



Übung zur Vorlesung *Grundlagen: Datenbanken* im WS18/19

Moritz Sichert, Lukas Vogel (gdb@in.tum.de)

<https://db.in.tum.de/teaching/ws1819/grundlagen/>

Blatt Nr. 06

Tool zum Üben von SQL-Anfragen: <https://hyper-db.com/interface.html>.

Hausaufgabe 1

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: `cast('00:00:00' as INTERVAL)`.

- 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.
- 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 letzten Verkehrsmittels.
- Erweitern Sie ihre Anfrage aus Teilaufgabe a) oder b) nochmals und geben Sie die Anzahl der Umstiege für jede Verbindung aus.
- 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 letzten 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,
        cast('00:00:00' as interval) 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,
        cast('00:00:00' as interval) 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

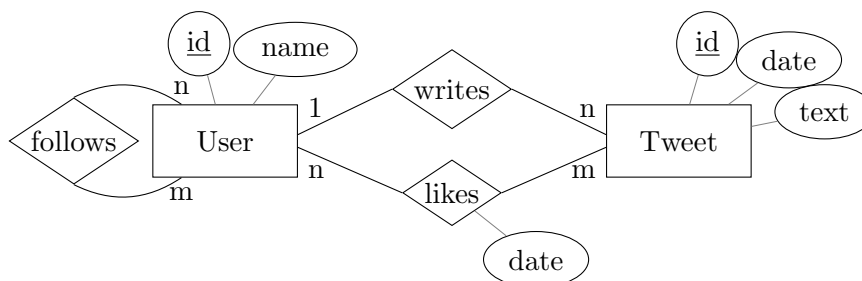
weniger Umstiege hat.

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 2

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



User : { [id,name] }
Tweet : { [id,user_id, date, text] }
follows : { [follower_id, follows_id] }
likes : { [user_id, tweet_id, date] }

- 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).
- 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 referenzielle 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,  
    primary key (user_id, tweet_id)
```

```

        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 referenzielle 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
    ))
);

```

Hausaufgabe 3

Schreiben Sie eine SQL-Anfrage, die das kleine Einmaleins¹ in Tabellenform ausgibt. Tipp: Verwenden Sie WITH ... (VALUES ...).

Lösung:

Das komplette Einmaleins kann direkt in einen VALUES-Ausdruck gefasst werden und ausgegeben werden:

```

with einmaleins as (values
    ( 1,  2,  3,  4,  5,  6,  7,  8,  9, 10),
    ( 2,  4,  6,  8, 10, 12, 14, 16, 18, 20),
    ( 3,  6,  9, 12, 15, 18, 21, 24, 27, 30),
    ( 4,  8, 12, 16, 20, 24, 28, 32, 36, 40),
    ( 5, 10, 15, 20, 25, 30, 35, 40, 45, 50),
    ( 6, 12, 18, 24, 30, 36, 42, 48, 54, 60),
    ( 7, 14, 21, 28, 35, 42, 49, 56, 63, 70),
    ( 8, 16, 24, 32, 40, 48, 56, 64, 72, 80),
    ( 9, 18, 27, 36, 45, 54, 63, 72, 81, 90),
    (10, 20, 30, 40, 50, 60, 70, 80, 90, 100)
)
select * from einmaleins

```

¹<https://de.wikipedia.org/wiki/Einmaleins>

Alternativ kann das Einmaleins auch von der Datenbank berechnet werden, was ein wenig Schreibarbeit spart:

```
with einmaleins_spalten(a, b, c, d, e, f, g, h, i, k) as (values
    (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
),
einmaleins_zeilen(x) as (values
    (1), (2), (3), (4), (5), (6), (7), (8), (9), (10)
)
select
    a * x,
    b * x,
    c * x,
    d * x,
    e * x,
    f * x,
    g * x,
    h * x,
    i * x,
    k * x
from einmaleins_spalten, einmaleins_zeilen
```