Teil 8: 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 und erklären.

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

Inhalt

Beispieldatenbank (erneut)

STUDENTEN

SID	VORNAME	NAME	EMAIL
101	Ann	Smith	···
102	Michael	Jones	(null)
103	Richard	Turner	···
104	Maria	Brown	···

AUFGABEN

KAT	ANR	THEMA	MAXPT
H	1	Rel. Algeb.	10
H	2	SQL	10
Z	1	SQL	14

RESULTATE

SID	KAT	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

Nichtmonotones Verhalten (1)

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

  Werden weitere Zeilen eingefügt, so erhält man mindestens die gleiche Antwort wie zuvor, und eventuell mehr.

- 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 Maria Brown eine korrekte Antwort. Würde man jedoch ein Ergebnis 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 indizieren Formulierungen wie "es gibt kein" oder "existiert nicht" nichtmonotones Verhalten.

- Auch "für alle" oder "minimale/maximale" indizieren nichtmonotones Verhalten: Hierbei darf 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 existiert, 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   RESULTATE
                   WHERE  KAT = 'H')
```

FIRST	LAST
Maria	Brown

NOT IN (2)

- Zumindest konzeptionell wird die Unteranfrage vor Beginn der Ausführung der Hauptanfrage ausgewertet:

STUDENTEN

SID	VORNAME	NACHNAME	EMAIL
101	Ann	Smith	···
102	Michael	Jones	(null)
103	Richard	Turner	···
104	Maria	Brown	···

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 auch DISTINCT in einer Unteranfrage verwenden:

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

- Dies ist lokal äquivalent. Der Effekt 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, und dass die Verwendung von DISTINCT den Effekt haben kann, dass der Optimierer gewisse Auswertungsstrategien, die das Ergebnis der Unteranfrage nicht verwirklichen, nicht bedenkt.

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 ("Namen") der Hausaufgaben, die von mindestens einem Studenten gelöst wurden:

```
SELECT THEMA
FROM   AUFGABEN
WHERE  KAT='H' AND ANR IN (SELECT ANR
                           FROM   RESULTATE
                           WHERE  KAT='H')
```

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

```
SELECT DISTINCT THEMA
FROM   AUFGABEN A, RESULTATE R
WHERE  A.KAT='H' AND A.ANR=R.ANR AND R.KAT='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), und nicht eine gewöhnliche Relation, ist.

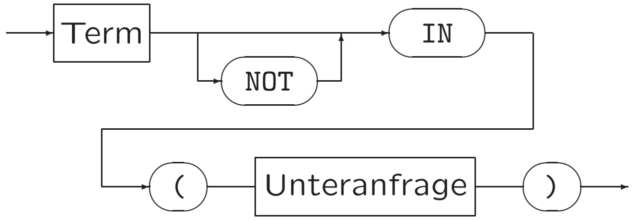
- In SQL-92 wurden Vergleiche auf das Tupel-Level erweitert, so dass man z.B. auch schreiben kann

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

  Das ist aber nicht übertragbar. SQL Server und Access unterstützen es nicht (MySQL erlaubt gar keine Unteranfragen, s.u.). Eine EXISTS Unteranfrage (s.u.) wäre besser, wenn man mehr als eine Spalte vergleichen muss. Oracle und DB2 erlauben IN mit mehreren Spalten.

NOT IN (7)

Atomare Formel (Form 6):



- Die Unteranfrage muss eine Tabelle mit einer einzelnen Spalte ergeben (eine Menge).
- 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 keine Unteranfragen.
- Die Spaltennamen links und rechts von `IN` müssen nicht übereinstimmen, aber die Datentypen müssen kompatibel sein.

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 erklärt werden, verwendet werden.

  Dies ist auch bei Unteranfragen mit `IN` möglich, aber es ist dort eine unnötige und unerwartete Komplikation (schlechter Stil).
- Das bedeutet, dass die Unteranfrage einmal für jeden Wert der zugeordneten Tupelvariablen der äußeren Anfrage ausgewertet werden muss.

  Die Unteranfrage kann als parametrisiert angesehen werden.

NOT EXISTS (2)

- Studenten ohne eine abgegebene Hausaufgabe:

```
SELECT VORNAME, NACHNAME
FROM   STUDENTEN S
WHERE  NOT EXISTS (SELECT * FROM RESULTATE R
                   WHERE  R.KAT = 'H'
                   AND    R.SID = S.SID)
```

- Die Tupelvariable `S` läuft über die vier Reihen in `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	Ann	Smith	···

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

```
SELECT * FROM RESULTATE R
WHERE  R.KAT = 'H'
AND    R.SID = 101
```

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

- Das Ergebnis ist nicht leer. Somit ist die `NOT EXISTS`-Bedingung in der äußeren Anfrage für dieses Tupel `S` nicht erfüllt.

NOT EXISTS (4)

- Das gleiche passiert bei der zweiten Zeile in `STUDENTEN`. Die Unteranfrage wird für `S.SID`=102 ausgeführt:

```
SELECT * FROM RESULTATE R
WHERE  R.KAT = 'H'
AND    R.SID = 102
```

SID	KAT	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	Maria	Brown	···

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

```
SELECT * FROM RESULTATE R
WHERE  R.KAT = 'H'
AND    R.SID = 104
```

keine Zeilen ausgewählt

- Somit ist die `NOT EXISTS`-Bedingung der Hauptanfrage für dieses Tupel `S` erfüllt. Maria Brown 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, R.ANR   Falsch!
FROM   STUDENTEN S
WHERE  NOT EXISTS (SELECT * FROM RESULTATE R
                   WHERE  R.KAT = 'H'
                   AND    R.SID = S.SID)
```

- Man kann dies wie globale und lokale Variablen betrachten: Variablen, die in der äußeren Anfrage definiert sind, gelten für die gesamte Anfrage. In der Unteranfrage definierte Variablen gelten nur dort.

  Das entspricht der Blockstruktur z.B. von Pascal.

NOT EXISTS (7)

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

  Korrelierte Unteranfragen kann man verstehen, als parametrisiert mit Tupeln der äußeren Anfrage. Man kann dies optimieren, aber konzeptionell werden sie einmal, für jede Zuweisung von Tupeln zu Tupeln 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 Unteranfragen mit `NOT EXISTS` sind fast immer falsch (aber mit `IN` sind sie ok):

```
SELECT VORNAME, NACHNAME   Falsch!
FROM   STUDENTEN S
WHERE  NOT EXISTS (SELECT * FROM RESULTATE R
                   WHERE  KAT = 'H')
```

Hier wurde die Verbundbedingung in der Unteranfrage vergessen. Die Unteranfrage wurde somit zu einer unkorrelierten Unteranfrage.

- Wenn es mindestens einen HA-Eintrag in `RESULTATE` gibt, egal für welchen Studenten, wird das `NOT EXISTS` stets falsch und das Anfrageergebnis leer sein.

NOT EXISTS (9)

- Bisher musste sich eine Attributreferenz ohne Tupelvariable ("unqualifizierter Attributname") auf eine eindeutige Tupelvariable beziehen.
- Bei Unteranfragen fordert SQL nur, dass es eine eindeutig 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 RESULTATE R
                   WHERE  KAT = '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.
- Diese Regel hilft, dass unkorrelierte Unteranfragen unabhängig entwickelt und ohne Veränderungen in andere Anfragen eingefügt werden können.

NOT EXISTS (11)

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

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

- Alle Referenzen auf `X` in der Unteranfrage meinen `RESULTATE X`. Die Variable der äußeren Anfrage wird überschattet. 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 sagen, dass in einigen Systemen SELECT null oder SELECT 1 schneller als SELECT * ist.

  "SELECT null" wird von Oracle's Programierern verwendet (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):

- Eine Unteranfrage ist ein Ausdruck der Form SELECT ...FROM ...[WHERE ...][GROUP BY ...][HAVING ...].

  [...] bedeutet, dass diese Teile optional sind. 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 werden in Klammern (...) eingeschlossen.

NOT EXISTS (14)

- Man kann auch EXISTS ohne Negation verwenden.
- Wer hat mindestens eine Hausaufgabe gelöst?

```
SELECT SID, VORNAME, NACHNAME
FROM   STUDENTEN S
WHERE  EXISTS (SELECT * FROM RESULTATE R
               WHERE  R.SID = S.SID
               AND    R.KAT = 'H')
```

- Die gleiche Anfrage kann mit einem gewöhnlichen Verbund gestellt werden:

```
SELECT DISTINCT S.SID, VORNAME, NACHNAME
FROM   STUDENTEN S, RESULTATE R
WHERE  S.SID = R.SID AND R.KAT = 'H'
```


"For all" (1)

- Wer hat das beste Ergebnis für Hausaufgabe 1?

```
SELECT VORNAME, NACHNAME, PUNKTE
FROM   STUDENTEN S, RESULTATE X
WHERE  S.SID = X.SID
AND    X.KAT = 'H' AND X.ANR = 1
AND    NOT EXISTS
       (SELECT * FROM RESULTATE Y
        WHERE  Y.KAT = 'H' AND Y.ANR = 1
        AND    Y.PUNKTE > X.PUNKTE)
```

- D.h. man sucht ein Ergebnis X für Hausaufgabe 1, für das es kein Ergebnis Y (auch für Hausaufgabe 1) gibt, mit mehr Punkten als X.

"For all" (2)

- In der math. Logik gibt es zwei Quantoren:
  - $\exists s X: F$: Es gibt ein X, sodass F gilt. (Existenzquantor)
  - $\forall s X: F$: Für alle X ist F wahr. (Universalquantor)
- Im relationalen Tupelkalkül wird die maximale Anzahl der Punkte für Hausaufgabe 1 z.B. wie folgt ausgedrückt:

$$\{\texttt{X.PUNKTE [RESULTATE X]} \mid \texttt{X.KAT} = \texttt{'H'} \land \texttt{X.ANR} = 1 \land \forall\, \texttt{RESULTATE Y:}\ \texttt{Y.KAT} = \texttt{'H'} \land \texttt{Y.ANR} = 1 \rightarrow \texttt{Y.PUNKTE} \leq \texttt{X.PUNKTE})\}$$

"For all" (3)

- 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 nur den Existenzquantor ("`EXISTS`"), keinen Universalquantor.

  Aber man betrachte "`>= ALL`" unten.
- Das ist kein Problem, da $\forall s\, X: F$ 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 obige Muster ist äquivalent zu $\neg\exists s\, X: F_1 \land \neg F_2$.

"For all" (4)

- Das obige Beispiel ist logisch äquivalent zu:

$$\{\texttt{X.PUNKTE [RESULTATE X]} \mid \texttt{X.KAT} = \texttt{'H'} \land \texttt{X.ANR} = 1 \land \neg\exists\, \texttt{RESULTATE Y:}\ \texttt{Y.KAT} = \texttt{'H'} \land \texttt{Y.ANR} = 1 \land \texttt{Y.PUNKTE} > \texttt{X.PUNKTE})\}$$

- In SQL schreibt man dies wie folgt:

```
SELECT X.PUNKTE
FROM   RESULTATE X
WHERE  X.KAT = 'H' AND X.ANR = 1
AND    NOT EXISTS
       (SELECT * FROM RESULTATE Y
        WHERE  Y.KAT = 'H' AND Y.ANR = 1
        AND    Y.PUNKTE > X.PUNKTE)
```


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  KAT = 'H'
        AND    NOT EXISTS
               (SELECT * FROM RESULTATE R
                WHERE  R.SID = S.SID
                AND    R.ANR = A.ANR
                AND    R.KAT = 'H'))
```

Häufige Fehler (1)

Übungen:

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

```
SELECT DISTINCT S.SID, VORNAME, NACHNAME
FROM   STUDENTEN S, RESULTATE R
WHERE  S.SID <> R.SID AND R.KAT = 'H'
```

- Findet diese Anfrage Übungen, die noch nicht gelöst wurden?

```
SELECT DISTINCT A.KAT, A.ANR
FROM   AUFGABEN A, RESULTATE R
WHERE  A.KAT = R.KAT AND A.ANR = R.ANR
AND    R.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 die 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)

- Berechnet diese Anfrage den Studenten mit dem besten Ergebnis für Hausaufgabe 1?

```
SELECT DISTINCT S.VORNAME, S.NACHNAME, X.PUNKTE
FROM   STUDENTEN S, RESULTATE X, RESULTATE Y
WHERE  S.SID = X.SID
AND    X.KAT = 'H' AND X.ANR = 1
AND    Y.KAT = '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            Falsch!
       (SELECT *
        FROM   RESULTATE R, STUDENTEN S
        WHERE  R.SID = S.SID
        AND    R.KAT = 'H' AND R.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?
  Die Aufgabe ist, alle Studenten aufzulisten, 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, RESULTATE R
WHERE  S.SID = R.SID
AND    NOT EXISTS (SELECT *
                   FROM   RESULTATE R
                   WHERE  S.SID = R.SID)
```


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 Mengenelemente (ALL) oder für mindestens ein Element (ANY) wahr ist:

```
SELECT S.VORNAME, S.NACHNAME, X.PUNKTE
FROM   STUDENTEN S, RESULTATE X
WHERE  S.SID=X.SID AND X.KAT='H' AND X.ANR=1
AND    X.PUNKTE >= ALL (SELECT Y.PUNKTE
                        FROM   RESULTATE Y
                        WHERE  Y.KAT = 'H'
                        AND    Y.ANR = 1)
```


ALL, ANY, SOME (2)

- Folgendes ist logisch äquivalent zu obiger Anfrage:

```
SELECT S.VORNAME, S.NACHNAME, X.PUNKTE
FROM   STUDENTEN S, RESULTATE X
WHERE  S.SID=X.SID AND X.KAT='H' AND X.ANR=1
AND    NOT X.PUNKTE < ANY (SELECT Y.PUNKTE
                           FROM   RESULTATE Y
                           WHERE  Y.KAT = 'H'
                           AND    Y.ANR = 1)
```

- Auch hier kann "for all" durch "not exists not" ersetzt werden.

  Natürlich ist auch "exists" äquivalent zu "not for all not".

ALL, ANY, SOME (3)

- Dieses Konstrukt ist nicht zwingend erforderlich, da

  T_1 `< ANY (SELECT` T_2 `FROM ... WHERE ...)`

  äquivalent ist zu

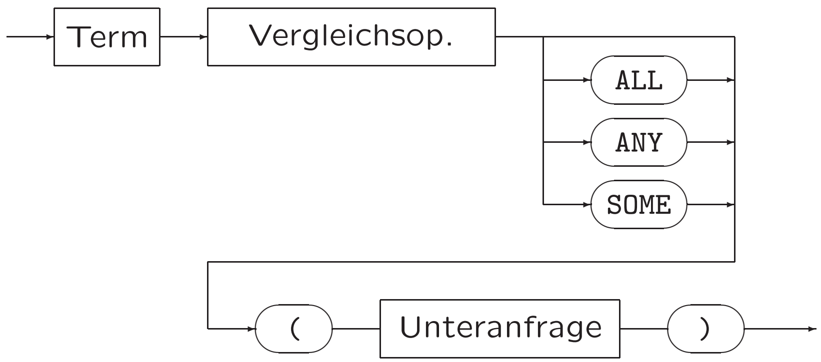
  `EXISTS (SELECT * FROM ... WHERE ... AND` T_1 `<` T_2`)`

  Dies erfordert, dass sich T_1 explizit auf eine Tupelvariable bezieht, die nicht in der Unteranfrage überschrieben wird (so dass die Bedeutung von T_1 nicht verändert wird, wenn sie in die Unteranfrage geschoben wird).

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

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.

- Tritt keins der Stichworte `ALL`, `ANY`, `SOME` auf, darf die Unteranfrage höchstens eine Ergebniszeile haben.

  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, RESULTATE R
WHERE  S.SID=R.SID AND R.KAT='H' AND R.ANR=1
AND    R.PUNKTE = (SELECT MAXPT
                   FROM   AUFGABEN
                   WHERE  KAT='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, 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 ...) = R.PUNKTE
```

- In Oracle8 und SQL86 muss die Unteranfrage auf der rechten Seite stehen.
- Man kann mit dem Ergebnis einer Unteranfrage auch weitere Berechnungen durchführen, z.B. (nicht in SQL86, Oracle8):

```
R.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   RESULTATE R
        WHERE  R.SID = S.SID
        AND    R.KAT = 'H' AND R.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 verschachtelte Aggregationen benötigt, sieher unten.
- In folgendem Beispiel wird der Verbund von `RESULTATE` und `AUFGABEN` in einer Unteranfrage berechnet (die aus einer Sichtdefinition resultieren könnte, s.u.):

```
SELECT X.SID, ROUND(X.PUNKTE*100/X.MAXPT) AS PZT
FROM   (SELECT A.KAT, A.ANR, R.SID, R.PUNKTE,
               A.MAXPT
        FROM   AUFGABEN A, RESULTATE R
        WHERE  A.KAT=R.KAT AND A.ANR=R.ANR) X
WHERE  X.KAT = '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 und SQL Server (aber nicht Oracle8 und Access) lassen folgende Umbenennung von Spalten zu:

```
FROM (...) X(KATEGORIE, 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 übertragbare 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 R.ANR, R.PUNKTE
                     FROM   RESULTATE R   Falsch!
                     WHERE  R.KAT = 'H'
                     AND    R.SID = S.SID) X
```

- Außerdem sollten Unteranfragen unter FROM nur verwendet werden, wenn es wirklich notwendig ist. Dadurch werden Anfragen 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, RESULTATE R
    WHERE    S.SID=R.SID AND KAT = 'H'
```

- Sichten können in Anfragen wie gespeicherte Tabellen verwendet werden:

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

- 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

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

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)

- Einige Aggregationsfunktionen reagieren sensibel auf Duplikate (z.B. Sum), andere nicht (z.B. Minimum).

  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 $\cong$ Menge), oder nicht (Eingabe $\cong$ 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)` sind meist falsch. Einige Studenten verwechseln `SUM` und `COUNT`.

Einfache Aggregationen (1)

- Zunächst werden Aggregationen über alle Ergebniszeilen einer Anfrage erklärt.

  Aggregationen über gewisse Gruppen von Anfragen werden im nächsten Abschnitt behandelt.

- 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   RESULTATE
WHERE  KAT = 'H' AND ANR = 1
```

MAX(PUNKTE)
10

Einfache Aggregationen (2)

- Wieviele Studenten haben eine Hausaufgabe gelöst?

```
SELECT COUNT(DISTINCT SID)
FROM   RESULTATE
WHERE  KAT = 'H'
```

COUNT(DISTINCT SID)
3

- Wieviele Punkte hat Student 101 insgesamt für Hausaufgaben bekommen?

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

Gesamtpunkte
18

Einfache Aggregationen (3)

- Wieviele Punkte der Maximalpunktzahl haben die Studenten für HA 1 durchschnittlich bekommen?

```
SELECT AVG((R.PUNKTE/A.MAXPT)*100)
FROM   RESULTATE R, AUFGABEN A
WHERE  R.KAT = 'H' AND A.KAT = 'H'
AND    R.ANR = 1 AND A.ANR = 1
```

- Z.B. Hausaufgabenpunkte von Student 1 plus 3 Extrapunkte:

```
SELECT SUM(PUNKTE) + 3 "Gesamte HA-Punkte"
FROM   RESULTATE
WHERE  SID = 101 AND KAT = '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   RESULTATE
WHERE  KAT = '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 unter SELECT, aber ohne GROUP BY (genannt "einfache Aggregationen", siehe oben):
    Das Ergebnis ist immer genau eine Zeile.
  - ◇ 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 Ausgangstabellen konstruiert (kartesisches Produkt, verschachtelte Schleifen).
- Als zweites wird die WHERE-Klausel ausgewertet.

  Nur die Kombinationen, die die Bedingung erfüllen, werden weiter betrachtet (Selektion, Filter). Natürlich kann man in realen Systemen den ersten und zweiten Schritt für eine effizientere Auswertung kombinieren.
- Als drittes, wenn es keine Aggregation, GROUP BY, und HAVING gibt, wird die SELECT-Klausel ausgewertet, indem man die Werte der Terme in der SELECT-Liste für die restlichen Tupelkombinationen ausgibt.

Auswertung (2)

- Enthält die SELECT-Liste einen Aggregationsterm, es gibt aber kein GROUP BY, so wird nur eine einzelne Ausgabezeile berechnet.
- Anstatt die Spaltenwerte wie gewohnt auszugeben, werden die Werte einer Menge/Multimenge zugefügt, die Eingabe für die Aggregationsfunktion ist.

  Enthält die SELECT-Liste mehrere Aggregationen, müssen mehrere solcher Mengen verwaltet werden.
- Ohne DISTINCT können die aggregierten Werte schrittweise ohne explizites Speichern einer zeitweilige Menge berechnet werden (vgl. nächste Folie).

Auswertung (3)

- Z.B. betrachte man die Anfrage:

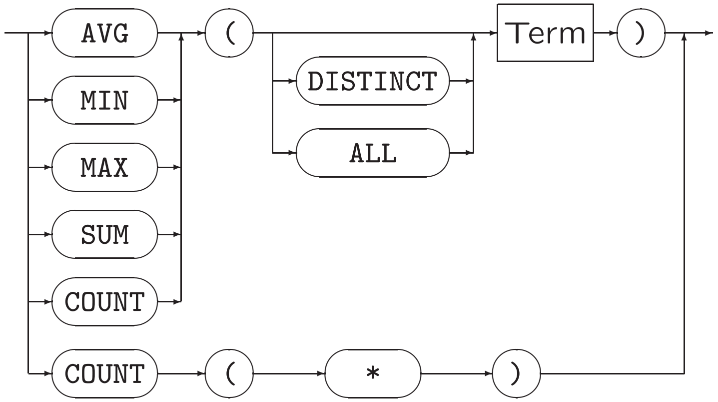
```
SELECT SUM(MAXPT), COUNT(*)
FROM   AUFGABEN A
WHERE  KAT = 'H'
```

- Dies wird so ausgewertet:

```
out1 = 0; out2 = 0;
foreach row A in AUFGABEN do
    if A.KAT = 'H' then begin
        out1 = out1 + A.MAXPT;
        out2 = out2 + 1;
    end;
print out1, out2;
```

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)

- Aggregationen dürfen nicht in der `WHERE`-Klausel verwendet werden.

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

  `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 KAT, ANR, AVG(PUNKTE)   Falsch!
FROM   RESULTATE
```


Syntax / Restriktionen (3)

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

```
SELECT SID
FROM   RESULTATE
WHERE  KAT = '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

- Gewöhnlich werden Nullwerte herausgefiltert, bevor die Aggregationsfunktion angewendet wird.
- Nur `COUNT(*)` beinhaltet Nullwerte (da es Reihen 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 Attribute, wie in `COUNT(DISTINCT KAT)` 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 ist zumindest für `SUM` zählerintuitiv. 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

GROUP BY (1)

- Die obigen SQL-Konstrukte können nur eine einzelne aggregierte Ausgabezeile erzeugen.
- Mit der `GROUP BY` Klausel kann man über Gruppen statt über alle Tupel, aggregieren.
- Berechnen Sie die durchschnittliche Punktzahl für jede Hausaufgabe:

```
SELECT   ANR, AVG(PUNKTE)
FROM     RESULTATE
WHERE    KAT = 'H'
GROUP BY ANR
```

ANR	AVG(PUNKTE)
1	8
2	8.5

GROUP BY (2)

- Die `GROUP BY`-Klausel spaltet die Ergebnistabelle nach Auswertung von `FROM` und `WHERE` in Gruppen, die den gleichen Wert in den `GROUP BY`-Spalten haben.

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

- Die Aggregation wird dann auf jede Gruppe angewendet, und man erhält eine Zeile für jede 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).

- Auf der anderen Seite ergeben einfache Aggregationen (ohne `GROUP BY`) immer genau eine Ausgabezeile, und die Engabemenge kann auch leer sein (dann kann `COUNT(*)` 0 sein).

  Eine `GROUP BY`-Anfrage kann keine, eine oder mehrere Ausgabezeilen ergeben.

GROUP BY (4)

- Da `GROUP BY`-Attribute einen eindeutigen Wert für jede Gruppe haben, können sie unter `SELECT` stehen.

  Andere Attribute können unter `SELECT` nur in Aggregationen stehen.

- Z.B. folgendes ist unzulässig:

```
SELECT   A.ANR, A.THEMA, AVG(R.PUNKTE)   Falsch!
FROM     AUFGABEN A, RESULTATE R
WHERE    A.KAT='H' AND R.KAT='H' AND A.ANR=R.ANR
GROUP BY A.ANR
```

  `A.THEMA` taucht nicht unter `GROUP BY` auf, deshalb kann es nicht in der `SELECT`-Liste außerhalb von Aggregationsfunktionen verwendet werden. Das ist besonders seltsam, da `ANR` ein Schlüssel von `AUFGABEN` ist, so dass `THEMA` eigentlich eindeutig innerhalb der Gruppen ist. Aber die SQL-Regel ist rein syntaktisch.

GROUP BY (5)

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

```
SELECT   A.ANR, A.THEMA, AVG(R.PUNKTE)
FROM     AUFGABEN A, RESULTATE R
WHERE    A.KAT='H' AND R.KAT='H' AND A.ANR=R.ANR
GROUP BY A.ANR, A.THEMA
```

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

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

GROUP BY (6)

- Übung: Gibt es irgendeinen semantischen Unterschied zwischen

```
SELECT   THEMA, AVG(PUNKTE/MAXPT)
FROM     AUFGABEN A, RESULTATE R
WHERE    A.KAT='H' AND R.KAT='H' AND A.ANR=R.ANR
GROUP BY THEMA
```

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

```
SELECT   THEMA, AVG(PUNKTE/MAXPT)
FROM     AUFGABEN A, RESULTATE R
WHERE    A.KAT='H' AND R.KAT='H' AND A.ANR=R.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)*10+0.5) PROZENTE,
         COUNT(*)
FROM     AUFGABEN A, RESULTATE R
WHERE    A.KAT = R.KAT AND A.ANR = R.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. Übertragbare Alternative: Unteranfrage unter `FROM` oder Verwendung einer Sicht.

GROUP BY (8)

- Die Reihenfolge der Attribute in der `GROUP BY`-Klausel ist nicht wichtig.

  `GROUP BY A, B` bedeutet, dass zwei Tupel t, u in die gleiche Gruppe gehören, falls $t.\mathtt{A} = u.\mathtt{A}$ und $t.\mathtt{B} = u.\mathtt{B}$.
  `GROUP BY B, A` bedeutet, dass zwei Tupel t, u in die gleiche Gruppe gehören, falls $t.\mathtt{B} = u.\mathtt{B}$ und $t.\mathtt{A} = u.\mathtt{A}$.

- Beachte, dass es sinnlos ist, 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 einzelne Gruppe geben kann.

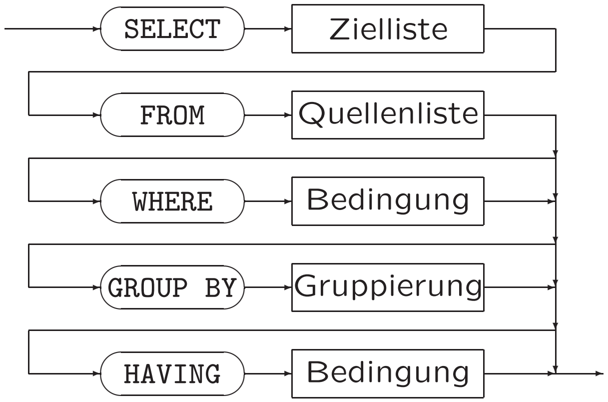
GROUP BY (9)

Warnung:

- Viele Studenten verwechseln “`GROUP BY`” und “`ORDER BY`”:
  - ◇ `GROUP BY` ist wichtig für das Anfrageergebnis.
  - ◇ `ORDER BY` ist nur kosmetisch (schöne Ausgabe).
- `GROUP BY` sortiert normalerweise intern die Tupel (so dass Tupel mit gleichem Wert benachbart sind).
- Aber dann macht `GROUP BY` die Gruppierung, während die Sortierung von `ORDER BY` am Ende geschieht.
- Manchmal kann das DBMS das `GROUP BY` effizienter ohne Sortierung auswerten.

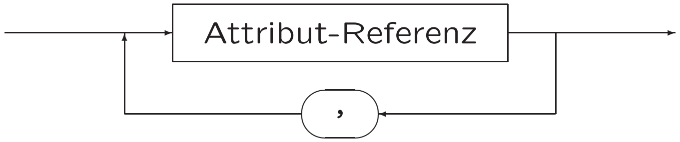
Syntax (1)

SELECT-Ausdruck:



Syntax (2)

Gruppierung:



- Z.B. `GROUP BY TITEL, K.KNR`

- Oracle, SQL Server, DB2, Access und MySQL unterstützen den allgemeinen "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 hat SQL eine zweite Bedingung, die `HAVING`-Klausel. Das Ziel der `HAVING`-Klausel ist, ganze Gruppen zu eliminieren.
- Unter `HAVING` kann man Aggregationsoperatoren verwenden. Aber wie unter `SELECT` dürfen außerhalb der Aggregationen nur `GROUP BY`-Attribute stehen.

HAVING (2)

- Wer hat mindestens 18 Hausaufgabenpunkte?

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

VORNAME	NACHNAME
Ann	Smith
Michael	Jones

- Die `WHERE`-Bedingung bezieht sich auf eine einzelne Tupelkombination, die `HAVING`-Bedingung auf ganze Gruppen.

Auswertung

1. Alle Kombinationen von Zeilen von Tabellen unter FROM werden betrachtet.
2. Die WHERE-Bedingung filtert eine Teilmenge heraus.
3. Die verbleibenden verbundenen Tupel werden in Gruppen, mit gleichen Werten in den GROUP BY-Attributen, aufgespalten.
4. Gruppen von Tupeln, 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 außerhalb von Aggregationen nur GROUP BY-Attribute unter SELECT und HAVING genutzt werden.

  Innerhab von Aggregationsfunktionen, d.h. als ihre Argumente, können alle Attribute verwendet werden. Z.B. AVG(A)/B: Das Attribut A steht 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

- Normalerweise legen die Restriktionen eindeutig fest, ob eine Bedingung unter WHERE oder unter HAVING stehen muss.

  Nur wenn eine Bedingung nur 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, RESULTATE R
GROUP BY S.SID, R.SID, VORNAME, NACHNAME
HAVING   S.SID = R.SID AND SUM(PUNKTE) >= 18
```


Aggregationsunteranfragen (1)

- Wer hat das beste Ergebnis für Hausaufgabe 1?

```
SELECT S.VORNAME, S.NACHNAME, R.PUNKTE
FROM   STUDENTEN S, RESULTATE R
WHERE  S.SID=R.SID AND R.KAT='H' AND R.ANR=1
AND    R.PUNKTE = (SELECT MAX(PUNKTE)
                   FROM   RESULTATE
                   WHERE  KAT='H' AND ANR=1)
```

- Für eine Aggregationsanfrage ohne GROUP BY ist es sicher, dass man genau eine Zeile erhält. Somit sind ANY/ALL hier nicht erforderlich.

Aggregationsunteranfragen (2)

- Da man in SQL92, DB2, SQL Server und Access eine Unteranfrage, die ein Datenelement zurückgibt, wie einen Term verwenden kann, sind Unteranfragen unter `SELECT` erlaubt.

  Oracle 8.0 unterstützt dies 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   RESULTATE R
                           WHERE  R.SID = S.SID
                           AND    R.KAT = 'H')
FROM   STUDENTEN S
```


Verschachtelte Aggregationen (1)

- Verschachtelte Aggregationen erfordern eine Unteranfrage unter `FROM`.
- Was ist die durchschnittliche Anzahl der HA-Punkte? (es zählen nur Studenten, die HA gelöst haben)

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

X

SID	HA_PKT
101	18
102	18
103	5

AVG(X.HA_PKT)
13.67

Verschachtelte Aggregationen (2)

- Oracle unterstützt auch auf diese Weise geschriebene, verschachtelte Aggregationen:

```
SELECT   AVG(SUM(PUNKTE))    Nur Oracle!
FROM     RESULTATE
WHERE    KAT = 'H'
GROUP BY SID
```

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.

Aggregating Different Sets (1)

- Durch Unteranfragen unter `FROM` kann man über verschiedene Mengen aggregieren:

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

Aggregating Different Sets (2)

- Das ist auch mit konditionalen Ausdrücken möglich z.B. in Oracle:

```
SELECT VORNAME, NACHNAME,
       SUM(DECODE(R.KAT, 'H', R.PUNKTE, 0)) HA
       SUM(DECODE(R.KAT, 'Z', R.PUNKTE, 0)) ZP
FROM   STUDENTEN S, RESULTATE R
WHERE  S.SID = R.SID
```

- Z.B. der konditionale Ausdruck

```
DECODE(R.KAT, 'H', R.PUNKTE, 0)
```

gibt `R.PUNKTE` zurück, falls `R.KAT = '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, RESULTATE R
WHERE    S.SID = R.SID AND R.KAT = 'H'
GROUP BY S.SID, VORNAME, NACHNAME
HAVING   SUM(PUNKTE) >= ALL(SELECT   SUM(PUNKTE)
                            FROM     RESULTATE
                            GROUP BY SID)
```

- Eine alternative Lösung mit einer Sicht wird auf der nächsten Folie gezeigt.

Aggregationen maximieren (2)

- Gesamtpunktzahl der HA für jeden Studenten:

```
CREATE VIEW HA_SUMMEN AS
   SELECT   SID, SUM(PUNKTE) AS SUMME
   FROM     RESULTATE
   WHERE    KAT = '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)

- Was denken Sie über diese Anfrage? Die Aufgabe ist, alle Studenten aufzulisten, die mindestens zwei Hausaufgaben gelöst haben.

```
SELECT VORNAME, NACHNAME
FROM   STUDENTEN S
WHERE  2 <= (SELECT COUNT(S.SID)
             FROM   RESULTATE R
             WHERE  R.SID = S.SID
             AND    R.KAT = 'H')
```

Übung: Mögliche Fehler (2)

- Und was über diese Anfrage? Wieder ist die Aufgabe, alle Studenten mit mindestens zwei gelösten Hausaufgaben aufzulisten.

```
SELECT VORNAME, NACHNAME
FROM   STUDENTEN S, RESULTATE R
WHERE  S.SID = R.SID
AND    R.KAT = 'H'
AND    COUNT(R.ANR) >= 2
```


Übung: Mögliche Fehler (3)

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

```
SELECT   S.SID, S.VORNAME, S.NACHNAME, SUM(R.ANR)
FROM     STUDENTEN S, RESULTATE R
WHERE    S.SID = R.SID
AND      R.KAT = 'H'
GROUP BY S.SID, S.VORNAME, S.NACHNAME, R.ANR
```


Inhalt

UNION (1)

- In SQL kann man die Ergebnisse von zwei Anfragen durch `UNION` verbinden.

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

- `UNION` wird benötigt, da es sonst keinen Weg gibt, eine Ergebnisspalte mit Werten verschiedener Tabellen/Spalten 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` ist auch für Fallunterscheidung sehr sinnvoll (um ein **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 in dem Fall künstliche Spaltennamen (1, 2, . . .).

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

  Duplikatelimination ist ziemlich teuer.


```
SELECT   VORNAME, NACHNAME, SUM(PUNKTE) AS SUMME
FROM     STUDENTEN S, RESULTATE R
WHERE    S.SID = R.SID AND KAT = '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   RESULTATE
                       WHERE  KAT = 'H')
```


UNION (3)

- Geben Sie für jeden Studenten ihre/seine Gesamtpunktzahl der Hausaufgabenpunkte aus (0 bei keiner gelösten Aufgabe).

UNION (4)

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

```
SELECT S.SID, S.VORNAME, S.NACHNAME, 'A' NOTE
FROM   STUDENTEN S, RESULTS R
WHERE  S.SID=R.SID AND R.KAT='H' AND R.ANR=1
AND    R.PUNKTE >= 9
    UNION ALL
SELECT S.SID, S.VORNAME, S.NACHNAME, 'B' NOTE
FROM   STUDENTEN S, RESULTATE R
WHERE  S.SID=R.SID AND R.KAT='H' AND R.ANR=1
AND    R.PUNKTE >= 7 AND R.PUNKTE < 9
    UNION ALL
...
```

Andere Mengenop. in SQL

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

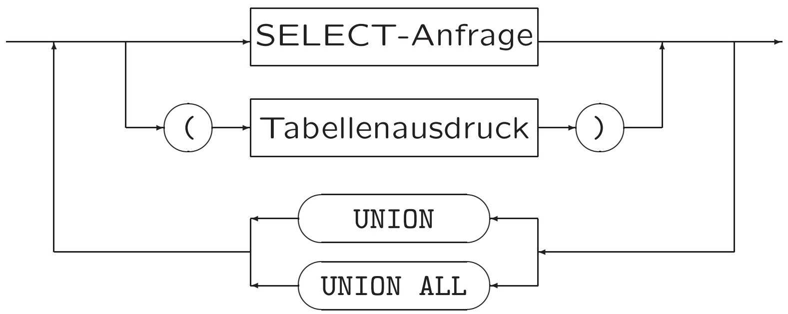
  SQL-86, SQL Server und Access unterstützen nur UNION [ALL]. MySQL unterstützt keinen der Operatoren. DB2 unterstützt alle SQL-92 Mengenoperatoren. In Oracle 8.0, wird MINUS statt EXCEPT für − verwendet. Für MINUS und INTERSECT wird ALL in Oracle nicht unterstützt.

- Diese Operationen tragen nichts zur Ausdruckskraft einer Sprache 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.

Union vs. Verbund

Übung:

- Zwei Alternativen zur Representation der Hausaufgaben- und Prüfungsergebnisse der Studenten sind:

Resultate_1

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

Resultate_2

STUDENT	KAT	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

- Schreiben Sie SQL-Anfragen, um zwischen beiden zu übersetzen.

Konditionale Ausdrücke (1)

- Während UNION ein übertragbarer Weg für die Fallunterscheidung ist, kann man manchmal auch einen (effizienteren) konditionalen Ausdruck verwenden.

  Konditionale Ausdrücke sind für jedes DBMS verschieden.

- Z.B. Oracle hat Ausdrücke der Form:

  DECODE(X, X_1, Y_1, X_2, Y_2, ..., Z)

- Dies wird ausgewertet, indem man 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.

Konditionale Ausdrücke (2)

- Z.B. Geben Sie die volle Übungskategorie für die Ergebnisse von Ann Smith aus (Oracle Version):

```
SELECT   DECODE(KAT, 'H', 'Hausaufgabe',
                     'Z', 'Zwischenklausur',
                     'F', 'Endklausur',
                          'Unbekannte Kat.'),
         ANR, PUNKTE
FROM     STUDENTEN S, RESULTATE R
WHERE    S.SID = R.SID
AND      VORNAME = 'Ann' AND NACHNAME = 'Smith'
ORDER BY DECODE(KAT, 'H', 1, 'Z', 2, 'E', 3, 4)
```


Konditionale Ausdrücke (3)

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

```
SELECT CASE WHEN KAT='H' THEN 'Hausaufgabe'
            WHEN KAT='Z' THEN 'Zwischenklausur'
            WHEN KAT='E' THEN 'Endklausur'
            ELSE 'Unbekannte Kat.' END
       ANR, PUNKTE
FROM   STUDENTEN S, RESULTATE R
WHERE  S.SID = R.SID
AND    VORNAME = 'Ann' AND NACHNAME = 'Smith'
```

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

Konditionale Ausdrücke (4)

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

```
SELECT CASE KAT WHEN 'H' THEN 'Hausaufgabe',
                WHEN 'Z' THEN 'Zwischenklausur',
                WHEN 'E' THEN 'Endklausur',
                ELSE 'Unbekannte Kat.' END,
       ANR, PUNKTE
FROM   STUDENTEN S, RESULTATE R
WHERE  S.SID = R.SID
AND    VORNAME = 'Ann' AND NACHNAME = 'Smith'
```


Konditionale Ausdrücke (5)

- Eine typische Anwendung von konditionalen Ausdrücken ist die Ersetzung von Null-Werten.
- 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`

Konditionale Ausdrücke (6)

- Z.B. Geben Sie die E-Mail-Adressen aller Studenten aus, schreiben Sie "(keine)", wenn die Spalte Null ist:

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

- Man beachte aber, dass konditionale Ausdrücke normale Terme sind, so dass sie als Eingabe für andere Datentypfunktionen, oder z.B. für Aggregationsfunktionen, dienen können.

Inhalt

Sortieren der Ausgabe (1)

- Eine Ausgabe, die länger als einige Zeilen ist, sollte verständlich 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 der 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, oder negative Werte rot zu färben, etc. Die Sortierung kann jedoch auch wichtig sein, wenn ein Anwendungsprogramm die Daten erhält.

Sortieren der Ausgabe (2)

- Z.B. 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, RESULTATE R
WHERE    S.SID = R.SID
AND      R.KAT = 'H' AND R.ANR = 1
ORDER BY S.NACHNAME
```

VORNAME	NACHNAME
Michael	Jones
Ann	Smith
Richard	Turner

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. Geben Sie die HA-Ergebnisse, sortiert nach Übung, für jede Übung nach Punkten (absteigend), und falls erforderlich alphabetisch nach Namen, aus:

```
SELECT   ANR, PUNKTE, VORNAME, NACHNAME
FROM     STUDENTEN S, RESULTATE R
WHERE    S.SID = R.SID AND R.KAT = 'H'
ORDER BY ANR, PUNKTE DESC, NACHNAME, VORNAME
```


Sortieren der Ausgabe (4)

- Ergebnis der Beispielanfrage der vorherigen Folie:

ANR	PUNKTE	VORNAME	NACHNAME
1	10	Ann	Smith
1	9	Michael	Jones
1	5	Richard	Turner
2	9	Michael	Jones
2	8	Ann	Smith

- Die ersten beiden Tupel haben den gleichen Wert im ersten Suchkriterium (`ANR`), das zweite Kriterium (`PUNKTE DESC`) legt 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 Studenten sortiert nach Gesamtpunktzahl zu erstellen, ohne diese auszugeben. Tools wie SQL*Plus können Ausgabespalten eines Anfrageergebnisses unterdrücken.

- Trotzdem kann man 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 steht. Z.B. könnte man nach `UPPER(NACHNAME)` sortieren, aber `NACHNAME` ausgeben. Mit `DISTINCT` kann man nur nach Ergebnisspalten sortieren (in Oracle kann man sie auch 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 String "VORNAME NACHNAME" gespeichert, wäre es (wahrscheinlich) unmöglich nach dem Nachnamen zu sortieren.

  Wichtige Frage im DB-Design: 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, wo man Ergebnisspalten nicht explizit nennen konnte. Heute sollte man wahrscheinlich Spaltennamen verwenden.
- Nullwerte werden alle als erstes oder als letztes aufgelistet (abhängig vom DBMS).

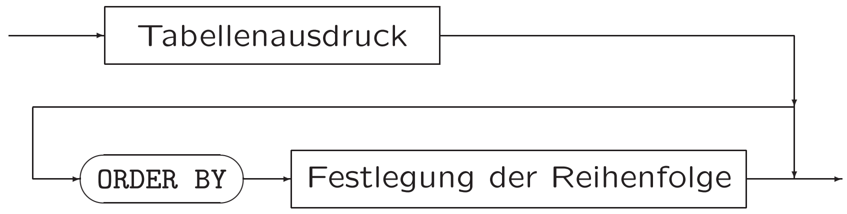
  In Oracle kann man `NULLS FIRST` oder `NULLS LAST` festlegen.

Sortieren der Ausgabe (8)

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

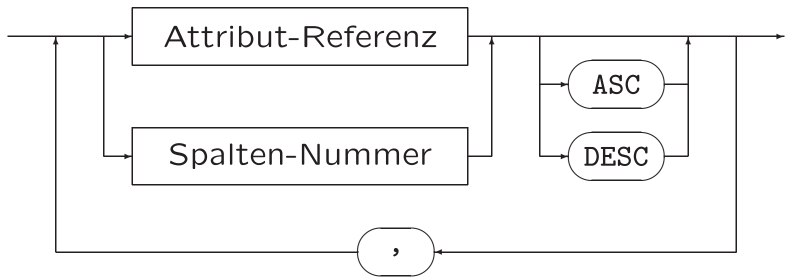
Sortieren der Ausgabe (9)

SQL-Anfrage:



Sortieren der Ausgabe (10)

Festlegung der Reihenfolge:



- 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

Beispieldatenbank (erneut)

STUDENTEN

SID	VORNAME	NAME	EMAIL
101	Ann	Smith	···
102	Michael	Jones	(null)
103	Richard	Turner	···
104	Maria	Brown	···

AUFGABEN

KAT	ANR	THEMA	MAXPT
H	1	Rel. Algeb.	10
H	2	SQL	10
Z	1	SQL	14

RESULTATE

SID	KAT	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

Verbunde in SQL-92 (1)

- Eine wichtige und nützliche Operation der relationalen Algebra ist der Verbund (und Variationen).
- 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.
- Z.B. der natürl. Verbund von RESULTATE und AUFGABEN:

```
SELECT R.KAT AS KAT, R.ANR AS ANR, SID,
       PUNKTE, THEMA, MAXPT
FROM   RESULTATE R, AUFGABEN A
WHERE  R.KAT = A.KAT AND R.ANR = A.ANR
```


Verbunde in SQL-92 (2)

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

```
SELECT SID, ANR, (PUNKTE/MAXPT)*100
FROM   RESULTATE R NATURAL JOIN AUFGABEN A
WHERE  KAT = 'H'
```

- Durch die Schlüsselwörter “NATURAL JOIN” addiert das System automatisch die Verbundbedingung

```
R.KAT = A.KAT AND R.ANR = A.ANR
```

- SQL-92 lässt Verbunde in der FROM-Klausel und auch auf dem Level der äußeren Anfrage (UNION) zu.

  Somit kann man viel im Stil der relationalen Algebra schreiben.

Verbunde in SQL-92 (3)

- Die aktuellen Systeme unterstützen den Standard nur teilweise:
  - ◇ SQL-92 Verbunde werden nur in Oracle 8i unterstützt, Oracle 9i unterstützt fast alle.
  - ◇ Einige Verbundtypen werden in DB2, SQL Server und Access unterstützt, aber der "natürliche Verbund" nicht. Ein Verbund mit expliziter Bedingung ist möglich:

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


Verbunde in SQL-92 (4)

- Mit der expliziten Verbundbedingung wird die Anfrage nicht kürzer als die äquivalente Anfrage mit der `WHERE`-Bedingung.
- Die Ausdruckskraft von SQL wird durch die neuen Verbund-Konstrukte nicht verstärkt.

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

Äußerer Verbund: Wdh.

- Der gewöhnliche Verbund eliminiert Tupel ohne Verbundpartner:

$$\begin{array}{|c|c|}\hline A & B \\ \hline a_1 & b_1 \\ a_2 & b_2 \\ \hline\end{array} \bowtie \begin{array}{|c|c|}\hline B & C \\ \hline b_2 & c_2 \\ b_3 & c_3 \\ \hline\end{array} = \begin{array}{|c|c|c|}\hline A & B & C \\ \hline a_2 & b_2 & c_2 \\ \hline\end{array}$$

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

$$\begin{array}{|c|c|}\hline A & B \\ \hline a_1 & b_1 \\ a_2 & b_2 \\ \hline\end{array} ⟕ \begin{array}{|c|c|}\hline B & C \\ \hline b_2 & c_2 \\ b_3 & c_3 \\ \hline\end{array} = \begin{array}{|c|c|c|}\hline A & B & C \\ \hline a_1 & b_1 & \\ a_2 & b_2 & c_2 \\ \hline\end{array}$$

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

Äußerer Verbund in SQL-92 (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 RESULTATE R
         ON A.KAT = R.KAT AND A.ANR = R.ANR
WHERE    A.KAT = '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-92 (2)

- Äquivalente Anfrage ohne äußeren Verbund:
  (12 vs. 5 Zeilen)

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


Äußerer Verbund in SQL-92 (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 RESULTATE R
         ON S.SID = R.SID
WHERE    R.KAT = 'H'
GROUP BY S.SID, VORNAME, NACHNAME
```

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

Äußerer Verbund in SQL-92 (4)

- Mögliche Verbundpartner dürfen nicht eliminiert werden, nachdem der äußere Verbund gemacht wurde.
- Man muss die Hausaufgabenergebnisse selektieren bevor der äußere Verbund gemacht wird:

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


Äußerer Verbund in SQL-92 (5)

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

```
SELECT   VORNAME, NACHNAME, COUNT(R.ANR)
FROM     STUDENTEN S LEFT OUTER JOIN RESULTATE R
         ON S.SID = R.SID AND R.KAT = '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-92 (6)

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

```
SELECT A.KAT, A.ANR, R.SID, R.PUNKTE
FROM   AUFGABEN A LEFT OUTER JOIN RESULTATE R
       ON A.KAT = 'H' AND R.KAT = 'H'
          AND A.ANR = R.ANR
```

- Übung:

  Wird A.KAT = 'Z' in der Ausgabe auftauchen?

  ☐ ja ☐ nein

Äußerer Verbund in SQL-92 (7)

- MySQL hat keine Unteranfragen, aber manchmal kann man dafür einen äußeren Verbund verwenden.
- Z.B. Studenten ohne eine gelöste Hausaufgabe:

```
SELECT S.SID. S.VORNAME, S.NACHNAME
FROM   STUDENTEN S LEFT OUTER JOIN RESULTATE R
       ON S.SID = R.SID AND R.KAT = 'H'
WHERE  R.KAT IS NULL
```

- Natürlich kann man statt R.KAT jedes Attribut von RESULTATE 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 Tupel der rechten Tabelle.
  - ◇ FULL [OUTER] JOIN: Erhält alle Eingabetupel.
  - ◇ CROSS JOIN: Kartesisches Produkt $\times$.
  - ◇ UNION JOIN: Diese Vereinigung füllt die Spalten der anderen Tabelle mit Nullwerten auf.
- Das in Klammern eingeschlossene ist 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 und die Verbundbedingung ist $R.A_1 = S.A_1 \wedge \cdots \wedge R.A_n = S.A_n$. 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)

- In Übereinstimmung mit dem Standard, liefern der NATURAL Join und der Join mit USING eine Tabelle mit nur einer Kopie der gleichen Attribute.
- Die gleichen Attribute werden zuerst aufgelistet und können nicht mit Tupelvariablen referenziert werden.

```
SELECT *
FROM   RESULTATE R NATURAL JOIN AUFGABEN A
```

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

  Es ist unzulässig auf R.KAT oder A.KAT zu referenzieren, es kann nur KAT 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 SQL-92 lässt 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 auch Verbunde und die Deklaration weiterer Tupelvariablen (getrennt durch ",") in der FROM-Klausel verbinden.
- 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 in Oracle (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 angefügt, 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 in Oracle (2)

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

```
SELECT   A.KAT, A.ANR, COUNT(SID)
FROM     AUFGABEN A, RESULTATE R
WHERE    A.KAT = R.KAT(+) AND A.ANR = R.ANR(+)
GROUP BY A.KAT, 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.