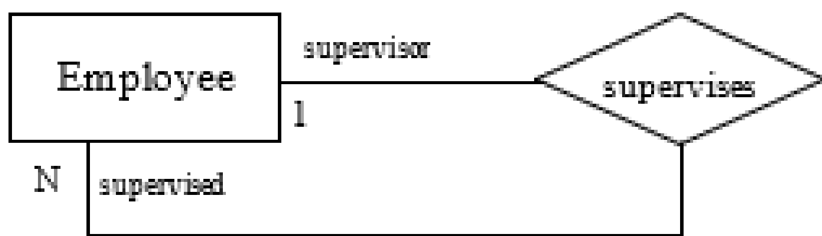


Figure 7.25: Recursive relationship with role names

This one-to-many *supervises* relationship can be visualized as a hierarchy. In the following we show five instances of the relationship: John supervises Lee, John supervises Peter, Peter supervises Don, Peter supervises Mary, and John supervises Noel.

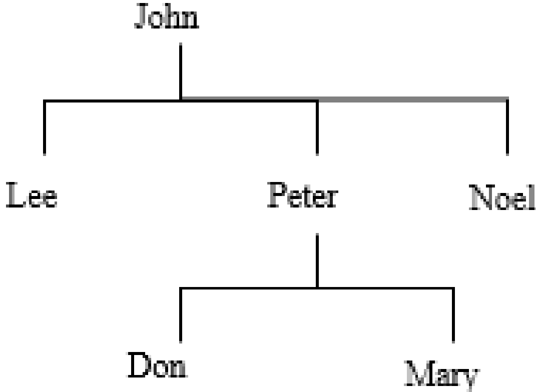


Figure 7.26: The supervising hierarchy

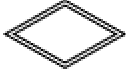
In the above example note the participation constraint at both ends of *supervises* is optional. This must be the case because some employee will not be supervised, and, for some employees there are no employees they supervise.

Generally recursive relationships are difficult to master. Some other situations where recursive relationships can be used:

- A person marries another person
- A person is the parent of a person
- A team plays against another team
- An organizational unit reports to another organizational unit
- A part is composed of other parts.

7.4.5: Identifying Relationships

When entity types were first introduced, we discussed an example where a department offers courses and that a course must exist in the context of a department. In that case the Course entity type is considered a weak entity type as it is existence-dependent on Department. It is typical in such situations that the key of the strong entity type is used in the identification scheme for the weak entity type. For example, courses could be identified as MATH-123 or PHYS-329, or as Mathematics-123 or Physics-329. In order to convey the composite identification scheme for a weak entity type we specify the relationship as an *identifying* relationship which is visualized using a double-lined diamond symbol:



Additionally, in situations where we have an identifying relationship we usually have:

- a weak entity type with a partial key
- a weak entity type that must participate in the relationship (total participation) and so the ERD for our hypothetical educational institution could be:

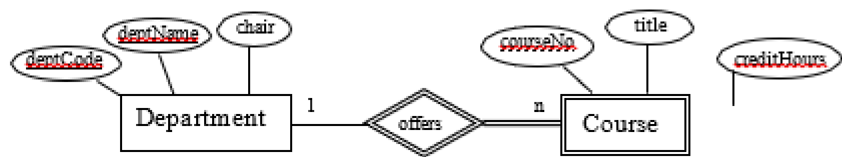


Figure 7.27: An identifying relationship

Note the keys for the strong entity type appear only at the strong entity type. The identifying relationship tells one that a department key will be needed to complete the identification of a course.

Exercises

1) Consider a company that owns and operates parking lots. Draw an ERD to include the following specifications. Each parking lot has a unique address (use the typical fields for addresses) and each parking lot has a certain number, say n, of parking spaces. Each space in a lot has a number between 1 and n. The cost of renting a parking space is the same for all spaces in a lot. The company rents individual spaces out to its customers. Each customer is identified by a driver's license id, has a first and last name. Each customer will identify possibly several cars that they will park in the space rented to them. For each car the company needs to know the year, make, model, colour and its license plate number.

2) Modify your model from the previous question to allow for scrambled parking. By this we mean that a customer is rented a space in a lot, but the customer may park in any available space.

3) Draw an ERD involving employees and their dependents where each employee has a unique id number and where dependents of the same employee are numbered starting at 1. It may be rare, but we will allow for dependents of the same employee to have the same name and birthdates. Include typical attributes for an employee, and for a dependent include the birthdate, first and last names.

4)Draw an ERD for marriages between two people. For persons include birthdate, first name, last name, and a unique person id. Consider marriage to be a relationship between two people and suppose we want our model to allow for people to have more than one marriage. Use the date of the marriage as a discriminator.

5) Consider marriages again but now let marriage be an entity type. Suppose when people marry there is a marriage certificate that is granted by a government authority. Include attributes applicable to a marriage.

6) Suppose we are modeling marriage as a relationship between two people. When, or under what circumstances, can we model this as a one-to-one relationship?

7) Draw an ERD that allows for marriages between possibly more than two people.

8) Consider the one-to-one *operates* relationship in this chapter. Modify the example so that drivers have attributes: driver license, name (which comprises first name and last name), and vehicles have attributes: license plate number, VIN, year, colour, make and model. Note that VIN stands for vehicle

identification number and this is unique for each vehicle. Assume that each driver must be assigned to a vehicle.

9) Consider the *enroll in* relationship used in this chapter. Suppose we must allow for a student to repeat a course to improve their grade. Develop an ERD and include typical attributes for student, course, etc. We need to keep a complete history of all course attempts by students.

10) What problems arise if one makes the *supervises* relationship mandatory for either the supervising employee or the employee who is supervised?

11) Consider requirements for teams, players and games, and develop a suitable ERD. Each team would have a unique name, have a non-player who is the coach, and have several players. Each player has a first and last name and is identified by a number (1, 2, 3, etc.). One player is designated the captain of the team. Assume a game occurs on some date and time and is played by two teams where one team is called the home team and the other team is called the visiting team. At the end of the game the score must be recorded.

12) Modify your ERD for the above to accommodate a specific sport such as curling, baseball, etc.

13) Consider an ERD for modelling customers, phones, and phone calls. Each customer owns one phone and so the phone number identifies the customer. Include other attributes such as credit card number, first name, and last name for a customer. We must record information for each phone call that is made: for each call there is a start time, end time, and of course the phone number/customers involved.

14) Create an ERD suitable for a database that will keep genealogy data. Suppose there is one entity type Person and you must model the two relationships: *marries* and *child of*.

15) Develop an ERD to support home real estate sales. Consider there are several sales employees who list and sell properties. For each employee we need to know their name (first and last), the date they started working for this company, and the number of years they have been with the company. Each property has owners (one or more people) and may have certain features such as number of baths, number of levels, number of bedrooms. For each owner we must keep track of their names (first and last). Each property has an address; each address has the usual attributes: street (comprising apartment number, street number, street name), city, province, and postal code. A home is listed at a certain price and sold at possibly a different price. Of course, we need to track the names of the buyers, the date of a listing and the date of a sale.

16) Develop an ERD to keep track of information for an educational institution. Assume each course is taught by one instructor, and an instructor could teach several courses. For each instructor suppose we

have a unique identifier, a first name, a last name, and a gender. Each course belongs to exactly one department. Within a department courses are identified by a course number. Departments are identified by a department code.

17) Develop an ERD to allow us to keep information on a survey. Suppose a survey will have several questions that can be answered true or false. Over a period of time the survey is conducted and there will be several responses.

18) Modify the ERD above to allow for surveys that have multiple choice answers.

19) Develop an ERD to support the management of credit cards. Each credit card has a unique number and has a customer associated with it. A customer may have several credit cards. The customer has a first name, last name, and an address. Each time a customer uses a credit card we must record the time, the date, the vendor, and the amount of money involved.

20) Modify the ERD for the above to accommodate the monthly billing of customers. Each month a customer receives a statement detailing the activity that month.

21) Develop an ERD to be used by a company to manage the orders it receives from its customers. Each customer is identified uniquely by a customer id; include the first name, last name, and address for each customer. The company has several products that it stocks and for which customers place orders. Each product has a unique id, unique name, unit price, and a quantity on hand. At any time, a customer may place an order which will involve possibly many products. For each product ordered the database must know the quantity ordered and the unit price at that point in time. If the customer does this through a phone call, then an employee is involved in the call and will be responsible for the order from the company side. Some orders are placed via the internet. For each order an order number is generated. For each order the database must keep track of the order number, the date the order was placed and the date by which the customer needs to receive the goods.

8. MAPPING AN ERD TO A RELATIONAL DATABASE

We use an Entity Relationship Diagram to represent the informational needs of a system. When we are convinced it is satisfactory, we map the ERD to a relational database and implement as a physical database.

In general, relations are used to hold entity sets and to hold relationship sets. The considerations to be made are listed below. After we present the mapping rules, we illustrate their application in a few examples.

8.1: Mapping Rules

To complete the mapping from an ERD to relations we must consider the entity types, relationship types, and attributes that are specified for the model.

8.1.1: Entity Types

Each entity type is implemented with a separate relation. Entity types are either strong entity types or weak entity types.

- Strong Entities
  - Strong, or regular, entity types are mapped to their own relation. The PK is chosen from the set of keys available.

- Weak Entities
  - Weak entity types are mapped to their own relation, but the primary key of the relation is formed as follows. If there are any identifying relationships, then the PK of the weak entity is the combination of the PKs of entities related through identifying relationships and the discriminator of the weak entity type; otherwise, the PK of the relation is the PK of the weak entity.

8.1.2: Relationship Types

The implementation of relationships involves foreign keys. Recall, as discussed under Weak Entities (previous page), that if the relationship is identifying, then the primary key of an entity type must be propagated to the relation for a weak entity type. We must consider both the degree and the cardinality of the relationship. The first three bullet-points deal with binary relationships and the last bullet-point concerns *n*-ary relationships.

- Binary One-To-One
  - In general, with a one-to-one relationship, a designer has a choice regarding where to implement the relationship. One may choose to place a foreign key in one of the two relations, or in both. Consider placing the foreign key such that nulls are minimized. If there are attributes on the relationship those can be placed in either relation.
- Binary One-To-Many
  - With a one-to-many relationship the designer must place a foreign key in the relation corresponding to the 'many' side of the relationship. Any other attributes defined for the relationship are also included on the 'many' side.
- Binary Many-To-Many
  - A many-to-many relationship must be implemented with a separate relation for the relationship. This new relation will have a composite primary key comprising the primary keys of the participating entity types and any discriminator attribute, plus other attributes of the relationship if any.
- *n*-ary, n>2
  - A new relation is generated for an *n*-ary relationship. This new relation has a composite primary key comprising the *n* primary keys of the participating entity types and any discriminator attribute, plus any other attributes. There is one exception to the formation of the PK: if the cardinality related for any entity type is 1, then the primary key of that entity type is only included as a foreign key and not as part of the primary key of the new relation.

8.1.3: Attributes

All attributes, with the exception of derived and composite attributes, must appear in relations. In the following we consider attributes according to whether they are simple, atomic, multi-valued, derived, or composite.

- Simple, atomic
    - These are included in the relation created for the pertinent entity type, many-to- many relationship, or *n*-ary relationship.

- Multi-valued
    - Each multi-valued attribute is implemented using a new relation. This relation will include the primary key of the entity type. The primary key of the new relation will be the primary key of the entity type plus the multi-valued attribute. Note that in this new relation, the attribute is no longer multi-valued.

- Derived
    - Derived attributes are not included. However, a database designer could choose to include derived attributes if their presence would improve performance.

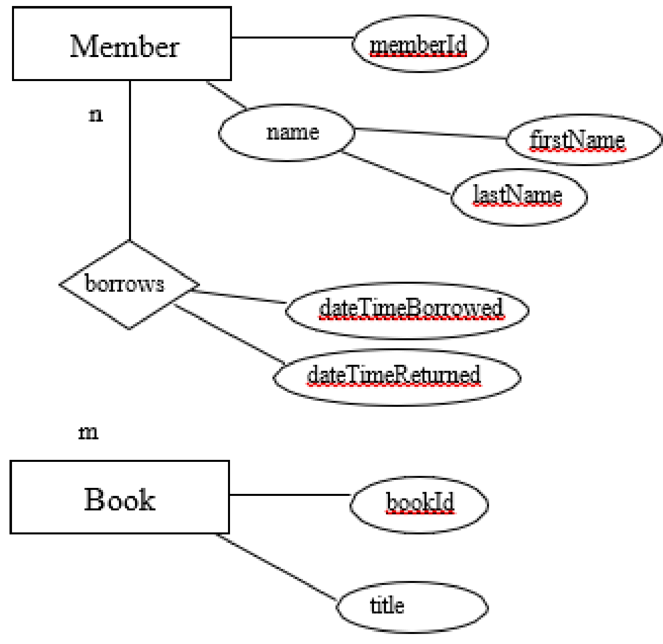
- Composite attributes: only attributes at the lowest levels of a composite hierarchy will appear in relations, according to whether they are simple/atomic, derived, or multivalued. Attributes above the lowest levels are not included in a relation.

The above constitutes the standard rules for mapping an ERD to relations. A designer may make other choices, but one expects there would be good reasons for doing so.

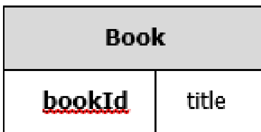
8.2: Examples

Example 1

Consider the ERD:



The mapping rules lead to the relations:



Book	
bookId	title

Member		
memberId	firstName	lastName

Borrow			
memberId	**bookId**	**dateTimeBorrowed**	dateTimeReturned

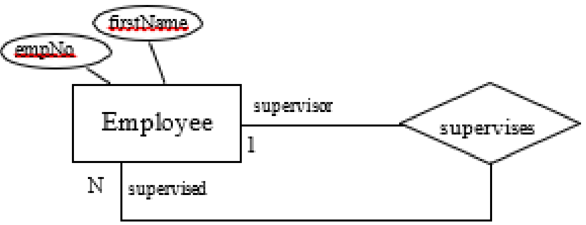
Notes:

- The Member relation does not have a composite attribute *name*.
- Since Borrows is a many-to-many relationship the Borrow relation is defined with a composite primary key {*memberId*, *bookId*, *dateTimeBorrowed*}.
- *memberId* in the Borrow relation is a foreign key referencing Member.

- *bookId* in the Borrow relation is a foreign key referencing Book.

Example 2

Consider the ERD:



The mapping rules lead to the relation:

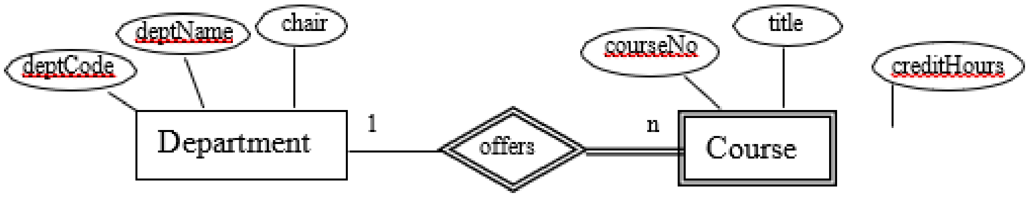
Employee		
empNo	firstName	supervisor

Notes:

- The attribute *supervisor* is a foreign key referencing Employee.
- A foreign key is placed on the 'many' side of a relationship and so in this case the foreign key references the employee who is the supervisor (the role name on the 'one' side); hence the name *supervisor* was chosen as the attribute name.

Example 3

Consider the ERD:



The mapping rules lead to the relations.

Department		
deptCode	deptName	chair

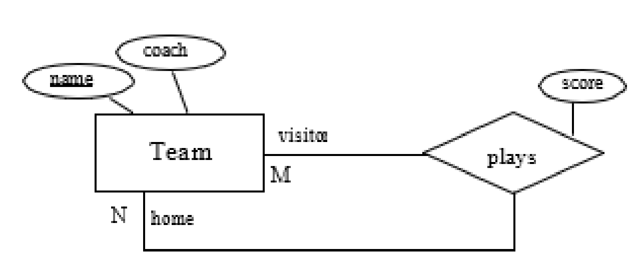
Course			
deptCode	**courseNo**	title	creditHours

Notes:

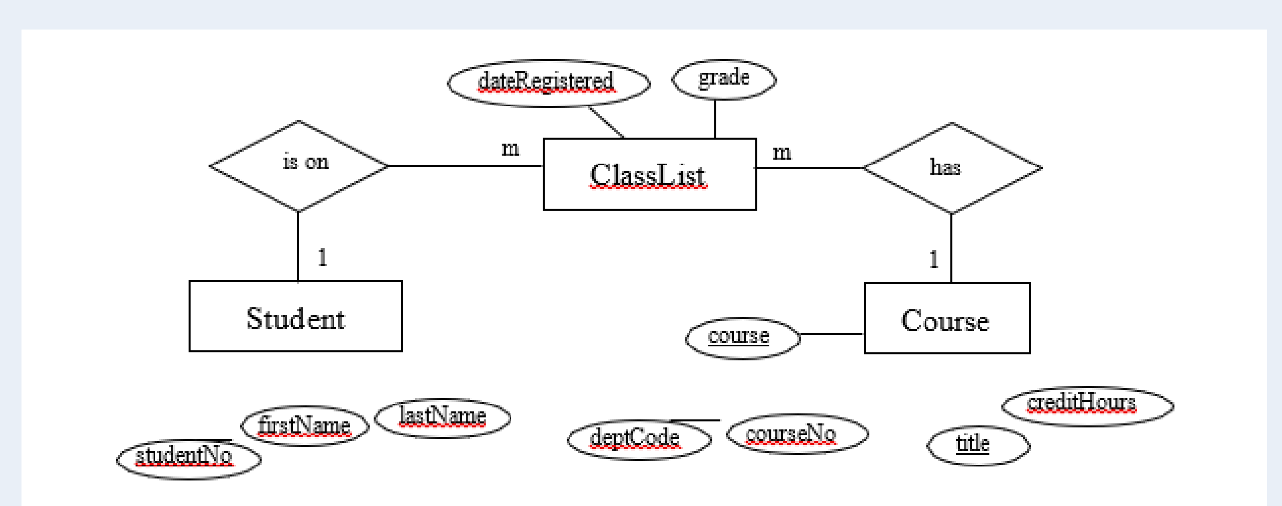
- *deptCode* was chosen as the primary key of Department.
- *deptName* is a key and so a unique index can be defined to ensure uniqueness.
- Since Course is a weak entity type and is involved in an identifying relationship, the primary key of Course is composite comprising {*deptCode*, *courseNo*}.
- *deptCode* in Course is a foreign key referencing Department.

Exercises

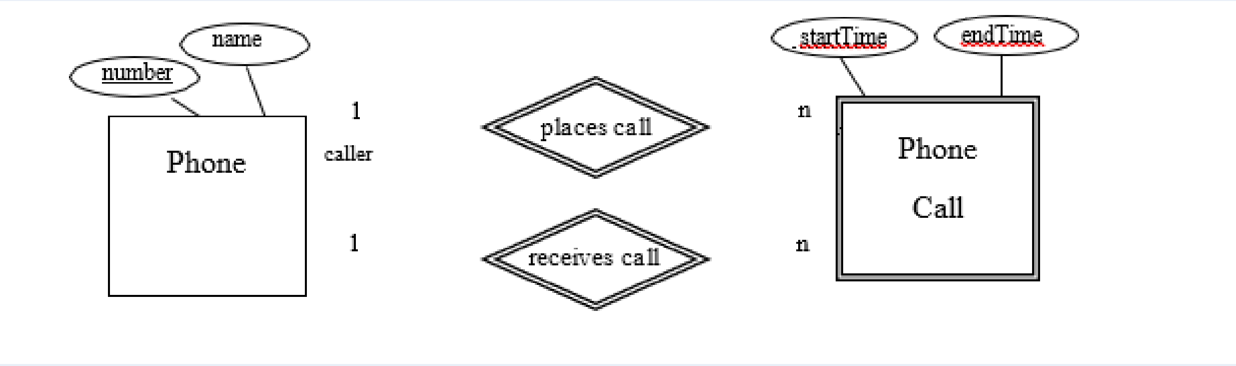
1) Map the ERD to relations.



2) Map the ERD to relations.



3) Map the ERD to relations.



9. DATA DEFINITION LANGUAGE (DDL)

Many of the tools available for constructing ERDs are capable of generating data definition language commands that are used for creating tables, indexes, and relationships. You can find many references easily to DDL. For instance, if you are interested try http://en.wikipedia.org/wiki/Data_Definition_Language, or enter the phrase *Data Definition Language* in your favourite search engine.

9.1: Running DDL in MS Access

Most database systems provide a way for you to run data definition language commands. When such facility exists, it can be relatively easy to create and re-create databases from a file of DDL commands. One way to run DDL commands in MS Access is through a query that is in SQL View. To run a DDL command we follow these two steps:

- Open a database and choose to create a query, but do not add any tables to the query
- Then, choose SQL View and you will be able to type a DDL command or paste one in:

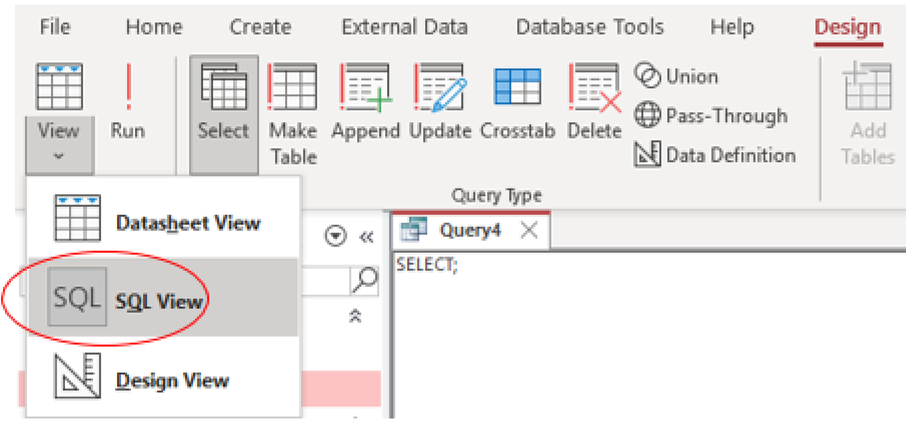


Figure 9.1: Choose SQL view for the query

9.2: Example

In this chapter we will create the tables for a Library database using DDL. Suppose we require the three tables: Book, Patron, Borrow:

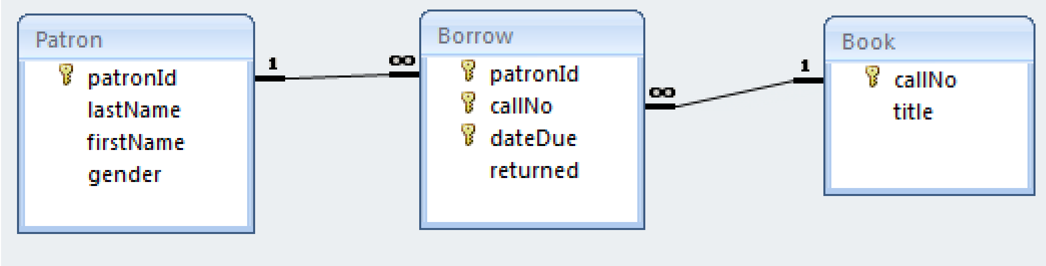


Figure 9.2: Sample database to create

The above diagram (produced from the Relationships Tool) represents the database we wish to create but where we will do so using DDL commands.

9.2.1: DDL Commands

We will illustrate three DDL commands (create table, alter table, create index) as we create the library database.

CREATE TABLE	Create the Book table
	Create the Patron table with a primarykey
	Create the Borrow table with a primarykey and a foreign key referencing Patron
ALTER TABLE	Alter the Book table so it has a primarykey
	Alter the Borrow table with a foreignkey referencing Book
	Add an attribute named gender toPatron
CREATE INDEX	Create an index
DROP TABLE and DROP INDEX	Remove a table or index from the database

Figure 9.3: Data Definition Commands

In some database environments we can run more than one command at a time; the commands would be in a file and would be submitted as a batch to be executed. In the following we will demonstrate and run one command at a time.

9.2.2: Creating the Database

Example 1

Consider the following create table command which is used to create a table named Book. The table has two fields: callNo and title.

- CREATE TABLE Book
- (
- callNo Text(50), titleText(100)
-);

The command begins with the keywords CREATE TABLE. It's usual for keywords in DDL to be written in upper case, but it's not required to do so. The command is just text that is parsed and executed by a command processor. If humans are expected to read the DDL then the command is typically written on several lines as shown, one part per line.

Example 2

Now consider the following Create Table command which creates a table and establishes an attribute as the primary key:

- CREATE TABLE Patron
- (
- patronIdCounterPRIMARY KEY, lastNameText(50),
- firstNameText(50)
-);

The primary key of Patron is the patronId field. Notice the data type is shown as Counter. After running this command, you will be able to see that the Counter data type is transformed to AutoNumber.

Example 3

Our last example of the create table command is one that creates a table, sets its primary key, and creates a foreign key reference to another table:

- CREATE TABLE Borrow
- (
- patronIdInteger,
- callNoText(50),
- dateDueDATETIME,
- returnedYESNO,
- PRIMARY KEY (patronId, callNo, dateDue),
- FOREIGN KEY (patronId) REFERENCES Patron
-);

There are several things to notice in the above command:

- The primary key is composite and so it is defined in a separate PRIMARY KEY clause.

- The data type of patron id must match the data type used in the Patron table and so the data type is defined as Integer.
- The dateDue field will hold a due date and so its data type is defined as DATETIME.
- The returned field will hold a value to indicate whether a book has been returned or not, and so its data type is defined as YESNO.
- A row in the Borrow table must refer to an existing row in Patron and so we establish a relationship between Borrow and Patron using the FOREIGN KEY clause. After running this create table command you can see the relationship in Access by opening the Relationships Tool.

Example 4

The Book table was created previously but there is no specification for a primary key. To add a primary key, we use the alter table command as shown below.

- ALTER TABLE Book
- ADD PRIMARY KEY (callNo);

Example 5

Now that Book has a primary key, we can define the relationship that should exist between Borrow and Book. To do so we use the alter table command again:

- ALTER TABLE Borrow
- ADD FOREIGN KEY (callNo)
- REFERENCES Book (callNo) ;

Example 6

Notice that the Patron table does not have a gender attribute. To add this, we can use the alter table command:

- ALTER TABLE Patron ADD
- COLUMN gender Text(6) ;

Example 7

For performance reasons we can add indexes to a table. DDL provides create index and drop index commands for managing these structures. To create an index for Patron on the combination last name and first name, we can execute:

- CREATE INDEX PatronNameIndex ON Patron (LastName, FirstName);

Example 8

To remove the above index we need to identify the index by name:

- DROP INDEX PatronNameIndex;

Example 9

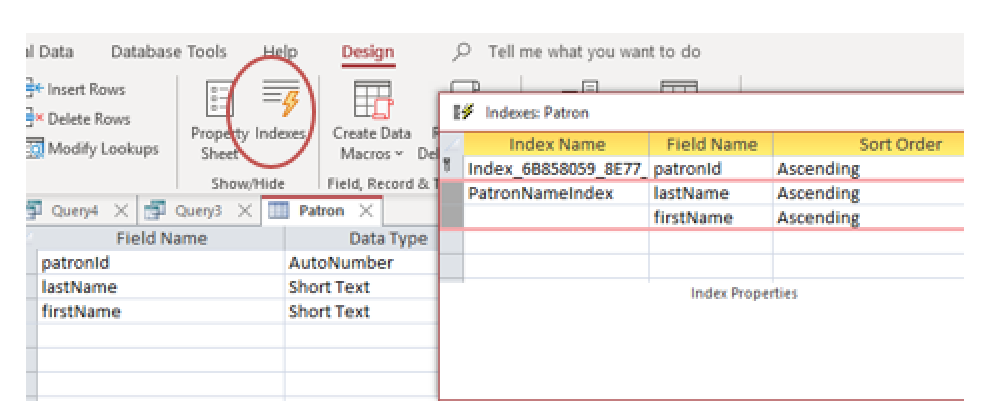
To remove a table we use the drop table command.

- DROP TABLE Person;

Exercises

1) The effect of executing the commands in the first 6 examples can be accomplished by 3 create table commands. Example 9 shows a drop table command; use similar drop commands to delete all the tables you created in exercises 1 and 2. Now, write 3 create table commands that have the same effect as examples 1 through 6. After running the DDL statements open the relationships tool to verify your commands created the 3 tables and the 2 relationships.

2) Example 7 creates an index. Run this command in your database and then verify the index has been created. You can view index information: put the table in Design View and then click the Indexes icon:



Notice that the (primary) index has a name that was generated by MS Access.

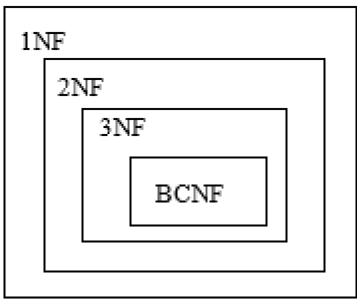
3) Consider an ERD from the previous chapter. Write the DDL that could create the required relations.

10. NORMALIZATION

The theory of normal forms is concerned with the structure of relations in a relational database. There are several normal forms of which 1NF, 2NF, 3NF and BCNF are the most important for practical OLTP database design. OLTP stands for online transaction processing – OLTP systems are used to run the day-to-day events of a business.

Normalization theory gives us a theoretical basis to judge the quality of a database and helps one understand the impact of some design decisions. In practice Entity Relationship Modelling is the primary technique used for designing databases, and experienced practitioners will typically develop BCNF relations as a result. Normalization can be applied by the practitioner to understand better the semantics behind some relations and possibly make some design modifications.

1NF, 2NF, 3NF and BCNF are acronyms for first, second, third, and Boyce-Codd normal forms. There is a sequence to normal forms: 1NF is considered the weakest, 2NF is stronger than 1NF, 3NF is stronger than 2NF, and BCNF is considered the strongest of these four normal forms. Also, any relation that is in BCNF, is in 3NF; any relation in 3NF is in 2NF; and any relation in 2NF is in 1NF. This correspondence can be shown as:



BCNF relations are in 3NF
3NF relations are in 2NF
2NF relations are in 1NF

Transactions are *units of work* designed to meet the goals of users. For instance, in a banking environment we would expect to find a deposit transaction, a withdrawal transaction, a transfer transaction, and a balance lookup transaction. A unit of work is a collection of database operations that are executed in their entirety or not at all. For example, if you are transferring money from one account to another it's important for the integrity of accounts that the transfer be completely done, and never partly done. If a transfer transaction is partly done (say, because of a system failure) then accounts would be out of balance. A database environment has capabilities to back out partly executed transactions so the system can be back where it was prior to a failed transfer transaction.

A banking system could have thousands of users and we expect transactions such as these to be correctly and efficiently executed. A normalized database is such that every relation is in at least 1NF, and preferably BCNF. Normalized databases lead to the most efficient designs for these types of transactions.

Normalization is a process that replaces a relation with other relations of a higher normal form. The process involves decomposing a relation into other relations in such a way as to preserve the original information and reduce redundancy of data. Reducing redundant data increases the number of relations but makes the data easier to maintain. Later we give examples of decomposition. We say normalization is a process that

improves a database design. The objective of normalization is sometimes stated: *to create relations where every dependency is on the key, the whole key, and nothing but the key*[1]. A relation that is fully normalized is about a single concept such as a student entity type, a course entity type, and so on.

De-normalization is a process that changes relations from higher to lower normal forms, and hence generates redundant data in the tuples (rows/records) of a relation (table). If deemed necessary, this would be done to improve the performance (reduce the cost) of retrieving information from the database. The cost of querying de-normalized relations is generally less because fewer joins are required.

We consider higher normal forms to be better because the update semantics for data are simplified. By this we mean that applications required to maintain the database are simpler to code and so they are easier to maintain. In the following we discuss:

- Functional Dependencies
- Update Anomalies
- Partial Dependencies
- Transitive Dependencies
- Normal Forms

[1] Kent, William. "A Simple Guide to Five Normal Forms in Relational Database Theory", Communications of the ACM 26 (2), Feb. 1983, pp. 120–125.

10.1: Functional Dependencies

To understand normalization theory (first, second, third and Boyce-Codd normal forms), we must understand what is meant by the term functional dependency. There is another type of dependency called a multi-valued dependency but that is important to the understanding of higher normal forms not covered in this text.

A functional dependency is an association between two attributes. We say there is a **functional dependency** from attribute A to an attribute B if and only if for each value of A there can be at most one value for B. We can illustrate this by writing:

- A functional determines B, or
- B is functionally determined by A, or
- by a drawing such as: A → B

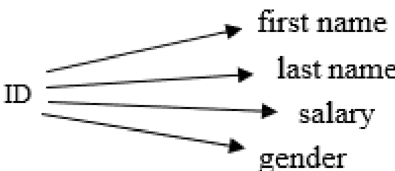
When we have a functional dependency from A to B, we refer to attribute A as the **determinant**.

EXAMPLE 1.

Consider a company collects information about each employee such as the employee's identification number (ID), their first name, last name, salary and gender. As is typical, each employee is given a unique ID which serves to identify the employee. Hence for each value of ID there is at most one value for first name, last name, salary and gender.

Therefore, we have four functional dependencies where ID is the determinant; we can show this as a list or graphically:

ID → first name
ID → last name
ID → salary
ID → gender



If you think about this case, there cannot be any other FDs. For example, consider the gender attribute – we need to allow for more than one employee for a given gender, and so we cannot have a situation where gender functionally determines ID. So, gender ⎝ ID cannot exist. Now consider the first name attribute. Again, we need to allow for more than one employee to have the same first name and so first name cannot determine anything. Similarly, for other attributes.

EXAMPLE 2.

Recall the Department and Course tables introduced in Chapter 2 – sample data is shown below:

deptCode	deptName	deptLocn	deptPhone	chairName
MATH	Mathematics	2R33	786-0033	Peter Smith
HIST	History	3D07	786-0300	Simon Lee
IS	Indigenous Studies	3C11	786-3322	Leslie Roman
MENN	Mennonite Studies	3C11	786-3322	Leslie Roman
BIOL	Biology	2L88	786-9843	James Dunn

deptCode	courseNo	title	description	creditHours
HIST	1010	Introduction to History	Within a relatively small lecture/seminar setting, this course introduces you to the ways in which people try to understand their present by studying their past.	6
IS	1010	Indigenous Ways of Knowing	This course offers an introduction to Indigenous ways of knowing through active participation in strategies that facilitate the production of Aboriginal knowledge and through comparisons with Euro-American ways of knowing.	3
MENN	1010	Mennonites and the Modern World	This course is a history of the ethnic identity and religious faith of the Mennonites from the sixteenth century to the present.	6
IS	1201	Introductory Ojibwe	This course is intended for students who are not fluent in Ojibwe and have never taken a course in the language	6
BIOL	2401	Forest Field Skills Camp	This intensive two-week field course is mandatory for students in the Forest Ecology program and is designed to give students field survival and basic forestry skills.	1

Recall the primary keys (underlined above) of these two tables:

Table	PK
Department	deptCode
Course	deptCode, courseNo

Consider the Department table where deptCode is the primary key. For each value of deptCode there is at most one value for deptName, deptLocn, deptPhone, and chairName. You should agree the following FDs exist:

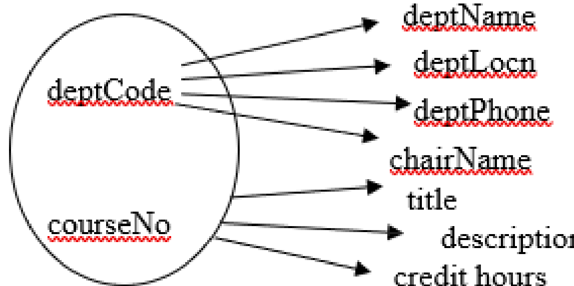
- deptCode –> deptName
- deptCode –> deptLocn
- deptCode –> deptPhone
- deptCode –> chairName

Each row of the Course table has one value for title, one value for description, and one value for credit hours. The primary key of Course consists of two attributes, deptCode and courseNo. The following FDs exist for the Course table:

- deptCode, courseNo –> title
- deptCode, courseNo –> description
- deptCode, courseNo –> credit hours

In this case we have a determinant comprising two attributes; the determinant is composite.

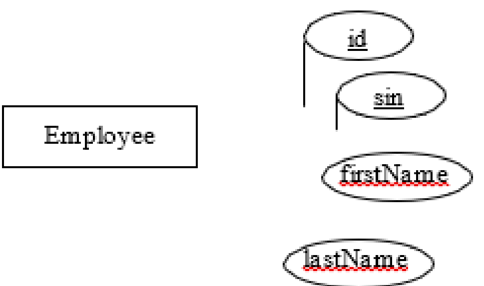
We can draw the functional dependencies as:



Could there be other functional dependencies in this situation?
These examples demonstrate that there is a FD from the primary key to each of the other attributes in a table.

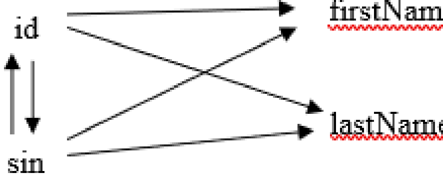
EXAMPLE 3.

The following ERD is shown in the Chen notation. There is one entity type named Employee that has 4 attributes. In this design there are two keys (*id* and *sin*) and two descriptive attributes (*firstName* and *lastName*):



Each symbol in an ERD contains information about a model. From the above we know there are two keys, *id* and *sin*. An *id* value, or a *sin* value, will uniquely identify an employee and so we have the six FDs:

id ➔ firstName
id ➔ lastName
sin ➔ firstName
sin ➔ firstName
id ➔ sin
sin ➔ id



This example shows that an ERD carries information that can be expressed in terms of FDs.

Exercises

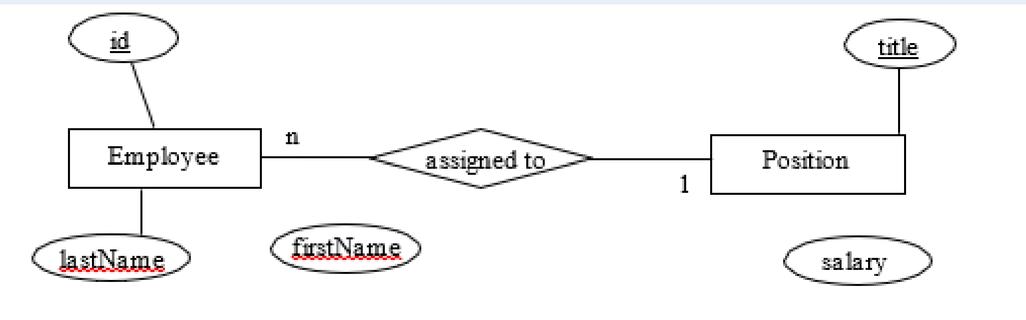
1) Consider the Product table below where productID is the PK. What FDs must exist in this table:

productID	description	unit price	quantity on hand
33	16 oz. can tomato soup	1.00	50
41	454 grams box corn flakes	4.50	39
45	Package red licorice	1.00	39
46	Package black licorice	1.00	50
47	1 litre 1% milk	1.99	25

2) Consider the ERD where the entity type *Employee* has one key attribute, *id*, and the entity type

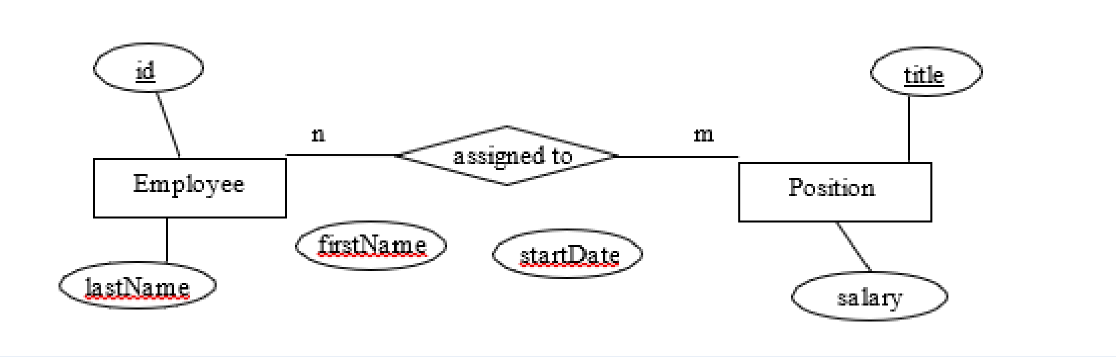
Position has one key attribute, *title*. As well the ERD shows a one-to-many relationship *assigned to* which can be expressed as:

- An employee is assigned to at most one position. A position can be assigned to many employees.

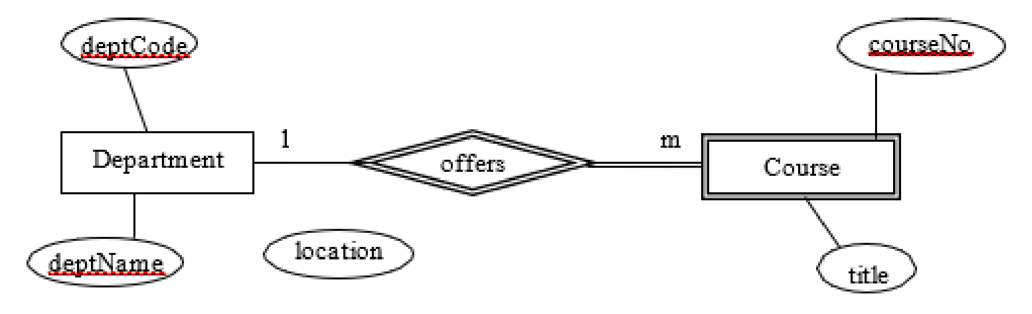


- List the FDs that must be present.

3) Consider the ERD below which is similar to the above, but where the *assigned to* relationship is many-to-many, and where *assigned to* has an attribute *startDate*. List the FDs that are present.



4) Consider the ERD below where *Department* has two keys *deptCode* and *deptName* – each department has a unique department code and has a unique department name. *Course* is a weak entity type with a partial key *courseNo*, and where *offers* is an identifying relationship.



- List the FDs that must exist.

5) Consider the table T with attributes A, B and C.

A	B	C
1	33	100
2	33	200
3	22	200
1	33	101
2	33	350
4	67	350
5	67	101

6) Suppose there are many more rows that are not shown:

- Is there a functional dependency from B to A? Explain your answer.
- The rows that are shown suggest there could be a functional dependency A **–>** B. Compose a database query that would indicate counter examples, if they exist, for the functional dependency A **–>** B. Such a query would list values of A in the table where two or more rows have the same value for A but different values for B.

10.1.1: Keys and Non-Keys

Before going further, we need to be clear regarding the concept of key. We define the ***key*** of a relation to be any minimal set of attributes that uniquely identify tuples in the relation. We say minimal to eliminate trivial cases. Consider: if attribute *k* is a key and uniquely identifies a tuple then any combination of attributes that include *k* must also uniquely identify tuples. So, we restrict keys to be minimal sets of attributes that retain

the property of unique identification. Further, we define ***candidate keys*** to be the collection of keys for a relation; a database designer must choose one of the candidate keys to be the primary key.

Additionally, we define ***key attributes*** to be those attributes that are part of a key, and ***non-key attributes*** are those attributes that are not part of any key.

10.1.2: Anomalies

An anomaly is a variation that differs in some way from what is considered normal. With regards to maintaining a database, we consider the actions that must occur when data is updated, inserted, or deleted. In database applications where these update, insert, and/or delete operations are common (e.g. OLTP databases), it is desirable for these operations to be as simple and efficient as possible.

When relations are not fully normalized, they exhibit update anomalies because basic operations are not as simple as possible. When relations are not fully normalized, some aspect of the relation will be awkward to maintain.

Consider the relation structure and sample data:

deptNum	courseNum	studNum	grade	studName
92	101	3344	A	Joe
92	115	7654	A	Brenda
81	101	7654	C	Brenda
92	226	3344	B	Joe

This relation is used for keeping track of student enrollments, the grade assigned, and (oddly) the student's name.

What must happen if a student's name were to change? We should want our databases to have correct information, and so the name may need to be changed in several records, not just one. This is an example of an update anomaly – the simple change of a student's name affects, not just one record, but potentially several in the database. The update operation is more complex than necessary, and this means it is more expensive to do, resulting in slower performance. When operations become more complex than necessary, there is also a chance the operation is programmed incorrectly resulting in corrupted data — another unfortunate consequence.

Consider the Course and Department tables again, but now consider that they are combined into a single table. Obviously, this is a table with a considerable redundancy – for each course in the same department, the department location, phone, and chair must be repeated.

Department Course								
dept Code	dept Name	dept Location	dept Phone	chair Name	course No	title	description	credit Hours

The primary key of such a table must be {deptCode, courseNo}. Consider for the following, however unlikely the situation seems, that the Deparment_Course table is the only table where department information is kept. Note that our point here is only to show, for a simple example, how redundancy leads to difficult semantics for database operations.

Insert anomaly

Suppose the university added a new department but there are no courses for that department yet. How can a row be added to the above table? Since no part of a primary key can be null, we cannot insert a row for a new department because we do not have a value for courseNo. This is an example of an insertion anomaly.

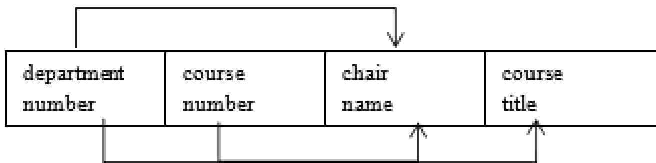
Delete anomaly

Suppose some department is undergoing a major reorganization. All courses are to be removed and later some new courses will be added. If we delete all courses, then we lose all the information in the database for that department.

The previous discussion concerning anomalies highlights some of the data management issues that arise when a relation is not fully normalized. Another way of describing the general problem here, as far as updating a database is concerned, is that redundant data makes it more complicated for us to keep the data consistent.

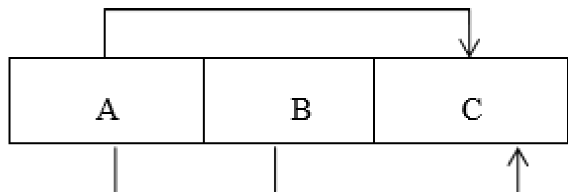
10.1.3: Partial Functional Dependencies

Consider a relation with department number, department chair name, course number and course title attributes. The combination {department number, course number} must be a key. The directed lines depict the FDs that are present:



Note the functional dependency of chair name on department number. If two or more rows in the relation have the same value for department number, they must have the same value for chair name. We say this redundancy is due to the FD of chair name on department number. Because chair name is a non-key attribute and is dependent on department number, a subset of a key, we call this dependency a partial dependency.

In general, if we have a composite key {A, B} and the dependencies below:



we say that C is partially dependent on {A, B}.

Exercises

1) Suppose each delivery of a course is called a section. In any one term suppose a course may have multiple sections and each section is assigned an instructor. Each course has a course title. Consider a Section relation where the PK is {dept number, course number, section number}. What FDs exist? Is there a partial dependency?

deptNo	courseNo	sectionNo	instructor	title
91	1906	001	J. Smith	Java I
91	1906	002	D. Grand	Java I
91	1910	001	J. Smith	Java II
91	1910	002	J. Daniels	Java II
53	1906	001	S. Farrell	History of the World
...	...	...	...	...

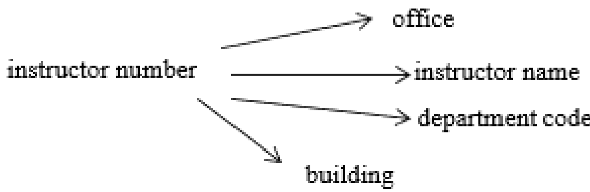
2) Consider a relation with attributes X, Y, Z, W where the only CK is {X,Y}, and where the FDs are {X,Y} –> Z, {X,Y} –> W, and Y –> W. Is there a partial dependency?

10.1.4: Transitive Functional Dependencies

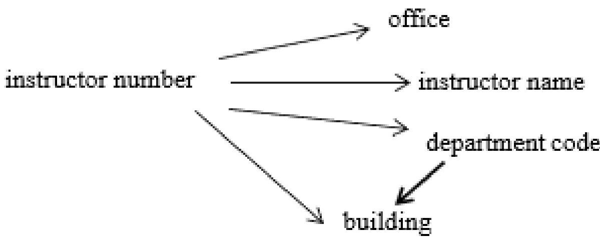
Consider a relation that describes a couple of concepts, say instructor and department, and where the building shown is the building where the department is located, and the attribute instructor number is the only key:

instructor number	instructor name	office	department code	building
33	Joe	3D15	B&A	Buhler
44	Joe	3D16	ACS	Duckworth
45	April	3D17	ACS	Duckworth
50	Susan	3D17	ACS	Duckworth
21	Peter	3D18	B&A	Buhler
22	Peter	3D18	MATH	Duckworth

As instructor number is the only key, we have the following FDs:



Suppose we also have the FD: department code determines building. Now our FD diagram becomes:



and we say the FD from instructor number to building is **transitive** via department code.

In general, if we have a relation with key A and functional dependencies:
A –> B and B –> C, then we say attribute A **transitively** determines attribute C.

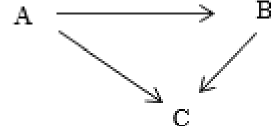


Figure 10.9: Non-key attributes and a transitive dependency

Note: B and C above are non-key attributes. If we also had the functional dependency B –> A (and so A and B are candidate keys) then A does **not** transitively determine C.

Exercises

1) Consider a relation that describes an employee including the province where the employee was born. Suppose the only key is employeeId and we have the attributes: name, birthDate, birthProvince, currentPopulation.

employeeId	name	birthDate	birthProvince	currentPopulation
123	Joe	Jan 1, 1990	MB	1,200,000
222	Jennifer	Jan 5, 1988	SK	1,450,000
345	Jimmy	Feb 5, 1987	MB	1,200,000
…	…	…	…	…

- What FDs would exist? Is there a transitive dependency?

2) Consider a relation with attributes X, Y, Z, W where the only CK is X, and the FDs are X –> Y, X –> Z, X –> W and Y –> Z. Is there a transitive dependency?

10.2: Normal Forms

The normal forms usually of interest to the database designer are 1NF, 2NF, 3NF and BCNF. There are more (higher) normal forms that we leave to follow-up courses. We discuss 1NF and BCNF; 2NF and 3NF are mentioned in our summary. 1NF is so important, it is actually a property of a relation; that is, to say something is a relation means that it is at least in 1NF. BCNF has a simple definition (compared to 2NF and 3NF) and is the usual objective of the designer.

If you understand 1NF and BCNF then you have good insight into the nature of relations that are easy to understand and maintain. If you understand why a relation is not BCNF then you will know the source of its redundant data, which is necessary to know how to properly maintain the data contained in the relation. In most practical cases when a relation is not BCNF the reason will be related to partial or transitive dependencies. 2NF relations do not have partial dependencies, and 3NF relations do not have partial nor transitive dependencies.

10.2.1: First Normal Form(1NF)

We say a relation is in ***1NF*** if all values stored in the relation are single-valued and atomic. With this rule, we are simplifying the structure of a relation and the kinds of values that are stored in the relation.

EXAMPLE 1.

Consider the following EmployeeDegrees relation:

- empNo is the PK
- each employee has one first name and one salary
- each employee has zero or more university degrees ... stored as a single attribute

empNo	first name	salary	degrees
111	Joe	29,000	BSc, MSc
200	April	41,000	BA, MA
205	Peter	33,000	BEng
210	Joe	20,000	

This relation is not in 1NF because the degrees attribute can have multiple values. Below are two relations formed by splitting EmployeeDegrees into two relations – one relation has attributes empNo, first name, and salary and the other has empNo and degree. We say we have *decomposed* EmployeeDegrees into two relations and we have populated each with data from EmployeeDegrees. Each of these is in 1NF, and if we join them on empNo, we can get back the information shown in the relation above.

empNo	first name	salary
111	Joe	29,000
200	April	41,000
205	Peter	33,000
210	Joe	20,000

empNo is the PK.

each employee has one name and one salary.

empNo	degree
111	BSc
111	MSc
200	BA
200	MA
205	BEng

{empNo, degree} is the PK.

degree is single-valued.

EXAMPLE 2.

Consider the Student relation below. The name attribute comprises both first and last names, and so it's not atomic. Student is not in 1NF:

studentNo	name	gender
444	Jim Smith	m
254	Donna Jones	f
333	Peter Thomas	m
765	Jim Smith	m

If we modify Student so there are two attributes (say, first and last) then Student would be in 1NF:

studentNo	first	last	gender
444	Jim	Smith	m
254	Donna	Jones	f
333	Peter	Thomas	m
765	Jim	Smith	m

If we can say that a relation (or table) is in 1NF then we are saying that every attribute is atomic, and every value is single-valued. This simplifies the form of a relation.

It is very common for names to be separated out into two or more attributes. However, attributes such as birth dates, hire dates, etc. are usually left as a single attribute. Dates could be separated out into day, month, and year attributes, but that is usually beyond the needs of the intended system. Some would take the view that separating a date into 3 separate attributes is carrying the concept of normalization a little too far. Database systems have convenient functions that can be used to obtain a day, month, or year values from a date.

Exercises

1) Consider the relation below that holds information about courses and sections. Suppose departments have courses and offer these courses during the terms of an academic year. A section has a section number, is offered in a specific term (e.g. Fall 2016, Winter 2017) and is assigned a slot (e.g. 1, 2, 3, …15) for the term. Each time a course is delivered there is a section for that purpose. Each section of a course has a different number. As you can see a course may be delivered many times in one term. The delivery attribute is multi-valued and is composite.

deptNo	courseNo	delivery
ACS	1903	001, Fall 2016, 05; 002, Fall 2016, 06; 003, Winter 2017, 06
ACS	1904	001, Fall 2016, 12; 002, Winter 2017, 12
Math	2201	001, Fall 2016, 11; 050, Fall 2016, 15
Math	2202	050, Fall 2016, 15

- Modify CourseDelivery to be in 1NF. Show the contents of the rows for the above data.

2) Chapter 8 covered mapping an ERD to a relational database. Consider the examples from Chapter 8; are the relations in 1NF?

10.2.2: Boyce-Codd Normal Form (BCNF)

Initial research into normal forms led to 1NF, 2NF, and 3NF, but later[1] it was realized that these were not strong enough. This realization led to BCNF which is defined very simply:

> A relation R is in **BCNF** if R is in 1NF and every determinant of a non-trivial functional dependency in R is a candidate key.

BCNF is the usual objective of the database designer; BCNF is based on the notions of candidate key (CK) and functional dependency (FD). When we investigate a relation to determine whether it is in BCNF or not, we must know what attributes or attribute combinations are CKs for the relation, and we must know the FDs that exist in the relation. Our knowledge of the semantics of a relation guides us in determining CKs and FDs.

Recall that a CK is an attribute, or attribute combination, that uniquely identifies a row. Also, recall a CK is minimal – no attribute can be removed without losing the property of being a key.

Recall that a FD X –> Y in a relation R means that for each row in the relation R that has the same value for X the value of Y must also be the same. Recall that when we consider a FD X –> Y we refer to the left-hand side, attribute X, as the determinant. We are concerned with minimal FDs – all attributes comprising the determinant are required for the FD property to hold. If X –> Y is a FD, then the determinant augmented with any other attribute is also a FD, but it would not be a minimal FD.

We consider several examples. We keep the examples simple and to the point, each relation involves very few attributes. This is of course unrealistic – in practice relations usually have many attributes. However, the examples illustrate one point each, and more attributes in the relations may cloud the issues. Each example begins with a relation that is in 1NF.

In general, when we determine the relation under consideration is not in BCNF we obtain BCNF relations by decomposing the relation into two or more relations that are in BCNF. In this process we say we take a projection of the original relation on a subset of its attributes and at the same time we eliminate any duplicate rows. An important property of the decomposition is that it must be lossless – the new relations will have attributes in common that can be used to join the new relations whereby we can realize the original relation. All rows of the original relation are obtained in the join, and no new or spurious rows are generated – we get back the original relation exactly.

In Example 1 we have a 'good' relation, one that is in BCNF. Hence, no decomposition is required. We discuss the CDs and FDs for the relation thereby knowing it is in BCNF.

Example 2 presents a relation that is not in BCNF. There is a type of redundancy present in its data. We illustrate how to decompose the relation into two relations that are each in BCNF. This example illustrates a type of dependency known as a partial functional dependency.

Example 3 presents another relation that is not in BCNF. There is a type of redundancy present in its data. We illustrate how to decompose the relation into two relations that are each in BCNF. This example illustrates a type of dependency known as a transitive functional dependency.

Our last example is a case where FDs involve overlapping candidate keys, and where FDs exist amongst attributes that make up CKs. There is a type of redundancy present which is not related to 2NF and 3NF. BCNF gives us a theoretical basis for recognizing the source of the redundant data.

EXAMPLE 1.

Consider the Employee relation below that depicts sample data for 5 employees. The semantics are quite simple: for each employee identified by a unique employee number we store the employee's first name and last name.

id	first	last
1	Joe	Jones
2	Joe	Smith
3	Susan	Smith
4	Abigail	McDonald
5	Abigail	McDonald

Candidate Keys.

The hypothetical company that uses this relation identifies employees by an identification number that is assigned by the Human Resources Department, and they ensure each employee has a different id from every other employee. Clearly id is a candidate key.

When an employee is hired they have a first and last name, and the company has no control over these names. As the sample data shows, more than one employee can have the same first name (id 1 and 2), can have the same last name (id 2 and 3), and can even have the same first and last names (id 4 and 5). So, id is the only candidate key for this relation.

Functional Dependencies.

Since each row/employee has a unique identifier, it is easy to see there are two FDs for this relation:

- id –> first
- id –> last

There are no other FDs. For example, we cannot have first –> last. The sample data shows there can be many last names associated with any one first name.

These two FDs are minimal as the determinant, id, cannot be reduced at all.

BCNF?

In this example we have one candidate key, id, and this attribute is the determinant in all FDs. Therefore, **Employee relation is in BCNF**; it is a 'good' relation.

This relation has a 'nice' simple structure; there is one candidate key which is the determinant for every FD.

EXAMPLE 2.

Consider the following relation named Enrollment:

stuNum	courseId	birthdate
111	2914	Jan 1, 1995
113	2914	Jan 1, 1998
113	3902	Jan 1, 1998
118	2222	Jan 1, 1990
118	3902	Jan 1, 1990
202	1805	Jan 1, 2000

The semantics of this relation are:

- Each row represents an enrollment of a student in a course.
- A student is identified by their student number.
- A course is identified by a course identifier.
- A student can only enroll in a course once. Hence the combinations {stuNum,courseId} are unique.
- The birthdate column holds the date of birth for the student of that row. When the same student number appears in more than one row then the birthdate appears redundantly.
- A course can have many students registered in it

Candidate Keys.

It should be clear that several rows may exist for any given student number, and several rows may exist for any given course number. Also, since we cannot control when someone is born there can be many rows for a value of birthdate. All this just means that no single attribute uniquely identifies a row and so no single attribute can be a CK. Any CKs for this relation must be composite – comprising more than one attribute. It should be fairly clear, given the semantics of the relation, the only attribute combination that is a CK is {stuNum, courseId}. For any given value of {stuNum, courseId} there can be at most one row.

Functional Dependencies.

This relation is quite simple in that there is just one FD: stuNum –> birthdate. If a specific student number appears in more than one row, the value stored for birthdate must be the same in all such rows.

BCNF?

Enrollment has one CK: {stuNum, courseId}, and has one FD (stuNum –> birthdate) where the determinant is not a candidate key. Therefore, **Enrollment is not in BCNF**.

In this relation we have an attribute that does not describe the whole key – it describes a part of the key. In normalization theory the FD stuNum –> birthdate is called a **partial functional dependency** as its determinant is a subset of a candidate key.

When you think of the Enrollment relation now, you should consider that it is about two very different things:

- Enrollment presents enrollment information.
- Enrollment presents information about students (their birthdates).

Decomposition.

We now consider how Enrollment can be replaced by two relations where the new relations are each in BCNF. Above we mentioned that Enrollment is about two very different things – what we need to do is arrange for two relations, one for each of these concerns.

Consider the following two relations as a decomposition of the above, where we have placed information about enrollments in one relation and information about students in another relation. Note that these two relations have the stuNum attribute in common.

Enrollments	
stuNum	courseId
111	2914
113	2914
113	3902
118	2222
118	3902
202	1805

Students	
stuNum	birthdate
111	Jan 1, 1995
113	Jan 1, 1998
118	Jan 1, 1990
202	Jan 1, 2000

Enrollments and Students can be joined on stuNum to reproduce the exact information in Enrollment. Because we have not lost any information, and noting that the FD has been preserved, these two relations are equivalent to the one we started with.

- Enrollments has one candidate key: {stuNum, courseId}, and no FDs. Therefore, **Enrollments is in BCNF**.
- Students has one CK: stuNum, and has one FD: stuNum –> birthdate. Therefore, **Students is in BCNF**.

EXAMPLE 3.

Consider the following relation named Course.

courseId	teacherId	lastName
2914	11	Smith
3902	22	Jones
3913	11	Smith
4902	33	Jones
4906	11	Smith
4994	22	Jones

The purpose of this relation is to record who is teaching courses. Note that a teacher's id and last name may appear in several rows – this information is repeated for each course the teacher is teaching. For example, teacher 11 (Smith) is teaching 3 courses (2914, 3913, 4906) and so we see the same id and last name in three rows.

The semantics of this relation are:

- Each course is identified by a course identifier.
- For each course there is one row.
- Each teacher is identified by a teacher identifier.
- Each course has one teacher, and so for each course one teacher Id is recorded.
- A teacher may teach several courses.
- A teacher's last name must be the same in every row where the teacher's Id appears. This point leads to redundant data in the relation.

Candidate Keys.

The semantics of the relation are that there is one row per course, and so a course id uniquely identifies a row; so, courseId is a candidate key. No other attribute or combination can be a candidate key for this relation.

Functional Dependencies.

It is stated there is one teacher per course and so for each courseId there is at most one teacherId, and so we have courseId→teacherId. The opposite, teacherId→courseId, does not hold for this relation since a teacher can teach more than one course.

Another FD that is present is teacherId→lastName. This is because for each teacher there is a single last name. Note the opposite, lastName→teacherId does not hold in this relation. The sample data shows multiple teachers who have the same last name.

Note that since courseId→teacherId and teacherId→lastName, it must be true we have the FD courseId→lastName. For each course we have one teacher and so one last name. For any value of course id there will only be one value for teacher last name. In relational database theory the FD courseId→lastName is called a **transitive functional dependency** – lastName is dependent on courseId but this dependency is via teacherId.

BCNF?

Hopefully you agree the only FDs are these:

- courseId –> teacherId
- teacherId –> lastName
- courseId –> lastName

The only candidate key is courseId, and there is a FD, teacherId –> lastName, where the determinant is not a candidate key. Therefore, **Course is not BCNF**.

When you think of the Course relation now, you should see that it is about two very different things:

- Course presents teacher information (teacherId) for courses
- Course presents information about teachers (their last names)

Decomposition.

Course can be replaced by two relations where the new relations are each in BCNF. Above we mentioned that Course is about two very different things – what we need to do is arrange for two relations, one for each of these concerns.

Consider the following two relations as a decomposition of the above where we have placed information about courses in one table and information about teachers in another table. These relations have a common attribute, teacherId.

Courses	
courseId	**teacherId**
2914	11
3902	22
3913	11
4902	33
4906	11
4994	22

Teachers	
teacherId	**lastName**
11	Smith
22	Jones
33	Jones

Courses and Teachers can be joined on teacherId to reproduce exactly the information in Course. Because we have not lost any information and noting that the FD has been preserved as well, these two relations are equivalent to the one we started with.

- Courses has one candidate key: courseId. The only FD is courseId –> teacherId. Therefore, **Courses is in BCNF**.
- Teachers has one candidate key: teacherId. There is one FD: teacherId –> lastName. Therefore, **Teachers is in BCNF**.

EXAMPLE 4.

This example uses a relation that contains data obtained from a 2011 Statistics Canada survey. Each row gives us information about the percentage of people in a Canadian province who speak a language considered their mother tongue[2]. The ellipsis "…" indicate there are more rows.

provCode	provName	language	percentMotherTongue
MB	Manitoba	English	72.9
MB	Manitoba	French	3.5
MB	Manitoba	non-official	21.5
SK	Saskatchewan	English	84.5
SK	Saskatchewan	French	1.6
SK	Saskatchewan	non-official	12.7
NU	Nunavut	English	28.1
…	…	…	…

The ProvinceLanguageStatistics relation has redundant data. In the rows listed above we see that each province name and each province code appear multiple times.

Candidate Keys.

There can be more than one row for any province, but for the combination of province and language there can be only one row and so there are two composite candidate keys:

- {provCode, language}
- {provName, language}

Functional Dependencies.

Since province codes and province names are unique, we have the FDs:

- provCode –> provName
- provName –> provCode

For each combination of province and language there is one value for percent mother tongue, and so we have FDs:

- provCode,language –> percentMotherTongue
- provName,language –> percentMotherTongue

BCNF?

The first two FDs listed above have determinants that are subsets of candidate keys. Therefore, **ProvinceLanguageStatistics is not BCNF**.

The ProvinceLanguageStatistics relation has information about two different things:

- It has information about provinces (names/codes).
- It has information about mother tongues in the provinces.

Decomposition.

To obtain BCNF relations we must decompose ProvinceLanguageStatistics into two relations; for example, consider Province and ProvinceLanguages below:

Province	
provCode	provName
MB	Manitoba
SK	Saskatchewan
NU	Nunavut
...	...

ProvinceLanguages		
provCode	language	percentMotherTongue
MB	English	72.9
MB	French	3.5
MB	non-official	21.5
SK	English	84.5
SK	French	1.6
SK	non-official	12.7
NU	English	28.1
NU	French	1.4
NU	non-official	69.6
...	...	...

These relations can be joined on provCode to produce exactly the information shown in ProvinceLanguageStatistics.

- Province has two CKs: provCode and provName, and has two FDs:
  - provCode –>provName
  - provName –> provCode.

  Therefore, **Province is in BCNF**.

- ProvinceLanguages has one CK: {provCode,language}, and one FD:
  - {provCode,language} –> percentMotherTongue.

  Therefore, **ProvinceLanguages is in BCNF**.

1 Codd, E.F. (1974) — Recent Investigations in Relational Database Systems, Proceedings of the IFIP Congress, pp. 1017–1021.
2 Mother tongue refers to the first language learned at home in childhood and still understood by the person at the time the data was collected. The person has two mother tongues only if the two languages were used equally often and are still understood by the person.

10.3: Summary

We have discussed functional dependencies, candidate keys, 1NF and BCNF. BCNF is the usual objective of the database designer.

When a relation is not BCNF then one or more of the following will be the source of redundancy in a relation:

- partial dependencies
- transitive dependencies
- functional dependencies amongst key attributes.

2NF involves the concepts of candidate key and non-key attributes. A relation is considered to be in **2NF** if it is in 1NF, and every non-key attribute is fully dependent on each candidate key. In Example 2 we mentioned that stuNum –> birthdate was considered a partial functional dependency as stuNum is a subset of a candidate key. A 2NF relation does not contain partial dependencies.

3NF involves the concepts of candidate key and non-key attributes. We say a relation is in **3NF** if the relation is in 1NF and all determinants of non-key attributes are candidate keys. In Example 3 we mentioned that courseId –> lastName was considered a transitive dependency; lastName is dependent on teacherId which is not a candidate key. A 3NF relation does not have partial dependencies nor transitive dependencies.

The definition of BCNF concerns FDs and CKs – there is no mention of non-key attributes. Hence, BCNF is a stronger form than 2NF or 3NF (a BCNF relation will be in 2NF and 3NF).

A database designer may decide to not normalize completely to BCNF. This is sometimes done to ensure that certain data can be retrieved without having to join relations in a query – when a join is avoided the data is typically retrieved more quickly from the database. This is often done in a data warehouse environment (outside the scope of these notes).

Exercises

In each of these exercises, consider the relation, CKs, and FDs. Determine if the relation is in BCNF, and if not in BCNF give a non-loss decomposition into BCNF relations. The last 5 questions are abstract and give no context for the relation nor attributes.

1) Consider a relation Player which has information about players for some sports league. Player has attributes id, first, last, gender – id is the only CK and the FDs are:

- id –> first
- id –> last
- id –> gender

Player – sample data:

id	first	last	gender
1	Jim	Jones	Male
2	Betty	Smith	Female
3	Jim	Smith	Male
4	Lee	Mann	Male
5	Sarah	McDonald	Female

2) Consider a relation Employee which has information about employees in some company. Employee has attributes id, first, last, sin (social insurance number) where id and sin are the only CKs, and the FDs are:

- id –> first
- id –> last
- sin –> first
- sin –> last
- id –> sin
- sin –> id

Employee – sample data:

id	first	last	sin
1	Jim	Jones	111222333
2	Betty	Smith	333333333
3	Jim	Smith	456789012
4	Lee	Mann	123456789
5	Samantha	McDonald	987654321

3) Consider a relation Player which contains information about players and their teams. Player has

attributes playerId, first, last, gender, teamId, teamName, teamCity where playerId is the only CK and the FDs are:

- playerId –>first
- playerId –> last
- playerId –> gender
- playerId –> teamId
- playerId –> teamName
- playerId –> teamCity
- teamId –> teamName
- teamId –> teamCity

Player – sample data:

playerId	first	last	gender	teamId	teamName	teamCity
1	Jim	Jones	M	1	Flyers	Wpg
2	Betty	Smith	F	5	OilKings	Cgy
3	Jim	Smith	M	10	Oilers	Edm
4	Lee	Mann	M	1	Flyers	Wpg
5	Samantha	McDonald	F	5	OilKings	Cgy
6	Jimmy	Jasper	M	99	OilKings	Wpg

4) Consider a relation Building which has information about buildings and floors. Building has attributes buildingCode, floor, numRooms, campus where {buildingCode,floor} is the only CK and the FDs are:

- {buildingCode,floor} –> numRooms
- buildingCode –> campus

Building – sample data:

buildingCode	floor	numRooms	campus
D	3	15	Downtown
C	2	5	Downtown
RP	1	20	Selkirk
D	2	5	Downtown
D	1	20	Downtown

5) Consider a relation Course which contains information about courses. Course has attributes deptCode, deptName, courseNum, creditHours where {deptCode,courseNum} and {deptName,courseNum} are the only CKs, and the FDs are:

- {deptCode,courseNum} –> creditHours
- {deptName,courseNum} –> creditHours deptCode ⎝ deptName
- deptName –> deptCode

Course – sample data:

deptCode	deptName	courseNum	creditHours
Math	Mathematics	2101	3
Stat	Statistics	2102	3
Math	Mathematics	2102	1
Stat	Statistics	4001	6
Math	Mathematics	4001	6

6) Consider the relation Student Performance below which describes student performance in courses. The value stored in the gradePoint column is the grade point that corresponds to the grade received in a course. Assume that students are identified by their student number, and that courses are identified by their course id. Assume each student can take a course only once and so each row is

uniquely identified by {stuNum, courseId}. Each student's overall gpa is stored – gpa is the average of gradePoint for all courses taken by a student.

Student Performance – sample data:

stuNum	courseId	grade	gradePoint	gpa
111	3030	C	2.0	2.0
113	3030	C	2.0	2.5
113	4040	B	3.0	2.5
118	2222	C	2.0	2.25
118	4040	C+	2.5	2.25
202	1188	B	3.0	3.0

7) Consider Example 4. Is there another decomposition of ProvinceLanguageStatistics that leads to BCNF relations?

8) Consider a relation R with attributes X, Y, W, Z where X is the only CK, and where there are FDs:

- X –> Y X –> W X –> Z

9) Consider a relation R with attributes X, Y, W, V where X and V are the only CKs, and where there are FDs:

- X –> Y X –>W V –>Y V –> W X –> V V –> X

10) Consider a relation R with attributes X, Y, W, V, Z where X is the only CK, and where there are FDs:

- X –> Y X –> W W –> Z W –> V

11) Consider a relation R with attributes A, B, C, D, E, F where {A,B} is the only CK, and where there are FDs:

- {A,B} –> C
- {A,B} –> D

- A –> E A –> F

12) Consider a relation R with attributes A, B, C, D, E where {A,C} and {B,C} are the only CKs, and where there are FDs:

- {A,C} –> D
- {B,C} –> D
- {A,C} –> E
- {B,C} –> E
- A –> B B –> A

APPENDIX A

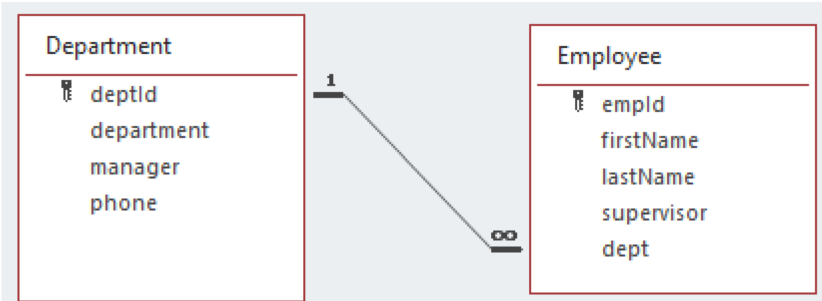
In chapter 3 we created simple forms for single tables. A very useful form is one where the user can interact with data that comes from more than one table. We will consider how this can be done in cases where two tables are related by a one-to-many relationship.

We will illustrate creating such a form using the form wizard. As you will see the form wizard will create a form and a *subform*. These two forms will have a connection established based on related fields: a primary key and a foreign key.

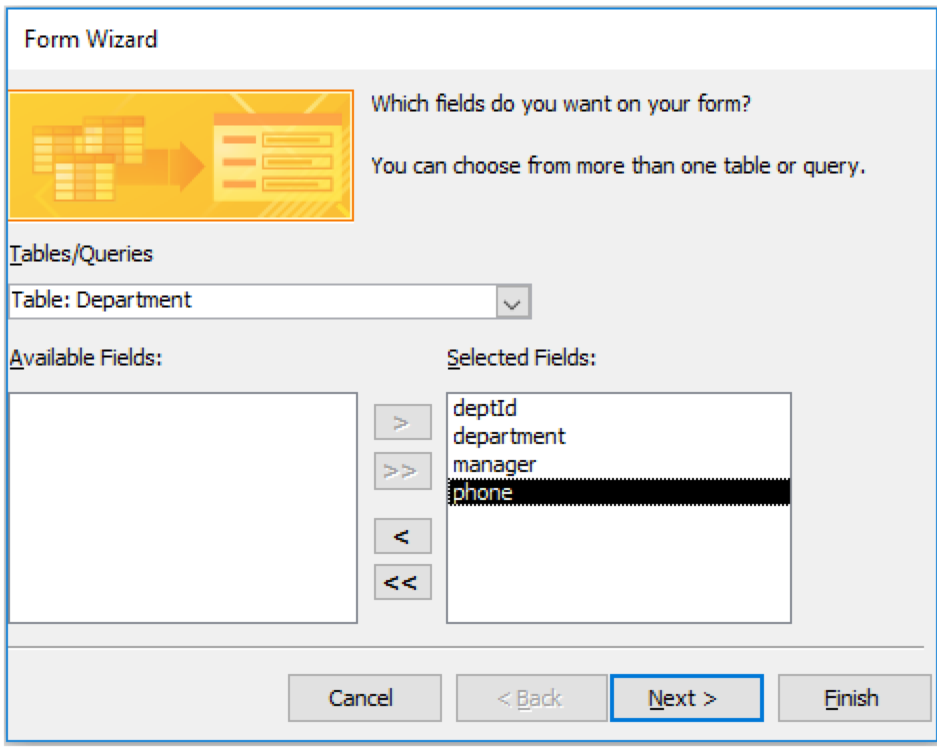
Forms Involving Multiple Tables

Consider the Company database:

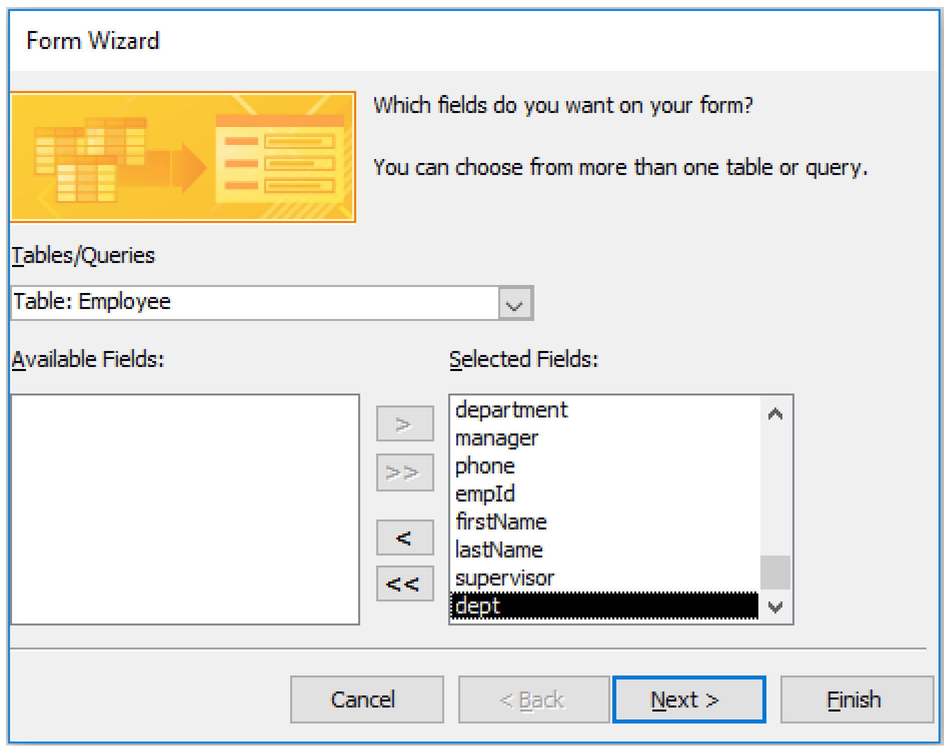
- If the one-to-many relationship between Department and Employee does not exist, then create this now. Note that this is Exercise 1 in Chapter 5. After doing this you should have the relationship as shown:



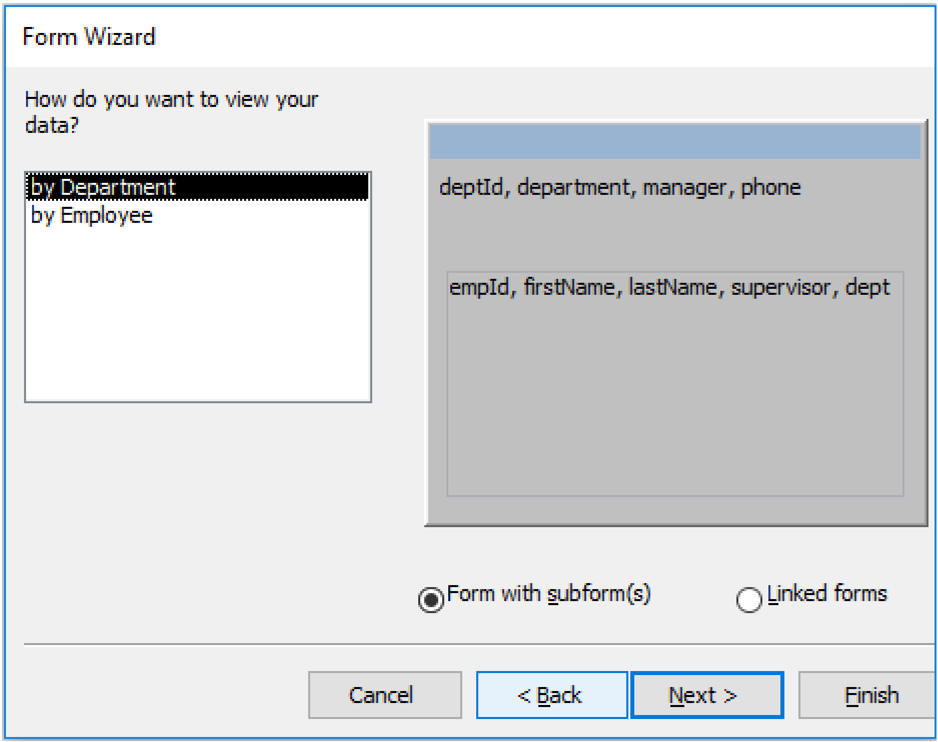
- Use the Create tab and create a form using the Form Wizard. Select all fields from the Department table:



- Do not click Next or Finish, instead choose the Employee table and select all of its fields and now the Selected Fields component shows fields from both tables:

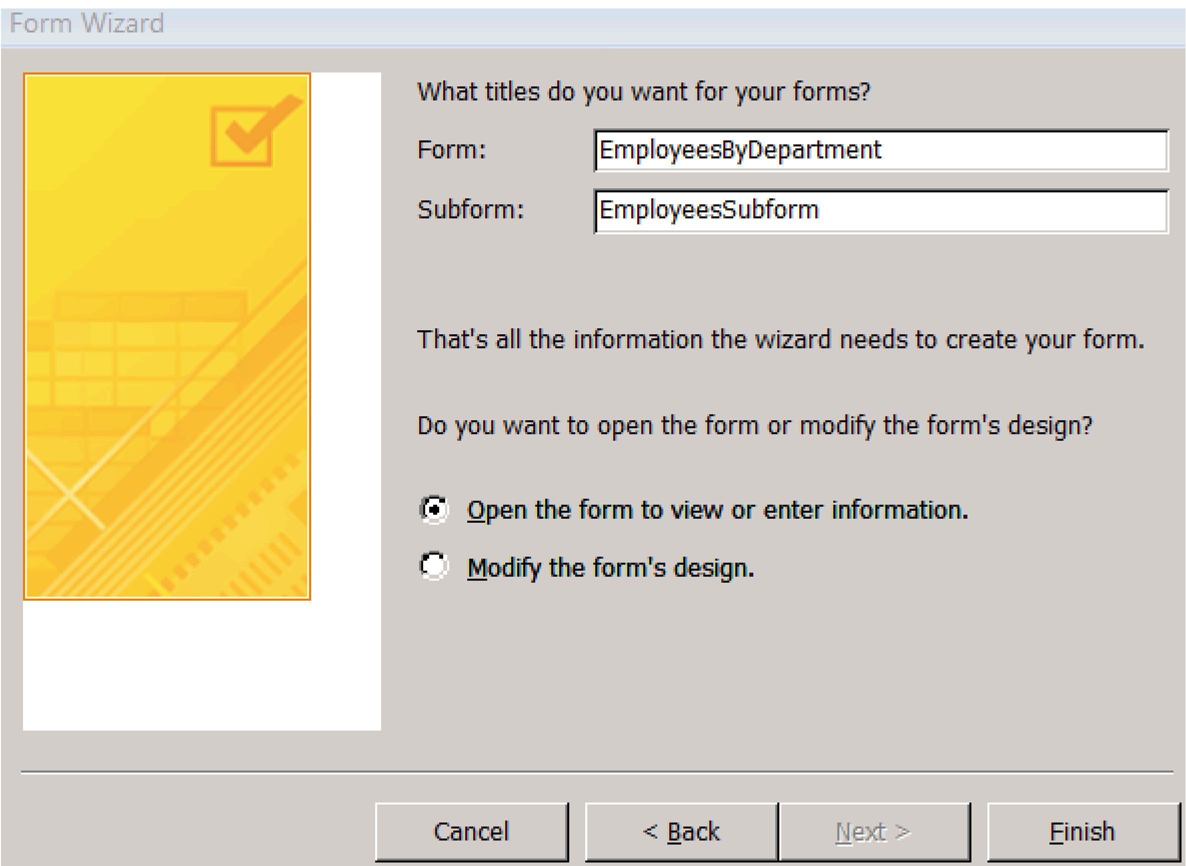


- Now, click Next and MS Access asks you how the data should be viewed:

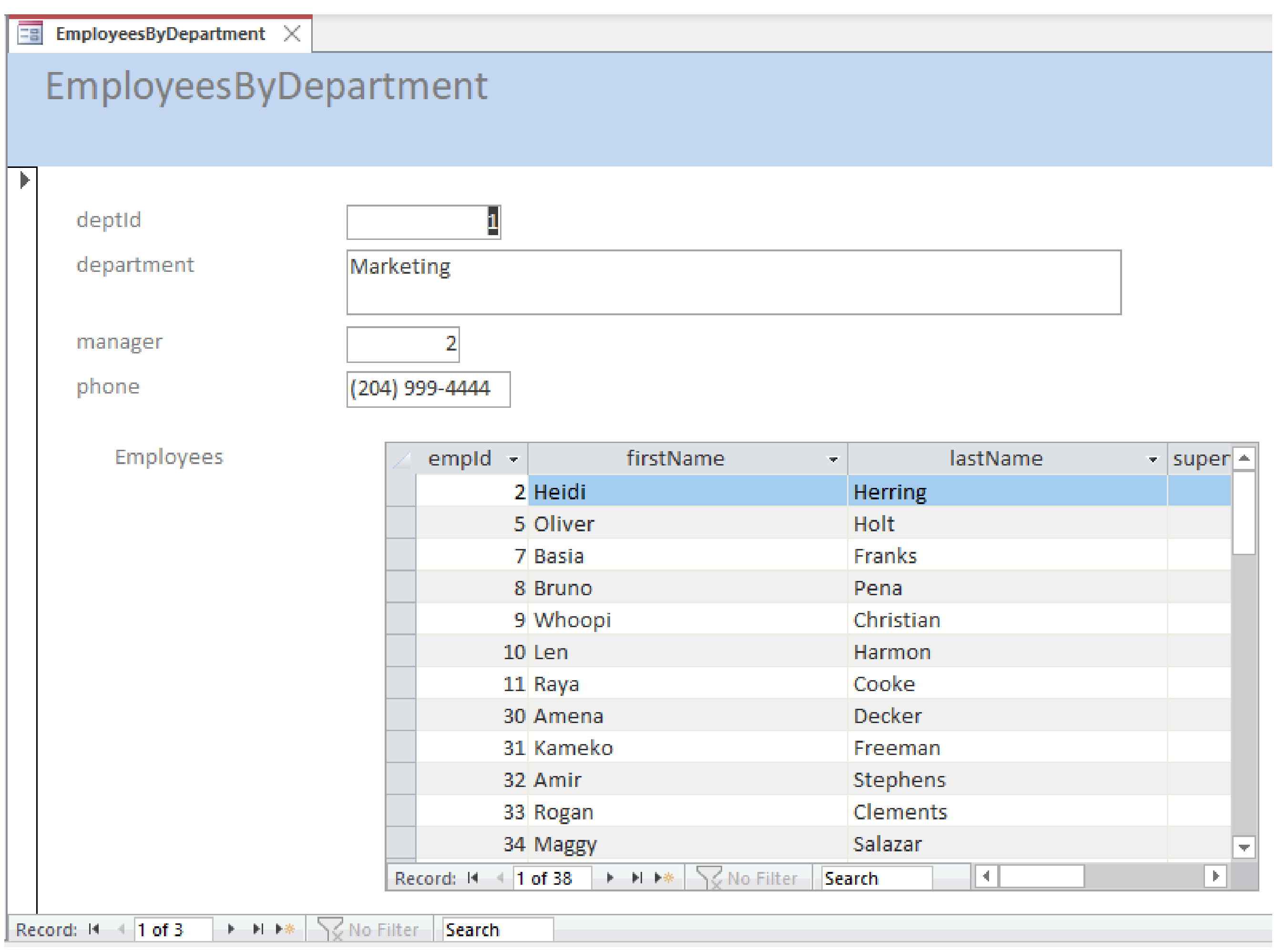


- We want the data displayed “by Department” and we want MS Access to use “Form with subform(s)” so

you can select Next and MS Access will let you choose a layout. Choose Datasheet Layout. Click Next and MS Access will ask you to name the form – name the form EmployeesByDepartment and name the subform EmployeesSubform:



- Click Finish. MS Access will display the finished form called EmployeesByDepartment – see below. Experiment with the form: notice the two sets of navigation buttons – one that controls the department being viewed, and the other that controls the view of the department's employees.



Exercises

1) Consider the University database. Create a form to allow a user to view courses by department.

2) Consider the Library database. There are two one-to-many relationships. Create a form to list the loan records for a book. Create another form to list the loan records for a member.

3) Consider the Orders database. This database has several one-to-many relationships. Create appropriate forms to list:

- a customer and the customer’s orders;
- an order and its detail lines;
- a product and the order detail lines where the product is referenced;
- a category and the products belonging to the category.

APPENDIX B

We have covered the basics of entity-relationship modeling and now we will extend our design capabilities to include supertypes and subtypes. It is often the case that an entity type has subgroupings that are useful to include in a data model. For instance, in a university environment, persons could be grouped into employees and students; courses could be grouped into graduate courses and undergraduate courses. Previously we considered a library database where one entity type was book; instances of book are loaned out to library members. A library could have many other kinds of things that it loans out such as videos and magazines. A more general thing the library loans out can be referred to as an item; videos, magazines, and books can be considered subtypes of item.

We will consider only supertype and subtype hierarchies. Hierarchies arise when an entity type appears as a subtype of only one supertype. So, we are disallowing cases where an entity type has two or more supertypes.

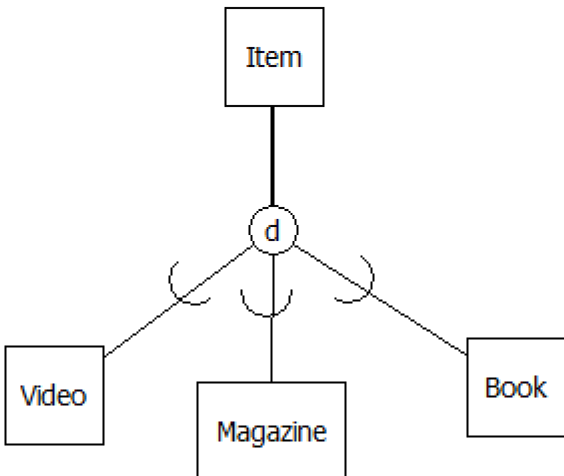
B.1: Drawing Supertypes and Subtypes on the Red

There are different ways that supertypes and subtypes can be shown on an ERD. We will continue with the Peter Chen notation in this appendix. Between a supertype and its subtypes we show a connection symbol (a circle) where one line is drawn from the supertype to the connection symbol and then lines are drawn from the connection symbol to each subtype.

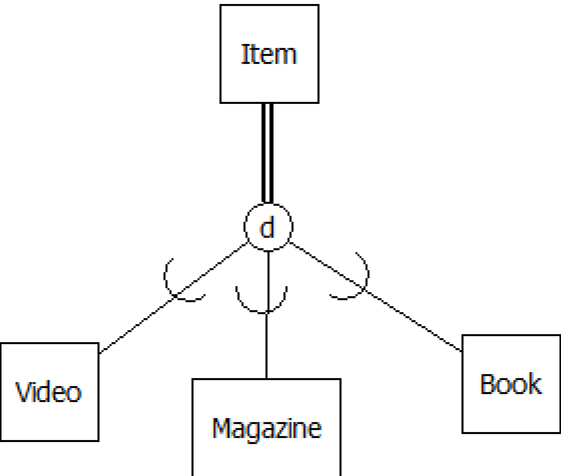
A collection of related subtypes can be regarded as overlapping or disjoint. Subtypes are considered as disjoint if it is impossible for an instance of a supertype to be regarded as being an instance of more than one subtype. For example, a library item will be one and only one of its subtypes (magazine, video, book). Subtypes are considered as overlapping if it is possible for an instance of a supertype to be regarded as being an instance of more than one subtype. An example of overlapping can exist with people in a university environment: it is possible that some person could be both an employee and a student at the same time. In our Peter Chen notation, we will use a “d” in the connection symbol to represent disjoint subtyping, and we will use “o” to represent overlapping.

In our notation we also include an arc on each of the lines joining the connection symbol to the subtypes that implies “containment”.

To illustrate the drawing technique, consider a library where items are loaned to members and where an item can be either a video, a magazine, or a book, and suppose also that an item belongs to exactly one (i.e., disjoint subtypes) of these subtypes. We can show this as:



We extend our notation once more. To indicate that a supertype must exist as one of its subtypes we show total participation in subtyping by using a double line. For example, if we want to show that each item must be one of the subtypes:

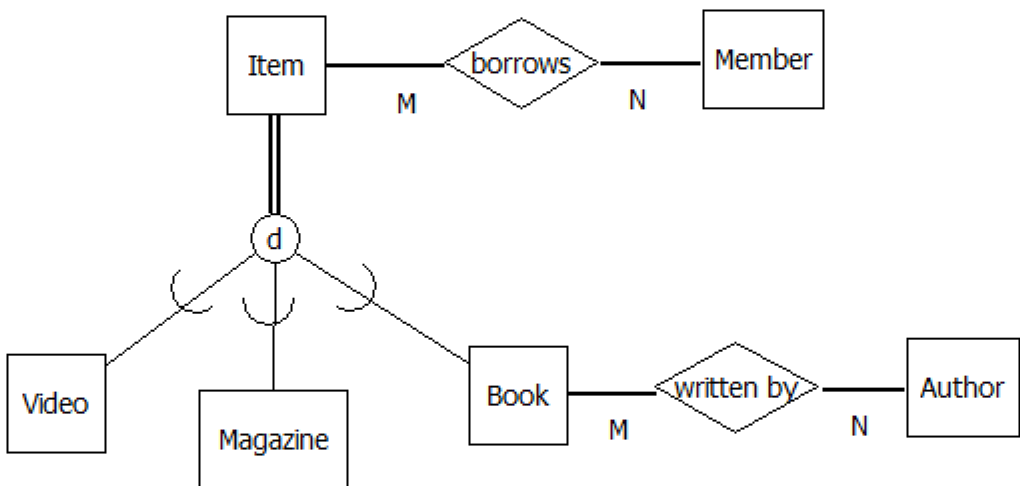


The double line from Item to the connection symbol shows total participation of Item in the subtyping: whenever there is an instance of an Item, then that item must also be one of the subtypes shown – a video, a magazine, or a book. If we did not specify total participation then we would be allowing an item to exist where that item is not a video, nor is it a magazine, nor is it a book. So, participation of a supertype in the subtyping is either total or optional. The converse is always true: if we have an instance of a subtype then that instance is an instance of the supertype. In the library model then, if we have an instance of book then that instance is of course an item.

B.2: Supertypes, Subtypes and Relationships

If a supertype participates in a relationship, then all of its subtypes also participate in that relationship. We say that a supertype's relationships are inherited by its subtypes. The converse is not true: if the model specifies specifically that a subtype participates in a relationship, then its siblings (other entity types that are subtypes of the same supertype) do not participate in that relationship.

As an example, consider that members can borrow items (i.e., any item of any type) from the library but only books have authors. Our model can be extended as follows:



This model excludes the database from storing an author of a magazine or that a video has an author, but the model allows videos, magazines, and books to be borrowed by members.

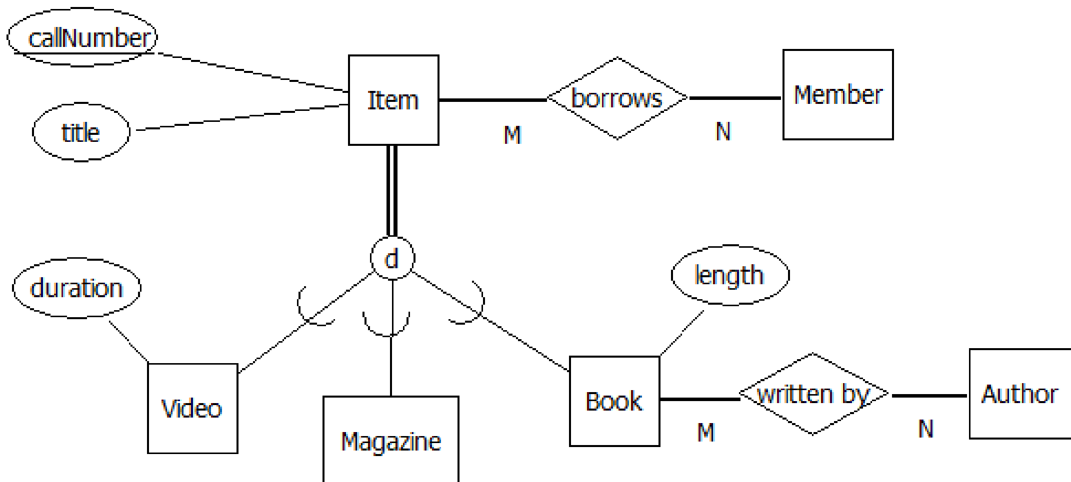
B.3: Supertypes, Subtypes and Attributes

All entity types including supertypes and subtypes can have attributes. Continuing with our library example suppose:

- all items have a call number and a title, and call number is a key (each item has a unique call number),
- videos have a duration (time required to play),
- books have a length (number of pages).

Just as subtypes inherit relationships, they also inherit any attributes of their supertype, and we also have that supertypes do not inherit the attributes of their subtypes. Attributes that are common to a supertype and its subtypes are only shown at the supertype.

Consider our model now:



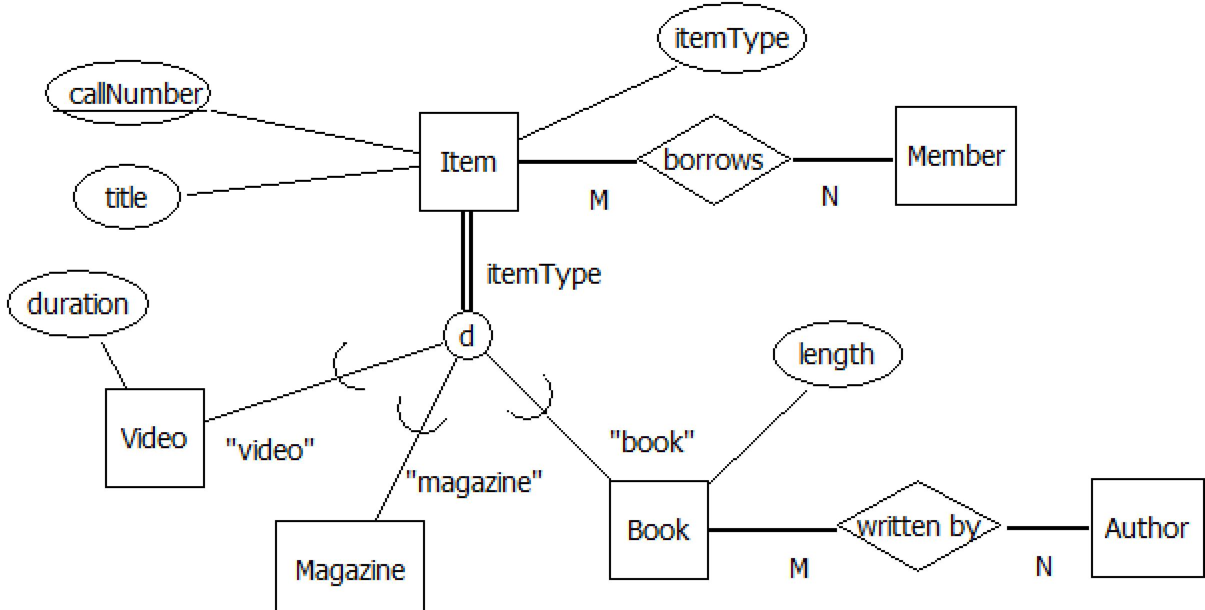
Our examples have been two-level hierarchies. In general, a hierarchy can be as many levels as the designer requires. For instance, books could be categorized as fiction and non-fiction and so book can be a subtype of item and at the same time a supertype of fiction and non-fiction.

B.3.1: Discriminator Attributes

It is common for designers to introduce or discover an attribute such that its value can be used to explicitly determine the subtype an entity belongs to. For example, the item entity type can have an attribute, say itemType, which can have a value from the domain

{“video”, “magazine”, “book”}. When this is done the diagram must include the attribute of course, but additionally the attribute is shown as a discriminator attribute for subtyping purposes and the pertinent value for discriminating shown as well.

Below you will see how these are laid out above and below the connection symbol:



This works well for disjoint subtyping but not necessarily for overlapping subtypes. When overlap is possible a designer may include a discriminator for each subtype, and so there are as many discriminator attributes as there are subtypes. Typically, this is a boolean-valued attribute.

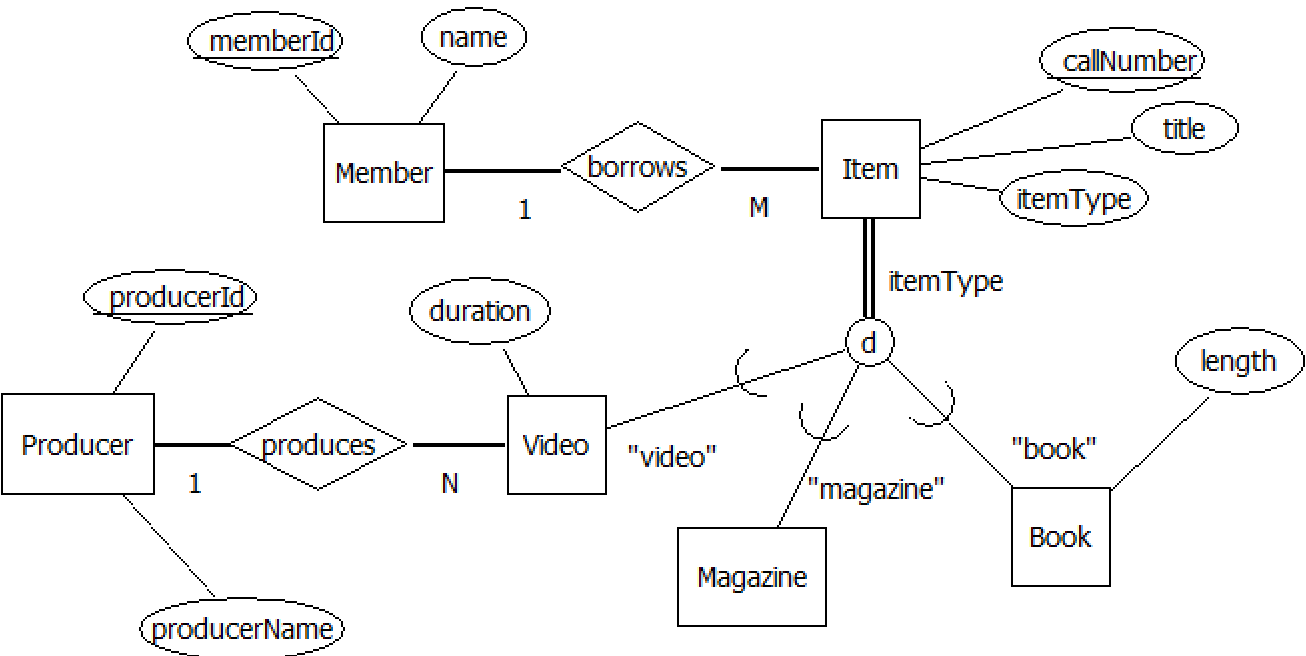
B.4: Mapping Supertypes and Subtypes to a Relational Database

In chapter 8 we covered rules to be used when an ERD is mapped to a relational database. In this section we add rules for mapping supertypes and subtypes to relations. There are three basic options a designer considers when mapping these structures to a database:

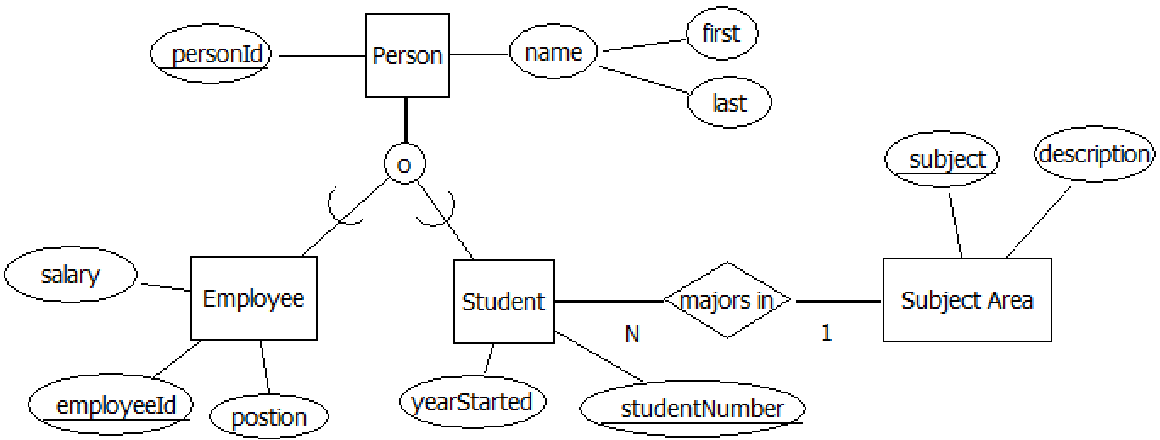
- Create a relation for each entity type in the hierarchy.
- Create relations for only the bottom-most entity types.
- Create one relation to represent the whole hierarchy.

We use two examples to exhibit the mapping options; one where total participation is specified for the supertype and the other where participation is optional.

The previous library model is modified to show that an item can be out on loan to a member, and that one of the subtypes, video, is produced by a producer:



A university model where a person may be a student and/or an employee, and where students declare a major subject area:



Regardless of the option selected for hierarchies, the rules for mapping an ERD to a relational database discussed previously (Chapter 8) still apply. We must apply rules regarding relationships and attributes consistently. For example, if any entity type in a hierarchy is involved in a one-to-many relationship, we must ensure the proper use of foreign keys.

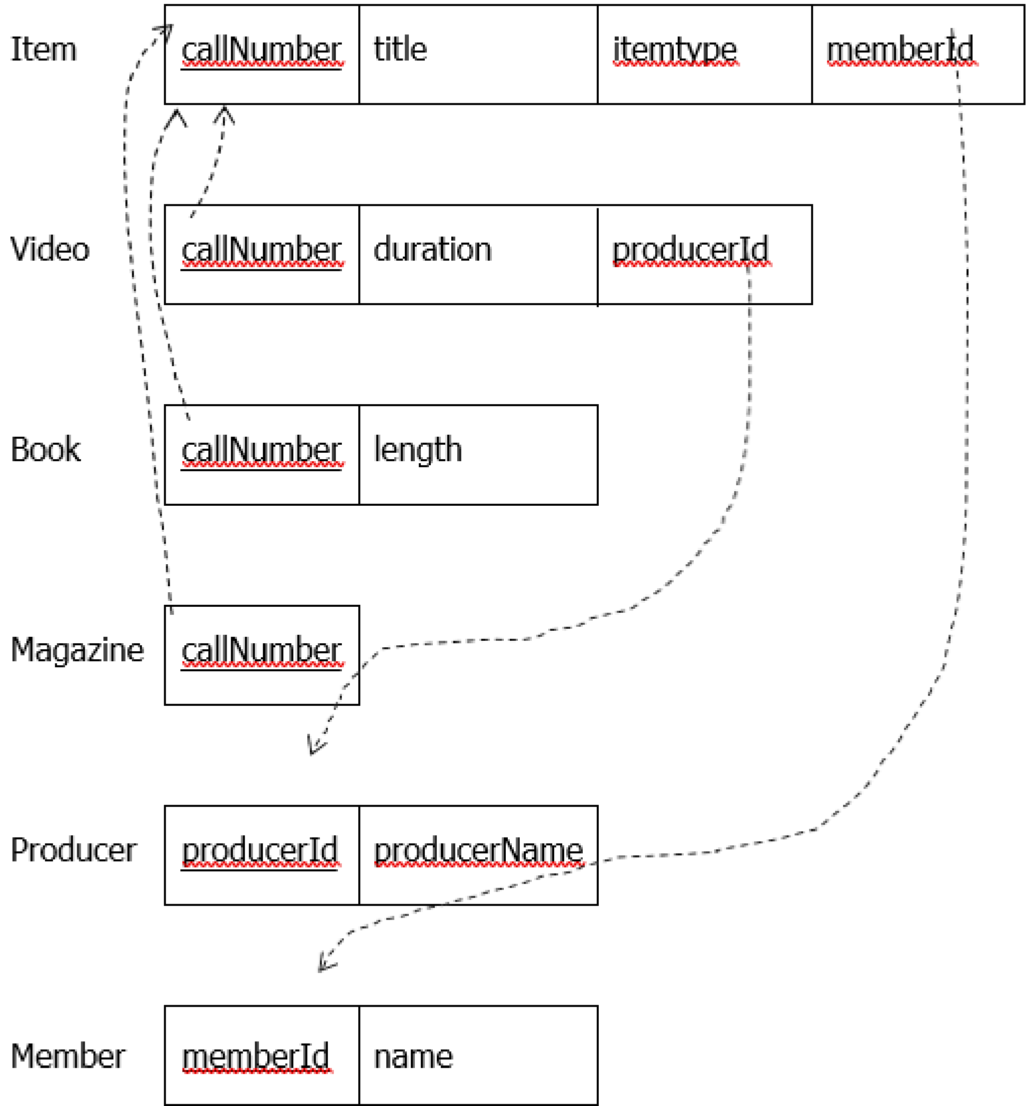
B.4.1: Relations For All Entity Types

With this option each entity type in a hierarchy is represented by its own relation. Important points here are that

- All relations representing entity types in the same hierarchy have the same primary key.
- The primary key of a subtype relation will also be a foreign key that references its supertype relation.
- Attributes of a supertype (except for the primary key) appear only in the relation that represents the supertype.

Example:

The library model maps to the following relational design:

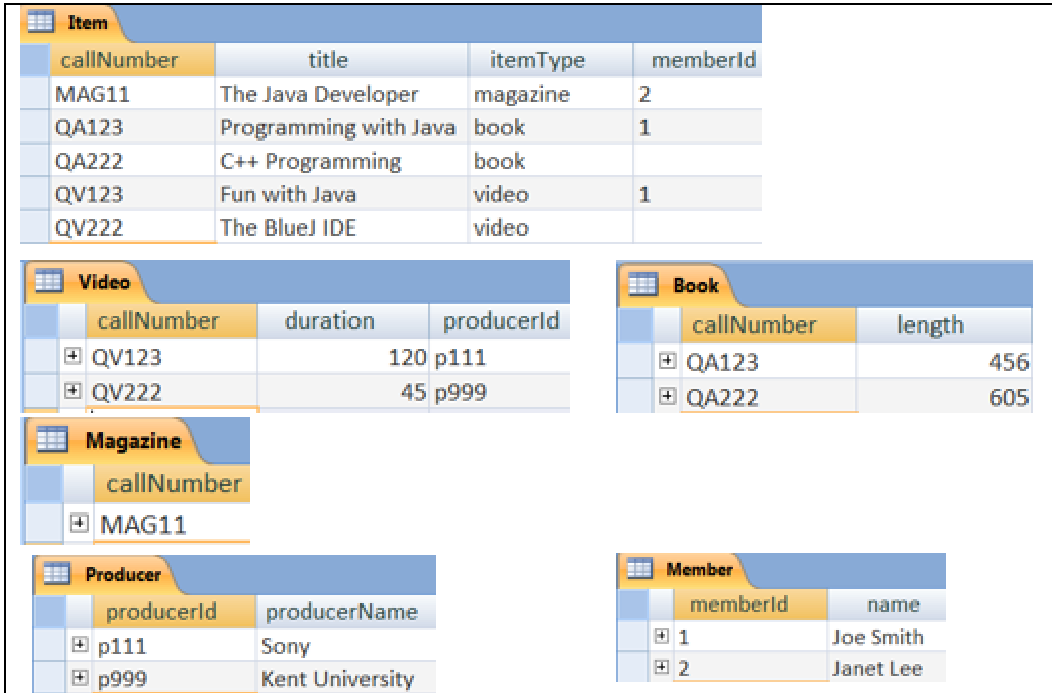


Note the foreign keys:

- Item has a foreign key referencing Member
- Video has a foreign key referencing Producer
- Each of Video, Book, and Magazine has a foreign key referencing Item. If a row exists in Video, Book, or Magazine then there must be a corresponding row in Item.

A sample database is presented on the next page.
The tables are shown here with sample data. Note that:

- each row of Video, Book, and Magazine has a related row in Item
- some items are out on loan to a member
- each video has a producer



Item

callNumber	title	itemType	memberId
MAG11	The Java Developer	magazine	2
QA123	Programming with Java	book	1
QA222	C++ Programming	book	
QV123	Fun with Java	video	1
QV222	The BlueJ IDE	video	

Video

callNumber	duration	producerId
QV123	120	p111
QV222	45	p999

Book

callNumber	length
QA123	456
QA222	605

Magazine

callNumber
MAG11

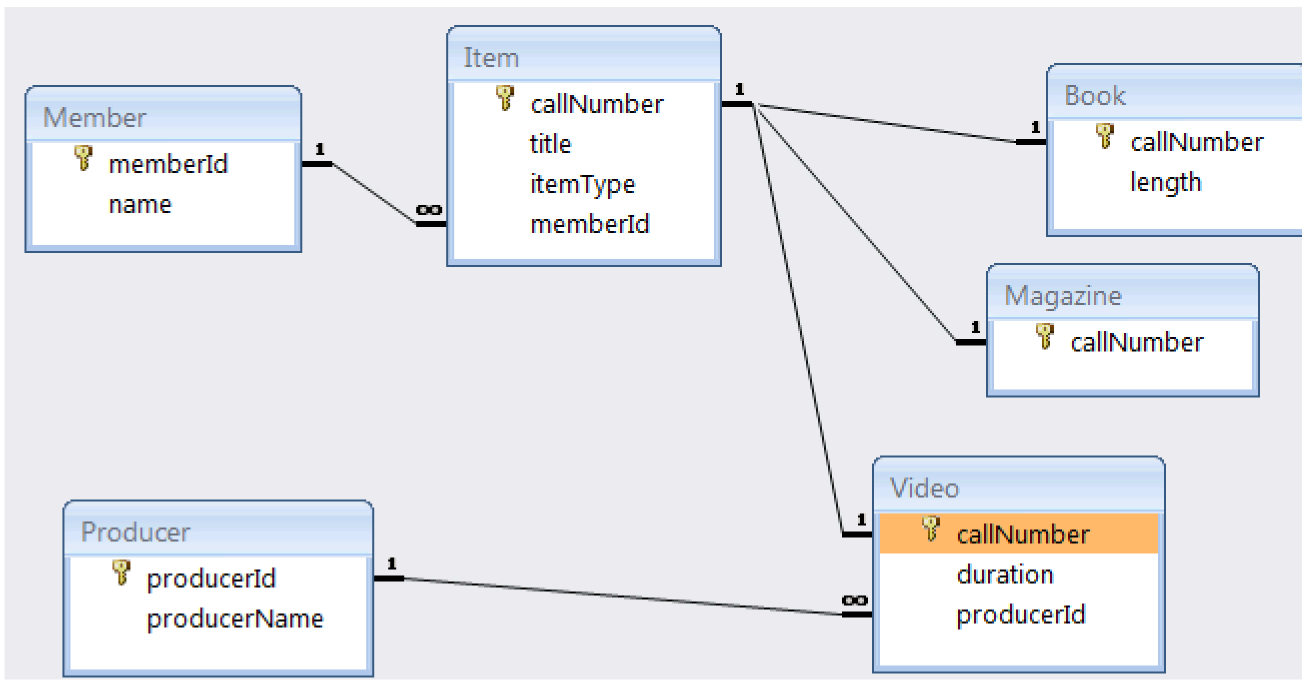
Producer

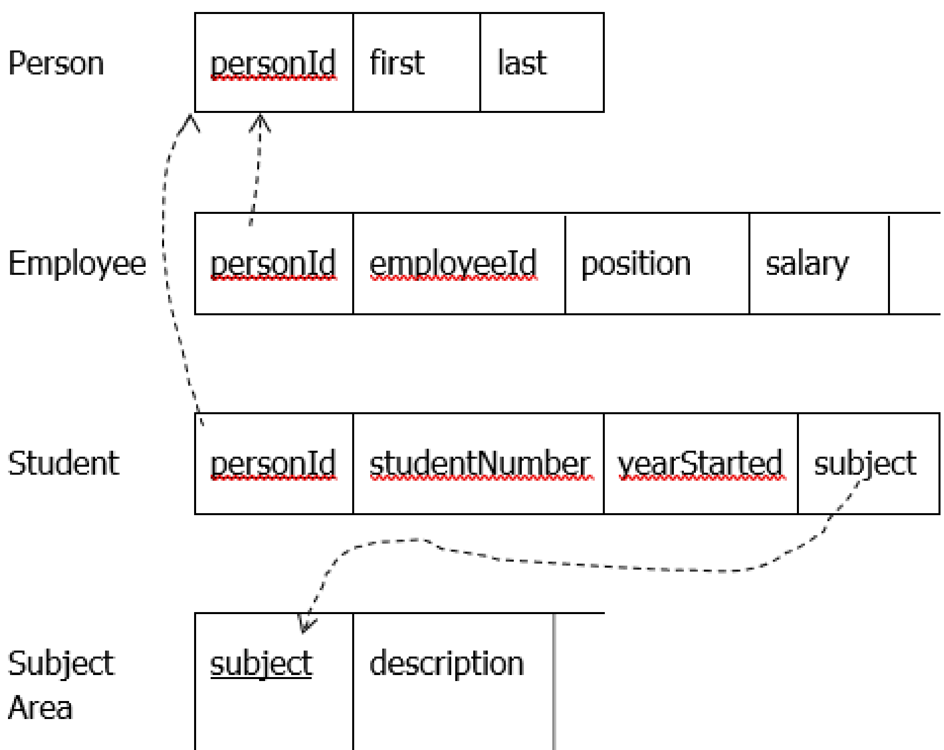
producerId	producerName
p111	Sony
p999	Kent University

Member

memberId	name
1	Joe Smith
2	Janet Lee

In the relationships diagram note the one-to-one relationships between the supertype relation and each of its subtype relations:





Example: Now consider the university model. The relational design for this mapping option:

- Since subtyping is optional in the university model there can be a row in Person with no corresponding row in Employee or Student. A person does not have to exist as one of the subtypes.

Note the foreign keys:

- Student has a foreign key referencing SubjectArea
- Employee and Student have foreign keys referencing Person. If a row exists in Employee or Student, then a corresponding row must exist in Person.

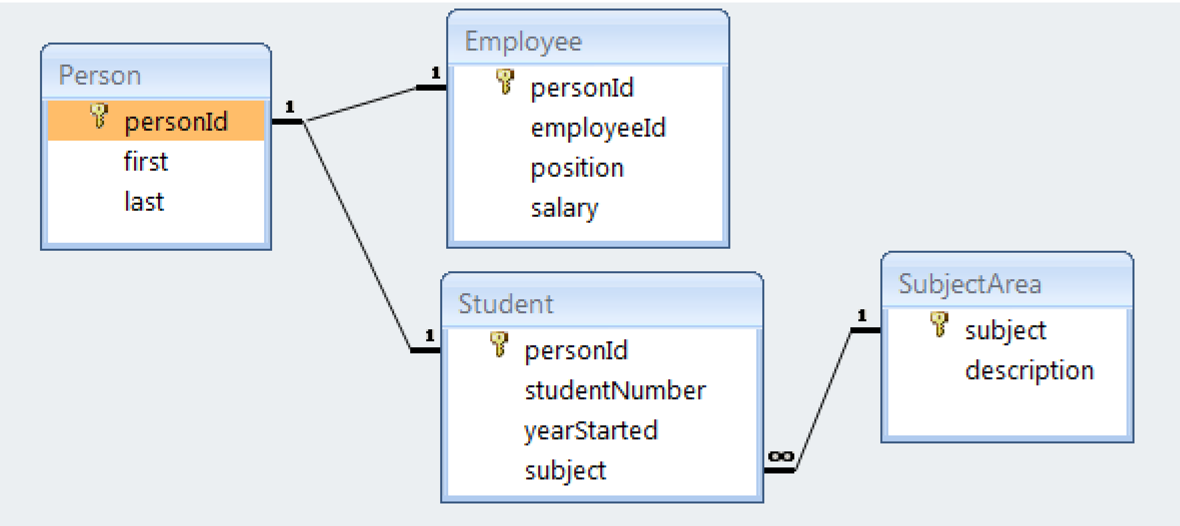
On the following page we show tables with some sample data and the relationships diagram.

A sample database is presented below. Note that person 2 is both a student and an employee, and that person 4 is neither a student nor an employee.

Person

personId	first	last
1	Joe	Smith
2	Janet	Lee
3	Pat	Jones
4	Jack	Lee

In the relationships diagram note the relationships are one-to-one between the supertype relation and each of its subtype relations:

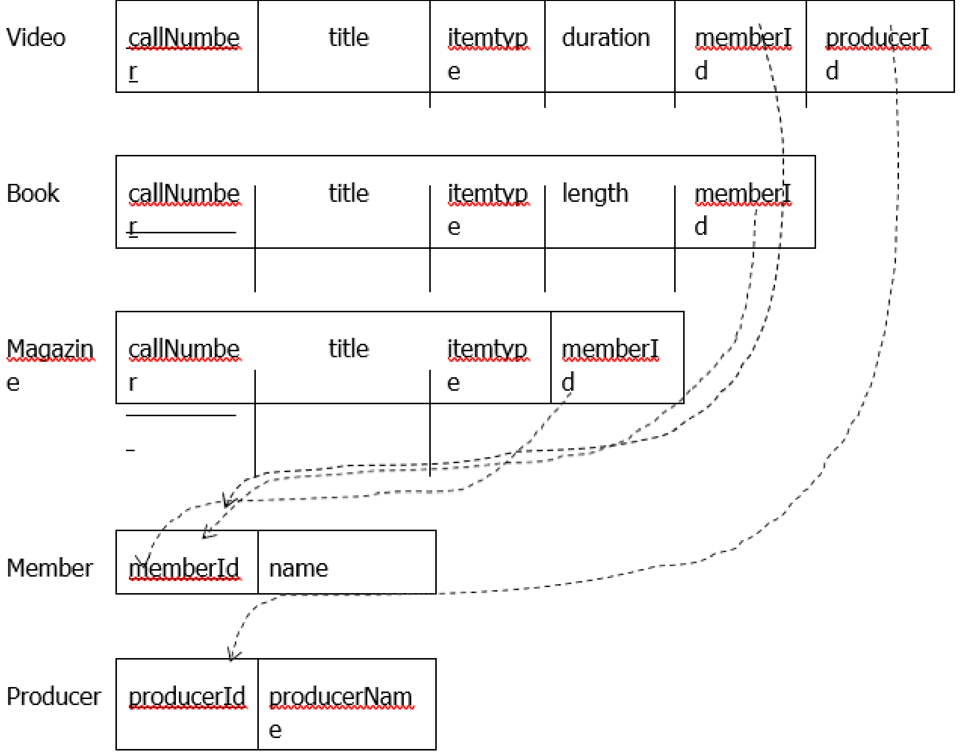


B.4.2: Relations for Bottom-Most Entity Types

In this case relations are created for only entity types that are at the "bottom" of the hierarchy. There are no relations created for a supertype. Important points here are that:

- All relations derived from entity types in the same hierarchy will have the same primary key.
- No primary key value can be repeated (We have not seen how to handle this in MS Access. Further study of relational systems can include techniques that automate the checking for this kind of integrity constraint.)
- Attributes of a supertype must be included in each of its subtype relations.

Example:
For the library model, since there is total participation in subtyping this option works well. Every item will be stored in a relation, and each item is stored exactly once. The resulting design:



Note the foreign keys:

- Because there is no Item relation, each of Video, Book, and Magazine have foreign keys referencing Member.

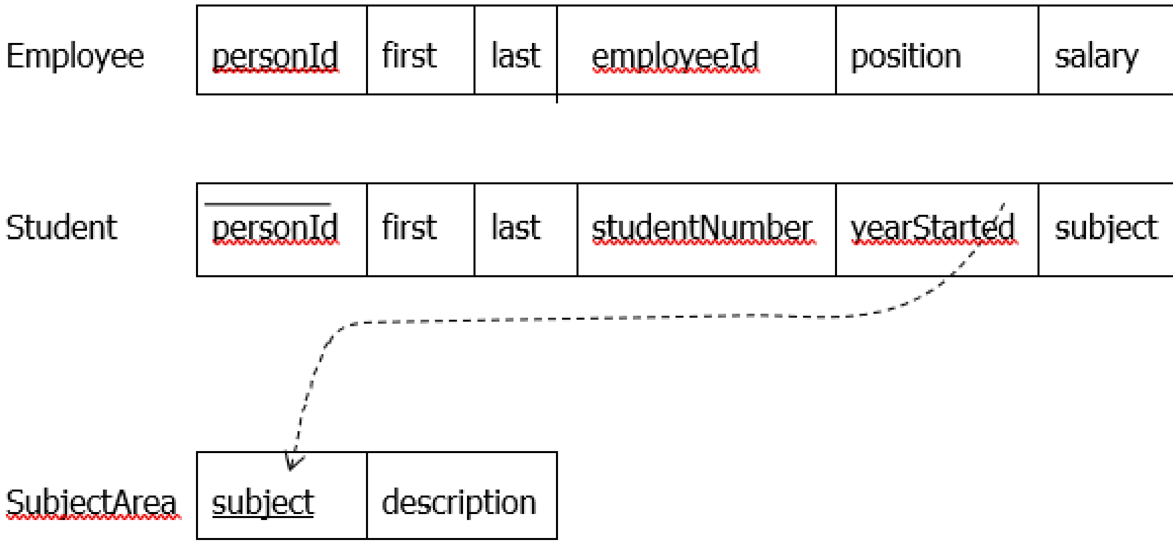
- Video is the only relation with a foreign key referencing Producer.

An issue the designer should be aware of is that callNumbers across the three relations must be unique (call number is the primary key of Item). Further study of database systems is needed to know how this rule can be enforced.

It is left as an exercise for the student to create a database with sample data.

Example:

Consider the university model. This approach (creating relations for bottom- most entity types) is not suitable for the university model because of the overlapping subtypes and because the participation in subtyping is not total. Applying the option, we have:



If an entity exists in more than one subtype, then such an entity will have data stored redundantly in the database. In the design above if a person is both an employee and a student then that person's first and last names would be stored twice (in two different relations).

The Employee and Student relations are not sufficient to store Person data. The participation is optional and so a person may exist who is neither an employee nor a student; in such a case the data for the person cannot be stored!

It is left as an exercise for the student to create a database with sample data.

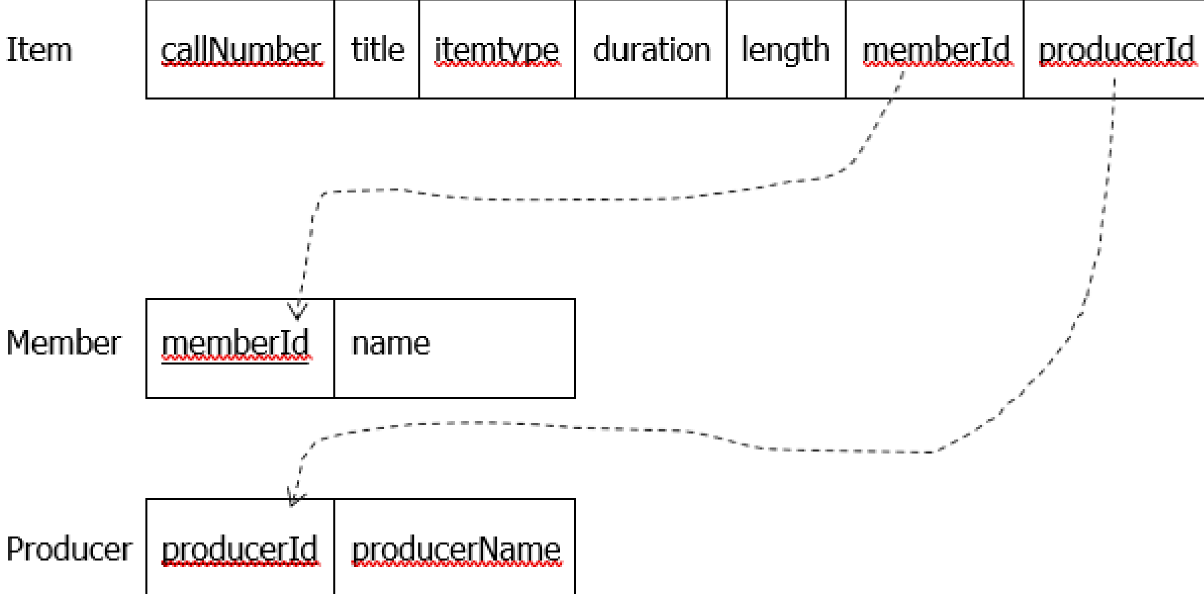
B.4.3: One Relation Representing the Whole Hierarchy

When this option is applied one relation is created for a complete hierarchy. All attributes appearing in the hierarchy are placed in one relation. Note that the value of a discriminator attribute will enable the user to know easily the subtype of a particular entity. For our example models, when we map a hierarchy to a single relation we obtain very simple relational designs.

It is left as an exercise for the student to create databases with sample data.

Example:

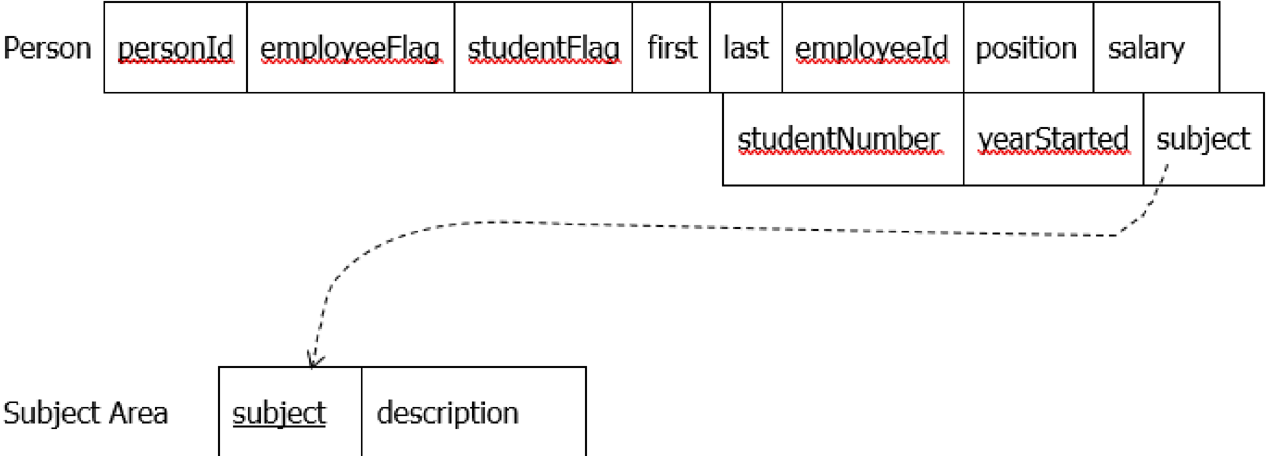
The library model maps to the following design:



In the Item relation the itemType attribute indicates if the row represents a video, a magazine, or a book. The memberId may have a value if the item is out on loan. producerId can only have a value if itemType is "video".

Example:

When mapping a hierarchy to a single relation for the university model the designer should include discriminator attributes that are boolean-valued with one discriminator attribute per subtype. Applying this option to the university model we have:



With this database each person is stored at most once in the database; there is no duplicated data as with the previous mapping option.

If a person is neither an employee nor a student then the only attributes that can have values are: personId, employeeFlag, studentFlag, first and last – the others must be null. The values of employeeFlag and studentFlag would be false.

Exercises

Exercises

1) Consider the database designs illustrated in this appendix. Implement one or more of these and populate with data.

2) Consider the two designs used in the examples of this appendix. Combine these two designs by replacing Member with the Person hierarchy. Illustrate the relational structures when the model is mapped to a database. Choose mapping options for the hierarchies.

3) Consider the design you created in exercise 2 but modify the one-to-many borrows relationship to be a many-to-many with attributes dateBorrowed and dateReturned where dateBorrowed is a discriminator for the relationship. Recall this discriminator is not the same as the discriminators suggested for mapping supertypes and subtypes.

- Note that this modification to the library example will allow history to be recorded for the borrowing of items.

4) For exercise 3 create the database and populate the database with sample data.

5) Create an ERD for a service station business that provides goods and services to its customers. Typically, a customer comes in with their vehicle and requests certain work to be performed. For example, a customer may request an oil change and for a new set of four tires to be provided and installed.

- The work items that can be performed or supplied can be of two types: a service (such as the oil change) and actual physical items (such as litres of oil). There will be several services that can be performed such as tire installation, changing oil, or fixing a flat tire.
- Each of these will have some cost to be charged to a customer. There are many concrete items that are supplied and charged to a customer such as fan belts, litres of oil, or tires – these are things that are kept in inventory. Consider creating a hierarchy for products (goods / services); make up reasonable attributes.
- This service station has customers that fall into two groups: some are private individuals and others are businesses. Individuals will have a first name, last name, address and phone number. A business will have a business name, address, phone number and a contact person who has a first name and last name. Consider creating a hierarchy for customers.
- The service station needs to keep track of all the goods and services it provides to its customers so that it has a historical record and knows what it has charged to each customer. Each visit to the service station by a customer will generate a work order that keeps track of the work that was done for the customer's vehicle. Vehicles have license plate numbers, and other attributes to describe them (make, model, colour, …). For each visit of a customer to the station the system needs to know the date the visit occurred, the details of the work performed and goods provided, and the total charge to the customer.