Teil 6: SQL II

Literatur:

- Elmasri/Navathe:Fundamentals of Database Systems, 3. Auflage, 1999. Chap. 8, “SQL — The Relational Database Standard” (Sect. 8.2, 8.3.3, part of 8.3.4.)
- Silberschatz/Korth/Sudarshan: Database System Concepts, 3. Auflage. McGraw-Hill, 1999: Chapter 4: “SQL”.
- Kemper/Eickler: Datenbanksysteme , Kap. 4, Oldenbourg, 1997.
- Lipeck: Skript zur Vorlesung Datenbanksysteme, Univ. Hannover, 1996.
- Heuer/Saake: Datenbanken, Konzepte und Sprachen, Thomson, 1995.
- Date/Darwen: A Guide to the SQL Standard, 4. Auflage, Addison-Wesley, 1997.
- Date: A Guide to the SQL Standard, 1. Auflage, Addison-Wesley, 1987.
- van der Lans: SQL, Der ISO-Standard, Hanser, 1990.
- Sunderraman: Oracle Programming, A Primer. Addison-Wesley, 1999.
- Oracle 8i SQL Reference, Release 2 (8.1.6), Dec. 1999, Part No. A76989-01.
- Chamberlin: A Complete Guide to DB2 Universal Database. Morgan Kaufmann, 1998.
- Microsoft SQL Server Books Online: Accessing and Changing Data.
- Microsoft Jet Database Engine Programmer’s Guide, 2. Auflage (Part of MSDN Library Visual Studio 6.0).
- DuBois: MySQL. New Riders Publishing, 2000, ISBN 0-7357-0921-1, 756 pages.

Lernziele

Nach diesem Kapitel sollten Sie Folgendes können:

- Fortgeschrittene Anfragen in SQL schreiben, die Aggregationen, Unteranfragen und `UNION` enthalten.
- Die Teile einer SQL-Anfrage aufzählen/erklären.

  `SELECT, FROM, WHERE, GROUP BY, HAVING, ..., ORDER BY`
- Verbunde in SQL-92 erklären.
- Eine gegebene Anfrage auf syntaktische Korrektheit prüfen.
- Die Portabilität von Konstrukten beurteilen.

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

Nichtmonotones Verhalten (1)

- SQL-Anfragen, die nur die oben eingeführten Konstrukte beinhalten, berechnen monotone Funktionen auf den existierenden Tabellen.

  Nach Einfügungen erhält man mindestens die früheren Antworttupel.

- Aber nicht alle Anfragen verhalten sich auf diese Weise monoton: 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.

- Somit kann diese Anfrage nicht mit den bisher eingeführten SQL-Konstrukten formuliert werden.

Nichtmonotones Verhalten (2)

- 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.

- Bei der Formulierung einer Anfrage in SQL ist es wichtig festzustellen, ob die Anfrage benötigt, daß gewisse Tupel nicht existieren.

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	···

Unteranfragenergebnis

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)

- 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 (4)

- 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 (5)

- 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')
```

- Übung: 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 (6)

- 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.

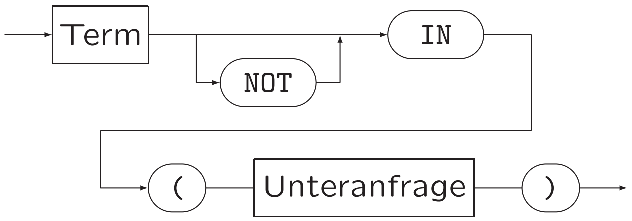
- In SQL-92 wurden Vergleiche auf Tupel-Ebene eingeführt, so daß man z.B. auch schreiben kann

```
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 (7)

Atomare Formel (Form 6):



- 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.

NOT IN (8)

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.

- Unterabfragen müssen immer in Klammern (...) eingeschlossen werden.

NOT EXISTS (1)

- Man kann in der äußeren Anfrage testen, ob das Ergebnis der Unteranfrage leer ist (`NOT EXISTS`).
- In der inneren Anfrage können Tupelvariablen, die in der `FROM`-Klausel der äußeren Anfrage deklariert sind, verwendet werden.

  Dies ist auch bei Unteranfragen mit `IN` möglich, aber es ist dort eine unnötige und unerwartete Komplikation (schlechter Stil).

- 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
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, damit ist die `NOT EXISTS`-Bedingung 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
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.

NOT EXISTS (7)

- 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).

NOT EXISTS (8)

- Unkorrelierte EXISTS-Unteranfragen sind fast immer falsch (aber unkorrelierte IN-Unteranfragen sind ok):

```
SELECT VORNAME, NACHNAME     Falsch!
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.

NOT EXISTS (9)

- 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)
```


NOT EXISTS (10)

- Im allgemeinen sucht der SQL-Parser bei Attributreferenzen ohne Tupelvariable die FROM-Klauseln beginnend mit der aktuellen Unteranfrage, hin zu den äußeren Anfragen, ab (verschachtelte Level).
- 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.

NOT EXISTS (11)

- 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.

NOT EXISTS (12)

- 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.
- Einige Autoren behaupten, daß in einigen Systemen `SELECT null` oder `SELECT 1` schneller als `SELECT *` ist.

  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.

NOT EXISTS (13)

Atomare Formel (Form 7):

- 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.

NOT EXISTS (14)

- 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:

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

- Das Muster $\forall s\ X\colon (F_1 \rightarrow F_2)$ 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 $\forall s\ X\colon F$ äquivalent ist zu $\neg\exists s\ X\colon \neg F$. Ein Quantor genügt also.

  "F ist wahr für alle X" ist das gleiche wie "F ist falsch für kein X".
- Das Muster $\forall s\ X\colon (F_1 \rightarrow F_2)$ ist äquivalent zu $\neg\exists s\ X\colon F_1 \land \neg F_2$.
- Im Beispiel ergibt sich:

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

- 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)

Übungen:

- 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
```


Häufige 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.

Häufige 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?

Häufige Fehler (4)

- 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 (5)

- Was ist der Fehler in dieser 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)
```

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

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

Häufige Fehler (6)

- 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)
```


Häufige Fehler (7)

- Läßt sich diese Anfrage vielleicht vereinfachen?

```
SELECT X.ATYP, X.ANR
FROM   AUFGABEN X
WHERE  X.THEMA NOT IN (SELECT Y.THEMA
                       FROM   AUFGABEN Y
                       WHERE  Y.THEMA = 'SQL'
                       OR     Y.THEMA = 'ER')
```

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 `IN (SELECT` t_2
  `FROM` $R_1\ X_1$, ..., $R_n\ X_n$
  `WHERE` φ`)`

  ist (unter gewissen Voraussetzungen) äquivalent zu

  `EXISTS (SELECT *`
  `FROM` $R_1\ X_1$, ..., $R_n X_n$
  `WHERE (`φ`) 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.

IN vs. EXISTS (3)

- Außerdem gilt die Äquivalenz nur, wenn die Unteranfrage für t_2 keine Nullwerte liefert.
- Falls die Unteranfrage nur Werte liefert, die verschieden von t_1 sind, und einen Nullwert, so
  - ◇ Liefert die `IN`-Bedingung den dritten Wahrheitswert "unbekannt".

    Sie wird wie eine große Disjunktion von Gleichungen $t_1 = c$ behandelt, wobei für c alle Werte eingesetzt werden, die die Unteranfrage liefert.
  - ◇ Die `EXISTS`-Bedingung dagegen "falsch".

IN vs. EXISTS (4)

- Der Unterschied zwischen den Wahrheitswerten "unbekannt" und "falsch" ist wichtig, wenn anschließend eine Negation erfolgt (wegen `NOT IN`).
- Beispiel: Die Punkte-DB sei um eine Tabelle mit Kapiteln der Vorlesung erweitert. `THEMA` in `AUFGABEN` verweist jetzt auf diese Tabelle und auch Null sein:

AUFGABEN

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

KAPITEL

THEMA	...
Einführung	...
ER	...
SQL	...

IN vs. EXISTS (5)

- Die Anfrage nach Kapiteln ohne Aufgaben funktioniert nicht, wenn sie mit `IN` formuliert wird:

```
SELECT K.THEMA
FROM   KAPITEL K
WHERE  K.THEMA NOT IN (SELECT A.THEMA
                       FROM   AUFGABEN A)
```

- Die Ausgabe ist leer, obwohl intuitiv "`Einführung`" herauskommen sollte.
- Grund ist der Nullwert, den die Unteranfrage liefert.

  Vermutlich ist es fast immer ein Fehler, "`NOT IN`" mit einer Unteranfrage zu verwenden, die Nullwerte liefern kann.

IN vs. EXISTS (6)

- Die entsprechende Anfrage mit `EXISTS` funtioniert dagegen (es wird "`Einführung`" ausgegeben):

```
SELECT K.THEMA
FROM   KAPITEL K
WHERE  NOT EXISTS(SELECT *
                  FROM   AUFGABEN A
                  WHERE  A.THEMA = K.THEMA)
```

- Dies zeigt wieder die Tücken dreiwertiger Logik.

  Bei der `NOT EXISTS`-Bedingung entsteht der "unbekannt" im Innern der Unteranfrage. Die Unteranfrage behandelt ihn dann wie "falsch" (sie gibt ja in diesem Fall nichts aus). Bei "`NOT IN`" entsteht der dritte Wahrheitswert erst außerhalb der Unteranfrage. Bei "`NOT IN`" muß man "`WHERE A.THEMA IS NOT NULL`" in der Unteranfrage fordern.

IN vs. EXISTS (7)

- Theoretisch ist interessant, daß man

```
t1 IN (SELECT t2
       FROM R1 X1, ..., Rn Xn
       WHERE φ)
```

  exakt in eine `EXISTS`-Bedingung übersetzen kann:

```
   EXISTS (SELECT *
           FROM R1 X1, ..., RnXn
           WHERE (φ) AND t1 = t2)
OR 1 = NULL AND  -- Wahrheitswert "unbekannt"
   EXISTS (SELECT *
           FROM R1 X1, ..., RnXn
           WHERE (φ) AND t2 IS NULL)
```


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" (∀) und "es gibt kein X, so daß nicht" (¬∃¬) 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`)`

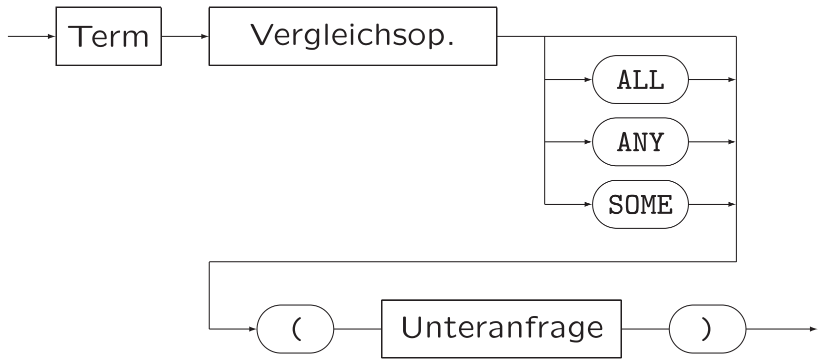
  Es gelten die gleichen Einschränkungen wie oben für `IN` erklärt, auch das Problem mit dem Nullwert für t_2.

- 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)

Atomare Formel (Form 8):



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
```


Ein-Wert-Unteranfragen (3)

- Wenn die Unteranfrage ein leeres Ergebnis hat, wird stattdessen der Nullwert verwendet.
- Z.B. ist dies eine seltsame Art nach Studenten zu fragen, die Hausaufgabe 1 noch nicht gelöst haben:

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

Schlechter Stil!

- In SQL86 und Oracle8 ist dies ein Syntaxfehler.

Unteranfragen unter FROM (1)

- Da das Ergebnis einer SQL-Anfrage eine Tabelle ist, ist es klar, 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 verbinden kann.

  In der relationalen Algebra kann man stets an Stelle eines Relationsnamen eine Unteranfrage schreiben (Ausdruck der relationalen Algebra).

- Trotzdem werden Unteranfragen unter FROM selten benötigt, und verkomplizieren evtl. nur die Anfrage.

Unteranfragen unter FROM (2)

- Unteranfragen unter FROM werden z.B. für geschachtelte Aggregationen benötigt, siehe unten.
- In diesem Beispiel wird der Verbund von AUFGABEN und BEWERTUNGEN in einer Unteranfrage berechnet (ergibt sich z.B. aus Verwendung einer Sicht):

```
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 (3)

- 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)

- 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!

- Unteranfragen unter FROM sind häufig unnötig und machen die Anfrage schwerer zu verstehen.

Unteranfragen unter FROM (5)

- 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.

Unteranfragen unter FROM (6)

- Wird eine Sicht in einer Anfrage verwendet, so ersetzt das DBMS nur den Sichtnamen durch die Anfrage, für die er steht.

  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.

  Ist der Anfrageoptimierer nicht sehr gut, kann es sein, daß eine so gebildete Anfrage langsamer als eine einzelne "monolithische" Anfrage läuft. Aber es sollte keine Unterschiede geben zur Nutzung von Unteranfragen unter `FROM`. Eine Verbesserung der Performance ist nur möglich, wenn man die Anfrage ohne Unteranfragen formulieren kann.

Inhalt

Aggregationen (1)

- Aggregationsfunktionen sind Funktionen von einer Menge oder Multimenge zu einem einzelnen Wert.

  E.g.: $min\{41, 57, 19, 23, 27\} = 19$

- Aggregationsfunktionen fassen eine ganze Menge von Werten zu einem einzelnen Wert zusammen.

  Aggregationsfunktionen nennt man auch "Mengenfunktionen", "Gruppenfunktionen" oder "Spaltenfunktionen". Sie haben nicht einen einzelnen Wert als Eingabe, sondern eine ganze Spalte (eine Menge). Die Spalte muss keine Spalte einer gespeicherten Tabelle sein, sie kann auch durch eine Anfrage erstellt werden.

- Aggregationsfunktionen werden oft für statistische Auswertungen verwendet (z.B. Durchschnitt/Avg).

Aggregationen (2)

- SQL-86/92 hat die fünf Aggregationsfunktionen `COUNT`, `SUM`, `AVG`, `MAX`, `MIN`.

  Zusätzliche Aggregationsfunktionen in einigen Systemen:
  Oracle 8i: `CORR` (Korrelation, arbeitet auf einer Menge von Paaren), `COVAR_POP`, `COVAR_SAMP`, lineare Regressionsfunktionen, `STDDEV`, `STDDEV_POP`, `STDEV_SAMP`, `VARIANCE`, `VAR_POP`, `VAR_SAMP`.
  DB2: `CORRELATION`, `COUNT_BIG`, `COVARIANCE`, Regressionsfunktionen, `STDDEV`, `VARIANCE`.
  SQL Server: `VAR`, `VARP`, `STDEV`, `STDEVP`.
  Access: `VAR`, `VARP`, `STDEV`, `STDEVP`, `FIRST`, `LAST`.
  MySQL: `STD`. MySQL unterstützt `DISTINCT` aber nur für `COUNT`.

- Jeder kommutative, assoziative Binäroperator mit neutralem Element kann so erweitert werden, daß er auf Mengen arbeitet. Z.B. ist sum die Mengenversion von $+$.

Aggregationen (3)

- Für einige Aggregationsfunktionen sind Duplikate wichtig (z.B. `SUM`), für andere nicht (z.B. `MIN`).

  Z.B. die Summe aller Bestandteile einer Rechnung. Auch wenn zwei Teile das gleiche kosten, müssen trotzdem beide aufsummiert werden.

- In SQL kann man Duplikatelimination fordern (Eingabe: Menge), oder nicht (Eingabe: Multimenge).

  Eine Multimenge ist eine Menge, in der jedes Element eine Vielfachheit hat, z.B. kann ein Element in einer Multimenge zweimal vorkommen. Im Gegensatz zu einer Liste gibt es keine spezielle Anordnung.

- `SUM(DISTINCT X)` und `AVG(DISTINCT X)`: meist falsch. Einige Studenten verwechseln `SUM` und `COUNT`.

Einfache Aggregationen (1)

- Zunächst werden Aggregationen über alle Ergebniszeilen einer Anfrage erklärt.

  Aggregationen über Gruppen von Zeilen: Siehe nächster Abschnitt.

- Wieviele Studenten gibt es in der Datenbank?

```
SELECT COUNT(*)
FROM   STUDENTEN
```

COUNT(*)
4

- Was ist das beste Ergebnis für Hausaufgabe 1?

```
SELECT MAX(PUNKTE)
FROM   BEWERTUNGEN
WHERE  ATYP = 'H' AND ANR = 1
```

MAX(PUNKTE)
10

Einfache Aggregationen (2)

- Wie viele Studierende haben mindestens eine Hausaufgabe abgegeben?

```
SELECT COUNT(DISTINCT SID)
FROM   BEWERTUNGEN
WHERE  ATYP = 'H'
```

COUNT(DISTINCT SID)
3

- Wieviele Punkte hat Studentin 101 insgesamt für Hausaufgaben bekommen?

```
SELECT SUM(PUNKTE) "Gesamtpunkte"
FROM   BEWERTUNGEN
WHERE  SID = 101 AND ATYP = 'H'
```

Gesamtpunkte
18

Einfache Aggregationen (3)

- Wieviel Prozent der Maximalpunktzahl haben die Studenten für HA 1 durchschnittlich bekommen?

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

- Z.B. Hausaufgabenpunkte von Studentin 101 plus 3 Extrapunkte:

```
SELECT SUM(PUNKTE) + 3 "Gesamte HA-Punkte"
FROM   BEWERTUNGEN
WHERE  SID = 101 AND ATYP = 'H'
```

Einfache Aggregationen (4)

- Man kann auch mehr als eine Aggregation in der SELECT-Liste berechnen, z.B.: Was ist die minimale und maximale Punktzahl für Hausaufgabe 1?

```
SELECT MIN(PUNKTE), MAX(PUNKTE)
FROM   BEWERTUNGEN
WHERE  ATYP = 'H' AND ANR = 1
```

- Die Aggregationen können sich auf verschiedene Spalten beziehen:

```
SELECT COUNT(DISTINCT THEMA), AVG(MAXPT)
FROM   AUFGABEN A
```


Aggregationsanfragen

- Es gibt drei Typen von Anfragen in SQL:
  - ◇ Anfragen ohne Aggregationsfunktionen und ohne GROUP BY und HAVING: siehe oben.
  - ◇ Anfragen mit Aggregationsfunktionen, aber ohne GROUP BY: Ergebnis ist immer genau eine Zeile.

    Diese wurden oben "einfache Aggregationen" genannt. Die Aggregationsfunktion kann dann nur unter SELECT auftauchen (außerdem sind bei jedem Typ Aggregationen in Unteranfragen möglich).
  - ◇ Anfragen mit GROUP BY.
- Jeder Typ hat verschiedene Syntaxrestriktionen und wird auf verschiedene Weisen ausgewertet.

Auswertung (1)

- Zunächst wird die FROM-Klausel ausgewertet.

  Theoretisch werden alle möglichen Tupelkombinationen der unter FROM genannten Tabellen konstruiert (mögliche Variablenbelegungen, $\times$).
- Als zweites wird die WHERE-Klausel ausgewertet.

  Nur die Tupelkombinationen (Variablenbelegungen), die die Bedingung erfüllen, werden weiter betrachtet (Selektion, Filter). Reale Systeme kombinieren beide Schritte für eine effizientere Auswertung.
- Gibt es keine Aggregation, GROUP BY, und HAVING, so wird anschließend die SELECT-Klausel ausgewertet, indem man die Werte der Terme in der SELECT-Liste für die restlichen Tupelkombinationen ausgibt.

Auswertung (2)

- Beim zweiten Anfragetyp (SELECT enthält Aggregationsterm, aber es gibt aber kein GROUP BY) wird nur eine einzelne Ausgabezeile berechnet.
- Anstatt die Werte der unter SELECT genannten Spalten auszugeben, werden sie in eine (Multi-)Menge eingefügt, die dann als Eingabe für die Aggregationsfunktion dient.

  Enthält die SELECT-Liste mehrere Aggregationen, müssen mehrere solcher Mengen verwaltet werden. Ist kein DISTINCT angegeben (Multimenge), so können die aggregierten Werte inkrementell ohne explizites Speichern der temporären Menge berechnet werden (vgl. nächste Folie).

Auswertung (3)

- Beispiel für inkrementelle Berechnung:

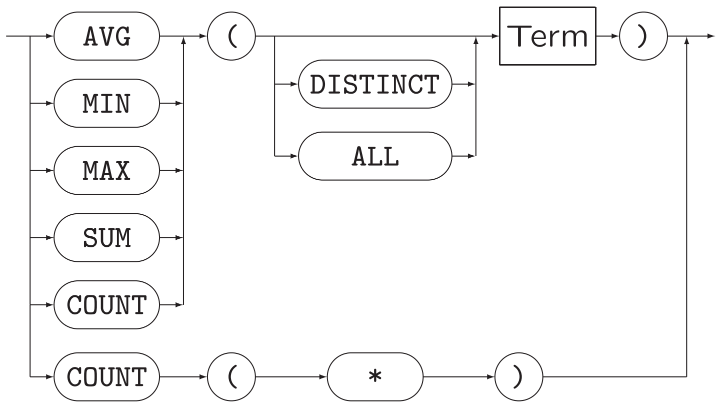
```
SELECT SUM(MAXPT), COUNT(*)
FROM   AUFGABEN A
WHERE  ATYP = 'H'
```

- Dies wird so ausgewertet:

```
ausgabe1 = 0; ausgabe2 = 0;
foreach row A in AUFGABEN do
        if A.ATYP = 'H' then begin
                ausgabe1 = ausgabe1 + A.MAXPT;
                ausgabe2 = ausgabe2 + 1;
        end;
print ausgabe1, ausgabe2;
```


Syntax

Aggregationsterm:



- MySQL unterstützt `DISTINCT` nur für `COUNT` (und versteht `ALL` nicht).

Syntax / Restriktionen (1)

- `SUM` und `AVG` müssen numerische Argumente haben. `COUNT`, `MIN`, und `MAX` akzeptieren jeden Datentyp.
- Aggregationen können nicht verschachtelt werden, z.B. ist folgendes unzulässig:

  `AVG(COUNT(*))` **Falsch!**

  `COUNT` liefert einen einzelnen Wert. Damit ist die Anwendung einer zweiten Aggregation sinnlos.

  Es ist möglich Aggregationen zunächst auf Gruppen von Zeilen anzuwenden, und dann das Ergebnis als Eingabe für eine andere Aggregation zu verwenden. Z.B. Was ist die durchschnittliche Gesamtpunktzahl, die Studenten für ihre Hausaufgaben erhalten haben? Dazu verwendet man `GROUP BY` und Unteranfragen (siehe unten).

Syntax / Restriktionen (2)

- Die `WHERE`-Bedingung kann nicht direkt Aggreationsterme enthalten, nur in Unteranfragen.

  Die `WHERE`-Bedingung wird vor der Berechnung der Aggregation ausgewertet (sie legt fest, welche Tupel in die Aggregation eingehen). Bedingungen an Aggregationen können unter `HAVING` festgelegt werden.

  `WHERE COUNT(*) > 1` **Falsch!**

- Bei einfachen Aggregationen können keine normalen Attribute in der `SELECT`-Liste auftauchen.

  Da kein Attribut außerhalb der Aggregation nur einen einzelnen Ausgabewert hat. Aber man beachte `GROUP BY`.

```
SELECT ATYP, ANR, AVG(PUNKTE)   Falsch!
FROM   BEWERTUNGEN
```

Syntax / Restriktionen (3)

- Jeder Aggregationsoperator benötigt ein Argument (welches die Eingabewerte spezifiziert).

```
SELECT SID
FROM   BEWERTUNGEN
WHERE  ATYP = 'H' AND ANR = 1
AND    PUNKTE = MAX
```
Falsch! Falsch!

Aggregationen sind auch unter WHERE nicht erlaubt.

- Es wird eine Unteranfrage benötigt, um den Studenten mit dem besten Ergebnis für Hausaufgabe 1 zu finden (siehe unten).

Nullwerte in Aggregationen

- Normalerweise werden Nullwerte herausgefiltert, bevor die Aggregationsfunktion angewendet wird.
- Nur COUNT(*) beinhaltet Nullwerte (da es Zeilen und keine Attributwerte zählt)
- Der einzige Unterschied zwischen COUNT(EMAIL) und COUNT(*) ist, daß das erste nur die Zeilen zählt, wo EMAIL nicht Null ist, und das zweite alle Zeilen zählt.

  Andererseits ist der Attributwert nicht wichtig für COUNT, und man sollte wahrscheinlich COUNT(*) verwenden. Wenn natürlich Tupel, wie in COUNT(DISTINCT ATYP) eliminiert werden, dann ist der Attributwert offensichtlich wichtig.

Leere Aggregationen

- Ist die Eingabemenge leer, ergeben die meisten Aggregationen einen Nullwert, nur COUNT ergibt 0.

  Dies widerspricht zumindest für SUM der Intuition. Man würde erwarten, daß die Summe über die leere Menge 0 ist, aber in SQL erhält man NULL. (Ein Grund dafür könnte sein, daß die SUM-Aggregationsfunktion keinen Unterschied entdeckt, zwischen der leeren Eingabemenge, weil es keine qualifizierenden Tupel gibt, und der leeren Eingabemenge, weil alle qualifizierenden Tupel Nullwerte in diesem Argument haben.)

- Da es vorkommen kann, daß keine Zeile die WHERE-Bedingung erfüllt, müssen Programme mit dem resultierenden Nullwert arbeiten können.

  Alternativ: Verwende z.B. NVL(SUM(PUNKTE),0) in Oracle, um den Nullwert zu ersetzen.

Inhalt

1. Unteranfragen, Nichtmonotone Konstrukte
2. Aggregationen I: Aggregationsfunktionen
3. Aggregationen II: GROUP BY, HAVING
4. UNION, Bedingte Ausdrücke
5. Sortieren der Ausgabe: ORDER BY
6. SQL-92 Verbunde, Äußerer Verbund in Oracle

GROUP BY (1)

- Die obigen SQL-Konstrukte können nur eine einzelne aggregierte Ausgabezeile erzeugen.
- Mit der "GROUP BY" Klausel kann man über Gruppen von Tupeln aggregieren (anstatt über alle Tupel).
- Berechnen Sie die durchschnittliche Punktzahl für jede Hausaufgabe:

```
SELECT   ANR, AVG(PUNKTE)
FROM     BEWERTUNGEN
WHERE    ATYP = 'H'
GROUP BY ANR
```

ANR	AVG(PUNKTE)
1	8
2	8.5

GROUP BY (2)

- Das Zwischenergebnis nach Auswertung von FROM und WHERE wird jetzt in Gruppen aufgespalten, wobei die Tupel einer Gruppe jeweils den gleichen Wert in den GROUP BY-Spalten haben.

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

- Man erhält dann eine Ausgabezeile pro Gruppe.

GROUP BY (3)

- Diese Konstruktion kann niemals zu leeren Gruppen führen. Somit ist es unmöglich, daß ein COUNT(*) den Wert 0 ergibt.

  Der Wert 0 kann mit COUNT(A) entstehen, wenn das Attribut A Null ist. Wenn eine Anfrage Gruppen mit count 0 ergeben muss, wird vermutlich ein äußerer Verbund benötigt (siehe unten).

- Einfache Aggregationen (ohne GROUP BY) ergeben immer genau eine Ausgabezeile: Wenn die Eingabemenge leer ist, erhält man COUNT(*) = 0.

  Dagegen kann eine GROUP BY-Anfrage keine, eine oder mehrere Ausgabezeilen liefern.

GROUP BY (4)

- Da GROUP BY-Attribute einen eindeutigen Wert für jede Gruppe haben, können sie ausgegeben werden.

  Andere Attribute können unter SELECT nur in Aggregationen stehen.

- Z.B. folgendes ist unzulässig:

```
SELECT A.ANR, A.THEMA, AVG(B.PUNKTE)   Falsch!
FROM   AUFGABEN A, BEWERTUNGEN B
WHERE  A.ATYP='H' AND B.ATYP='H' AND A.ANR=B.ANR
GROUP BY A.ANR
```

  A.THEMA ist kein GROUP BY-Attribut, deshalb darf es nicht unter SELECT außerhalb von Aggregationsfunktionen verwendet werden. Die SQL-Regel ist rein syntaktisch: Weil (ATYP, ANR) Schlüssel von AUFGABEN ist, wäre THEMA durchaus eindeutig innerhalb der Gruppen.

GROUP BY (5)

- Also muss man nach A.ANR und A.THEMA gruppieren:

```
SELECT A.ANR, A.THEMA, AVG(B.PUNKTE)
FROM   AUFGABEN A, BEWERTUNGEN B
WHERE  A.ATYP='H' AND B.ATYP='H' AND A.ANR=B.ANR
GROUP BY A.ANR, A.THEMA
```

A.ANR	A.THEMA	AVG(B.PUNKTE)
1	Rel. Algeb.	8
2	SQL	8.5

- Das Hinzufügen von A.THEMA zu GROUP BY ändert die Gruppen nicht, aber man kann es nun ausgeben.

GROUP BY (6)

- Übung: Gibt es einen echten Unterschied zwischen

```
SELECT THEMA, AVG(PUNKTE*100/MAXPT)
FROM   AUFGABEN A, BEWERTUNGEN B
WHERE  A.ATYP='H' AND B.ATYP='H' AND A.ANR=B.ANR
GROUP BY THEMA
```

und der Anfage, die zusätzlich nach A.ANR gruppiert, die Aufgabennummer aber nicht ausgibt?

```
SELECT THEMA, AVG(PUNKTE*100/MAXPT)
FROM   AUFGABEN A, BEWERTUNGEN B
WHERE  A.ATYP='H' AND B.ATYP='H' AND A.ANR=B.ANR
GROUP BY THEMA, A.ANR
```


GROUP BY (7)

- GROUP BY wird vor SELECT ausgewertet. Deshalb kann man sich nicht auf neue Attributnamen beziehen:

```
SELECT   FLOOR((PUNKTE/MAXPT)*100+0.5) PROZENTE,
         COUNT(*)
FROM     AUFGABEN A, BEWERTUNGEN B
WHERE    A.ATYP = B.ATYP AND A.ANR = B.ANR
GROUP BY PROZENTE
```
Falsch!

- Oracle, SQL Server, DB2, MySQL und Access unterstüzen GROUP BY mit beliebigen Termen. SQL92 Standard erlaubt GROUP BY nur mit Spaltennamen.

  D.h. GROUP BY FLOOR(...) funktioniert in diesen Systemen. Portabele Alternative: Unteranfrage unter FROM oder Verwendung einer Sicht.

GROUP BY (8)

- Die Reihenfolge der Attribute unter "GROUP BY" ist nicht wichtig.

  GROUP BY A, B bedeutet, daß zwei Tupel t, u in die gleiche Gruppe gehören, falls $t.A = u.A$ und $t.B = u.B$.
  GROUP BY B, A bedeutet, daß zwei Tupel t, u in die gleiche Gruppe gehören, falls $t.B = u.B$ und $t.A = u.A$.

- Es macht keinen Sinn, nach einem Schlüssel zu gruppieren (bei nur einer Tabelle unter FROM): Dann besteht jede Gruppe nur aus einer Zeile.
- Ebenso ist GROUP BY nicht sinnvoll, wenn es nur eine einzige Gruppe liefern kann.

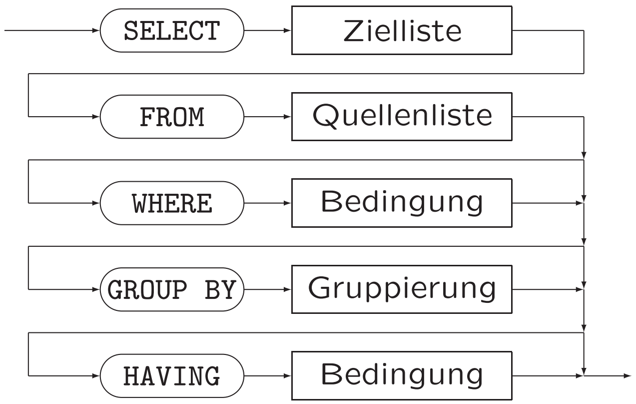
GROUP BY (9)

Warnung:

- "GROUP BY" wird öfters mit "ORDER BY" verwechselt:
  - ◇ GROUP BY ist wichtig für das Anfrageergebnis.
  - ◇ ORDER BY ist nur kosmetisch (schönere Ausgabe).
- "GROUP BY" sortiert normalerweise intern die Tupel (so daß Tupel mit gleichem Wert benachbart sind).
- Aber dann führt GROUP BY die Gruppierung durch.
- Man kann sich auch nicht darauf verlassen, daß "GROUP BY" als Nebeneffekt sortiert: Eventuell kann das DBMS es effizienter anders auswerten.

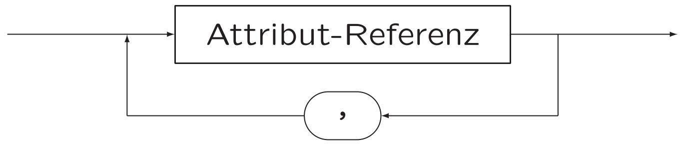
Syntax (1)

SELECT-Ausdruck:



Syntax (2)

Gruppierung:



- Z.B. GROUP BY VORNAME, NACHNAME, B.SID
- Oracle, SQL Server, DB2, Access und MySQL unterstützen den allgemeineren "Term" anstatt "Attribut-Referenz". Natürlich sind Aggregationen unter GROUP BY nicht gestattet.

HAVING (1)

- Aggregationen sind unter WHERE verboten.
- Manchmal benötigt man Aggregationen zur Filterung von Ausgabezeilen.

  Und nicht nur zur Berechnung von Ausgabewerten.
- Deshalb gibt es die HAVING-Klausel: Hier kann man eine Bedingung mit Aggregationsfunktionen angeben. So kann man ganze Gruppen eliminieren.
- Wie bei SELECT dürfen auch unter HAVING außerhalb von Aggregationen nur GROUP BY-Attribute stehen.

HAVING (2)

- Wer hat mindestens 18 Hausaufgabenpunkte?

```
SELECT   VORNAME, NACHNAME
FROM     STUDENTEN S, BEWERTUNGEN B
WHERE    S.SID = B.SID AND B.ATYP = 'H'
GROUP BY S.SID, VORNAME, NACHNAME
HAVING   SUM(PUNKTE) >= 18
```

VORNAME	NACHNAME
Lisa	Weiss
Michael	Grau

- Die WHERE-Bedingung bezieht sich auf jeweils eine Tupelkombination (Variablenbelegung), die HAVING-Bedingung dagegen auf ganze Gruppen.

Auswertung

1. Alle Kombinationen von Zeilen der Tabellen unter FROM (Variablenbelegungen) werden betrachtet.
2. Die WHERE-Bedingung filtert eine Teilmenge heraus.
3. Dieses Zwischenergebnis wird in Gruppen aufgespalten, jeweils mit gleichem Wert in den GROUP BY-Attributen.
4. Gruppen, die die Bedingung in der HAVING-Klausel nicht erfüllen, werden eliminiert.
5. Für jede Gruppe wird durch Auswertung der Terme in der SELECT-Klausel eine Ausgabezeile erstellt.

Syntax: Restriktionen

- Eine Aggregation wird ausgeführt, wenn
  - ◇ eine Aggregation unter SELECT verwendet wird,
  - ◇ oder die GROUP BY oder HAVING-Klausel auftritt.
- Wird eine Aggregation ausgeführt, dann können unter SELECT und HAVING außerhalb von Aggregationen nur GROUP BY-Attribute genutzt werden.

  Innerhalb von Aggregationsfunktionen, d.h. als ihre Argumente, sind alle Attribute erlaubt. Man betrachte z.B. AVG(A)/B: Das Attribut A steht hier innerhalb der Aggregationsfunktion, B außerhalb.
- HAVING ohne GROUP BY ist legal, aber ungewöhnlich. Die Anfrage kann nur 0 oder 1 Ausgabezeile haben.

WHERE vs. HAVING

- Meist legen die Syntaxregeln schon fest, ob eine Bedingung unter WHERE oder unter HAVING gehört.

  Nur wenn eine Bedingung ausschließlich GROUP BY-Attribute, aber keine Aggregationen enthält, wäre sie in beiden Klauseln erlaubt.
- Wenn beides möglich ist, ist es wesentlich effizienter es unter WHERE zu stecken. Z.B. diese Anfrage ist zulässig, aber langsam und braucht viel Speicher:

```
SELECT   VORNAME, NACHNAME
FROM     STUDENTEN S, BEWERTUNGEN B
GROUP BY S.SID, B.SID, VORNAME, NACHNAME
HAVING   S.SID = B.SID AND SUM(PUNKTE) >= 18
```

Aggregationsunteranfragen (1)

- Wer hat das beste Ergebnis 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    B.PUNKTE = (SELECT MAX(PUNKTE)
                   FROM   BEWERTUNGEN
                   WHERE  ATYP='H' AND ANR=1)
```

- Bei der Unteranfrage ist garantiert, daß man genau eine Ergebniszeile erhält (Aggregationsanfrage ohne GROUP BY): Daher sind ANY/ALL nicht erforderlich.

Aggregationsunteranfragen (2)

- In Systemen, in denen Ein-Wert-Unteranfragen wie ein Term verwendet werden können, sind Unteranfragen auch unter SELECT möglich.

  Dies funktioniert in SQL-92, DB2, Oracle 9i, SQL Server und Access. SQL-86 und z.B. Oracle 8.0 unterstützten es nicht.

- Das kann GROUP BY ersetzen. Z.B. Geben Sie für jeden Studenten die Summe der HA-Punkte aus:

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


Geschachtelte Aggregation (1)

- Verschachtelte Aggregationen (z.B. Durchschnitt über Summen) benötigen Unteranfragen unter FROM.
- Was ist die durchschnittliche Anzahl HA-Punkte?

```
SELECT AVG(X.HA_PKT)
FROM   (SELECT   SID, SUM(PUNKTE) AS HA_PKT
        FROM     BEWERTUNGEN
        WHERE    ATYP = 'H'
        GROUP BY SID) X
```

X

SID	HA_PKT
101	18
102	18
103	5

AVG(X.HA_PKT)
13.67

Bemerkung:
Es werden hier nur Studierende gezählt, die mindestens eine Aufgabe gelöst haben.

Geschachtelte Aggregation (2)

- Oracle unterstützt auch folgende Syntax für verschachtelte Aggregationen:

```
SELECT   AVG(SUM(PUNKTE))
FROM     BEWERTUNGEN
WHERE    ATYP = 'H'
GROUP BY SID
```

Nur Oracle!

Das ist aber nicht Standard (wird nicht unterstützt in SQL92, DB2, SQL Server, Access).

Da es wesentlich kürzer, als die äquivalente Standard-Anfrage ist, kann es bei Ad-hoc-Anfragen bequemer sein. In Anwendungsprogrammen sollte man aber keine unnötigen Übertragbarkeitsprobleme schaffen.

Agg. über mehrere Mengen (1)

- Durch Unteranfragen unter FROM kann man in einer Anfrage über verschiedene Mengen aggregieren:

```
SELECT VORNAME, NACHNAME, H.PT AS HA, Z.PT AS ZK
FROM   STUDENTEN S,
       (SELECT   SID, SUM(PUNKTE) AS PT
        FROM     BEWERTUNGEN
        WHERE    ATYP = 'H'
        GROUP BY SID) H,
       (SELECT   SID, SUM(PUNKTE) AS PT
        FROM     BEWERTUNGEN
        WHERE    ATYP = 'Z'
        GROUP BY SID) Z
WHERE  S.SID = H.SID AND S.SID = Z.SID
```


Agg. über mehrere Mengen (2)

- Aggregation über verschiedenen Mengen ist auch mit bedingten Ausdrücken möglich, z.B. in Oracle:

```
SELECT VORNAME, NACHNAME,
       SUM(DECODE(B.ATYP, 'H', B.PUNKTE, 0)) HA
       SUM(DECODE(B.ATYP, 'Z', B.PUNKTE, 0)) ZK
FROM   STUDENTEN S, BEWERTUNGEN B
WHERE  S.SID = B.SID
```

- Z.B. liefert der bedingte Ausdruck

```
DECODE(B.ATYP, 'H', B.PUNKTE, 0)
```

B.PUNKTE, falls B.ATYP = 'H', und 0 sonst.

Aggregationen maximieren (1)

- Wer hat das beste Ergebnis in den Hausaufgaben (maximale Summe der Hausaufgabenpunkte)?

```
SELECT   VORNAME, NACHNAME, SUM(PUNKTE) AS SUMME
FROM     STUDENTEN S, BEWERTUNGEN B
WHERE    S.SID = B.SID AND B.ATYP = 'H'
GROUP BY S.SID, VORNAME, NACHNAME
HAVING   SUM(PUNKTE) >= ALL(SELECT   SUM(PUNKTE)
                            FROM     BEWERTUNGEN
                            WHERE    ATYP = 'H'
                            GROUP BY SID)
```

- Alternative Lösung (mit Sicht): siehe nächste Folie.

Aggregationen maximieren (2)

- Gesamtpunktzahl der HA für jeden Studenten:

```
CREATE VIEW HA_SUMMEN AS
   SELECT   SID, SUM(PUNKTE) AS SUMME
   FROM     BEWERTUNGEN
   WHERE    ATYP = 'H'
   GROUP BY SID
```

- Dann kann man dies wie folgt verwenden:

```
SELECT S.VORNAME, S.NACHNAME, H.SUMME
FROM   STUDENTEN S, HA_SUMMEN H
WHERE  S.SID = H.SID
AND    H.SUMME = (SELECT MAX(SUMME)
                  FROM   HA_SUMMEN)
```

Übung: Mögliche Fehler (1)

- Die folgende Anfrage soll alle Studenten ausgeben, die mindestens zwei Hausaufgaben gelöst haben.

```
SELECT VORNAME, NACHNAME
FROM   STUDENTEN S
WHERE  2 <= (SELECT COUNT(S.SID)
             FROM   BEWERTUNGEN B
             WHERE  B.SID = S.SID
             AND    B.ATYP = 'H')
```

- In der Unteranfrage wird aber `S.SID` gezählt, das für jede (konzeptionelle) Ausführung der Unteranfrage einen festen Wert hat. Funktioniert es trotzdem?

Übung: Mögliche Fehler (2)

- Was halten Sie von dieser Anfrage? Wieder ist die Aufgabe, alle Studenten aufzulisten, die mindestens zwei Hausaufgaben gelöst haben.

```
SELECT VORNAME, NACHNAME
FROM   STUDENTEN S, BEWERTUNGEN B
WHERE  S.SID = B.SID
AND    B.ATYP = 'H'
AND    COUNT(B.ANR) >= 2
```


Übung: Mögliche Fehler (3)

- Und was ist mit dieser Anfrage? Hier ist die Aufgabe, die Anzahl der Hausaufgaben für jeden Studenten aufzulisten.

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


Inhalt

UNION (1)

- In SQL kann man die Ergebnisse von zwei Anfragen mit `UNION` verknüpfen.

  $R \cup S$ ist die Menge aller Tupel, die in R, in S, oder in beiden sind.

- `UNION` wird benötigt, da es sonst keine Möglichkeit gibt, eine Ergebnisspalte mit Werten aus mehreren Tabellenspalten (Eingabespalten) zu konstruieren.

  Dies wird z.B. benötigt, wenn Subklassen durch verschiedene Tabellen representiert werden. Z.B. könnte es eine Tabelle `STUDENTEN` und eine andere Tabelle `GASTHÖRER` geben.

- `UNION` wird auch für Fallunterscheidungen verwendet (um **if . . . then . . . else . . .** darzustellen).

UNION (2)

- Die Unteranfragen, die durch `UNION` verbunden werden, müssen Tabellen mit der gleichen Anzahl von Spalten liefern. Die Datentypen der korrespondierenden Spalten müssen kompatibel sein.

  Die Attributnamen müssen nicht übereinstimmen. Oracle und SQL Server verwenden im Ergebnis die Attributnamen der ersten Unteranfrage. DB2 verwendet ggf. künstliche Spaltennamen (1, 2, . . .).

- SQL unterscheidet zwischen
  - ◇ `UNION`: $\cup$ mit Duplikatelimination und
  - ◇ `UNION ALL`: Konkatenation (erhält Duplikate).

  Duplikatelimination ist ziemlich teuer.

UNION (3)

- Geben Sie für jeden Studenten die Gesamtpunktzahl für Hausaufgaben aus (auch für Studierende, die keine Aufgaben abgegeben haben).

```
SELECT   VORNAME, NACHNAME, SUM(PUNKTE) AS SUMME
FROM     STUDENTEN S, BEWERTUNGEN B
WHERE    S.SID = B.SID AND ATYP = 'H'
GROUP BY S.SID, VORNAME, NACHNAME
UNION ALL
SELECT   VORNAME, NACHNAME, 0 AS SUMME
FROM     STUDENTEN S
WHERE    S.SID NOT IN (SELECT SID
                       FROM   BEWERTUNGEN
                       WHERE  ATYP = 'H')
```


UNION (4)

- Erstellen Sie Noten für die Studenten basierend auf Hausaufgabe 1:

```
SELECT S.SID, S.VORNAME, S.NACHNAME, 1 NOTE
FROM   STUDENTEN S, BEWERTUNGEN B
WHERE  S.SID=B.SID AND B.ATYP='H' AND B.ANR=1
AND    B.PUNKTE >= 9
    UNION ALL
SELECT S.SID, S.VORNAME, S.NACHNAME, 2 NOTE
FROM   STUDENTEN S, BEWERTUNGEN B
WHERE  S.SID=B.SID AND B.ATYP='H' AND B.ANR=1
AND    B.PUNKTE >= 7 AND B.PUNKTE < 9
    UNION ALL
...
```

Andere Mengenop. in SQL

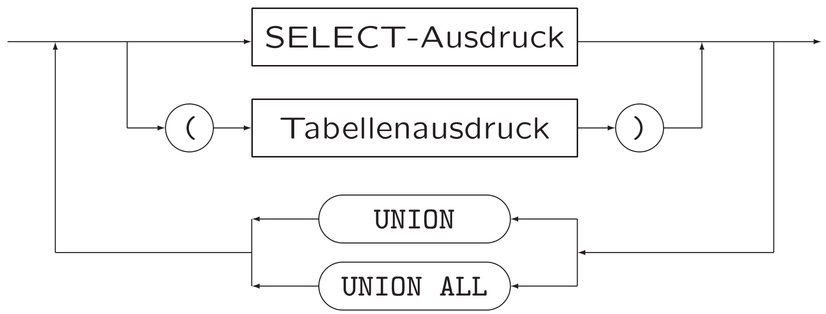
- SQL-86 enthielt nur UNION [ALL].
- Der SQL-92 Standard enthält zusätzlich EXCEPT (Mengendifferenz, $-$) und INTERSECT ($\cap$).

  SQL-86, SQL Server, Access und MySQL (ab Ver. 4) unterstützen nur UNION [ALL]. Vor Ver. 4 hatte MySQL kein UNION. DB2 bietet alle SQL-92 Mengenoperatoren. In Oracle heißt es MINUS statt EXCEPT. Für MINUS und INTERSECT wird ALL in Oracle nicht unterstützt.
- Diese Operationen tragen nichts zur Ausdruckskraft von SQL bei.

  Anfragen, die EXCEPT/MINUS und INTERSECT enthalten, können in äquivalente SQL-Anfragen ohne diese Konstrukte umgeformt werden. Anfragen die UNION enthalten, können dies im Allgemeinen nicht. Damit ist nur UNION wirklich wichtig.

UNION: Syntax

Tabellenausdruck:



- MySQL unterstützt Union nicht. SQL-86 enthält UNION und UNION ALL.
- SQL-92 und DB2 unterstützen auch INTERSECT, INTERSECT ALL, EXCEPT, und EXCEPT ALL. Oracle 8 unterstützt UNION, UNION ALL, INTERSECT und MINUS.
- In Access kann man Klammern nicht um eine ganze Anfrage setzen.

Vereinigung vs. Verbund

Übung:

- Zwei Alternativen zur Repräsentation der Bewertungen für Hausaufgaben und Klausuren sind:

Bewertungen_1

STUDENT	H	Z	E
Jim Ford	95	60	75
Ann Lloyd	80	90	95

Bewertungen_2

STUDENT	ATYP	PZT
Jim Ford	H	95
Jim Ford	Z	60
Jim Ford	E	75
Ann Lloyd	H	80
Ann Lloyd	Z	90
Ann Lloyd	E	95

- Übersetzen Sie in beiden Richtungen mittels SQL.

Bedingte Ausdrücke (1)

- Während UNION eine portabele Lösung für Fallunterscheidungen ist, kann man manchmal auch einen (effizienteren) bedingten Ausdruck verwenden.

  Bedingte Ausdrücke sind für jedes DBMS verschieden.
- Z.B. hat Oracle Ausdrücke der Form:

  DECODE(X, X_1, Y_1, X_2, Y_2, ..., Z)
- Dies wird ausgewertet, indem das DBMS zunächst X mit X_1, dann mit X_2, usw. vergleicht. Ist X_i der erste Wert mit $X = X_i$, dann wird Y_i zurückgegeben. Wenn kein X_i passt, wird Z zurückgegeben.

Bedingte Ausdrücke (2)

- Z.B.: Ausgabe der Ergebnisse von Lisa Weiss, dabei Aufgabentyp ausgeschrieben (Oracle Version):

```
SELECT   DECODE(ATYP, 'H', 'Hausaufgabe',
                      'Z', 'Zwischenklausur',
                      'E', 'Endklausur',
                           'Unbekannte Kat.'),
         ANR, PUNKTE
FROM     STUDENTEN S, BEWERTUNGEN B
WHERE    S.SID = B.SID
AND      VORNAME = 'Lisa' AND NACHNAME = 'Weiss'
ORDER BY DECODE(ATYP, 'H', 1, 'Z', 2, 'E', 3, 4)
```


Bedingte Ausdrücke (3)

- Im SQL-Standard (und z.B. DB2) schreibt man dies wie folgt:

```
SELECT CASE WHEN ATYP='H' THEN 'Hausaufgabe'
            WHEN ATYP='Z' THEN 'Zwischenklausur'
            WHEN ATYP='E' THEN 'Endklausur'
            ELSE 'Unbekannte Kat.' END,
       ANR, PUNKTE
FROM   STUDENTEN S, BEWERTUNGEN B
WHERE  S.SID = B.SID
AND    VORNAME = 'Lisa' AND NACHNAME = 'Weiss'
```

- Oracle 8i (nicht 8.0) unterstützt eine ähnliche Syntax, mit einem Komma zwischen den `WHEN`-Klauseln.

Bedingte Ausdrücke (4)

- Der SQL-92 Standard (und DB2), nicht Oracle 8i, unterstützen auch die folgende Abkürzung, die ähnlich zu Oracle's `DECODE` ist:

```
SELECT CASE ATYP WHEN 'H' THEN 'Hausaufgabe'
                 WHEN 'Z' THEN 'Zwischenklausur'
                 WHEN 'E' THEN 'Endklausur'
                 ELSE 'Unbekannte Kat.' END,
       ANR, PUNKTE
FROM   STUDENTEN S, BEWERTUNGEN B
WHERE  S.SID = B.SID
AND    VORNAME = 'Lisa' AND NACHNAME = 'Weiss'
```


Bedingte Ausdrücke (5)

- Bedingte Ausdrücke werden z.B. verwendet, um Null-Werte zu ersetzen.
- In Oracle ist `NVL(`X`, `Y`)` äquivalent zu

  `DECODE(`X`, NULL, `Y`, `X`)`

  D.h. ist X nicht Null, dann ist X das Ergebnis. Ist X Null, dann ist Y das Ergebnis.
- `COALESCE(`X`, `Y`)` ist das gleiche im SQL-92 Standard. Es ist dort die Abkürzung für

  `CASE WHEN` X `IS NOT NULL THEN` X `ELSE` Y `END`

Bedingte Ausdrücke (6)

- Z.B.: Ausgabe der E-Mail-Adressen aller Studenten, bzw. Text "keine", wenn die Spalte Null ist:

```
SELECT VORNAME, NACHNAME, NVL(EMAIL, 'keine')
FROM   STUDENTEN
```

- Bedingte Ausdrücke sind normale Terme.
- Daher kann man sie auch als Eingabe für Datentypfunktionen oder Aggregationsfunktionen verwenden.

Inhalt

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).

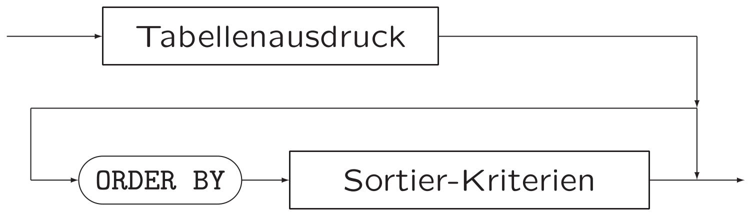
  In Oracle kann man NULLS FIRST oder NULLS LAST festlegen.

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).

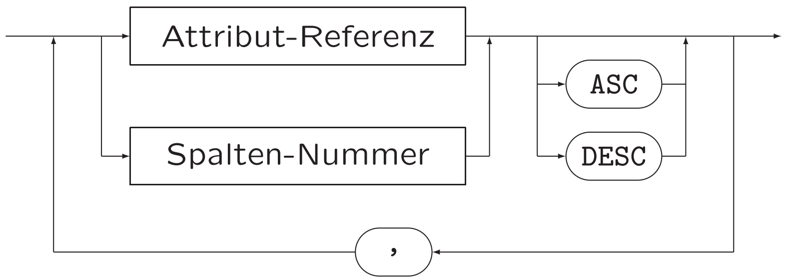
Sortieren der Ausgabe (9)

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).

Inhalt

1. Unteranfragen, Nichtmonotone Konstrukte
2. Aggregationen I: Aggregationsfunktionen
3. Aggregationen II: GROUP BY, HAVING
4. UNION, Bedingte Ausdrücke
5. Sortieren der Ausgabe: ORDER BY
6. SQL-92 Verbunde, Äußerer Verbund in Oracle

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

Verbunde in SQL-92 (1)

- Eine wichtige und nützliche Operation der relationalen Algebra ist der Verbund (mit Varianten).
- In SQL-86 kann man einen Verbund nicht direkt spezifizieren. Man verwendet das kartesische Produkt (`FROM`) und selektiert dann (`WHERE`).

  Dies ist noch immer der normale Fall.
- Natürlicher Verbund von `BEWERTUNGEN` und `AUFGABEN`:

```
SELECT B.ATYP AS ATYP, B.ANR AS ANR, SID,
       PUNKTE, THEMA, MAXPT
FROM   BEWERTUNGEN B, AUFGABEN A
WHERE  B.ATYP = A.ATYP AND B.ANR = A.ANR
```


Verbunde in SQL-92 (2)

- In SQL-92 kann man z.B. schreiben:

```
SELECT SID, ANR, (PUNKTE/MAXPT)*100
FROM   BEWERTUNGEN B NATURAL JOIN AUFGABEN A
WHERE  ATYP = 'H'
```

- Durch die Schlüsselwörter "`NATURAL JOIN`" fügt das System automatisch die Verbundbedingung hinzu:

```
B.ATYP = A.ATYP AND B.ANR = A.ANR
```

- SQL-92 erlaubt Verbunde in der `FROM`-Klausel, sowie auch auf der äußersten Anfrage-Ebene (wie `UNION`).

  Somit kann man viel im Stil der relationalen Algebra schreiben.

Verbunde in SQL-92 (3)

- Aktuelle Systeme unterstützen den Standard nur teilweise:
  - ◇ SQL-92 Verbunde können nicht in Oracle 8i verwendet werden, in Oracle 9i dagegen fast alle.
  - ◇ Einige Verbundtypen werden in DB2, SQL Server und Access unterstützt, aber der "natürliche Verbund" nicht. Man muß in diesen Systemen die Bedingung explizit aufschreiben:

```
SELECT SID, B.ANR, (PUNKTE/MAXPT)*100
FROM   BEWERTUNGEN B INNER JOIN AUFGABEN A
       ON B.ATYP = A.ATYP AND B.ANR = A.ANR
WHERE  B.ATYP = 'H'
```


Verbunde in SQL-92 (4)

- Mit der expliziten Verbundbedingung ist die Anfrage nicht kürzer als die äquivalente Anfrage in klassicsher Syntax (Verbund-Bedingung unter WHERE).
- Die Ausdruckskraft von SQL wird durch die neuen Verbund-Konstrukte nicht vergrößert.

  Jede Anfrage mit den neuen Verbund-Konstrukten kann in eine äquivalente Anfrage ohne diese Konstrukte überführt werden.
- Der Grund, warum Verbunde zu SQL hinzugefügt wurden, ist wahrscheinlich der "äußere Verbund": Hierfür ist die äquivalente Formulierung in SQL-86 deutlich länger.

Äußerer Verbund: Wdh.

- Der normale Verbund eliminiert Tupel ohne Verbundpartner:

A	B
a_1	b_1
a_2	b_2

$\bowtie$

B	C
b_2	c_2
b_3	c_3

$=$

A	B	C
a_2	b_2	c_2

- Der linke äußere Verbund stellt sicher, daß Tupel der linken Tabelle auch im Ergebnis vorhanden sind:

A	B
a_1	b_1
a_2	b_2

$⟕$

B	C
b_2	c_2
b_3	c_3

$=$

A	B	C
a_1	b_1	
a_2	b_2	c_2

Zeilen der linken Seite werden, falls notwendig, mit "Null" aufgefüllt. Es gibt auch einen rechten und einen vollen äußeren Verbund.

Äußerer Verbund in SQL (1)

- Z.B. Anzahl der Abgaben pro Hausaufgabe. Falls es keine Abgabe gibt, soll 0 ausgegeben werden:

```
SELECT   A.ANR, COUNT(SID)
FROM     AUFGABEN A LEFT OUTER JOIN BEWERTUNGEN B
         ON A.ATYP = B.ATYP AND A.ANR = B.ANR
WHERE    A.ATYP = 'H'
GROUP BY A.ANR
```

- Im Ergebnis des linken äußeren Verbunds treten alle Übungen auf. In Übungen ohne Abgaben werden die Attribute SID und PUNKTE mit Nullwerten aufgefüllt.
- COUNT(SID) zählt nur Zeilen, wo SID nicht Null ist.

Äußerer Verbund in SQL (2)

- Äquivalente Anfrage in SQL-86 (12 vs. 5 Zeilen):

```
SELECT   A.ANR, COUNT(*)
FROM     AUFGABEN A, BEWERTUNGEN B
WHERE    A.ATYP = 'H' AND B.ATYP = 'H'
AND      A.ANR = B.ANR
GROUP BY A.ANR
UNION ALL
SELECT   A.ANR, 0
FROM     AUFGABEN A
WHERE    A.ATYP = 'H'
AND      A.ANR NOT IN (SELECT B.ANR
                       FROM   BEWERTUNGEN B
                       WHERE  B.ATYP = 'H')
```


Äußerer Verbund in SQL (3)

- Geben Sie für jeden Studenten die Anzahl der abgegebenen Hausaufgaben aus (einschließlich 0).
- Die folgende Anfrage funktioniert nicht: Studenten ohne Hausaufgaben werden nicht aufgelistet.

```
SELECT VORNAME, NACHNAME, COUNT(ANR)   Falsch!
FROM   STUDENTEN S LEFT OUTER JOIN BEWERTUNGEN B
       ON S.SID = B.SID
WHERE  B.ATYP = 'H'
GROUP BY S.SID, VORNAME, NACHNAME
```

- Der äußere Verbund wird konstruiert, bevor die WHERE-Bedingung ausgewertet wird.

Äußerer Verbund in SQL (4)

- Mögliche Verbundpartner dürfen nicht nach Konstruktion des äußeren Verbundes eliminiert werden.
- Man muss die Hausaufgabenergebnisse selektieren bevor der äußere Verbund gemacht wird:

```
SELECT VORNAME, NACHNAME, COUNT(B.ANR)
FROM   STUDENTEN S LEFT OUTER JOIN
           (SELECT SID, ANR
            FROM   BEWERTUNGEN
            WHERE  ATYP = 'H') B
       ON S.SID = B.SID
GROUP BY S.SID, VORNAME, NACHNAME
```


Äußerer Verbund in SQL (5)

- Man kann auch die Bedingung für die rechte Tabelle in die Verbundbedingung integrieren:

```
SELECT VORNAME, NACHNAME, COUNT(B.ANR)
FROM   STUDENTEN S LEFT OUTER JOIN BEWERTUNGEN B
       ON S.SID = B.SID AND B.ATYP = 'H'
GROUP BY S.SID, VORNAME, NACHNAME
```

- SQL-92 lässt jede WHERE-Bedingung, die sich nur auf die rechte oder linke Tabelle bezieht, zu.
  (Das sollte aber nicht missbraucht werden.)

  Es scheint, daß DB2 und Access keine Unteranfragen in der ON-Klausel zulassen. In Access müssen komplexere Bedingungen in Klammern eingeschlossen werden.

Äußerer Verbund in SQL (6)

- Bedingungen für die linke Tabelle machen in einem linken äußeren Verbund wenig Sinn.
- Z.B. betrachte man diese Anfrage:

```
SELECT A.ATYP, A.ANR, B.SID, B.PUNKTE
FROM   AUFGABEN A LEFT OUTER JOIN BEWERTUNGEN B
       ON A.ATYP = 'H' AND B.ATYP = 'H'
          AND A.ANR = B.ANR
```

- Übung:
  Wird `A.ATYP = 'Z'` in der Ausgabe auftauchen?

  ☐ ja ☐ nein

Äußerer Verbund in SQL (7)

- Zum Teil kann `NOT EXISTS` durch einen äußeren Verbund ersetzt werden (z.B. für MySQL vor Ver. 4.1).
- Z.B. Studenten ohne eine gelöste Hausaufgabe:

```
SELECT S.SID. S.VORNAME, S.NACHNAME
FROM   STUDENTEN S LEFT OUTER JOIN BEWERTUNGEN B
       ON S.SID = B.SID AND B.ATYP = 'H'
WHERE  B.ATYP IS NULL
```

- Natürlich kann man statt `B.ATYP` jedes Attribut von `BEWERTUNGEN` auf Null testen.

  Der Test auf den Nullwert prüft, ob das aktuelle `STUDENTEN`-Tupel einen Verbundpartner gefunden hat.

Verbundsyntax in SQL-92 (1)

- SQL-92 hat folgende Verbundtypen:
  - ◇ `[INNER] JOIN`: Gewöhnlicher Verbund.
  - ◇ `LEFT [OUTER] JOIN`: Erhält Tupel der linken Tabelle.
  - ◇ `RIGHT [OUTER] JOIN`: Erhält alle Tupel von rechts.
  - ◇ `FULL [OUTER] JOIN`: Erhält alle Eingabetupel.
  - ◇ `CROSS JOIN`: Kartesisches Produkt $\times$.
  - ◇ `UNION JOIN`: Diese Vereinigung füllt die Spalten der anderen Tabelle mit Nullwerten auf.
- Schlüsselworte in [...] sind optional.

Verbundsyntax in SQL-92 (2)

- Mögliche Spezifikationen der Verbundbedingung:
  - ◇ Schlüsselwort `NATURAL` vor dem Verbundnamen.
  - ◇ "`ON` ⟨`Bedingung`⟩" folgt dem Verbund.
  - ◇ "`USING` $(A_1, \ldots, A_n)$" folgt dem Verbund.

    `USING` listet alle Verbundattribute (z.B. um den natürlichen Verbund zu spezifizieren). Attribute mit den Namen $A_1, \ldots, A_n$ müssen in beiden Tabellen auftauchen. Die Verbundbedingung ist dann $R.A_1 = S.A_1 \wedge \cdots \wedge R.A_n = S.A_n$. `NATURAL` ist äquivalent zu `USING` mit allen gleichen Attributnamen.
- Nur eines der Konstrukte kann verwendet werden.
- `CROSS JOIN` und `UNION JOIN` haben keine Verbundbedingung.

Verbundsyntax in SQL-92 (3)

- Nach dem Standard liefern der NATURAL Join und der Join mit USING eine Tabelle mit nur einer Kopie der gemeinsamen Attribute.
- Diese Attribute sind die ersten Spalten im Ergebnis. Man kann sie nicht mit Tupelvariablen ansprechen.

```
SELECT *
FROM   BEWERTUNGEN B NATURAL JOIN AUFGABEN A
```

- Die Ergebnisspalten sind ATYP, ANR, B.SID, B.PUNKTE, A.THEMA, A.MAXPT (in dieser Reihenfolge).

  Es ist unzulässig, sich auf B.ATYP oder A.ATYP zu beziehen: Es kann nur ATYP verwendet werden (das gleiche gilt für ANR).

Verbundsyntax in SQL-92 (4)

- Oracle 9i unterstützt die SQL-92 Verbunde.

  Einschließlich des Natural Join, aber ohne UNION JOIN (der in SQL:1999 entfernt wurde). Oracle 8i unterstützte keinen der SQL-92 Verbunde.

- Innerer und äußerer Verbund mit ON funktionieren auch in DB2, SQL Server, Access und MySQL.

  In Access und MySQL ist das Schlüsselwort INNER nicht optional.

- USING und NATURAL funktionieren nur in Oracle 9i.

  NATURAL existiert auch in MySQL, aber MySQL vereinigt die gleichen Attribute nicht. Das verletzt den SQL-92 Standard.

Verbundsyntax in SQL-92 (5)

- CROSS JOIN wird nur in Oracle 9i, SQL Server und MySQL, aber nicht in Access und DB2 unterstützt.

  Da man ein Komma für den CROSS JOIN schreiben kann, ist dies auch nicht sehr sinnvoll.

- UNION JOIN unterstützt keins der fünf Systeme.

  Man kann aber in SQL-92 (und z.B. Oracle, DB2, SQL Server, nicht Access) eine Unteranfrage unter FROM schreiben, die UNION oder UNION ALL enthält. Mit etwas mehr Aufwand kann also der Union Join simuliert werden. Nebenbei bemerkt, ist es etwas seltsam, daß z.B. "FROM A NATURAL JOIN B" in SQL-92 zulässig ist, aber "FROM A UNION B" nicht. Auch lässt SQL-92 die Schreibweise "FROM (SELECT * FROM A UNION SELECT * FROM B) X" zu, aber das gleiche mit "NATURAL JOIN" statt "UNION" liefert einen Syntaxfehler [Date/Darwen, 1997, S. 148].

Verbundsyntax in SQL-92 (6)

- Man kann in der FROM-Klausel auch sowohl Verbunde verwenden, als auch weitere Tupelvariablen deklarieren (getrennt durch ",").
- Das Ergebnis eines Verbundes zweier Tabellen kann mit einer dritten Tabelle verbunden werden (usw.). Die Syntax ist:

```
SELECT ...
FROM   R LEFT JOIN S ON R.A=S.B
         LEFT JOIN T ON S.C=T.D
```

- Man kann auch Klammern setzen, aber dann muss man nach (...) eine neue Tupelvariable deklarieren.

Äußerer Verbund: Oracle 8 (1)

- In Oracle wird der äußere Verbund traditionell unter `WHERE` spezifiziert (nicht länger notwendig in 9i).
- Statt der Bedingung $R.A = S.B$ schreibt man
  - $R.A = S.B$(+) für den linken äußeren Verbund
  - $R.A$(+) $= S.B$ für den rechten äußeren Verbund

  D.h. das Zeichen "(+)" wird an die Attribute angehängt, die durch Null ersetzt werden können.

  D.h. dies erhält die Tupel der anderen Tabelle (die nicht mit "(+)" markiert sind). Viele syntaktische Restriktionen sichern, daß dies wirklich ein äußerer Verbund ist. Wird der Verbund über mehrere Attribute durchgeführt, müssen alle markiert werden. Man kann auch $S.B$(+) $= c$ mit einer Konstante c schreiben, oder z.B. $R.A = S.B$(+)+1.

Äußerer Verbund: Oracle 8 (2)

- Z.B. Anzahl der Abgaben pro HA (kann 0 sein):

```
SELECT   A.ATYP, A.ANR, COUNT(SID)
FROM     AUFGABEN A, BEWERTUNGEN B
WHERE    A.ATYP = B.ATYP(+) AND A.ANR = B.ANR(+)
GROUP BY A.ATYP, A.ANR
```

- Wie im äußeren Verbund von SQL-92, wird der äußere Verbund durchgeführt, bevor irgendeine andere Bedingung der `WHERE`-Klausel angewandt wird.

  Egal in welcher Reihenfolge die Bedingungen stehen. Aber wie oben gezeigt, kann man eine Unteranfrage unter `FROM` machen, um vor dem äußeren Verbund zu selektieren.