Part 4: Logical Design

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.

Objectives

After completing this chapter, you should be able to:

- translate given ER-schemas manually into the relational model (most important goal of this chapter).
- explain the steps in which a database schema is developed with Oracle Designer and name the tools that are used in this process.
- write a short paragraph about the Database Design Transformer of Oracle Designer: What it can do and what its limitations are.
- read Server Model Diagrams in Oracle Designer.

Overview

General Remarks (1)

- In order to develop a relational schema, one usually first designs an ER-schema, and then transforms it into the relational model, because the ER-model
  - ◇ allows better documentation of the relationship between the schema and the real world.

    E.g. entity types and relationships are distinguished.
  - ◇ has a useful graphical notation.
  - ◇ 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_E, the goal is to construct a relational schema S_R such that there is a one-to-one mapping τ between the states for S_E and S_R.

  I.e. each possible DB state with respect to S_E has exactly one counterpart state with respect to S_R 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., in the ER-model, relationships can be always only between currently existing entities. In the relational model, "dangling pointers" must be explicitly excluded by means of foreign key constraints.

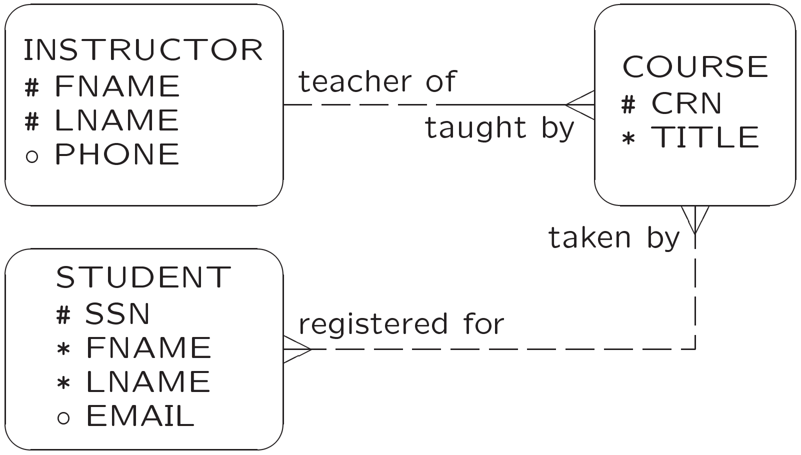
General Remarks (3)

- In addition, it must be possible to translate queries referring to S_E into queries with respect to S_R, evaluate them in the relational system, and then translate the answers 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



Entities Types (1)

- First a table is created for each entity type.

  The tables created in this step are not necessarily the final result. When one-to-many relationships are translated, columns are added to them. In rare cases, they will later turn out as unnecessary.

- The name of this table is the name of the entity type (maybe in plural form, as in Oracle Designer).

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

  Optional attributes translate into columns that permit null values. Depending on how much one considers the goal DBMS in this step, it might be necessary to map attribute data types into something the DBMS supports.

Entities Types (2)

- The primary key of the table is the primary key of the entity type. The same for alternative keys.

  Weak entity types are discussed below.

- If the entity type has no key, an artificial key is added (e.g. Oracle Designer does this).

  The designer really should explicitly define a key for each entity type.

- Result in the example:

  INSTRUCTORS(FNAME, LNAME, PHONE^o)
  STUDENTS(SSN, FNAME, LNAME, EMAIL^o)
  COURSES(CRN, TITLE)

Entity Types (3)

Example State for the Tables Generated So Far:

INSTRUCTORS

FNAME	LNAME	Phone
Stefan	Brass	624-9404
Michael	Spring	624-9424
Nina	Brass	

Entity Types (4)

COURSES

CRN	TITLE
12345	DB Management
24816	DB Analysis&Design
56789	Client-Server

STUDENTS

SSN	FIRST	LAST	EMAIL
111-22-3333	John	Smith	js@acm.org
123-45-6789	Ann	Miller	
235-71-1131	David	Meyer	dm@hotmail.com

One:Many Relationships (1)

- One-to-many Relationships are normally translated by adding the primary key from the "one" side as a foreign key to the "many" side.

  In this way, every entity on the "many" side can refer to the related entity on the "one" side.

- E.g. in the example, first name and last name of the instructor are added to the course table in order to implement the relationship "teacher of/taught by":

  COURSES(CRN, TITLE,
  (FNAME,LNAME)→INSTRUCTORS)

One:Many Relationships (2)

- The example shows already a difficult case because the primary key (and therefore also the foreign key) consists of two columns.

  This is why some designers would prefer primary keys consisting only of one column. But that is a matter of taste.

- Example State:

COURSES			
CRN	TITLE	FNAME	LNAME
12345	DB Management	Stefan	Brass
24816	DB Analysis&Design	Stefan	Brass
56789	Client-Server	Michael	Spring

One:Many Relationships (3)

- The rows corresponding to both entities will be combined with a join (which equates the foreign key on the "many" side to the primary key on the "one" side).
- Although a "pointer" (foreign key) was added only on the "COURSES" side, the join permits to "follow pointers in both directions".

  Of course, one can formulate queries that contain conditions on instructors and then find all their courses. The exact evaluation sequence for the query is a question of query optimization and depends also on the existing indexes.

One:Many Relationships (4)

- It is a common error of beginners to add the foreign key to the wrong side.

  Of course, this cannot happen when one uses a tool that does the translation automatically (like Oracle Designer). But one nevertheless needs to understand the correct translation.

- Adding a foreign key to the table is only possible if the maximum cardinality in the (min,max) notation is 1, i.e. there is at most one related entity.
- This holds for the "many" side of a one-to-many relationship.

One:Many Relationships (5)

- Since one instructor can teach many courses, adding the key of COURSES to the INSTRUCTORS table would give a set-valued attribute which is not permitted in the standard relational model:

INSTRUCTORS **WRONG!**			
FNAME	LNAME	Phone	CRN
Stefan	Brass	624-9404	{12345, 24816}
Michael	Spring	624-9424	{56789}
Nina	Brass		∅

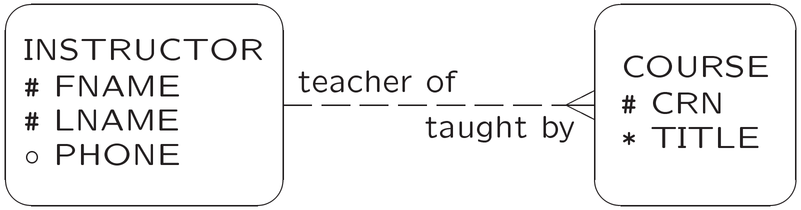
One:Many Relationships (6)

- Unfolding the set-valued attribute would destroy the key, store information redundantly (instructors of multiple courses), and lead to the loss of other information (instructors of no course).

INSTRUCTORS **WRONG!**			
<u>FNAME</u>	<u>LNAME</u>	Phone	CRN
Stefan	Brass	624-9404	12345
Stefan	Brass	624-9404	24816
Michael	Spring	624-9424	56789

One:Many Relationships (7)

- Above, every course had to be taught by an instructor (mandatory participation).
- The translation for the case of optional participation is similar (courses without instructors).



One:Many Relationships (8)

- The only difference is that the foreign key can now be null:

  COURSES(<u>CRN</u>, TITLE,
  (FNAME^o,LNAME^o)
  → INSTRUCTORS)

- Example State:

COURSES			
<u>CRN</u>	TITLE	FNAME	LNAME
12345	DB Management	Stefan	Brass
24816	DB Analysis&Design		
56789	Client-Server	Michael	Spring

One:Many Relationships (9)

- If the foreign key consists of more than one attribute (as in the example), all its attributes must be together null or together not null.

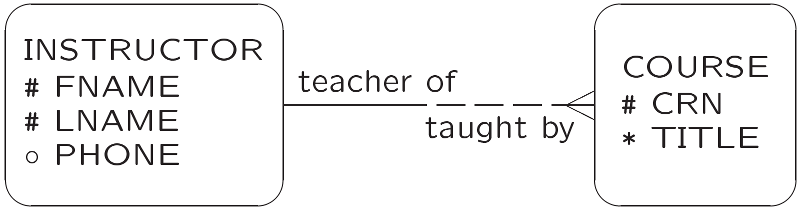
  A partially defined foreign key would make no sense in terms of the relationship that has to be implemented.

- Fortunately, this condition can be enforced declaratively with a `CHECK`-constraint:

```
CHECK((FNAME IS NOT NULL AND LNAME IS NOT NULL)
    OR (FNAME IS NULL AND LNAME IS NULL))
```

One:Many Relationships (10)

- Mandatory participation on the "one" side of a one-to-many relationship cannot be translated into the relational model using only the standard constraints (not null, keys, foreign keys, `CHECK`).
- Instructors must teach at least one course:



One:Many Relationships (11)

- In this case, one uses the same translation as if the participation on the " INSTRUCTOR" side would be optional.
- This is more general: The cardinality restriction $(1, *)$ is weakend to $(0, *)$.
- Thus, all DB states required by the ER-schema can be represented in the relational schema.
- But the relational schema permits DB states that would be illegal with respect to the ER-schema.

One:Many Relationships (12)

- In order to make the two schemas equivalent, one needs to add a constraint that excludes instructors without courses.
- E.g. one could run from time to time an SQL query that finds violations of the constraint:

```
SELECT FNAME, LNAME
FROM   INSTRUCTORS I
WHERE  NOT EXISTS (SELECT *
                   FROM   COURSES C
                   WHERE  C.FNAME = I.FNAME
                   AND    C.LNAME = I.LNAME)
```


Integrity Control (1)

- The problem with the above approach (searching for violations e.g. every night) is that it does not really enforce the integrity of the DB state.
- The invalid information can be entered, and is detected only after some time.
- In the meantime, it might have been used already.

  E.g. a salary was paid.
- It is also more difficult to correct the integrity violation if it is not immediately detected.

  Who has entered this? What did he/she meant to do?

Integrity Control (2)

- One can also program the check in the application programs used for entering data.

  The instructor can only be added with his/her first course, and when the last course is deleted, the instructor is deleted, too.

- Then one has to exclude direct changes to the database that do no use the application programs.
- Also, one must be very careful that all application programs check this condition.

  E.g. also the one used for updating instructor assignments for courses.

Integrity Control (3)

- Triggers can be used to enforce the constraint in a more reliable way.

  Triggers are procedures stored in the database that the DBMS automatically calls when a certain event has happend, e.g. when an instructor was inserted. Triggers often consist of the three parts "event, condition, action".

- One can also define elementary transactions as stored procedures in the database and change the DB state only via these stored procedures.

  Then checks do not have to be repeated in the application programs, it is more likely that checks are not forgotten, and they are more clearly separated from the user interface.

Integrity Control (4)

- The SQL-92 standard would permit to specify the constraint declaratively ("`CREATE ASSERTION`").

  This is not implemented in any DBMS I know. However, DBMS vendors now feel some pressure from their customers to offer more support for integrity enforcement.

- The constraint that would be needed here is similar to a foreign key.

  It also requires the inclusion of attribute values in one table in attribute values in another table. Every combination of FNAME, LNAME values in the INSTRUCTORS table must also appear in the COURSES table.

- But it is no foreign key since the referenced attribute combination is no key.

Integrity Control (5)

- Because of these problems, one can of course ask: "Should I use such cardinality specifications?"
- But if in the real world, there cannot be instructors that do not teach courses, the ER-schema with optional participation would be simply wrong.

  Of course, as for any constraint, one must always ask: Could there possibly be exceptional situations that would permit an instructor without courses? In that case, the mandatory participation would be wrong, because constraints do not permit any exceptions.

- Clearer example: Invoices without line items really do not make sense.

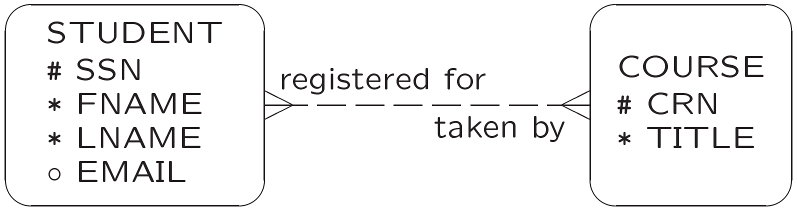
Integrity Control (6)

- When definiting the conceptual schema, one should not think about limitations of current technology.
- That is the task of logical (and physical) design.
- The problem can be solved (e.g. with checks in application programs and by searching for integrity violations with a query at least once a month).
- When technology advances, the same conceptual schema can be translated in a nicer way.

  More tasks are given to the system, less is explicitly programmed.

Many:Many Relationships (1)

- In the example, a many-to-many relationship still remains:



- Such relationships cannot be implemented by adding a foreign key to one of the two tables, because there can be more than one related entity.

Many:Many Relationships (2)

- Thus, a new table is created for the relationship (it is sometimes called an "intersection table").
- The new table contains the primary keys of both entity types that participate in the relationship.
- The two keys together form the composed key of the intersection table, and each is a foreign key referencing the table for its entity type:

  REGISTERED_FOR(<u>SSN</u>→STUDENTS,
  <u>CRN</u>→COURSES)

Many:Many Relationships (3)

- The intersection table for the relationship simply contains key value pairs of entities that are related:

REGISTERED_FOR

<u>SSN</u>	<u>CRN</u>
111-22-3333	12345
111-22-3333	56789
123-45-6789	12345

- E.g. John Smith (SSN 111-22-3333) is registered for Database Management (CRN 12345) and for Client-Server (CRN 56789).

Many:Many Relationships (4)

- Optional participation (minimum cardinality 0) is actually the only form of many-to-many relationship that can be implemented in this way with the standard constraints supported in SQL.

  If a student has to register for at least one course, it would be possible to store the CRN for the first course redundantly in the STUDENTS table and then one could declare SSN and CRN in STUDENTS as a foreign key referencing REGISTERED_FOR, but this is at least very ugly (one would also get severe problems inserting any data). One could also leave the foreign key out and take in all queries the union of the registration in the STUDENTS table and the registrations in the REGISTERED_FOR table. Again, such strange and tricky solutions lead to complicated programs and possibly errors.

Many:Many Relationships (5)

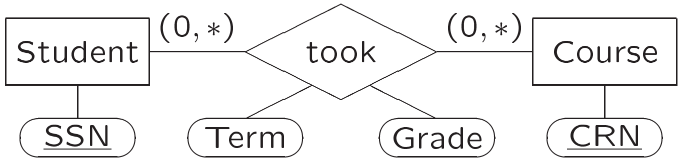
- As before, if one has mandatory participation, one uses the more general translation, and adds a constraint (to be checked e.g. in application programs).
- If a student can register for at most three courses, and could discuss also the following solution:

  STUDENTS(<u>SSN</u>, FNAME, LNAME, EMAIL^o,
  CRN1^o→COURSES, CRN2^o→COURSES,
  CRN3^o→COURSES)

- However, this significantly complicates queries (one will need a lot of "OR" and "UNION").

Many:Many Relationships (6)

- Suppose the relationship has attributes:



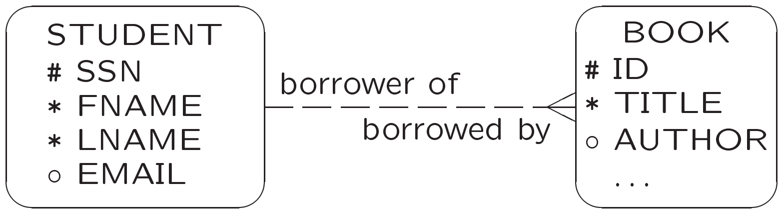
- Then one can simply add the relationship attributes to the relationship table:

  TOOK(<u>SSN</u>→STUDENTS, <u>CRN</u>→COURSES,
  TERM, GRADE)

- These attributes do not become part of the key.

One:Many: Alternative (1)

- One can also translate one-to-many relationships (with optional partcipation on both sides) into tables of their own.
- E.g. consider the following example: The university library wants to store who has borrowed with book:



One:Many: Alternative (2)

- This can also be translated in a similar way to a many-to-many relationship:

  BORROWED_BY($\underline{\text{ID}}$→BOOKS,
  SSN→STUDENTS)

- In contrast to a many-to-many relationship, ID alone suffices as key, since every book can be related to at most one student, so there can never be two entries for the same book.

One:Many: Alternative (3)

- Note that this alternative solution needs one more join in most queries than the standard solution.

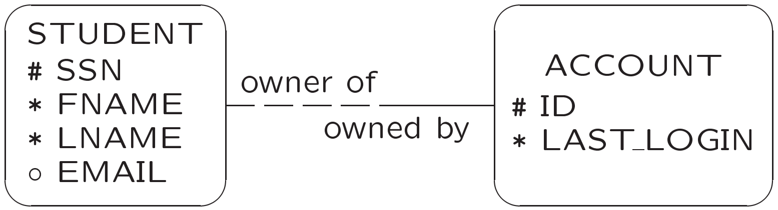
  The standard solution explicitly stores the outer join of the entity table and this relationship table, so that one does not have to compute the join at runtime.

- However, if there are very many books and very few of them are borrowed, permits fast access to the borrowed books.

  It might also be a bit more space-efficient.

One:One Relationships (1)

- Suppose we want to store which student is responsible for which computer account:



- The translation is basically done like a one-to-many relationship. If one side has mandatory participation, one treats that side as the "many" side.

One:One Relationships (2)

- The result of the translation is

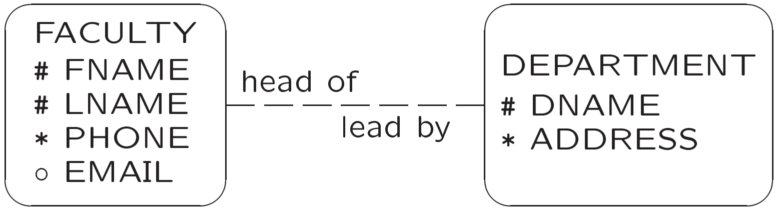
  STUDENTS($\underline{\text{SSN}}$, FNAME, LNAME, EMAIL^o)
  ACCOUNTS($\underline{\text{ID}}$, LAST_LOGIN,
  SSN→STUDENTS)

- The important difference to a "one-to-many" relationship is that the foreign key that implements the relationship now becomes an alternative key for the ACCOUNTS table.

- I.e. for every student SSN, there can be at most one account.

One:One Relationships (3)

- Now consider the case that the participation is optional on both sides:



- Now the situation is symmetric, and one can choose either side as "many" side.

  It would be a mistake to add a foreign key on both sides (redundant information).

One:One Relationships (4)

- In the example, it is probably an exceptional situation that departments do not have a head.
- One needs less null values if one chooses the side on which participation is "less optional" and adds the foreign key on this side:

  FACULTY(FNAME, LNAME, PHONE, EMAIL^o)
  DEPARTMENTS(DNAME, ADDRESS, (LNAME^o, FNAME^o) →FACULTY)

One:One Relationships (5)

- The relationship becomes one-to-one by specifying that LNAME, FNAME are an alternative key for DEPARTMENTS.

  Note that as always for optional composed foreign keys, one needs a `CHECK`-constraint specifying that LNAME and FNAME can only be together null.

- Not every DBMS supports alternative keys that can be null.

  And if they are supported, one has to check what the semantics is. E.g. in SQL server, at most one record with a null value in the key is permitted, which would not help here.

One:One Relationships (6)

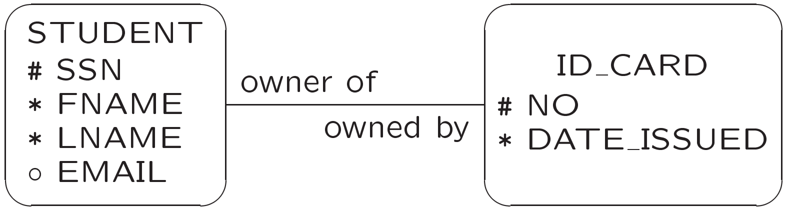
- However, if that does not work, one can also use the alternative translation for one-to-many relationships (with their own table):

  FACULTY(FNAME, LNAME, PHONE, EMAIL^o)
  DEPARTMENTS(DNAME, ADDRESS)
  DEPT_HEAD(DNAME→DEPARTMENTS (LNAME, FNAME)→FACULTY)

- LNAME and FNAME together are an alternative key for the relation DEPT_HEAD.

One:One Relationships (7)

- Finally, consider the case with mandatory participation on both sides:

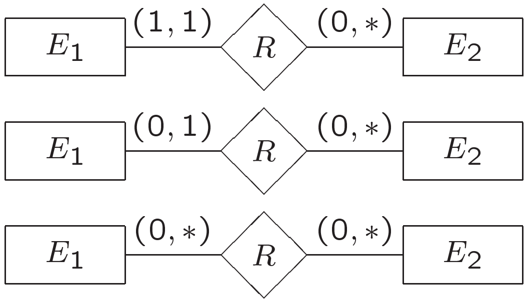


- In this case, one would translate the two entity types into only one table.

  One must select one of the two keys as primary key, the other becomes an alternative key.

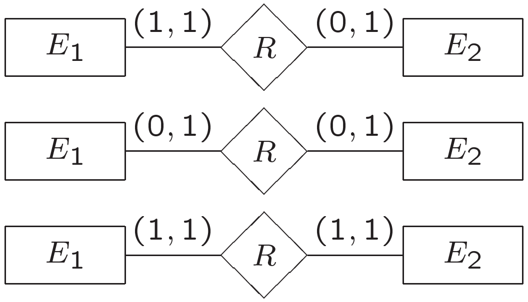
Summary: Limitations (1)

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



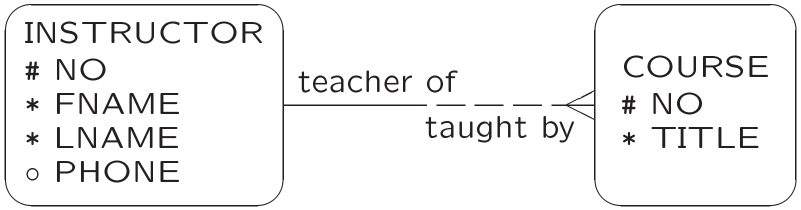
Summary: Limitations (2)

- In addition, all kinds of one-to-one relationships can be handled:



Renaming of Columns (1)

- Sometimes the direct application of the translation rules would lead to a name clash:



- In this example, one would get:

  COURSES(<u>NO</u>, TITLE, NO→INSTRUCTORS)

- But column names must be unique within a table.

Renaming of Columns (2)

- One can rename attributes during the translation in any understandable way.
- E.g. one could also use the role name in the relationship:
  COURSES(NO, TITLE,
  TAUGHT_BY→INSTRUCTORS)
- One could also add the name of the referenced table, or maybe a shorthand for it:
  COURSES(NO, TITLE,
  INST_NO→INSTRUCTORS)

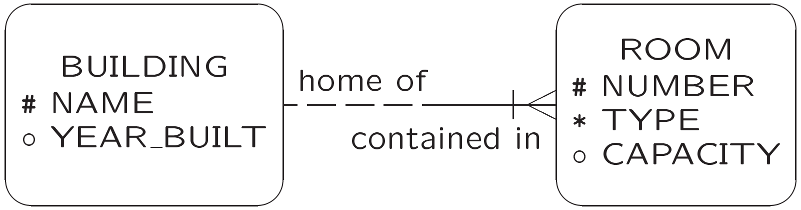
Renaming of Columns (3)

- The renaming must be carefully documented such that the ER-diagram is still useful as documentation for the implemented relational schema.
- Sometimes, it might be good to change the attribute name already on the ER-level.

  However, this is not always possible (e.g. in the case of recursive relationships).
- Also the table names generated for many-to-many relationships are often not very good and should be renamed.

Weak Entity Types (1)

- When weak entities are translated, the "borrowed" key attributes of the parent entity must be added.



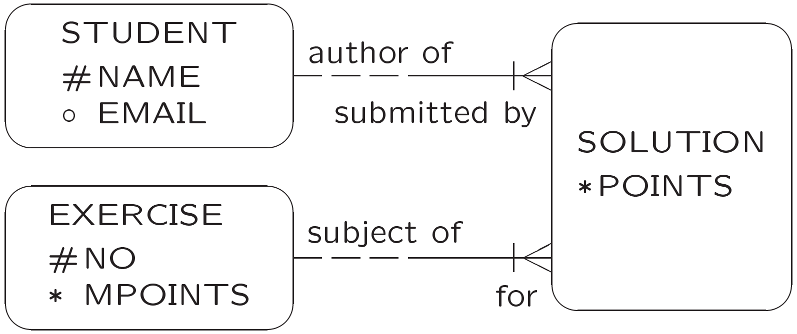
- The key of the "ROOMS" table will consist of the building name and the room number.

Weak Entity Types (2)

- The result of the translation is:
  BUILDINGS(NAME, YEAR_BUILT^o)
  ROOMS(NAME→BUILDINGS, NUMBER,
  TYPE, CAPACITY^o)
- I.e. the foreign key that is added to the weak entity table in order to implement the relationship with the parent type becomes part of the key.

Weak Entity Types (3)

- Next, consider a weak entity type with more than one parent ("Association Entity Type"):



Weak Entity Types (4)

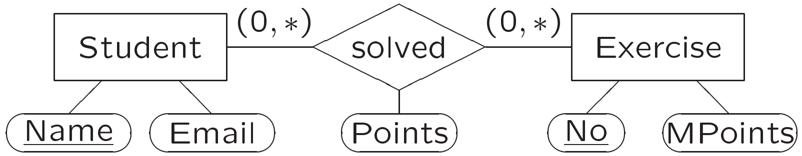
- The translation is done in the same way: The key of the weak entity type now consists of the keys of the two parent entity types (i.e. the two foreign keys added to implement the relationships):

  STUDENTS(NAME, EMAIL^o)
  EXERCISES(NO, MPOINTS)
  SOLUTIONS(NAME→STUDENTS,
  NO→EXERCISES,
  POINTS)

- Of course, any key attributes declared in the weak entity type itself would be added.

Weak Entity Types (5)

- Note that the translation result is exactly the same as if we had used a relationship with an attribute:

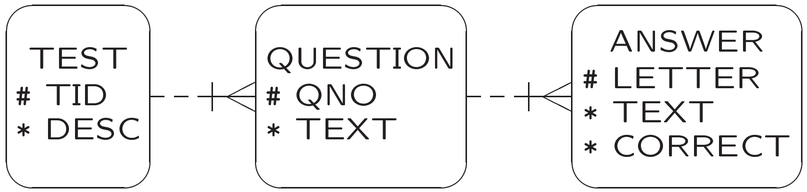


- This demonstrates again that the two ER-schemas are equivalent.

  When one has to check two ER-constructs for equivalence, one can try to translated them into the relational model. If the results are the same, the ER-schemas are equivalent. The converse does not hold.

Weak Entity Types (6)

- A weak entity can also be constructed over several steps. Consider a database schema for storing multiple choice online tests:



  Each test consists of several questions. For each question, the student has to check the correct answer among several alternatives. Within a test, questions are identified by a number. For a given question, each possible answer is identified by a letter (a, b, c).

Weak Entity Types (7)

- Before a weak entity type can be translated, all its parent entity types must be translated.

  In the example, first TEST must be translated, then QUESTION, then ANSWER.

- The reason is that in order to construct the primary key for a weak entity type, one must know the primary key of its parent entity type(s).
- This also means that any cycles in the "parent of" relation would give an ill-formed schema that has no meaning and cannot be translated.

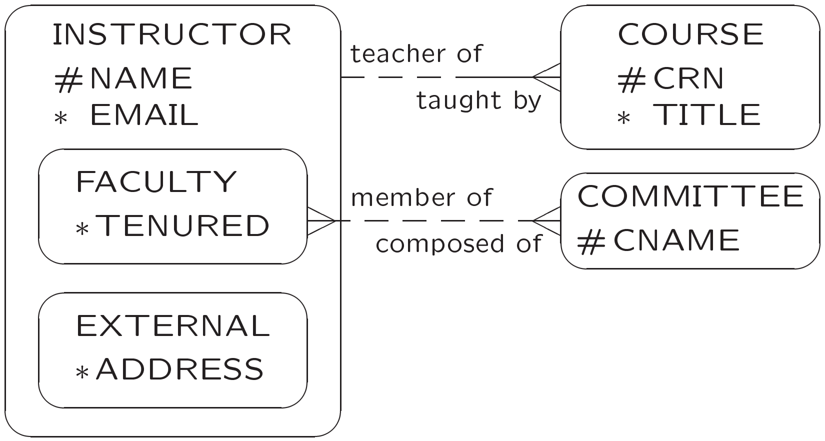
Weak Entity Types (8)

- The result of the translation in the example is:

  TESTS(TID, DESC)
  QUESTIONS(TID→TESTS, QNO, TEXT)
  ANSWERS((TID, QNO)→QUESTIONS,
  LETTER, TEXT, CORRECT)

- ANSWERS contains a foreign key that references its direct parent entity table QUESTIONS.
- This contains a foreign key referencing TESTS.
- It is logically implied that any TID value appearing in ANSWERS also appears in TESTS.

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`^{o}`, ADDRESS`^{o}`)`

- 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)

- 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 (6)

- 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 (7)

- 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 (8)

- 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 (9)

- 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 (10)

- 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 (11)

- 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 (12)

- 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 (13)

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 (14)

- 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 (15)

- 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 (16)

- 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 (17)

- 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 (18)

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

  COURSES(CRN, TITLE, FAC_NAME^o→FACULTY,
EXT_NAME^o→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 (19)

- 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 (20)

- 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 (21)

- 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 (22)

- 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 (23)

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 (24)

- 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 (25)

- 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 (26)

- 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 (27)

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 (numbers) are added to the subtype tables.
- Foreign keys are added to the supertype table (one for each subtype).

Subtypes/Specialization (28)

- 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 (29)

- 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 (30)

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 (31)

- 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.

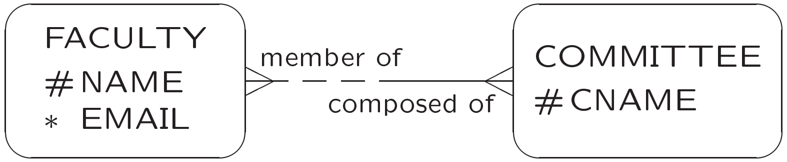
    It is a bit uncommon, but one can hide it in views. `UNION ALL` should really run fast. Modern optimizers should be able to work with it, old might produce not very efficient query execution plans.

Subtypes/Specialization (32)

- 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!

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 5).

  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.

Overview

Development Steps (1)

- First (during the conceptual design phase), one develops ER-diagrams with the ER-Diagrammer.

  The Repository Object Navigator can be used to check the global schema (and alter it, if necessary).

  Actually, one might start with business process diagrams and then design application program functions and the ER-schema concurrently.

- Then the Database Design Transformer is used to translate the ER-Schema (as stored in the Repository) into the relational model.

  One can choose to either translate the global schema or subsets of it step by step. The first option seems clearer.

Development Steps (2)

- The resulting relational schema is stored in the repository.
- One can then edit the relational schema (with the Design Editor or the Repository Object Navigator).
  - ◇ E.g. rename certain tables and columns.
  - ◇ View definitions, indexes, triggers, and other information that is not present in the ER-schema can be added at this stage.

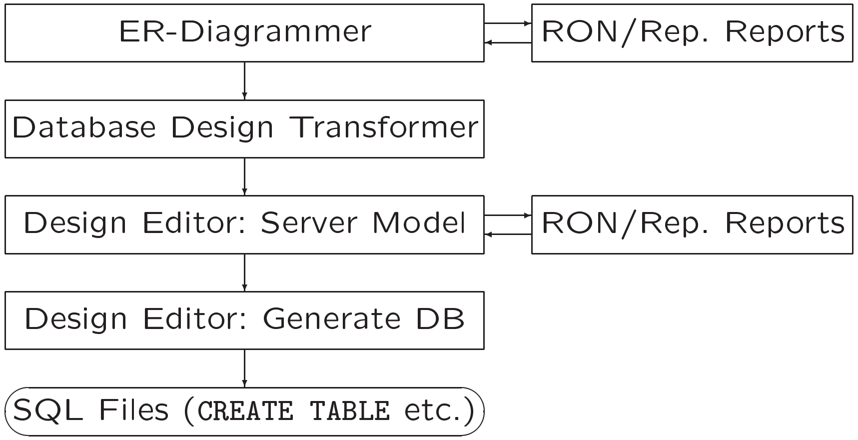
    The DB Design Transformer does not generate certain constraints that would be necessary for an exact translation of the given ER-schema. These must be added manually in this step.

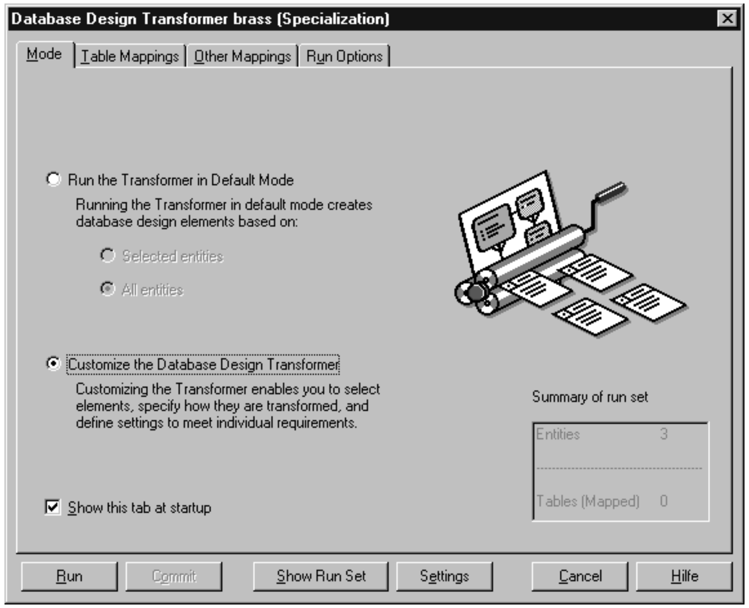
Development Steps (3)

- In the Design Editor, "Server Model Diagrams" can be developed that are a graphical representation of the relational schema.
- Finally, one can generate SQL code (for various database management systems) from the definitions stored in the repository.

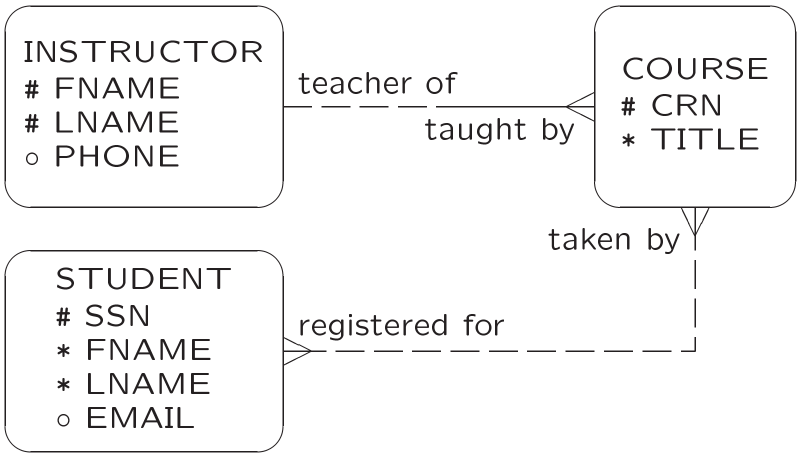
  This is also done with the Design Editor.

Development Steps (4)





Example



DB Design Transformer (1)

- As explained above, each entity type is transformed into a table:
  - ◇ The plural form of the entity type name is used as table name.
  - ◇ Spaces and punction characters in entity type and attribute names are mapped to underscores.
  - ◇ If a name is a reserved word in SQL, that name is modified (e.g. `FROM` becomes `FROM_FROM`).

    Reserved words depend in part on the DBMS, which is a problem in this step.

DB Design Transformer (2)

- Attributes of the entity type are translated into columns of the corresponding table:

```
INSTRUCTORS(FNAME, LNAME, PHONE°).
STUDENTS(SSN, FNAME, LNAME, EMAIL°).
```

- Columns are optional (null values allowed) if the corresponding source attribute is optional.

  The ER-Diagrammer permits optional attributes in primary keys. The DB Design Transformer silently corrects this mistake and makes the column not optional. Alternate key attributes remain optional.

- Primary/Alternate keys (UIDs) of the entity type become primary/alternate keys of the table.

DB Design Transformer (3)

- If an entity type has no primary key, a surrogate key is automatically added.
- E.g. for the INSTRUCTOR entity type, an attribute `INST_ID` of type `NUMBER(10)` would be added (where `INST` is the short name).

  In addition, a sequence called `INST_ID` is generated (for producing unique numbers). One can choose the domain for the `ID` columns.

- An option of the Database Design Transformer is to create a surrogate key for each table in this way.

  Then the declared primary keys for the entity types become alternate keys for the tables.

DB Design Transformer (4)

- For one-to-many relationships, foreign keys are added to the table at the “many” side (as expected):

```
COURSES(CRN, TITLE,
        (INST_FNAME, INST_LNAME) → INSTRUCTORS)
```

  The Database Design Transformer is able to produce foreign keys consisting of more than one column as in this case.

- Foreign key column names are constructed from the short name of the referenced entity type and the name of its primary key attribute.

  One can choose whether one wants the prefix. If the surrogate primary keys already have prefixes, one gets names like `INST_INST_ID`.

DB Design Transformer (5)

- If there name clashes (the table already has a column of that name), column names are made unique by adding the name of the relationship end (if that still does not help, numbers are added).
- If the participation in the relationship is optional, the foreign key attributes are declared as optional.

  However, if the foreign key consists of more than one attribute, a check constraint should be added that they can only be both null, or both not null. But the Database Design Transformer does not generate such constraints.

DB Design Transformer (6)

- If relationships are mutually exclusive (participate in an arc), the corresponding arc is stored for the generated foreign keys.

  However, when SQL code is later generated, the corresponding `CHECK` constraint is missing.

- One can choose how the generated foreign keys behave in case of deletions of the referenced row (restrict, cascade, nullify).

  One can also choose what happens in case of updates of the primary key values of the referenced row. The default value is "restrict" for updates and deletes, i.e. the deletion or update is not possible.

DB Design Transformer (7)

- For many-to-many relationships, an intersection table is generated:

  ```
  COURSES_STUDENTS(CRS_CRN→COURSES,
                   STUD_SSN→STUDENTS).
  ```

- The name of the table for the relationship is composed out of the plural forms of both entity types.

  Probably it would have been nicer if the relationhip names were used in some way. It is possible to change generated table and column names later in the Design Editor. Also, I would have preferred "`STUDENTS_COURSES`", with the "from" side of the relationship first. But the Database Design Transformer always uses the alphabetic sequence.

Restrictions (1)

- The alternate keys that would enforce one-to-one relationships are not generated.

  One-to-one relationships are translated by the DB Design Transformer in the same way as one-to-many relationships.

- Mandatory participation for many-to-many relationships or on the "one" side of one-to-many relationships are also lost in the translation.

  As explained above, this is no fault of the Database Design Transformer, since there is no good translation.

- No warning is generated.

Restrictions (2)

- Of the nine types of relationships that can be used in the ER-diagrammer, only three are exactly translated (see next page), the other ones are approximated by more liberal relationship types.

  As explained above, it would have been possible to implement also the three kinds of one-to-one relationships.

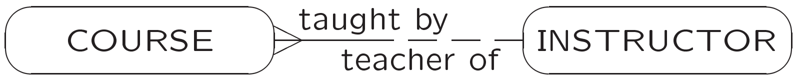
- Even constraints that cannot be enforced declaratively in the `CREATE TABLE` statements should be documented in the repository.

  The DB Design Transformer does not generate such constraints for the problematic cardinalities.

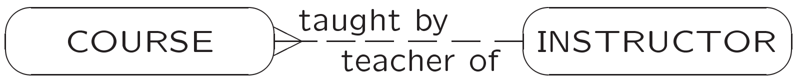
Restrictions (3)

Exactly Translated Relationship Types:

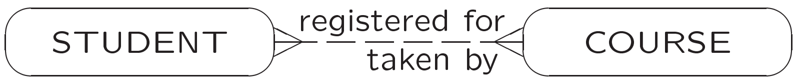
- Many-to-one, mandatory-to-optional:

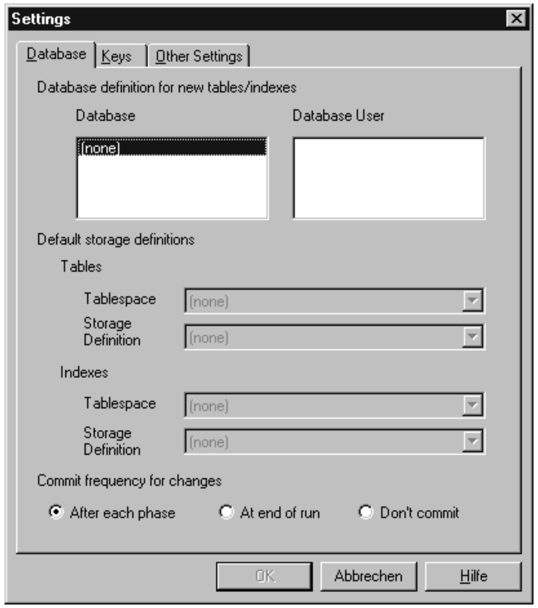


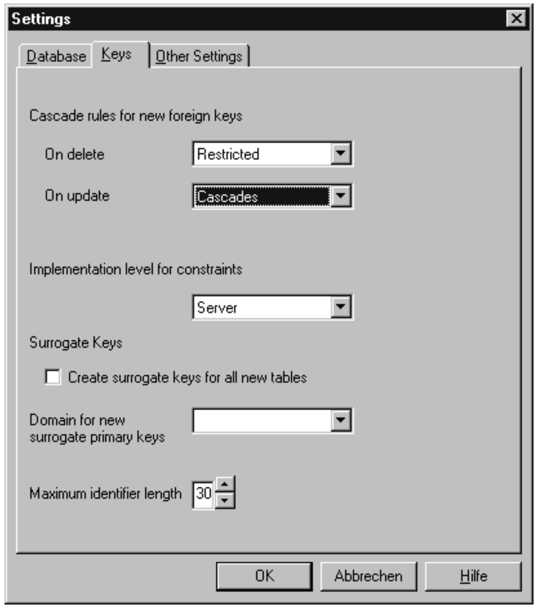
- Many-to-one, optional-to-optional:

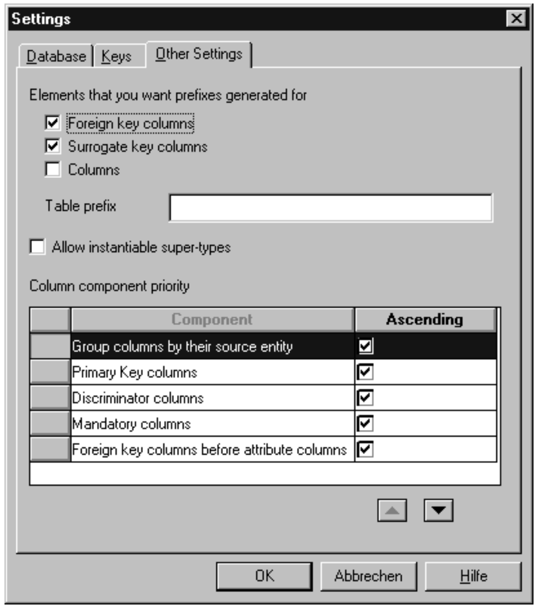


- Many-to-many, optional-to-optional:



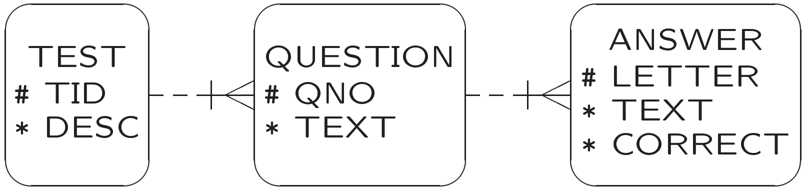






Translation of Weak Entities

- The DB Design Transformer translates the example from Slide 4-56 as expected:




```
TESTS(TID, TEST_DESC) -- DESC is a reserved word
QUESTIONS(TEST_TID→TESTS, QNO, TEXT)
ANSWERS((QUEST_TEST_TID, QUEST_QNO)→QUESTIONS,
        LETTER, TEXT, CORRECT)
```

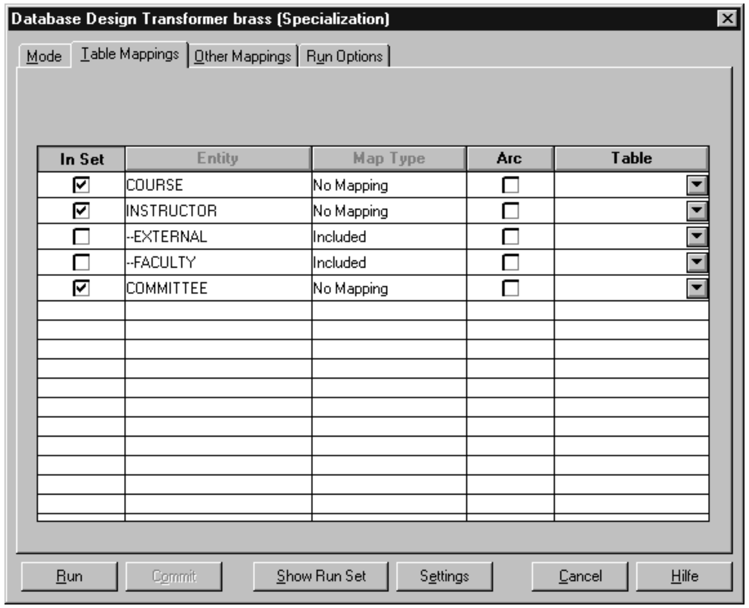

Translation of Subtypes (1)

- The Database Design Transformer supports
  - ◇ Method 1 (“Single Table Approach”)

    This is the default. One can specify in the Database Design Transformer which entity types are mapped to tables. Method 1 means that only the supertype is mapped to a table, the subtypes are marked as “Included”.

  - ◇ Method 2 (“Separate Table Approach”)

    One gets this transformation if one selects the subtypes to be mapped to tables, but not the supertype. Select the radio button “Customize the Database Design Transformer”. Then the tab “Table Mappings” appears. There select the “In Set” checkbox for the subtypes and deselect it for the supertype.

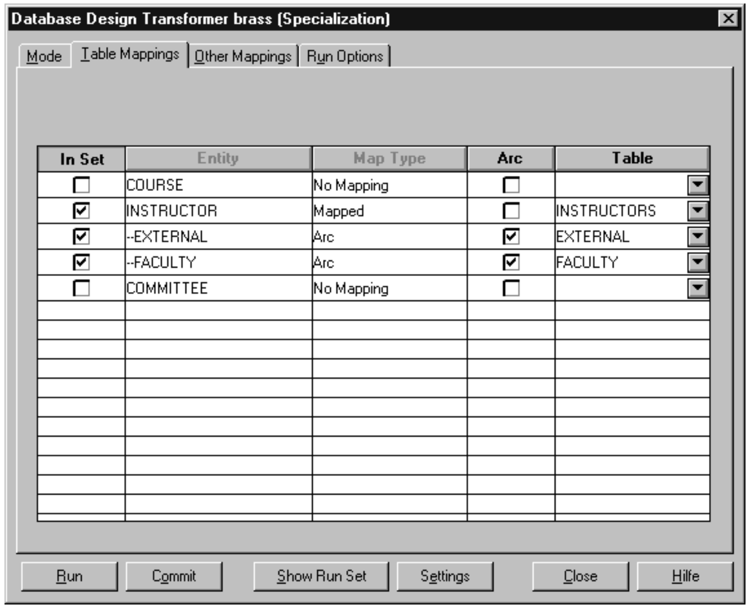


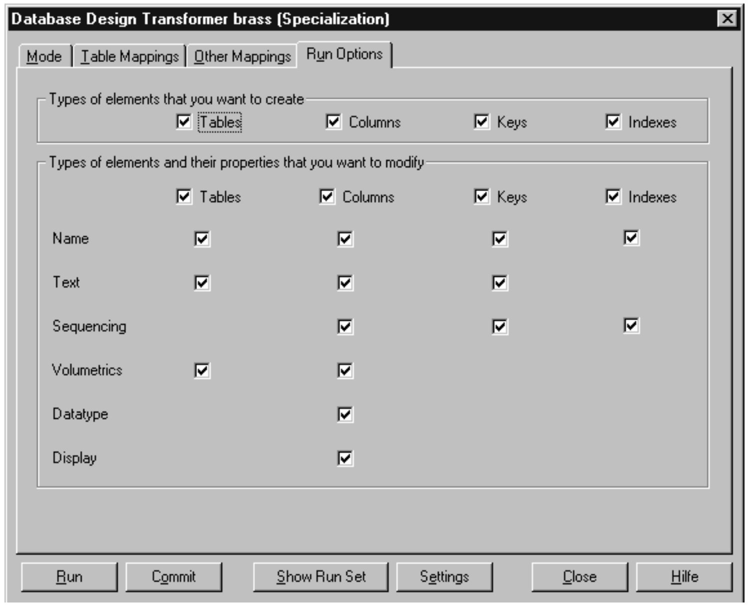
Translation of Subtypes (3)

- Supported Translation Methods, continued:
  - ◇ Method 2, Variant for Partial Specialization (“Implicit Sub-Type Approach”)

    One gets this option if supertype and subtype are mapped to tables. Instantiable subtypes can be selected under “Settings/Other Settings”. But one gets this translation also if it is not selected.
  - ◇ Method 4 (“Arc Approach”).

    For this transformation, the DB Design Transformer must be started two times (only for the supertype and the subtype, other entity types should be mapped in a third run): First map supertype and subtypes (check “In Set”) but in the “Run Options” permit only the generation of tables, not of columns or keys. In the second run, permit to create and modify tables, columns, and keys. For each subtype, the “Arc” flag must be set under “Table Mappings”.





Translation of Subtypes (6)

- Result of Method 1 (“Single Table Approach”):

  COMMITTEES(<u>CNAME</u>)
  COMMITTEES_FACULTY(COM_CNAME→COMMITTEES,
  INST_NAME→INSTRUCTORS)
  COURSES(<u>CRN</u>, TITLE, INST_NAME→INSTRUCTORS)
  INSTRUCTORS(ADDRESS^o, TENURED^o, <u>NAME</u>, EMAIL,
  INST_TYPE)
- The column “INST_TYPE” is declared to have values “EXT” and “FAC” (the short names of the subtypes).
- No CHECK-constraints are generated.
- The column sequence in INSTRUCTORS is strange.

Translation of Subtypes (7)

- Result of Method 2 (“Separate Table Approach”):

```
COMMITTEES(CNAME)
COMMITTEES_FACULTY(COM_CNAME→COMMITTEES,
                   FAC_NAME→FACULTY)
COURSES(CRN, TITLE, EXT_NAME^o→EXTERNAL,
                    FAC_NAME^o→FACULTY)
EXTERNAL(NAME, EMAIL, ADDRESS)
FACULTY(NAME, EMAIL, TENURED)
```

- An arc is generated for the foreign keys in COURSES.
- The split table method for many-to-many relationships with the supertype (“AWARD1/2”) is supported.

Translation of Subtypes (8)

- Result of “Implicit Sub-Type Approach”:

```
COMMITTEES(CNAME)
COMMITTEES_FACULTY(COM_CNAME→COMMITTEES,
                   FAC_NAME→FACULTY)
COURSES(CRN, TITLE, INST_NAME^o→INSTRUCTORS,
        EXT_NAME^o→EXTERNAL, FAC_NAME^o→FACULTY)
EXTERNAL(NAME, EMAIL, ADDRESS)
FACULTY(NAME, EMAIL, TENURED)
INSTRUCTORS(NAME, EMAIL)
```

- An arc is generated for the foreign keys in COURSES.
- This is Method 2 for partial specialization.

Translation of Subtypes (9)

- Result of Method 4 (“Arc Approach”):

```
COMMITTEES(CNAME)
COMMITTEES_FACULTY(COM_CNAME→COMMITTEES,
                   FAC_1_FAC_ID→FACULTY)
COURSES(CRN, TITLE, INST_NAME→INSTRUCTORS)
EXTERNAL(ADDRESS, EXT_ID)
FACULTY(TENURED, FAC_ID)
INSTRUCTORS(NAME, EMAIL, EXT_EXT_ID^o→EXTERNAL,
                         FAC_FAC_ID^o→FACULTY)
```

- The foreign keys in INSTRUCTORS are connected with an (optional) arc and marked as non-transferable.

Propagating Changes (1)

- It is probably best to start the DB Design Transformer only when one is finished with the ER-design.
- If one has already changed the relational schema, and then changes the ER-schema and runs the DB Design Transformer again, it is a difficult problem to merge both changes into one version.
- In general, it is important that the ER-Schema and the relational schema remain in sync — otherwise the ER-schema loses its value as a documentation for the created tables.

Propagating Changes (2)

- Of course, if one has not yet worked on the relational schema, one can simply delete it and run the DB Design Transformer again.

  Actually, it is not so simple to delete table definitions from the repository since they might be referenced in foreign keys. One must delete the foreign keys first. If one wants to delete all table definitions, one can click on the first, shift-click on the last, and then press the delete key. This will give an error message if a table is deleted that is still referenced by a foreign key. However, in Designer 6i (not Designer 6.0), one can choose to continue. After this is done, one simply presses "delete" again to remove the remaining tables (more runs might be needed, but if there are no cyclic foreign keys, finally all tables are deleted). In case of cyclic references, one must first delete at least one foreign key in the cycle before one can start to delete the tables.

Propagating Changes (3)

- Deleting the entire relational schema and running the DB Design Transformer again is the only completely automatic way that is guaranteed to keep both schemas in sync.
- The DB Design Transformer will never
  - ◇ remove existing tables (from a previous run) even if the corresponding entity type was deleted in the meantime,
  - ◇ remove columns from tables when the corresponding attribute was deleted.

Propagating Changes (4)

- The reason is probably that for denormalization, one could add columns and tables to the relational schema which are not present in the ER-schema.

  This should be a big exception, only if the performance requirements cannot be met with a good schema. But in earlier times it was done quite often (programmer time was cheap compared to hardware).

- The DB Design Transformer protects this work.

  The real reason probably is that in order to propagate deletions from the ER-schema to the relational schema, one must keep information about deleted schema elements. Also, the DB Design Transformer can be applied to a subset of the entity types. If one wants to delete tables, transforming the subset consisting of all entity types would be different from transforming the entire schema.

Propagating Changes (5)

- Under "Run Options" one can specify what the DB Design Transformer is allowed to modify.

  E.g. table names, column names, column sequence, column datatypes, etc.

- With the default (nothing can be modified) the DB Design Transformer remembers which elements in the ER-diagram are already mapped, and translates only new elements.

  E.g. if an attribute is added to an existing entity, it will be mapped to a new column in the existing table.

Propagating Changes (6)

- In the other extreme case (all modify options are checked), the new translation of the ER-schema overwrites the entire relational schema except that tables/columns are not deleted.
- E.g. even if one has renamed a column in the relational schema, running the DB Design Transformer again will reset it to its old name.

  I.e. the correspondence between ER-attributes and columns in tables is remembered in the repository, even if one of the two is renamed. One can see this information in the Repository Object Navigator under "`Usages/Implemented by Columns`" from the entity attribute.

Propagating Changes (7)

- One should not do arbitrary "last minute" changes in the relational schema. Go back to the ER-Schema and perform the required changes there!
- Depending on the kind of change, one can select the right modify options and run the DB Design Transformer only for the modified entity type.
- If something was deleted in the ER-schema, one must manually perform the corresponding deletion in the relational schema.

Overview

1. Schema Translation
2. Database Design Transformer
3. Design Editor: Server Model Diagrams
4. Design Editor: Database Administration
5. Generation of SQL Code

Design Editor (1)

- The relational schema generated by the database design transformer often still needs some work:
  - ◇ The names of the "intersection tables" for many-to-many relationships often must be changed.
  - ◇ Column names and the sequence of columns within a table might need changes.
  - ◇ Often, some constraints are missing.

    The DB Design transformer only generates keys, foreign keys, `NOT NULL`, and `CHECK` constraints for enumeration types or ranges automatically. Keys for one-to-one relationships are missing, as well as `CHECK`-constraints for subtypes, arcs, and other `CHECK`-constraints.

Design Editor (2)

- Manual work on the relational schema, continued:
  - ◇ For some foreign keys, one might have to select "ON DELETE CASCADES" etc.

    A default can be specified in the settings of the DB design transformer, but it might be useful to consider each case individually. E.g. for weak entities "ON DELETE CASCADES" is probably right.
- In addition, information necessary for the generation of application programs must be collected.

  E.g. display title of the form generated for a table, labels of input fields for columns, field type (text, radio buttons, etc.), field width, help text, display format (e.g. for date values), columns that are not displayed, etc.

Design Editor (3)

- After the logical design is finished, the following things must be defined:
  - ◇ Views.
  - ◇ Possibly triggers, stored procedures.
  - ◇ Users, table owners, access rights.
  - ◇ Physical design information.

    E.g. indexes, storage parameters for tables, distribution of tables over disks/tablespaces, etc. It is quite likely that the physical design will need to change when it turns out that the assumptions about the system load were not quite right. However, changing it after the data was loaded can be quite a lot of work.

Design Editor (4)

- Although this information can be edited directly with the Repository Object Navigator, Oracle offers a special tool for all this work: The Design Editor.
- The Design Editor consists of four distinct tools:
  - ◇ Server Model (Relational Database Schema)
  - ◇ Modules (Application Programs)
  - ◇ DB Administration (Users, Tablespaces, etc.)
  - ◇ Distribution (for Distributed Databases)

Design Editor (5)

- Later, first-cut application programs (for Oracle Developer Forms, Visual Basic, etc.) will be generated from the "module definitions".
- However, the module definitions contain only a link to the table name. The details such as the display width of input fields are defined in the server model (attached to tables).

  Of course, some things such as the exact position of the input fields on the form cannot be generated, and must be later edited with the programming tool itself (e.g. Oracle Developer Forms).

Design Editor (6)

- One window of the Design Editor is the Server Model Navigator.
- It looks very similar to the Repositor Object Navigator, but shows only objects that part part of the relational schema.

  A student thought that she could remove the relational schema (for a fresh run of the DB Design Transformer) by selecting the application system name at the top of the Server Model Navigator window and pressing "Delete". This removed her entire application system, not only the part shown in the window. For safety, export your design data at least once a day (with the Repository Object Navigator: "`Application`→`Export`") and copy them on a floppy disk.

Design Editor (7)

- The Design Editor uses normally wizards/tabbed dialog boxes instead of the simple property palette in the Repository Object Navigator.

  One can get also a property palette window under "`Tools`→`Property Palette`".
- The Design Editor also contains a tool to put information about the schema of an existing relational database in the repository.

  The "Design Capture Utility" ("`Generate`→`Capture Design of`→`Server Model`") can read the information from the data dictionary of an Oracle Database, from a file with SQL DDL (Create Table) commands, or via the ODBC interface.

Design Editor (8)

- The Design Editor has also a "Server Model Guide" which shows a tree of all server model object types:
  - ◇ Domains
  - ◇ Tables (Indexes, Triggers, Constraints)

    Constraints: Primary Keys, Foreign Keys, Unique Keys, Check.
  - ◇ Sequences
  - ◇ Advanced (Views, Snapshots, Clusters)
  - ◇ PL/SQL
  - ◇ Oracle8 (Collection Types, Object Types, Object Tables, Object Views).

Design Editor (9)

- When one selects an object type in the map, all objects of that type are shown. One can create, edit, or delete an object of the selected type.
- Basically, this is the same functionality as the "Server Model Navigator" which is also part of the Design Editor. Only the user interface is a bit different.

  One can also choose that the two tools are linked: When an object is selected in the Server Model Guide, it is automatically also selected in the Server Model Navigator. The Server Model Guide gives more advice what to do in which sequence and sometimes has links to documentation.

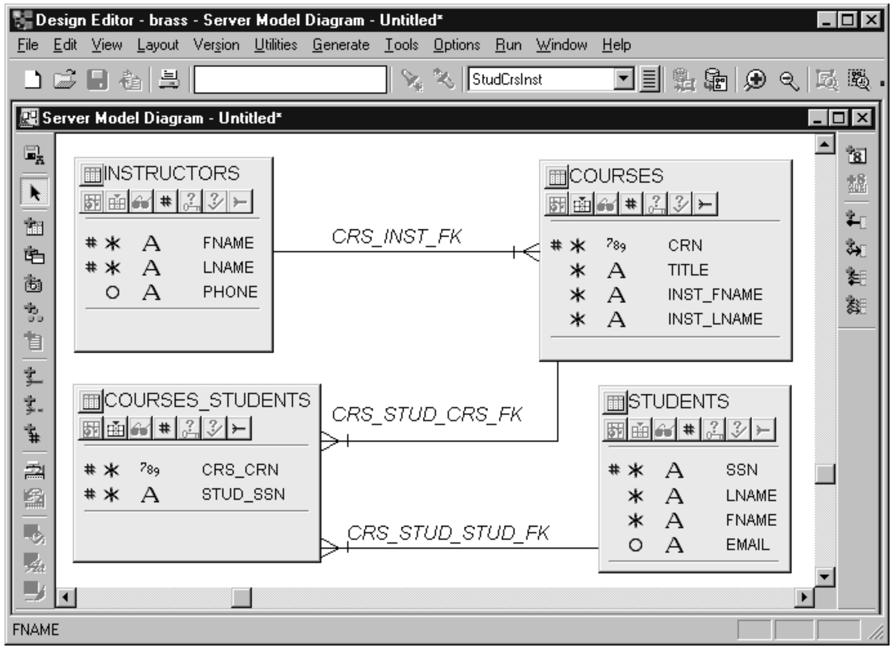
Server Model Diagrams (1)

- The “Server Model” part of the Design Editor has a graphical interface showing tables and their foreign key connections in “Server Model Diagrams”.

  The easiest way to create a diagram is to expand the “Relational Table Definitions” in the “Server Model Navigator” on the left, then to select the tables that should appear on the diagram (e.g. click on the first table and shift-click on the last) and then to select “File→New→Server Model Diagram”.

- These diagrams are quite similar to ER-Diagrams.

  However, the orientation on ER-diagrams is simpler. Server model diagrams are overloaded with information, table boxes are larger than entity boxes. Also many-to-many relationships are now shown as tables of their own, and foreign key columns do not appear on ER-diagrams.



Server Model Diagrams (3)

- Tables are shown as boxes with three sections:
  - ◇ The first section contains the table name and a number of buttons.

    Buttons: “Database Triggers”, “Indexes”, “Database Synonyms”, “Primary Key”, “Unique Keys”, “Check Constraints”, “Foreign Keys”. A dimmed button means that the table has no object of that type. The button left to the table name is unusable.
  - ◇ The second section lists the columns (→ below).
  - ◇ The third section shows additional information as selected by the buttons in the first section.

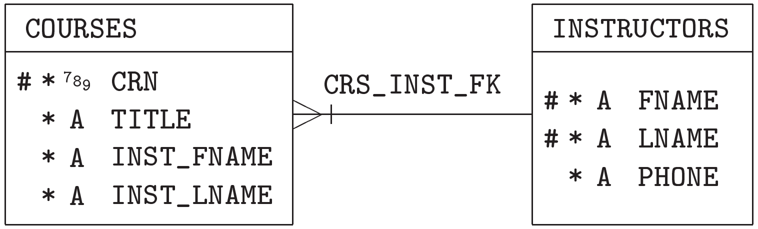
    If one selects “View→Track Associations” and clicks on e.g. an index, the corresponding columns are shown inverted above.

Server Model Diagrams (4)

- The second section of the table box contains one row per column with the following information:
  - ◇ “#”: member of the primary key.
  - ◇ “*”: mandatory column (not null),
    “○”: optional column.
  - ◇ “”: enumeration type value list.
  - ◇ “A”: character/string data type,
    “789”: numeric data type.
  - ◇ “123”: sequence (unique number generator).
  - ◇ “”: column belongs to domain.

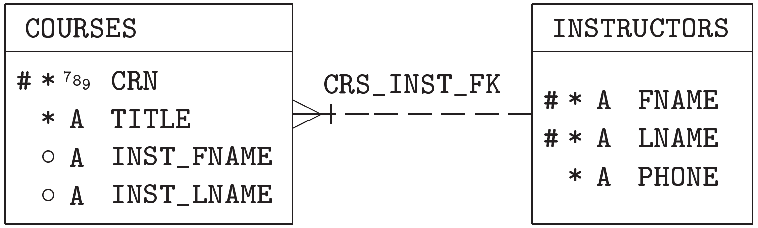
Server Model Diagrams (5)

- Foreign keys are shown as lines between the tables and use symbols similar to "one-to-many" relationships (but beware of the differences).
- Mandatory foreign keys (i.e. foreign keys that must be not null) are shown as solid lines:



Server Model Diagrams (6)

- Optional foreign keys (i.e. foreign keys that can be null) are shown as dashed lines:



- The entire line is either solid or dashed, there are no longer two halves.

Server Model Diagrams (7)

- Corresponding to the one-to-many relationship, the "crows foot" is on the side with the foreign key.

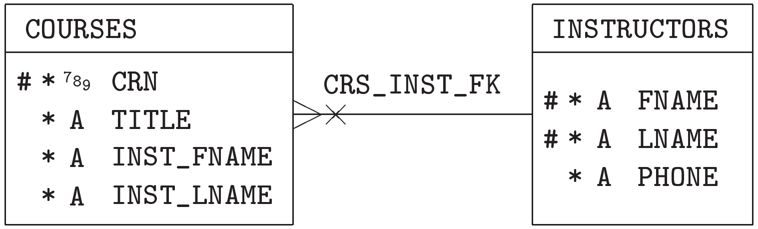
  It can also be seen as indicating the direction of the pointer, although a real arrowhead would be on the opposite side.

- The names of the foreign keys are often not helpful, but take space on the diagram.

  With "`Options→Show/Hide`" one can determine what is shown on the diagram. Removing the check mark from "Text" of "Associations" hides the foreign key names. One can specify which kinds of columns are shown, e.g. hide the foreign key columns on the diagram. One can also select which of the column type symbols are shown.

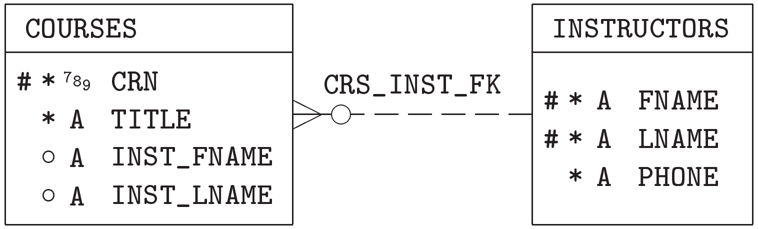
Server Model Diagrams (8)

- The small vertical bar near the crowsfoot means that deletions do not cascade ("restricted", one cannot delete an instructor that teaches courses).
- If one selects "`ON DELETE CASCADES`", the line is crossed with an "x":



Server Model Diagrams (9)

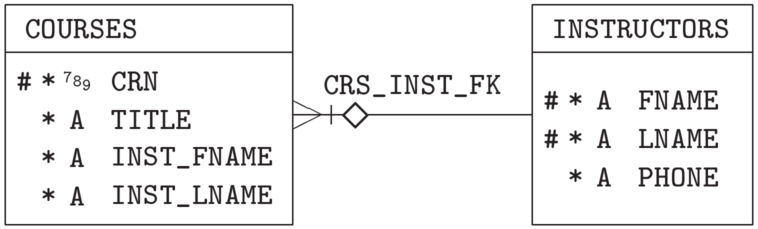
- If "ON DELETE SET NULL" is selected, a circle is used:



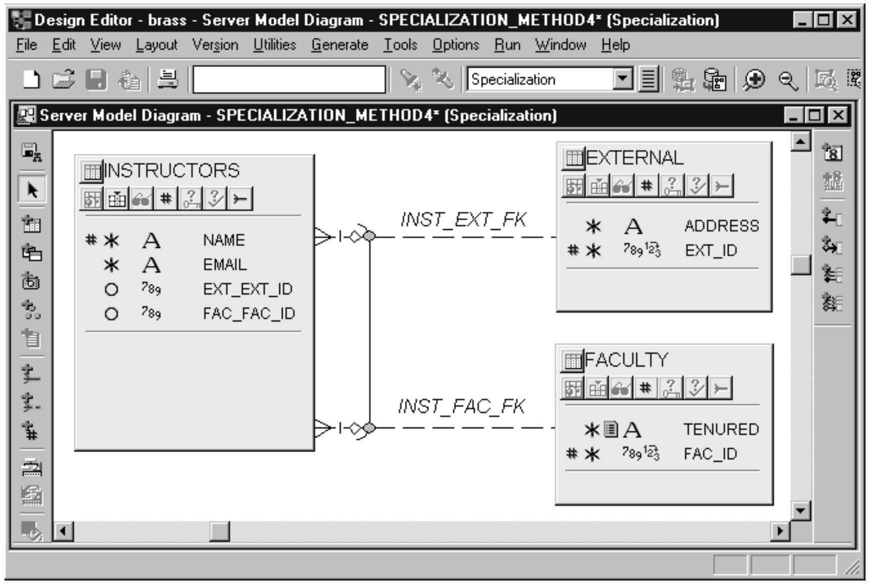
- A filled circle means "ON DELETE SET DEFAULT".
- The cascade rule for "ON UPDATE" is not shown on the diagram.

Server Model Diagrams (10)

- One can also mark foreign keys as non updatable (corresponding to a non-transferable relationship):



- Foreign keys can be marked as mutually exclusive by means of arcs (as on ER-diagrams).



Server Model Diagrams (12)

- The properties dialog box for tables ("Edit Table") has tabs "Name", "Columns", "Display", "Controls", "UI".

  Under "Display", one can define which columns correspond to input fields in a form. Under "Controls" the type, size, etc. of these input fields is defined. Under "UI" (User Interface), more information about input fields is defined, e.g. a help text and a display format.

- Column and table names can be edited directly in the diagram, one does not have to go over the properties dialog box.

Server Model Diagrams (13)

- Tables also have the “Edit Text” dialog box, where one can define a description, notes, help text, and code for insert, update, delete, and locks.

  One can open this dialog box from the menu that appears if one right-clicks on the table. This menu also permits to add columns, triggers, indexes, synonyms, keys, check constraints, foreign keys.

- The properties dialog box for foreign keys has tabs “Foreign Key Mandatory”, “Foreign Key Column”, “Cascade Rules”, “Validation”.

  E.g. under “Validation” one can choose whether the constraint should be enforced on the server, the client, or both. One can also specify an error message and a table for exceptions (rows that violate it).

Server Model Diagrams (14)

- If one chooses to add e.g. a check constraint to a table, a wizard is opened that asks for the required information.

  To edit it later, display it in the third part of the box (by clicking on the button for the object type) and click on the symbol in front of the name. Clicking on the name only permits to edit the name. Editing an existing check constraint etc. shows the same screens as the wizard, but now one can jump with tabs between them.

- Oracle Designer does not check the SQL syntax e.g. of `CHECK`-constraint definitions.

  One can enter any text. Column names can be selected from a list. Of course, the exact SQL syntax depends on the DBMS.

Server Model Diagrams (15)

- The DB Design Transformer has already created indexes for foreign keys.

  In addition, the DBMS automatically creates indexes for primary and alternate keys.

- As part of the physical design, one can add further indexes to a table.
- One can also add triggers (e.g. for enforcing complex constraints or logging changes to a table).

Server Model Diagrams (16)

- Server model diagrams can also contain other objects, such as
  - ◇ Views (shown as grey-blue boxes).

    In order to create a view, one can e.g. right-click on the background of a server model diagram. Alternatively there is also a symbol on the left toolbar. A wizard is started that asks the required information. One can select base tables (`FROM`) and columns (`SELECT`), and then any `WHERE` clause can be entered.

  - ◇ Object types (shown as red boxes).

    In Oracle, an object type is a generalization of a record/row type. One can create one or more tables over an object type. Object type an tables are connected with a line that ends in a diamond attached to the table.

Server Model Diagrams (17)

- Objects on Server Model Diagrams, continued:
  - ◇ Clusters (shown as grey boxes).

    In Oracle, a cluster is a storage area in which rows of one or more tables may be stored, such that rows with the same value in the cluster column are stored together.
  - ◇ Snapshots (shown as light blue boxes).

    In Oracle, a snapshot is a copy of another table or view, used in distributed DBs for performance or failure safety reasons. One can specify that it is automatically refreshed at certain intervalls.
- The diagram legend can be shown in the upper left corner (it contains diagram title, author, date etc.).

Repository Reports

- Again, there are many repository reports which can be printed for documenting the DB Design, e.g.:
  - ◇ Entity to Table Implementation
  - ◇ Table Definition
  - ◇ Column Definition
  - ◇ Columns in Domain
  - ◇ Constraint Definition
  - ◇ Database Trigger
  - ◇ Cluster Definition
  - ◇ Tables, Columns, and Foreign Key Derivations

Overview

Database Administration (1)

- The following information can be specified with this part of the Design Editor:
  - ◇ Database Name and connection information.
  - ◇ Access information: Users, Roles, Profiles.
  - ◇ Storage Information: Tablespaces, Datafiles, Logfiles, Rollback Segments, Directories.
- In the Server Model view, the really physical information (like storage parameters) was not yet asked.

  Also, in the server model, the tables do not yet belong to users (there is no such property). One must move from the Server Model Relational Table Definitions to Table Implementations under "DB Admin".

Database Administration (2)

- One now can "implement" the tables under a user account in a database. A new wizard asks for a table from the server model and takes the designer through the physical options.

  One gets this wizard e.g. by selecting a user in the Database Administrator Guide, then selecting "Tables" and clicking on "Create". This does not mean that a table is created from scratch, one can select a table from the Server Model. Since often several tables have the same storage parameters, one can create named sets of such parameters ("Storage Definitions") and assign to tables.

- Of course, the resulting data are still stored in the repository. The table is not yet really implemented.

Database Administration (3)

- The Database Administration part of the Design Editor is only a subset of the Repository Object Navigator (there are no new diagrams).

  However, again wizards/tabbed dialog boxes are used instead of the property palette. And it has a "Database Administrator Guide" that shows the steps for specifying a database.

- The tool is similar to a graphical user interface for a DBA (but stores all information in the repository).
- From the collected information, database creation scripts can be generated.

Overview

1. Schema Translation
2. Database Design Transformer
3. Design Editor: Server Model Diagrams
4. Design Editor: Database Administration
5. Generation of SQL Code

DDL Generation (1)

- The Database Design Transformer stores the relational schema in the repository. It does not actually create the tables.
- The reason for this is that in most cases, some things must still be changed/added manually.
- Once one is satisfied with the relational schema, one can generate SQL DDL code containing e.g. `CREATE TABLE` statements.

  DDL = Data Definition Language. The generation is done with the Design Editor: "`Generate`→`Generate Database from Server Model`".

DDL Generation (2)

- Oracle Designer can create DDL code for different DBMS: ANSI 92, DB2, Oracle (different versions), RDB7, SQL Server, Sybase.
- The creation of tables etc. can be done as follows:
  - ◇ Files with DDL statements are created, these must be executed manually in the target DB.
  - ◇ If the target database is an Oracle Database, Oracle Designer can directly create the tables.
  - ◇ If the target DB supports ODBC connections, tables can also be directly created.

DDL Generation (3)

- One can select for which schema objects should be generated (e.g. only a subset of the tables).

  This is done on the "`Objects`" tab. E.g. click on the double right arrow: "`Generate All`".
- What can be generated, depends on the DBMS chosen, e.g.:
  - ◇ "`ANSI 92`": Only tables and views.
  - ◇ "`SQL Server`" Only domains, tables, and views.

    Which is strange, since it has indexes.

DDL Generation (4)

- When creating files, one defines a file prefix (e.g. `courses`) and a directory. The different kinds of schema elements will then be written to different files (for Oracle8):
  - ◇ `courses.tab`: Table Definitions
  - ◇ `courses.con`: Constraints (as `ALTER TABLE ...`)
  - ◇ `courses.ind`: Indexes
  - ◇ `courses.sqs`: Sequence Definitions
  - ◇ `courses.sql`: Includes all of the above files.