Datenbanken II A: DB-Entwurf

Chapter 6: Logical Design II

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Wintersemester 2024/25

http://www.informatik.uni-halle.de/~brass/dd24/

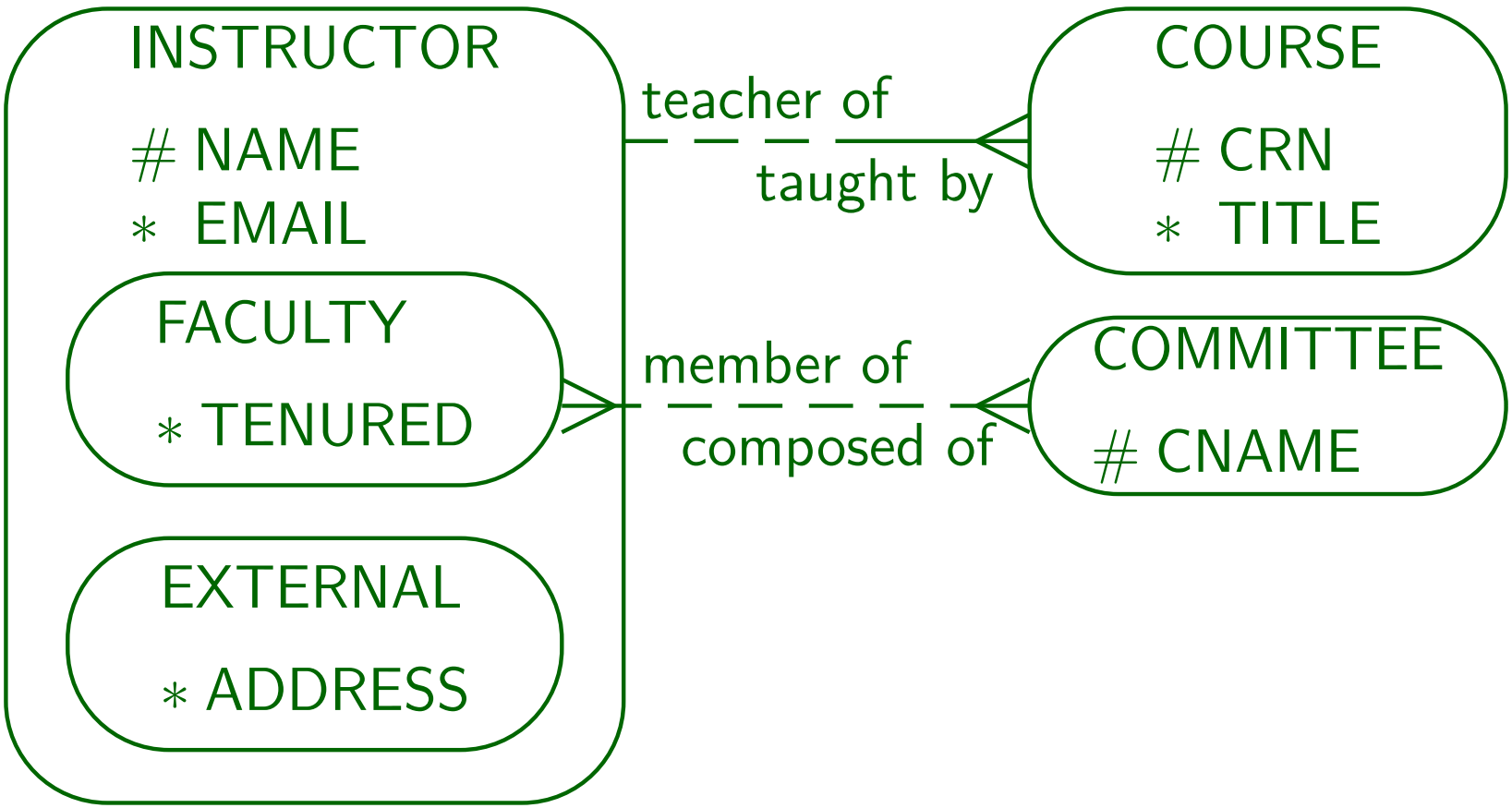
Objectives

After completing this chapter, you should be able to:

- translate given ER-schemas (including subclasses) manually into the relational model.
- explain and compare the alternatives for translating subclasses.

Contents

Subtypes/Specialization (1)



Subtypes/Specialization (2)

Method 1 (Table for the Supertype):

- One big relation is created that contains all attributes of the supertype and of all subtypes.

  Including possibly indirect subtypes.

- In the example, the result is:

  INSTRUCTORS(NAME, EMAIL, TYPE,
  TENURED°, ADDRESS°)

- The column "TYPE" identifies to which subtype the entity belongs, e.g. "F" for Faculty and "E" for External: CHECK(TYPE = 'F' OR TYPE = 'E').

Subtypes/Specialization (3)

- Example State:

INSTRUCTORS				
NAME	EMAIL	TYPE	TENURED	ADDRESS
Brass	sb@...	F	N	
Spring	spring@...	F	Y	
Mundie	mundie@...	E		CMU

- Attributes of subtypes are defined only for rows corresponding to elements of the subtype.
- This means that the corresponding columns in the table must permit null values.

Subtypes/Specialization (4)

- With the following constraints one can make sure that subtype attribute columns are really defined only for the subtype:

```
CHECK(TYPE = 'F' OR TENURED IS NULL)
CHECK(TYPE = 'E' OR ADDRESS IS NULL)
```

- Conversely, if an attribute was not optional in the ER-schema, one must add a CHECK-constraint to make sure that the corresponding column is not null for elements of this subtype:

```
CHECK(TYPE <> 'F' OR TENURED IS NOT NULL)
CHECK(TYPE <> 'E' OR ADDRESS IS NOT NULL)
```

Subtypes/Specialization (5)

- Such constraints can be developed by thinking in "if-then" rules:

  If `TYPE = 'F' then TENURED IS NOT NULL`

- Since SQL has no "if-then" condition, one must use the equivalence of $A \rightarrow B$ to $\neg A \vee B$:

  ```
  NOT (TYPE = 'F') OR TENURED IS NOT NULL
  ```

- This can be simplified to

  ```
  TYPE <> 'F' OR TENURED IS NOT NULL
  ```

  Since there are only two types in the example, `TYPE <> 'F'` is equivalent to `TYPE = 'E'`.

Subtypes/Specialization (6)

- It might be simpler to all constraints in a single formula in DNF (disjunctive normal form) with one case per subclass:

```
CHECK(    (TYPE = 'F' AND TENURED IS NOT NULL
                      AND ADDRESS IS NULL)
       OR (TYPE = 'E' AND TENURED IS NULL
                      AND ADDRESS IS NOT NULL))
```

The parentheses inside the formula are not needed, since `AND` binds stronger than `OR`, but they might improve the readability.

If an attribute is optional in a subclass, the corresponding `IS NOT NULL` condition is simply left out in the case for that subclass.

Subtypes/Specialization (7)

- It might be useful to declare views for the subtypes:

```
CREATE VIEW FACULTY AS
            SELECT NAME, EMAIL, TENURED
            FROM   INSTRUCTORS
            WHERE  TYPE = 'F'
```

- Sometimes, the "TYPE" column is not really needed.

  E.g. in the example, all instructors where "TENURED" is a null value are external instructors.

- But it might be clearer to retain it. This might also help to adapt the schema to additional subtypes.

Subtypes/Specialization (8)

- With this method, relationships refering to the supertype are no problem:

  COURSES(CRN, TITLE, INST_NAME→INSTRUCTOR)

- Example State:

COURSES		
CRN	TITLE	INST_NAME
11111	Database Management	Brass
22222	DB Analysis&Design	Brass
33333	Client-Server	Spring
44444	Document Processing	Mundie

Subtypes/Specialization (9)

- Relationships with a subtype can only be translated in the same way as a relationship to the supertype:

COMMITTEE_MEMBERS(CNAME→COMMITTEES,
FAC_NAME→INSTRUCTOR)

COMMITTEE_MEMBERS	
CNAME	FAC_NAME
PhD Admissions	Spring
PhD Admissions	Brass

- The table declaration does not prevent that an external instructor is entered as a committee member.

Subtypes/Specialization (10)

- The standard constraints of the relational model do not help in this case.

  As mentioned before, one can run a query that finds violations from time to time, one can do checks in application programs or stored procedures, or one can use triggers. Note that a foreign key cannot reference a view. One can hope that in future DBMS vendors will implement more general constraints. In this case one needs something like a foriegn key that specifies in addition a condition on the referenced tuple.

- If there are relationships on subclasses, one should consider using one of the other translation methods (or do the trick on the next page).

Subtypes/Specialization (11)

- In the special case that one uses artificial keys (i.e. numbers that one can assign), one can reserve different ranges for the different subtypes.
- E.g. faculty members have IDs from 100 to 499, external instructs have IDs from 500 to 999:

INSTRUCTORS					
ID	NAME	EMAIL	TYPE	TENURED	ADDRESS
101	Brass	sb@...	F	N	
102	Spring	spring@...	F	Y	
501	Mundie	mundie@...	E		CMU

Subtypes/Specialization (12)

- The column "TYPE" should now be removed, since it is redundant.

  Of course, one can define a view that reconstructs it. If one really wants to retain it, one must add at least a CHECK constraint that ensures that IDs are in the correct range for the instructor type.

- Some designers would leave part of the possible range of IDs for future subtypes.

Subtypes/Specialization (13)

- Now relationships defined on subtypes are no problem. Consider again:

COMMITTEE_MEMBERS(CNAME→COMMITTEES,
FAC_ID→INSTRUCTOR)

COMMITTEE_MEMBERS	
CNAME	FAC_ID
PhD Admissions	101
PhD Admissions	102

- This constraint ensures that only the subtype is referenced: `CHECK(FAC_ID BETWEEN 100 AND 499)`

Subtypes/Specialization (14)

- This method can be easily adapted for partial or overlapping specialization:
  - If specialization is partial, one simply has one more `TYPE` value for elements of the supertype that do not belong to any subtype.

    Actually, partial specialization is never a problem: One can always add an "Other" subclass.
  - If specialization is overlapping, one uses instead of the `TYPE` column one boolean column for each subtype (e.g. `IS_FACULTY`, `IS_EXTERNAL`).

Subtypes/Specialization (15)

Method 2 (Tables for the Subtypes):

- In this case, one table is created for each subtype.
  It contains the attributes of the subtype plus all inherited attributes.
- In the example, the result is:

  FACULTY(NAME, EMAIL, TENURED)
  EXTERNAL(NAME, EMAIL, ADDRESS)

- Since each entity of the supertype belongs to only one subtype, no data is stored redundantly.

  This method would not work for overlapping specialization.

Subtypes/Specialization (16)

- Example State:

FACULTY		
NAME	EMAIL	TENURED
Brass	sb@...	N
Spring	spring@...	Y

EXTERNAL		
NAME	EMAIL	ADDRESS
Mundie	mundie@...	CMU

- This method does not need null values and the corresponding CHECK-constraints like Method 1.

Subtypes/Specialization (17)

- One can define a view for the supertype:

```
CREATE VIEW INSTRUCTOR(NAME, EMAIL) AS
            SELECT NAME, EMAIL FROM FACULTY
            UNION  ALL
            SELECT NAME, EMAIL FROM EXTERNAL
```

  Without the view, queries will often be more complicated than with the first method. In any case, queries refering to the supertype will run a bit slower, although `UNION ALL` is only concatenation.

- Queries refering only to a subtype are slightly simpler and will run slightly faster than with Method 1.

  If there are subtypes that contain only a small fraction of the entities of the supertype, queries to these subtypes will be significantly faster.

Subtypes/Specialization (18)

- This method cannot enforce the uniqueness of keys between subtypes: E.g. a faculty member and an external instructor with the same name can exist.

  The constraint that the values in the `NAME` columns of the tables `FACULTY` and `EXTERNAL` must be disjoint is not one of the standard constraints and cannot be specified (today) in the `CREATE TABLE` statement.

- If one can assign numbers as key values, one can use `CHECK` constraints that enforce that the key value ranges in the two tables are disjoint.

  E.g. `FACULTY` uses only IDs 100 to 499, `EXTERNAL` only 500 to 999.

Subtypes/Specialization (19)

- For Method 2, relationships with a subtype are no problem (since each subtype has its own table):

COMMITTEE_MEMBERS(CNAME→COMMITTEES,
FAC_NAME→FACULTY)

COMMITTEE_MEMBERS	
CNAME	FAC_NAME
PhD Admissions	Spring
PhD Admissions	Brass

- However, the translation of relationships with a supertype is significantly more complicated.

Subtypes/Specialization (20)

- Since there is no table for the supertype, one must split foreign keys that are generated for relationships with the supertype:

$$\texttt{COURSES(}\underline{\texttt{CRN}}\texttt{, TITLE, FAC\_NAME}^{o}\!\!\rightarrow\!\texttt{FACULTY,}$$
$$\texttt{EXT\_NAME}^{o}\!\!\rightarrow\!\texttt{EXTERNAL)}$$

COURSES			
CRN	TITLE	FAC_NAME	EXT_NAME
11111	Database Management	Brass	
22222	DB Analysis&Design	Brass	
33333	Client-Server	Spring	
44444	Document Processing		Mundie

Subtypes/Specialization (21)

- Only one of the two foreign keys can be defined:

```
CHECK(FAC_NAME IS NULL OR EXT_NAME IS NULL)
```

- In addition, one must be defined (because the relationship has mandatory participation):

```
CHECK(FAC_NAME IS NOT NULL
      OR EXT_NAME IS NOT NULL)
```

- Queries become more complicated in this way.

  It would be possible to hide these complications with another view defined for `COURSES` that merges the two columns (using `UNION ALL`). But in any case, query evaluation will be slower (with today's query optimizers). Of course, if the tables are small, this is no problem.

Subtypes/Specialization (22)

- When the foreign key would be part of a primary key (for many-to-many relationships or weak entities), there are two options:
  - Either one uses the splitting of foreign keys as above and accepts null values in keys: This translation works only for some DBMS.

    DBMS differ in whether they support `UNIQUE`-constraints for columns that can be null, and in the exact semantics for this. One would need here that only exact copies are excluded. If necessary, one could replace the null value by a single "invalid" faculty member or external instructor.

Subtypes/Specialization (23)

- Translation of many-to-many and weak entity relationsships, continued:
  - Or one splits the entire table: E.g. suppose that instructors can suggest students for awards (i.e. there is a many-to-many relationship between instructors and students).

    AWARD1(NAME→FACULTY, SSN→STUDENTS)
    AWARD2(NAME→EXTERNAL, SSN→STUDENTS)

- Because of these problems, one would probably use one of the other methods for translating specialization in this case.

Subtypes/Specialization (24)

- Method 2 can work also with partial specialization.

  The trick is to add another subclass and works with any method.

- E.g. if there are instructors that are neither faculty members nor external (e.g. PhD students), one would simply add another table for them:

  ```
  FACULTY(NAME, EMAIL, TENURED)
  EXTERNAL(NAME, EMAIL, ADDRESS)
  OTHER_INSTRUCTORS(NAME, EMAIL)
  ```

- The `OTHER_INSTRUCTORS` table contains only those entities that are direct instances of the supertype, it does not contain the subtype entities.

Subtypes/Specialization (25)

Method 3 (Tables for Supertype and Subtypes):

- Method 3 creates
  - a table for the supertype that contains all entities, including those of subtypes, but has only columns for the supertype attributes, and
  - one table for each subtype which contains columns for the attributes that are specific to the subtype, plus the key of the supertype.

Subtypes/Specialization (26)

- In the example, the result is:

```
INSTRUCTORS(NAME, EMAIL)
FACULTY(NAME→INSTRUCTORS, TENURED)
EXTERNAL(NAME→INSTRUCTORS, ADDRESS)
```

- One must use a join to get all attributes of an entity together (the same entity is now represented in two different tables):

```
CREATE VIEW FACULTY2(NAME, EMAIL, TENURED) AS
            SELECT I.NAME, I.EMAIL, F.TENURED
            FROM   INSTRUCTORS I, FACULTY F
            WHERE  I.NAME = F.NAME
```

Subtypes/Specialization (27)

- Example State:

INSTRUCTORS	
NAME	EMAIL
Brass	sb@...
Spring	spring@...
Mundie	mundie@...

FACULTY	
NAME	TENURED
Brass	N
Spring	Y

EXTERNAL	
NAME	ADDRESS
Mundie	CMU

Subtypes/Specialization (28)

- For Method 3, relationships defined on the supertype and relationships defined on the subtypes are both no problem.
- A problem of this method is that it really supports only partial, overlapping specialization.

  Nothing prevents that instructors are also entered in one or both of the two subtype tables (needs a general constraint). With key value ranges, at least disjoint specialization can be enforced.

- Also the join can be a performance problem.

  If one uses artificial numbers as keys, the join will be basically always necessary whenever one accesses the subtype.

Subtypes/Specialization (29)

Method 4 (Variant of Method 3 Using an "Arc"):

- Method 4 creates a table for the supertype and one table for each subtype (like Method 3).
- Artificial keys are added to the subtype tables.
- Foreign keys are added to the supertype table (one for each subtype).
- So the direction of the foreign keys is the real difference to Method 3.

  In Method 3, they point from the subclass tables to the superclass table, here from the superclass table to sublass tables.

Subtypes/Specialization (30)

- In the example, the result is:

```
INSTRUCTORS(NAME, EMAIL,
            FNO°→FACULTY, ENO°→EXTERNAL)
FACULTY(FNO, TENURED)
EXTERNAL(ENO, ADDRESS)
```

- Check constraints are needed to ensure that exactly one of the two columns FNO and ENO are defined (not null) in INSTRUCTORS.

  By adapting this constraint, Method 4 also works with partial or overlapping specialization.

- In this way, the problem of Method 3 is avoided.

Subtypes/Specialization (31)

- Relationships on supertype and subtypes can be represented.

  Although it is a bit strange that relationships defined on the subtypes now have to use the artificial numbers.

- This method does not prevent rows in the subtype tables without entry in the supertype table.

  Such rows are meaningless: One does not even have the name of the instructor. One possibility would be to treat such rows as "not really present". Practically all queries have to join the subtype tables with the supertype table, and then the problematic rows are filtered out. From time to time, one can simply remove such rows. The drawback of this solution is that one does not get an error message if one enters such a row. But if all queries do the join, bad rows are never used.

Subtypes/Specialization (32)

Comparison:

- Method 1 is probably most often chosen, but:
  - If one cannot assign key value ranges, and there are relationships with subtypes, it does not work.
  - The many null values might be a problem.

    Real world designers are used to null values. One should not leave out the `CHECK`-constraints that restrict them.
  - If small subtypes (few rows) of a large supertype (many rows) are accessed often, Method 1 might have a performance problem.

    Powerful DBMS offer partition features that solve this problem.

Subtypes/Specialization (33)

- Method 2 is good when one accesses the subtypes often, but:
  - Relationships with the supertype are a problem, especially if these are many-to-many relationships or weak entity relationships.
  - Uniqueness of keys cannot be enforced between subtypes unless one can assign key value ranges.
  - Some people don't like `UNION` in their queries.

    It is a bit uncommon, but one can hide it in views. `UNION ALL` should really run fast. Modern optimizers should be able to work with it, old might produce not very efficient query execution plans.

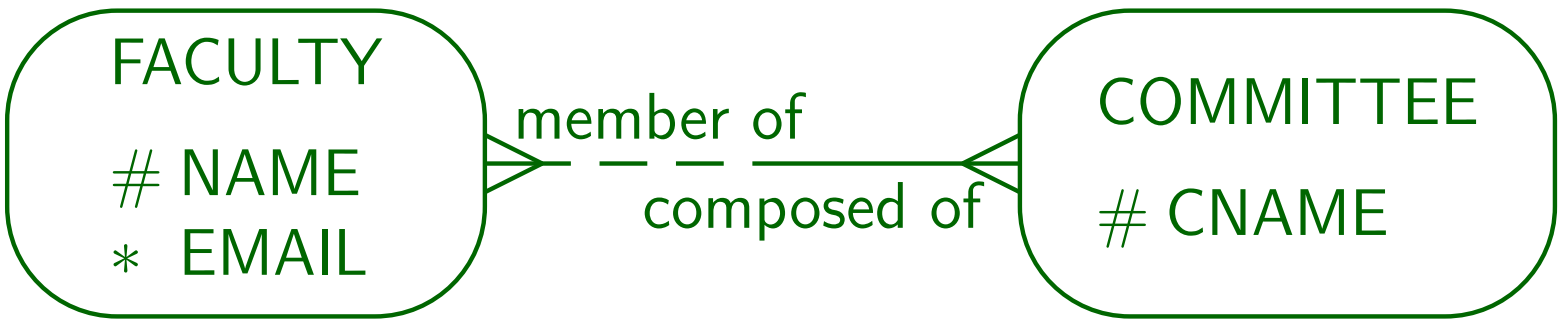
Subtypes/Specialization (34)

- Method 3 can easily represent relationships on supertypes and subtypes, but:
  - This method works only for partial specialization.
  - The joins are a performance problem.
- Method 4 is similar, and also has problems:
  - Integrity violations are possible (partial entity data), but the invalid data is never used.
  - Joins are needed as in Method 3.
- There is no perfect solution!

Contents

Unnecessary Tables (1)

- Sometimes, tables generated for entity types might seem unnecessary. E.g. consider this example:



- The translation result is:

```
FACULTY(NAME, EMAIL)
COMMITTEES(CNAME)
COMMITTEE_MEMBERS(CNAME→COMMITTEE,
                  FAC_NAME→FACULTY)
```

Unnecessary Tables (2)

- The entire contents of the table COMMITTEES can be derived from the table COMMITTEE_MEMBERS:

```
SELECT DISTINCT CNAME
FROM   COMMITTEE_MEMBERS
```

- This works because of the mandatory participation of COMMITTEE in the relationship.

  Therefore, all committee names must be present in COMMITTEE_MEMBERS.

- It is also important in this example that the entity type COMMITTEE has only the key attributes, and no additional information.

Unnecessary Tables (3)

- Formally, the table `COMMITTEES` is indeed redundant and one must discuss to delete it.
- However, deleting the table changes the behaviour of updates:
  - With the table, COMMITTEE entities are explicitly created by inserting a row into `COMMITTEES`.
  - Without the table, COMMITTEE entities are only implicitly created by inserting a member of a new committee.

Unnecessary Tables (4)

- Therefore, when inserting a committee member, a typing error in the committee name would be detected with the table, but maybe not without it.
- However, this also depends on the application program: Even without the table, one could distinguish
  - Create a new committee and add its first member (e.g. the chairman).
  - Add a member to a committee (with all currently existing committees shown in a selection box).

Unnecessary Tables (5)

- With the COMMITTEES table, one has the problem how to enforce the mandatory participation (see above).
- The entire problem would vanish if it turns out that
  - there can be committees without members (at least temporarily or in exceptional situations), or
  - some other information has to be stored about committees.

    It would be even interesting if such changes in the requirements can be expected for future extensions.
- Again, there is no unique, perfect solution.

Final Step: Check (1)

- At the end, one should check the generated tables to see whether they really make sense.
- E.g. one should fill them with a few example rows.

  This is also a useful part of the documentation.

- A correct translation of a correct ER-schema results in a correct relational schema.
- However, a by-hand translation can result in mistakes, and the ER-schema can contain hidden flaws.

Final Step: Check (2)

- Think a last time about renaming tables/columns.

  Later changes will be difficult: The table/column names are already used in the application programs, and the DBMS might not permit to rename tables or columns (without deleting and recreating them).

- Check for normal forms (see Chapter 8).

  This is not an automatic step: It requires that the designer thinks about possible functional dependencies.

- If there are tables with the same key, one might consider to merge them.

  But this is not always the right thing to do: E.g. Methods 2–4 for translating specialization generate such tables, merging them would move back to Method 1.

References

- Teorey: Database Modeling & Design, 3rd Edition.
  Morgan Kaufmann, 1999, ISBN 1-55860-500-2, ca. $32.
- Elmasri/Navathe: Fundamentals of Database Systems, 3rd Ed.
- Rauh/Stickel: Konzeptuelle Datenmodellierung (in German), Teubner, 1997.
- Kemper/Eickler: Datenbanksysteme (in German), Oldenbourg, 1997.
- Graeme C. Simsion, Graham C. Witt: Data Modeling Essentials, 2nd Edition.
  Coriolis, 2001, ISBN 1-57610-872-4, 459 pages.
- Barker: CASE*Method, Entity Relationship Modelling.
  Addison-Wesley, 1990, ISBN 0-201-41696-4, ca. $61.
- Koletzke/Dorsey: Oracle Designer Handbook, 2nd Edition.
  ORACLE Press, 1998, ISBN 0-07-882417-6, ca. $40.
- A. Lulushi: Inside Oracle Designer/2000.
  Prentice Hall, 1998, ISBN 0-13-849753-2, ca. $50.
- Oracle/Martin Wykes: Designer/2000, Release 2.1.1, Tutorial.
  Part No. Z23274-02, Oracle, 1998.
- Oracle Designer Model, Release 2.1.2 (Element Type List).
- Oracle Designer Online Help System.
- Lipeck: Skript zur Vorlesung Datenbanksysteme (in German), Univ. Hannover, 1996.