Einführung in Datenbanken

Kapitel 10: Logik in SQL, Teil II: Quantoren, Unteranfragen

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Wintersemester 2020/21

http://www.informatik.uni-halle.de/~brass/db20/

Lernziele

Nach diesem Kapitel sollten Sie Folgendes können:

- „Monotone Anfrage“ definieren. Anfragen klassifizieren.
- Logische Äquivalenzen anwenden, um z.B. aus einer „für alle“-Bedingung eine `NOT EXISTS`-Unteranfrage in SQL zu machen.
- Korrelierte/unkorrelierte Unteranfragen in SQL unterscheiden, den Fehler „unkorrelierte `EXISTS`-Unteranfrage“ vermeiden.
- Regeln für die Sichtbarkeit von Tupelvariablen in verschiedenen Teilen der Anfrage kennen und anwenden,

  „Verschattete Tupelvariablen“ erkennen und vermeiden.
- Regeln für die Möglichkeit, auf Attribute ohne explizite Tupelvariable zuzugreifen, auch in Unteranfragen kennen.
- `IN`-Unteranfragen anwenden, `>= ALL`, Unteranfragen als Terme.

Inhalt

Beispiel-Datenbank

STUDENTEN

SID	VORNAME	NACHNAME	EMAIL
101	Lisa	Weiss	···
102	Michael	Grau	NULL
103	Daniel	Sommer	···
104	Iris	Winter	···

BEWERTUNGEN

SID	ATYP	ANR	PUNKTE
101	H	1	10
101	H	2	8
101	Z	1	12
102	H	1	9
102	H	2	9
102	Z	1	10
103	H	1	5
103	Z	1	7

AUFGABEN

ATYP	ANR	THEMA	MAXPT
H	1	ER	10
H	2	SQL	10
Z	1	SQL	14

Einschränkungen bisheriger Anfragen (1)

- Bisher wurden nur SQL-Anfragen ohne Unteranfragen behandelt (also mit einer quantorenfreien Formel als `WHERE`-Bedingung).

  Damit kann man schon viele interessante Anfragen stellen, aber es gibt auch wesentliche Einschränkungen.

- Eine abstrakte Anfrage (für ein Schema $\mathcal{SCH}$) ist eine Abbildung von Datenbank-Zuständen (zu $\mathcal{SCH}$) in Antwort-Tabellen.

- Wenn man eine Anfragesprache studiert, kann man sich fragen, welche solchen abstrakten Anfragen in der Sprache syntaktisch ausdrückbar sind.

  Die Semantik einer SQL-Anfrage ist ja gerade so eine Abbildung, d.h. abstrakte Anfrage.

Einschränkungen bisheriger Anfragen (2)

- Bisher war es so, dass ein Antworttupel von einer festen Anzahl Tupel in der Datenbank erzeugt wird.
- Wenn eine Variablenbelegung $\mathcal{A}$ für die unter `FROM` deklarierten Tupelvariablen die `WHERE`-Bedingung erfüllt, erzeugt sie eine Ausgabe.
- Sind unter `FROM` z.B. die beiden Tupelvariablen X und Y deklariert, sind für die Ausgabe nur die beiden Tupel $\mathcal{A}(X)$ und $\mathcal{A}(Y)$ relevant:
  - Man könnte alle anderen Tabellenzeilen löschen, und beliebige neue Zeilen einfügen,
  - und würde doch dieses Antworttupel bekommen.

  Die anderen Tupel der Antwort könnten sich natürlich ändern.

Einschränkungen bisheriger Anfragen (3)

- Es gibt aber auch praktisch relevante Anfragen, bei denen ein Ausgabetupel nicht in dieser Weise von einer festen Anzahl Eingabetupel (aus der Datenbank) produziert wird, z.B.
  - Wer hat noch keine Hausaufgaben abgeben?

    Hier wird also nach der Nicht-Existenz eines Tupels gefragt.
    Das kann der bisherige Typ von Anfrage nicht leisten.
  - Wer hat alle Hausaufgaben abgegeben?

    Hier muss es für jeden Eintrag in der Tabelle `AUFGABEN` mit `ATYP = 'H'` einen passenden Eintrag in der Tabelle `BEWERTUNGEN` geben (mit der `SID` des Studenten, der als Ergebnis geliefert werden soll). Die Anzahl der Tupel in der Tabelle `BEWERTUNGEN`, die es für eine Ausgabe geben muss, hängt von der Anzahl Tupel in der Tabelle `AUFGABEN` ab.

Monotone und nichtmonotone Anfragen (1)

- Definition: Ein DB-Zustand $\mathcal{I}_1$ ist ist kleinergleich einem DB-Zustand $\mathcal{I}_2$, geschrieben $\mathcal{I}_1 \subseteq \mathcal{I}_2$, gdw. $\mathcal{I}_1(R) \subseteq \mathcal{I}_2(R)$ für alle Relationen R im Schema.
- Erläuterung: Das bedeutet, der Zustand $\mathcal{I}_2$ entsteht aus dem Zustand $\mathcal{I}_1$ durch Einfügung von Tabellenzeilen.
- Definition: Eine Anfrage Q heißt monoton gdw. für alle DB-Zustände $\mathcal{I}_1, \mathcal{I}_2$ gilt:

  $$\mathcal{I}_1 \subseteq \mathcal{I}_2 \implies \mathcal{I}_1[Q] \subseteq \mathcal{I}_2[Q].$$

  Anderfalls heißt sie nichtmonoton.
- Erläuterung: Bei einer monotonen Anfrage bekommt man nach der Einfügung also mindestens die gleichen Antworttupel wie vorher (eventuell noch zusätzliche).

Monotone und nichtmonotone Anfragen (2)

- Satz: SQL-Anfragen Q der Form

  $$\begin{array}{ll} \texttt{SELECT} & t_1, \ldots, t_k \\ \texttt{FROM} & R_1\ X_1,\ \ldots,\ R_n\ X_n \\ \texttt{WHERE} & F \end{array}$$

  wobei die Bedingung F keine Unteranfragen enthält (also eine quantorenfreie Formel ist), sind monoton.

- Das bedeutet:
  - Sei $\mathcal{I}_1$ ein DB-Zustand, und resultiere $\mathcal{I}_2$ aus $\mathcal{I}_1$ durch Einfügen von einem oder mehreren Tupeln.
  - Dann ist jedes Antworttupel t der Anfrage Q in $\mathcal{I}_1$ auch in der Antwort auf Q in $\mathcal{I}_2$ enthalten.

    D.h. korrekte Antworten bleiben auch nach Einfügungen gültig.

Monotone und nichtmonotone Anfragen (3)

- Wenn sich die gewünschte Anfrage nichtmonoton verhält, so folgt, dass die obige Form von SQL-Anfragen nicht ausreicht, man also z.B. Unteranfragen verwenden muss.

  Oder Aggregationsfunktionen wie `COUNT` (siehe Kapitel 13).

- Beispiele solcher Anfragen:
  - Welcher Student hat noch keine Übung gelöst?
  - Wer hat die meisten Punkte auf Hausaufgabe 1?
  - Wer hat alle Übungen in der Datenbank gelöst?
- Aufgabe: Geben Sie für jede dieser Fragen ein Antworttupel aus dem Beispielzustand an und für jede solche Antwort ein Tupel, das man einfügen kann, um diese Antwort ungültig zu machen (Lösung für erste Anfrage auf nächster Folie).

Monotone und nichtmonotone Anfragen (4)

- Z.B. „Geben Sie alle Studenten aus, die noch keine Hausaufgabe gelöst haben.“
  - Momentan wäre Iris Winter eine korrekte Antwort.
  - Würde man jedoch eine Bewertung für sie eingefügen, wäre dies nicht länger richtig.
- In natürlicher Sprache weisen Formulierungen wie „es gibt kein“ auf nichtmonotones Verhalten hin.
- Auch „für alle“ oder „minimale/maximale“ sind Indikatoren für nichtmonotones Verhalten: Es darf dann keine Verletzung der „für alle“-Bedingung existieren.

  Für einige solcher Anfragen könnte eine Formulierung mit Aggregationen (`HAVING`) angebracht sein, siehe Kapitel 13.

Monotone und nichtmonotone Anfragen (5)

- Bei der Formulierung einer Anfrage in SQL ist es wichtig, zu erkennen, ob die Anfrage verlangt, dass gewisse Tupel nicht existieren.
- In der Sprache QBE kann man ganze Tabellenzeilen negieren, d.h. fordern, dass es keine Zeile gibt, die auf das Muster passt:

STUDENTEN			
SID	VORNAME	NACHNAME	EMAIL
_X	P.	P.	

BEWERTUNGEN			
SID	ATYP	ANR	PUNKTE
¬ _X	H		

Beispiele für Fehler (1)

Aufgaben:

- Findet diese Anfrage Studenten ohne eine Hausaufgabe in der DB? Wenn nicht, was berechnet sie?

```
SELECT DISTINCT S.SID, S.VORNAME, S.NACHNAME
FROM   STUDENTEN S, BEWERTUNGEN B
WHERE  S.SID <> B.SID AND B.ATYP = 'H'
```

- Bekommt man so Übungen (noch) ohne Abgaben?

```
SELECT DISTINCT A.ATYP, A.ANR
FROM   AUFGABEN A, BEWERTUNGEN B
WHERE  A.ATYP = B.ATYP AND A.ANR = B.ANR
AND    B.SID IS NULL
```

Beispiele für Fehler (2)

- Liefert diese Anfrage den Studenten / die Studentin mit den meisten Punkten für Hausaufgabe 1?

```
SELECT DISTINCT S.VORNAME, S.NACHNAME, X.PUNKTE
FROM   STUDENTEN S, BEWERTUNGEN X, BEWERTUNGEN Y
WHERE  S.SID = X.SID
AND    X.ATYP = 'H' AND X.ANR = 1
AND    Y.ATYP = 'H' AND Y.ANR = 1
AND    X.PUNKTE > Y.PUNKTE
```

- Wenn nicht, was berechnet sie?

Inhalt

Erweiterung einer Variablendeklaration

Definition:

- Sei ν eine Variablendeklaration, $X \in VARS$, und $s \in \mathcal{S}$.
- Dann schreiben wir $\nu\langle X/s\rangle$ für die lokal modifizierte Variablendeklaration ν' mit

$$\nu'(V) := \begin{cases} s & \text{falls } V = X \\ \nu(V) & \text{sonst.} \end{cases}$$

Bemerkung:

- Beides ist möglich: ν kann für X schon definiert sein, oder an dieser Stelle bisher undefiniert sein.
- In SQL können Variablen durch eine Variable gleichen Namens in Unteranfragen „verschattet“ werden.

Erweiterung einer Variablenbelegung

Definition:

- $\mathcal{A}\langle X/d\rangle$ sei die Variablenbelegung $\mathcal{A}'$, die bis auf $\mathcal{A}'(X) = d$ mit $\mathcal{A}$ übereinstimmt.

Bemerkung:

- Wie bei den Variablendeklarationen braucht man auch bei den Variablenbelegungen die Möglichkeit zur lokalen Veränderung der Abbildung für eine einzelne Variable (zur Behandlung von Quantoren).

Formeln (1)

Definition:

- Sei eine Signatur $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F})$ und eine Variablendeklaration ν für Σ gegeben.
- Die Mengen $FO_{\Sigma,\nu}$ der (Σ, ν)-Formeln sind folgendermaßen rekursiv definiert:
  - Jede atomare Formel $F \in AT_{\Sigma,\nu}$ ist eine Formel.
  - Wenn F und G Formeln sind, so auch $(\neg F)$, $(F \wedge G)$, $(F \vee G)$, $(F \leftarrow G)$, $(F \rightarrow G)$, $(F \leftrightarrow G)$.
  - $(\forall\, s\, X\colon F)$ und $(\exists\, s\, X\colon F)$ sind in $FO_{\Sigma,\nu}$ falls $s \in \mathcal{S}$, $X \in VARS$, und F eine $(\Sigma, \nu\langle X/s\rangle)$-Formel ist.
  - Nichts anderes ist eine Formel.

Formeln (2)

- Die intuitive Bedeutung der quantifizierten Formeln ist wie folgt (andere Formeln wurden in Kap. 8 besprochen):
  - $\forall\, s\, X : F$: „für alle X (der Sorte s) gilt F“.
  - $\exists\, s\, X : F$: „es gibt ein X (aus s), so dass F gilt“.
- Da man im SQL-Tupelkalkül Variablen nur über Tupel-Sorten (mit endlicher Interpretation, den Tabellenzeilen) und nicht über Datentypen (wie `VARCHAR(100)`) deklarieren kann, sind die Quantoren endlich auswertbar.

  Im Bereichskalkül oder dem theoretischen Tupelkalkül, bei denen Variablen über unendliche Wertebereiche laufen, sind folgende Muster üblich, die die Variable V über eine Formel F auf einen endlichen Bereich einschränken:
  $\forall\, s\, X : F \rightarrow G$: „für alle X (der Sorte s), die F erfüllen, gilt G“.
  $\exists\, s\, X : F \wedge G$: „es gibt ein X (aus s) mit F, so dass G gilt“.
  Allgemein verwendet man in Datenbanken nur „bereichsbeschränkte Formeln“.

Formeln (3)

Formale Behandlung der Bindungsstärke:

- Eine Formel der Stufe 0 (Formel-0) ist eine atomare Formel oder eine in $(\ldots)$ eingeschlossene Formel-5.

  Die Stufe einer Formel entspricht der Bindungsstärke des äußersten Operators (kleinste Nummer bedeutet höchste Bindungsstärke).
  Man kann jedoch eine Formel-i wie eine Formel-j verwenden mit $j > i$.
  In entgegengesetzter Richtung werden Klammern benötigt.

- Eine Formel-1 ist eine Formel-0 oder eine Formel der Form $\neg F$, wobei F eine Formel-1 ist.

- Eine Formel-2 ist eine Formel-1 oder eine Formel der Form $F_1 \wedge F_2$, wobei F_1 eine Formel-2 ist, und F_2 eine Formel-1 (implizite Klammerung von links).

Formeln (4)

Formale Behandlung der Bindungsstärke, fortgesetzt:

- Eine Formel-3 ist eine Formel-2 oder eine Formel der Form $F_1 \vee F_2$ mit einer Formel-3 F_1 und einer Formel-2 F_2.
- Eine Formel-4 ist eine Formel-3 oder eine Formel der Form $F_1 \leftarrow F_2$, $F_1 \rightarrow F_2$, $F_1 \leftrightarrow F_2$, wobei F_1 und F_2 Formeln der Stufe 3 sind.
- Eine Formel-5 ist eine Formel-4 oder eine Formel der Form $\forall s\ X : F$, $\exists s\ X : F$ mit einer Formel-5 F.

  D.h. die Quantoren binden am schwächsten.

- Eine Formel ist eine Formel der Stufe 5 (Formel-5).

Formeln (5)

Abkürzungen für Quantoren:

- Wenn es nur eine mögliche Sorte für eine quantifizierte Variable gibt, kann man sie weglassen, d.h. $\forall X : F$ statt $\forall s\, X : F$ schreiben (entsprechend für $\exists$).

  Oft ist der Typ der Variablen durch ihre Verwendung eindeutig festgelegt.

- Wenn ein Quantor direkt auf einen anderen Quantor folgt, kann man den Doppelpunkt weglassen, z.B. $\forall X\, \exists Y : F$.

- Statt einer Sequenz von Quantoren gleichen Typs, z.B. $\forall X_1 \ldots \forall X_n : F$, schreibt man $\forall X_1, \ldots, X_n : F$.

Abkürzung für Ungleichheit:

- $t_1 \neq t_2$ kann als Abkürzung für $\neg(t_1 = t_2)$ verwendet werden.

Formeln in SQL (1)

- Es gibt in SQL nur drei logische Junktoren:
  - `AND` wird für $\wedge$ geschrieben (Konjunktion).
  - `OR` wird für $\vee$ geschrieben (Disjunktion).
  - `NOT` wird für $\neg$ geschrieben (Negation).
- Statt $\exists\, s\, X : F$ schreibt man

```
EXISTS(SELECT *
       FROM   s X
       WHERE  F)
```

  Man kann also testen, ob das Ergebnis der Unteranfrage nicht leer ist.

  Wie bei der logischen Formel kann man in der Bedingung F auch die Variablen der äußeren Anfrage verwenden, weil es eine Formel bezüglich $(\Sigma, \nu\langle X/s\rangle)$ ist.

Formeln in SQL (2)

Beispiel:

- Gesucht sind Studenten (Vorname und Nachname), die Hausaufgabe 1 nicht abgegeben haben.

  D.h. es existiert kein Eintrag in der Bewertungstabelle mit der Nummer (SID) dieses Studenten für Hausaufgabe 1 (d.h. 'H' als Aufgabentyp und 1 als Nr.).

- Logik: $\neg\exists\,\mathtt{BEWERTUNGEN\ B}:\ \mathtt{S.SID} = \mathtt{B.SID} \wedge \mathtt{B.ATYP} = \mathtt{'H'} \wedge \mathtt{B.ANr} = 1$

- SQL:

```
SELECT S.VORNAME, S.NACHNAME
FROM   STUDENTEN S
WHERE  NOT EXISTS (SELECT *
                   FROM   BEWERTUNGEN B
                   WHERE  S.SID  = B.SID
                   AND    B.ATYP = 'H'
                   AND    B.ANR  = 1)
```

Geschlossenene Formeln

Definition:

- Sei eine Signatur Σ gegeben.
- Eine geschlossene Formel (für Σ) ist eine (Σ, ν)-Formel für die leere Variablendeklaration ν.

  D.h. die Variablendeklaration, die überall undefiniert ist.

Bemerkung:

- Um festzulegen, ob eine Formel wahr oder falsch ist, braucht man außer einer Interpretation auch Werte für die Variablen, die nicht durch Quantoren gebunden sind (solche Variablen nennt man „freie Variablen“, s.u.).
- Bei geschlossenen Formeln reicht die Interpretation.

Variablen in einem Term

Definition:

- Die Funktion *vars* berechnet die Menge der Variablen, die in einem gegebenem Term t auftreten.
  - Wenn t eine Konstante c ist:

$$vars(t) := \emptyset.$$

  - Wenn t eine Variable V ist:

$$vars(t) := \{V\}.$$

  - Wenn t die Form $f(t_1, \ldots, t_n)$ hat:

$$vars(t) := \bigcup_{i=1}^{n} vars(t_i).$$

Freie Variablen einer Formel

Definition:

- $free(F)$ ist die Menge der freien Variablen in F:
  - Ist F atomare Formel $p(t_1, \ldots, t_n)$ oder $t_1 = t_2$:

$$free(F) := \bigcup_{i=1}^{n} vars(t_i).$$

  - Ist F logische Konstante $\top$ oder $\bot$: $free(F) := \emptyset$.
  - Wenn F die Form $(\neg G)$ hat: $free(F) := free(G)$.
  - Wenn F die Form $(G_1 \wedge G_2)$, $(G_1 \vee G_2)$, etc. hat: $free(F) := free(G_1) \cup free(G_2)$.
  - Wenn F die Form $(\forall\, s\, X :\, G)$ oder $(\exists\, s\, X :\, G)$ hat: $free(F) := free(G) \setminus \{X\}$.

Anfragen

- Geschlossene Formeln werden in Datenbanken als Integritätsbedingungen verwendet.
- Anfragen sind dagegen typischerweise keine geschlossenen Formeln: Man möchte Werte für die freien Variablen („Antwortvariablen“) berechnen, die die Formel wahr machen.
- Beim Bereichskalkül laufen die Variablen über Datenwerte, dort sind Anfragen einfach Formeln mit freien Variablen.
- Beim Tupelkalkül, SQL und einem Kalkül für das ER-Modell laufen die Variablen über Tupel bzw. Objekte, man braucht dann Terme zur Umrechnung in druckbare Datenwerte:

$$\{t_1, \ldots, t_k \; [s_1 \; X_1, \ldots, s_n \; X_n] \mid F\}.$$

Dies entspricht genau `SELECT` t_1 `,` $\ldots$ `,` t_k `FROM` s_1 X_1 `,` s_n X_n `WHERE` F.

$t_1, \ldots, t_k$ sind Terme und F eine Formel bzgl. (Σ, ν) mit $\nu := \{X_1/s_1, \ldots, X_n/s_n\}$.

Wahrheit einer Formel (1)

Definition:

- Der Wahrheitswert $\langle \mathcal{I}, \mathcal{A} \rangle[F] \in \{\mathbf{f}, \mathbf{w}\}$ einer Formel F in $\langle \mathcal{I}, \mathcal{A} \rangle$ ist definiert als (**f** bedeutet falsch, **w** wahr):
  - Für atomare Formeln $p(t_1, \ldots, t_n)$ und $\top$, $\bot$, sowie Formeln der Gestalt $(\neg G)$, $(G_1 \wedge G_2)$, $(G_1 \vee G_2)$, $(G_1 \leftarrow G_2)$, $(G_1 \rightarrow G_2)$, $(G_1 \leftrightarrow G_2)$: Siehe Kapitel 8.
  - Hat F die Form $(\forall\, s\, X :\, G)$:

$$\langle \mathcal{I}, \mathcal{A} \rangle[F] := \begin{cases} \mathbf{w} & \text{falls } \langle \mathcal{I}, \mathcal{A}\langle X/d \rangle \rangle[G] = \mathbf{w} \\ & \quad \text{für alle } d \in \mathcal{I}[s] \\ \mathbf{f} & \text{sonst.} \end{cases}$$

  - (Fortsetzung auf der nächsten Folie . . .)

Wahrheit einer Formel (2)

- Wahrheitswert einer Formel, fortgesetzt:
  - Hat F die Form $(\exists\, s\, X :\, G)$:

$$\langle \mathcal{I}, \mathcal{A} \rangle [F] := \begin{cases} \mathbf{w} & \text{falls } \langle \mathcal{I}, \mathcal{A}\langle X/d \rangle \rangle [G] = \mathbf{w} \\ & \quad \text{für mindestens ein } d \in \mathcal{I}[s] \\ \mathbf{f} & \text{sonst.} \end{cases}$$

- Ist $\langle \mathcal{I}, \mathcal{A} \rangle [F] = \mathbf{w}$, so schreibt man auch $\langle \mathcal{I}, \mathcal{A} \rangle \models F$. Dann heißt $\langle \mathcal{I}, \mathcal{A} \rangle$ ein Modell von F.

  Man sagt dann auch $\langle \mathcal{I}, \mathcal{A} \rangle$ erfüllt F bzw. F ist in $\langle \mathcal{I}, \mathcal{A} \rangle$ wahr.

- Sei F eine (Σ, ν)-Formel. Gilt $\langle \mathcal{I}, \mathcal{A} \rangle [F] = \mathbf{w}$ für alle Variablenbelegungen $\mathcal{A}$ (für $\mathcal{I}$ und ν), so schreibt man $\mathcal{I} \models F$ und nennt $\mathcal{I}$ ein Modell von F.

  D.h. freie Variablen werden als implizit $\forall$-quantifiziert behandelt.

Wiederholung (1)

- Die Begriffe aus Kap. 8 übertragen sich auf beliebige Formeln:
  - Konsistenz: Eine Formel F heisst konsistent, wenn sie für mindestens ein $\mathcal{I}$ und $\mathcal{A}$ wahr ist: $\langle \mathcal{I}, \mathcal{A} \rangle \models F$.
  - Inkonsistenz: Eine Formel F heißt inkonsistent gdw. sie für alle $\mathcal{I}$ und $\mathcal{A}$ falsch ist: $\langle \mathcal{I}, \mathcal{A} \rangle \not\models F$.
  - Tautologie: Eine Formel F heißt Tautologie gdw. sie für alle $\mathcal{I}$ und $\mathcal{A}$ wahr ist: $\langle \mathcal{I}, \mathcal{A} \rangle \models F$.
  - Äquivalenz: Zwei Formeln F und G heißen äquivalent gdw. sie für alle $\mathcal{I}$ und $\mathcal{A}$ den gleichen Wahrheitswert haben: $\langle \mathcal{I}, \mathcal{A} \rangle \models F \iff \langle \mathcal{I}, \mathcal{A} \rangle \models G$.
  - Logische Folgerung: Eine Formel G ist eine logische Folgerung aus einer Formelmenge Φ gdw. für alle $\mathcal{I}$ und $\mathcal{A}$, die jede Formel $F \in \Phi$ erfüllen, auch $\langle \mathcal{I}, \mathcal{A} \rangle \models G$ gilt.

Wiederholung (2)

- Die obigen Begriffe sind sehr relevant für Datenbanken:
  - Bei äquivalenten Formeln als `WHERE`-Bedingung ist egal, welche man schreibt.

    Das Anfrageergebnis ist ja gleich (in jedem DB-Zustand).
    Wenn natürlich eine Formel deutlich komplexer ist als die andere, würde man besser die einfachere wählen.
  - Inkonsistente Formeln werden gelegentlich in Anfragen geschrieben, sind aber natürlich falsch.

    Die Anfrage liefert ja immer die leere Menge als Ergebnis, unabhängig vom DB-Zustand.
  - Tautologien als Integritätsbedingung kann man sich auch sparen.

    In der relationalen Normalformtheorie gibt es „triviale funktionale Abhängigkeiten“, die gerade Tautologien sind.

Anmerkung zur Äquivalenz

- Die obige „logische Äquivalenz“ betrachtet beliebige Interpretationen, auch solche, die in Datenbanken nicht vorkommen können. Dies ist teils zu streng.
- Äquivalent (im Datenbank-Sinn) sind Formeln F und G, wenn für alle Interpretationen $\mathcal{I}$, die einem möglichen Datenbank-Zustand entsprechen, also
  - die Standard-Interpretation der Datentypen haben,
  - die Einschränkungen des Datenmodells erfüllen,

    beim relationalen Modell die Tabellen-Sorten also als endliche Relationen interpretieren, wobei die Zugriffsfunktionen für die Attribute die jeweilige Komponente selektieren,
  - die Integritätsbedingungen im Datenbank-Schema erfüllen,

  und alle Variablenbelegungen $\mathcal{A}$: $\langle \mathcal{I}, \mathcal{A} \rangle \models F \iff \langle \mathcal{I}, \mathcal{A} \rangle \models G$.

Äquivalenzen mit Quantoren (1)

- Ersetzung von Quantoren:
  - $\forall\, s\, X\colon\ F \equiv \neg(\exists\, s\, X\colon\ (\neg F))$
  - $\exists\, s\, X\colon\ F \equiv \neg(\forall\, s\, X\colon\ (\neg F))$
- Logische Junktoren über Quantoren bewegen:
  - $\neg(\forall\, s\, X\colon\ F) \equiv \exists\, s\, X\colon\ (\neg F)$
  - $\neg(\exists\, s\, X\colon\ F) \equiv \forall\, s\, X\colon\ (\neg F)$
  - $\forall\, s\, X\colon\ (F \wedge G) \equiv (\forall\, s\, X\colon\ F) \wedge (\forall\, s\, X\colon\ G)$
  - $\exists\, s\, X\colon\ (F \vee G) \equiv (\exists\, s\, X\colon\ F) \vee (\exists\, s\, X\colon\ G)$

Äquivalenzen mit Quantoren (2)

- Quantoren bewegen: Sei $X \notin \mathit{free}(F)$:
  - $\forall s\, X\colon\ (F \vee G) \equiv F \vee (\forall s\, X\colon\ G)$
  - $\exists s\, X\colon\ (F \wedge G) \equiv F \wedge (\exists s\, X\colon\ G)$

  Falls zusätzlich $\mathcal{I}[s]$ nicht leer sein kann:
  - $\forall s\, X\colon\ (F \wedge G) \equiv F \wedge (\forall s\, X\colon\ G)$
  - $\exists s\, X\colon\ (F \vee G) \equiv F \vee (\exists s\, X\colon\ G)$
- Eliminierung überflüssiger Quantoren:
  Ist $X \notin \mathit{free}(F)$ und kann $\mathcal{I}[s]$ nicht leer sein:
  - $\forall s\, X\colon\ F \equiv F$
  - $\exists s\, X\colon\ F \equiv F$

Äquivalenzen mit Quantoren (3)

- Vertauschung von Quantoren: Ist $X \neq Y$:
  - $\forall\, s_1\, X\colon\ (\forall\, s_2\, Y\colon\ F) \equiv \forall\, s_2\, Y\colon\ (\forall\, s_1\, X\colon\ F)$
  - $\exists\, s_1\, X\colon\ (\exists\, s_2\, Y\colon\ F) \equiv \exists\, s_2\, Y\colon\ (\exists\, s_1\, X\colon\ F)$

    Beachte, dass Quantoren verschiedenen Typs ($\forall$ und $\exists$) nicht vertauscht werden können.

- Umbenennung gebundener Variablen:
  Ist $Y \notin \mathit{free}(F)$ und F' entsteht aus F durch Ersetzen jedes freien Vorkommens von X in F durch Y:
  - $\forall\, s\, X\colon\ F \equiv \forall\, s\, Y\colon\ F'$
  - $\exists\, s\, X\colon\ F \equiv \exists\, s\, Y\colon\ F'$

Inhalt

NOT EXISTS (1)

- Man kann in der äußeren Anfrage testen, ob das Ergebnis der Unteranfrage leer ist (`NOT EXISTS`).

  Jetzt sollen die syntaktischen Details der Realisierung des Existenzquantors in SQL mittels Unteranfragen in SQL besprochen werden. Dies sollte auch für Studierende verständlich sein, die mit der Logik Schwierigkeiten hatten.

- In der inneren Anfrage kann man auf Tupelvariablen zugreifen, die in der `FROM`-Klausel der äußeren Anfrage deklariert sind.

  Wie in der Logik auch: Bei einer Teilformel $(\exists\, s\, X : F)$ können in F natürlich auch andere Variablen außer X verwendet werden.

- Daher muss die Unteranfrage einmal für jeden Wert der benutzten Tupelvariablen der äußeren Anfrage ausgewertet werden (zumindest konzeptionell).

  Die Unteranfrage kann also als parametrisiert angesehen werden.

NOT EXISTS (2)

- Studenten ohne eine abgegebene Hausaufgabe:

```
SELECT VORNAME, NACHNAME
FROM   STUDENTEN S
WHERE  NOT EXISTS (SELECT * FROM BEWERTUNGEN B
                   WHERE  B.ATYP = 'H'
                   AND    B.SID = S.SID )
```

- Die Tupelvariable `S` läuft über die vier Zeilen in der Tabelle `STUDENTEN`. Konzeptionell wird die Unteranfrage viermal ausgewertet. Jedes Mal wird `S.SID` durch den `SID`-Wert des aktuellen Tupels `S` ersetzt.

  Natürlich kann das DBMS eine andere, effizientere Auswertungsstrategie wählen, wenn diese garantiert das gleiche Ergebnis liefert.

NOT EXISTS (3)

- Zunächst zeigt S auf das STUDENTEN-Tupel

S →

SID	VORNAME	NACHNAME	EMAIL
101	Lisa	Weiss	...

- S.SID in der Unteranfrage wird konzeptionell durch 101 ersetzt und folgende Anfrage wird ausgeführt:

```
SELECT * FROM BEWERTUNGEN B
WHERE  B.ATYP = 'H'
AND    B.SID = 101
```

SID	ATYP	ANR	PUNKTE
101	H	1	10
101	H	2	8

- Das Ergebnis ist nicht leer.
  Somit ist die NOT EXISTS-Bedingung für dieses Tupel S nicht erfüllt.

NOT EXISTS (4)

- Dann wird S die zweite Zeile in STUDENTEN zugewiesen. Die Unteranfrage wird nun für S.SID = 102 ausgeführt:

```
SELECT * FROM BEWERTUNGEN B
WHERE  B.ATYP = 'H'
AND    B.SID = 102
```

SID	ATYP	ANR	PUNKTE
102	H	1	9
102	H	2	9

- Das Ergebnis ist nicht leer, also ist die NOT EXISTS-Bedingung wieder nicht erfüllt.
- Auch für die dritte Zeile in STUDENTEN ist die Bedingung nicht erfüllt.

NOT EXISTS (5)

- Schließlich zeigt S auf das STUDENTEN-Tupel

S →

SID	VORNAME	NACHNAME	EMAIL
104	Iris	Winter	...

- Für S.SID = 104 ist das Unteranfragenergebnis leer:

```
SELECT * FROM BEWERTUNGEN B
WHERE  B.ATYP = 'H'
AND    B.SID = 104
```

no rows selected

- Somit ist die NOT EXISTS-Bedingung der Hauptanfrage für dieses Tupel S erfüllt.
  Iris Winter wird als Anfrageergebnis ausgegeben.

NOT EXISTS (6)

- Während man Variablen der äußeren Anfrage in der inneren verwenden kann, gilt das umgekehrt nicht:

```
SELECT VORNAME, NACHNAME, B.ANR
FROM   STUDENTEN S
WHERE  NOT EXISTS (SELECT * FROM BEWERTUNGEN B
                   WHERE  B.ATYP = 'H'
                   AND    B.SID = S.SID)
```

Falsch!

- Dies entspricht einer Blockstruktur (global/lokal):
  - In der äußeren Anfrage deklarierte Tupelvariablen gelten für die gesamte Anfrage.
  - Variablen der Unteranfrage gelten nur dort.

Korrelierte/Unkorrelierte Unteranfragen (1)

- Unteranfragen, die Variablen der äußeren Anfrage verwenden, nennt man „korrelierte Unteranfragen“.

  Korrelierte Unteranfragen kann man sich als parametrisiert mit Tupeln der äußeren Anfrage vorstellen. Man kann dies optimieren, aber konzeptionell werden diese Unteranfragen einmal für jede Belegung der Tupelvariablen der äußeren Anfrage ausgeführt.

- Unteranfragen, die nicht auf Variablen der äußeren Anfrage zugreifen, nennt man „unkorrelierte Unteranfragen“.

  Es genügt, eine unkorrelierte Unteranfrage nur einmal auszuführen (da das Ergebnis nicht von Tupelvariablen der äußeren Anfrage abhängt).

Korrelierte/Unkorrelierte Unteranfragen (2)

- Unkorrelierte `EXISTS`-Unteranfragen sind fast immer falsch:

Falsch!

```
SELECT VORNAME, NACHNAME
FROM   STUDENTEN S
WHERE  NOT EXISTS (SELECT * FROM BEWERTUNGEN B
                   WHERE  B.ATYP = 'H')
```

Hier wurde die Verbundbedingung in der Unteranfrage vergessen.

Die Unteranfrage wurde somit zu einer unkorrelierten Unteranfrage.

- Wenn es mindestens einen Hausaufgaben-Eintrag in `BEWERTUNGEN` gibt, egal für welchen Studenten, ist das `NOT EXISTS` falsch und das Anfrageergebnis leer.
- Für andere Typen von Unteranfragen (z.B. mit `IN`, s.u.) sind unkorrelierte Unteranfragen in Ordnung.

Syntaktische Details: Attribut-Zugriff (1)

- Bisher musste es bei einer Attributreferenz ohne Tupelvariable nur eine passende Variable geben.
- Bei Unteranfragen fordert SQL nur, dass es eine eindeutige nächste Tupelvariable mit dem Attribut gibt, z.B. ist folgendes legal (aber schlechter Stil):

```
SELECT VORNAME, NACHNAME
FROM   STUDENTEN S
WHERE  NOT EXISTS (SELECT * FROM BEWERTUNGEN B
                   WHERE  ATYP = 'H'
                   AND    SID = S.SID)
```

Syntaktische Details: Attribut-Zugriff (2)

- Bei Attributreferenzen ohne Tupelvariable sucht der SQL-Parser die `FROM`-Klauseln beginnend mit der aktuellen Unteranfrage, hin zur äußersten Anfrage ab.

  Die verschachtelten Anfragen werden von innen nach außen betrachtet.

- Die erste `FROM`-Klausel, die eine Tupelvariable mit dem Attribut enthält, darf nur eine solche Variable haben. Das Attribut referenziert dann diese Variable.

- Durch diese Regel können unkorrelierte Unteranfragen unabhängig entwickelt und ohne Veränderungen in andere Anfragen eingefügt werden.

Syntaktische Details: Verschattung

- Es ist auch zulässig, in der Unteranfrage Tupelvariablen zu deklarieren, die den gleichen Namen wie Variablen der äußeren Anfrage haben.

```
SELECT VORNAME, NACHNAME
FROM   STUDENTEN X
WHERE  NOT EXISTS (SELECT * FROM BEWERTUNGEN X
                   WHERE ???)
```

- Alle Referenzen auf `X` in der Unteranfrage meinen `BEWERTUNGEN X`. Die Variable der äußeren Anfrage wird verschattet. Sie kann in der Unteranfrage nicht verwendet werden.

SELECT-Liste der Unteranfrage

- Es ist zulässig, in der Unteranfrage eine SELECT-Liste zu spezifizieren, aber da die zurückgegebenen Spalten für NOT EXISTS nicht interessieren, sollte „SELECT *“ in der Unteranfrage verwendet werden.
- Man hört gelegentlich, dass in einigen Systemen SELECT null oder SELECT 1 schneller als SELECT * sei.

  Oracle's Programmierer verwenden „SELECT null“ (in „catalog.sql“). Dies funktioniert aber in DB2 nicht (Null kann dort nicht als Term verwendet werden). Heutzutage sollten gute Optimierer wissen, dass die Spaltenwerte nicht wirklich benötigt werden, und die SELECT-Liste keine Rolle spielen sollte, auch nicht für die Performance.

Syntax (1)

Bedingung (Form 5: EXISTS):

- Die Syntax braucht hier das `NOT` von `NOT EXISTS` nicht explizit zu behandeln, da jede Formel durch Voranstellen von `NOT` negiert werden kann. Bei `LIKE`, `IN`, etc. stand das `NOT` dagegen nicht vor der atomaren Formel, sondern an einer anderen Stelle. Daher mussten dort die Syntaxregeln das `NOT` explizit erlauben.
- MySQL unterstützt Unteranfragen erst ab Version 4.1.

Syntax (2)

Unteranfrage:

- Eine Unteranfrage ist also ein Ausdruck der Form `SELECT ... FROM ... WHERE ...`.

  Später auch mit `GROUP BY` und `HAVING`. SQL-92 erlaubt auch `UNION` (s.u.) in Unteranfragen (ebenso PostgreSQL, Oracle, DB2, SQL Server, MariaDB 5.5). SQL-86 erlaubt dies nicht (und Access unterstützt es nicht).

- `ORDER BY` ist in Unteranfragen nicht erlaubt.

  Das macht hier keinen Sinn, sondern ist nur für die Ausgabe wichtig.
  (Dieser Punkt hat sich inzwischen geändert, siehe Fortsetzung im Sommer.)

- Unteranfragen müssen immer in Klammern `(...)` eingeschlossen werden.

EXISTS ohne NOT

- Man kann `EXISTS` auch ohne `NOT` benutzen (semijoin).
- Wer hat mindestens eine Hausaufgabe gelöst?

```
SELECT S.SID, S.VORNAME, S.NACHNAME
FROM   STUDENTEN S
WHERE  EXISTS (SELECT * FROM BEWERTUNGEN B
               WHERE  B.SID = S.SID
               AND    B.ATYP = 'H')
```

- Äquivalente Anfrage mit normalem Verbund:

```
SELECT DISTINCT S.SID, S.VORNAME, S.NACHNAME
FROM   STUDENTEN S, BEWERTUNGEN B
WHERE  S.SID = B.SID AND B.ATYP = 'H'
```

Allaussagen (1)

- Bei welchen Aufgaben haben alle Abgaben mindestens 80% der vollen Punktzahl?
- Da SQL keinen Allquantor hat, muss man die Anfrage mit `NOT EXISTS` formulieren:

```
SELECT A.ATYP, A.ANR
FROM   AUFGABEN A
WHERE  NOT EXISTS
       (SELECT * FROM BEWERTUNGEN B
        WHERE  B.ATYP = A.ATYP AND B.ANR = A.ANR
        AND    B.PUNKTE < A.MAXPT * 0.8)
```

Allaussagen (2)

- Man kann die natürlichsprachliche Formulierung der Anfrage ganz direkt in den Tupelkalkül übersetzen:

$$\{\texttt{A.ATYP},\ \texttt{A.ANR}\,[\texttt{AUFGABEN A}] \mid \forall\,\texttt{BEWERTUNGEN B}:\ \texttt{B.ATYP} = \texttt{A.ATYP} \wedge \texttt{B.ANR} = \texttt{A.ANR} \rightarrow\ \texttt{B.PUNKTE} \geq \texttt{A.MAXPT} * (80/100)\}$$

- Das Muster $\forall s\, X : (F_1 \rightarrow F_2)$ ist sehr typisch: F_2 muss wahr sein für alle X, die F_1 erfüllen.
- SQL hat aber nur den Existenzquantor („`EXISTS`“), und keinen Allquantor.

  Es gibt allerdings Spezialkonstrukte wie „`>= ALL`“, siehe unten.

Allaussagen (3)

- Man nützt in SQL aus, dass $\forall s\, X : F$ äquivalent ist zu $\neg\exists s\, X : \neg F$. Ein Quantor genügt also.

  „F ist wahr für alle X“ ist das gleiche wie „F ist falsch für kein X“.

- Das Muster $\forall s\, X : (F_1 \to F_2)$ ist äquivalent zu $\neg\exists s\, X : F_1 \wedge \neg F_2$.

- Im Beispiel ergibt sich:

$$\{\texttt{A.ATYP},\ \texttt{A.ANR}\,[\texttt{AUFGABEN A}] \mid \neg\exists\,\texttt{BEWERTUNGEN B}:\ \texttt{B.ATYP} = \texttt{A.ATYP} \wedge \texttt{B.ANR} = \texttt{A.ANR} \wedge \texttt{B.PUNKTE} < \texttt{A.MAXPT} * (80/100)\}$$

- Dies kann man direkt in SQL ausdrücken (s.o.).

Allaussagen (4)

- Wer hat die meisten Punkte für Hausaufgabe 1?

```
SELECT S.VORNAME, S.NACHNAME, B.PUNKTE
FROM   STUDENTEN S, BEWERTUNGEN B
WHERE  S.SID = B.SID
AND    B.ATYP = 'H' AND B.ANR = 1
AND    NOT EXISTS
       (SELECT * FROM BEWERTUNGEN C
        WHERE  C.ATYP = 'H' AND C.ANR = 1
        AND    C.PUNKTE > B.PUNKTE)
```

- Gesucht ist also eine Bewertung `B` für HA 1, zu der es keine Bewertung `C` mit mehr Punkten als `B` gibt.

Verschachtelte Unteranfragen

- Unteranfragen kann man beliebig verschachteln.
- Welche Studenten haben alle Hausaufgaben gelöst?

```
SELECT VORNAME, NACHNAME
FROM   STUDENTEN S
WHERE  NOT EXISTS
       (SELECT * FROM AUFGABEN A
        WHERE  ATYP = 'H'
        AND    NOT EXISTS
               (SELECT * FROM BEWERTUNGEN B
                WHERE  B.SID = S.SID
                AND    B.ANR = A.ANR
                AND    B.ATYP = 'H'))
```

Häufige Fehler (1)

- Wie oben erwähnt, ist die Verwendung einer unkorrelierten Unteranfrage mit `NOT EXISTS` meist falsch.
- Trifft dies auch in diesem Fall zu?
  (Es gibt eine Verbundbedingung in der Unteranfrage.)

```
SELECT VORNAME, NACHNAME
FROM   STUDENTEN S
WHERE  NOT EXISTS
       (SELECT *
        FROM   BEWERTUNGEN B, STUDENTEN S
        WHERE  B.SID = S.SID
        AND    B.ATYP = 'H' AND B.ANR = 1)
```

Häufige Fehler (2)

- Gibt es irgendein Problem mit dieser Anfrage?
  Es sollen alle Studenten ausgegeben werden, die noch nicht aktiv an der Vorlesung teilgenommen haben, d.h. weder eine Hausaufgabe gelöst, noch eine Prüfung absolviert haben.

```
SELECT S.VORNAME, S.NACHNAME
FROM   STUDENTEN S, BEWERTUNGEN B
WHERE  S.SID = B.SID
AND    NOT EXISTS (SELECT *
                   FROM   BEWERTUNGEN B
                   WHERE  S.SID = B.SID)
```

Inhalt

NOT IN (1)

- Mit `IN` ($\in$) und `NOT IN` ($\notin$) kann man testen, ob ein Attributwert in einer Menge vorkommt, die von einer weiteren SQL-Anfrage berechnet wird.
- Z.B. Studenten ohne ein Hausaufgabenergebnis:

```
SELECT VORNAME, NACHNAME
FROM   STUDENTEN
WHERE  SID NOT IN (SELECT SID
                   FROM   BEWERTUNGEN
                   WHERE  ATYP = 'H')
```

VORNAME	NACHNAME
Iris	Winter

NOT IN (2)

- Konzeptionell wird die Unteranfrage vor Beginn der Ausführung der Hauptanfrage ausgewertet:

STUDENTEN			
SID	VORNAME	NACHNAME	EMAIL
101	Lisa	Weiss	...
102	Michael	Grau	(null)
103	Daniel	Sommer	...
104	Iris	Winter	...

Unteranfrage
SID
101
101
102
102
103

- Dann wird für jedes STUDENTEN-Tupel eine passende SID im Ergebnis der Unteranfrage gesucht.
  Gibt es keine, so wird der Studentenname ausgegeben.

NOT IN (3)

- Der wesentliche Unterschied von `IN`-Bedingungen im Vergleich zu `EXISTS`-Unteranfragen ist, dass der Vergleich der Werte (im Beispiel `SID`)
  - hier durch das `IN`-Konstrukt selbst erledigt wird,
  - während man bei `EXISTS`-Unteranfragen die Join-Bedingung explizit in die `WHERE`-Klausel der Unteranfrage schreiben muss.

    Bei `NOT EXISTS` ist es eigentlich ein Antijoin, siehe Kapitel 14.

- Damit sind unkorrelierte `IN`-Unteranfragen in Ordnung, während unkorrelierte `EXISTS`-Unteranfragen fast immer ein Fehler sind.

  „Unkorreliert“ heißt, dass die Join-Bedingung fehlt. Korrelierte `IN`-Unteranfragen sind möglich, aber eher schlechter Stil (weil man es nicht erwartet).

NOT IN (4)

- Man kann `DISTINCT` in Unteranfragen verwenden:

```
SELECT VORNAME, NACHNAME
FROM   STUDENTEN
WHERE  SID NOT IN (SELECT DISTINCT SID   ?
                   FROM   BEWERTUNGEN
                   WHERE  ATYP = 'H')
```

- Dies ist äquivalent. Der Einfluss auf die Performance hängt von den Daten und dem DBMS ab.

  Ich würde erwarten, dass Optimierer wissen, dass Duplikate in diesem Fall nicht wichtig sind. Die Verwendung von `DISTINCT` könnte aber den Effekt haben, dass der Optimierer Auswertungsstrategien, die das Ergebnis der Unteranfrage nicht materialisieren, nicht berücksichtigt.

NOT IN (5)

- Man kann auch `IN` (ohne `NOT`) für einen Elementtest verwenden.
- Das wird relativ selten getan, da es äquivalent zu einem Verbund ist, der in der Unteranfrage formuliert wird.
- Manchmal ist diese Formulierung jedoch eleganter.
  Es kann auch helfen, Duplikate zu vermeiden.

  Oder auch um die exakt benötigten Duplikate zu erhalten
  (vgl. Beispiel auf nächster Folie).

NOT IN (6)

- Z.B. Themen der Hausaufgaben, die von mindestens einem Studenten gelöst wurden:

```
SELECT THEMA
FROM   AUFGABEN
WHERE  ATYP='H' AND ANR IN (SELECT ANR
                            FROM   BEWERTUNGEN
                            WHERE  ATYP='H')
```

- Aufgabe: Gibt es einen Unterschied zu dieser Anfrage (mit oder ohne `DISTINCT`)?

```
SELECT DISTINCT THEMA
FROM   AUFGABEN A, BEWERTUNGEN B
WHERE  A.ATYP='H' AND A.ANR=B.ANR AND B.ATYP='H'
```

NOT IN (7)

- In SQL-86 musste die Unteranfrage rechts von `IN` eine einzelne Ausgabespalte haben.

  So dass das Ergebnis der Unteranfrage wirklich eine Menge (oder Multimenge) ist, und nicht eine beliebige Relation.

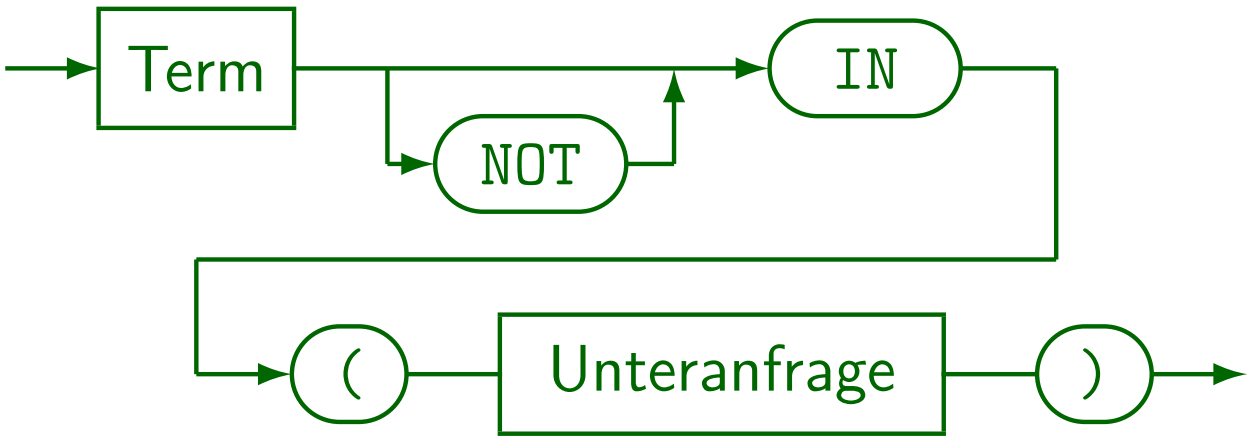
- In SQL-92 wurden Vergleiche auf Tupel-Ebene eingeführt, so dass man z.B. auch schreiben kann

```
WHERE (VORNAME, NACHNAME) NOT IN
      (SELECT VORNAME, NACHNAME
       FROM   ...)
```

  Das ist aber nicht portabel. SQL Server und Access unterstützen es nicht.
  MySQL ab Version 4.1 kann es, davor hatte es keine Unteranfragen.
  Eine `EXISTS` Unteranfrage (s.u.) wäre in diesem Fall besser (Stilfrage).

NOT IN (8)

Bedingung (Form 6: IN mit Unteranfrage):



- Die Unteranfrage muss eine Tabelle mit einer einzelnen Spalte liefern.
- In SQL-92, Oracle und DB2 ist es möglich, auf die linke Seite Tupel der Form (`Term`$_1$, ..., `Term`$_n$) zu schreiben. Dann muss die Unteranfrage eine Tabelle mit genau n Spalten ergeben.
- MySQL unterstützt Unteranfragen erst ab Version 4.1.
- Die Spaltennamen links und rechts von `IN` müssen nicht übereinstimmen, aber die Datentypen müssen kompatibel sein.

IN vs. EXISTS (1)

- `IN`-Bedingungen sind praktisch, aber nicht wirklich nötig: Man kann jede `IN`-Bedingung in eine äquivalente `EXISTS`-Bedingung übersetzen.
- Die Bedingung

$$t_1 \texttt{ IN (SELECT } t_2 \texttt{ FROM } R_1\, X_1, \ \ldots, \ R_n\, X_n \texttt{ WHERE } \varphi\texttt{)}$$

  ist (unter gewissen Voraussetzungen) äquivalent zu

$$\texttt{EXISTS (SELECT * FROM } R_1\, X_1, \ \ldots, \ R_n X_n \texttt{ WHERE (}\varphi\texttt{) AND } t_1 = t_2\texttt{)}$$

IN vs. EXISTS (2)

- Voraussetzung ist, dass die Bedeutung von t_1 nicht verändert wird, wenn es in die Unteranfrage verschoben wird (läßt sich immer erreichen):
  - Alle Tupelvariablen, die in t_1 vorkommen, müssen verschieden von $X_1, \ldots, X_n$ sein.

    Ggf. kann man die X_i umbenennen: Die Namen der Tupelvariablen in der Unteranfrage sind ja nur lokal wichtig.
  - Enthält t_1 Attributreferenzen A ohne Tupelvariable, so dürfen die R_i kein Attribut A haben.

    Das ist kein Problem: Notfalls fügt man die Tupelvariable ein.
- Außerdem gilt die Äquivalenz nur, wenn die Unteranfrage für t_2 keine Nullwerte liefert (siehe Kapitel 12).

Beispiel für interessanten Fehler

- Aufgabe: Betrachten Sie die folgende merkwürdige Anfrage. Sie sollte Studenten finden, die weder eine Hausaufgabe gelöst, noch an einer Prüfung teilgenommen haben.

```
SELECT VORNAME, NACHNAME          Falsch!
FROM   STUDENTEN S
WHERE  SID NOT IN (SELECT SID
                   FROM   AUFGABEN)
```

  Die Tabelle `AUFGABEN` hat kein Attribut `SID`. Wahrscheinlich war die Tabelle `BEWERTUNGEN` gemeint.

- Diese Anfrage ist syntaktisch korrekt. Warum?
- Was ist die Ausgabe dieser Anfrage?

  Unter der Annahme, dass `AUFGABEN` nicht leer ist.

Inhalt

ALL, ANY, SOME (1)

- Man kann einen Wert mit allen Werten einer Menge (berechnet durch eine Unteranfrage) vergleichen.
- Man kann fordern, dass der Vergleich für alle Elemente (`ALL`) oder mindestens eines (`ANY`) wahr ist:

```
SELECT S.VORNAME, S.NACHNAME, B.PUNKTE
FROM   STUDENTEN S, BEWERTUNGEN B
WHERE  S.SID=B.SID AND B.ATYP='H' AND B.ANR=1
AND    B.PUNKTE >= ALL (SELECT H1.PUNKTE
                        FROM   BEWERTUNGEN H1
                        WHERE  H1.ATYP = 'H'
                        AND    H1.ANR = 1)
```

ALL, ANY, SOME (2)

- Folgendes ist logisch äquivalent zu obiger Anfrage:

```
SELECT S.VORNAME, S.NACHNAME, B.PUNKTE
FROM   STUDENTEN S, BEWERTUNGEN B
WHERE  S.SID=B.SID AND B.ATYP='H' AND B.ANR=1
AND    NOT B.PUNKTE < ANY (SELECT H1.PUNKTE
                           FROM   BEWERTUNGEN H1
                           WHERE  H1.ATYP = 'H'
                           AND    H1.ANR = 1)
```

- Hier wurde nur die bekannte Äquivalenz von „für alle X“ ($\forall$) und „es gibt kein X, so dass nicht“ ($\neg\exists\neg$) ausgenutzt.

ALL, ANY, SOME (3)

- Dieses Konstrukt ist nicht zwingend erforderlich, da

  t_1 `<` `ANY (SELECT` t_2 `FROM ... WHERE ...)`

  äquivalent ist zu

  `EXISTS (SELECT * FROM ... WHERE ...` `AND` t_1 `<` t_2`)`

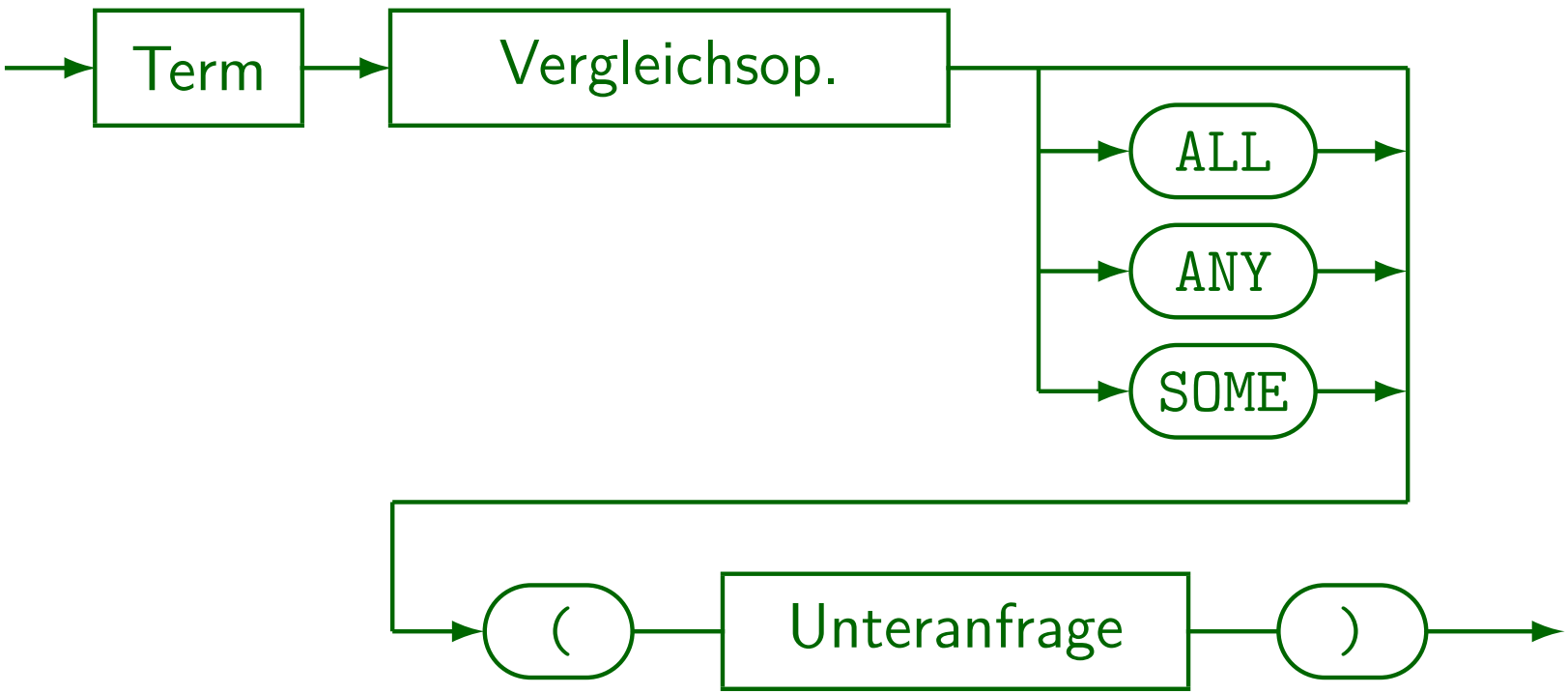
  Es gelten die gleichen Einschränkungen wie oben für `IN` erklärt, auch das Problem mit dem Nullwert für t_2.

- Z.B. macht Oracle intern solche Transformationen, so dass der Anfrageoptimierer nicht so viele Fälle behandeln muss (syntaktische Varianten).

  Dabei wird das Problem mit z.B. `IN` bei einer Unteranfrage, die einen Nullwert liefert, richtig behandelt. Mir ist unklar, wie das funktioniert.

ALL, ANY, SOME (4)

Bedingung (Form 7: ALL/ANY):



ALL, ANY, SOME (5)

Syntaktische Bemerkungen:

- `ANY` und `SOME` sind Synonyme.
- „`x IN S`“ ist äquivalent zu „`x = ANY S`“.
- Die Unteranfrage darf nur eine Spalte ausgeben.

  SQL-92 erlaubt auch Vergleiche auf Tupelbasis. Oracle unterstützt dies nur mit <> und =, DB2 unterstützt nur =`ANY` (äquivalent zu `IN`). SQL86, SQL Server, und Access unterstützten keine Tupelvergleiche.

- Ist kein Schlüsselwort `ALL`/`ANY`/`SOME` angegeben, darf die Unteranfrage max. eine Ergebniszeile liefern.

  Da es auch nur eine Spalte gibt, bedeutet dies, dass die Unteranfrage einen einzelnen Datenwert zurückgibt. Ist das Ergebnis der Unteranfrage leer, so wird der Nullwert verwendet.

Ein-Wert-Unteranfragen (1)

- Wer hat volle Punkte für Hausaufgabe 1?

```
SELECT S.VORNAME, S.NACHNAME
FROM   STUDENTEN S, BEWERTUNGEN B
WHERE  S.SID=B.SID AND B.ATYP='H' AND B.ANR=1
AND    B.PUNKTE = (SELECT MAXPT
                   FROM   AUFGABEN
                   WHERE  ATYP='H' AND ANR=1)
```

- Es ist nur möglich `ANY`/`ALL` wegzulassen, wenn die Unteranfrage garantiert höchstens eine Zeile liefert.

  In diesem Beispiel ist das durch den Schlüssel von `AUFGABEN` garantiert.
  Im allgemeinen kann es aber von den Daten abhängen (Laufzeitfehler).
  Die Anfrage könnte bei Tests gut laufen, aber später Fehler geben.
  Verwenden Sie Integritätsbedingungen zur Sicherung der notwendigen Annahmen!

Ein-Wert-Unteranfragen (2)

- In SQL-92, Oracle, PostgreSQL, DB2, SQL Server und Access kann eine Unteranfrage, die einen einzelnen Datenwert liefert, wie ein Term/Ausdruck verwendet werden.
- Daher kann die Unteranfrage auch links stehen:

```
(SELECT MAXPT FROM ...) = B.PUNKTE
```

  In Oracle8 und SQL-86 musste die Unteranfrage auf der rechten Seite stehen.

  Oracle8 war aktuell von 1997 bis 1999.

- Das Ergebnis einer Unteranfrage kann Eingabe für Berechnungen sein, z.B. (nicht in SQL-86, Oracle8):

```
B.PUNKTE >= (SELECT MAXPT FROM ...) * 0.9
```

- Wenn die Unteranfrage ein leeres Ergebnis hat, wird stattdessen der Nullwert verwendet (siehe Kap. 12).