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

Übung zur Vorlesung *Grundlagen: Datenbanken* im WS23/24

Christoph Anneser, Michael Jungmair, Stefan Lehner, Moritz Sichert, Lukas Vogel
(gdb@in.tum.de)
https://db.in.tum.de/teaching/ws2324/grundlagen/

Blatt Nr. 06

Hausaufgabe 1

Gegeben sei ein erweitertes Universitätsschema mit den folgenden zusätzlichen Relationen *StudentenGF* und *ProfessorenF*:

$$\begin{aligned} \text{StudentenGF} &: \{[\underline{\text{MatrNr} : \text{integer}}, \text{Name} : \text{varchar}(20), \text{Semester} : \text{integer}, \\ &\quad \text{Geschlecht} : \text{char}, \text{Fakultaet} : \text{varchar}(20)]\} \\ \text{ProfessorenF} &: \{[\underline{\text{PersNr} : \text{integer}}, \text{Name} : \text{varchar}(20), \text{Rang} : \text{char}(2), \\ &\quad \text{Raum} : \text{integer}, \text{Fakultaet} : \text{varchar}(20)]\} \end{aligned}$$

Die erweiterten Tabellen sind auch in der Webschnittstelle angelegt.

a) Ermitteln Sie den Männeranteil an den verschiedenen Fakultäten in SQL!

b) Ermitteln Sie in SQL die Studenten, die alle Vorlesungen ihrer Fakultät hören. Geben Sie zwei Lösungen an, höchstens eine davon darf auf Abzählen basieren.

Lösung:

a)
```
with
FakultaetTotal as (
  select Fakultaet, count(*) as Total
  from StudentenGF
  group by Fakultaet),
FakultaetMaenner as (
  select Fakultaet,count(*) as Maenner
  from StudentenGF
  where Geschlecht = 'M'
  group by Fakultaet)
select
   FakultaetTotal.Fakultaet,
   (case when Maenner is null then 0 else Maenner end)/(total*1.00)
from FakultaetTotal left outer join FakultaetMaenner
on FakultaetTotal.Fakultaet = FakultaetMaenner.Fakultaet
```

Wir müssen beachten, dass nicht jede Fakultät Männer beherbergt, weswegen diese Fakultäten (in der Standardausprägung im SQL Interface ist dies für Theologie der Fall) dann aus dem Ergebnis herausfallen würden. Aus diesem Grund verwenden wir einen `left outer join` um die Zahl der Männer und die Zahl der Studenten insgesamt

zu verbinden, wodurch auch die Theologie-Fakultät im Ergebnis enthalten ist, auch wenn es keine Männer gibt.

Das `case`-Konstrukt dient in der oberen Anfrage dazu, den NULL Wert, die durch den Left Join für die Anzahl der Männer entstehen, wenn es keine Männer gibt, durch die Zahl 0 zu ersetzen. Alternativ ist dies möglich, indem man `coalesce(maenner,0) / (total*1.00)` verwendet.

Alternativ können wir das `case`-Konstrukt verwenden, um die Anzahl der Männer an den jeweiligen Fakultäten zu ermitteln. Den Männeranteil erhalten wir dann, indem wir die Anzahl der Männer durch die Gesamtanzahl der Studenten an der Fakultät teilen.

```
select Fakultaet,
 (sum(case when Geschlecht = 'M' then 1.00 else 0.00 end)) / count(*)
from StudentenGF
group by Fakultaet
```

b) Wir fordern hier, dass es keine Vorlesung an der Fakultät des Studenten (d.h. von einem Professor der gleichen Fakultät gelesen) geben darf, die vom Studenten nicht gehört wird.

```
select s.*
from StudentenGF s
where not exists (select *
                  from Vorlesungen v, ProfessorenF p
                  where v.gelesenVon = p.PersNr
                    and p.Fakultaet = s.Fakultaet
                    and not exists
                       (select *
                        from hoeren h
                        where h.VorlNr = v.VorlNr
                          and h.MatrNr = s.MatrNr));
```

Alternativ:

```
select * from StudentenGF s
where
(select count(*)
from Vorlesungen v, ProfessorenF p
where v.gelesenVon = p.PersNr and p.Fakultaet = s.Fakultaet)
=
(select count(*)
from hoeren h, Vorlesungen v, ProfessorenF p
where
   h.MatrNr = s.MatrNr and
   h.VorlNr = v.VorlNr and
   p.PersNr = v.gelesenVon and
   p.Fakultaet = s.Fakultaet
)
```

Hausaufgabe 2

Klausuraufgabe aus dem WiSe 2016/17:

Gegeben sei das bekannte Uni-Schema. Formulieren Sie in SQL-92: Finden Sie alle Vorlesungen **(VorlNr und Titel duplikatfrei ausgeben)**, die nicht vor dem dritten Semester gehört werden sollten. – Ein Beispiel hierfür ist die Vorlesung Bioethik (5216), da diese die Vorlesung Ethik (5041) voraussetzt, welche wiederum die Vorlesung Grundzüge (5001) als Voraussetzung hat.

Lösung:

Ohne Rekursion:

```
SELECT DISTINCT v.VorlNr, v.Titel
FROM Vorlesungen v, voraussetzen v1, voraussetzen v2
WHERE v.VorlNr = v1.Nachfolger
  AND v1.Vorgaenger = v2.Nachfolger
```

Mit Rekursion:

```
WITH recursive vor as (
  SELECT *, 1 as cnt FROM voraussetzen
  UNION
  SELECT v1.vorgaenger, v2.nachfolger, v1.cnt + 1
  FROM vor v1, voraussetzen v2
  WHERE v1.nachfolger = v2.vorgaenger
  )
SELECT DISTINCT nachfolger, v.Titel
FROM vor, vorlesungen v
WHERE vor.nachfolger = v.VorlNr AND vor.cnt >= 2
```

Hausaufgabe 3

Gegeben sei die Relation `Fahrplan`, die strukturell dem folgenden Beispiel gleicht:

Von	Nach	Linie	Abfahrt	Ankunft
Garching, Forschungszentrum	Garching	U6	09:06	09:09
Garching	Garching-Hochbrück	U6	09:09	09:11
Garching-Hochbrück	Fröttmaning	U6	09:11	09:15
...	...			
Fröttmaning	Garching-Hochbrück	U6	09:00	09:04
Garching-Hochbrück	Garching	U6	09:04	09:06
Garching	Garching, Forschungszentrum	U6	09:06	09:09
...	...			
Garching, Forschungszentrum	Technische Universität	690	17:56	17:57

Formulieren Sie die folgenden Anfragen auf diese Relation in SQL. Sie können die Typen `TIME` für Uhrzeiten und `INTERVAL` für Zeitintervalle verwenden.

Schreiben Sie z.B. für 10:30 Uhr: `TIME '10:30:00'`.

Das 0-Intervall kann z.B. so konstruiert werden: `INTERVAL '00:00:00'`.

a) Geben Sie eine Anfrage an, welche für alle Stationen ermittelt, welche **anderen** Stationen erreicht werden können. Beachten Sie, dass nur tatsächlich mögliche Verbindungen ausgegeben werden sollen, d.h. die Abfahrt an einer Haltestelle darf nicht vor der Ankunft liegen.

b) Erweitern Sie ihre Anfrage aus Teilaufgabe a), sodass zusätzlich die summierte Fahrtzeit und Wartezeit sowie die gesamte Reisezeit ausgegeben wird. Die Fahrtzeit ist dabei nur die Zeit, in der man sich in einem Verkehrsmittel befindet. Die Wartezeit ist die Zeit, die bei einem Umstieg zwischen Ankunft des alten und Abfahrt des neuen Verkehrsmittels vergeht. Die Reisezeit ist die Zeit zwischen Abfahrt des ersten und Ankunft des letzen Verkehrsmittels.

c) Erweitern Sie ihre Anfrage aus Teilaufgabe a) oder b) nochmals und geben Sie die Anzahl der Umstiege für jede Verbindung aus.

d) Finden Sie die „guten“ Verbindungen, um von Fröttmaning pünktlich zur Vorlesung „Grundlagen: Datenbanken“ um 10:30 Uhr zu kommen. Verwenden Sie dazu Ihre Anfrage aus Teilaufgabe c). Eine Verbindung ist „gut“, wenn sie spätestens um 10:30 in „Garching, Forschungszentrum“ ist und es keine andere Verbindung gibt, die später abfährt aber noch rechtzeitig eintrifft, deren Reisezeit geringer ist und bei der man weniger Umstiege hat.

Lösung:

a) Geben Sie eine Anfrage an, welche für alle Stationen ermittelt, welche **anderen** Stationen erreicht werden können. Beachten Sie, dass nur tatsächlich mögliche Verbindungen ausgegeben werden sollen, d.h. die Abfahrt an einer Haltestelle darf nicht vor der Ankunft liegen.

   Für diese Aufgabe werden *rekursive Sichten* mit `with recursive` benötigt. Die Relation `Fahrplan` bildet dort den Rekursionsanfang. Dann werden rekursiv mit `union all` weitere Verbindungen hinzugefügt. Eine neue Verbindung soll nur dann eingefügt werden, wenn die Ankunftshaltestelle gleich der Abfahrtshaltestelle der neuen Teilverbindung ist, wenn diese zeitlich erreichbar ist und wenn die neue Ankunftshaltestelle ungleich der ursprünglichen Abfahrtshaltestelle ist. Dadurch ergeben sich die `where`-Bedingungen.

```
with recursive fahrplan_rec as (
    select von, nach, abfahrt, ankunft from fahrplan
    union all
    select fr.von, f.nach, fr.abfahrt, f.ankunft
    from fahrplan_rec fr, fahrplan f
    where
        fr.nach = f.von and
        fr.ankunft <= f.abfahrt and
        fr.von != f.nach
)
```

```
select * from fahrplan_rec
```

b) Erweitern Sie ihre Anfrage aus Teilaufgabe a), sodass zusätzlich die summierte Fahrtzeit und Wartezeit sowie die gesamte Reisezeit ausgegeben wird. Die Fahrtzeit ist dabei nur die Zeit, in der man sich in einem Verkehrsmittel befindet. Die Wartezeit ist die Zeit, die bei einem Umstieg zwischen Ankunft des alten und Abfahrt des neuen Verkehrsmittels vergeht. Die Reisezeit ist die Zeit zwischen Abfahrt des ersten und Ankunft des letzen Verkehrsmittels.

Ausgehend von der Lösung von Teilaufgabe a) muss hier als Rekursionsanfang zusätzlich die Fahrzeit und die Wartezeit gesetzt werden. Bei einer Direktverbindung ergibt sich die Fahrtzeit aus der Differenz zwischen Ankunft und Abfahrt, die Wartezeit ist am Anfang 0. Im Rekursionsschritt muss nun die Fahrtzeit der neuen Teilverbindung der aktuellen Fahrtzeit hinzuaddiert werden. Ähnlich muss zur Wartezeit die Dauer zwischen der Ankunft der aktuellen Verbindung und der Abfahrt der neuen hinzuaddiert werden. Da sich die Reisezeit aus der Summe der Fahrtzeit und der Wartezeit ergibt, muss diese nicht notwendigerweise schon während der Rekursion berechnet werden, sondern kann durch eine weitere Sicht berechnet werden.

```
with recursive fahrplan_rec_linie as (
    select
        von,
        nach,
        abfahrt,
        ankunft,
        ankunft - abfahrt as fahrtzeit,
        INTERVAL '00:00:00' as wartezeit
    from fahrplan
    union all
    select
        fr.von,
        f.nach,
        fr.abfahrt,
        f.ankunft,
        fr.fahrtzeit + (f.ankunft - f.abfahrt),
        fr.wartezeit + (f.abfahrt - fr.ankunft)
    from fahrplan_rec_linie fr, fahrplan f
    where
        fr.nach = f.von and
        fr.ankunft <= f.abfahrt and
        fr.von != f.nach
),
fahrplan_rec as (
    select
        von,
        nach,
        abfahrt,
        ankunft,
        fahrtzeit,
```

```
        wartezeit,
        fahrtzeit + wartezeit as reisezeit
    from fahrplan_rec_linie
)
select * from fahrplan_rec
```

c) Erweitern Sie ihre Anfrage aus Aufgabe a) oder b) nochmals und geben Sie die Anzahl der Umstiege für jede Verbindung aus.

Man muss immer genau dann umsteigen, wenn durch den Rekursionsschritt eine neue Teilverbindung hinzukommt, die zu einer anderen Linie gehört als der aktuellen, oder wenn die Ankunftszeit der bisherigen Verbindung kleiner als die Abfahrtszeit der neuen Teilverbindung ist. Um ersteres zu erkennen, muss in der rekursiven Sicht immer die aktuelle Linie mitgeführt werden. Diese wird im Rekursionsanfang auf die Linie der Teilverbindung gesetzt. Im Rekursionsschritt wird dann mithilfe von `case` ermittelt, ob ein Umstieg stattfindet oder nicht. Im Rekursionsschritt muss auch die aktuelle Linie aktualisiert werden, damit die Erkennung von Umstiegen weiterhin korrekt funktioniert.

```
with recursive fahrplan_rec_linie as (
    select
        von,
        nach,
        abfahrt,
        ankunft,
        linie as aktuelle_linie,
        0 as umstiege,
        ankunft - abfahrt as fahrtzeit,
        INTERVAL '00:00:00' as wartezeit
    from fahrplan
    union all
    select
        fr.von,
        f.nach,
        fr.abfahrt,
        f.ankunft,
        f.linie,
        fr.umstiege + case
            when
                f.linie != fr.aktuelle_linie or
                f.abfahrt > fr.ankunft
            then 1
            else 0
        end,
        fr.fahrtzeit + (f.ankunft - f.abfahrt),
        fr.wartezeit + (f.abfahrt - fr.ankunft)
    from fahrplan_rec_linie fr, fahrplan f
    where
        fr.nach = f.von and
```

```
        fr.ankunft <= f.abfahrt and
        fr.von != f.nach
),
fahrplan_rec as (
    select
        von,
        nach,
        abfahrt,
        ankunft,
        umstiege,
        fahrtzeit,
        wartezeit,
        fahrtzeit + wartezeit as reisezeit
    from fahrplan_rec_linie
)
select * from fahrplan_rec
```

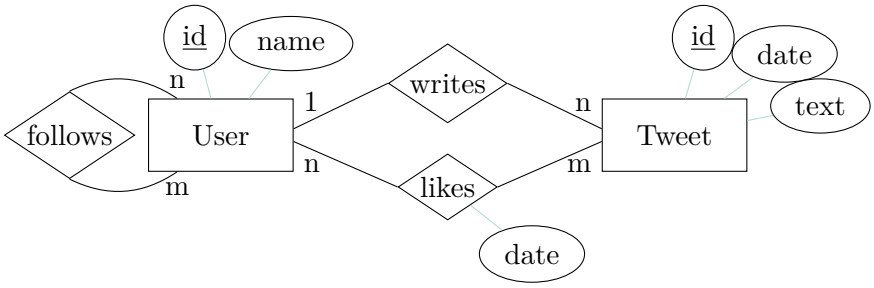
d) Finden Sie die „guten“ Verbindungen, um von Fröttmaning pünktlich zur Vorlesung „Grundlagen: Datenbanken“ um 10:30 Uhr zu kommen. Verwenden Sie dazu Ihre Anfrage aus Teilaufgabe c). Eine Verbindung ist „gut“, wenn sie spätestens um 10:30 in „Garching, Forschungszentrum“ ist und es keine andere Verbindung gibt, die später abfährt aber noch rechtzeitig eintrifft, deren Reisezeit geringer ist und bei der man weniger Umstiege hat.

Ausgehend von der Lösung von Teilaufgabe c) kann diese Aufgabe mit `not exists` gelöst werden:

```
select * from fahrplan_rec fr
where
    fr.von = 'Fröttmaning' and
    fr.nach = 'Garching,␣Forschungszentrum' and
    fr.ankunft <= TIME '10:30:00' and
    not exists (
        select * from fahrplan_rec fr2
        where
            fr2.von = fr.von and
            fr2.nach = fr.nach and
            fr2.ankunft <= TIME '10:30:00' and
            fr2.abfahrt > fr.abfahrt and
            fr2.reisezeit < fr.reisezeit and
            fr2.umstiege < fr.umstiege
    )
```

Hausaufgabe 4

Gegeben sei folgendes ER-Diagramm, das User, deren Tweets, Likes und Follows modelliert, und das dazugehörige relationale Schema:



$$\begin{aligned}
\text{User}: &\ \{[\underline{\text{id}},\text{name}]\} \\
\text{Tweet}: &\ \{[\underline{\text{id}},\text{user_id}, \text{date}, \text{text}]\} \\
\text{follows}: &\ \{[\underline{\text{follower_id}, \text{follows_id}}]\} \\
\text{likes}: &\ \{[\underline{\text{user_id}, \text{tweet_id}}, \text{date}]\}
\end{aligned}$$

a) Geben Sie SQL-Statements zum Erzeugen der Relationen an. Überlegen Sie sich dazu sinnvolle Typen für die Attribute. Verwenden Sie Angaben zu `NULL` und Schlüsseln (`primary key`, `unique`).

b) Ergänzen Sie die SQL-Statements mit referentiellen Integritätsbedingungen. Es soll sichergestellt werden, dass wenn ein User gelöscht wird, auch alle seine Follows, Follower und Likes gelöscht werden. Seine Tweets sollen aber erhalten bleiben, indem die user_id seiner Tweets auf `NULL` gesetzt wird. Wenn ein Tweet gelöscht wird, sollen ebenfalls dessen Likes gelöscht werden.

c) Fügen Sie statische Integritätsbedingungen hinzu, die folgende Eigenschaften garantieren:

- Wenn die user_id eines Tweets `NULL` ist, muss der Text des Tweets „removed“ lauten
- Das Datum eines Likes darf nicht vor dem Datum des Tweets liegen.

Lösung:

a) Geben Sie SQL-Statements zum Erzeugen der Relationen an. Überlegen Sie sich dazu sinnvolle Typen für die Attribute. Verwenden Sie Angaben zu `NULL` und Schlüsseln (`primary key`, `unique`).

Da „User“, „date“ und „text“ von manchen Datenbanken nicht als Namen für Relationen oder Attribute erlaubt sind, sind die betroffenen Namen hier geändert.

Für alle id-Attribute wird der Typ `integer` verwendet. Falls 2^{32} IDs nicht ausreichen sollten, kann stattdessen auch `bigint` verwendet werden. Die Attribute „name“ und „tweet_text“ haben beide den Typen `varchar`, da es sich um einen kurzen Text variabler Länge handelt. Die Datumsattribute haben hier den Typen `timestamp` der ein Datum mit Zeitangabe darstellt statt dem in der Vorlesung vorgestellten Typen `date`, mit dem nur Tage dargestellt werden können.

Generell sollten Attribute so selten wie möglich `NULL` sein, weswegen alle Attribute auf `not null` gesetzt werden. Nur das Attribut „user_id“ der Relation „Tweet“ kann potentiell `NULL` sein, was aber erst in Aufgabe b) verlangt wird.

Die Primärschlüssel sind in dem relationalen Schema schon unterstrichen und müssen in den SQL-Statements beachtet werden. Bei Primärschlüsseln mit nur einem Attribut, kann `primary key` direkt an das Attribute angehängt werden, ansonsten muss der Schlüssel in einer neuen Zeile aufgeführt werden. Zusätzlich sollten Namen eindeutig sein, weswegen das Attribute „name“ den Zusatz `unique` erhält.

```
create table Twitter_User (
    id integer not null primary key,
    name varchar(50) not null unique
);
create table Tweet (
    id integer not null primary key,
    user_id integer null references Twitter_User,
    tweet_date timestamp not null,
    tweet_text varchar(500) not null
);
create table follows (
    follower_id integer not null references Twitter_User,
    follows_id integer not null references Twitter_User,
    primary key (follower_id, follows_id)
);
create table likes (
    user_id integer not null references Twitter_User,
    tweet_id integer not null references Tweet,
    like_date timestamp not null,
    primary key (user_id, tweet_id)
);
```

b) Ergänzen Sie die SQL-Statements mit referentiellen Integritätsbedingungen. Es soll sichergestellt werden, dass wenn ein User gelöscht wird, auch alle seine Follows, Follower und Likes gelöscht werden. Seine Tweets sollen aber erhalten bleiben, indem die user_id seiner Tweets auf `NULL` gesetzt wird. Wenn ein Tweet gelöscht wird, sollen ebenfalls dessen Likes gelöscht werden.

An allen Stellen, wo User oder Tweets mit `references` referenziert werden, muss entweder `on delete cascade` oder `on delete set null` hinzugefügt werden.

```
create table Twitter_User (
    id integer not null primary key,
    name varchar(50) not null unique
);
create table Tweet (
    id integer not null primary key,
    user_id integer null references Twitter_User on delete set null,
    tweet_date timestamp not null,
    tweet_text varchar(500) not null
);
create table follows (
    follower_id integer not null references Twitter_User on delete cascade,
    follows_id integer not null references Twitter_User on delete cascade,
```

```
    primary key (follower_id, follows_id)
);
create table likes (
    user_id integer not null references Twitter_User on delete cascade,
    tweet_id integer not null references Tweet on delete cascade,
    like_date timestamp not null,
    primary key (user_id, tweet_id)
);
```

c) Fügen Sie statische Integritätsbedingungen hinzu, die folgende Eigenschaften garantieren:

- Wenn die user_id eines Tweets `NULL` ist, muss der Text des Tweets „removed“ lauten
- Das Datum eines Likes darf nicht vor dem Datum des Tweets liegen.

Für die erste Eigenschaft muss eine `check`-Bedingung zur Relation Tweet hinzugefügt werden:

```
create table Tweet (
    id integer not null primary key,
    user_id integer null references Twitter_User on delete set null,
    tweet_date timestamp not null,
    tweet_text varchar(500) not null,
    check (user_id is not null or tweet_text = 'removed')
);
```

Für die zweite Eigenschaft wird eine `check`-Bedingung mit einer Unterabfrage in der Relation likes benötigt.

```
create table likes (
    user_id integer not null references Twitter_User on delete cascade,
    tweet_id integer not null references Tweet on delete cascade,
    like_date timestamp not null,
    primary key (user_id, tweet_id),
    check (exists (
        select *
        from Tweet t
        where
            t.id = tweet_id and
            t.tweet_date <= like_date
    ))
);
```