Datenbanken II A: DB-Entwurf

Chapter 5: Logical Design I

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Wintersemester 2020/21

`http://www.informatik.uni-halle.de/~brass/dd20/`

Objectives

After completing this chapter, you should be able to:

- translate given ER-schemas (without subclasses) manually into the relational model.

  Entity types, one-to-many relationships (two alternative translations), many-to-many relationships (including attributes), one-to-many-relationships, weak entity types.

- explain which cardinalities cannot be enforced with standard constraints of the relational model.
- Explain several options for enforcing general constraints.

  And compare them, e.g. name weaknesses.

- Write SQL queries to search for constraint violations.

  In particular for cardinality constraints from the ER-model that could not be translated into standard constraints of the relational model.

Contents

1 Foundations

2 Relationships

3 Limitations, Integrity Control

4 Weak Entity Types

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

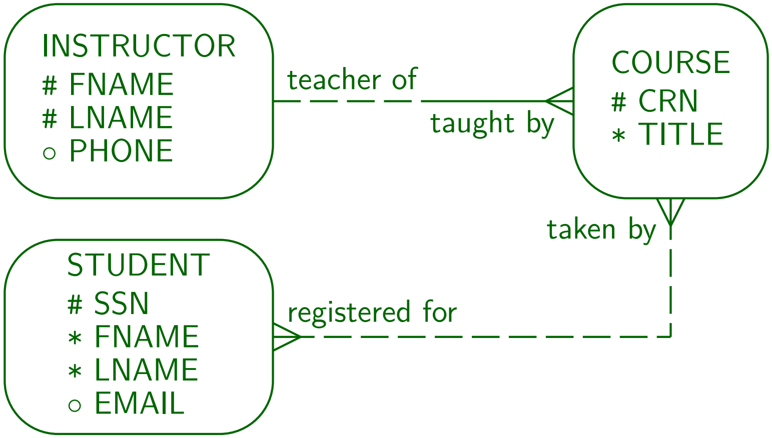
  E.g., in the ER-model, relationships can be always only between currently existing entities. In the relational model, "dangling pointers" must be explicitly excluded by means of foreign key constraints.

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



INSTRUCTOR
# FNAME
# LNAME
○ PHONE
teacher of
taught by
COURSE
# CRN
∗ TITLE
taken by
STUDENT
# SSN
∗ FNAME
∗ LNAME
○ EMAIL
registered for

Entity 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.

Entity 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:

$$\texttt{INSTRUCTORS}(\underline{\texttt{FNAME}},\ \underline{\texttt{LNAME}},\ \texttt{PHONE}^{o})$$
$$\texttt{STUDENTS}(\underline{\texttt{SSN}},\ \texttt{FNAME},\ \texttt{LNAME},\ \texttt{EMAIL}^{o})$$
$$\texttt{COURSES}(\underline{\texttt{CRN}},\ \texttt{TITLE})$$

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	

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

Contents

1. Foundations
2. Relationships
3. Limitations, Integrity Control
4. Weak Entity Types

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($\underline{\text{CRN}}$, TITLE,
  (FNAME,LNAME)$\rightarrow$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			
<u>CRN</u>	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!**

<u>FNAME</u>	<u>LNAME</u>	Phone	CRN
Stefan	Brass	624-9404	$\{12345, 24816\}$
Michael	Spring	624-9424	$\{56789\}$
Nina	Brass		$\emptyset$

One:Many Relationships (6)

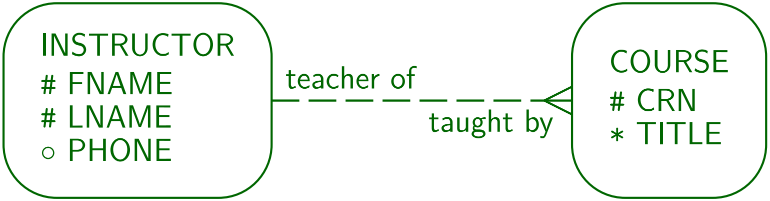
- Unfolding the set-valued attribute would destroy the key, store information redundantly (instructors of multiple courses), and lead to the loss of other information (instructors of no course).

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($\underline{\text{CRN}}$, TITLE,
  (FNAME^o,LNAME^o) $\rightarrow$ INSTRUCTORS)

- Example State:

COURSES			
$\underline{\text{CRN}}$	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.

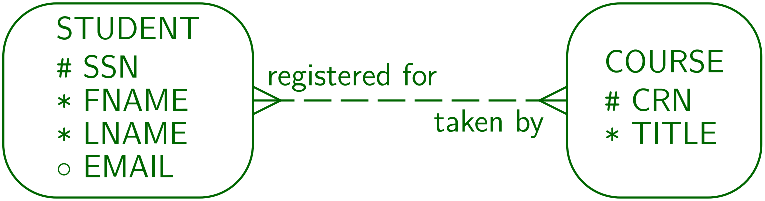
- Fortunately, this condition can be enforced declaratively with a CHECK-constraint:

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

  It depends on the DBMS whether this constraint is really necessary, often the foreign key constraint will actually suffice. But at least is constraint is a good documentation.

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.
- 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:

$$\text{REGISTERED\_FOR}(\underline{\text{SSN}}\rightarrow\text{STUDENTS},\ \underline{\text{CRN}}\rightarrow\text{COURSES})$$

Many:Many Relationships (3)

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

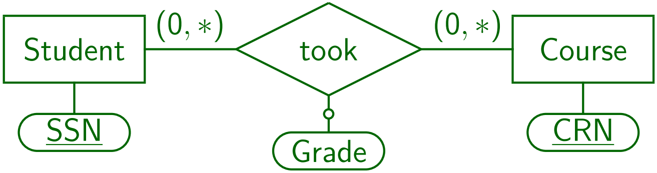
REGISTERED_FOR

<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)

- Suppose the relationship has attributes:



The circle in the connection to the attribute means that the attribute is optional (can be null).

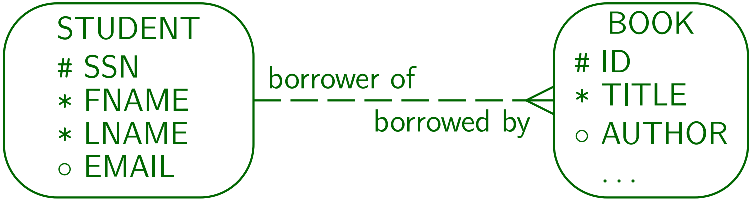
- Then one can simply add the relationship attributes to the relationship table:

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

- Relationship 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 which 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)

  Some professors first explain the translation of all relationships as tables of their own with the two foreign keys, and then merge relations with the same key. E.g. this relation has the same key as the BOOKS relation, and if we merge the two relations we get to the standard translation of one-to-many relationships.

- 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.

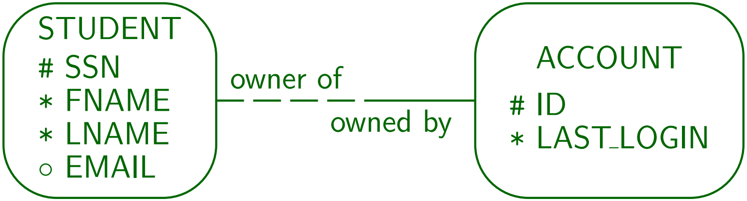
  The standard solution explicitly stores the outer join of the entity table and this relationship table, so that one does not have to compute the join at runtime.

- However, if there are very many books and very few of them are borrowed, the alternative solution 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

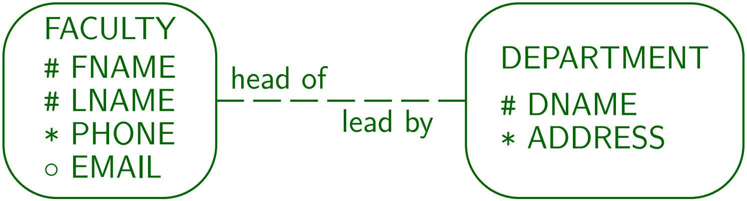
  STUDENTS($\underline{\text{SSN}}$, FNAME, LNAME, EMAIL^o)
  ACCOUNTS($\underline{\text{ID}}$, LAST_LOGIN, SSN$\rightarrow$STUDENTS)

- 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($\underline{\text{FNAME}}$, $\underline{\text{LNAME}}$, PHONE, EMAIL^o)
DEPARTMENTS($\underline{\text{DNAME}}$, ADDRESS,
(LNAME^o, FNAME^o) $\rightarrow$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)

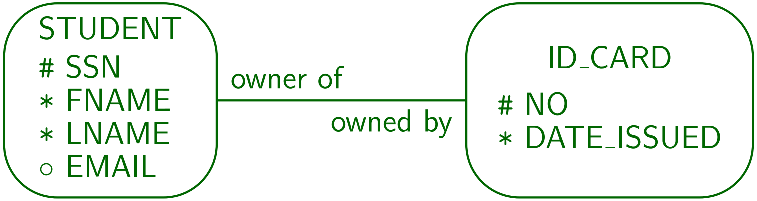
- However, if that does not work, one can also use the alternative translation for one-to-many relationships (with their own table):

  FACULTY(<u>FNAME</u>, <u>LNAME</u>, PHONE, EMAIL^o)
  DEPARTMENTS(<u>DNAME</u>, ADDRESS)
  DEPT_HEAD(<u>DNAME</u>→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:

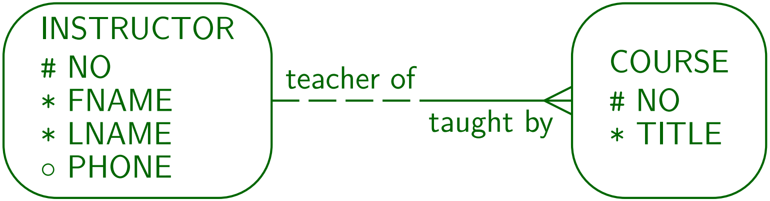


- In this case, one would translate the two entity types into only one table.

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

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

- One could also add the name of the referenced table, or maybe a shorthand for it:

  COURSES(<u>NO</u>, 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.

Contents

1 Foundations

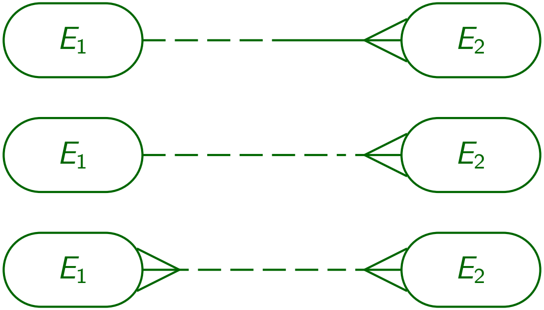
2 Relationships

3 Limitations, Integrity Control

4 Weak Entity Types

Summary: Limitations (1)

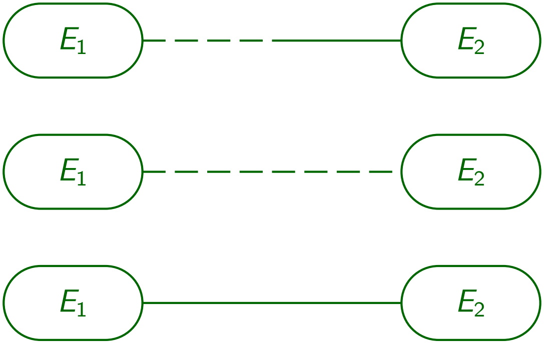
- The following cardinalities can be translated with the methods explained above (using only the standard constraints of the relational model):



(continued on next slide . . .)

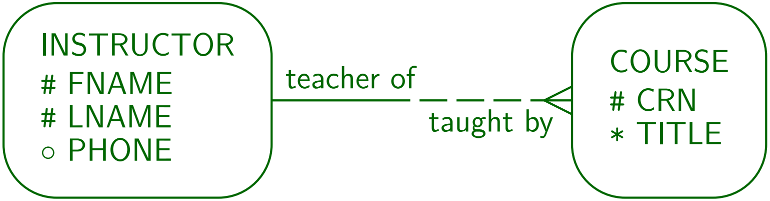
Summary: Limitations (2)

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



One:Many Relationships (1)

- 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 (2)

- 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 (3)

- 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)
```

- One could even generate an error message:

```
SELECT 'Instructor without courses: ' ||
       FNAME || ' ' || LNAME   AS ERRMSG
```

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 not 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)

- Good application programs anyway should handle all possible constraint violations, even if the DBMS enforces the constraint.

  At least all constraint violations that could possibly occur due to bad user input. Other constraint violations are automatically prevented by the application logic (e.g. if the user first selects a customer and then enters an order), and then the check in the DBMS suffices (in case the program contains a bug or somebody else deletes the customer in the meantime).

  The error message generated by the DBMS is normally not very clear for the untrained user, therefore at least some form of exception handling that produces a better error message for the specific application context should be done. Of course, the application could simply check for these constraint violations itself before executing the critical update. But this duplication could also be considered bad style.

Integrity Control (4)

- Thus constraint checks in the DBMS basically give a second level of protection against:
  - application programs that contain bugs,
  - application programs that contain holes,

    It is easy to overlook that a specific update might violate a certain constraint, although there is a formal theory that can compute all possible "critical updates" from the given constraint formula.
  - users that have direct SQL access to the DBMS,
  - unexpected interference of concurrent users.
- Declarative constraints are also a formal and concise specification for the checks in the software.

Integrity Control (5)

- If a constraint is not declaratively supported by the DBMS, triggers can be used to enforce it.

  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” (ECA-rules).

- 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.
  This approach is similar to a class that permits to change its attributes only via the methods defined in the class. One can use access rights in the database to enforce this.

Integrity Control (6)

- 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 needed in the example (no instructor without course) is similar to a foreign key.

  Like a foreign key it requires the inclusion of attribute values: 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 (7)

- 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 (8)

- When defining 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.

Integrity Control (9)

- Defining the right cardinalities is also important because it influences the application programs:
  - If there cannot be instructors without courses, the application program to insert an instructor must also insert at least one course.

    Probably, the application should permit to insert more than one course, since there is no real reason to select one specific out of the many courses an instructor might teach.
  - Otherwise, there will probably be different programs to insert instructors and to insert courses.

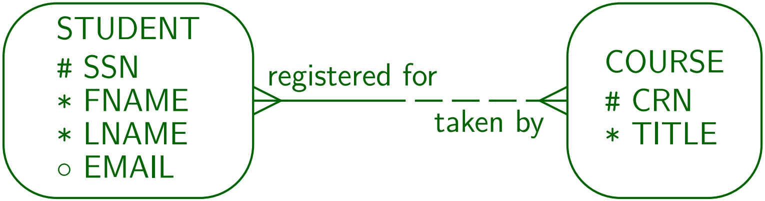
Integrity Control (10)

- One can analyse ER-diagrams for elementary transactions as given by the cardinalities.
- If these should turn out to be too complicated, one should think again about the minimal cardinalities.
- For such an approach, it would make sense to define already on the ER-level
  - Which attributes are updatable?
  - Which entities are deletable?
  - Which entities can be independently inserted?

    Can an existing order be extended by new positions?

Many:Many Relationships (1)

- Optional participation (minimum cardinality 0) is the only form of many-to-many relationship that can be implemented with a “relationship table” and the standard constraints supported in SQL.
- Suppose students must take at least one course:



- Relation generated for relationship:

  REGISTERED_FOR($\underline{\text{SSN}}$ → STUDENT, $\underline{\text{CRN}}$ → COURSE)

Many:Many Relationships (2)

- 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).

  The translation shown on the previous slide gives optional participation (dashed lines). E.g. the relation `REGISTERED_FOR` can be empty, while the `STUDENTS`-relation is not empty.

- If a student can register for at most three courses, one could discuss also the following solution:

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

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

Many:Many Relationships (3)

- Even in the general case, there are tricky solutions that would formally solve the problem (mandatory participation in a many-to-many relationship).

  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.

- However, such strange solutions lead to complicated programs and possibly errors.

Contents

1 Foundations

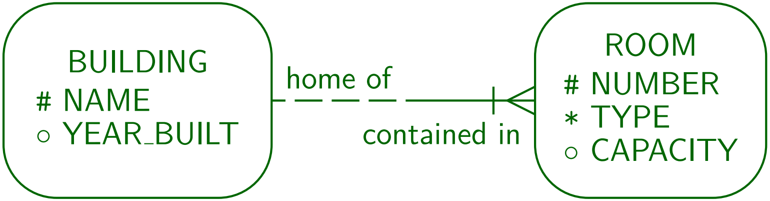
2 Relationships

3 Limitations, Integrity Control

4 Weak Entity Types

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)

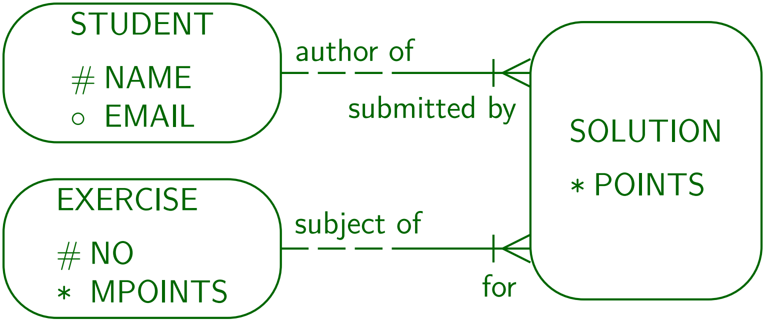
- The result of the translation is:

  BUILDINGS($\underline{\text{NAME}}$, YEAR_BUILT^o)
  ROOMS($\underline{\text{NAME}}\rightarrow$BUILDINGS, $\underline{\text{NUMBER}}$,
  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 entity 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)

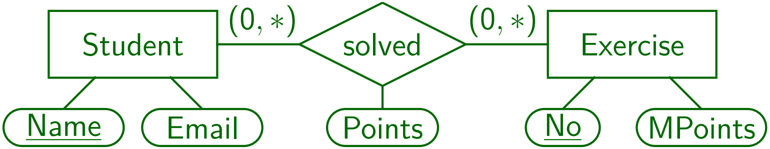
- The translation is done in the same way: The key of the weak entity type now consists of the keys of the two parent entity types (i.e. the two foreign keys added to implement the relationships):

  STUDENTS($\underline{\text{NAME}}$, EMAIL^o)
  EXERCISES($\underline{\text{NO}}$, MPOINTS)
  SOLUTIONS($\underline{\text{NAME}}\rightarrow$STUDENTS,
  $\underline{\text{NO}}\rightarrow$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:

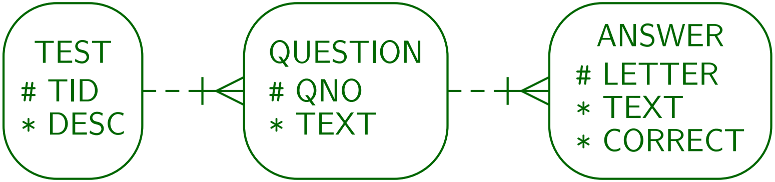


- This demonstrates again that the two ER-schemas are equivalent.

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(<u>TID</u>, DESC)
  QUESTIONS(<u>TID</u>→TESTS, <u>QNO</u>, TEXT)
  ANSWERS((<u>TID</u>, <u>QNO</u>)→QUESTIONS,
  <u>LETTER</u>, 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.

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.