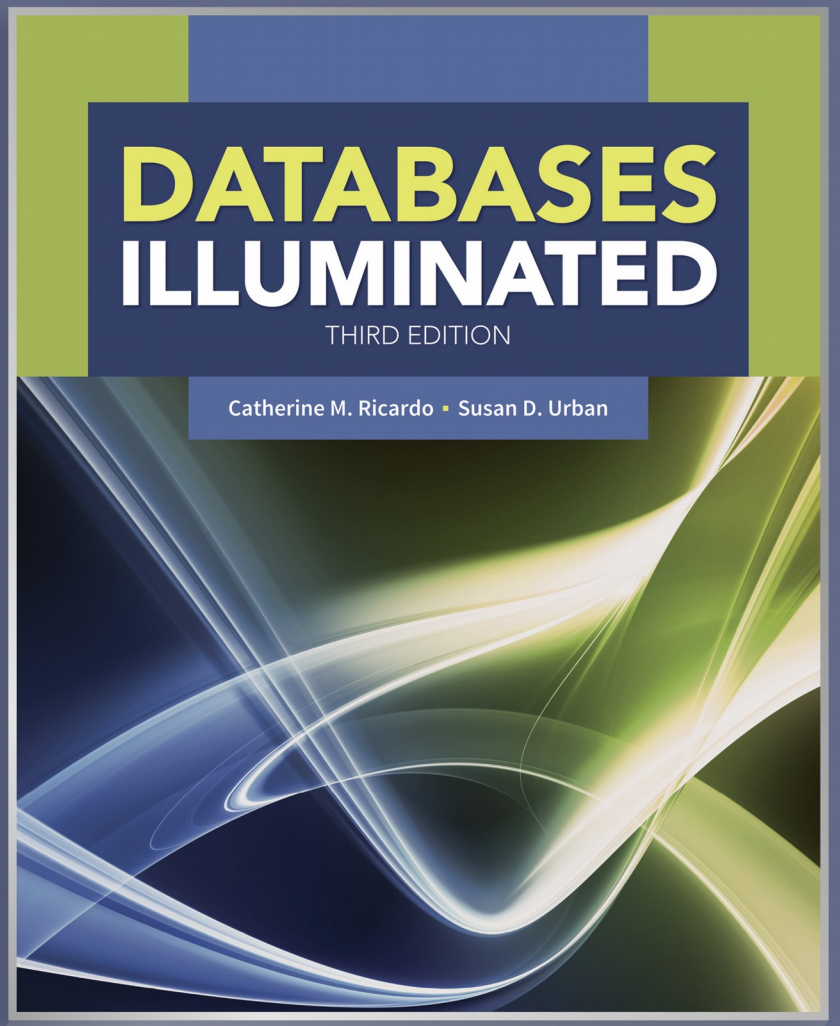
Databases Illuminated

Chapter 4
The Relational Model



Advantages of Relational Model

- Provides data independence-separates logical from physical level
- Based on mathematical notion of relation
- Can use power of mathematical abstraction
- Can develop body of results using theorem and proof method of mathematics–apply to many different applications
- Can use expressive, exact mathematical notation
- Theory provides tools for improving design
- Basic structure is simple, easy to understand
- Data operations are easy to express, using a few powerful commands
- Operations do not require user to know storage structures used

Data Structures

- Relations are represented physically as tables
- Tables are related to one another
- Table holds information about entities (objects)
- Rows (tuples) correspond to individual records
- Columns correspond to attributes
- A column contains values from one domain
- Domains consist of atomic values

Properties of Tables

- Each cell contains at most one value
- Each column has a distinct name, the name of the attribute it represents
- Values in a column all come from the same domain
- Each tuple is distinct – no duplicate tuples
- Order of tuples is immaterial

Example of Relational Model

- `Student` table tells facts about students
- `Faculty` table shows facts about faculty
- `Class` table shows facts about classes, including what faculty member teaches each class
- `Enroll` table relates students to classes

stuId	lastName	firstName	major	credits
S1001	Smith	Tom	History	90
S1002	Chin	Ann	Math	36
S1005	Lee	Perry	History	3
S1010	Burns	Edward	Art	63
S1013	McCarthy	Owen	Math	0
S1015	Jones	Mary	Math	42
S1020	Rivera	Jane	CSC	15

Figure 4.1A The `Student` Table

facId	name	department	rank
F101	Adams	Art	Professor
F105	Tanaka	CSC	Instructor
F110	Byrne	Math	Assistant
F115	Smith	History	Associate
F221	Smith	CSC	Professor

Figure 4.1B The `Faculty` Table

classNumber	facId	schedule	room
ART103A	F101	MWF9	H221
CSC201A	F105	TuThF10	M110
CSC203A	F105	MThF12	M110
HST205A	F115	MWF11	H221
MTH101B	F110	MTuTh9	H225
MTH103C	F110	MWF11	H225

Figure 4.1C The `Class` Table

stuId	classNumber	grade
S1001	ART103A	A
S1001	HST205A	C
S1002	ART103A	D
S1002	CSC201A	F
S1002	MTH103C	B
S1010	ART103A	
S1010	MTH103C	
S1020	CSC201A	B
S1020	MTH101B	A

Figure 4.1D The `Enroll` Table

Mathematical Relations

- For two sets D_1 and D_2, the **Cartesian product**, $D_1 \times D_2$, is the set of all ordered pairs where the first element is from D_1 and the second from D_2
- A **relation** is any subset of the Cartesian product
- Could form Cartesian product of 3 sets; relation is any subset of the ordered triples so formed
- Could extend to n sets, using n-tuples

Database Relations

- A **relation schema**, **R**, is a set of attributes A_1, A_2,…,A_n with their domains D_1, D_2,…D_n
- A **relation r** on relation schema **R** is a set of mappings from the attributes to their domains
- **r** is a set of n-tuples (A_1:d_1, A_2:d_2, …, A_n:d_n) such that $d_1 \epsilon D_1$, $d_2 \epsilon D_2$, …, $d_n \epsilon D_n$
- In a table to represent the relation, list the A_i as column headings, and let the (d_1, d_2, …d_n) become the n-tuples, the rows of the table

Properties of Relations

- **Degree**: the number of attributes (columns)
  - 2 attributes - binary; 3 attributes - ternary; n attributes - n-ary
  - A property of the intension – does not change unless database design changes
- **Cardinality**: the number of tuples (rows)
  - Changes as tuples are added or deleted
  - A property of the extension – changes often
- **Keys**
- **Integrity constraints**

Relation Keys

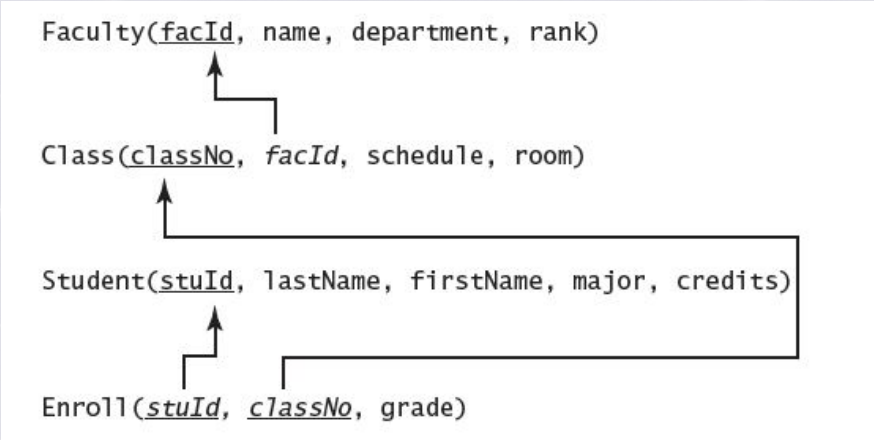
- Relations never have duplicate tuples, you can always tell tuples apart; implies there is always a key (which may be a composite of all attributes, in worst case)
- **Superkey**: set of attributes that uniquely identifies tuples
- **Candidate key**: superkey such that no proper subset of itself is also a superkey (i.e. it has no unnecessary attributes)
- **Primary key**: candidate key chosen for unique identification of tuples
- Cannot verify a key by looking at an instance; need to consider semantic information to ensure uniqueness
- A **foreign key** is an attribute or combination of attributes that is the primary key of some relation (called its **home** relation)

Integrity Constraints

- **Integrity:** correctness and internal consistency
- **Integrity constraints**-rules or restrictions that apply to all instances of the database
- Enforcing integrity constraints ensures only legal states of the database are created
- Types of constraints
  - **Domain constraint** - limits set of values for attribute
  - **Entity integrity**: no attribute of a primary key can have a null value
  - **Referential integrity**: each foreign key value must match the primary key value of some tuple in its home relation or be completely null
  - **General constraints** or **business rules**: may be expressed as table constraints or assertions

Representing Relational Database Schemas

- Can have any number of relation schemas
- For each schema, list name of relation followed by list of attributes in parentheses
- Underline primary key in each relation schema
- Indicate foreign keys (We use italics – arrows are best)
- Database schema includes domains, views, character sets, constraints, stored procedures, authorizations, etc.
- Example: `University` database schema



Types of Relational Data Manipulation Languages

- **Procedural**: proscriptive - user tells system how to manipulate data - e.g. relational algebra
- **Non-procedural**: declarative - user tells what data is needed, not how to get it - e.g. relational calculus, SQL
- Other types:
  - **Graphical**: user provides illustration of data needed e.g. Query By Example(QBE)
  - **Fourth-generation**: 4GL uses user-friendly environment to generate custom applications
  - **Natural language**: 5GL accepts restricted version of English or other natural language

Relational Algebra

- Theoretical language with operators that apply to one or two relations to produce another relation
- Both operands and results are tables
- Can assign name to resulting table (rename)
- `SELECT`, `PROJECT`, `JOIN` allow many data retrieval operations

SELECT Operation

- Applied to a single table, returns rows that meet a specified predicate, copying them to new table
- Returns a horizontal subset of original table

`SELECT` *`tableName`* `WHERE` *`condition`* `[GIVING` *`newTableName`*`]`

Symbolically, `[`*`newTableName`* `= ]` $\sigma_{\text{predicate}}$ `(`*`table-name`*`)`

- Predicate is called **theta-condition**, as in σ_{θ}`(`*`tablename`*`)`
- Result table is horizontal subset of operand
- Predicate can have operators

`<, <=, >, >=, =, <>,` $\wedge$`(AND),` $\vee$`(OR),` $\neg$`(NOT)`

SELECTT Operation

- **SELECT** Class **WHERE** room = 'H225' **GIVING** Answer

  or symbolically

  Answer = $\sigma_{room='H225'}$ (Class)

- This command produces the following table:

Figure 4.1C The Class Table

classNumber	facId	schedule	room
ART103A	F101	MWF9	H221
CSC201A	F105	TuThF10	M110
CSC203A	F105	MThF12	M110
HST205A	F115	MWF11	H221
MTH101B	F110	MTuTh9	H225
MTH103C	F110	MWF11	H225

Answer:

classNo	facId	schedule	room
MTH101B	F110	MTuTh9	H225
MTH103C	F110	MWF11	H225

PROJECT Operation

- Operates on single table
- Returns unique values in a column or combination of columns

```
PROJECT tableName OVER (colName,...,colName)
[GIVING  newTableName]
```

Symbolically

$$[newTableName\ =]\ \Pi_{colName,...,colName}\ (tableName)$$

- Can compose SELECT and PROJECT, using result of first as argument for second

PROJECT Operation

- **PROJECT** Student OVER major **GIVING** Temp
  or Temp = Π_{major}(Student)
- The resulting table, Temp, looks like this:

Figure 4.1A The Student Table

stuId	lastName	firstName	major	credits
S1001	Smith	Tom	History	90
S1002	Chin	Ann	Math	36
S1005	Lee	Perry	History	3
S1010	Burns	Edward	Art	63
S1013	McCarthy	Owen	Math	0
S1015	Jones	Mary	Math	42
S1020	Rivera	Jane	CSC	15

Temp:

major
History
Math
Art
CSC

PROJECT Operation

- **PROJECT** Class **OVER** (facId,room) or
  $\Pi_{facId,\ room}$ (Class)
- This gives the result in an unnamed table:

Figure 4.1C The `Class` Table

classNumber	facId	schedule	room
ART103A	F101	MWF9	H221
CSC201A	F105	TuThF10	M110
CSC203A	F105	MThF12	M110
HST205A	F115	MWF11	H221
MTH101B	F110	MTuTh9	H225
MTH103C	F110	MWF11	H225

facId	room
F101	H221
F105	M110
F115	H221
F110	H225

PROJECT Operation

- **SELECT** Student **WHERE** major = 'History' **GIVING** Temp
  PROJECT Temp **OVER** (lastName, firstName, stuId)
  GIVING Result or

$$\Pi_{\text{lastName, firstName, stuId}} (\sigma_{\text{major='History'}} (\text{Student}))$$

- After the first command is executed, we have the following table:

Temp:

stuId	lastName	firstName	major	credits
S1001	Smith	Tom	History	90
S1005	Lee	Perry	History	3

- The second command is performed on this temporary table, and the result is:

Result:

lastName	firstName	stuId
Smith	Tom	S1001
Lee	Perry	S1005

Product, A x B

- Binary operation – applies to two tables
- Cartesian product; cross-product of A and B; A TIMES B, written A x B
  - Concatenates all rows of A with all rows of B
  - Columns are the columns of A followed by the columns of B
  - Degree of result is deg of A + deg of B
  - Cardinality of result is (card of A) * (card of B)
- Can be formed by nested loops algorithm

Product, A x B

FIGURE 4.2

Student × Enroll

Student × Enroll

Student .stuId	lastName	firstName	major	credits	Enroll .stuId	classNumber	grade
S1001	Smith	Tom	History	90	S1001	ART103A	A
S1001	Smith	Tom	History	90	S1001	HST205A	C
S1001	Smith	Tom	History	90	S1002	ART103A	D
S1001	Smith	Tom	History	90	S1002	CSC201A	F
S1001	Smith	Tom	History	90	S1002	MTH103C	B
S1001	Smith	Tom	History	90	S1010	ART103A	
S1001	Smith	Tom	History	90	S1010	MTH103C	
S1001	Smith	Tom	History	90	S1020	CSC201A	B
S1001	Smith	Tom	History	90	S1020	MTH101B	A
S1002	Chin	Ann	Math	36	S1001	ART103A	A
...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...
S1020	Rivera	Jane	CSC	15	S1020	MTH101B	A

Theta Join and Equijoin

- Theta join is Product followed by a Select with predicate, theta (Θ)
- $A \ |x|_{\theta} \ B = \sigma_{\theta}(A \times B)$
- Equijoin can be formed when tables have common columns, or columns with same domains
- For equijoin, select rows of the product where the values of the common columns are equal
- If we have more than one common column, both sets of values have to match.

Equijoin

```
Student EQUIJOIN Enroll
```

Note that this is equivalent to

```
Student TIMES Enroll GIVING Temp3
SELECT Temp3 WHERE Student.stuId = Enroll.stuId
```

or

$$\sigma_{\text{Student.stuId=Enroll.stuId}} (\text{Student} \times \text{Enroll})$$

FIGURE 4.3
Student EQUIJOIN Enroll

Student EQUIJOIN Enroll

Student .stuId	lastName	firstName	major	credits	Enroll .stuId	classNumber	grade
S1001	Smith	Tom	History	90	S1001	ART103A	A
S1001	Smith	Tom	History	90	S1001	HST205A	C
S1002	Chin	Ann	Math	36	S1002	ART103A	D
S1002	Chin	Ann	Math	36	S1002	CSC201A	F
S1002	Chin	Ann	Math	36	S1002	MTH103C	B
S1010	Burns	Edward	Art	63	S1010	ART103A	
S1010	Burns	Edward	Art	63	S1010	MTH103C	
S1020	Rivera	Jane	CSC	15	S1020	CSC201A	B
S1020	Rivera	Jane	CSC	15	S1020	MTH101B	A

NATURAL JOIN

- Same as `Equijoin`, but drop repeated column(s)
- symbolized by `|x|` as in

  `[`*`newTableName`* `= ] Student |x| Enroll`

  or

  `Student JOIN Enroll [GIVING` *`newTableName`*`]`

NATURAL JOIN

Figure 4.1B The `Faculty` Table

facId	name	department	rank
F101	Adams	Art	Professor
F105	Tanaka	CSC	Instructor
F110	Byrne	Math	Assistant
F115	Smith	History	Associate
F221	Smith	CSC	Professor

Figure 4.1C The `Class` Table

classNumber	facId	schedule	room
ART103A	F101	MWF9	H221
CSC201A	F105	TuThF10	M110
CSC203A	F105	MThF12	M110
HST205A	F115	MWF11	H221
MTH101B	F110	MTuTh9	H225
MTH103C	F110	MWF11	H225

FIGURE 4.4

Faculty |x| Class

Faculty NATURAL JOIN Class

FacId	name	department	rank	classNo	schedule	room
F101	Adams	Art	Professor	ART103A	MWF9	H221
F105	Tanaka	CSC	Instructor	CSC203A	MThF12	M110
F105	Tanaka	CSC	Instructor	CSC201A	TuThF10	M110
F110	Byrne	Math	Assistant	MTH103C	MWF11	H225
F110	Byrne	Math	Assistant	MTH101B	MTuTh9	H225
F115	Smith	History	Associate	HST205A	MWF11	H221

Semi Join

- `Left-semijoin A |x B`
  - take the `natural join` of `A` and `B` and then `project` the result onto the attributes of `A`
  - result is just those tuples of `A` that participate in the `natural join`
- `Right-semijoin A x| B`
  - `Projection` onto `B` of the `natural join A|x|B`

Semi Join

Figure 4.1A The `Student` Table

stuId	lastName	firstName	major	credits
S1001	Smith	Tom	History	90
S1002	Chin	Ann	Math	36
S1005	Lee	Perry	History	3
S1010	Burns	Edward	Art	63
S1013	McCarthy	Owen	Math	0
S1015	Jones	Mary	Math	42
S1020	Rivera	Jane	CSC	15

Figure 4.1D The `Enroll` Table

stuId	classNumber	grade
S1001	ART103A	A
S1001	HST205A	C
S1002	ART103A	D
S1002	CSC201A	F
S1002	MTH103C	B
S1010	ART103A	
S1010	MTH103C	
S1020	CSC201A	B
S1020	MTH101B	A

FIGURE 4.5(A)

`Student |x Enroll`

`Student LEFT SEMIJOIN Enroll`

StuId	lastName	firstName	major	credits
S101	Smith	Tom	History	90
S1002	Chin	Ann	Math	36
S1010	Burns	Edward	Art	63
S1020	Rivera	Jane	CSC	15

Outer Join

- **Left outer join of A and B**
  - Form `natural join`, but add rows for all the tuples of `A` with no matches in `B`
  - Fill in the `B` attributes for those unmatched tuples with null values
- **Right outer join of A and B**
  - Add unmatched `B` tuples to `natural join`, filling in null values for the `A` attributes
- **Full outer join of A and B**

  Add unmatched tuples for both `A` and `B`, filling in null values for unmatched tuples on both sides

Outer Join

Figure 4.1A The Student Table

stuId	lastName	firstName	major	credits
S1001	Smith	Tom	History	90
S1002	Chin	Ann	Math	36
S1005	Lee	Perry	History	3
S1010	Burns	Edward	Art	63
S1013	McCarthy	Owen	Math	0
S1015	Jones	Mary	Math	42
S1020	Rivera	Jane	CSC	15

Figure 4.1D The Enroll Table

stuId	classNumber	grade
S1001	ART103A	A
S1001	HST205A	C
S1002	ART103A	D
S1002	CSC201A	F
S1002	MTH103C	B
S1010	ART103A	
S1010	MTH103C	
S1020	CSC201A	B
S1020	MTH101B	A

FIGURE 4.5(B)

Student LEFT OUTER JOIN Enroll

stuId	lastName	firstName	major	credits	classNumber	grade
S1001	Smith	Tom	History	90	HST205A	C
S1001	Smith	Tom	History	90	ART103A	A
S1002	Chin	Ann	Math	36	MTH103C	B
S1002	Chin	Ann	Math	36	CSC201A	F
S1002	Chin	Ann	Math	36	ART103A	D
S1010	Burns	Edward	Art	63	MTH103C	
S1010	Burns	Edward	Art	63	MTH103C	A
S1020	Rivera	Jane	CSC	15	MTH101B	A
S1020	Rivera	Jane	CSC	15	CSC201A	B
S1005	Lee	Perry	History	3		
S1013	McCarthy	Owen	Math	0		
S1015	Jones	Mary	Math	42		

Set Operations

- Tables must be **union compatible** – have same schema
- `A` **`UNION`** `B`: set of tuples in either or both of `A` and `B`, written `A ∪ B`
- `A` **`INTERSECTION`** `B`: set of tuples in both `A` and `B` simultaneously, written `A ∩ B`
- `Difference`, or `A` **`MINUS`** `B`: set of tuples in `A` but not in `B`, written `A - B`

Set Operations

FIGURE 4.6(A)
Union-Compatible Relations MainFac and BranchFac

MainFac

facId	name	department	rank
F101	Adams	Art	Professor
F105	Tanaka	CSC	Instructor
F221	Smith	CSC	Professor

BranchFac

facId	name	department	rank
F101	Adams	Art	Professor
F110	Byrne	Math	Assistant
F115	Smith	History	Associate
F221	Smith	CSC	Professor

FIGURE 4.6(B)
MainFac UNION BranchFac

MainFac **UNION** BranchFac

facId	name	department	rank
F101	Adams	Art	Professor
F105	Tanaka	CSC	Instructor
F110	Byrne	Math	Assistant
F115	Smith	History	Associate
F221	Smith	CSC	Professor

FIGURE 4.6(C)
MainFac INTERSECTION BranchFac

MainFac **INTERSECTION** BranchFac

facId	name	department	rank
F101	Adams	Art	Professor
F221	Smith	CSC	Professor

FIGURE 4.6(D)
MainFac MINUS BranchFac

MainFac MINUS BranchFac

facId	name	department	rank
F105	Tanaka	CSC	Instructor

Views

- External models in 3-level architecture are called **external views**
- **Relational** views are slightly different
- A relational view is constructed from existing (base) tables
- Can be a window into a base table (subset)
- Can contain data from more than one table
- Can contain calculated data
- Can hide portions of database from users
- External model may have views and base tables

Mapping ER Diagrams to Relational Schemas

- Each strong entity set becomes a table
- Non-composite, single-valued attributes become attributes of table
- Composite attributes: either make the composite a single attribute or use individual attributes for components, ignoring the composite
- Multi-valued attributes: remove them to a new table along with the primary key of the original table; also keep key in original table
- Weak entity sets become tables-add primary key of owner entity
- Binary Relationships:
  - 1:M-place primary key of 1 side in table of M side as foreign key
  - 1:1- use either key as foreign key in the other table
  - M:M-create a relationship table with primary keys of related entities, along with any relationship attributes
- Ternary or higher degree relationships: construct relationship table of keys, along with any relationship attributes
- Recursive relationship-use foreign key if 1:M; use relationship table if M:M

ER to Relational Schema: University

Figure 3.13

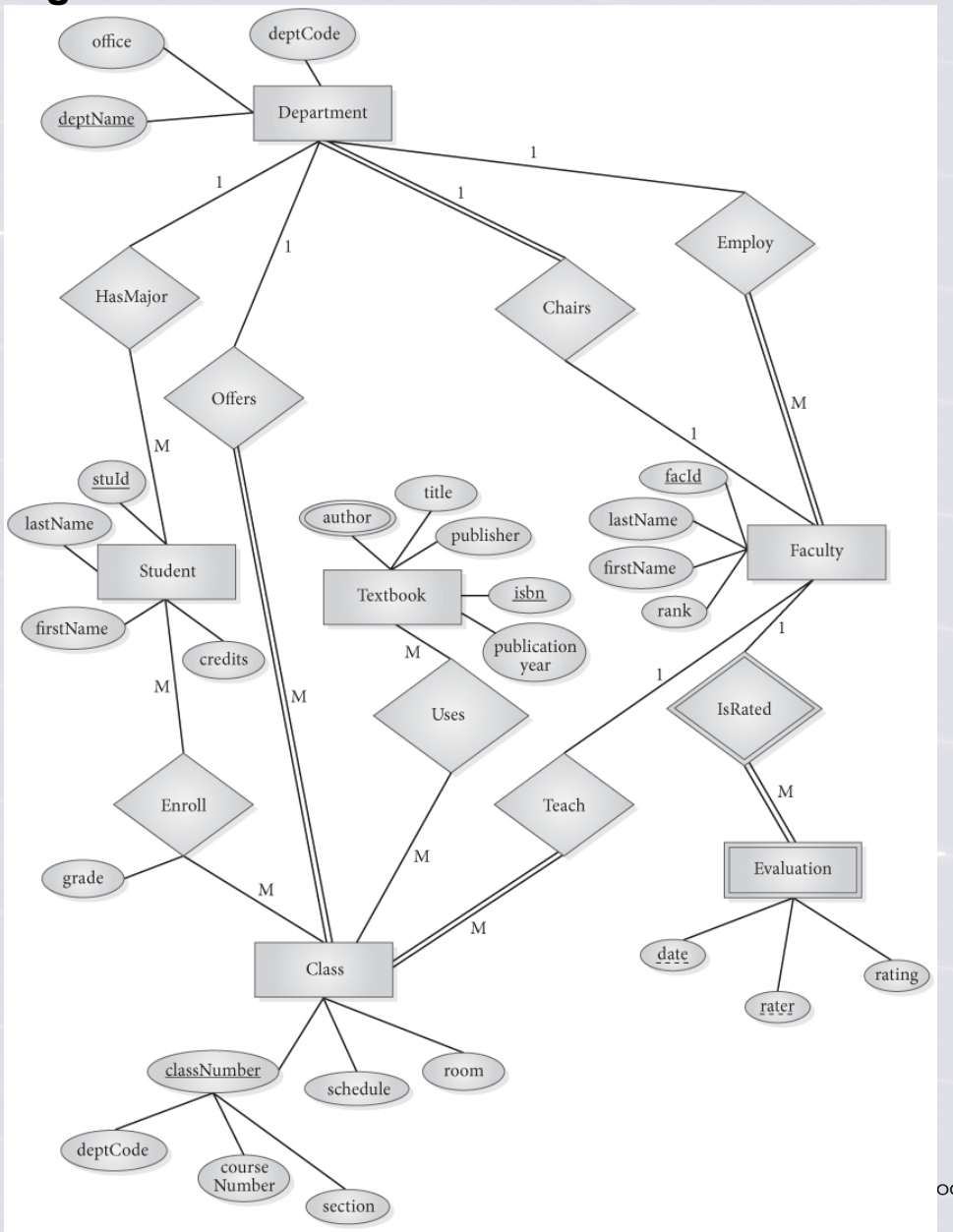
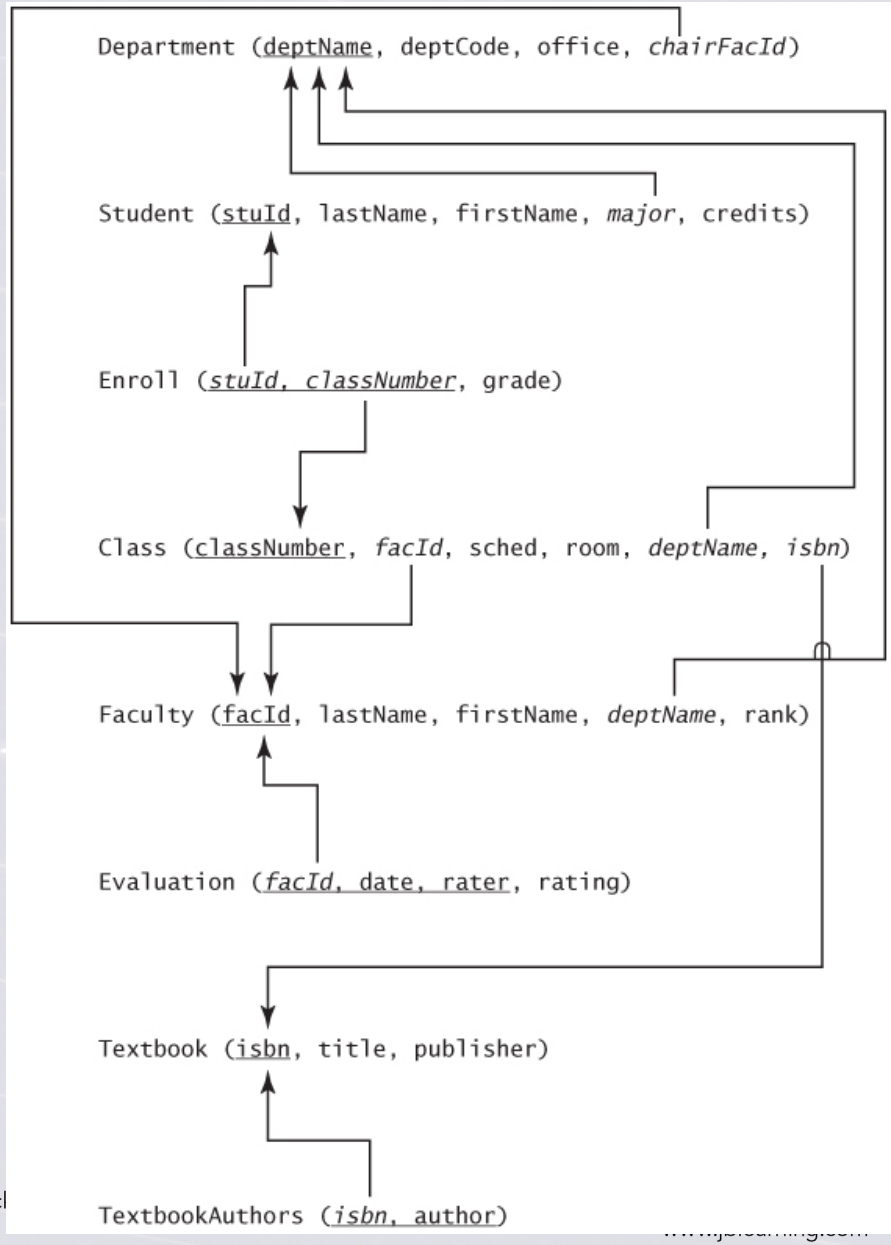


Figure 4.7



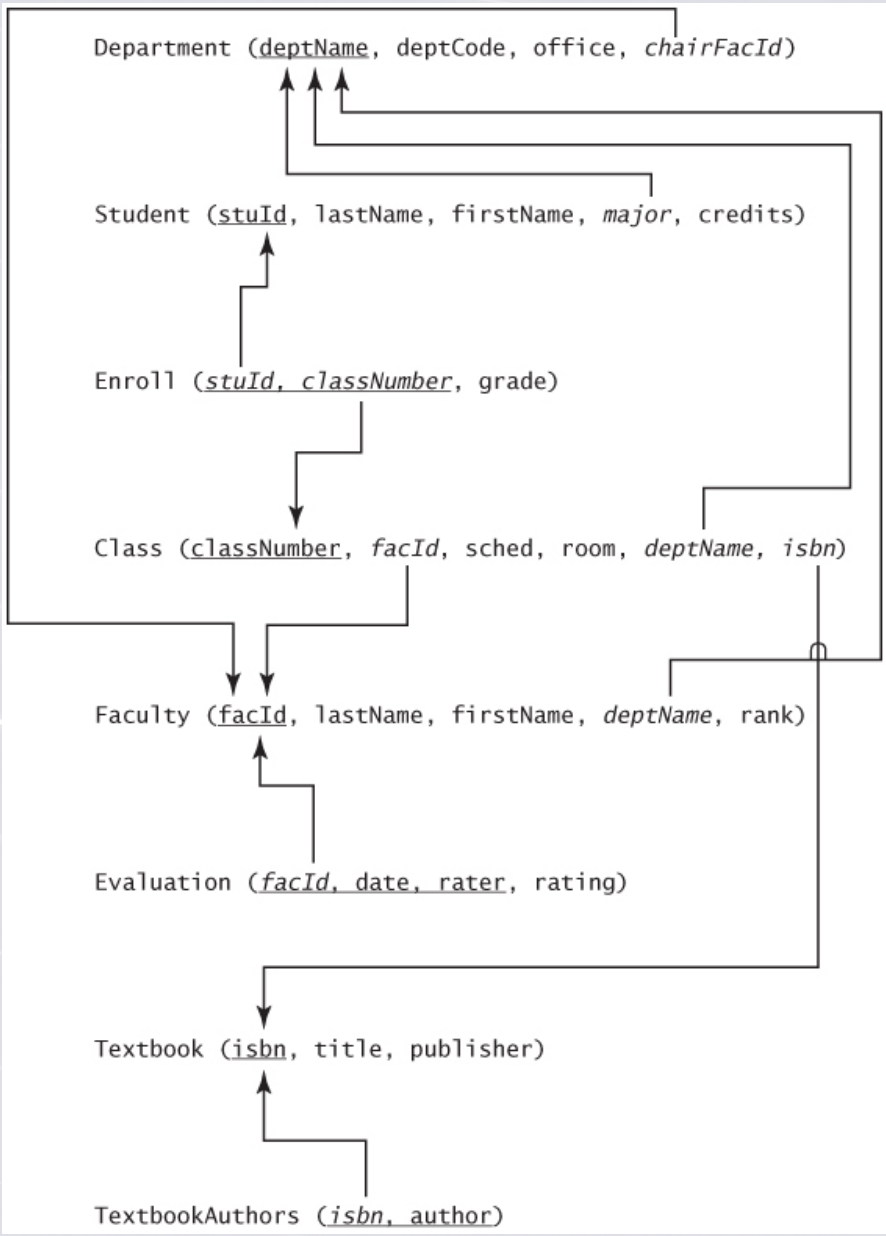
University Example

- The strong entity sets shown as rectangles become relations represented by base tables.
- The table name is the same as the entity name written inside the rectangle.
- For strong entity sets, non-composite, single-valued attributes, represented by simple ovals, become attributes of the relation, or column headings of the table.
- We assume students have only one major, which we will add as an attribute.
- The strong entity sets, Department, Student, and Faculty can be represented by the following tables:

Department(deptName, deptCode, office)
Student(stuId,lastName, firstName, major,

credits)
Faculty(facId, lastName, firstName, rank)

Figure 4.7



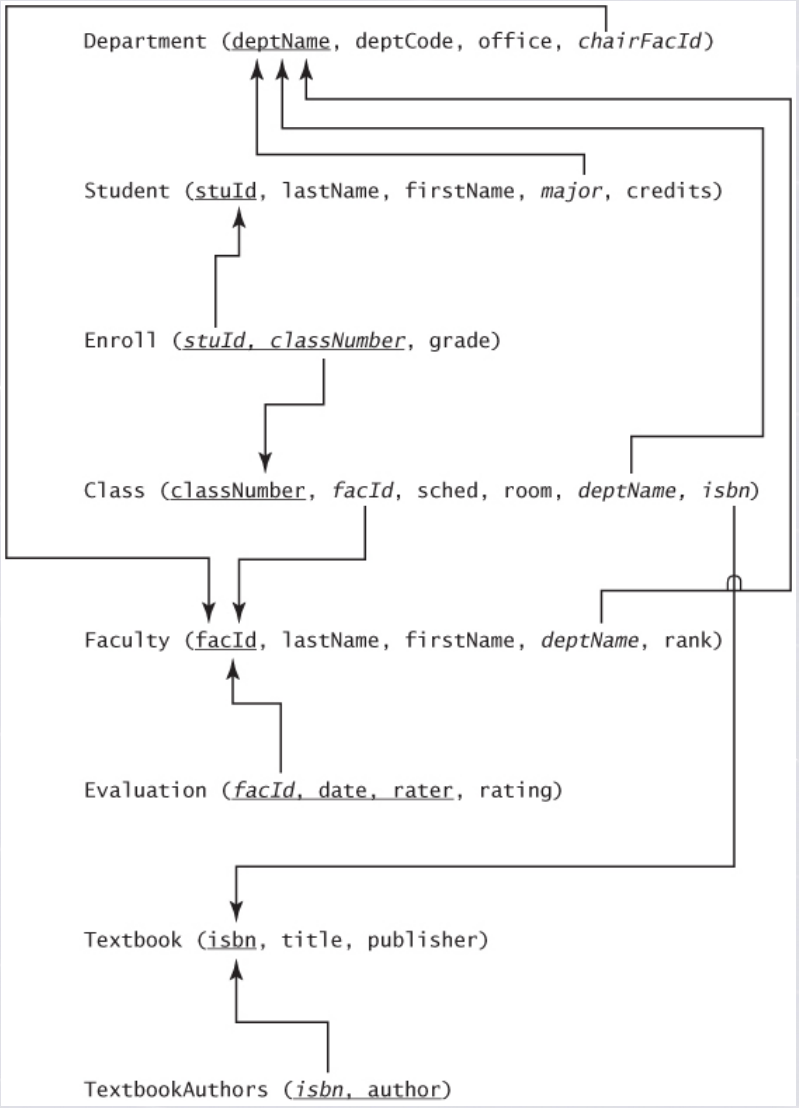
- For E-R attributes that are composites, the relational model does not directly allow us to represent the fact that the attribute is a composite.
- Instead, we can simply make a column for each of the simple attributes that form the composite, or we can choose to leave the composite as a single attribute.
- For example, if we had the composite address, we would not have a column with that name, but instead would have individual columns for its components: street, city, state, and zip.
- For example, if we included an address in the Student table, we could use the schema:

  Student1 (stuId, lastName, firstName, street, city, state, zip, major, credits)

  or, keeping address as a single attribute:

  Student2 (stuId, lastName, firstName, address, major, credits)
- The second choice would make it more difficult to select records on the basis of the value of parts of the address, such as zip code or state.
- In Figure 3.13, we saw that classNumber is actually a composite consisting of deptCode, courseNumber and section.

Figure 4.7

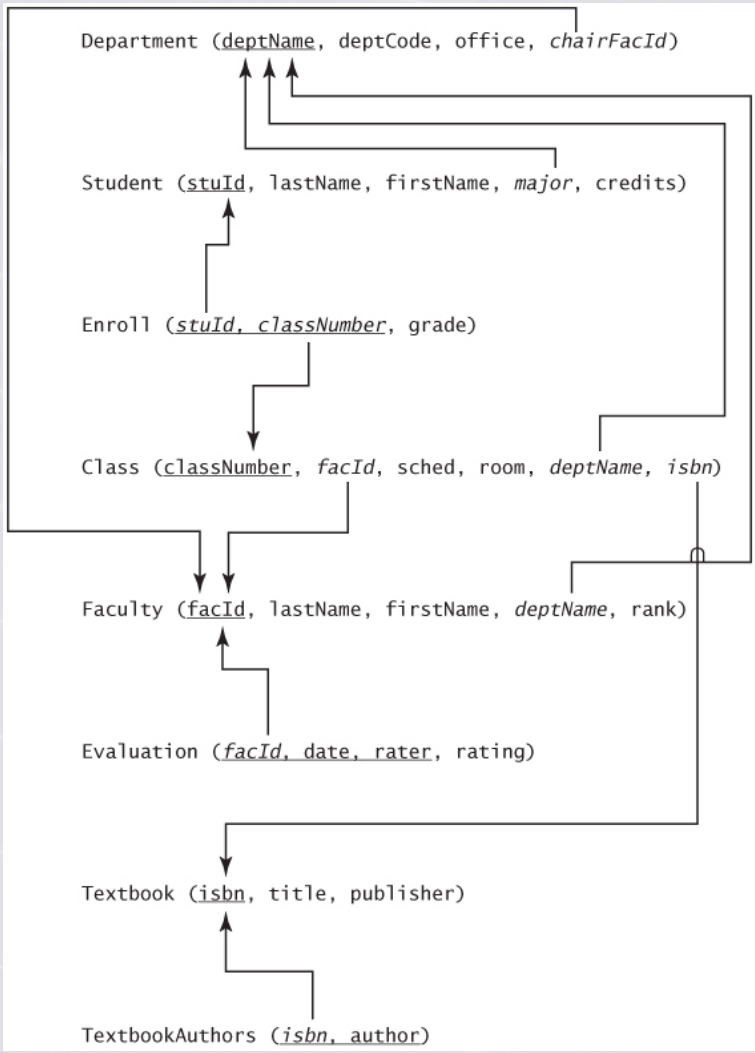


- Here, we are choosing to leave that composite as a single attribute.
- Therefore, for the present we form the Class table as **Class (classNumber, schedule, room)**

- Multivalued attributes pose a special problem when converting to a pure relational model, which does not allow multiple values in a cell.
- The usual solution is to remove them from the table and to create a separate relation in which we put the primary key of the entity along with the multivalued attribute.
- The key of this new table is the combination of the key of the original table and the multivalued attribute.
- The key of the original table becomes a foreign key in the new table.
- If there are multiple multivalued attributes in the original table, we have to create a new table for each one.
- These new tables are treated as if they are weak entities, with the original table (without the multivalued attributes) acting as the owner entity.
- **It is convenient to name the new tables using the plural form of the multivalued attribute name.**
- As an illustration of how to handle multivalued attributes, suppose students could have more than one major.
- To represent this, we would change the Student table by removing major from it, and create instead two tables:

```
Student3(stuId, lastName, firstName, credits)
                    ↑
StuMajors(stuId, major)
```

Figure 4.7



- Another solution is to put in additional columns for the multiple values in the original table.
- Using this solution, if students could have at most two majors, the Student table would become
  Student4(stuId, lastName, firstName, major1, major2, credits)
- However, we are assuming that students have at most one major, so we will retain the original Student table instead.
- If telephone numbers were included for faculty, and faculty members could have more than one telephone number stored in the database, the multiple column solution would produce the schema
  Faculty2 (facId, lastName, firstName, rank, phone, alternatePhone)
- Alternatively, we could omit the telephone number from the Faculty table and add a second table for Faculty:
  FacultyPhones(facId, phoneNo)
- In Figure 3.13, we saw that the attribute author for Textbook can have multiple values. We will choose the first method, representing the auth[illegible]

Textbook(isbn, title, publisher)

TextAuthors(isbn, author)

Figure 4.7

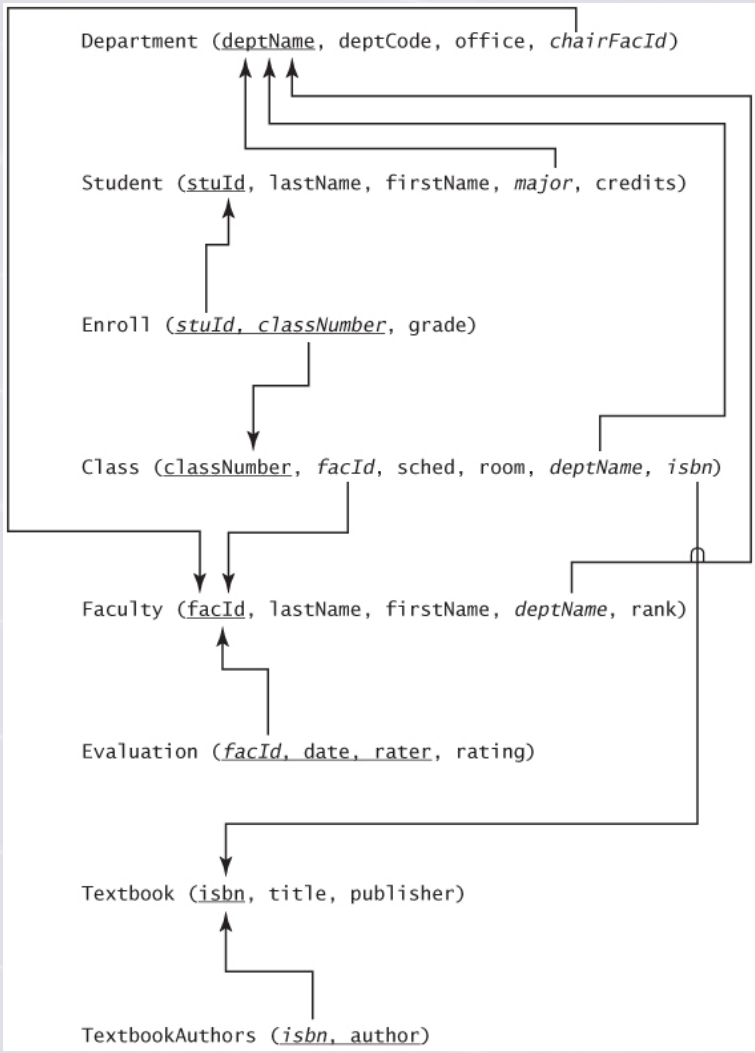
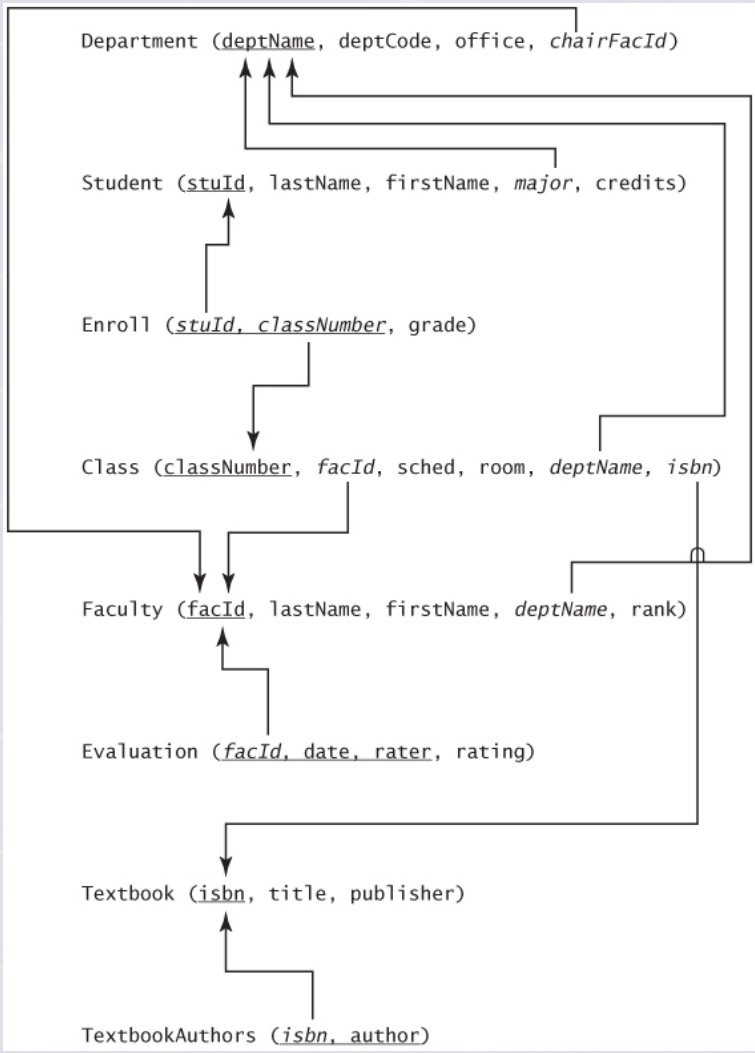


Figure 4.7

- **Weak entity sets are also represented by tables, but they require additional attributes.**
- Recall that a weak entity is dependent on another (owner) entity and has no candidate key consisting of just its own attributes.
- Therefore, the primary key of the corresponding owner entity is used to show which instance of the owner entity a weak entity instance depends on.
- To represent the weak entity, we use a table whose attributes include all the attributes of the weak entity, plus the primary key of the owner entity.
- The weak entity itself may have a **discriminant**—some attribute or attribute set that, when coupled with the owner's primary key, enables us to tell instances apart.
- We use the combination of the owner's primary key and the discriminant as the key.
- In Figure 3.13, the weak entity set, Evaluation, is given the primary key of its owner entity set, Faculty, resulting in the table:

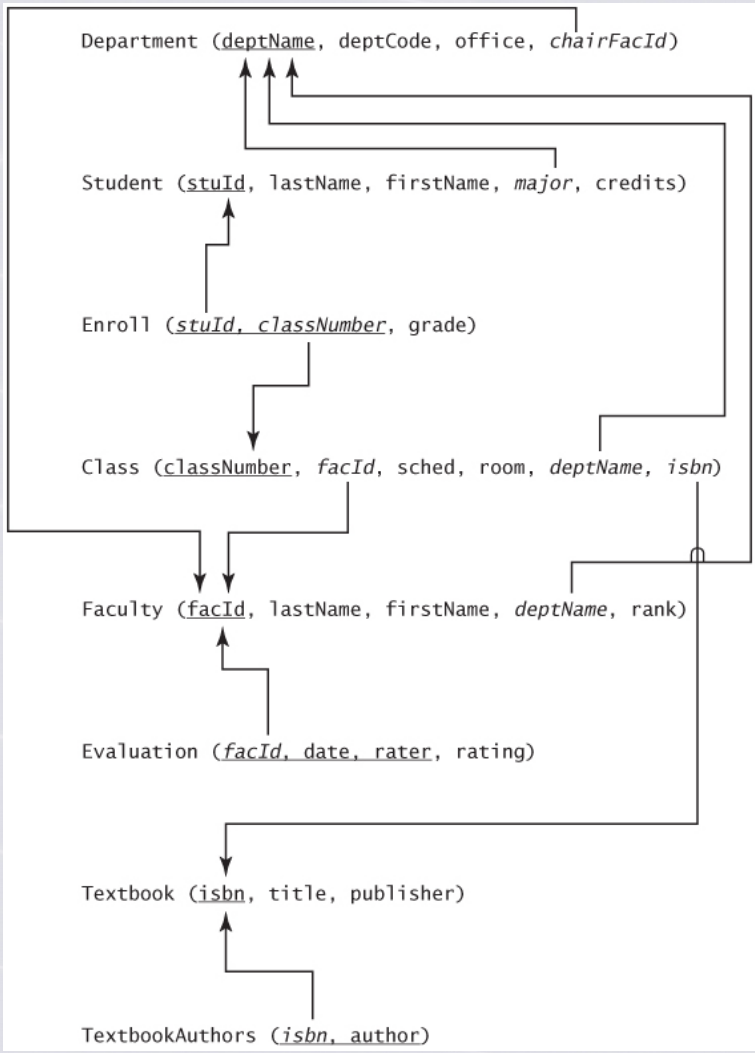
  Evaluation(*facId*, date, rater, rating)



- **Relationship sets must be represented in the tables as well, either by foreign keys or by a separate relationship table, depending on the cardinality of the relationship in the E-R model.**
- If A and B are strong entity sets and the binary relationship A:B is **one-to-many**, we place the **key of A** (the one side) in the **table for B** (the many side), where it becomes a **foreign key**.
- Teach is a one-to-many relationship set connecting the strong entity sets Class and Faculty.
- We use a foreign key to represent this relationship, by placing facId, the key of the "one" side, Faculty, in the table for the "many" side, Class.
- We therefore change the Class table to

  Class (classNo, schedule, room, *facId*)
- Notice that in the resulting base table, each Class row will have one facId value that tells us who the teacher is.
- If we had tried to represent the relationship the opposite way, by placing the classNumber in the Faculty table, it would not work.
- Since a faculty member may teach many courses, a row in the Faculty table would need several values in the classNumber cell, which is not allowed.

Figure 4.7



- All strong entity sets that have a one-to-one relationship should be examined carefully to determine whether they are actually the same entity.
- If they are, they should be combined into a single entity.
- **If A and B are truly separate entities having a one-to-one relationship, then we can put the key of either relation in the other table to show the connection (i.e., either put the key of A in the B table, or vice versa, but not both).**
- For example, if students can get a parking permit to park one car on campus, we might add a Car entity, which would have a one-to-one relationship with Student.
- Each car belongs to exactly one student and each student can have at most one car.
- The initial schema for Car might be

  Car(licNo, make, model, year, color)
- We assume licNo includes the state where the car is licensed.
- To store the one-to-one relationship with Student, we could put stuId in the Car table, getting:

  Car(licNo, make, model, year, color, *stuId*)
- Alternatively, we could add licNo to the Student table—but not both.

Figure 4.7

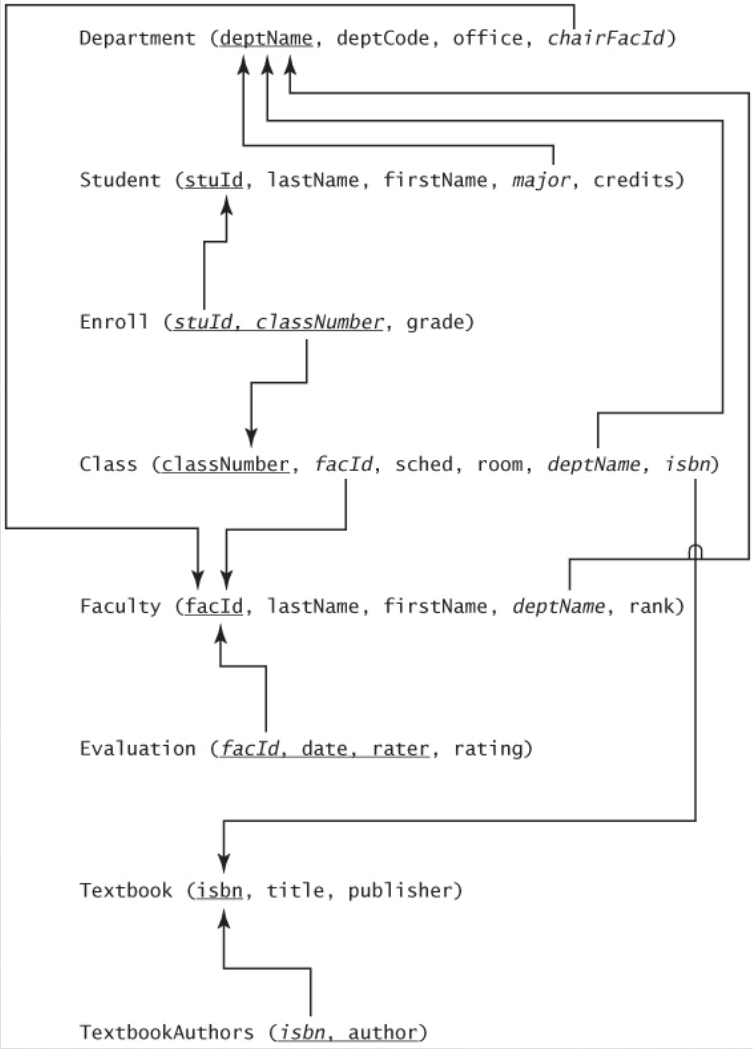
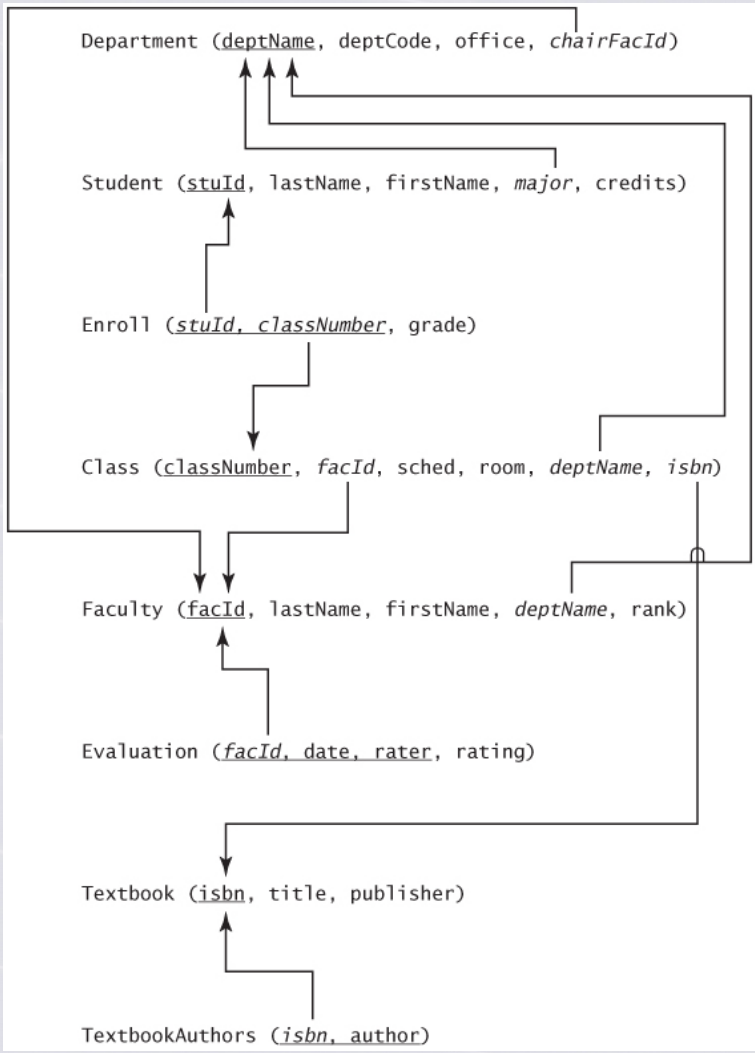


Figure 4.7

- **For a binary relationship, the only time that we cannot use a foreign key to represent the relationship set is the many-to-many case.**
- Here, the connection must be shown by a separate relationship table.
- The Enroll relationship is a many-to-many relationship.
- The table that represents it must contain the primary keys of the associated Student and Class entities, stuId and classNo respectively.
- Since the relationship also has a descriptive attribute, grade, we include that as well.
- The relationship table is

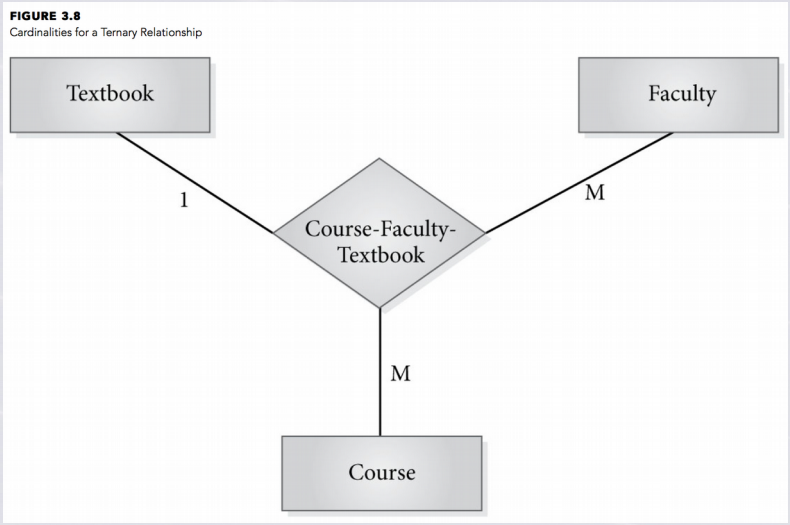
  Enroll(*stuId, classNo*, grade)
- Note that we need a relationship table even if there are no such descriptive attributes for the M:M relationship.
- For such a relationship, the table consists entirely of the two keys of the related entities.



- When we have a ternary or *n*-ary relationship involving three or more entity sets, we construct a table for the relationship, in which we place the primary keys of the related entities.
- If the ternary or *n*-ary relationship has a descriptive attribute, it goes in the relationship table. Figure 3.8 showed a ternary relationship, Course-Faculty-Textbook.
- Recall that this relationship represented courses, not classes.
- It showed which textbook was used whenever a particular faculty member taught a course.
- We would represent it by the table:

  Course-Faculty-Textbook(*facId, courseNo,* isbn)
- We choose the combination of courseNo and facId as the key because we assume that for each combination of courseNo and facId values, there is only one isbn.
- **So for a ternary or higher-order relationship table, we examine the cardinalities involved and choose as the key the attribute or combination that guarantees unique values for rows of the table.**

Figure 3.8



- When we have a recursive relationship, its representation depends on the cardinality.
- **If the cardinality is many-to-many, we must create a relationship table.**
- **If it is one-to-one or one-to-many, the foreign key mechanism can be used.**
- For example, if we wanted to represent a Roommate relationship, and we assume each student can have exactly one roommate, we could represent the one-to-one recursive relationship by adding an attribute to the Student table, as in

```
Student(stuId, lastName, firstName, major, credits,
          ↑
roommateStuId)
```

Figure 4.7

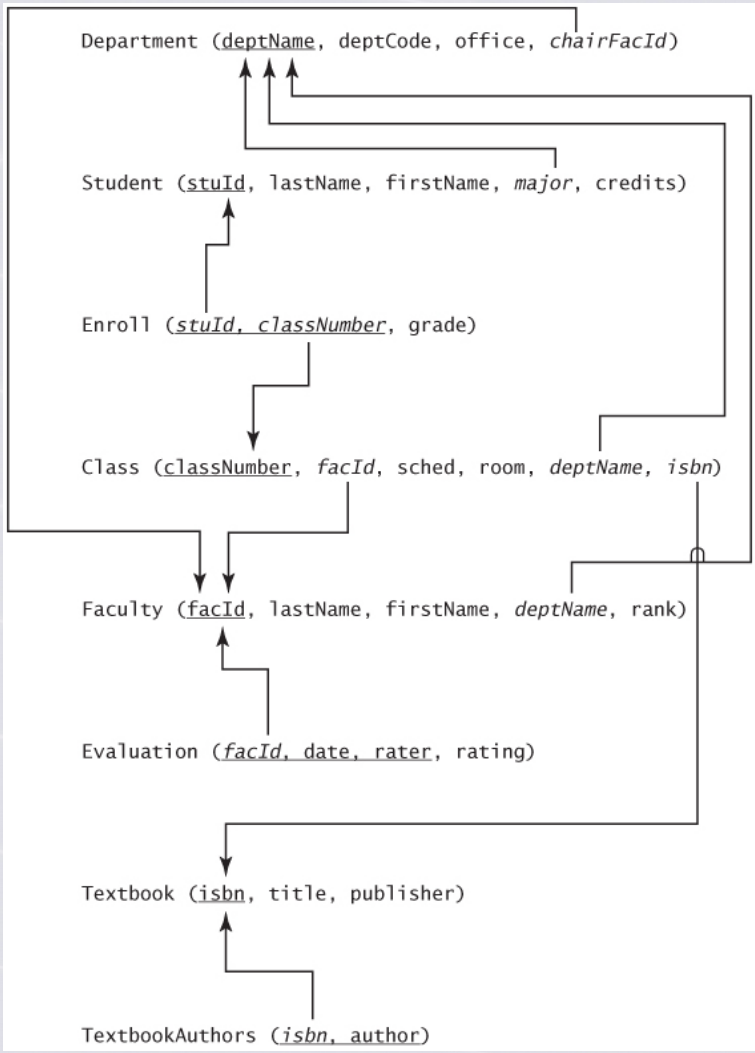
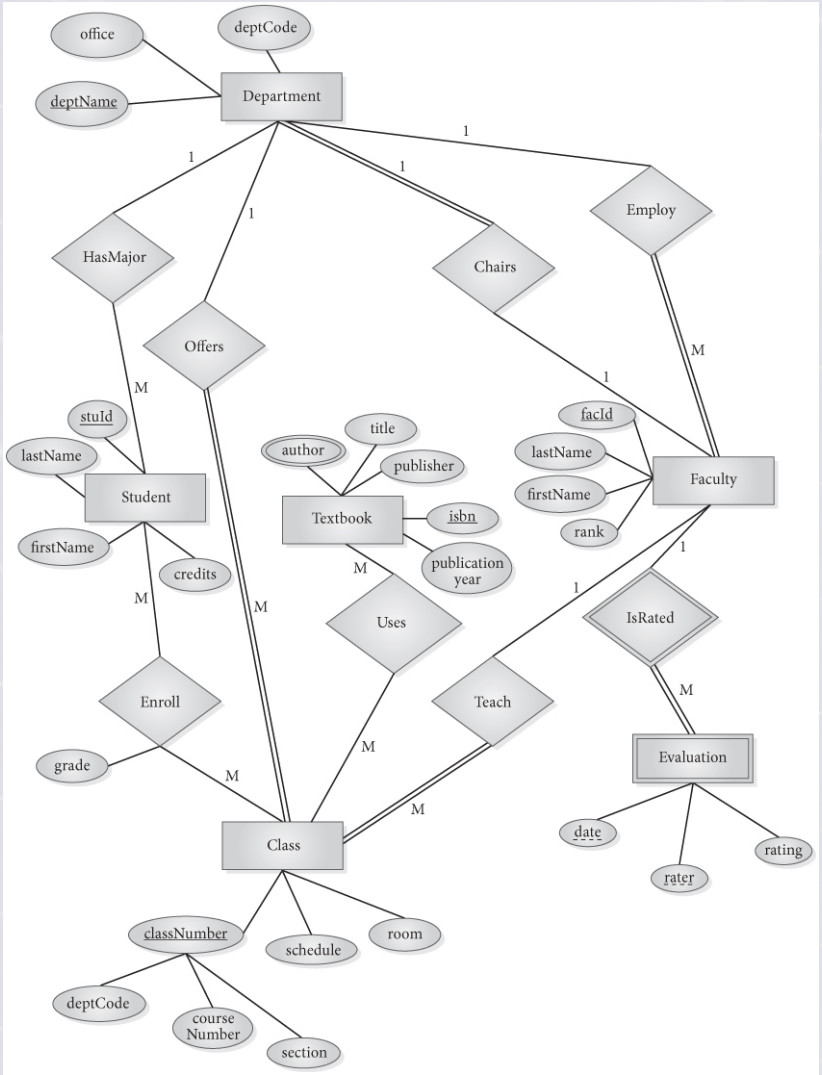


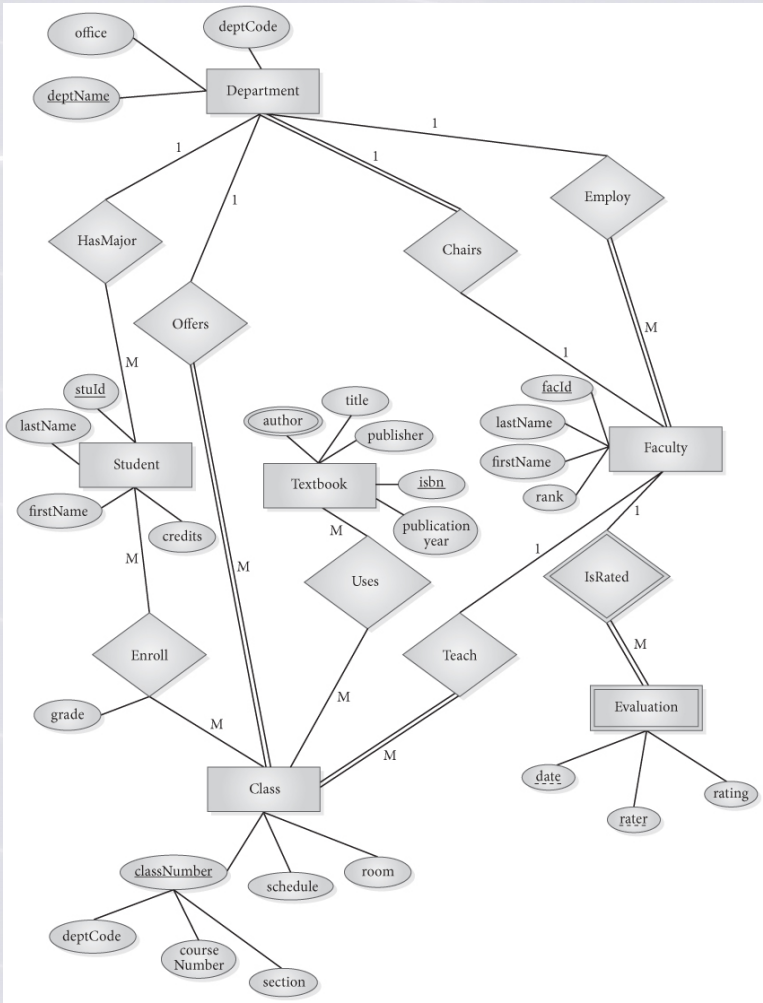
Figure 3.13

- There are eight relationship sets represented in Figure 3.13 by diamonds.
- To represent the 1:M **HasMajor** relationship set connecting Department to Student, we place the primary key of the “one” side, Department, in the table for the “many” side, Student.
- We note that Student already contains major.
- If the names of all major programs are actually just the department names, then this is a foreign key already, so we do not need to add deptName as well.
- If that is not the case, i.e., program majors can differ from department names, then we need to add the department name as a foreign key in Student.
- Note that the foreign key has only one value for each Student record (only one major).
- If we had done the opposite, placing the stuId in the Department table, since a department has many student majors, we would have an attribute with multiple values.



- **Offers** is a one-to-many relationship that connects Department to Class.
- We represent this relationship by putting the key of Department, deptName, in the Class table.
- Note that if we had chosen to split the composite attribute classNumber into its components initially, deptName would have been an attribute in Class already.
- This example shows how an early design decision can resurface as a problem at a later stage of development.
- **Employ** is a one-to-many relationship that represents the association between Department and Faculty.
- We use the foreign key mechanism, adding deptName as an attribute in the Faculty table, making that table: **Faculty(facId, lastName, firstName, rank, *deptName*)**
- **Chairs** is a one-to-one relationship that connects Department to Faculty.
- We will push the primary key of Faculty into Department as a foreign key, changing its name slightly for clarity, making the Department schema: **Department(deptName, deptCode, office, *chairFacId*)**

Figure 3.13



- In Figure 3.13, **Textbook** is related only to Class, so it is represented using the foreign key mechanism, and we place its primary key, isbn, in Class.
- The **isRated** relationship connects the weak entity Evaluation to its owner, Faculty.
- It will be represented by having the primary key of the owner entity in the weak entity's table as part of its primary key.

- Horse Racing Example

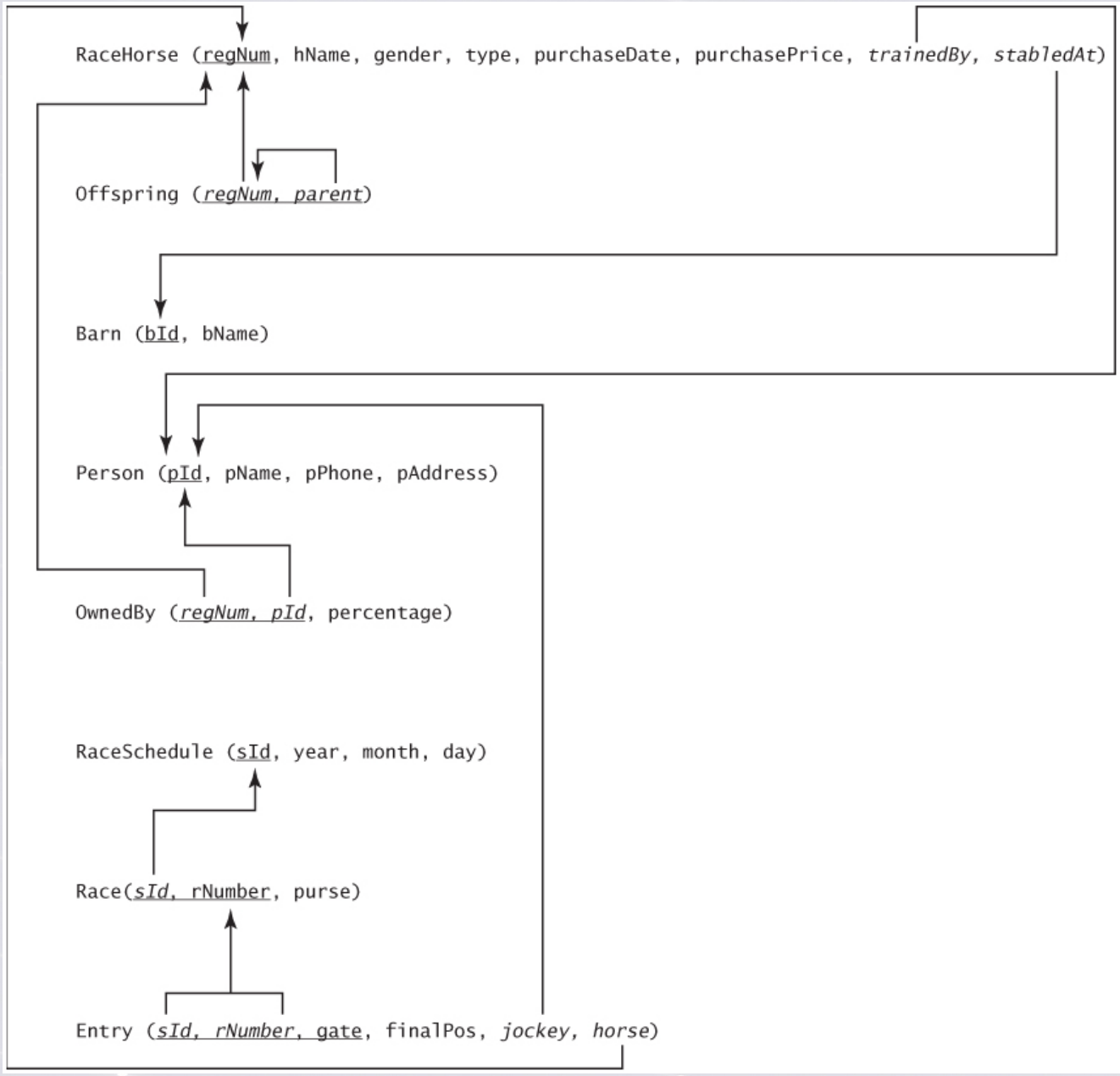


Figure 4.8

Summary of E-R to Strictly Relational Mapping Concepts

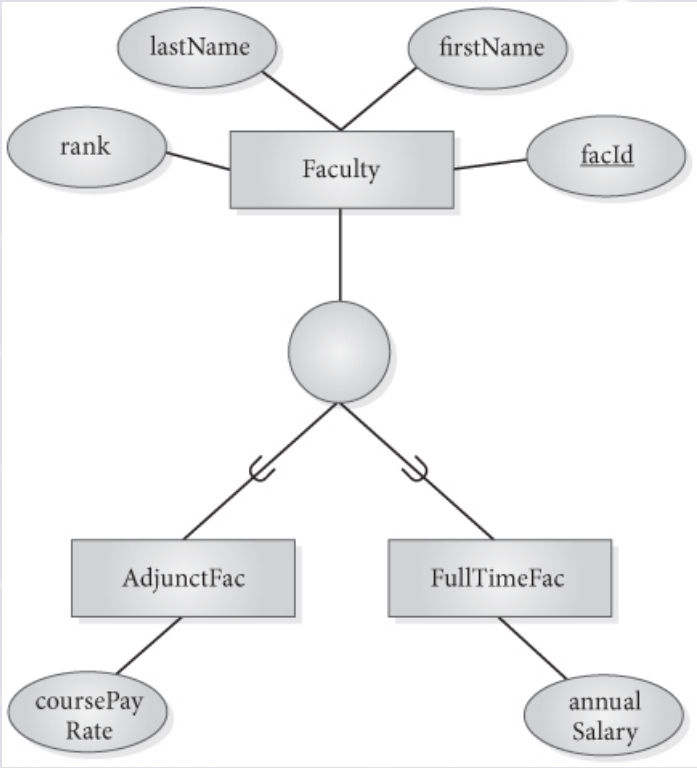
- Map strong entity sets to tables having a column for each single-valued, non-composite attribute.
- For composite attributes, create a column for each component and do not represent the composite. Alternatively, we can create a single column for the composite and ignore the components.
- For each multi-valued attribute, create a separate table having the primary key of the entity, along with a column for the multivalued attribute. Alternatively, if the maximum number of values for the attribute is known, create multiple columns for these values.
- For each weak entity set, create a table that includes the primary key of the owner entity along with the attributes of the weak entity set.
- For one-to-many binary relationships, place the primary key of the one side in the table of the many side, creating a foreign key.
- For one-to-one binary relationships, place either one of the primary keys in the table for the other entity.
- For many-to-many binary relationships, and for all higher-level relationships, create a relationship table that consists of the primary keys of the related entities, along with any descriptive attributes of the relationship.

Mapping EE-R Diagrams to Relational Schemas

- Non-hierarchical entities-map as in ER
- Hierarchies-3 methods
    1. Create a table for the superset and one for each of the subsets. Placing primary key of the superset in each subset table
    2. Create tables for each of the subsets, and no table for the superset. Place all attributes of the superset in each of the subset tables
    3. Create a single table with all the attributes of the superset and all the attributes of all subsets

Mapping EE-R Set Hierarchies to Relational Tables

- To illustrate the representation of subsets, we will use Figure 3.15(A), which shows Faculty with subsets AdjunctFac and FullTimeFac.
- We can choose one of the following methods.



- **Method 1**. Create a table for the superset and one for each of the subsets, placing the primary key of the superset in each subset table.
- Faculty has the attributes shown on the diagram, including a primary key, facId.
- Both AdjunctFac and FullTimeFac tables have columns for their own attributes, plus columns for the primary key of the superset, facId, which functions both as the primary key and a foreign key in those tables. The tables are:

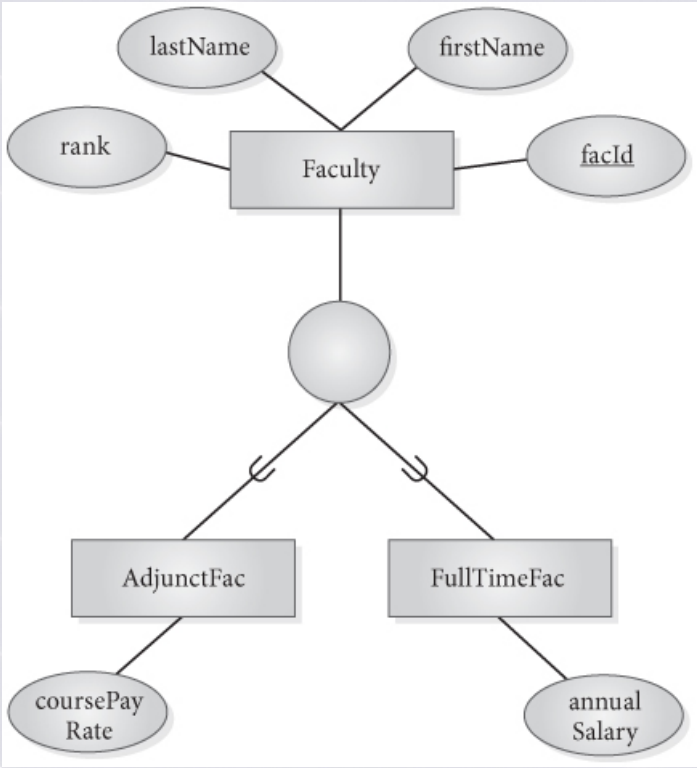
  Faculty(facId, lastName, firstName, rank)
  AdjunctFac(*facId*, coursePayRate)
  FullTimeFac(*facId*, annualSalary)

Mapping EE-R Set Hierarchies to Relational Tables

- **Method 2**. Create tables for each of the subsets and no table for the superset.
- Place all of the attributes of the superset in each of the subset tables.
- For this example, we would have tables for AdjunctFac and FullTimeFac, and none for Faculty.
- The tables are:

AdjunctFac(facId, lastName, firstName, rank, coursePayRate)

FullTimeFac(facId, lastName, firstName, rank, annualSalary)

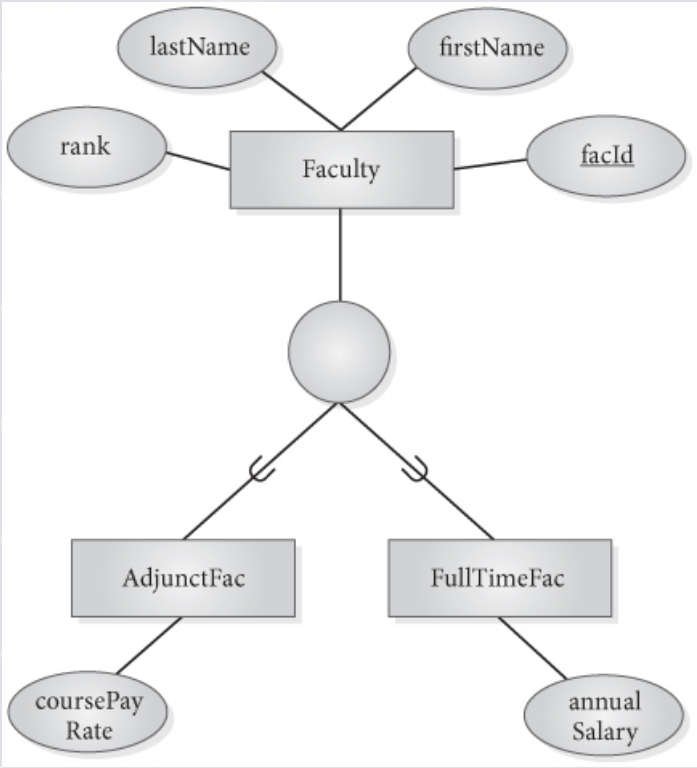


Mapping EE-R Set Hierarchies to Relational Tables

- **Method 3**. Create a single table that contains all the attributes of the superset, along with all the attributes of all subsets.
- For our example, we could create the table:

  AllFac(facId, lastName, firstName, rank,

  annualSalary, coursePayRate)



- A variation of Method 3 is to add a "type field" to the record, indicating which subset the entity belongs to.
- For attribute-defined disjoint specializations, this is simply the defining attribute.
- For overlapping specializations, we could add a type field for each specialization, indicating whether or not the entity belongs to each of the subsets.

Mapping EE-R Set Hierarchies to Relational Tables

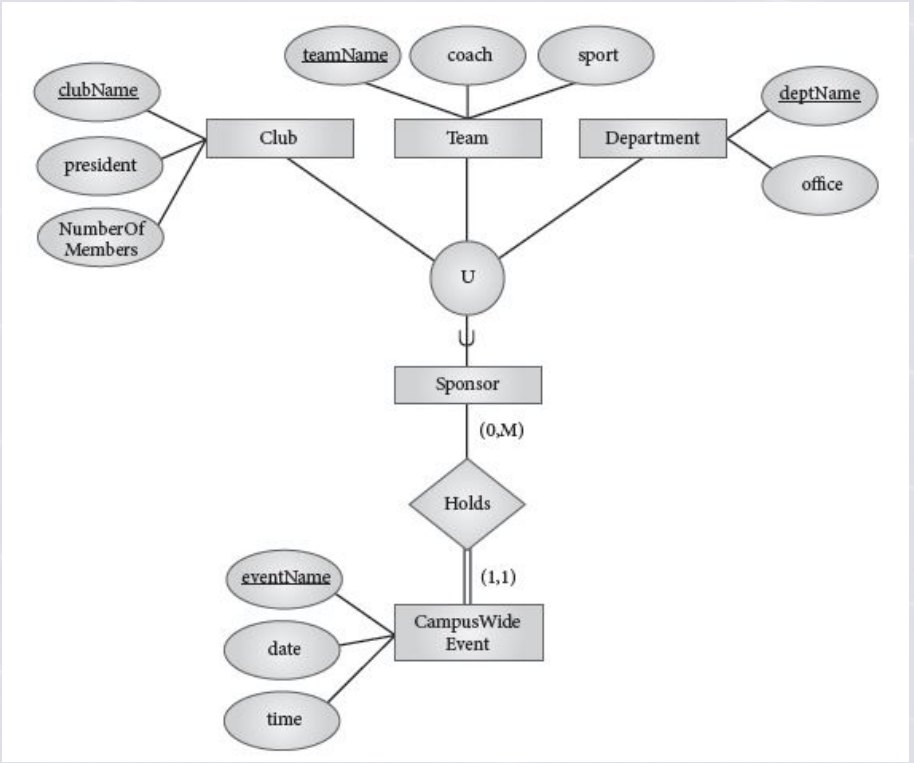
- There are tradeoffs to be considered for each of these methods.
- The first works for all types of subsets, regardless of whether the specialization is disjoint or overlapping; partial or total; and attribute-defined or user-defined.
- For our example, queries about faculty that involve only the common attributes are easily answered using the superset table.
- However, queries about members of a subset will, in general, require a join with the superset table.
- The second method handles queries about the subsets well, since one table has all the information about each subset.
- However, it should not be used if the subsets are overlapping, because data about the instances that belong to more than one subset will be duplicated.
- It cannot be used if the specialization is partial, because the entity instances that do not belong to any subset cannot be represented at all using this method.
- The third method allows us to answer queries about all the subsets and the superset without doing joins, but it will result in storing many null values in the database.

Mapping Unions

- Create table for the union, and individual tables for each of the supersets, using foreign keys to connect them
- If superset keys are different types, create a surrogate key for union

Mapping Unions

- Figure 3.19(A) shows Sponsor as the union of Club, Team, and Department.
- Although these are all names, they might not be comparable because of differences in length.
- Some unions might involve grouping together entity sets with radically different keys.
- The solution is to create a surrogate key that will be the primary key of the union.

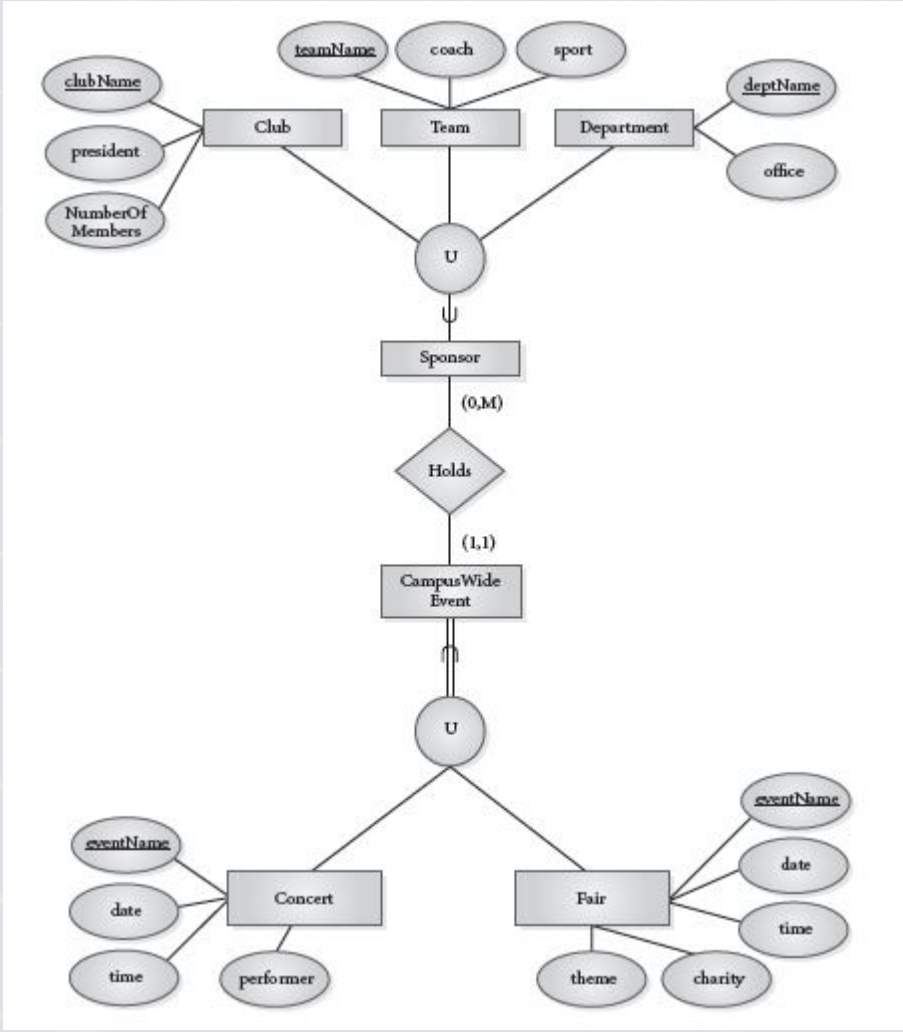


- It will be a foreign key in the tables for each of the supersets.
- For the Sponsor example, we create a surrogate key for sponsor, sponsorId, and we insert that attribute as a foreign key in the tables for Club, Team, and Department.
- We also add SponsorType to indicate whether the sponsor is a club, team, or department. The schema will be:

Sponsor(sponsorId, sponsorType)
Club(clubName, president, numberOfMembers, *sponsorId*)
Team(teamName, coach, sport, *sponsorId*)
Department(deptName, office, *sponsorId*)

Mapping Unions

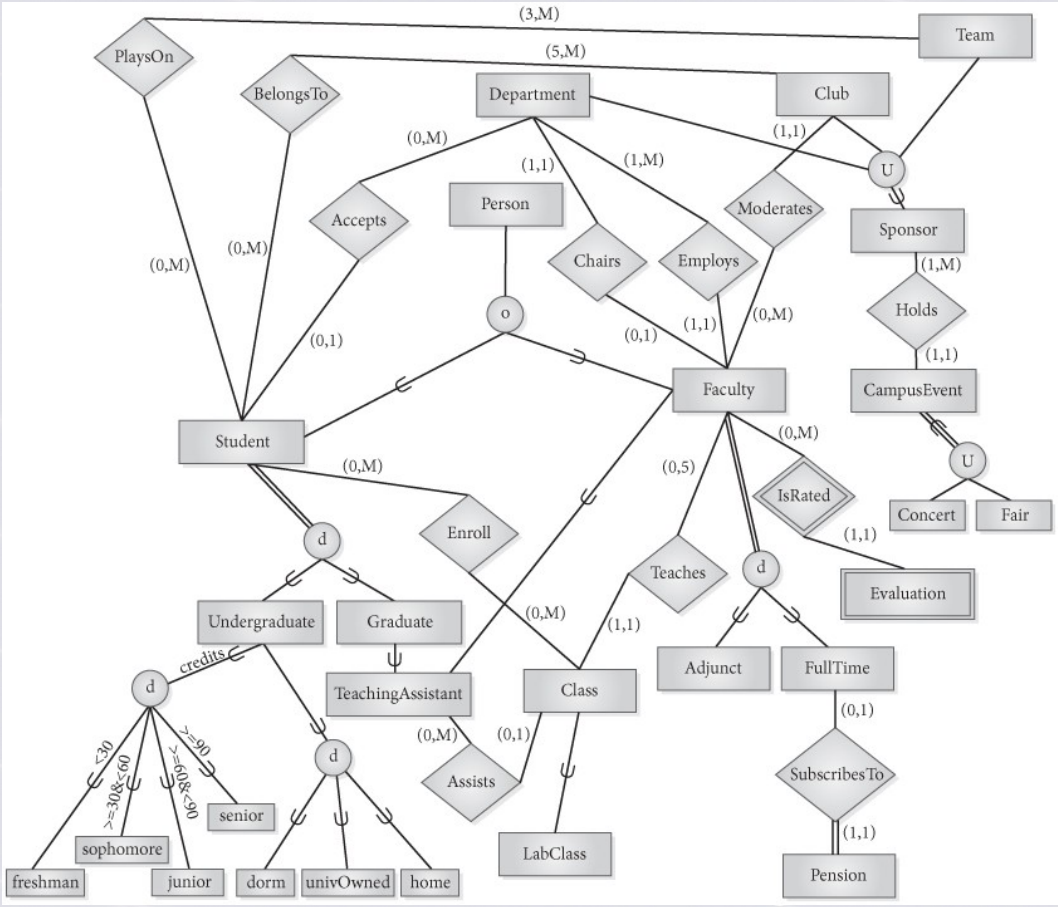
- If the primary keys of the supersets are the same, it is not necessary to create a surrogate key.
- The table for the union will consist of the common primary key field and a type field.
- The common primary key field will also function as a foreign key for the super class tables.
- For the CampusWideEvent example shown as a union in Figure 3.19(B), the schema will be:



CampusWideEvent(eventName, eventType)
Concert(*eventName*, date, time, performer)
Fair(*eventName*, date, time, theme, charity)

Extended University Example

- Applying these techniques to the EE-R diagram shown in Figure 3.21, we have a hierarchy for Person, with subsets Student and Faculty, each of which itself has subsets.
- Our first decision is to not represent the Person superset, since it does not participate as a whole in any relationships.
- We create separate tables for Studet and Faculty, repeating the common Person attributes in each, which is Method 2 for representing hierarchies, as described earlier.
- Since the specialization is partial, however, there may be people other than students and faculty in the mini-world.



- If that is the case, we would also create a table called OtherPeople with attributes pId, lastName, and firstName.

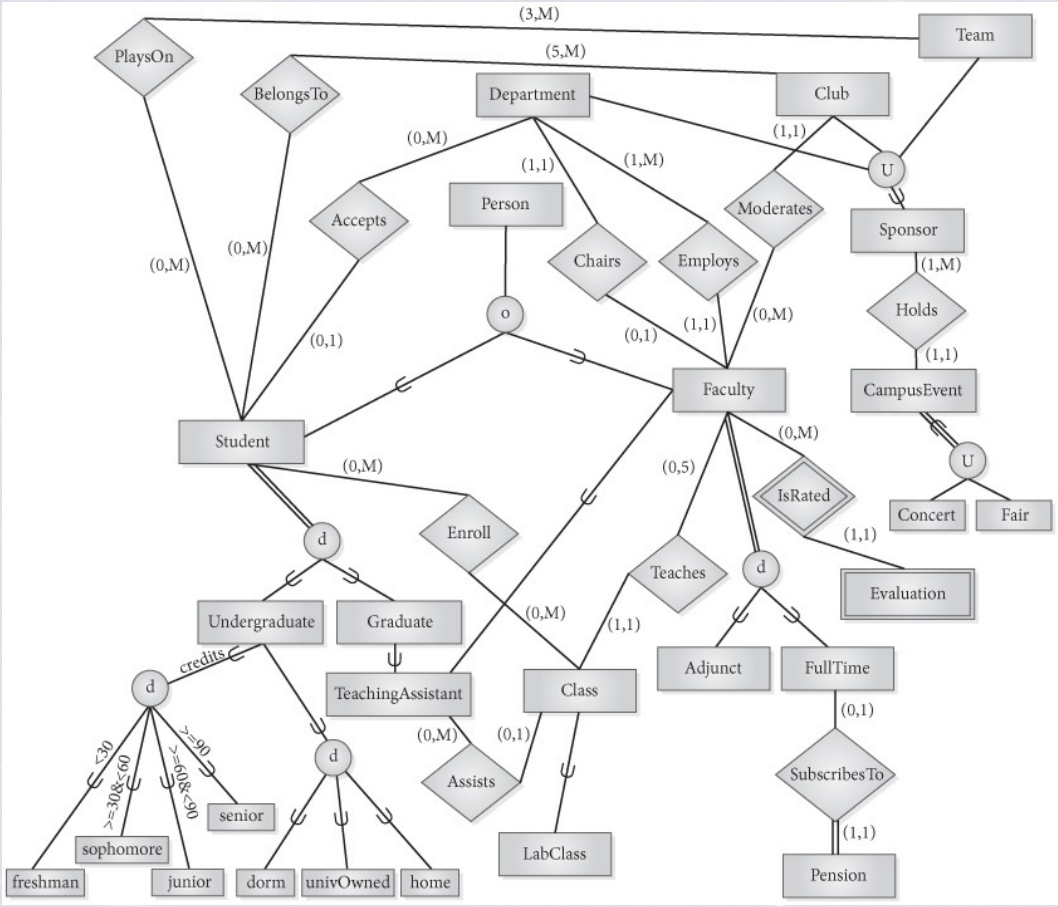
Faculty(<u>facId</u>, lastName, firstName, rank)
Student(<u>stuId</u>, lastName, firstName, credits)

Extended University Example

- Going down the hierarchy, we see that although Student has subsets, it also participates in relationships as a whole, so we will represent the Student hierarchy by a table and then create tables for both undergraduate and graduate students, which is Method 1.
- We change the name of the stuId field slightly to reduce confusion:

  Student(<u>stuId</u>, lastName, firstName, credits)
  Undergraduate(*<u>undergradStuId</u>*, major)
  Graduate(*<u>gradStuId</u>*, program)
- Similarly, Faculty participates as a whole in many relationships, but its subsets have distinctive attributes, and full-time faculty may have a pension plan, so we use Method 1 again, creating the tables

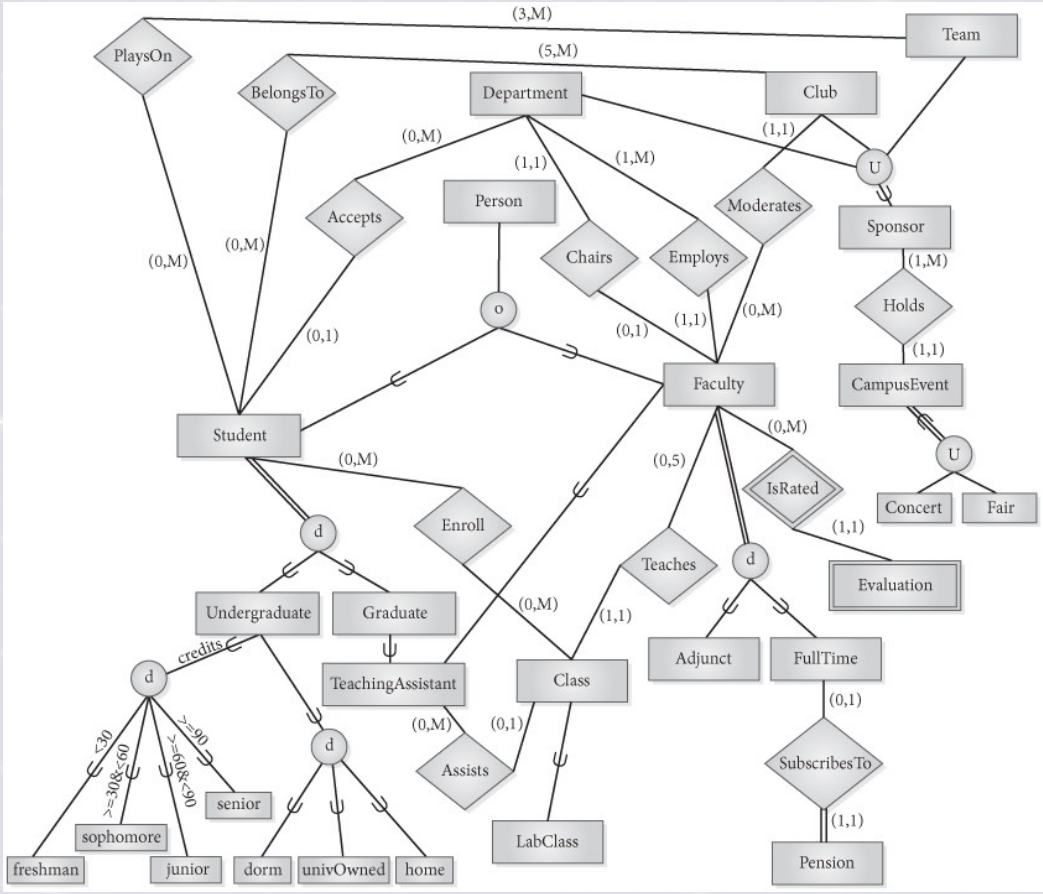
  Faculty(<u>facId</u>, lastName, firstName, rank)
  AdjunctFac(*<u>adjFacId</u>*, coursePayRate)
  FullTimeFac(*<u>fullFacId</u>*, annualSalary)



Extended University Example

- The Undergraduate subset has subsets for class years and for residences, but the one for class years is attribute-defined.
- Since it would be a simple matter to examine the credits value to determine the

  class year, it does not seem necessary to break up the Undergraduate table by class year.
- If we need to record any attributes specific to a class year, such as a thesis advisor for seniors, we add those to theUndergraduate table.
- Similarly, for residence, the residence type and address might be sufficient, so we use Method 3 and we change the undergraduate table slightly to be



Undergraduate(*undergradStuId*, major, thesisAdvisor, resType, address)

- Graduate has a single subset, TeachingAssistant, and we decide to represent that specialization using Method 3, by adding its two attributes to the Graduate table, changing to

  Graduate(*gradStuId*, program, tuitionRemission, fundingSource)

Extended University Example

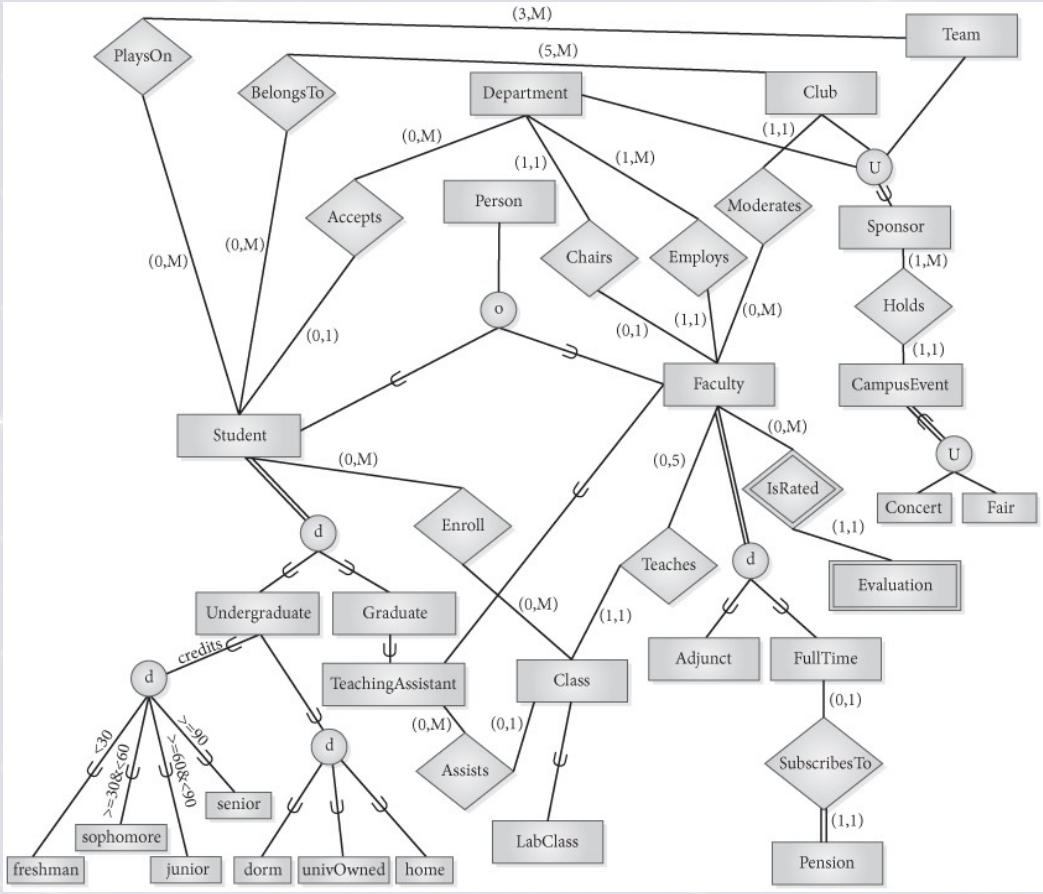
- The diagram shows that TeachingAssistant is also a subclass off Faculty, so we put facId in as an attribute for those graduate students who are TAs.

Graduate(*gradStuId*, program, tuitionRemission, fundingSource, facId)

- The strong entities Department, Club, and Team have no subsets, so we initially represent them by the tables

Department (deptName, deptCode, office)
Club(clubName, president, numMembers)
Team(teamName, coach, sport)

- We notice that these entities participate in a union for Sponsor, as discussed earlier, so we create a surrogate key and add it to the three existing tables:



Sponsor(sponsorId, sponsorType)
Department(deptName, deptCode, office, *sponsorId*)
Club(clubName, president, numberOfMembers, *sponsorId*)
Team(teamName, coach, sport, *sponsorId*)

Extended University Example

- Similarly, we have a CampusEvent union of Concert and Fair, which we represent by the tables

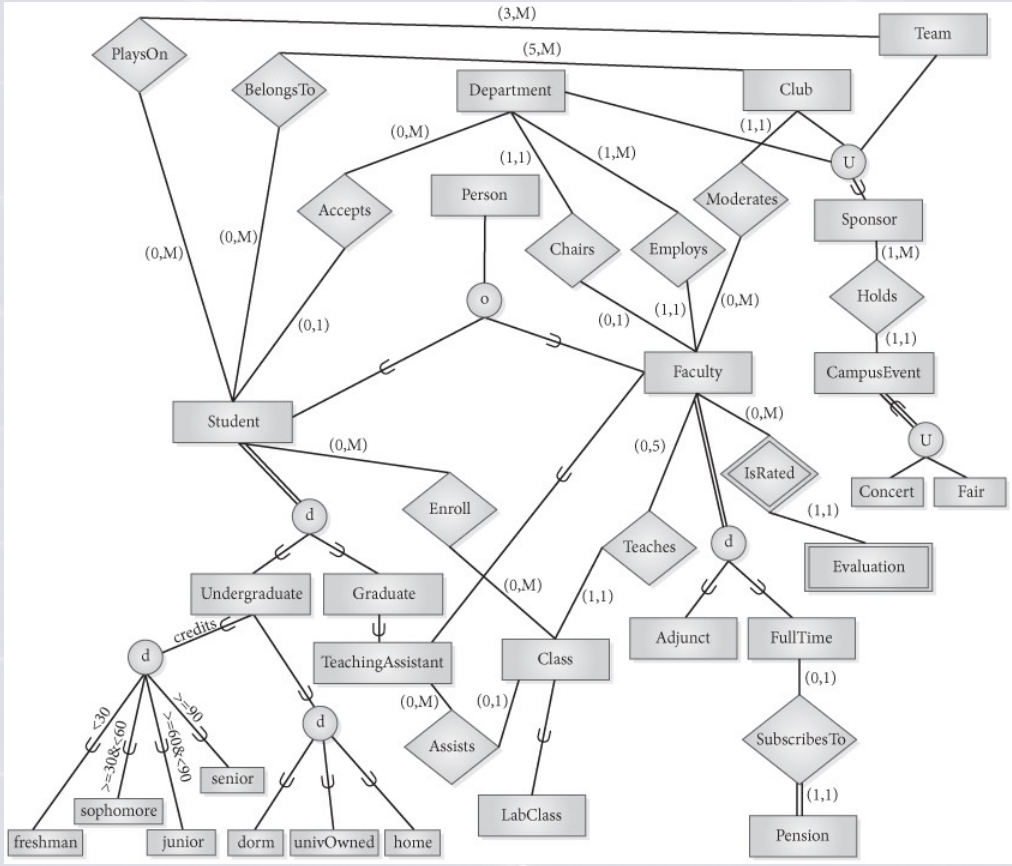
CampusEvent(<u>eventName</u>, eventType)
Fair(*<u>eventName</u>*, date, time, theme, charity)
Concert(*<u>eventName</u>*, date, time,performer)

- The entity Class is represented along with its subset labClass using Method 3, by the table

Class(<u>classNo</u>, className, schedule, room, labNo, labSched)

- The table for the weak entity Evaluation has the primary key of its associated strong entity, Faculty, giving us

Evaluation(<u>facId, date, raterName</u>, rating)

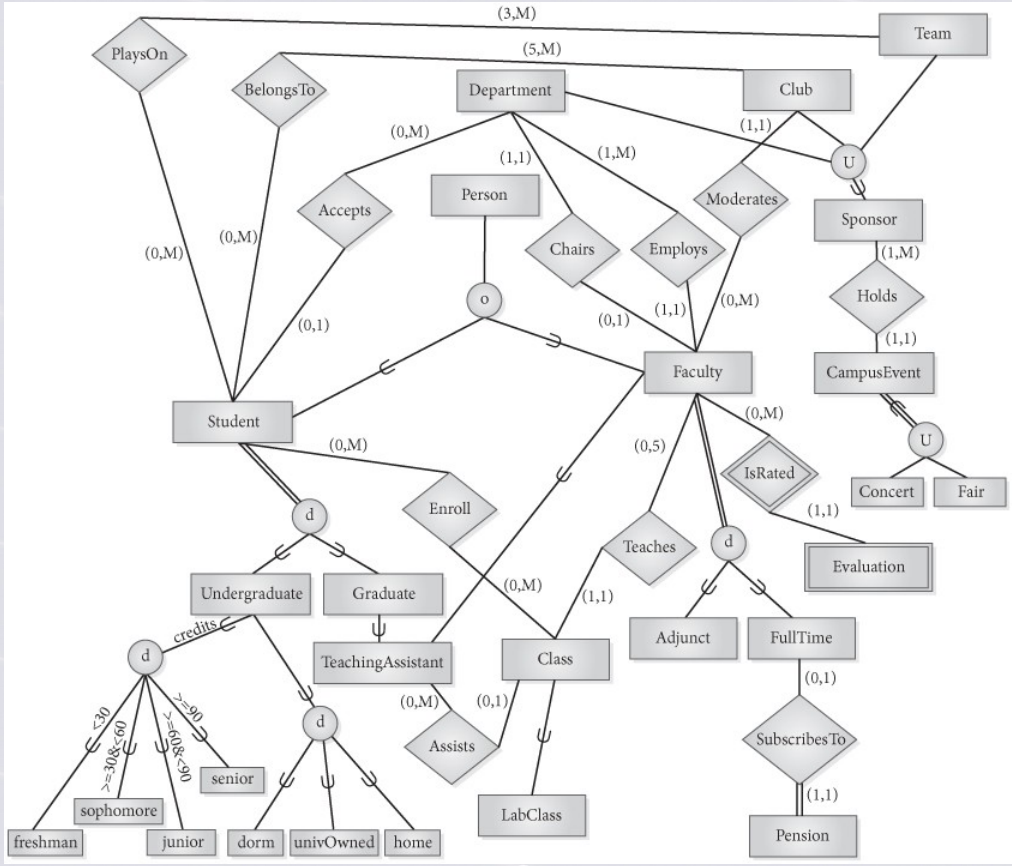


- The entity Pension is represented by the table

Pension(<u>companyName, contractNo</u>, contactPerson)

Extended University Example

- For the one-to-many relationships EmploysModerates, Teaches, Assists, Accepts, and Holds, we push the key of the one side into the table for the many side, modifying those tables with foreign keys. We show the foreign keys in italics in the resulting tables below.



Faculty(facId, lastName, firstName, rank, *deptName*)
Club(clubName, president, numberOfMembers, *sponsorId, moderatorFacId*)
Class(classNo, className, schedule, room, labNo, labSched, *facId, gradStuId*)
Student(stuId, lastName, firstName, credits, *deptName*)
CampusEvent(eventName, eventType, *sponsorId*)

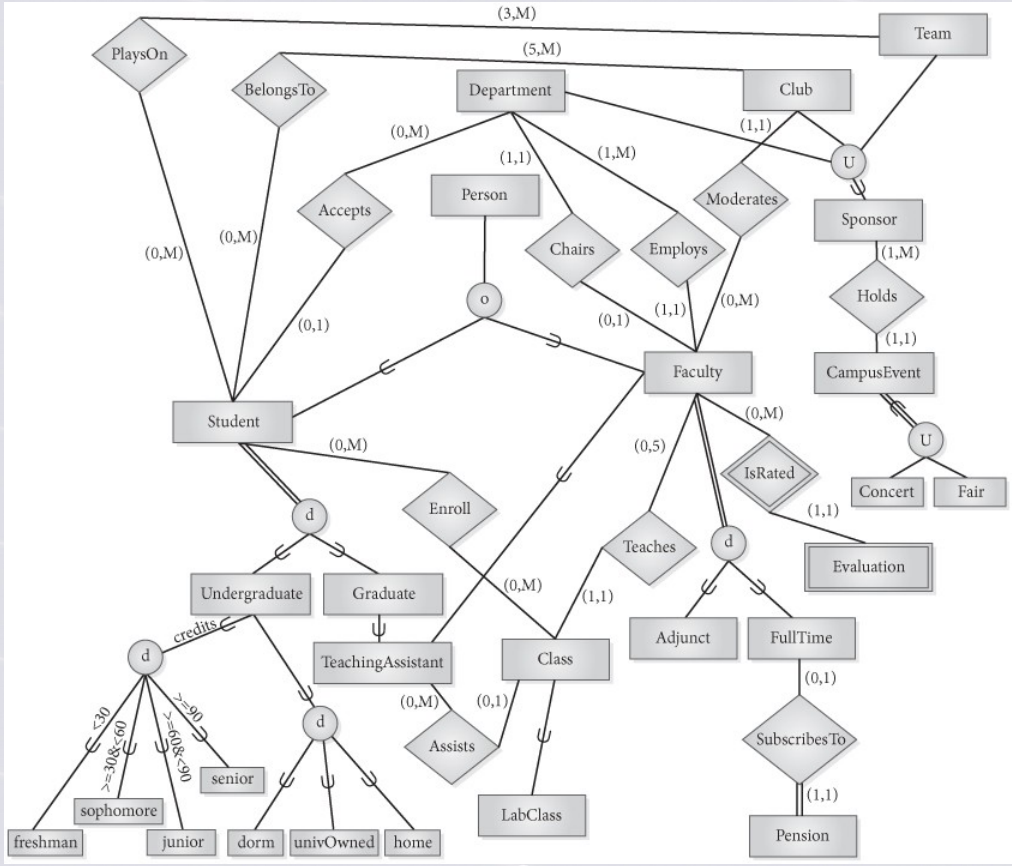
- We note that the isRated relationship is already represented, since the key of Faculty is already present in the table for the weak entity Evaluation.

Extended University Example

- For the one-to-one relationship Chairs we put the key of Faculty in Department, giving us

Department(deptName, deptCode, office, *sponsorId, chairFacId*)

- Similarly, the one-to-one relationship SubscribesTo can be represented by placing the key of PensionPlan in the table for full-time faculty, giving us
  FullTimeFac(*fullFacId*, annualSalary, *companyName, contractNo*)
- For the many-to-many relationships Enroll, PlaysOn, and BelongsTo we create relationship tables
- **Enroll(*stuId, classNo*)**
  PlaysOn(*stuId, teamName*)
  BelongsTo(*stuId, clubName*)



Extended University Example

Faculty(facId, lastName, firstName, rank, *deptName*)
AdjunctFac(adjF*acId,* coursePayRate)
FullTimeFac(fullF*acId,* annualSalary, *companyName, contractNo*)
Pension(companyName, contractNo, contactPerson)
Evaluation(facid, date, raterName, rating)
Class(classNo, className, schedule, room, labNo, labSched, *facId, gradStuId*)
Student(stuid, firstName, lastName, credits, *deptName*)
Undergraduate(*undergradstuId*, major, thesisAdvisor, resType, address)
Graduate(*gradStuId*, program, tuitionRemission, fundingSource, *facId*)
Sponsor(sponsorId, sponsorType)
Department(deptName, deptCode, office, *sponsorId, chairFacId*)
Club(clubName, president, numberOfMembers, *sponsorId, moderatorFacId*)
CampusEvent(eventName, eventType, *sponsorId*)
Team(teamName, coach, sport, *sponsorId*)
Fair(*eventName*, date, time, theme, charity)
Concert(*eventName*, date, time, performer)
Enroll(*stuId,classNo*)
PlaysOn(*stuId, teamName*)
BelongsTo(*stuid, clubName*