TU München, Fakultät für Informatik
Lehrstuhl III: Datenbanksysteme
Prof. Alfons Kemper, Ph.D.

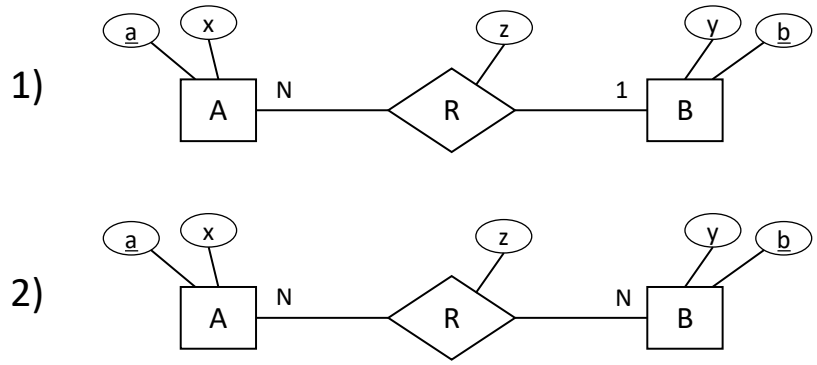
Exercise for *Database System Concepts for Non-Computer Scientist* im WiSe 19/20

Alexander van Renen (renen@in.tum.de)
http://db.in.tum.de/teaching/ws1920/DBSandere/?lang=en

Sheet 06

Exercise 1

Consider the following ER-diagram:



Refine and transform this diagram into a database schema (SQL DDL). You can assume that each attribute is an integer. Use **not null**, **primary key**, **references**, **unique** and **cascade** when possible/necessary.

Solution:

1)

```
create table A (
    a int not null primary key,
    x int
);

create table B (
    b int not null primary key,
    y int
);

create table R (
    a int not null references A primary key,
    b int not null references B,
    z int
);
```

Alternatively, we can merge the R relation into the A relation. Remember, a relationship can be merged into the entity with the same key or (graphically) the one on the "N"-side.

```
create table A (
    a int not null primary key,
    x int,
    b int references B,
    z int
);
```

The downside of this is that we now have information about the relation R inside of the A relation (namely, the z and b attribute). If R is a sparse relationship (not many tuples in A are connected to B), then we end up with a lot of null values.

Also take note of how the foreign keys in R (that is, $R.a$ and $R.b$) are marked as *not null*. Once R is merged into A, the reference to B (that is, $A.b$) becomes *nullable*. The reason for this is that the entity relationship diagram specifies that each tuple in A has zero or one partners in B. We need the *nullable* for $A.b$, otherwise zero partners would not work. This is not necessary if we translate R as its own relation because in this case the "zero partner"-case is expressed by simply not having a tuple in R.

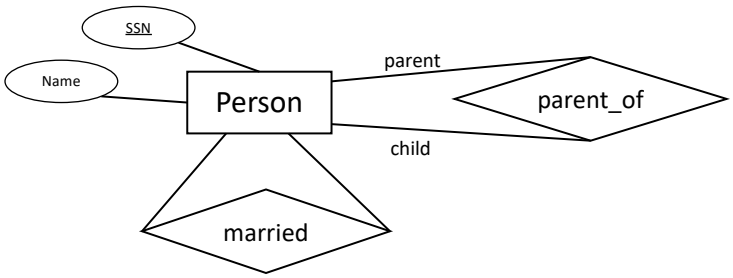
2)

```
create table A (
    a int not null primary key,
    x int
);

create table B (
    b int not null primary key,
    y int
);

create table R (
    a int not null references A,
    b int not null references B,
    z int,
    primary key(a, b)
);
```

Here we can not merge R into any of the relations. Remember, N-to-M relationships can not be refined. Note that we have to use $R.a$ and $R.b$ together as the primary key because neither attribute is unique on its own. In addition, we actually have to use this primary key for a correct translation of the entity relationship model. Otherwise, we have the following problem: Assume a relation A with values (1, 1) and B with (2, 2). Without a primary key on a and b in R, we could have the entries (a=1, b=2, z1) and (a=1, b=2, z2). This would mean that one entry in A can map to the same entry in B multiple times, which is not allowed in entity relationship models. Therefore, we have to exclude this and use the primary key constraint in R.

Exercise 2

In the following ER-diagram, we model people (*person*). The *married* relation models the German law (i.e., each person can have at most one spouse). The *parent_of* is to be interpreted in the traditional biological way (i.e., each person has exactly one mother and one father).



First, add min/max to the diagram. Then, create SQL-statements that would create the corresponding tables in a database system. Use **not null**, **primary key**, **references**, **unique** and **cascade** when possible/necessary.

Solution:

The following SQL statements created the desired tables:

```
create table Person (
    ssn char(9) not null primary key,
    name varchar(50) not null
);

create table married (
    ssn_partner_1 char(9) not null references person(ssn) on
       delete cascade,
    ssn_partner_2 char(9) not null references person(ssn) on
       delete cascade,
    unique (ssn_partner_1),
    primary key (ssn_partner_2)
);

create table parent_of (
    ssn_child char(9) not null references person(ssn) on
       delete cascade,
    ssn_parent char(9) not null references person(ssn) on
       delete cascade,
    primary key(ssn_child, ssn_parent)
);
```

In DB2 (IBM's flagship database system) all attributes which are part of the primary key have to be marked explicitly as *not null*, implicitly they are nullable. In general, making the primary key not nullable makes a lot of sense, as we do want actual values in our key attributes to be able to identify tuples. The SQL standard from 1992 (SQL92) actually enforces this rule and requires that all attributes that are part of a primary key are implicitly *not null* in the system. However, some database systems fail to implement this correctly and, therefore, diverge from the standard (e.g., DB2). Therefore, it might be a smart choice to explicitly declare the primary key attributes as *not null* and thus be on the safe side.

In the *married* relation, we can choose either of the partners to be the *primary key*. The other one should be marked as unique, this way we enforce that each person is only married

to one other person.

In the *parent_of* relation, we have to use both *child* and *parent* as the primary key. Using only the *child* attribute does not work because every person has two parents and therefore occurs twice in the *parent_of* relation. Using only the *parent* attribute fails in a similar wary: each person can have multiple children and can therefore occur any number of times in the *parent_of* relation.

As discussed in the lecture there are some alternatives for translating the entity relationship model into SQL statements. For example the *on delete cascade* semantic in the *married* relation could be left out. In this case it would not be possible to remove people from the *person* table as long as they are still married.

Another option for such a change, would be to merge the *married* relation into the *person* relation. This is possible because *married* is a 1-to-1 relationship. However, this would lead to many *null* in the *person* relation, as many people are not married. This would waste space and is therefore not to be recommended.

One change, that would actually change the entity relationship model (and is therefore not part of the question, but an interesting alternative) is to make the *parent_of* relation into a ternary relation with *mother*, *father*, and *child*. We could then translate the relation as follows:

```
create table parent_of (
    ssn_child char(9) not null references person(ssn) on
       delete cascade,
    ssn_monther char(9) references person(ssn) on delete
       set null,
    ssn_father char(9) references person(ssn) on delete
       set null,
    primary key(child)
);
```

Exercise 3

Write **relational algebra** queries for following tasks on our university database:

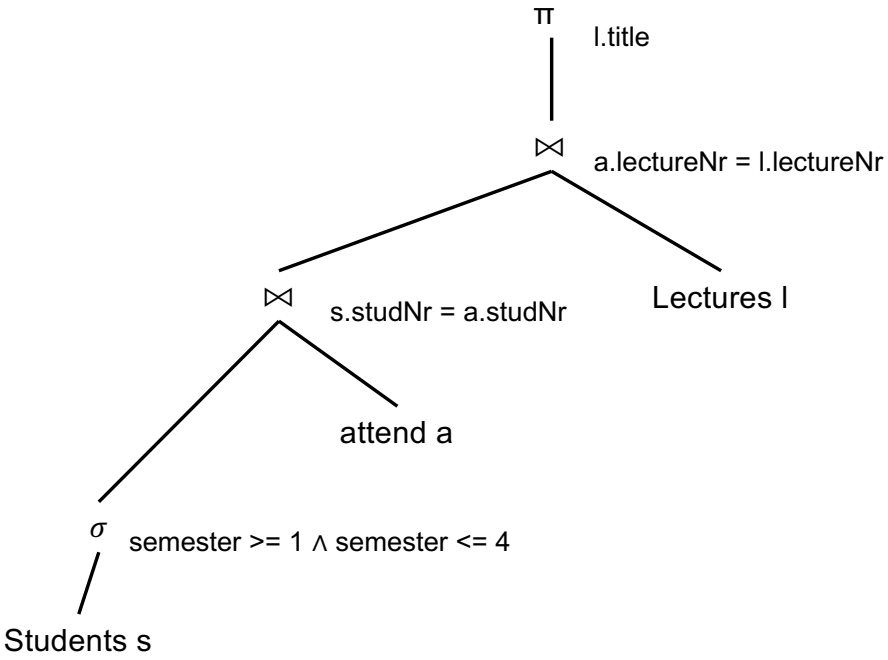
(a) Which *Lectures* are attended by *Students* in the 1.-4. semester? Only output the title of those *lectures*.

(b) Find all *Students* that know Sokrates from a *Lecture*.

(c) Find all *Students* that attend at least one *Lecture* together with Fichte.

Solution:

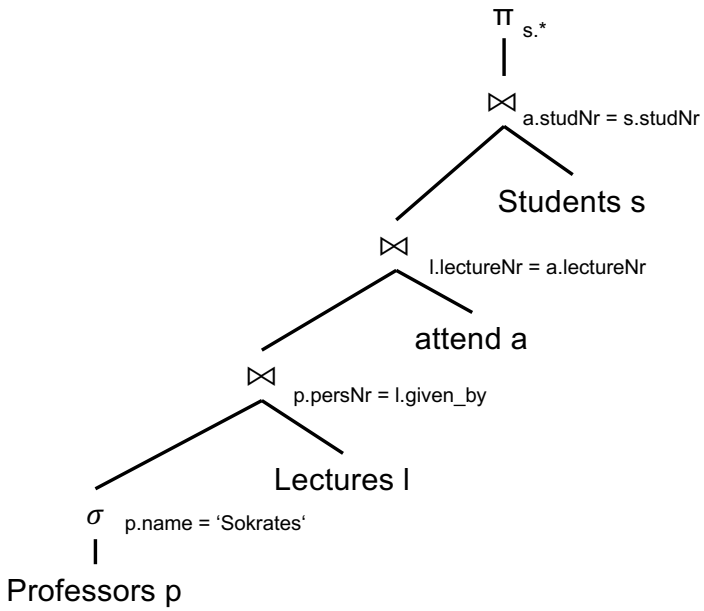
(a) Which *Lectures* are attended by *Students* in the 1.-4. semester? Print out the title of those *lectures*.



Or in SQL:

```
select distinct l.title
from Lectures l, attend a, Students s
where l.lectureNr = a.lectureNr
  and a.studNr = s.studNr
  and s.semester between 1 and 4;
```

(b) Find all *Students* that know Sokrates from a *Lecture*.



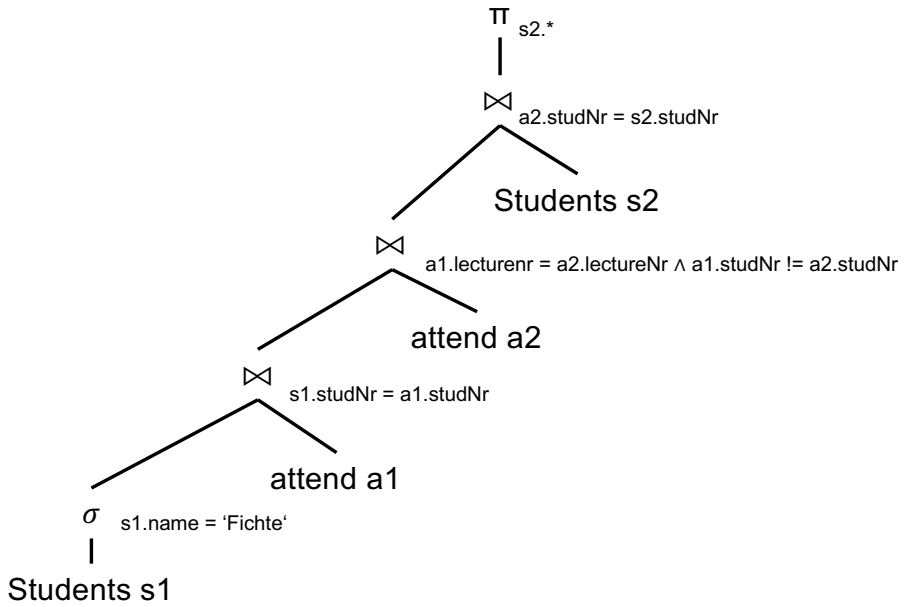
Or in SQL:

```
select distinct s.studNr, s.name
from Students s, attend a, Lectures l, Professors p
where s.studNr = a.studNr
  and a.lectureNr = l.lectureNr
  and l.given_by = p.persNr
  and p.name = 'Sokrates';
```

`DISTINCT` is necessary if we do not want duplicates in our result. Alternatively, we could also write the query using nested queries. However, by looking at the resulting query it should be clear that the former one should be preferred, as it is easier to read and understand.

```
select distinct s.studNr, s.name
from students s
where s.studNr in
  (select a.studNr
   from attend a
   where a.lectureNr in
    (select l.lectureNr
     from lectures l
     where l.given_by in
      (select p.persNr
       from professors p
       where p.name = 'Sokrates'
      )
    )
  )
```

(c) Find all *Students* that attend at least one *Lecture* together with Fichte.




```
select distinct other_s.studNr, other_s.name
from Students fichte_s, attend fichte_a, attend
   other_a, Students other_s
```

```
where fichte_s.name = 'Fichte'
  and fichte_a.studNr = fichte_s.studNr
  and other_a.lectureNr = fichte_a.lectureNr
  and other_s.studNr = other_a.studNr
  and other_s.studNr <> fichte_s.studnr
```

Exercise 4

Write the following queries in **SQL** on the known university schema:

(a) Find all students that are in the third semester.

(b) Figure out if there is a lecture with more than five *weeklyhours*.

(c) Print out a list with all professor names and avoid duplicates.

Solution:

(a) Find all students that are in the third semester.

```
select * from students where semester = 3;
```

(b) Figure out if there is a lecture with more than five *weeklyhours*.

```
select * from lectures where weeklyhours > 5;
```

→ No.

(c) Print out a list with all professor names and avoid duplicates.

```
select distinct name from professors;
```