Einführung in Datenbanken

Kapitel 7: Logik-basiertes Kern-SQL: Praxis, Teil II

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Wintersemester 2018/19

http://www.informatik.uni-halle.de/~brass/db18/

Inhalt

1. Monotonie
2. EXISTS-Unteranfragen
3. IN-Unteranfragen
4. ALL, ANY
5. Sichten
6. ORDER BY

Beispiel-Datenbank

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

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

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

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: Seien $\mathcal{I}_1$ und $\mathcal{I}_2$ Interpretationen der Logik für die gleiche Signatur $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F})$, wobei $\mathcal{I}_1 \subseteq \mathcal{I}_2$, d.h.
  - $\mathcal{I}_1[s] \subseteq \mathcal{I}_2[s]$ für alle Sorten s,
  - $\mathcal{I}_1[p, s_1, \ldots, s_n] = \mathcal{I}_1[p, s_1, \ldots, s_n] \cap (\mathcal{I}_1[s_1] \times \cdots \times \mathcal{I}_1[s_n])$ für alle $p \in \mathcal{P}_{s_1,\ldots,s_n}$
  - $\mathcal{I}_1[f, s_1, \ldots, s_n](d_1, \ldots, d_n) = \mathcal{I}_2[f, s_1, \ldots, s_n](d_1, \ldots, d_n)$ für alle $f \in \mathcal{F}_{s_1,\ldots,s_n}$ und $d_i \in \mathcal{I}_1[s_i]$, $i = 1, \ldots, n$.

  Sei Q eine Formel der Logik (z.B. Tupelkalkül), die aus einem Präfix von Existenzquantoren besteht, gefolgt von einer quantorenfreien Formel F, also von der Form

  $$\exists\, s_1\, X_1 \ldots \exists\, s_n\, X_n\!: \; F.$$

  Dann gilt für alle Variablenbelegungen $\mathcal{A}$:
  Wenn $\langle \mathcal{I}_1, \mathcal{A} \rangle \models Q$, dann $\langle \mathcal{I}_2, \mathcal{A} \rangle \models Q$.

Monotone und nichtmonotone Anfragen (3)

- Satz/Korollar: SQL-Anfragen Q der Form

```
SELECT t1, . . . , tk
FROM   R1 X1, . . . , Rn Xn
WHERE  F
```

  wobei die Bedingung F keine Unteranfragen enthält, 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 (4)

- 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 11).

- 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 (5)

- 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 unten.

Monotone und nichtmonotone Anfragen (6)

- Bei der Formulierung einer Anfrage in SQL ist es wichtig festzustellen, ob die Anfrage benötigt, 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)

- Es ist wichtig zu verstehen, dass es einen Unterschied gibt zwischen der Nicht-Existenz einer Zeile und der Existenz einer Zeile mit einem anderen Wert.

  Verhält sich die benötigte Anfrage nichtmonoton (d.h. Einfügung einer Zeile kann eine Antwort ungültig machen), dann wird `NOT EXISTS`, `NOT IN`, `<> ALL` etc. benötigt.

- Es gibt keine Möglichkeit, dies ohne eine Unteranfrage zu schreiben.

  Außer eventuell bei Verwendung eines äußeren Verbunds. Aggregationen verändern sich auch, wenn Tupel eingefügt werden, aber ohne Unteranfrage können sie nicht “for all” oder “not exists” ausdrücken.

Beispiele für Fehler (3)

- 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

1. Monotonie
2. EXISTS-Unteranfragen
3. IN-Unteranfragen
4. ALL, ANY
5. Sichten
6. ORDER BY

NOT EXISTS (1)

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

  Der Existenzquantor wurde schon im Logik-Kapitel 5 eingeführt, jetzt sollen die syntaktischen Details der Realisierung 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 $(\exists\, s\ X\colon F)$ 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

	SID	VORNAME	NACHNAME	EMAIL
S →	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

	SID	VORNAME	NACHNAME	EMAIL
S →	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 Amfrage abhängt).

Korrelierte/Unkorrelierte Unteranfragen (2)

- Unkorrelierte EXISTS-Unteranfragen sind fast immer falsch:

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

Falsch!

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 ...`.

  SQL-92 erlaubt auch `UNION` (siehe unten) in Unteranfragen (ebenso Oracle, DB2, und SQL Server), 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.

- 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 $\forall s\ X: (F_1 \rightarrow F_2)$ ist äquivalent zu $\neg\exists s\ X: F_1 \wedge \neg F_2$.

- Im Beispiel ergibt sich:

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

- 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

1 Monotonie
2 EXISTS-Unteranfragen
3 IN-Unteranfragen
4 ALL, ANY
5 Sichten
6 ORDER BY

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 9.

- 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.

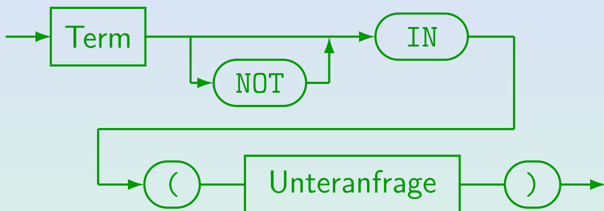
- In SQL-92 wurden Vergleiche auf Tupel-Ebene eingeführt, so dass man z.B. auch schreiben kann

```
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

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

ist (unter gewissen Voraussetzungen) äquivalent zu

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

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 8).

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
FROM    STUDENTEN S
WHERE   SID NOT IN (SELECT SID
                    FROM   AUFGABEN)
```

Falsch!

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

1. Monotonie
2. EXISTS-Unteranfragen
3. IN-Unteranfragen
4. ALL, ANY
5. Sichten
6. ORDER BY

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`)`

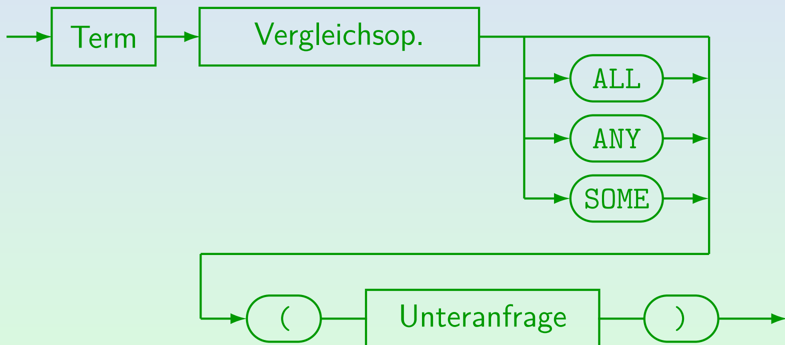
  Es gelten die gleichen Einschränkungen wie oben für `IN` erklärt, auch das Problem mit dem Nullwert für t_2.

- 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.

  SQL92 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 wird der Schlüssel von AUFGABEN spezifiziert. Im allgemeinen, kann das aber von den Daten abhängen. Die Anfrage könnte bei Tests gut laufen, aber später Fehler geben. Verwenden Sie Integritätsbedingungen zur Sicherung der notwendigen Annahmen.

Ein-Wert-Unteranfragen (2)

- In SQL92, DB2, Oracle 9i, SQL Server und Access kann eine Unteranfrage, die einen einzelnen Datenwert liefert, wie ein Term/Ausdruck verwendet werden. Somit ist dies zulässig:

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

- In Oracle8 und SQL86 muss die Unteranfrage auf der rechten Seite stehen.
- Das Ergebnis einer Unteranfrage kann Eingabe für Berechnungen sein, z.B. (nicht in SQL86, Oracle8):

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

- Wenn die Unteranfrage ein leeres Ergebnis hat, wird stattdessen der Nullwert verwendet (siehe Kap. 8).

Inhalt

1. Monotonie
2. EXISTS-Unteranfragen
3. IN-Unteranfragen
4. ALL, ANY
5. Sichten
6. ORDER BY

Unteranfragen unter FROM (1)

- Da das Ergebnis einer SQL-Anfrage eine Tabelle ist, bietet sich an, dass man Unteranfragen an Stelle einer Tabelle in der FROM-Klausel schreiben kann.

  Das war in SQL-86 verboten, und SQL wurde damals oft kritisiert, "nicht orthogonale Konstrukte" zu haben, die man nicht beliebig kombinieren kann.

- In folgendem Beispiel wird der Verbund von AUFGABEN und BEWERTUNGEN in einer Unteranfrage berechnet:

```
SELECT X.SID, ROUND(X.PUNKTE*100/X.MAXPT) AS PZT
FROM   (SELECT A.ATYP, A.ANR, B.SID, B.PUNKTE,
               A.MAXPT
        FROM   AUFGABEN A, BEWERTUNGEN B
        WHERE  A.ATYP=B.ATYP AND A.ANR=B.ANR) X
WHERE  X.ATYP = 'H' AND X.ANR = 1
```

Unteranfragen unter FROM (2)

- Im obigen Beispiel verbessert die Unteranfrage leider nicht die Lesbarkeit der gesamten Anfrage.
- Unteranfragen unter `FROM` werden aber für für geschachtelte Aggregationen unbedingt benötigt, siehe Kapitel 11.
- Außerdem sind Unterabfragen unter `FROM` die Grundlage für Sichten (virtuelle Tabellen) und "Common Table Subexpressions" (s.u.):
  - Man möchte große Anfragen stückweise aufbauen, so wie man Prozeduren in in der Programmierung einsetzt, um ein zu langes Programm sinnvoll zu strukturieren.
  - Mit Sichten kann man wiederverwendbare Bausteine für Anfragen definieren.

Unteranfragen unter FROM (3)

Syntaktische Feinheiten:

- SQL92, SQL Server und DB2 fordern die Definition einer Tupelvariable für die Unteranfrage; in Oracle und Access ist das optional.
- SQL92, DB2, SQL Server (nicht Oracle8, Access) lassen folgende Umbenennung von Spalten zu:

```
FROM (...) X(AUFG_TYP, AUFG_NR, ...)
```

- In Oracle und Access können Spalten nur innerhalb der Unteranfrage umbenannt werden.

  Alle Systeme unterstützen die Spezifikation neuer Spaltennamen in der SELECT-Klausel, so dass dies eine portabele Möglichkeit ist.

Unteranfragen unter FROM (4)

Syntaktische Feinheiten, Forts.:

- Innerhalb der Unteranfrage kann man nicht auf andere Tupelvariablen zugreifen, die in der gleichen FROM-Klausel definiert werden:

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

Falsch!

Die Unteranfragen der gleichen FROM-Klausel müssen also parallel auswertbar sein, und können nicht von einander abhängen. Man dürfte in den Unteranfragen aber auf Tupelvariablen zugreifen, die weiter außen deklariert sind.

Sichten (1)

- Eine Sichtdeklaration speichert eine Anfrage unter einem Namen in der Datenbank:

```
CREATE VIEW HA_PUNKTE AS
   SELECT   VORNAME, NACHNAME, ANR, PUNKTE
   FROM     STUDENTEN S, BEWERTUNGEN B
   WHERE    S.SID=B.SID AND ATYP = 'H'
```

- Sichten können in Anfragen wie gespeicherte Tabellen verwendet werden:

```
SELECT ANR, PUNKTE
FROM   HA_PUNKTE
WHERE  VORNAME='Michael' AND NACHNAME='Grau'
```

- Eine Sicht ist eine Abkürzung für eine Unteranfrage.

Sichten (2)

- Wird eine Sicht in einer Anfrage verwendet, so ersetzt das DBMS nur den Sichtnamen durch die Anfrage, für die er steht (man bekommt so Unteranfragen unter `FROM`).

  Sichten existieren schon in SQL-86. Da aber SQL-86 Unteranfragen unter `FROM` nicht enthielt, gab es komplexe Restriktionen zur Anwendung der Sichten.

- Durch Verwendung von Sichten kann man komplexe Anfragen Schritt für Schritt aufbauen.

  Man überlege aber, ob es das Verständnis wirklich fördert. Sind die einzelnen Schritte allzu klein, oder ist die Bedeutung der jeweiligen Sichten nicht klar, wäre vielleicht eine “monolithische” Anfrage einfacher.

- Sichten können auch angewendet werden, um Zugriffsrechte einzuschränken.

  Es ist möglich, dass ein Nutzer der Datenbank zwar Leserechte für eine Sicht hat, aber nicht für die zugrundeliegende Tabelle.

Lokale Sichten: WITH (1)

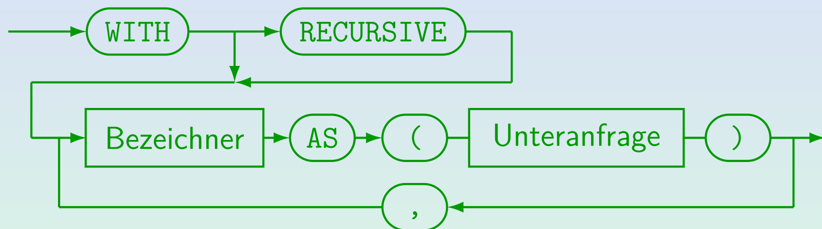
- Man kann eine Sicht auch nur für eine Anfrage definieren (“common table expression”):

```
WITH    HA_PUNKTE AS
        (SELECT VORNAME, NACHNAME, ANR, PUNKTE
         FROM   STUDENTEN S, BEWERTUNGEN B
         WHERE  S.SID=B.SID AND ATYP = 'H')
SELECT ANR, PUNKTE
FROM    HA_PUNKTE
WHERE   VORNAME='Michael' AND NACHNAME='Grau'
```

- Dies ist besonders nützlich, wenn man diese Unteranfrage mehrfach benötigt.

Lokale Sichten: WITH (2)

Lokale Sicht-Definition (vor SELECT-Anfrage):



- Lokale Sichtdefinitionen mit "`WITH`" wurden in SQL-99 eingeführt.
- Bei Oracle heisst es "`subquery_factoring_clause`".
- Bei Microsoft SQL Server heisst es "`common_table_expression`" (CTE). Dort kann man nach dem Sichtnamen in Klammern noch Spaltennamen angeben. "`RECURSIVE`" entfällt auch bei rekursiven Definitionen.
- Siehe auch: [Wikipedia: Hierarchical and Recursive Queries in SQL].

Inhalt

1. Monotonie
2. EXISTS-Unteranfragen
3. IN-Unteranfragen
4. ALL, ANY
5. Sichten
6. ORDER BY

Sortieren der Ausgabe (1)

- Wenn die Ausgabe länger als einige wenige Zeilen ist, sollte sie übersichtlich sortiert werden.

  Es ist viel einfacher, einen speziellen Wert in einer sortierten Tabelle zu suchen. Ohne "`ORDER BY`" bedeutet die Reihenfolge der Ausgabezeilen nichts (sie hängt von den verwendeten Algorithmen des DBMS ab).

- Es ist aber wichtig zu verstehen, dass die Entwicklung der Logik einer Anfrage und die Formatierung der Ausgabe zwei verschiedene Dinge sind.

  Während die Sortierung der einzige Formatierungsbefehl im SQL-Standard ist, bieten DBMS meist noch mehr Optionen. Z.B. einen Seitenumbruch zu machen bei Änderung des Wertes einer Spalte, negative Werte in Rot auszudrucken, etc. Die Sortierung kann jedoch auch wichtig sein, wenn ein Anwendungsprogramm die Daten erhält.

Sortieren der Ausgabe (2)

- Beispiel: Geben Sie die Namen der Studenten aus, die Hausaufgabe 1 gelöst haben. Sortieren Sie die Liste alphabetisch nach dem Nachnamen:

```
SELECT   S.VORNAME, S.NACHNAME
FROM     STUDENTEN S, BEWERTUNGEN B
WHERE    S.SID = B.SID
AND      B.ATYP = 'H' AND B.ANR = 1
ORDER BY S.NACHNAME
```

VORNAME	NACHNAME
Michael	Grau
Daniel	Sommer
Lisa	Weiss

Sortieren der Ausgabe (3)

- Man kann eine Liste von Sortierkriterien festlegen.

  Die "ORDER BY"-Liste kann mehrere Spalten enthalten. Die zweite Spalte wird nur zur Sortierung verwendet, wenn zwei Tupel den gleichen Wert in der ersten Spalte haben, usw. Weitere Sortierkriterien sind nur sinnvoll, wenn es Duplikate in den vorherigen Spalten geben kann.

- Z.B.: HA-Ergebnisse, sortiert nach Aufgabennummer, für jede Aufgabe nach Punkten (absteigend), bei gleicher Punktzahl alphabetisch nach Namen:

```
SELECT   ANR, PUNKTE, VORNAME, NACHNAME
FROM     STUDENTEN S, BEWERTUNGEN B
WHERE    S.SID = B.SID AND B.ATYP = 'H'
ORDER BY ANR, PUNKTE DESC, NACHNAME, VORNAME
```

Sortieren der Ausgabe (4)

- Ergebnis der Beispielanfrage der vorherigen Folie:

ANR	PUNKTE	VORNAME	NACHNAME
1	10	Lisa	Weiss
1	9	Michael	Grau
1	5	Daniel	Sommer
2	9	Michael	Grau
2	8	Lisa	Weiss

- Die ersten beiden Tupel haben den gleichen Wert im ersten Sortierkriterium (ANR), das zweite Kriterium (PUNKTE DESC) legt dann ihre Reihenfolge fest.

  Hierbei ist es egal, dass die Reihenfolge nach dem dritten Kriterium (NACHNAME) andersherum wäre.

Sortieren der Ausgabe (5)

- Nach dem SQL-92 Standard kann man nur nach Spalten sortieren, die ausgegeben werden.

  Z.B. ist es nicht möglich, eine Liste von Studierenden sortiert nach Gesamtpunktzahl zu erstellen, ohne diese auszugeben. Werkzeuge wie SQL*Plus können aber Ausgabespalten unterdrücken.

- Man kann aber in allen fünf Systemen (Oracle 8, DB2, SQL Server, Access, MySQL) nach jedem Term sortieren, der unter `SELECT` stehen könnte.

  In disem Systemen ist es nicht notwendig, dass der Term auch in der `SELECT`-Liste vorkommt. Z.B. könnte man nach `UPPER(NACHNAME)` sortieren, aber `NACHNAME` ausgeben. Bei Angabe von `DISTINCT` kann man dagegen nur nach Ergebnisspalten sortieren (in Oracle kann man sie noch in Ausdrücken verwenden, und MySQL hat keine Beschränkung).

Sortieren der Ausgabe (6)

- Manchmal muss man Spalten zu einer DB-Tabelle zufügen, um ein Sortierkriterium zu erhalten, z.B.
  - Die Ergebnisse sollen in der Reihenfolge “HA, Zwischen-, Endklausur” ausgegeben werden.
  - Die “MLU Halle-Wittenberg” sollte in einer Universitätsliste unter “H” stehen, nicht unter “M”.
- Wäre der Studentenname als eine Zeichenkette der Form “Vorname Nachname” gespeichert, wäre es sehr schwierig, nach dem Nachnamen zu sortieren.

Frage beim DB-Entwurf: Was will ich mit den Daten machen?

Sortieren der Ausgabe (7)

- "DESC" bedeutet descending/absteigend (von hoch zu tief), Default ist "ASC" (ascending/aufsteigend).
- Man kann sich auch durch Nummern auf Spalten beziehen, z.B.: `ORDER BY 2, 4 DESC, 1`

  Spaltennummern beziehen sich auf die Reihenfolge in der SELECT-Liste. Dies war in früheren SQL Versionen wichtig, weil man Ergebnisspalten wie z.B. SUM(PUNKTE) nicht benennen/umbenennen konnte. Heute sollte man Spaltennamen verwenden (übersichtlicher).
- Nullwerte werden alle als erstes oder als letztes aufgelistet (abhängig vom DBMS).

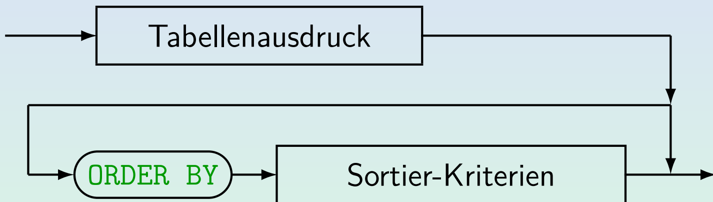
  In Oracle kann man NULLS FIRST oder NULLS LAST festlegen.

Sortieren der Ausgabe (8)

- Der Effekt von "`ORDER BY`" ist nur kosmetisch:
  Die Menge der Ausgabetupel wird nicht verändert.
- Deshalb kann "`ORDER BY`" nur am Ende einer Anfrage angewandt werden. Es kann nicht in Unteranfragen verwendet werden.
- Auch wenn mehrere `SELECT`-Ausdrücke mit `UNION` verknüpft werden, kann `ORDER BY` nur ganz am Ende stehen (es bezieht sich auf alle Ergebnistupel).

Sortieren der Ausgabe (9)

SQL-Anfrage:



Sortieren der Ausgabe (10)

Sortier-Kriterien:

- Die meisten DBMS lassen "Term" statt "Attribut-Referenz" zu (außer wenn `DISTINCT` oder `UNION` etc. spezifiziert wurden). Dann gelten die gleichen Beschränkungen wie für Terme in der `SELECT`-Liste (es kann weitere Beschränkungen für die Verwendung von Aggregationen geben).