Teil 5: 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 auch Unteranfragen enthalten.
- Eine gegebene Anfrage auf syntaktische Korrektheit prüfen.
- Die Portabilität von Konstrukten beurteilen.

Inhalt

1. Nullwerte
2. IN / NOT IN Unteranfragen, Nichtmonotonie
3. EXISTS / NOT EXISTS, Gültigkeitsbereiche
4. ALL, ANY, SOME
5. Unteranfragen als Terme
6. Unteranfragen unter FROM, Sichten

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

Nullwerte (1)

- Wie in Kapitel 2 erwähnt, können Tabelleneinträge einen Nullwert enthalten (falls nicht mit `NOT NULL` in der Tabellendeklaration ausgeschlossen).

  Im wesentlichen bedeutet das: Der Tabelleneintrag ist leer.

- Der Nullwert ist von allen normalen Werten des Datentyps verschieden, insbesondere ist er verschieden von der Zahl 0 und dem leeren String.

  In Oracle werden (auch in 9i) der Nullwert und der leere String identifiziert. Das ist eine Verletzung des SQL-Standards (Oracle listet dies als "nicht-übereinstimmend" in einem Anhang seines SQL-Referenz-Handbuches auf). Da in Eingabefeldern die beiden Werte nicht unterschieden werden können, ist das aber selten ein Problem.

Nullwerte (2)

- Nullwerte werden in vielen verschiedenen Situationen verwendet, z.B.:
  - ◇ Ein Wert existiert, ist aber unbekannt.

    Angenommen, man will in der Tabelle `STUDENTEN` auch die Telefonnummer der Studenten speichern, aber man kennt möglicherweise nicht von jedem Studenten die Telefonnummer, obwohl wahrscheinlich alle irgendwie telefonisch zu erreichen sind.
  - ◇ Es existiert kein Wert.

    In einer Tabelle mit Vorlesungsdaten könnte es eine Spalte `URL` geben, aber nicht jede Vorlesung hat eine Web-Seite (wenn aber eine Webseite existiert, wäre sie normalerweise auch eingetragen).
  - ◇ Es könnte ein (unbekannter) Wert existieren, oder auch keiner.

Nullwerte (3)

- Anwendungen von Nullwerten, fortgesetzt:
  - ◇ Spalte ist auf dieses Tupel nicht anwendbar.

    Z.B. müssen in Pittsburgh (USA) nur ausländische Studenten einen Toefl-Test ablegen, um ihre Englischkenntnisse zu beweisen. Eine Spalte für die Toefl-Punktzahl in der Tabelle `STUDENTEN` ist für U.S.-Studenten nicht anwendbar. Selbst wenn diese Studenten früher einmal einen Toefl-Test gemacht haben (z.B. weil sie Immigranten sind), ist die Universität an dem Resultat nicht interessiert.
  - ◇ Wert wird später zugewiesen/bekannt gegeben.
  - ◇ Jeder Wert ist möglich.
- Ein Ausschuss fand 13 verschiedene Bedeutungen von Nullwerten.

Nullwerte (4)

Vorteile von Nullwerten:

- Ohne Nullwerte wäre es nötig, die meisten Relationen in viele aufzuspalten ("Subklassen"):
  - ◇ Z.B. `STUDENTEN_MIT_EMAIL`, `STUDENTEN_OHNE_EMAIL`.
  - ◇ Oder extra Relation: `STUD_EMAIL(SID, EMAIL)`.
  - ◇ Das erschwert Anfragen.

    Man braucht Verbunde und Vereinigungen.
- Sind Nullwerte nicht erlaubt, werden sich die Nutzer Werte ausdenken, um die Spalten zu füllen.

  Das macht die DB-Struktur sogar noch unklarer.

Nullwerte (5)

Probleme:

- Da der gleiche Nullwert für verschiedene Zwecke genutzt wird, kann es keine klare Semantik geben.
- SQL benutzt dreiwertige Logik, um Bedingungen mit Nullwerten auszuwerten.

  Da man an die normale zweiwertige Logik gewöhnt ist, kann es Überraschungen geben — einige Äquivalenzen gelten nicht.
- Fast alle Programmiersprachen haben keine Nullwerte. Das erschwert Anwendungsprogramme.

  Wenn also ein Spaltenwert in eine Programmvariable eingelesen wird, muss er auf Nullwerte überprüft werden (→ Indikatorvariablen).

Nullwerte ausschließen (1)

- Da Nullwerte zu Komplikationen führen, kann für jedes Attribut festgelegt werden, ob Nullwerte erlaubt sind oder nicht.
- Es ist wichtig, genau darüber nachzudenken, wo Nullwerte gebraucht werden.
- Viele Spalten als “not null” zu deklarieren, vereinfacht Programme und verringert Überraschungen.
- Die Flexibilität geht jedoch verloren: Nutzer werden gezwungen, für alle “not null”-Attribute Werte einzutragen.

Terme mit Nullwerten (1)

- Da ein Spaltenzugriff einen Nullwert liefern kann, gilt dies auch allgemein für Terme (Wertausdrücke).
- Für Datentyp-Funktionen (z.B. +) gilt:
  - ◇ Ist einer der Eingabewerte (Argumente) Null, so das Ergebnis auch Null.

    Es gibt einige wenige Ausnahmen, z.B. `NVL`. In Oracle auch `||` etc.
  - ◇ Ist z.B. `A` Null, so ist `A+B` ebenfalls Null.
- Das Schlüsselwort `NULL` ist selbst kein Term, obwohl es an vielen Stellen verwendet werden kann, an denen ein Term verlangt wird.

Terme mit Nullwerten (2)

- `NULL` hat keinen Datentyp, also braucht man einen Kontext, so daß der Typ klar ist:
  - ◇ In SQL-92 und DB2 gibt `CAST(NULL AS type)` einen Nullwert des angegebenen Typs zurück.
  - ◇ In Oracle kann `NULL` oft als Term verwendet werden, aber dies ist z.B. ein Fehler:

    ```
    select 1 from dual union select null from dual
    ```

    Man muß `TO_NUMBER(null)` schreiben.
  - ◇ In SQL Server, Access, MySQL wird “`NULL`” als normaler Term verwendet (mit beliebigem Typ).

Dreiwertige Logik (1)

- Betrachten Sie folgende Anfrage:

```
SELECT VORNAME, NACHNAME
FROM   STUDENTEN
WHERE  EMAIL = 'xyz@acm.org'
```

- Was passiert, wenn ein Student in der Spalte EMAIL einen Nullwert hat? Er wird nicht ausgegeben.
- Aber er tritt auch nicht im Ergebnis dieser Anfrage auf (weil der Wert nicht bekannt ist):

```
SELECT VORNAME, NACHNAME
FROM   STUDENTEN
WHERE  NOT (EMAIL = 'xyz@acm.org')
```


Dreiwertige Logik (2)

- Die Bedingung

  `EMAIL = 'xyz@acm.org'`

  ist nicht falsch, wenn EMAIL Null ist, da sonst die Zeile in der negierten Anfrage auftauchen würde.

  Natürlich ist sie auch nicht wahr.

- SQL verwendet eine dreiwertige Logik, um Nullwerte zu behandeln. Die drei Wahrheitswerte sind `wahr`, `falsch` und `unbekannt`.

  Anstelle von "`unbekannt`" liest man auch oft "`Null`".

Dreiwertige Logik (3)

- Die Idee ist, daß Tupel "herausgefiltert" werden sollten, die einen Nullwert in einem Attribut haben, welches für die Anfrage wichtig ist — sie sollten das Anfrageergebnis nicht beeinflussen.
- Der wahre Attribut-Wert ist unbekannt oder existiert nicht, also wäre es falsch zu sagen, daß das Ergebnis eines Vergleichs mit einem Nullwert wahr oder falsch ist.
- In SQL ergibt ein Vergleich mit einem Nullwert immer den dritten Wahrheitswert "`unbekannt`".

Dreiwertige Logik (4)

- Eine Ergebniszeile wird nur dann ausgegeben, wenn die WHERE-Bedingung "`wahr`" ist.
- Somit hat folgende Anfrage ein leeres Ergebnis:

```
SELECT VORNAME, NACHNAME
FROM   STUDENTEN
WHERE  EMAIL = null
```

  Anfrage eigentlich illegal in SQL-92, DB2 lehnt sie ab. Oracle, SQL Server, Access, MySQL akzeptieren sie und geben das leere Ergebnis aus.

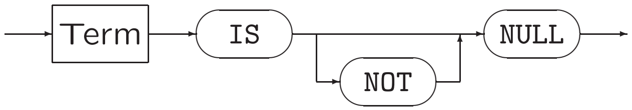
- "`AND`" / "`OR`" leiten den Wahrheitswert "`unbekannt`" weiter, außer das Ergebnis ist klar:
  Z.B. "`wahr OR unbekannt = wahr`".

Dreiwertige Logik (5)

P	Q	NOT P	P AND Q	P OR Q
falsch	falsch	wahr	falsch	falsch
falsch	unbek.	wahr	falsch	unbekannt
falsch	wahr	wahr	falsch	wahr
unbek.	falsch	unbek.	falsch	unbekannt
unbek.	unbek.	unbek.	unbekannt	unbekannt
unbek.	wahr	unbek.	unbekannt	wahr
wahr	falsch	falsch	falsch	wahr
wahr	unbek.	falsch	unbekannt	wahr
wahr	wahr	falsch	wahr	wahr

Test auf Null (1)

Atomare Formel (Form 5):



- Beispiel: `EMAIL IS NULL`
- Man beachte, daß `EMAIL = NULL` nicht funktioniert.

  In Oracle und SQL Server ist dies immer "unbekannt" (nicht "wahr" oder "falsch") und in SQL-92 und DB2 ist es ein Syntax-Fehler. In SQL Server 7 funktioniert "`EMAIL = NULL`" nach dem Befehl "`SET ANSI_NULLS OFF`" (dann wird zweiwertige Logik verwendet).
- `EMAIL NOT NULL` ist ein Syntax-Fehler ("`IS`" fehlt).

Test auf Null (2)

Aufgabe:

- Die folgende Anfrage gibt alle Studenten mit einer Email-Adresse in der Domäne "`.pitt.edu`" aus:

```
SELECT VORNAME, NACHNAME
FROM   STUDENTEN
WHERE  EMAIL IS NOT NULL
AND    EMAIL LIKE '%.pitt.edu'
```

- Ist der Test auf Null notwendig? Oder ist folgende Anfrage äquivalent?

```
SELECT VORNAME, NACHNAME
FROM   STUDENTEN
WHERE  EMAIL LIKE '%.pitt.edu'
```


Probleme mit Nullwerten (1)

- Für diejenigen, die an die normale zweiwertige Logik gewöhnt sind (alle von uns), können Nullwerte manchmal zu Überraschungen führen: Manche logischen Äquivalenzen gelten in SQL nicht.
- Zählt man z.B. alle Studenten mit einer Email-Adresse in der Domäne "`.pitt.edu`" und alle Studenten mit einer anderen Email-Adresse, vermutet man normalerweise, alle Studenten zu erhalten.
- Das ist in SQL nicht wahr — diejenigen mit einem Nullwert in der `EMAIL`-Spalte werden nicht gezählt.

Probleme mit Nullwerten (2)

- Z.B. ist `X = X` “unbekannt” und nicht “wahr”, wenn `X` Null ist.
- Da ein Nullwert verschiedene Bedeutungen haben kann, kann es keine zufriedenstellende Semantik für eine Anfragesprache geben.

  Z.B. würde die Bedeutung “Wert existiert, ist jedoch unbekannt” einem Existenzquantor (“es gibt”) in der Logik entsprechen, und dann die Verwendung der normalen logischen Äquivalenzen erlauben. Die Anfrageauswertung wäre dann allerdings wesentlich komplexer (möglicherweise nicht immer möglich).

Inhalt

1. Nullwerte
2. IN / NOT IN Unteranfragen, Nichtmonotonie
3. EXISTS / NOT EXISTS, Gültigkeitsbereiche
4. ALL, ANY, SOME
5. Unteranfragen als Terme
6. Unteranfragen unter FROM, Sichten

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.
  - ◇ Werden weitere Tabellenzeilen in die Datenbank eingefügt, so erhält man mindestens die gleichen Antworttupel wie zuvor, und eventuell mehr.
- Falls sich eine gewünschte Anfrage aber nicht monoton verhält (Beispiel siehe nächste Folie), kann sie mit diesen Konstrukten nicht formuliert werden.

Nichtmonotones Verhalten (2)

- Beispiel: “Geben Sie alle Studenten aus, die noch keine Hausaufgabe gelöst haben.”
  - ◇ Momentan (im Beispiel-Zustand auf Folie 5-23) wäre Iris Winter eine korrekte Antwort.
  - ◇ Würde man jedoch eine Bewertung für sie eingefügen, wäre dies nicht länger richtig.
  - ◇ Man hat jetzt also eine Obermenge der Tabellenzeilen als Eingabe der Anfrage, aber nicht eine Obermenge der Ergebniszeilen als Ausgabe.

Nichtmonotones Verhalten (3)

- 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.

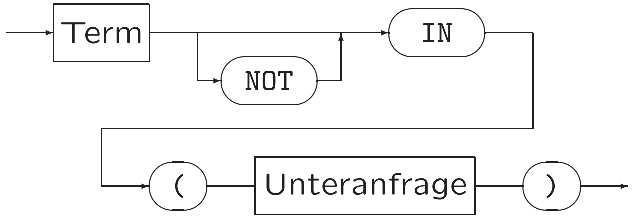
- In SQL-92 wurden Vergleiche auf Tupel-Ebene eingeführt, so daß man z.B. auch schreiben kann

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

  Das ist aber nicht portabel. SQL Server und Access unterstützen es nicht (MySQL erlaubt gar keine Unteranfragen, s.u.). 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 keine Unteranfragen.
- 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.

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

Inhalt

1. Nullwerte
2. IN / NOT IN Unteranfragen, Nichtmonotonie
3. EXISTS / NOT EXISTS, Gültigkeitsbereiche
4. ALL, ANY, SOME
5. Unteranfragen als Terme
6. Unteranfragen unter FROM, Sichten

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

- 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 keine Unteranfragen.

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)

- Beispiel: "Bei welchen Aufgaben haben alle Abgaben mindestens 80% der vollen Punktzahl?"
- In der Logik kann man dies direkt mit einem Allquantor "für alle X" formulieren, in SQL gibt es aber nur einen Existenzquantor EXISTS.
- Das ist aber kein Problem, da "für alle X gilt A" äquivalent ist zu "für kein X ist A falsch", d.h. zu "NOT es gibt ein X mit NOT A".

  So wie sich AND und OR umdrehen, wenn man ein NOT daran vorbeibewegt, gilt das auch für Allquantor und Existenzquantor. Man macht sich ein "NOT NOT" Paar, von dem man eins am Quantor vorbeibewegt.

Allaussagen (2)

- Äquivalente Umformulierung des Beispiels:
  - ◇ Gegeben: "Bei welchen Aufgaben haben alle Abgaben mindestens 80% der vollen Punktzahl?"
  - ◇ SQL: "Bei welchen Aufgaben gibt es keine Abgabe mit weniger als 80% der vollen Punktzahl?"

```
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 (3)

- 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 besteht ein wichtiger Unterschied zwischen der Nicht-Existenz einer Zeile und der Existenz einer Zeile mit einem anderen Wert.
- Verhält sich die benötigte Anfrage nichtmonoton (d.h. die Einfügung einer Zeile kann eine Antwort ungültig machen), dann wird `NOT EXISTS`, `NOT IN`, 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 "für alle" 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

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

ist (unter gewissen Voraussetzungen) äquivalent zu

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

IN vs. EXISTS (2)

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

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

    Das ist kein Problem: Notfalls fügt man die Tupelvariable ein.

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 OR-Verknüpfung 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)

- NOT IN mit einer Unteranfrage, die einen Nullwert liefern kann, ist fast immer ein Fehler.

  Wenn die Unteranfrage den Nullwert liefert, kann das Ergebnis der NOT IN-Bedingung nicht mehr “wahr” sein, selbst wenn der Wert auf der linken Seite des NOT IN nicht von der Unteranfrage geliefert wird.

- Theoretisch ist noch interessant, daß man NOT IN schon exakt in eine äquivalente Formulierung mit NOT EXISTS übersetzen kann, das ist aber recht kompliziert, und man simuliert dabei ja gerade das unerwünschte Verhalten.

Inhalt

1. Nullwerte
2. IN / NOT IN Unteranfragen, Nichtmonotonie
3. EXISTS / NOT EXISTS, Gültigkeitsbereiche
4. ALL, ANY, SOME
5. Unteranfragen als Terme
6. Unteranfragen unter FROM, Sichten

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)

- Die obige Anfrage liefert die besten Ergebnisse für Hausaufgabe 1.
- Aufgabe: Was würde passieren, wenn man "`> ALL`" statt "`>= ALL`" schreibt?

```
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 (3)

- Man kann die korrekte Anfrage (von Folie 5-69) auch so ausdrücken:

```
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 die Äquivalenz von "für alle `X`" und "es gibt kein `X`, so daß nicht" ausgenutzt.

ALL, ANY, SOME (4)

- 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`)`

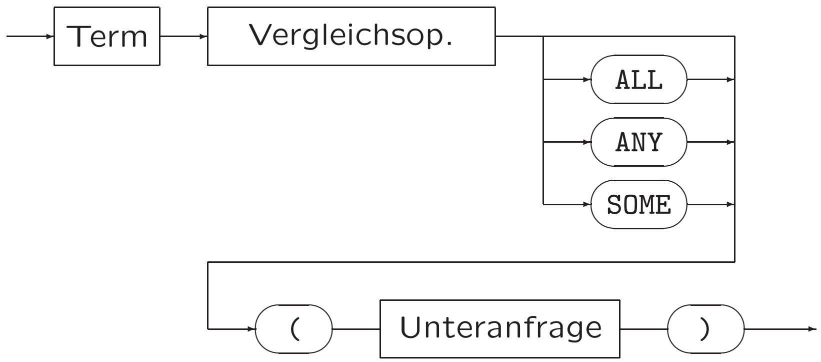
  Es gelten die gleichen Einschränkungen wie oben für `IN` erklärt, auch das Problem mit dem Nullwert für t_2.

- 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 (5)

Atomare Formel (Form 8):



ALL, ANY, SOME (6)

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.

  Siehe Abschnitt ab Folie 5-76. 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.

Inhalt

1. Nullwerte
2. IN / NOT IN Unteranfragen, Nichtmonotonie
3. EXISTS / NOT EXISTS, Gültigkeitsbereiche
4. ALL, ANY, SOME
5. Unteranfragen als Terme
6. Unteranfragen unter FROM, Sichten

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.

  Im Beispiel gilt das (wegen Schlüssel von `AUFGABEN`). Im allgemeinen kann es aber von den Daten abhängen. Die Anfrage könnte bei Tests gut laufen, aber später im Einsatz Fehler liefern. 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.

SELECT-Unteranfragen (1)

- In Systemen, die Unteranfragen als Terme zulassen, können Unteranfragen auch unter `SELECT` verwendet werden.
- Z.B. Ergebnisse für Hausaufgabe 1 mit Nachnamen des Studierenden:

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


SELECT-Unteranfragen (2)

- Man kann das gleiche Ergebnis übersichtlicher und portabeler mit einem klassisch formulierten Verbund erhalten:

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

Man sollte allgemein eine möglichst lesbare Formulierung wählen, d.h. insbesondere eine übliche, gewohnte Formulierung.

- `SELECT`-Unteranfragen sind gelegentlich im Zusammenhang mit Aggregationen interessant (s. Kap. 6).

Inhalt

1. Nullwerte
2. IN / NOT IN Unteranfragen, Nichtmonotonie
3. EXISTS / NOT EXISTS, Gültigkeitsbereiche
4. ALL, ANY, SOME
5. Unteranfragen als Terme
6. Unteranfragen unter FROM, Sichten

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

Unteranfragen unter FROM (1)

- Da das Ergebnis einer SQL-Anfrage eine Tabelle ist, sollte man man Unteranfragen an Stelle einer Tabelle in der FROM-Klausel schreiben können.
- Das war in SQL-86 verboten, und SQL wurde damals oft kritisiert, daß die Konstrukte nicht beliebig kombinierbar ("orthogonal") seien.

  Eine Computersprache ist einfacher/eleganter, wenn es nur wenige Konstrukte gibt, die aber beliebig zusammengesteckt werden können.
- Die Einschränkung war merkwürdig, weil es schon Sichten (s.u.) gab, die den gleichen Effekt hatten.

Unteranfragen unter FROM (2)

- Seit SQL-92 darf man eine Unteranfrage anstelle einer Tabelle unter FROM benutzen.
- In diesem Beispiel wird der Verbund von AUFGABEN und BEWERTUNGEN in einer Unteranfrage berechnet (unnötig kompliziert, siehe nächste Folie):

```
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)

- Die Unteranfrage im obigen Beispiel wirkt eher verkomplizierend: Man sollte sie hier nicht verwenden, sondern den Verbund normal aufschreiben.

  Man sollte für die Arbeit mit der Datenbank immer das am besten geeignete Werkzeug benutzen: Eine Unteranfrage unter FROM wird das nur selten sein. Sie sollten optimale Verständlichkeit Ihrer Anfrage anstreben. Wenn sich eine Anfrage mit dem klassischen Muster (Konstrukte aus Kapitel 4) formulieren läßt, wäre das wohl das beste.

- Intern könnte so eine Anfrage durch Auflösung einer Sicht (s.u.) entstehen.
- Man braucht Unteranfragen unter FROM nur selten, z.B. für geschachtelte Aggregationen (s. Kap. 6).

Unteranfragen unter FROM (4)

- 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 (5)

- 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!

- Weiter außen definierte Tupelvariablen sind aber zugreifbar (wenn dies selbst eine Unteranfrage ist).

Sichten (1)

- Sichten erlauben es, eine Anfrage in der Datenbank abzuspeichern, und ihr einen Namen zu geben (man kann auch die Ergebnisspalten umbenennen):

```
CREATE VIEW HA(VORNAME, NACHNAME, GP)
AS   SELECT VORNAME, NACHNAME, SUM(PUNKTE)
     FROM   STUDENTEN S, BEWERTUNGEN B
     WHERE  S.SID = B.SID AND B.ATYP = 'H'
     GROUP  BY VORNAME, NACHNAME, S.SID
```

- Das Anfrage-Ergebnis ist ja eine Tabelle.
- Man kann Sichten ("virtuelle Tabellen") in Anfragen wie normale Tabellen ("Basistabellen") nutzen.

Sichten (2)

- Z.B. kann man diese Anfrage an die Sicht stellen:

```
SELECT X.VORNAME, X.NACHNAME
FROM   HA X
WHERE  X.GP > 15
```

- Das DBMS kann intern den Namen der Sicht einfach durch die definierende Anfrage ersetzen:

```
SELECT X.VORNAME, X.NACHNAME
FROM  (SELECT VORNAME, NACHNAME, SUM(PUNKTE) GP
       FROM   STUDENTEN S, BEWERTUNGEN B
       WHERE  S.SID = B.SID AND B.ATYP = 'H'
       GROUP  BY VORNAME, NACHNAME, S.SID) X
WHERE  X.GP > 15
```


Sichten (3)

- Sichten sind abgeleitete, virtuelle Tabellen, die aus den (tatsächlich abgespeicherten) Basistabellen berechnet werden (z.B. Alter aus Geburtsdatum).

  Bei Sichten wird die definierende Anfrage gespeichert (intensionale Definition), bei Basistabellen die Tupel (extensionale Definition).

- Sichten können also nie Informationen enthalten, die nicht schon in den Basistabellen enthalten ist.

  Man könnte allerdings eventuell die Definition der Sicht (die Berechnungsvorschrift) als zusätzliche Information ansehen.

- Sichten können aber die in den Basistabellen enthaltene Information anders strukturiert anzeigen.

Sichten (4)

- Die Anfrage der Sicht wird im Prinzip immer neu ausgewertet, wenn die Sicht benutzt wird.

  Der Anfrageoptimierer wird natürlich versuchen, nicht jedesmal die vollständige Sicht zu berechnen, sondern nur den Teil, der für die gegebene Anfrage relevant ist.

- Wenn die in der Sicht verwendeten Basistabellen (`STUDENTEN`, `BEWERTUNGEN`) geändert werden, spiegelt die Sicht automatisch diese Änderungen wider.

  Das ist ein wesentlicher Unterschied zu einer Lösung, die eine neue Tabelle anlegt, und da hinein das Ergebnis der definierenden Anfrage speichert. Eine solche Tabelle müßte manuell (oder über Programme) aktualisiert werden. Moderne Systeme bieten "materialisierte Sichten" (explizit gespeichert, automatisch aktualisiert, ggf. nicht sofort).

Sichten (5)

- Man kann Sichten definieren, die für verschiedene Benutzer unterschiedliche Ergebnisse liefern:

```
CREATE VIEW MEINE_PUNKTE(ATYP, ANR, PUNKTE) AS
SELECT B.ATYP, B.ANR, B.PUNKTE
FROM   STUDENTEN S, BEWERTUNGEN B
WHERE  S.SID = B.SID AND S.NACHNAME = USER
```

- "`USER`" liefert den DB-Login-Namen des aktuellen Nutzers (der die Anfrage stellt).

  Damit das funktioniert, müßten die Studierenden jeweils ihren Nachnamen als DB-Account haben. Man könnte in der `STUDENTEN`-Tabelle aber auch eine extra Spalte für die Benutzerkennung vorsehen.

- Sichten können auch vom Datum abhängen.

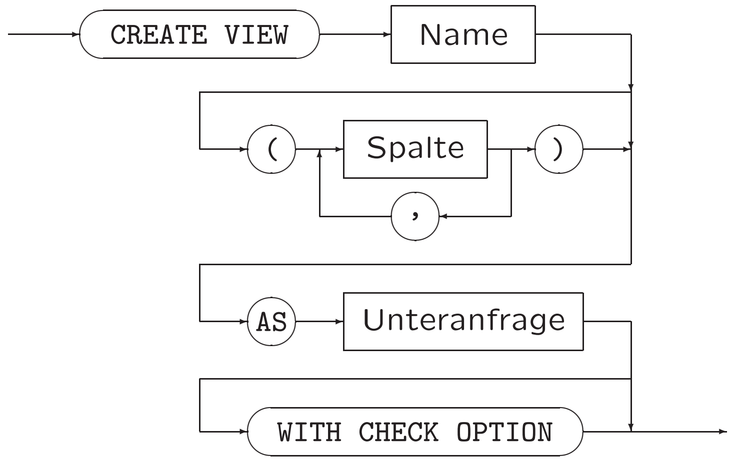
Sichten (6)

- Sichten können auch in der Definition anderer Sichten verwendet werden.
- Auf diese Art können komplexe Anfragen Schritt für Schritt aufgebaut werden.
- Rekursive Sichten waren in SQL-92 verboten.

  Alle in einer Sichtdefinition verwendeten Sichten müssen vorher schon vollständig definiert sein.
- SQL-99 erlaubt rekursive Sichten, sie werden aber erst in wenigen Systemen unterstützt (z.B. DB2).

  Rekursive Sichten sind eine Spezialität von Deduktiven DBen.

Syntax (1)



Syntax (2)

- `ORDER BY` ist nicht in View-Definitionen erlaubt.

  Normalerweise bewirkt es auch nur ganz am Ende einer Anfrage etwas, während Sichten als Teilanfragen einer größeren Anfrage genutzt werden. Oracle erlaubt `ORDER BY`. Im Zusammenhang mit `ROWNUM`-Bedingungen würde es auch in Unteranfragen Sinn machen.
- `CHECK OPTION`: Für Zugriffsrechte (s. Kap. 11).
- Sichtdefinitionen können gelöscht werden mit:

  `DROP VIEW ⟨NAME⟩`
- In Oracle kann man folgendes schreiben:

  `CREATE OR REPLACE VIEW ⟨NAME⟩ ...`

  Überschreibt ggf. bereits existierende Definition.

Anwendungen von Sichten (1)

- Bequemlichkeit / Wiederverwendung: Wiederkehrende Muster in Anfragen sind bereits vordefiniert.
- Die Basisrelationen sollten keine Redundanzen enthalten. Relationen mit abgeleiteten Informationen sind aber manchmal in Anfragen bequemer.

  Redundante Daten in Sichten sind kein Problem, weil diese Daten ja nicht abgespeichert werden, sondern nach Bedarf berechnet werden.
- Schrittweiser Aufbau komplexer Anfragen.
- Anpassung des DB-Schemas an die Wünsche verschiedener Benutzer / Benutzer-Gruppen.

Anwendungen von Sichten (2)

- Sicherheit: Bestimmte Benutzer sollten nur einen Teil einer Tabelle (gewisse Zeilen/Spalten) sehen können, oder nur aggregierte/anonymisierte Daten.

  Ohne Sichten ist die Granularität für Zugriffsrechte im wesentlichen die Tabelle. Nur die Änderungsrechte können üblicherweise auch für einzelne Spalten vergeben werden.

- Logische Datenunabhängigkeit: Man kann neue Attribute zu einer Tabelle hinzufügen, und die alte Version noch als Sicht zur Verfügung stellen.