Part 8: Relational Model III

References:

- Elmasri/Navathe:Fundamentals of Database Systems, 3rd Edition, 1999.
  9.1 Relational DB Design Using ER-to-Relational Mapping
  9.2 Mapping EER Model Concepts to Relations
  14. Functional Dependencies and Normalization for Relational Databases
- Siberschatz, Korth, Sudarshan: Database System Concepts, 3rd Edition. Section 2.9, 7.1–7.3. McGraw-Hill, 1999.
- Kemper/Eickler: Datenbanksysteme (in German), 2nd Ed., Section 3.2–3.3, Ch. 6, Oldenbourg, 1997.
- Lipeck: Skript zur Vorlesung Datenbanksysteme (in German), Univ. Hannover, 1996.
- Batini/Ceri/Navathe: Conceptual Database Design, An Entity-Relationship Approach. Benjamin/Cummings, 1992.
- Rauh/Stickel: Konzeptuelle Datenmodellierung (in German). Teubner, 1997.
- Teorey: Database Modeling and Design, Third Edition, 1999.
- Kent: A Simple Guide to Five Normal Forms in Relational Database Theory. Communications of the ACM 26(2), 120–125, 1983.

Objectives

After completing this chapter, you should be able to:

- translate a schema in the ER-model into an equivalent schema in the relational model.

  Including keys, foreign keys, and, if needed, further constraints.
- check two given ER-schemas for equivalence.

  The translation into the relational model might be helpful: If the translation results are equal, the schemas are certainly equivalent.
- determine functional dependencies (FDs) for a given relation.
- check a given relation for Boyce-Codd-Normalform (BCNF).

Overview

1. Translation of Basic ER-Constructs
2. Translation of Advanced ER-Constructs
3. Relational Normal Forms (BCNF)

General Remarks (1)

- Usually, designing a relational database schema is done by first designing an ER-schema for the application, and then transforming the ER-schema into the relational model:
  - An ER-schema allows better documentation of the relationship between the schema and the real world.

    E.g. entity types and relationships are distinguished.
  - The ER-model has a useful graphical notation.
  - The ER-model has constructs like inheritance which have no direct counterpart in the relational model.

    The difficult conceptual design can be simplified a bit by first using the extended possibilities.

General Remarks (2)

- Given an ER-schema s_{ER}, the goal is to construct a relational schema s_{REL} such that there is a one-to-one mapping τ between the possible states for s_{ER} and s_{REL}.

  I.e. each possible DB state with respect to s_{ER} has exactly one counterpart state with respect to s_{REL} and vice versa.

- States possible in the relational schema but meaningless with respect to the ER-schema must be excluded by integrity constraints.

  E.g. when relationships are translated into tables, foreign key constraints are required. In the ER-model, relationships can be always only between currently existing entities. In the RM, "dangling pointers" must be explicitly excluded.

General Remarks (3)

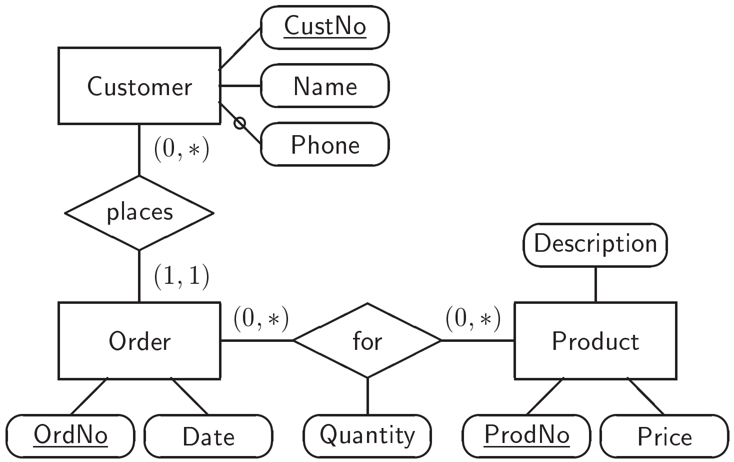
- In addition, it must be possible to translate queries referring to s_{ER} into queries with respect to s_{REL}, evaluate them in the relational system, and then translate the answers back.

  For every s_{ER}-state S and ER-query Q, the same answer must result when Q is evaluated or when the translated query is evaluated and result is translated back.

- I.e. it must be possible to simulate the designed ER-database with the actually implemented relational database.

  Any schema translation must explain the correspondance of schema elements such that, in our case, a query intended for the ER-schema can also be formulated with respect to the relational schema.

Example



Step 1: Entities (1)

- First you create a table for each entity.
- The name of this table is the name of the entity type.

  Alternatively, the plural form can be used.

- The columns of this table are the attributes of the entity type.

  Optional attributes translate into columns allowing null values.

- The primary key of the table is the primary key of the entity type. (If the entity type has no key, add an artificial key.)
- Result in the example:

  Customers(<u>CustNo</u>, Name, Phone^o)

  Orders(<u>OrdNo</u>, Date)

  Products(<u>ProdNo</u>, Description, Price)

Step 1: Entities (2)

Customers		
<u>CustNo</u>	Name	Phone
10	Jones	624-9404
11	Smith	

Orders	
<u>OrdNo</u>	Date
200	2/15/00
201	2/16/00

Products		
<u>ProdNo</u>	Description	Price
1	Apple	0.50
2	Kiwi	0.25
3	Orange	0.60

Step 2: One-To-Many Relationships (1)

- If a relationship has the maximum cardinality 1 on one side, it is one-to-many (and this side is the many side).

  If it has maximum cardinality 1 on both sides, it is actually one-to-one (see below).
- E.g. "places" is one-to-many from "Customer" to "Order".
- In this case you add the key of the "one" side (Customer) as a column to the "many" side (Order).
- This column will be a foreign key referencing the row which corresponds to the related entity.
- Result in the example:

  `Orders(`<u>`OrdNo`</u>`, Date, CustNo→Customers)`

Step 2: One-To-Many Relationships (2)

Customers		
<u>CustNo</u>	Name	Phone
10	Jones	624-9404
11	Smith	

Orders		
<u>OrdNo</u>	Date	CustNo
200	2/15/00	11
201	2/16/00	11

Products		
<u>ProdNo</u>	Description	Price
1	Apple	0.50
2	Kiwi	0.25
3	Orange	0.60

Step 2: One-To-Many Relationships (3)

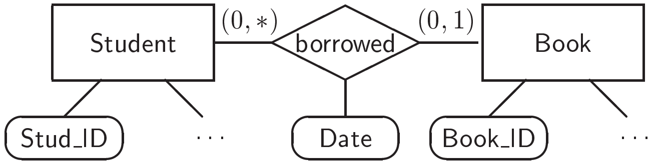
- If the minimum cardinality is 1 (as in this example), null values are not allowed for the new foreign key column.
- If the minimum cardinality should be 0, null values must be allowed for the foreign key column.

  It is null for entities not participating in the relationship.
- The relationship name can be used in the column name, e.g.

  `Orders(`<u>`OrdNo`</u>`, Date, placed_by→Customers)`
- This excludes natural joins, but it is matter of style.
- Of course, all columns in the table must have unique names. If the added foreign key has the same name as an existing column, one or both must be renamed.

Step 2: Extensions and Variants (1)

- Relationships can have attributes, e.g.:



- These attributes are stored together with the pointer to the related entity:

  Books(<u>Book_ID</u>, ..., Stud_ID^o→Students, Date^o)

- "Stud_ID" and "Date" can be null, since not every book is borrowed, but they can either be both null or both not null.

Step 2: Extensions and Variants (2)

- One-to-Many Relationships with the cardinality $(0, 1)$ can be translated into a table of their own:

  borrowed_by(<u>Book_ID</u>→Books, Stud_ID→Students, Date)

- The key values of the related entities are stored, plus attribute values of the relationship.
- The key attributes of the side with the $(0, 1)$ cardinality become the key of this relation.

  Every book can be borrowed only once at the same time.

- Exercise: Why does this not work with the cardinality (1,1)? Why are no null values necessary? Which translation results in more efficient queries?

Step 3: Many-To-Many Relationships (1)

- A relationship is many-to-many if it has the maximum cardinality $*$ on both sides (e.g. "for" in the example).

  Other large maximum cardinalities would be translated in the same way, but need additional constraints.

- Many-to-many relationships become their own tables.
- The columns of this table are the keys of the participating entity types, which together form the key of this table.
- These columns are at the same time foreign keys, referencing the tables for the entity types.
- Relationship attributes are added as columns (non-key):

  for(<u>OrdNo</u>→Orders, <u>ProdNo</u>→Products, Quantity)

Step 3: Many-To-Many Relationships (2)

Customers

<u>CustNo</u>	Name	Phone
10	Jones	624-9404
11	Smith	

Orders

<u>OrdNo</u>	Date	CustNo
200	2/15/00	11
201	2/16/00	11

Products

<u>ProdNo</u>	Description	Price
1	Apple	0.50
2	Kiwi	0.25
3	Orange	0.60

for

<u>OrdNo</u>	<u>ProdNo</u>	Quantity
200	1	1
200	2	1
201	1	5

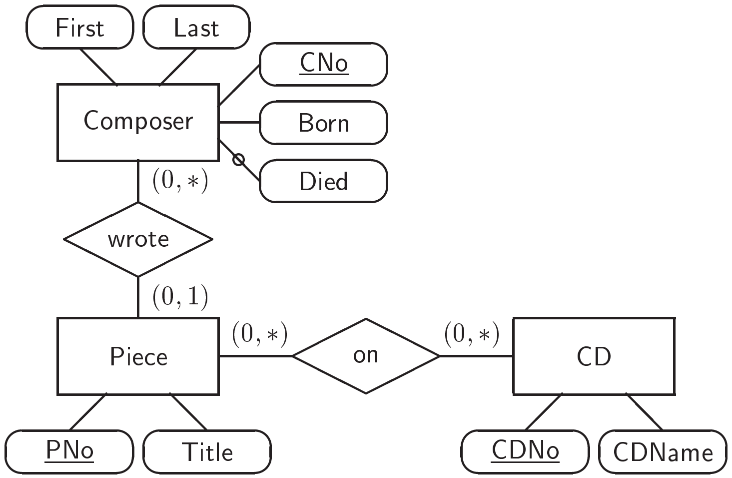
Step 3: Many-To-Many Relationships (3)

- Note that the key of "for" must really consist of both, "OrdNo" and "ProdNo".

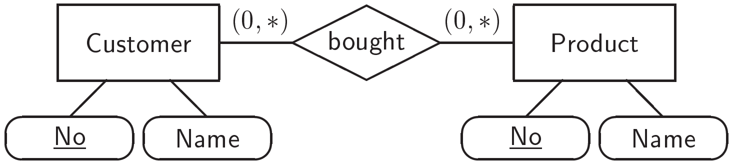
  Since one order can request multiple products, "OrdNo" alone cannot be key. Since the same product may be subject of multiple orders, also "ProdNo" alone does not suffice.
- The table "`for`" can be renamed to "`Order_Details`".
- Minimum cardinalities other than 0 for many-to-many relationships cannot be enforced by the standard constraints of the relational model.

  E.g. it would make sense to require that every order must be "for" at least one product. But this becomes a general constraint in the relational model.

Exercise



Renaming of Attributes (1)



- This would yield a table

  bought(No → Customer, No → Product).
- However, such a table is illegal, since columns of the same table must have different names.

Renaming of Attributes (2)

- Attributes can be renamed in any understandable way.

  Of course, the renaming must be carefully documented.
- E.g. combine entity type name and attribute name, as in "Customer_No" (or a shorthand like "CNo").
- Consider renaming the attribute not only in the relationship table (where the name clash occured), but also in the entity tables. This will simplify natural joins:

  Customer(Cust_No, Cust_Name)

  Product(Prod_No, Prod_Name)

  bought(Cust_No → Customer, Prod_No → Product)
- Actually, it is best to do the renaming in the ER-schema.

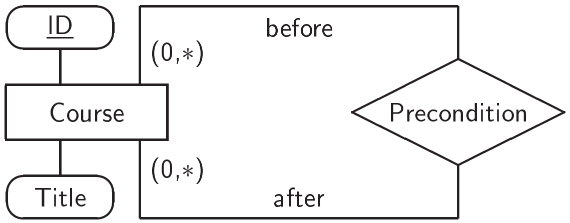
Renaming of Attributes (3)

- In the relational model, columns should only have the same name if they have the same domain, so that a join on them is possible.

  E.g. although the “Name” attributes do not result in a name clash, customer names and product names are entirely different, so different attribute names should be used. In the ER-model, this is not necessary, since attributes are strictly local to the entity types. Because of the natural join, this is not completely true in the relational model (although it is only a matter of taste, not a hard rule). In both models, it might be confusing if entirely different things have the same attribute name.

Renaming of Attributes (4)

- Renaming is unavoidable for recursive relationships:

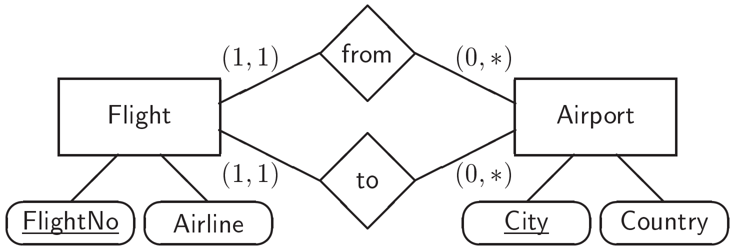


- Utilize role names for renaming, e.g. “before_ID” or simply:

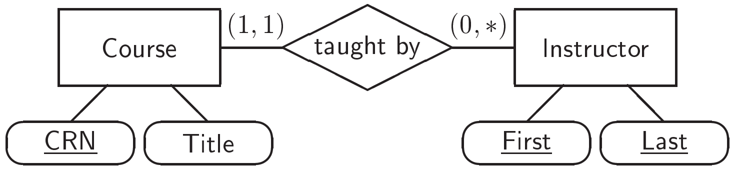
  Precondition(<u>before</u> → Course, <u>after</u> → Course)

  Course(<u>ID</u>, Title)

Exercise



Composite Foreign Keys



- This would yield a table

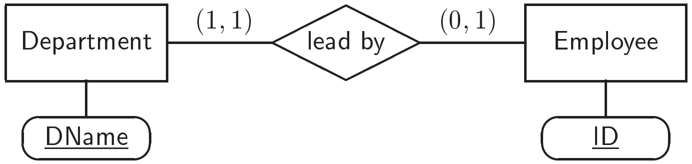
  Course(<u>CRN</u>, Title, (First, Last) → Instructor)

- The following is **wrong**:

  Course(<u>CRN</u>, Title, First → Instructor, Last → Instructor).

- If the minimum cardinality is 0 (courses without instructors), “First” and “Last” can be null, but only together.

Step 4: One-to-One Relationships (1)



- A relationship is one-to-one if it has maximum cardinality 1 on both sides.
- Basically, the translation is the same as for one-to-many relationships.

  However, there is an additional key constructed, see below.
- Put the key of "Employee" into the "Department" table or vice versa.

Step 4: One-to-One Relationships (2)

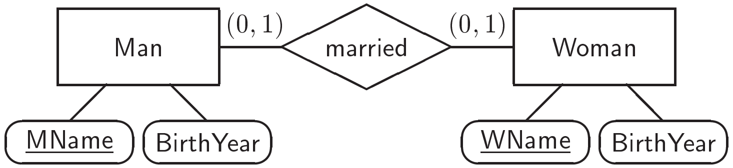
- In this example, it is better to include the Employee key in the Department table, since Department has cardinality (1,1):

  Department(<u>DName</u>, . . . , Head → Employee)
- In this way, null values are avoided and the minimum cardinality 1 is enforced.

  If the department name is included in the Employee table, it can be null. In addition, a general constraint is required to ensure that every departments has a department head.
- "Head" is now also a key for the table "Department" (!), since an employee can be head of at most one department.

  It is only an alternative key, not part of the primary key.
  It enforces the maximum cardinality 1 on the Employee side.

Step 4: One-to-One Relationships (3)

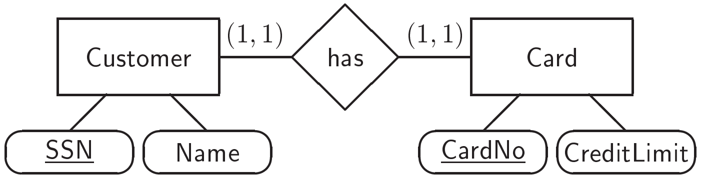


- Here the key of any of the two tables can be included in the other table (as a possibly null foreign key).

  However, it would be wrong to do both (redundancy).
- Or translate the relationship into a table on its own:

  Marriage(MName → Man, WName → Woman)
- Exercise: What is the key / keys?

Step 4: One-to-One Relationships (4)

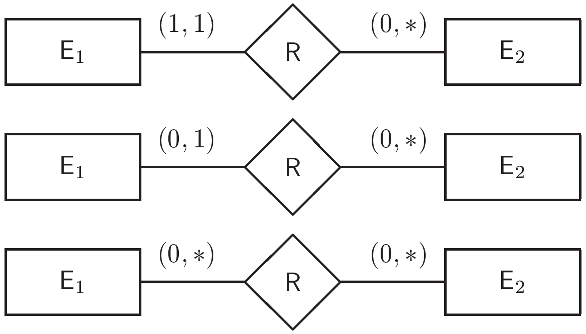


- Merge the tables to enforce the minimum cardinality 1:

  CustomerCard(SSN, Name, <u>CardNo</u>, CreditLimit)
- SSN and CardNo are both keys. One is selected as primary key, the other is an alternative (secondary) key.
- This merging method could be used also for other one-to-one relationships, but then requires null values (→ Specialization).

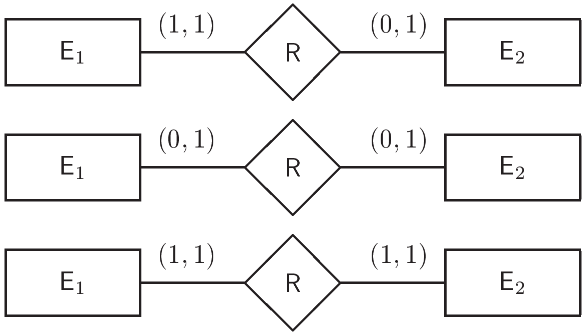
Limitations of the Translation (1)

- The following cardinalities can be translated with the methods explained above (using only the standard constraints of the relational model) (continued on next slide):



Limitations of the Translation (2)

- All kinds of one-to-one relationships can be handled:



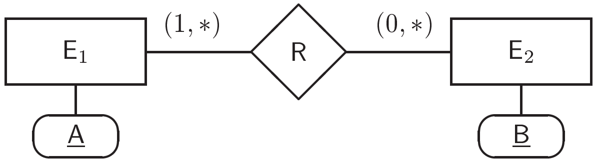
- If a relationship is none of these six cases, general constraints must be used (or a specialized solution must be invented).

Limitations of the Translation (3)

- Maximal cardinalities greater than 1 and smaller than "$*$" become general constraints.

  Minimal cardinalities greater than 1 are also not treated.

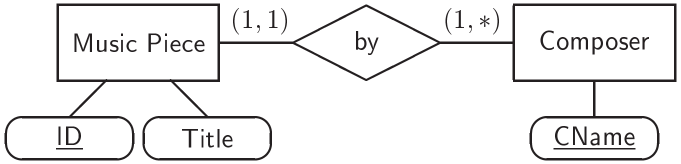
- The minimal cardinality 1 is only treated for $(1,1)$. E.g.:



$\mathsf{E}_1(\underline{\mathsf{A}}, \ldots)$ $\mathsf{E}_2(\underline{\mathsf{B}}, \ldots)$ $\mathsf{R}(\underline{\mathsf{A}} \rightarrow \mathsf{E}_1, \underline{\mathsf{B}} \rightarrow \mathsf{E}_2)$.

What is missing is the inclusion constraint $\pi_{\mathsf{A}}(\mathsf{E}_1) \subseteq \pi_{\mathsf{A}}(\mathsf{R})$. This is similar to a foreign key, but A is not a key of R.

Step 5: Unnecessary Tables (1)



- Sometimes relations created by this translation carry no additional information and can be deleted.

  Music_Piece(<u>ID</u>, Title, CName → Composer)

  Composer(<u>CName</u>)

- Composer is redundant because

  $\text{Composer} = \pi_{\text{CName}}(\text{Music\_Piece})$.

Step 5: Unnecessary Tables (2)

- The minimal cardinality 1 on the “Composer” side is important here.

  If there were also composers without music pieces, the “Composer” table would not be redundant. This is probably the only case where a $(1,*)$ cardinality is easier to translate than a $(0,*)$ cardinality.

- It is important that “Composer” had no other attributes besides the key.

  So it might have been better from the beginning to represent this entity type as an attribute. Something like “Day” or “Seat” would be a more typical example.

- This can be seen as a special case of table merging.

Step 5: Unnecessary Tables (3)

- However, operationally there is an important difference:
  - If there is an extra “Composer” table, new composers must be inserted explicitly.
  - Without this table, new composers are implicitly inserted with their music pieces.
  - Thus, typing errors and duplicates due to different spellings/shorthands will not be detected.
- Therefore, it makes sense to insist on the “Composer” table (design decision).
- But consider the minimum cardinality 1 which now requires a complicated constraint (choose 0).

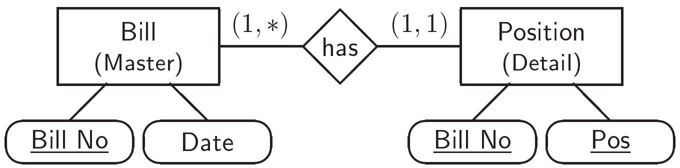
Step 6: Check

- At the end, check the generated tables to see whether they make sense.
- E.g. fill them with a few example rows.
- A correct translation applied to a correct ER-schema, will result in a correct relational schema.
- However, a by-hand translation can result in mistakes, and the ER-schema can contain hidden flaws.
- Think a last time about renaming tables or attributes.
- If two tables have the same key, consider merging them.
- You should also check the generated tables for a relational normal form (e.g. 3NF, BCNF, 4NF) (see below).

Overview

Weak Entities (1)

- Consider a solution without "weak entity":



- The standard translation results in

  Bill(<u>Bill_No</u>, Date)

  Position(<u>Bill_No</u>, <u>Pos</u>, . . . , Master_Bill_No → Bill)

- "Position" contains "Bill_No" twice: As part of its own key and in order to represent the relationship "has".

Weak Entities (2)

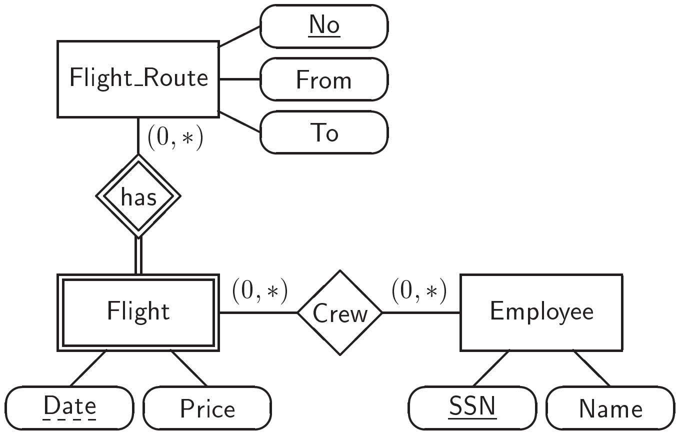
- Constraint: If a Bill B and a Position P are connected by "has", then B.Bill_No = P.Bill_No.
- So the two copies of "Bill_No" created by the general construction always contain equal values.
- Thus, the right translation in this case is

  Bill(<u>Bill_No</u>, Date)

  Position(<u>Bill_No</u> → Bill, <u>Pos</u>, . . .)

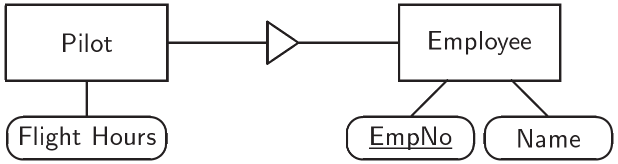
- A weak entity inherits the key attributes of every master (with a foreign key constraint), but the relationship to the master needs no extra representation.

Exercise



Specialization (1)

- A subset of the entities can have additional attributes:



- The relational model has no specialization mechanism.
- There are at least three different ways to handle this case, but none is perfect.

  Specialization is a feature that is missing in the relational model. It was added in object-relational models.

Specialization (2)

Method 1 (One Big Relation with Null Values):

- Employee(<u>EmpNo</u>, Name, is_Pilot, Flight_Hours^o)
- The attributes of the superclass and all subclasses are put into one relation.
- A boolean attribute "is_C" is added for every subclass C.
  Disjointness or totality specifications lead to constraints among the "is_C" attributes for different subclasses C. These are tuple-local (CHECK) constraints.
- "Flight_Hours" is not null if and only if "is_Pilot" is true.
  Constraints must enforce that attribute A is not null exactly when "is_C" is true for the subclass C containing A.
  In this example, the constraint makes "is_Pilot" redundant.

Specialization (3)

Advantages of Method 1:

- All information about an object is stored together (no join required).
- The number of tables is minimized (easier schema).
- This method is probably most often used in practice.
  But don't forget to specify the constraints!

Disadvantages of Method 1:

- Relationships defined only on a subclass (e.g. "can fly" between pilots and aircrafts) cannot be represented.
  This would require a general constraint. Trick: reserve certain ranges of employee numbers for certain jobs.

Specialization (4)

Variants of Method 1:

- In the example, the attribute "is_Pilot" is not really needed. A non-null value for "FlightHours" could be used to indicate that this is a pilot.
  Then the result is the same as with an optional attribute "FlightHours" for the entity type "Employee".
- If the subclasses are disjoint, a single attribute "Job" instead of "is_Pilot", "is_Technician" etc. can be used.

Specialization (5)

Method 2 (Different Relations for Subclasses):

- Normal_Employee(<u>EmpNo</u>, Name)
  Pilot(<u>EmpNo</u>, Name, Flight_Hours)
- This method is applicable only if subclasses are disjoint.
- One relation is created for every subclass. This relation also contains the attributes inherited from the superclass.
  If the specialization is not total, one additional relation is needed to store the entities in the superclass which do not belong to any subclass (here: "Normal_Employee").
- Requires the constraint that key values (EmpNo) in the relations are disjoint (e.g. 9xxx only for Pilots).

Specialization (6)

Advantages of Method 2:

- Method 2 makes a union necessary to get all employees together, but this union is fast since it needs no duplicate elimination. It is possible to define a view

  $\text{Employee} := \text{Normal\_Employee} \cup \pi_{\text{EmpNo,Name}}(\text{Pilot})$
- Method 2 is especially efficient if we often need to access only the pilots (including their name).

Disadvantages of Method 2:

- Cannot represent relationships defined on the superclass.

  This would require a "generalized foreign key constraint" referring to a union.

Specialization (7)

Method 3 (Splitting Attributes over Different Relations):

- Employee(<u>EmpNo</u>, Name)
  Pilot(<u>EmpNo</u> → Employee, Flight_Hours)
- This method creates one relation for every subclass which contains only the key and the attributes specific to this subclass. In addition, there is one relation for the superclass.
- So an entity of the subclass is now represented by two tuples: The "Employee" tuple which contains the general attributes (like Name) and the "Pilot" tuple which contains the added attributes (Flight_Hours).
- A join is needed to get all "Pilot" attributes together.

Specialization (8)

Advantages of Method 3:

- Relationships which are defined only on the subclass and relationships which are defined on the superclass can both be represented.

  This method is the only among the three without any restrictions.

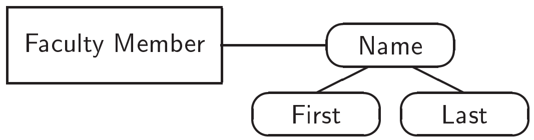
Disadvantages of Method 3:

- The join in Method 3 costs processing time.

  Unless the pilot-specific information is seldom accessed and quite large. In this case, factoring it out would actually speed up queries on employees.

Structured Attributes (1)

- Example: Attribute "Name" consisting of first and last name:



- The relational model has no structured attributes.
- Create separate columns for first name and last name (the smallest units needed in queries/programs).
- It might be tempting to create only one column "Name" and to use a format which easily allows to cut the string into two pieces, e.g. "Last, First".

Structured Attributes (2)

- With the string functions of Oracle SQL, it is possible to select last name and first name out of a "Name" column.

  Last Name: `substr(Name, 1, instr(Name, ',')-1)`

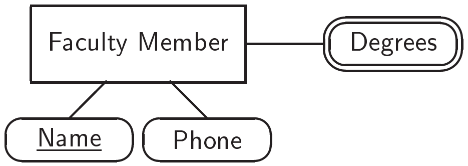
  First Name: `substr(Name, instr(Name, ',')+1)`.
- However, this makes programs non-portable: Current systems have different data type functions.

  The SQL86/89 standard contained only +, −, ∗, /.
- Indexes cannot be created on parts of attributes, e.g. on the last name only. Queries will probably be slower.
- All table entries must adhere to the "Last Name, First Name" format (requires constraint).

  What about John Smith, Jr.?

Multivalued Attributes (1)

- Example: The attribute "Degrees" can have several values:



- The relational model has only single-valued attributes.
- An extra table should be created for each multi-valued attribute (containing a row for every attribute value):

  Faculty_Member(<u>Name</u>, Phone)

  Degrees(<u>Name</u> → Faculty_Member, <u>Degree</u>)

Multivalued Attributes (2)

- Again, it is tempting to put all attribute values into one string with a specific delimiting character.
- In this case, it is not possible to reconstruct the single components unless the maximum number of components is known.

  And queries will become really complicated if this maximum number is larger than 2 or 3.
- Some queries can be formulated with the `like` operator in SQL, but there are other sensible queries which cannot be formulated in SQL.

  So unless it is exactly known how the multivalued attribute will be used in queries, this translation should be avoided.

Overview

Introduction (1)

- Relational database design theory is based mainly on a class of constraints called "Functional Dependencies" (FDs).
- This theory defines when a relation is in a certain normal form (e.g. Third Normal Form, 3NF) for a given set of FDs.
- It is usually bad if a schema contains relations which violate the conditions of a normal form.

  However, there are exceptions and tradeoffs.
- If a normal form is violated, data is stored redundantly, and information about different concepts is intermixed.
- This leads to several problems, especially insert, update, and deletion anomalies (see below).

Introduction (2)

- Today, Third Normal Form (3NF) is considered part of general database education.
- Boyce-Codd Normal Form (BCNF) is slightly stronger, easier to define, and better matches intuition.

  BCNF should really replace 3NF.
- A good Entity-Relationship design results in relations which automatically conform to BCNF (even 4NF).
- Awareness of normal forms can help to detect problems (design errors) already on the ER-level.
- Only BCNF is defined here.
- If a table is in BCNF, it is automatically in 3NF.

Functional Dependencies (1)

- Functional dependencies are generalizations of keys.
- A key uniquely determines every attribute, i.e. the FDs "CRN→TITLE", "CRN→INAME", "CRN→PHONE" hold here:

COURSES

<u>CRN</u>	TITLE	INAME	PHONE
22268	DB	Brass	9404
42232	DS	Brass	9404
31822	IS	Spring	9429

- Also the FD "INAME → PHONE" is satisfied: If two rows agree in the name, they must have the same phone number.

Functional Dependencies (2)

- An FD has the form

$$A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m.$$

- Formally, both sides of an FD are sets of attributes:

$$\{A_1, \ldots, A_n\} \rightarrow \{B_1, \ldots, B_m\}.$$

  So sequence and multiplicity of attributes are unimportant.
- In discussing FDs, the focus is on a single relation R.

  All attributes A_i, B_i are from this relation.
- $A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m$ holds for a relation R in a database state $\mathcal{S}$ if and only if for all tuples $t, u \in \mathcal{S}(R)$:

  If $t.A_1 = u.A_1$ and ... and $t.A_n = u.A_n$,

  then $t.B_1 = u.B_1$ and ... and $t.B_m = u.B_m$.

Functional Dependencies (3)

- I.e. $A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m$ means that whenever two rows contain the same values in all columns A_i, they must also contain the same values in all columns B_j.
- For instance, the FD "INAME $\rightarrow$ TITLE" does not hold: There are two rows with INAME "Brass", but different titles.
- An FD $A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m$ is equivalent to the FDs:

$$\begin{array}{ccc} A_1, \ldots, A_n & \rightarrow & B_1 \\ \vdots & \vdots & \vdots \\ A_1, \ldots, A_n & \rightarrow & B_m. \end{array}$$

- Thus, it suffices to consider FDs with a single column on the right hand side.

FDs vs. Keys

- FDs are a generalization of keys: $A_1, \ldots, A_n$ is a key of

  $$R(\underline{A_1}, \ldots, \underline{A_n}, B_1, \ldots, B_m)$$

  if and only if the FD "$A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m$" holds.

  Under the assumption that there are no duplicate rows.
- FDs are keys for some columns: E.g. "INAME $\rightarrow$ PHONE" means that INAME is a key of $\pi_{\mathtt{INAME,PHONE}}(\mathtt{COURSES})$:

INAME	PHONE
Brass	9404
Spring	9429

In general, $A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m$ holds if and only if $\pi_{A_1,\ldots,A_n,B_1,\ldots,B_m}(R)$ has no two rows with the same A_i-values (after duplicate elimination): These would have to differ in a B_j.

FDs Describe Functions

- A key means that a relation (a table) describes a function (with values for the key attributes as inputs and values for the other attributes as outputs).
- In the example, there is another function represented in the table which takes an instructor's name (INAME) as input and returns his/her phone number.
- Sometimes functions which can be computed:

  E.g. if the date of birth and the age are both stored.

  Then of course the FD "BIRTHDATE $\rightarrow$ AGE" holds.

  If two persons have the same birthdate, they must have the same age. But in this case, the exact formula is known, which is much more than what the FD expresses.

FDs are Constraints

- FDs are constraints: They must hold for all possible database states.
- In the example state, also "TITLE $\rightarrow$ CRN" holds (no two courses have the same title). But this is not true in general, only in this small example state.
- There are tools for analyzing example data for possible FDs, and then asking the designer whether these FDs hold in general.
- If an FD (or any constraint) does not hold in the example state, it certainly cannot hold in general.

Implication of FDs

- CRN→PHONE is nothing new when CRN→INAME and INAME→PHONE are already known.
  Whenever A → B and B → C are satisfied, A → C automatically also holds.
- PHONE→PHONE holds, but is not interesting.
  FDs of the form A → A always hold (for every DB state).
- A set of FDs $\{\alpha_1 \to \beta_1, \ldots, \alpha_n \to \beta_n\}$ implies an FD $\alpha \to \beta$ if and only if every database state which satisfies the $\alpha_i \to \beta_i$ for $i = 1, \ldots, n$ also satisfies $\alpha \to \beta$.
  α and β stand here for sets of attributes/columns.
- So only in a representative set of FDs should be specified, which implies all other FDs.

Exercise

- The following relation is used for storing orders:
  ORDER(ORD_NO, DATE, CUST_NO, PROD_NO, QUANTITY)
- Please list FDs which hold for this relation:
  One order can be about multiple products.
- Do these FDs imply the following FD?
  ORD_NO, PROD_NO → DATE
- Determine a key of the relation ORDER.

Problems (1)

- Tables should not contain an FD which is not implied by a key (e.g. INAME→PHONE).
- Among other problems, this leads to the redundant storage of certain facts. E.g. in the example table, the phone number of “Brass” is stored twice:

COURSES

CRN	TITLE	INAME	PHONE
22268	DB	Brass	9404
42232	DS	Brass	9404
31822	IS	Spring	9429

Problems (2)

- Redundant data requires a constraint to ensure that the two copies of this information agree.
- In the example, this constraint is precisely the FD “INAME→PHONE”.
- FDs are not supported as one of the standard constraints of the relational model.
- Avoid (proper) FDs by transforming them into key constraints. This is what normalization does.
- Redundant data leads to the Update Anomalies:
  Changing a single fact, e.g. the phone number of “Brass”, requires updating multiple tuples.

Problems (3)

- Storing information about two different entities (Course and Instructor) together in one table results in insertion and deletion anomalies.
- Insertion Anomalies: It is impossible to insert a new faculty member's phone number as long as he/she does not teach a course. (Since CRN is a key, we cannot fill it with "null").
- Deletion Anomalies: When the last course of a faculty member is deleted, his/her phone number is also lost.

  Even if null values were possible, it would be strange that all courses by an instructor can be deleted except the last. When the last should be deleted, the course data must instead be replaced by null values.

Boyce-Codd Normal Form

- A Relation R is in Boyce-Codd Normal Form (BCNF) if and only if all its FDs are already implied by key constraints.
- I.e. for every FD "$A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m$" for R one of the following conditions must hold:
  - The FD is trivial, i.e. $\{B_1, \ldots, B_m\} \subseteq \{A_1, \ldots, A_n\}$.
  - The FD follows from a key, because $\{A_1, \ldots, A_n\}$ or some subset of it is already a key of R.

    It can be any key, not necessarily the primary key.
- Thus, a relation in BCNF does not require FD constraints, only key constraints. The above-mentioned problems do not occur.

Examples (1)

- `COURSES(CRN, TITLE, INAME, PHONE)` with the FDs
  - `CRN` → `TITLE, INAME, PHONE`
  - `INAME` → `PHONE`

  is not in BCNF because the FD "`INAME` → `PHONE`" is not implied by a key:
  - "`INAME`" is not a key of the entire relation.
  - The FD is not trivial.
- However, without the attribute `PHONE` (and its FD), the relation is in BCNF:
  - `CRN` → `TITLE, INAME` corresponds to the key.

Examples (2)

- Suppose that each course meets only once per week and that there are no cross-listed courses. Then the relation

  `CLASS(CRN, TITLE, DAY, TIME, ROOM)`

  satisfies the following FDs (plus implied ones):
  - `CRN` → `TITLE, DAY, TIME, ROOM`
  - `DAY, TIME, ROOM` → `CRN`
- The keys are `CRN` and `DAY, TIME, ROOM`.
- Both FDs have a key on the left hand side, so the relation is in BCNF.

Examples (3)

- Consider the relation PRODUCT(NO, NAME, PRICE) with the following functional dependencies:

  (1) NO → NAME (3) PRICE, NAME → NAME

  (2) NO → PRICE (4) NO, PRICE → NAME

- This relation is in BCNF:
  - The first two FDs show that NO is a key. They have the key on the left hand side, so they are no problem.
  - The third FD is trivial and can be ignored.
  - The fourth FD has a superset of the key on the left hand side, which is also no problem.

Exercises

- Is RESULTS(STUD_ID, EX_NO, POINTS, MAX_POINTS) with the following FDs in BCNF? (First determine a key.)

  (1) STUD_ID, EX_NO → POINTS

  (2) EX_NO → MAX_POINTS

- Is the relation

  ORDER(ORD_NO, DATE, CUST_NO, PROD_NO, QUANTITY),

  for which you determined FDs above, in BCNF?

Splitting Relations

- A table which is not in BCNF can be split into two tables ("decomposition").
- E.g. split COURSES(CRN, TITLE, INAME, PHONE) into

  COURSES_NEW(CRN, TITLE, INAME)

  INSTRUCTORS(INAME, PHONE)

- In general, if $A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m$ violates BCNF, create a new relation $S(\underline{A_1}, \ldots, \underline{A_n}, B_1, \ldots, B_m)$ and remove $B_1, \ldots, B_m$ from the original relation.

  In unusual cases (multiple violations), it might be necessary to repeat the splitting step with one or both of the resulting relations. Then also implied FDs must be considered.