Teil 4: SQL I

Literatur:

- Elmasri/Navathe: Fundamentals of Database Systems, 3. Auflage, 1999. Chap. 8, "SQL — The Relational Database Standard" (Sect. 8.2, 8.3.3, Teil von 8.3.4.)
- Silberschatz/Korth/Sudarshan: Database System Concepts, 3. Auflage, 1999. Ch. 4.
- Kemper/Eickler: Datenbanksysteme, Kapitel 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.
- Oracle 8i SQL Reference, Release 2 (8.1.6), Dez. 1999, Teil 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, 2nd Ed. (Teil der MSDN Library).
- DuBois: MySQL. New Riders Publishing, 2000, ISBN 0-7357-0921-1, 756 Seiten.
- Boyce/Chamberlin: SEQUEL: A structured English query language. In ACM SIGMOD Conf. on the Management of Data, 1974.
- Astrahan et al: System R: A relational approach to database management. ACM Transactions on Database Systems 1(2), 97–137, 1976.

Lernziele

Nach diesem Kapitel sollten Sie Folgendes können:

- Fortgeschrittene Anfragen in SQL schreiben, z.B. mit mehreren Tupelvariablen über einer Relation.

  Unteranfragen, Aggregationen, `UNION`, Outer Joins und Sortierung werden in Kapitel 5 behandelt.

- Fehler und unnötige Komplikationen vermeiden.

  Z.B. sollten Sie das Konzept einer inkonsistenten Bedingung erklären können.

- Die Bedeutung von gegebenen Anfragen erklären können.
- Anfragen auf Fehler oder Äquivalenz überprüfen.

Inhalt

1. Einführung: SELECT-FROM-WHERE
2. Lexikalische Syntax
3. Tupelvariablen, Terme, Bedingungen
4. Verbunde, etwas Logik
5. Mehr über Vergleiche, weitere Bedingungen
6. SELECT-Klausel, Duplikate

Beispiel-Datenbank (1)

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

Beispiel-Datenbank (2)

- `STUDENTEN`: enthält eine Zeile für jeden Studenten.
  - ◇ `SID`: "Studenten-ID" (eindeutige Nummer).
  - ◇ `VORNAME`, `NACHNAME`: Vor- und Nachname.
  - ◇ `EMAIL`: Email-Adresse (kann NULL sein).
- `AUFGABEN`: enthält eine Zeile für jede Aufgabe.
  - ◇ `ATYP`: Typ/Kategorie der Aufgabe.

    Z.B. `'H'`: Hausaufgabe, `'Z'`: Zwischenklausur, `'E'`: Endklausur.
  - ◇ `ANR`: Aufgabennummer (innerhalb des Typs).
  - ◇ `THEMA`: Thema der Aufgabe.
  - ◇ `MAXPT`: Maximale/volle Punktzahl der Aufgabe.

Beispiel-Datenbank (3)

- `BEWERTUNGEN`: enthält eine Zeile für jede abgegebene Lösung zu einer Aufgabe.
  - ◇ `SID`: Student, der die Lösung abgegeben hat.

    Dies referenziert eine Zeile in `STUDENTEN`.
  - ◇ `ATYP`, `ANR`: Identifikation der Aufgabe.

    Zusammen identifiziert dies eine Zeile in `AUFGABEN`.
  - ◇ `PUNKTE`: Punkte, die der Student für die Lösung bekommen hat.
  - ◇ Falls es keinen Eintrag für einen Studenten und eine Aufgabe gibt: Aufgabe nicht abgegeben.

Beispiel-Datenbank (4)

Integritätsbedingungen:

- Schlüssel:
  - ◇ In `STUDENTEN` gibt es keine zwei Zeilen mit der gleichen `SID`.
  - ◇ In `AUFGABEN` gibt es keine zwei Zeilen, die sowohl im `ATYP` als auch in `ANR` übereinstimmen.

    Zeilen können den gleichen `ATYP` oder die gleiche `ANR` haben, aber nicht beides (kombinierter Schlüssel, bestehend aus zwei Spalten).
  - ◇ In `BEWERTUNGEN` gibt es keine zwei Zeilen, die in `SID`, `ATYP` und `ANR` übereinstimmen.

    Dieser Schlüssel setzt sich sogar aus drei Spalten zusammen.

Beispiel-Datenbank (5)

Integritätsbedingungen, Forts.:

- Fremdschlüssel:
  - ◇ Jeder Wert für `SID`, der in `BEWERTUNGEN` auftaucht, existiert auch in `STUDENTEN`.
  - ◇ Jede Kombination von Werten für `ATYP` und `ANR` aus `BEWERTUNGEN` existiert auch in `AUFGABEN`.
- Sonstiges (Beispiel):
  - ◇ Alle Spalten außer `EMAIL` sind nicht null.
  - ◇ Die Punktzahl ist nicht negativ.

Einfache SQL-Anfragen

Einfache Ein-Tabellen-Anfragen (siehe Kapitel 2):

- Eine einfache SQL-Anfrage hat folgende Form:

```
SELECT Spalten
FROM   Tabelle
WHERE  Bedingung
```

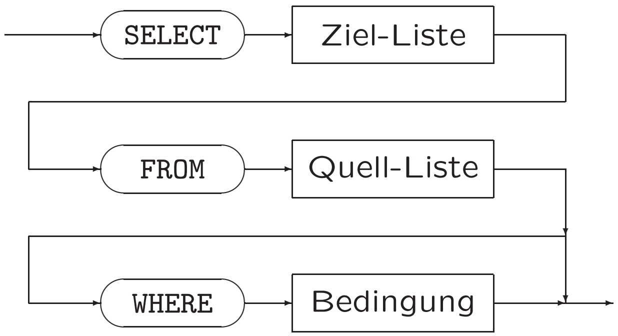
- Z.B. gibt die folgende Anfrage alle Hausaufgaben-ergebnisse von Studentin 101 aus:

```
SELECT ANR, PUNKTE
FROM   BEWERTUNGEN
WHERE  ATYP = 'H' AND SID = 101
```

ANR	PUNKTE
1	10
2	8

Basis-Anfrage-Syntax (1)

SELECT-Ausdruck (vereinfacht):



Basis-Anfrage-Syntax (2)

- Die drei Teile der Anfrage nennt man auch "SELECT-Klausel", "FROM-Klausel", "WHERE-Klausel".
- Jede SQL-Anfrage muß die Schlüsselwörter SELECT und FROM enthalten.

  Oracle stellt eine Relation "DUAL" zur Verfügung, die nur eine Zeile hat. Sie kann benutzt werden, wenn nur eine Berechnung ohne Zugriff auf die DB durchgeführt wird: "SELECT TO_CHAR(SQRT(2)) FROM DUAL" berechnet $\sqrt{2}$.

- In SQL Server, Access und MySQL kann jedoch die FROM-Klausel weggelassen werden, z.B. SELECT 1+1.

  In Oracle, DB2 und dem SQL-92-Standard ist dies ein Syntax-Fehler.

SQL-Syntax in der Vorlesung

- SQL:2003 ist zu groß, um es hier vollständig zu behandeln. Ein großer Teil des Standards ist auch noch nicht in derzeitigen DBMS implementiert.
- SQL/89 (∼ Einstiegs-Level von SQL-92) wird vollständig behandelt und der Teil von SQL:2003, der in den meisten DBMS implementiert ist.

  Manchmal werden Details der SQL-Syntax spezieller Systeme erklärt, meist im Kleingedruckten. Dies ist irrelevant für Klausuren. Sie sollen einen Eindruck von der Portabilität der Konstrukte geben (und helfen, wenn man mit diesem DBMS arbeiten muß). In Klausuren werden nur Punkte abgezogen, wenn man extrem nicht-portable Konstrukte verwendet (z.B. Anfragen, die nur in MySQL laufen).

Inhalt

Lexikalische Syntax

- Die lexikalische Syntax einer Sprache definiert, wie Wortsymbole ("Token") aus einzelnen Zeichen zusammengesetzt werden. Z.B. definiert sie die genaue Syntax von
  - ◇ Bezeichnern (Namen für z.B. Tabellen, Spalten),
  - ◇ Literalen (Datentyp-Konstanten, z.B. Zahlen),
  - ◇ Schlüsselwörtern, Operatoren, Trennzeichen.
- Anschließend wird die Syntax von Anfragen u.s.w. basierend auf diesen Wortsymbolen definiert.

  D.h. eine Anfrage wird dann als Folge von Token definiert, nicht als Folge von Zeichen. Diese Zweiteilung vereinfacht manches.

Leerzeichen und Kommentare

Leerplatz ist zwischen Wortsymbolen (Token) erlaubt:

- Leerzeichen (meist auch Tabulator-Zeichen)
- Zeilenumbrüche
- Kommentare (Bemerkungen für menschliche Leser, vom DBMS einfach ignoriert):
  - ◇ Von "--" bis ⟨Zeilenende⟩

    Unterstützt in SQL-92, Oracle, SQL Server, IBM DB2, MySQL. MySQL benötigt ein Leerzeichen nach "--", SQL-92 nicht. Access unterstützt diesen Kommentar nicht und auch nicht `/* ...*/`.
  - ◇ Von "/*" bis "*/"

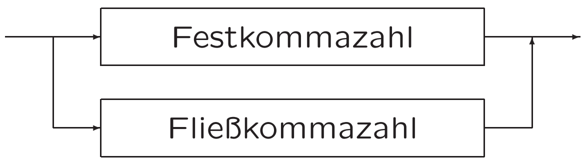
    Nur in Oracle, SQL Server und MySQL: weniger portabel.

Formatfreie Sprache

- Obige Regel (beliebige Leerzeichen zwischen Token) impliziert, daß SQL eine formatfreie Sprache wie Pascal, C oder Java ist:
  - ◇ Es ist nicht nötig, daß "`SELECT`", "`FROM`", "`WHERE`" am Anfang neuer Zeilen stehen. Man kann auch die ganze Anfrage auf eine Zeile schreiben.
  - ◇ Z.B. kann man komplexe Bedingungen auf mehrere Zeilen verteilen und die Struktur mit Hilfe von Einrückungen deutlich machen.

Zahlen (1)

- Numerische Literale sind Konstanten numerischer Datentypen (Fixpunkt- und Gleitkommazahlen).
- Z.B.: `1`, `+2.`, `-34.5`, `-.67E-8`
- Zahlen stehen nicht in Hochkommas!
- Numerisches Literal:



Zahlen (2)

- Bei einer Festkommazahl wird mit einer festen Anzahl Stellen vor und nach dem Komma gerechnet.

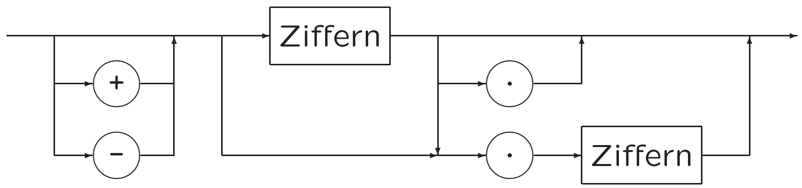
  Z.B. bei Geldbeträgen (in Euro) bietet es sich an, auf Cent genau, also mit zwei Nachkommastellen zu rechnen. Wenn ein Betrag zu groß für die vereinbarte Anzahl Stellen ist, gibt es einen Fehler. Ganze Zahlen ("integer") sind spezielle Festkommazahlen (0 Nachkommastellen).

- Bei Fließkommazahlen wird insgesamt mit einer festen Anzahl Stellen gerechnet, aber die Position des Kommas ist variabel, je nachdem, ob die Zahlen groß oder klein sind.

  Das führt zu unüberschaubaren Rundungen (bei größeren Zahlen wird auch mal nur auf Tausender genau gerechnet, dafür gibt es keinen Fehler). Bei Geldbeträgen verbieten sich Fließkommazahlen.

Zahlen (3)

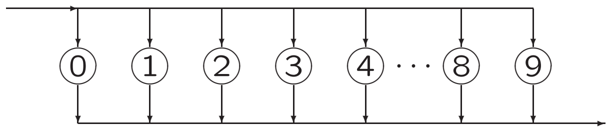
- Festkommazahl ("Exact Numeric Literal")



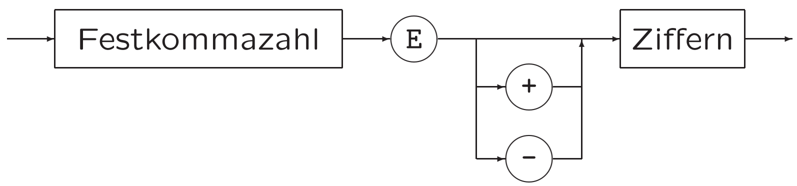
- Ziffern (vorzeichenlose ganze Zahl):

Zahlen (4)

- Ziffer:



- Fließkommazahl ("Approximate Numeric Literal"):



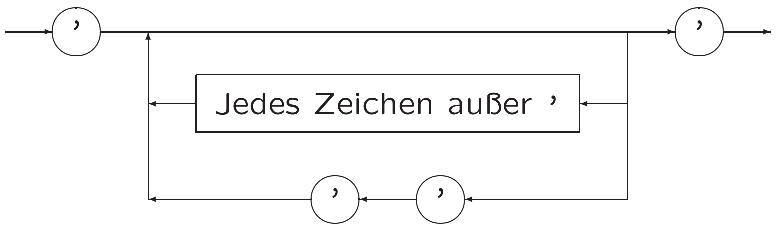
Zeichenketten (1)

- Ein Zeichenketten-Literal (String-Konstante) ist eine Folge von Zeichen, eingeschlossen in Hochkommas, z.B.
  - ◇ `'abc'`
  - ◇ `'Dies ist ein String.'`
- Hochkommas in Zeichenketten müssen verdoppelt werden, z.B. `'John''s book'`.

  Der tatsächliche Wert der Zeichenkette ist `John's book` (mit einfachem Hochkomma). Das Verdoppeln ist nur eine Art, es einzugeben. Ohne die Verdopplung würde das DBMS denken, daß die Zeichenkette bei dem Hochkomma schon zu Ende ist, und dann vermutlich kurz danach einen Syntaxfehler melden.

Zeichenketten (2)

- String Literal:



- Natürlich ist auch die leere Zeichenkette erlaubt: `''`.

  In Oracle werden die leere Zeichenkette und der Nullwert identifiziert. Das entspricht nicht dem Standard.

Zeichenketten (3)

- SQL-92 erlaubt das Splitten von Zeichenketten (jedes Segment eingeschlossen in ’) zwischen Zeilen.

  MySQL unterstützt diese Syntax, Oracle, SQL Server und Access nicht. Zeichenketten können aber mit dem Konkatenations-Operator (|| in Oracle, + in SQL Server und Access) kombiniert werden.
- SQL-92 und alle fünf DBMS erlauben Zeilenumbrüche in Zeichenketten-Konstanten.

  D.h., das Hochkomma kann man auf einer folgenden Zeile schließen.
- MS SQL Server, MS Access und MySQL erlauben auch in Anführungszeichen " eingeschlossene String-Literale. Nicht konform zum Standard!

Andere Konstanten (1)

- Es gibt mehr Datentypen als nur Zahlen und Zeichenketten, z.B. (s. Kapitel 8):
  - ◇ Zeichenketten mit nationalem Zeichensatz
  - ◇ Datum, Zeit, Zeitstempel, Datum-/Zeit-Intervall
  - ◇ Bit-Strings, binäre Daten
  - ◇ Large Objects (Dateien als Tabelleneintrag)
- Die Syntax der Konstanten dieser Datentypen ist allgemein sehr systemabhängig.

  Oft gibt es keine Konstanten dieser Typen, aber es gibt eine automatische Typ-Konvertierung (“coercion”) von Strings.

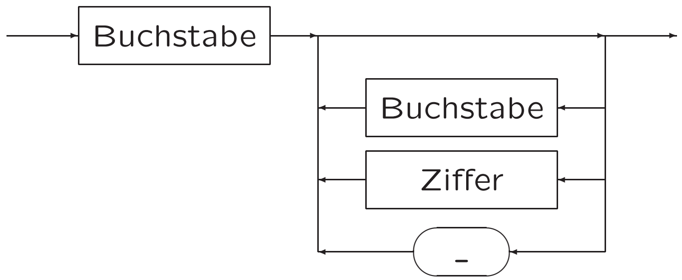
Andere Konstanten (2)

- Z.B. werden Datumswerte wie folgt geschrieben:
  - ◇ Oracle: `'31-OCT-02'` (US), `'31.10.2002'` (D).

    Das Default-Format (Teil der nationalen Sprach-Einstellungen) wird automatisch konvertiert, ansonsten: `TO_DATE('31.10.2002','DD.MM.YYYY')`.
  - ◇ SQL-92-Syntax: `DATE '2002-10-31'`.
  - ◇ MySQL nimmt diese Syntax (auch ohne "`DATE`").
  - ◇ DB2: `'2002-10-31'`, `'10/31/2002'`, `'31.10.2002'`.
  - ◇ SQL Server: z.B. `'20021031'`, `'10/31/2002'`, `'October 31, 2002'` (abhängig von Sprache).
  - ◇ Access: `#10/31/2002#` (US), `#31.10.2002#` (D).

Bezeichner (1)

- U.a. als Tabellen- und Spaltennamen verwendet.
- Bezeichner:



- Z.B. `Dozenten_Name`, `X27`, aber nicht `_XYZ`, `12`, `2BE`.

Bezeichner (2)

- Bezeichner können bis 18 Zeichen haben (mind.).

System	Länge	Erstes Zeichen	Andere Zeichen
SQL-86	$\leq$ 18	A-Z	A-Z,0-9
SQL-92	$\leq$ 128	A-Z,a-z	A-Z,a-z,0-9,_
Oracle	$\leq$ 30	A-Z,a-z	A-Z,a-z,0-9,_,#,$
SQL Server	$\leq$ 128	A-Z,a-z,_,(@,#)	A-Z,a-z,0-9,_,@,#,$
IBM DB2	$\leq$ 18 (8)	A-Z,a-z	A-Z,a-z,0-9,_
Access	$\leq$ 64	A-Z,a-z	A-Z,a-z,0-9,_
MySQL	$\leq$ 64	A-Z,a-z,0-9,_,$	A-Z,a-z,0-9,_,$

Intermediate SQL-92: "_" am Ende verboten. Entry-Level: Wie SQL-86 (plus "_"). In MySQL können Bezeichner mit Ziffern beginnen, aber müssen mind. einen Buchstaben enthalten. Access könnte mehr Zeichen zulassen, abhängig vom Kontext.

- Müssen verschieden von reservierten Wörtern sein.

  Es gibt viele reservierte Wörter, siehe unten. Einbettungen in Programmiersprachen (PL/SQL, Visual Basic) fügen noch mehr hinzu.

Bezeichner (3)

- Es ist grundsätzlich möglich, auch nationale Zeichen (z.B. Umlaute) zu verwenden.

  Das ist implementierungsabhängig. Z.B. wählt man in Oracle bei der Installation einen DB-Zeichensatz. Alphanumerische Zeichen von diesem Zeichensatz können in Bezeichnern verwendet werden.

  Ich persönlich würde Umlaute möglichst vermeiden, und in Tabellen- und Spaltennamen ist das noch eher möglich als in den eigentlichen Daten (da braucht man Umlaute schon wegen der korrekten Sortierung). Da Umlaute auf viele verschiedene Arten codiert werden können, könnte es Schwierigkeiten geben, wenn z.B. Programme mit SQL-Anfragen zwischen verschiedenen Rechnern ausgetauscht werden. Vielleicht ist das aber schon eine überholte Ansicht. Die Entwicklung geht wohl in Richtung Unicode, und wenn alle eine einheitliche Codierung verwenden, gibt es keine Schwierigkeiten mehr (allerdings definiert auch Unicode mehrere Codierungen ...).

Bezeichner (4)

- Die Groß-/Kleinschreibung ist bei Bezeichnern und Schlüsselwörtern egal (sie sind nicht case-sensitiv).

  Das scheint das zu sein, was der SQL-92-Standard sagt (das Buch von Date/Darwen über den Standard stellt es klar so dar). Oracle SQL*Plus konvertiert alle Zeichen außerhalb von Hochkommas in Großbuchstaben. In SQL Server kann Case-Sensitivität bei der Installation gewählt werden. In MySQL hängt die Case-Sensitivität der Tabellennamen von der Case-Sensitivität von Dateinamen im zugrundeliegenden Betriebssystem ab (Tabellen als Dateien gespeichert). Innerhalb einer Anfrage muß man in MySQL konsistent bleiben. Schlüsselwörter und Spaltennamen sind in MySQL jedoch nie case-sensitiv.

- Z.B. werden `ABC` und `abc` als gleich angesehen.

Bezeichner (5)

- Betrachten Sie z.B. die Anfrage:

```
SELECT VORNAME, NACHNAME
FROM   STUDENTEN
```

- Folgende Anfrage ist vollkommen äquivalent:

```
select   Vorname ,
           NachNaMe -- huebsch haesslich
      From Studenten
```

  (Dies zeigt auch die Formatfreiheit von SQL.)

  Achtung: Zeichenketten-Vergleiche sind normalerweise case-sensitive:

  `select x.name from studenten x where x.name = 'lisa'`

  wird keine Antwort liefern (obwohl es `'Lisa'` gibt).

Reservierte Wörter

- In SQL gibt es ziemlich viele Worte mit einer speziellen Bedeutung (Schlüsselworte), z.B. `SELECT`.
- Nach der Syntax wären es eigentlich Bezeichner, aber diese Worte sind von SQL reserviert, d.h. man darf sie nicht als Tabellen- oder Spaltennamen verwenden.
- Man könnte sonst mehrdeutige SQL-Anfragen konstruieren.

  Z.B. "`SELECT A FROM FROM R`", wenn es Tabellen "`FROM`" und "`R`" gibt, die beide eine Spalte "`A`" haben. Solche Fälle sind in SQL ausgeschlossen (man würde einen Syntaxfehler erhalten, weil man "`FROM`" nicht als Tabellennamen verwenden kann).

Reservierte Wörter - SQL (1)

[1] = Oracle 8.0
[2] = SQL-92
[3] = SQL Server 7

— A —

ABSOLUTE[2]
ACCESS[1]
ACTION[2]
ADD[1,2,3]
ALL[1,2,3]
ALLOCATE[2]
ALTER[1,2,3]
AND[1,2,3]
ANY[1,2,3]
ARE[2]
AS[1,2,3]
ASC[1,2,3]
ASSERTION[2]
AT[2]
AUTHORIZATION[2,3]
AUDIT[1]
AVG[2,3]

— B —

BACKUP[3]
BEGIN[2,3]
BETWEEN[1,2,3]
BIT[2]
BIT_LENGTH[2]
BOTH[2]
BREAK[3]
BROWSE[3]
BULK[3]
BY[1,2,3]

— C —

CASCADE[2,3]
CASCADED[2]
CASE[2,3]
CATALOG[2]
CHAR[1,2]
CHARACTER[2]
CHAR_LENGTH[2]
CHARACTER_LENGTH[2]
CHECK[1,2,3]
CHECKPOINT[3]
CLOSE[2,3]
CLUSTER[1]
CLUSTERED[3]
COALESCE[2,3]
COLLATE[2]
COLLATION[2]
COLUMN[1,3]
COMMENT[1]
COMMIT[2,3]
COMMITTED[3]
COMPRESS[1]
COMPUTE[3]

Reservierte Wörter - SQL (2)

CONFIRM[3]
CONNECT[1,2]
CONNECTION[2]
CONSTRAINT[2,3]
CONSTRAINTS[2]
CONTAINS[3]
CONTAINSTABLE[3]
CONTINUE[2,3]
CONTROLROW[3]
CONVERT[2,3]
CORRESPONDING[2]
COUNT[2,3]
CREATE[1,2,3]
CROSS[2,3]
CURRENT[1,2,3]
CURRENT_DATE[2,3]
CURRENT_TIME[2,3]
CURRENT_TIMESTAMP[2,3]
CURRENT_USER[2,3]
CURSOR[2,3]
— D —
DATABASE[3]
DATE[1,2]
DAY[2]
DBCC[3]
DEALLOCATE[2,3]
DEC[2]
DECIMAL[1,2]
DECLARE[2,3]
DEFAULT[1,2,3]
DEFERRABLE[2]
DEFERRED[2]
DELETE[1,2,3]
DENY[3]
DESC[1,2]
DESCRIBE[2]
DESCRIPTOR[2]
DIAGNOSTICS[2]
DISCONNECT[2]
DISK[3]
DISTINCT[1,2,3]
DISTRIBUTED[3]
DOMAIN[2]
DOUBLE[2,3]
DROP[1,2,3]
DUMMY[3]
DUMP[3]
— E —
ELSE[1,2,3]
END[2,3]
END-EXEC[2]
ERRLVL[3]
ERROREXIT[3]
ESCAPE[2,3]
EXCEPT[2,3]
EXCEPTION[2]

Reservierte Wörter - SQL (3)

EXCLUSIVE[1]
EXEC[2,3]
EXECUTE[2,3]
EXISTS[1,2,3]
EXIT[3]
EXTERNAL[2]
EXTRACT[2]
— F —
FALSE[2]
FETCH[2,3]
FILE[1,3]
FILLFACTOR[3]
FIRST[2]
FLOAT[1,2]
FLOPPY[3]
FOR[1,2,3]
FOREIGN[2,3]
FOUND[2]
FREETEXT[3]
FREETEXTTABLE[3]
FROM[1,2,3]
FULL[2,3]
— G —
GET[2]
GLOBAL[2]
GO[2]
GOTO[2,3]
GRANT[1,2,3]
GROUP[1,2,3]
— H —
HAVING[1,2,3]
HOLDLOCK[3]
HOUR[2]
— I —
IDENTITY[2,3]
IDENTITY_INSERT[3]
IDENTITYCOL[3]
IDENTIFIED[1]
IF[3]
IMMEDIATE[1,2]
IN[1,2,3]
INCREMENT[1]
INDEX[1,3]
INDICATOR[2]
INITIAL[1]
INITIALLY[2]
INNER[2,3]
INPUT[2]
INSENSITIVE[2]
INSERT[1,2,3]
INT[2]
INTEGER[1,2]
INTERSECT[1,2,3]
INTERVAL[2]
INTO[1,2,3]
IS[1,2,3]

Reservierte Wörter - SQL (4)

ISOLATION[2,3]
— J —
JOIN[2,3]
— K —
KEY[2,3]
KILL[3]
— L —
LANGUAGE[2]
LAST[2]
LEADING[2]
LEFT[2,3]
LEVEL[1,2,3]
LIKE[1,2,3]
LINENO[3]
LOAD[3]
LOCAL[2]
LOCK[1]
LONG[1]
LOWER[2]
— M —
MATCH[2]
MAX[2,3]
MAXEXTENTS[1]
MIN[2,3]
MINUS[1]
MINUTE[2]
MIRROREXIT[3]
MODE[1]
MODIFY[1]
MODULE[2]
MONTH[2]
— N —
NAMES[2]
NATIONAL[2,3]
NATURAL[2]
NCHAR[2]
NETWORK[1]
NEXT[2]
NO[2]
NOAUDIT[1]
NOCHECK[3]
NOCOMPRESS[1]
NONCLUSTERED[3]
NOT[1,2,3]
NOWAIT[1]
NULL[1,2,3]
NULLIF[2,3]
NUMBER[1]
NUMERIC[2]
— O —
OCTET_LENGTH[2]
OF[1,2,3]
OFF[3]
OFFLINE[1]
OFFSETS[3]
ON[1,2,3]

Reservierte Wörter - SQL (5)

ONCE[3]
ONLINE[1]
ONLY[2,3]
OPEN[2,3]
OPENDATASOURCE[3]
OPENQUERY[3]
OPENROWSET[3]
OPTION[1,2,3]
OR[1,2,3]
ORDER[1,2,3]
OUTER[2,3]
OUTPUT[2]
OVER[3]
OVERLAPS[2]
— P —
PARTIAL[2]
PCTFREE[1]
PERCENT[3]
PERM[3]
PERMANENT[3]
PIPE[3]
PLAN[3]
POSITION[2]
PRECISION[2,3]
PREPARE[2,3]
PRESERVE[2]
PRIMARY[2,3]
PRINT[3]
PRIOR[1,2]
PRIVILEGES[1,2,3]
PROC[3]
PROCEDURE[2,3]
PROCESSEXIT[3]
PUBLIC[1,2,3]
— R —
RAISERROR[3]
RAW[1]
READ[2,3]
READTEXT[3]
REAL[2]
RECONFIGURE[3]
REFERENCES[2,3]
RELATIVE[2]
RENAME[1]
REPEATABLE[3]
REPLICATION[3]
RESOURCE[1]
RESTORE[3]
RESTRICT[2,3]
RETURN[3]
REVOKE[1,2,3]
RIGHT[2,3]
ROLLBACK[2,3]
ROW[1]
ROWCOUNT[3]
ROWGUIDCOL[3]

Reservierte Wörter - SQL (6)

ROWID[1]
ROWNUM[1]
ROWS[1,2]
RULE[3]
— **S** —
SAVE[3]
SCHEMA[2,3]
SCROLL[2]
SECOND[2]
SECTION[2]
SELECT[1,2,3]
SERIALIZABLE[3]
SESSION[1,2]
SESSION_USER[2,3]
SET[1,2,3]
SETUSER[3]
SHARE[1]
SHUTDOWN[3]
SIZE[1,2]
SMALLINT[1,2]
SOME[2,3]
SQL[2]
SQLCODE[2]
SQLERROR[2]
SQLSTATE[2]
START[1]
STATISTICS[3]
SUBSTRING[2]
SUCCESSFUL[1]
SUM[2,3]
SYNONYM[1]
SYSDATE[1]
SYSTEM_USER[2,3]
— **T** —
TABLE[1,2,3]
TAPE[3]
TEMP[3]
TEMPORARY[2,3]
TEXTSIZE[3]
THEN[1,2,3]
TIME[2]
TIMESTAMP[2]
TIMEZONE_HOUR[2]
TIMEZONE_MINUTE[2]
TO[1,2,3]
TOP[3]
TRAILING[2]
TRAN[3]
TRANSACTION[2,3]
TRANSLATE[2]
TRANSLATION[2]
TRIGGER[1,3]
TRIM[2]
TRUE[2]
TRUNCATE[3]
TSEQUAL[3]

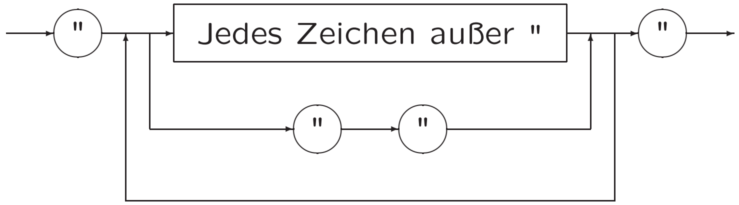
Reservierte Wörter - SQL (7)

— **U** —
UID[1]
UNCOMMITTED[3]
UNION[1,2,3]
UNIQUE[1,2,3]
UNKNOWN[2]
UPDATE[1,2,3]
UPDATETEXT[3]
UPPER[2]
USAGE[2]
USE[3]
USER[1,2,3]
USING[2]
— **V** —
VALIDATE[1]
VALUE[2]
VALUES[1,2,3]
VARCHAR[1,2]
VARCHAR2[1]
VARYING[2,3]
VIEW[1,2,3]
— **W** —
WAITFOR[3]
WHEN[2,3]
WHENEVER[1,2]
WHERE[1,2,3]
WHILE[3]
WITH[1,2,3]
WORK[2,3]
WRITE[2]
WRITETEXT[3]
— **Y** —
YEAR[2]
— **Z** —
ZONE[2]

Delimited Identifier (1)

- Es ist möglich, jede Zeichenfolge in Anführungszeichen als Bezeichner zu verwenden, z.B. "id, 2!".

  Solche Bezeichner sind case-sensitive und es gibt keine Konflikte mit reservierten Wörtern. SQL-86 enthält dies nicht. In Deutsch würde “Delimited Identifier” in etwa “Abgegrenzte Bezeichner” heißen.



Delimited Identifier (2)

- Delimited Identifier sind keine String-Konstanten! Solche haben die Form '...'.

  SQL Server akzeptiert ' und " für String-Konstanten und nimmt [...] für Delimited Identifier. “SET QUOTED_IDENTIFIER ON” schaltet auf den SQL-92-Standard um (aber Delimited Identifier sind nicht case-sensitive). Access versteht [...] und ‘...‘ für Delimited Identifier, schließt aber !.‘[]" und Leerzeichen am Anfang aus.

- Wenn man z.B. in Oracle schreibt:

  `SELECT * FROM STUDENTEN WHERE VORNAME = "Lisa"`

  Fehler: `"Lisa"` ist ein ungültiger Spaltenname.

  Delimited Identifier werden normalerweise nur verwendet, um ausgegebene Spaltennamen umzubenennen (oder wenn Spaltennamen in einer neuen DBMS-Version zu reservierten Wörtern werden).

Delimited Identifier (3)

- Delimited Identifier werden hauptsächlich verwendet, um Ausgabe-Spalten umzubenennen, z.B.

```
SELECT VORNAME AS "Vorname", NACHNAME "Name"
FROM   STUDENTEN
```

  "AS" ist optional (außer in MS Access).

- Ist aber der neue Spaltenname ein legaler Bezeichner, sind die Anführungszeichen unnötig:

```
SELECT VORNAME AS V_NAME, NACHNAME Name
FROM   STUDENTEN
```

- In Oracle wird alles in Großbuchstaben ausgegeben.

Lexikalische Fehler

- Anführungszeichen, z.B. "Lisa", für String-Literale verwenden (Delimited Identifier, kein String).

  Manche Systeme erlauben "...", aber das verletzt den Standard.

- Hochkommas für Zahlen verwenden, z.B. '123'.

  Das sollte einen Typfehler geben. Das DBMS könnte jedoch einfach den Typ von einem der Operanden konvertieren. Da < usw. für Strings und Zahlen anders definiert ist, kann dies gefährlich sein und sollte vermieden werden. Z.B. '12' < '3'.

- Reservierte Wörter als Tabellen-, Spalten- oder Tupelvariablennamen verwenden.

  Die Fehlermeldung könnte seltsam sein (nicht verständlich). Daher sollte man diese Möglichkeit im Auge behalten.

SQL-Anfragen: Begrenzung

- In Oracle SQL*Plus muß jede SQL-Anweisung mit einem Semikolon ";" abgeschlossen werden.

  Da SQL-Statements über mehrere Zeilen gehen können, ist dies notwendig, damit SQL*Plus weiß, wann das SQL-Statement beendet ist. Auch wenn SQL in C-Programme eingebettet ist, wird das Semikolon als Begrenzer verwendet.

- Aber eigentlich gehört das Semikolon nicht mit zur SQL-Anweisung.

  Z.B. ist in dem Anfrage-Analyse-Fenster von MS SQL Server kein Semikolon erforderlich. Es könnte sogar ein Fehler sein, wie im Kommandozeilen-Interface von DB2. Auch wenn SQL-Statements als Strings an Prozeduren übermittelt werden, wie z.B. in ODBC, ist kein Semikolon erforderlich.

Inhalt

Beispiel-Datenbank

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

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

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

Tupelvariablen (1)

- In der FROM-Klausel deklariert man Variablen, die über alle Tupel (Zeilen) der Relation laufen:

```
SELECT A.ANR, A.THEMA
FROM   AUFGABEN A
WHERE  A.ATYP = 'H'
```

- "A" steht nacheinander für jede Zeile der Tabelle.

  Weil das Tupel sich ändert, für das A steht, heißt A (Tupel-)Variable.

- Auf die Werte der Spalten in der gerade betrachteten Zeile kann man sich mit A.ATYP u.s.w. beziehen.

  Es gibt eine Abkürzung, bei der man die Tupelvariable weglassen kann. Dies wurde in Kapitel 2 verwendet, und wird unten näher erläutert.

Tupelvariablen (2)

- Z.B. könnte A zuerst für die erste Zeile der Tabelle stehen (die genaue Reihenfolge darf das DBMS wählen, da Relationen Mengen sind):

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

- Nun kann man für Ausdrücke wie A.ATYP jeweils den konkreten Wert aus der aktuellen (gerade betrachteten) Zeile einsetzen, im Beispiel also 'H'.

Tupelvariablen (3)

- Anfrage und aktuelle Zeile (nochmal):

```
SELECT A.ANR, A.THEMA
FROM   AUFGABEN A
WHERE  A.ATYP = 'H'
```

	ATYP	ANR	THEMA	MAXPT
A ⟶	H	1	ER	10

- Die WHERE-Bedingung ist für diese Zeile wahr.
- Daher werden die in der SELECT-Klausel angegebenen Werte ausgegeben:

ANR	THEMA
1	ER

Tupelvariablen (4)

- Anschließend wird die Tupelvariable `A` eine Zeile weiter geschaltet:

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

- Die `WHERE`-Bedingung `A.ATYP = 'H'` ist wieder wahr.
- Also erfolgt wieder die Ausgabe der `SELECT`-Daten:

ANR	THEMA
2	SQL

Tupelvariablen (5)

- Dann wird die Tupelvariable `A` noch eine Zeile weiter geschaltet:

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

- Die `WHERE`-Bedingung `A.ATYP = 'H'` ist diesmal nicht erfüllt (falsch).
- In diesem Fall wird nichts ausgegeben.

Tupelvariablen (6)

- Insgesamt ergibt die Anfrage für die geg. Tabelle

```
SELECT A.ANR, A.THEMA
FROM   AUFGABEN A
WHERE  A.ATYP = 'H'
```

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

also das Ergebnis:

ANR	THEMA
1	ER
2	SQL

Tupelvariablen (7)

Anmerkung für Hörer mit Programmierkenntnissen:

- Man kann sich die interne Auswertung der Anfrage:

```
SELECT A.ANR, A.THEMA
FROM   AUFGABEN A
WHERE  A.ATYP = 'H'
```

also etwa wie folgendes Programm (in einer hypothetischen Programmiersprache) vorstellen:

```
for A in AUFGABEN do
    if A.ATYP = 'H' then
        print A.ANR, A.THEMA
```

Abkürzungen (1)

- Man kann die Anfrage kürzer schreiben (s. Kap. 2):

```
SELECT ANR, THEMA
FROM   AUFGABEN
WHERE  ATYP = 'H'
```

- Dies macht Gebrauch von zwei Abkürzungen, die unabhängig von einander sind:
  - ◇ Statt `T.S` (Tupelvariable.Spalte) kann man auch einfach `S` (nur den Namen der Spalte) schreiben, wenn es nur eine passende Tupelvariable gibt.
  - ◇ Wenn man nicht explizit eine Tupelvariable benennt, heißt sie so wie die Tabelle.

Abkürzungen (2)

- Wenn unter `FROM` nur eine Tupelvariable deklariert ist, ist die eindeutige Zuordnung natürlich gegeben:

```
SELECT ANR, THEMA
FROM   AUFGABEN A
WHERE  ATYP = 'H'
```

- Man kann die Notation "Tupelvariable.Spalte" und nur "Spalte" auch in einer Anfrage mischen.
- Nur die unter `FROM` aufgezählten Tabellen werden für die Eindeutigkeit berücksichtigt.

  Im Beispiel ist es kein Problem, daß `ATYP` und `ANR` auch in `BEWERTUNGEN` vorkommen. Interessant wird es erst bei mehreren Tupelvariablen, s.u.

Abkürzungen (3)

- Eine Tupelvariable wird immer erstellt: Ist kein Name angegeben, erhält sie den Namen der Relation:

```
SELECT AUFGABEN.ANR, AUFGABEN.THEMA
FROM   AUFGABEN
WHERE  AUFGABEN.ATYP = 'H'
```

- Wenn man also nur `FROM AUFGABEN` schreibt, wird dies behandelt wie:

```
FROM   AUFGABEN AUFGABEN
```

  (Die Tupelvariable namens "`AUFGABEN`" läuft über alle Zeilen der Tabelle "`AUFGABEN`".)

Abkürzungen (4)

- Wird ein Tupelvariablen-Name angegeben, z.B.

```
FROM  AUFGABEN A
```

  so ist es ein Fehler, "`AUFGABEN.ANR`" zu schreiben.

  Die Tupelvariable heißt nun "`A`", nicht "`AUFGABEN`".

- Die Abkürzungen sind so gemacht, daß man nicht unbedingt "in Tupelvariablen denken" muß.

  Ich persönlich halte es allerdings für ein nützliches Konzept.

- Wenn man will, kann man auch nur in Tabellen und Spalten denken, und bei "`FROM AUFGABEN A`" sagen, daß "`A`" ein Alias für die Tabelle "`AUFGABEN`" ist.

Bemerkung zu Namen

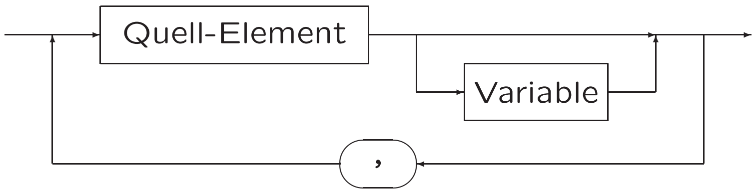
- Für Tupelvariablen sind beliebige Bezeichner (Folie 4-26) möglich, nicht nur einzelne Buchstaben.
- Man sollte Anfragen möglichst lesbar schreiben, besonders wenn man sie nicht nur ein Mal ausführt.

  Z.B. bei Anfragen in Skripten, Programmen, Sicht-Deklarationen.
- Dazu können auch gut gewählte Bezeichner helfen.

  Andererseits sind viele Anfragen sehr kurz, und dann bringt der längere Bezeichner oft keine Verbesserung. Bei Programmen sind einbuchstabige Bezeicher verpönt, es sei denn, man kann ihre Bedeutung mit einem Blick erfassen (z.B. bei Laufvariablen in kurzen Schleifen).

FROM-Syntax (1)

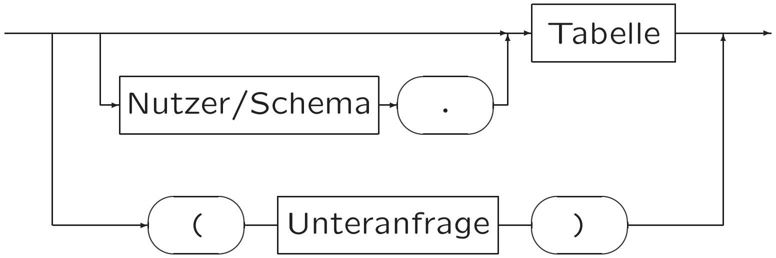
Quell-Liste (nach FROM):



- In SQL-92, SQL Server, Access, DB2 und MySQL (nicht in Oracle 8i) kann man "`AS`" zwischen Quell-Element und Variable schreiben.
- In SQL-92, DB2 (nicht Oracle, SQL Server, Access, MySQL) kann man neue Spaltennamen definieren: "`STUDENTEN AS S(NR,VNAME,NNAME,MAIL)`".
- Ist das "Quell-Element" eine Unteranfrage, wird in SQL-92, SQL Server und DB2 eine Tupelvariable verlangt (aber nicht Oracle, Access). Dann funktioniert obige Spaltenumbenennung plötzlich auch in SQL Server.
- SQL-92, SQL Server, Access, DB2 unterstützen Joins unter `FROM` (später).

FROM-Syntax (2)

Quell-Element:



- SQL-86 erlaubt keine Unteranfragen in der `FROM`-Liste.
- MySQL unterstützt überhaupt keine Unteranfragen.
- Vereinfachte Syntax der `FROM`-Klausel:

```
FROM Tabelle [Variable], ..., Tabelle [Variable]
```


FROM-Syntax (3)

Tabellennamen:

- Man kann sich auf Tabellen anderer Nutzer unter `FROM` beziehen (falls Leserecht erteilt wurde):

```
SELECT * FROM BRASS.AUFGABEN
```

- Der Nutzername ist hier der Name des DB-Schemas (ein DBMS kann mehrere Schemata verwalten).

  In Oracle sind Nutzer und Schema mehr oder weniger das gleiche: Jeder Nutzer hat sein eigenes Schema, jedes Schema gehört genau einem Nutzer. In DB2 kann es mehrere Schemata je Nutzer geben (man kann "Schema.Tabelle" schreiben). In SQL Server hat ein vollständiger Name die Form "Server.DB.Inhaber.Tabelle", aber es gibt viele Abkürzungen, z.B. "Inhaber.Tabelle" oder "Tabelle". In MySQL kann man "DB.Tabelle" schreiben.

Terme (1)

- Ein Term ist jeder Teil einer SQL-Anfrage, der sich zu einem Datenwert (Element eines Datentyps, z.B. Zeichenkette, Zahl) auswerten läßt.

  Statt "Term" (in der Logik üblich) sagt man auch "Ausdruck" bzw. "Wertausdruck" (in Programmiersprachen üblich), oder auch "skalarer Ausdruck" (im SQL-Standard, dort im Unterschied zu "Tabellenausdrücken"). Englisch: "expression".

- Terme sind (u.a.):
  - ◇ Spaltenzugriffe, z.B. `S.SID`.
  - ◇ Konstanten/Literale, z.B. `'Lisa'`.
  - ◇ Zusammengesetzte/geschachtelte Terme, z.B. `(B.PUNKTE/A.MAXPT)*100`.

Terme (2)

- Terme können aus kleineren Termen zusammengesetzt werden durch
  - ◇ die vier Grundrechenarten +, -, *, / (für Zahlen),
  - ◇ || (String-Konkatenation)
  - ◇ weitere Datentyp-Funktionen (siehe Kapitel 8), z.B. `UPPER`, `ROUND`.
- Die Argumente (Eingabewerte) eines Operators (wie "||") oder einer Funktion wie "`UPPER`" müssen den richtigen Datentyp haben (im Beispiel Zeichenketten sein, keine Zahlen).

Terme (3)

- Terme verwendet man in Bedingungen, z.B. enthält

  `B.PUNKTE > A.MAXPT * 0.8`

  die Terme "`B.PUNKTE`" und "`A.MAXPT * 0.8`".
- Auch `SELECT`-Liste kann beliebige Terme enthalten:

```
SELECT NACHNAME || ', ' || VORNAME
FROM   STUDENTEN
```

...
Weiss, Lisa
Grau, Michael
Sommer, Daniel
Winter, Iris

Zusammengesetzte Terme (1)

- Der SQL-86-Standard enthielt nur +, -, *, /.
- Derzeitige DBMS unterscheiden sich immer noch in anderen Datentyp-Operationen.

  Aber sie haben meist eine große Auswahl an Datentyp-Operationen, z.B. `sin`, `cos`, `substr`. In Kapitel 8 enthält Listen von Datentyp-Operationen für verschiedene Systeme.

- Z.B. ist der Operator || im SQL-92-Standard enthalten, aber funktioniert z.B. nicht in SQL Server.

  String-Konkatenation wird in SQL Server und Access "+" geschrieben. In MySQL muß man "`concat(`s_1`, `s_2`)`" schreiben (aber es gibt `--ansi`). Andere Datentyp-Funktionen (z.B. `SUBSTR`) sind sogar noch weniger standardisiert.

Zusammengesetzte Terme (2)

- SQL kennt die übliche Vorrangregel "Punktrechnung vor Strichrechnung", z.B. bedeutet `A+B*C`:

  `A+(B*C)`,

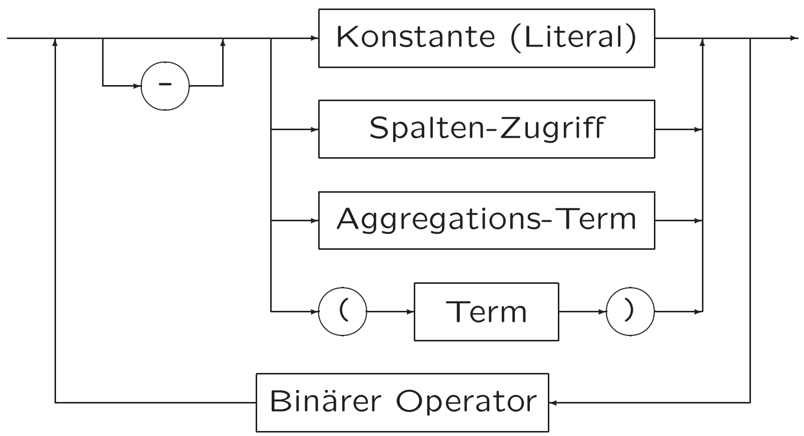
  und nicht

  `(A+B)*C`.
- Klammern `(...)` können verwendet werden, um eine bestimmte Struktur zu erzwingen.
- Übung: Was ist das Ergebnis von `7+3*2-4-1`?

  Es kann nützlich sein, einen Operator-Baum zu zeichnen.
  "-" ist links-assoziativ (von links ausgewertet).

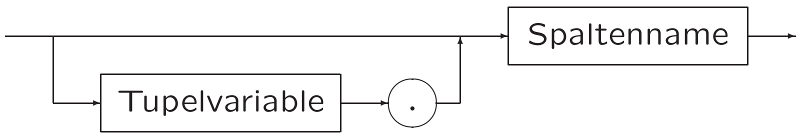
Terme: Syntax (1)

Term (Skalarer Ausdruck, Wert-Ausdruck):



Terme: Syntax (2)

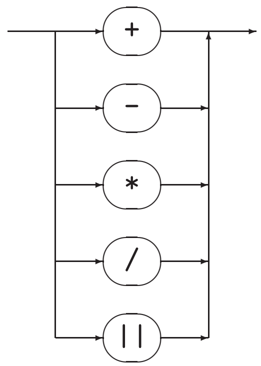
Spalten-Zugriff:



- Die Tupelvariable muß unter `FROM` deklariert sein (eventuell implizit der unter `FROM` angegebene Tabellenname).
- Die Relation, über der die Tupelvariable läuft, muß eine Spalte mit dem Namen haben.
- Die Tupelvariable darf weggelassen werden, wenn nur eine Tupelvariable deklariert ist, die über einer Relation mit einer solchen Spalte läuft.

Terme: Syntax (3)

Binärer Operator:



- SQL Server, Access, MySQL verwenden nicht "||" für die Konkatenation.

Atomare Formeln (1)

- Bedingungen in SQL (u.a. in der `WHERE`-Klausel) sind logische Formeln.
- Logische Formeln bestehen aus atomaren Formeln, die mit "und", "oder", "nicht" verknüpft werden (in SQL: `AND`, `OR`, `NOT`).

  Tatsächlich gibt es in der Logik auch noch Quantoren: "für alle X gilt" und "es gibt ein X mit". Das ist in SQL etwas anders gelöst (mit Unteranfragen, die aber im wesentlichen dem "es gibt" entsprechen, siehe Kapitel 5).
- Die wichtigste Art von atomarer Formel ist der Vergleich, z.B. `SID = 101` oder `PUNKTE > 8`.

Atomare Formeln (2)

Atomare Formel (Form 1):

→ Term → Vergleichs-Op. → Term →

- Vergleichsoperatoren: `=`, `<>`, `<`, `>`, `<=`, `>=`.
- Die beiden Terme sollten kompatible Datentypen haben (beides Zahlen oder beides Zeichenketten).

  Dies sowie weitere Details zum Zeichenkettenvergleich werden ab Folie 4-135 genauer erläutert.
- SQL hat eine ganze Reihe weiterer Formen von atomaren Formeln (u.a. nützliche Abkürzungen), s.u.

Bedingungen (1)

- Aus den atomaren Formeln kann man nun komplexere Formeln (Bedingungen) aufbauen, indem man
  - zwei Formeln mit `AND` ("und") verknüpft, z.B. `ATYP = 'H' AND ANR = 1`
  - zwei Formeln mit `OR` ("oder") verknüpft, z.B. `ANR = 1 OR ANR = 2`
  - eine Formel mit `NOT` ("nicht") negiert, z.B. `NOT ATYP = 'H'`
- Da das Ergebnis wieder eine Formel ist, kann man daraus dann noch komplexere Formeln aufbauen.

Bedingungen (2)

- Die `WHERE`-Bedingung wird für jede Kombination von Zeilen der unter `FROM` stehenden Tabellen ausgewertet. Ist sie wahr, wird die `SELECT`-Liste ausgegeben.
- Eine `AND`-Bed. ist wahr, wenn beide Teile wahr sind, eine `OR`-Bed. ist wahr, wenn ein Teil wahr ist:

B1	B2	B1 and B2	B1 or B2	not B1
falsch	falsch	falsch	falsch	wahr
falsch	wahr	falsch	wahr	wahr
wahr	falsch	falsch	wahr	falsch
wahr	wahr	wahr	wahr	falsch

Bedingungen (3)

- "Geben Sie die SIDs von Lisa und Iris aus".

```
SELECT SID
FROM   STUDENTEN        Falsch!
WHERE  VORNAME = 'Lisa' AND VORNAME = 'Iris'
```

STUDENTEN					
SID	VORNAME	NACHNAME	VORN.='Lisa'	VORN.='Iris'	WHERE
101	Lisa	Weiss	wahr	falsch	falsch
102	Michael	Grau	falsch	falsch	falsch
103	Daniel	Sommer	falsch	falsch	falsch
104	Iris	Winter	falsch	wahr	falsch

- Die Bedingung ist immer falsch (inkonsistent). Hier muß "OR" verwendet werden, nicht "AND".

Bedingungen (4)

- Um die syntaktische Struktur komplexerer Formeln eindeutig zu machen, gibt es für AND, OR, NOT wie bei "+" und "∗ Vorrangregeln:
  - ◇ NOT hat höchste Priorität,
  - ◇ AND hat mittlere Priorität (wie "∗"),
  - ◇ OR hat niedrigste Priorität (wie "+").
- Weil AND stärker als OR bindet, wird

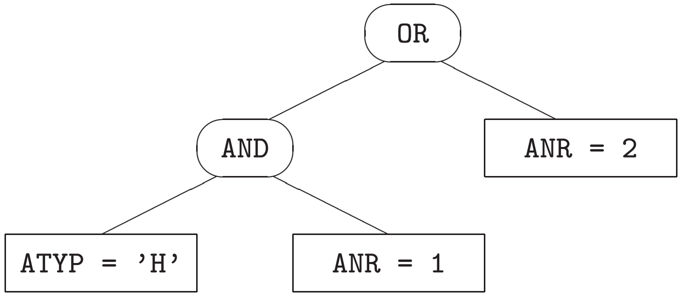
  `ATYP = 'H' AND ANR = 1 OR ANR = 2`

  implizit so geklammert:

  `(ATYP = 'H' AND ANR = 1) OR ANR = 2`

Bedingungen (5)

- Es kann helfen, komplexe Bedingungen oder Terme als "Operator-Baum" darzustellen:



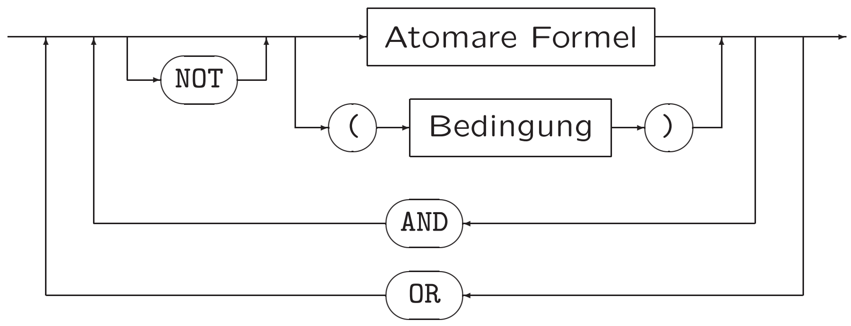
Bedingungen (6)

- Im Beispiel ist diese implizite Struktur vermutlich nicht erwünscht.
- Klammern (...) können verwendet werden, um eine bestimmte Struktur zu erzwingen.
- Manchmal ist es deutlicher Klammern zu verwenden, auch wenn sie nicht nötig wären, um die richtige Struktur der Bedingung zu erhalten.

  Anfänger neigen jedoch dazu, viele Klammern zu verwenden (wahrscheinlich weil sie sich über die Bindungsstärken nicht sicher sind). Das macht die Formel nicht verständlicher.

Bedingungen (7)

Bedingung:



- SQL-92 erlaubt "`IS NOT TRUE`", "`IS FALSE`" usw. nach Formeln (nicht in Oracle 8.0, SQL Server, DB2, MySQL, Access unterstützt).

Bedingungen (8)

- `AND` und `OR` müssen auf beiden Seiten vollständige logische Bedingungen haben (etwas, das wahr oder falsch ist).
- Somit ist Folgendes ein Syntaxfehler, obwohl es in der natürlichen Sprache erlaubt wäre:

```
SELECT DISTINCT SID        Falsch!
FROM   BEWERTUNGEN
WHERE  ATYP = 'H' AND PUNKTE >= 9
AND    ANR = 1 OR 2
```

- Ausnahme: ... `BETWEEN` ... `AND` ...

  Hier bezeichnet das Wort `AND` keinen logischen Operator.

Inhalt

Beispiel-Datenbank

STUDENTEN

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

BEWERTUNGEN

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

AUFGABEN

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

Verbunde/Joins (1)

- Wenn man Daten aus mehreren Tabellen verknüpft, spricht man von einem Verbund (eng. Join).

  Der Verbund ist eine Operation der relationalen Algebra (RA), einer theoretischen Anfragesprache. Man redet aber immer (nicht nur in der RA) von einem Verbund, wenn man Tabellen über eine Bedingung verknüpft. Diese Bedingung heißt dann "Verbund-Bedingung".

- Folgende Anfrage enthält einen Verbund der Tabellen STUDENTEN und BEWERTUNGEN:

```
SELECT B.ATYP, B.ANR, B.PUNKTE
FROM   STUDENTEN S, BEWERTUNGEN B
WHERE  S.SID = B.SID    -- Verbund-Bedingung
AND    S.VORNAME = 'Lisa'
AND    S.NACHNAME = 'Weiss'
```


Verbunde/Joins (2)

- Die obige Anfrage deklariert zwei Tupelvariablen:
  - ◇ S soll über die 4 Tupel in STUDENTEN laufen, und
  - ◇ B über die 8 Tupel in BEWERTUNGEN.
- Im Prinzip werden alle $4 * 8 = 32$ Kombinationen betrachtet, und jeweils die WHERE-Bedingung ausgewertet. Ist sie wahr, wird die SELECT-Liste gedruckt.

  Man hält z.B. zuerst das erste Tupel von STUDENTEN fest (als S) und läßt B über alle Tupel in BEWERTUNGEN laufen. Dann macht man das gleiche für das zweite Tupel in STUDENTEN, u.s.w. Die genaue Reihenfolge ist natürlich nicht vorgeschrieben. Das DBMS kann auch anders vorgehen. Es braucht z.B. Kombinationen nicht zu betrachten, wenn es schon weiß, daß sie die WHERE-Bedingung nicht erfüllen können.

Verbunde/Joins (3)

Anmerkung für Hörer mit Programmierkenntnissen:

- Man kann sich die interne Auswertung einer Anfrage mit zwei Tupelvariablen

  SELECT $A_1, \ldots, A_n$
  FROM STUDENTEN S, BEWERTUNGEN B
  WHERE C

  wie folgendes Programmstück mit zwei geschachtelten Schleifen vorstellen:

  for S **in** STUDENTEN **do**
      for B **in** BEWERTUNGEN **do**
          if C **then print** $A_1, \ldots, A_n$

Verbunde/Joins (4)

- Hier nochmal die gegebene Anfrage:

```
SELECT B.ATYP, B.ANR, B.PUNKTE
FROM   STUDENTEN S, BEWERTUNGEN B
WHERE  S.SID = B.SID
AND    S.VORNAME = 'Lisa'
AND    S.NACHNAME = 'Weiss'
```

- Eine Variablenbelegung legt konkrete Werte (hier Zeilen/Tupel) für Variablen (hier S und B) fest.

  Für eine gegebene Variablenbelegung kann die WHERE-Bedingung ausgewertet werden (zu wahr oder falsch). Im Beispiel ist sie nur dann wahr, wenn alle drei Teilbedingungen wahr sind (wegen der Verknüpfung mit AND). Ist dies erfüllt, werden die in der SELECT-Liste angegebenen Daten (für diese Variablenbelegung) ausgegeben.

Verbunde/Joins (5)

	STUDENTEN			
	SID	VORNAME	NACHNAME	EMAIL
S ⟶	101	Lisa	Weiss	···
	102	Michael	Grau	NULL
	⋮	⋮	⋮	⋮

	BEWERTUNGEN			
	SID	ATYP	ANR	PUNKTE
B ⟶	101	H	1	10
	101	H	2	8
	101	Z	1	12
	102	H	1	9
	⋮	⋮	⋮	⋮

```
S.SID = B.SID
(wahr)

S.VORNAME
    = 'Lisa'
(wahr)

S.NACHNAME
    = 'Weiss'
(wahr)

also drucken:
B.ATYP:   'H'
B.ANR:     1
B.PUNKTE: 10
```


Verbunde/Joins (6)

	STUDENTEN			
	SID	VORNAME	NACHNAME	EMAIL
S ⟶	101	Lisa	Weiss	···
	102	Michael	Grau	NULL
	⋮	⋮	⋮	⋮

	BEWERTUNGEN			
	SID	ATYP	ANR	PUNKTE
	101	H	1	10
B ⟶	101	H	2	8
	101	Z	1	12
	102	H	1	9
	⋮	⋮	⋮	⋮

```
S.SID = B.SID
(wahr)

S.VORNAME
    = 'Lisa'
(wahr)

S.NACHNAME
    = 'Weiss'
(wahr)

also drucken:
B.ATYP:   'H'
B.ANR:     2
B.PUNKTE:  8
```


Verbunde/Joins (7)

	STUDENTEN			
	SID	VORNAME	NACHNAME	EMAIL
S ⟶	101	Lisa	Weiss	···
	102	Michael	Grau	NULL
	⋮	⋮	⋮	⋮

	BEWERTUNGEN			
	SID	ATYP	ANR	PUNKTE
	101	H	1	10
	101	H	2	8
B ⟶	101	Z	1	12
	102	H	1	9
	⋮	⋮	⋮	⋮

```
S.SID = B.SID
(wahr)

S.VORNAME
    = 'Lisa'
(wahr)

S.NACHNAME
    = 'Weiss'
(wahr)

also drucken:
B.ATYP:   'Z'
B.ANR:     1
B.PUNKTE: 12
```


Verbunde/Joins (8)

	STUDENTEN			
	SID	VORNAME	NACHNAME	EMAIL
S ⟶	101	Lisa	Weiss	···
	102	Michael	Grau	NULL
	⋮	⋮	⋮	⋮

	BEWERTUNGEN			
	SID	ATYP	ANR	PUNKTE
	101	H	1	10
	101	H	2	8
	101	Z	1	12
B ⟶	102	H	1	9
	⋮	⋮	⋮	⋮

```
S.SID = B.SID
(falsch)

S.VORNAME
    = 'Lisa'
(wahr)

S.NACHNAME
    = 'Weiss'
(wahr)

nichts
drucken
(Bedingung
 falsch)
```

Verbunde/Joins (9)

	STUDENTEN			
	SID	VORNAME	NACHNAME	EMAIL
	101	Lisa	Weiss	···
S ⟶	102	Michael	Grau	NULL
	⋮	⋮	⋮	⋮

	BEWERTUNGEN			
	SID	ATYP	ANR	PUNKTE
B ⟶	101	H	1	10
	101	H	2	8
	101	Z	1	12
	102	H	1	9
	⋮	⋮	⋮	⋮

S.SID = B.SID
(falsch)

S.VORNAME
= 'Lisa'
(falsch)

S.NACHNAME
= 'Weiss'
(falsch)

nichts
drucken

Verbunde/Joins (10)

	STUDENTEN			
	SID	VORNAME	NACHNAME	EMAIL
	101	Lisa	Weiss	···
S ⟶	102	Michael	Grau	NULL
	⋮	⋮	⋮	⋮

	BEWERTUNGEN			
	SID	ATYP	ANR	PUNKTE
	101	H	1	10
B ⟶	101	H	2	8
	101	Z	1	12
	102	H	1	9
	⋮	⋮	⋮	⋮

S.SID = B.SID
(falsch)

S.VORNAME
= 'Lisa'
(falsch)

S.NACHNAME
= 'Weiss'
(falsch)

nichts
drucken

Verbunde/Joins (11)

	STUDENTEN			
	SID	VORNAME	NACHNAME	EMAIL
	101	Lisa	Weiss	···
S ⟶	102	Michael	Grau	NULL
	⋮	⋮	⋮	⋮

	BEWERTUNGEN			
	SID	ATYP	ANR	PUNKTE
	101	H	1	10
	101	H	2	8
B ⟶	101	Z	1	12
	102	H	1	9
	⋮	⋮	⋮	⋮

S.SID = B.SID
(falsch)

S.VORNAME
= 'Lisa'
(falsch)

S.NACHNAME
= 'Weiss'
(falsch)

nichts
drucken

Verbunde/Joins (12)

	STUDENTEN			
	SID	VORNAME	NACHNAME	EMAIL
	101	Lisa	Weiss	···
S ⟶	102	Michael	Grau	NULL
	⋮	⋮	⋮	⋮

	BEWERTUNGEN			
	SID	ATYP	ANR	PUNKTE
	101	H	1	10
	101	H	2	8
	101	Z	1	12
B ⟶	102	H	1	9
	⋮	⋮	⋮	⋮

S.SID = B.SID
(wahr)

S.VORNAME
= 'Lisa'
(falsch)

S.NACHNAME
= 'Weiss'
(falsch)

nichts
drucken

Verbunde/Joins (13)

- Das waren 8 verschiedene Variablenbelegungen. Nur für drei davon war die `WHERE`-Bedingung erfüllt.

  Für die übrigen $32-8=24$ möglichen Variablenbelegungen ist die Bedingung auch nicht erfüllt. Das DBMS versucht natürlich, möglichst wenig Belegungen wirklich zu betrachten. Dabei darf sich das Ergebnis aber nicht verändern, es müssen also mindestens alle Belegungen mit wahrer Bedingung betrachtet werden.

- Insgesamt liefert die Anfrage also alle Bewertungen für die Studentin Lisa Weiss:

ATYP	ANR	PUNKTE
H	1	10
H	2	8
Z	1	12

Verbund-Bedingungen (1)

- Die Verbund-Bedingung `S.SID = B.SID` muß explizit unter `WHERE` mit angegeben werden.
- Sonst werden auch Tupel kombiniert, die sich auf verschiedene Studierende beziehen.
- Übung: Was wäre das Ergebnis dieser Anfrage?

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

Falsch!

Verbund-Bedingungen (2)

- Es ist fast immer ein Fehler, wenn es zwei Tupelvariablen gibt, die nicht durch Verbund-Bedingungen verknüpft sind (eventuell indirekt).

  Oder es sind konstante Werte für die Verbund-Attribute gefordert. Der Verbund kann eventuell auch in einer Unteranfrage geschehen.

- Hier sind alle drei Tupelvariablen verbunden:

```
SELECT A.ATYP, A.ANR, B.PUNKTE, A.MAXPT
FROM   STUDENTEN S, BEWERTUNGEN B, AUFGABEN A
WHERE  S.SID = B.SID
AND    B.ATYP = A.ATYP AND B.ANR = A.ANR
AND    S.VORNAME = 'Lisa'
AND    S.NACHNAME = 'Weiss'
```


Verbund-Bedingungen (3)

- Die Tupelvariablen sind wie folgt verbunden:

```
S ——————————————— B ——————————————— A
   S.SID = B.SID     B.ATYP = A.ATYP
                     AND B.ANR = A.ANR
```

- Das entspricht den Schlüssel-Fremdschlüssel-Beziehungen zwischen den Tabellen.
- Wenn man eine Verbund-Bedingung vergisst, wird man oft viele Duplikate erhalten.

  Dann wäre es falsch `DISTINCT` anzuwenden, ohne über den Grund der Duplikate nachzudenken.

Anfrageformulierung (1)

- Aufgabe: “Geben Sie die Themen aller von Lisa Weiss gelösten Aufgaben aus.”
- Lisa Weiss ist eine Studentin, daher sind Tupelvariable `S` über `STUDENTEN` und folgende Bedingung nötig: `S.VORNAME = 'Lisa' AND S.NACHNAME = 'Weiss'`
- Aufgaben-Themen werden verlangt, so daß eine Tupelvariable `A` über `AUFGABEN` benötigt wird. Folgender Teil kann bereits erstellt werden:

```
SELECT DISTINCT A.THEMA
```

“`DISTINCT`”, da viele Aufgaben das gleiche Thema haben können.

Anfrageformulierung (2)

- Schließlich sind `S` und `A` nicht verbunden.
- Es kann helfen, einen Verbindungsgraphen der Tabellen, basierend auf gemeinsamen Spalten (Fremdschlüssel), zu zeichnen:

`STUDENTEN` — `BEWERTUNGEN` — `AUFGABEN`

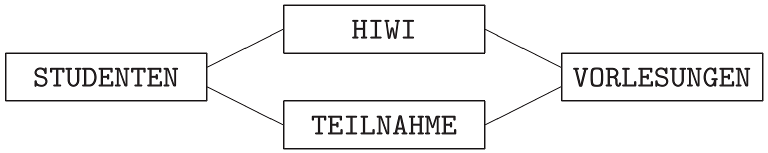
- Man sieht, daß eine Tupelvariable über `BEWERTUNGEN` benötigt wird mit folgender Verbund-Bedingung:

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


Anfrageformulierung (3)

- Es ist nicht immer so einfach. Der Verbindungsgraph kann Zyklen enthalten, die die Wahl des richtigen Pfades erschwert.
- Betrachten Sie z.B. eine Datenbank zur Belegung von Vorlesungen, die auch Hiwis beinhaltet.

  Hiwis sind oft fortgeschrittene Studenten, die bei der Korrektur von Hausaufgaben usw. helfen.



Etwas Logik (1)

- Zwei Anfragen heißen äquivalent genau dann, wenn sie in allen Datenbank-Zuständen beide immer das gleiche Ergebnis liefern.
- Wenn beide Anfragen ungefähr gleich lang (bzw. gleich übersichtlich) sind, spielt es also keine Rolle, welche der Anfragen man verwendet.

  Über Ausführungsgeschwindigkeiten kann man wenig sagen, da das DBMS das Verfahren wählt, und sich dies von einer Version zur nächsten ändern kann. Oft werden die Anfragen gleich schnell sein.
- Z.B. ist es völlig egal, ob man `S.SID = B.SID` oder `B.SID = S.SID` schreibt.

Etwas Logik (2)

- Bedingungen in SQL sind logische Formeln.

  Zwei Formeln sind äquivalent, wenn sie für alle Datenbank-Zustände und alle Belegungen der Tupelvariablen den gleichen Wahrheitswert liefern (also immer beide wahr oder beide falsch sind).

  Wenn sich zwei Anfragen nur in den `WHERE`-Bedingungen unterscheiden, und diese Bedingungen äquivalente Formeln sind, sind natürlich auch die Anfragen äquivalent.

- Es lohnt sich, etwas über Logik zu wissen, weil
  - ◇ man manchmal Anfragen vereinfachen kann,
  - ◇ oder zumindest sich um die Unterschiede zwischen logisch äquivalenten Formulierungen keine Gedanken machen muß.

Etwas Logik (3)

- Z.B. braucht man sich bei einer Kette von Bedingungen, die alle mit `AND` verknüpft sind, keine Gedanken über Reihenfolge und Klammerung machen: Es kommt nur auf die Menge der Bedingungen an.
- Dies folgt aus den Eigenschaften:
  - ◇ X `AND` Y hat immer den gleichen Wahrheitswert wie Y `AND` X (Kommutativität).
  - ◇ $(X$ `AND` $Y)$ `AND` Z hat immer den gleichen Wahrheitswert wie X `AND` $(Y$ `AND` $Z)$ (Assoziativität).
  - ◇ X `AND` X hat den gleichen Wahrheitswert wie X.

Etwas Logik (4)

- Aufgrund der Assoziativität ist es bei `AND`-Ketten nicht üblich, Klammern zu setzen.

  Das gleiche gilt auch für `OR`-Ketten.

- Die Syntaxanalyse im DBMS nimmt die Klammerung von links an, faßt also X `AND` Y `AND` Z als Abkürzung für $(X$ `AND` $Y)$ `AND` Z auf.
- Die Anfrageauswertung im DBMS ist aber weder durch diese implizite Klammerung noch durch explizit gesetzte Klammern eingeschränkt, weil sie Äquivalenzen ausnutzen darf (Ergebnis bleibt gleich).

Etwas Logik (5)

- Für Bedingungen, in denen sowohl `AND` als auch `OR` vorkommen, ist die Klammerung dagegen wichtig.
- Z.B. ist `ATYP = 'H' AND (ANR = 1 OR ANR = 2)` nicht äquivalent zu `(ATYP = 'H' AND ANR = 1) OR ANR = 2`.

  Die erste Formel wäre nur für Hausaufgabe 1 und Hausaufgabe 2 erfüllt, die zweite dagegen Hausaufgabe 1 und eine beliebige Aufgabe 2 (z.B. auch AUfgabe 2 der Zwischenklausur).

- Wenn man keine Klammern setzt, nimmt die Syntaxanalyse die Klammerung um `AND` an.

  Man sagt: "`AND`" bindet stärker als "`OR`", oder "`AND`" hat höhere Priorität als "`OR`". Dies entspricht "Punktrechnung vor Strichrechnung".

Etwas Logik (6)

- Es gilt das Distributivgesetz:
  - X `AND` (Y `OR` Z) ist äquivalent zu (X `AND` Y) `OR` (X `AND` Z).
  - X `OR` (Y `AND` Z) ist äquivalent zu (X `OR` Y) `AND` (X `OR` Z).
- Dies entspricht dem "Ausmultiplizieren".

  Allerdings gilt nur $X*(Y+Z) = (X*Y)+(X*Z)$ (wobei man die Klammern rechts wegen Punktrechnung vor Strichrechnung auch weglassen kann). Es gilt im allgemeinen nicht $X + (Y * Z) = (X + Y) * (X + Z)$ (man setze z.B. für alle drei Variablen 1 ein). Für `AND` und `OR` gelten dagegen beide Varianten.

Etwas Logik (7)

- Eine Formel F ist logische Folgerung aus einer Formel V (der Voraussetzung), wenn
  - immer wenn V erfüllt ist, ist auch F erfüllt.

    D.h. für jeden DB-Zustand und jede Variablenbelegung, für die V wahr ist, ist auch F wahr. Wenn V falsch ist, kann F wahr oder falsch sein (wenn die Voraussetzung nicht erfüllt ist, macht die logische Folgerung keine Aussage).
- Man sagt dann auch, V impliziert F.
- Z.B. impliziert `B.ATYP = A.ATYP AND A.ATYP = 'H'` die Formel `B.ATYP = 'H'` (Transitivität).

Etwas Logik (8)

- Wenn F logische Folgerung von V ist, ist V `AND` F äquivalent zu V.
- Die folgenden Bedingungen sind also äquivalent:
  - `B.ATYP = A.ATYP AND A.ATYP = 'H'`
  - `B.ATYP = A.ATYP AND A.ATYP = 'H' AND B.ATYP = 'H'`
  - `A.ATYP = 'H' AND B.ATYP = 'H'`
- Für die Korrektheit der Anfrage ist es egal, welche der drei Varianten man wählt. Die mittlere Variante ist allerdings unnötig kompliziert (Punktabzug!).

Etwas Logik (9)

- Eigenschaften der Gleichheitsrelation im Überblick:
  - $X = X$ (reflexiv)
  - $X = Y$ genau dann, wenn $Y = X$ (symmetrisch).

    D.h. die beiden Formeln sind äquivalent: Sie haben immer den gleichen Wahrgeitswert.
  - Wenn $X = Y$ und $Y = Z$, dann $X = Z$ (transitiv).
  - Verträglich mit Funktionen und Relationen, z.B. wenn $X = Y$, dann gilt auch $X + 5 = Y + 5$, außerdem sind dann $X < Z$ und $Y < Z$ äquivalent.

    Man kann Gleiches durch Gleiches ersetzen.

Etwas Logik (10)

- Aufgrund der Symmetrie der Gleichheit sind auch die folgenden beiden Bedingungen äquivalent:
  - ◇ `S.VORNAME = 'Lisa'`
  - ◇ `'Lisa' = S.VORNAME`

  Für das DBMS funktioniert beides, aber für den Menschen ist die erste Variante gewohnter.
- Gelegentlich versteht das DBMS eine Äquivalenz nicht, und verpasst damit eine besonders schnelle Auswertungsmöglichkeit.

  Falls ein Index (Datenstruktur für schnellen Zugriff) über `SID` existiert, wird er bei `SID = 101` verwendet, bei `SID - 101 = 0` eventuell nicht.

Etwas Logik (11)

- Die logische Negation `NOT` bindet noch stärker als `AND` und `OR`, wird also immer nur auf die direkt folgende Bedingung angewendet, sofern keine Klammern gesetzt sind.
- Man sieht sie in einfachen Anfragen (ohne Unteranfragen) selten, da sie sich durch logische Umformungen eliminieren läßt. Z.B. gilt:
  - ◇ `NOT` X = Y ist äquivalent zu X <> Y
  - ◇ `NOT` X < Y ist äquivalent zu X >= Y

Etwas Logik (12)

- Mit Hilfe der De Morgan'schen Regeln kann man `NOT` an `AND` und `OR` vorbei schieben, wobei sich der Operator umdreht:
  - ◇ `NOT` (X `AND` Y) ist äquivalent zu (`NOT` X) `OR` (`NOT` Y)
  - ◇ `NOT` (X `OR` Y) ist äquivalent zu (`NOT` X) `AND` (`NOT` Y)
- Die doppelte Negation hebt sich auf:
  - ◇ `NOT` (`NOT` X) ist äquivalent zu X.

Unnötige Verbunde (1)

- Beispiel-Anfrage: Ergebnisse für Hausaufgabe 1:

```
SELECT B.SID, B.PUNKTE
FROM   BEWERTUNGEN B, AUFGABEN A
WHERE  B.ATYP = A.ATYP AND B.ANR = A.ANR
AND    A.ATYP = 'H' AND A.ANR = 1
```

- Die folgende Anfrage liefert immer das gleiche Ergebnis (ist also äquivalent) und ist deutlich kürzer (der Verbund wurde eingespart):

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

Unnötige Verbunde (2)

- Warum kann man sicher sein, daß beide Anfragen immer das gleiche Ergebnis liefern?
- Angenommen, eine Belegung für B und A erfüllt die Bedingung der oberen Anfrage:

```
WHERE B.ATYP = A.ATYP AND B.ANR = A.ANR
AND   A.ATYP = 'H'    AND A.ANR = 1
```

- Aus B.ATYP = A.ATYP und A.ATYP = 'H' folgt B.ATYP = 'H'. Entsprechend erhält man B.ANR = 1.
- Also ist für diese Belegung von B auch die Bedingung der unteren Anfrage erfüllt.

Unnötige Verbunde (3)

- Daher wird jede Ergebniszeile der oberen Anfrage auch von der unteren Anfrage ausgegeben.
- Sei nun umgekehrt die Bedingung der unteren Anfrage für eine Belegung von B erfüllt:

```
WHERE B.ATYP = 'H' AND B.ANR = 1
```

- Wegen der Fremdschlüssel-Bedingung muß es immer eine passende Zeile A in der Tabelle AUFGABEN geben (mit B.ATYP = A.ATYP und B.ANR = A.ANR).
- Damit hat man eine Belegung für A und B, die die Bedingung der oberen Anfrage erfüllt.

Unnötige Verbunde (4)

- Nun muß man sich noch überzeugen, daß sich beide Anfragen auch in eventuellen Duplikaten nicht unterscheiden.
  - ◇ Da (ATYP, ANR) Schlüssel von AUFGABEN ist, gibt es zu jedem Tupel B nur ein passendes Tupel A.

    Tatsächlich kann man in diesem Beispiel zeigen, daß beide Anfragen keine Duplikate liefern.
- Da die untere Anfrage wesentlich kürzer und übersichtlicher ist, sollte man sie verwenden, und nicht die obere (diese enthält einen unnötigen Verbund).

  Die untere Anfrage wird häufig auch schneller ausgeführt.

Unnötige Verbunde (5)

- Was ist das Ergebnis dieser Anfrage?

```
SELECT B.SID, B.PUNKTE
FROM   BEWERTUNGEN B, AUFGABEN A
WHERE  B.ATYP = 'H' AND B.ANR = 1
```

- Unterscheiden sich die folgenden zwei Anfragen?

```
SELECT S.VORNAME, S.NACHNAME
FROM   STUDENTEN S
```

```
SELECT DISTINCT S.VORNAME, S.NACHNAME
FROM   STUDENTEN S, BEWERTUNGEN B
WHERE  S.SID = B.SID
```

Abkürzung: nochmal (1)

- Wie bekannt, kann man Spalten ansprechen mit:
  - ◇ `Variable.Spalte` (geht immer)
  - ◇ `Spalte` (falls nur eine Variable die Spalte hat)
- Z.B. ist diese Anfrage legal:

```
SELECT ATYP, ANR, PUNKTE
FROM   STUDENTEN S, BEWERTUNGEN B
WHERE  S.SID = B.SID
AND    VORNAME = 'Lisa' AND NACHNAME = 'Weiss'
```

"VORNAME" und "NACHNAME" gibt es nur in "S", "ATYP", "ANR" und "PUNKTE" nur in "B". "SID" allein wäre jedoch mehrdeutig, da sowohl "S" als auch "B" ein Attribut mit diesem Namen haben.

Abkürzung: nochmal (2)

- Beispiel (Syntaxfehler):

```
SELECT ANR, SID, PUNKTE, MAXPT        Falsch!
FROM   BEWERTUNGEN B, AUFGABEN A
WHERE  B.ANR = A.ANR
AND    B.ATYP = 'H' AND A.ATYP = 'H'
```

- SQL verlangt, daß der Nutzer festlegt, ob er `B.ANR` oder `A.ANR` unter `SELECT` auswählt, obwohl beide gleich sind, so daß es eigentlich egal wäre.

Die Regel ist rein syntaktisch: Hat mehr als eine Tupelvariable in der FROM-Klausel das Attribut "ANR", darf die Tupelvariable nicht fehlen oder das DBMS (z.B. Oracle) wird den Fehler "ORA-00918: column ambiguously defined" ausgeben. DB2, SQL Server, Access, MySQL sind auch so pedantisch.

Selbstverbund (1)

- Es ist möglich, daß mehr als ein Tupel derselben Relation benötigt wird, um ein bestimmtes Ergebnis zu erhalten.
- Gibt es einen Studenten, der in Hausaufgabe 1 und in Hausaufgabe 2 jeweils 10 Punkte hat?

```
SELECT S.VORNAME, S.NACHNAME
FROM   STUDENTEN S, BEWERTUNGEN H1, BEWERTUNGEN H2
WHERE  S.SID = H1.SID AND S.SID = H2.SID
AND    H1.ATYP = 'H' AND H1.ANR = 1
AND    H2.ATYP = 'H' AND H2.ANR = 2
AND    H1.PUNKTE = 10 AND H2.PUNKTE = 10
```


Selbstverbund (2)

- Studenten, die mind. zwei Aufgaben gelöst haben:

```
SELECT S.VORNAME, S.NACHNAME              Falsch!
FROM   STUDENTEN S, BEWERTUNGEN A1, BEWERTUNGEN A2
WHERE  S.SID = A1.SID AND S.SID = A2.SID
```

- Die Tupelvariablen `A1` und `A2` können auf das gleiche Tupel in `BEWERTUNGEN` zeigen.
- Man muß verlangen, daß sie verschieden sind:

```
WHERE S.SID = A1.SID AND S.SID = A2.SID
AND   (A1.ATYP <> A2.ATYP OR A1.ANR <> A2.ANR)
```

- Man kann dies aber auch mit Aggregationen lösen.

Beispiel: Logischer Fehler (1)

- Folgende Anfrage soll alle Studenten auszugeben, die eine Aufgabe über SQL und eine über ER-Entwurf gelöst haben:

```
SELECT S.VORNAME, S.NACHNAME       Falsch!
FROM   STUDENTEN S, BEWERTUNGEN B,
       AUFGABEN A1, AUFGABEN A2
WHERE  S.SID = B.SID
AND    B.ATYP = A1.ATYP AND B.ANR = A1.ANR
AND    B.ATYP = A2.ATYP AND B.ANR = A2.ANR
AND    A1.THEMA = 'SQL'
AND    A2.THEMA = 'ER'
```


Beispiel: Logischer Fehler (2)

- Wenn man obige Anfrage ausführt, bekommt man keine Fehlermeldung, aber ein leeres Ergebnis.
- Das kann nicht richtig sein, da im Beispielzustand Lisa Weiss und Michael Grau beide (mindestens) eine Aufgabe über SQL und eine über ER-Entwurf gelöst haben.

  Allgemein sollte man nicht zufrieden sein, wenn man keinen Syntaxfehler erhält, sondern prüfen, ob die Anfrage auch logisch richtig ist. Zum Beispiel kann man oft in einem kleinen Beispielzustand das Ergebnis manuell bestimmen. Oder die Bedingung abschwächen, z.B. Studenten, die eine SQL-Aufgabe gelöst haben, und solche, die eine ER-Aufgabe gelöst haben, getrennt bestimmen.

Beispiel: Logischer Fehler (3)

- Im Beispiel enthält die `WHERE`-Klausel u.a.:
  - ◇ `B.ATYP = A1.ATYP` `B.ANR = A1.ANR`
  - ◇ `B.ATYP = A2.ATYP` `B.ANR = A2.ANR`
- Logische Folgerungen daraus sind jeweils:
  - ◇ `A1.ATYP = A2.ATYP` `A1.ANR = A2.ANR`
- Wenn immer die `WHERE`-Bedingung erfüllt ist, gilt also, daß die Zeilen, auf die `A1` und `A2` zeigen, in den Spalten `ATYP` und `ANR` den gleichen Wert haben.
- Diese beiden Spalten bilden aber zusammen einen Schlüssel der Tabelle (Integritätsbedingung).

Beispiel: Logischer Fehler (4)

- Aufgrund der Schlüssel-Bedingung kann es keine zwei verschiedenen Zeilen geben, die in den beiden Spalten `ATYP` und `ANR` übereinstimmen.
- Also müssen `A1` und `A2` auf die gleiche Zeile zeigen, wenn die `WHERE`-Bedingung wahr ist.
- Dann kann aber nicht gleichzeitig auch gelten:
  - ◇ `A1.THEMA = 'SQL'`
  - ◇ `A2.THEMA = 'ER'`
- D.h. die `WHERE`-Bedingung ist immer falsch (in sich widersprüchlich, "inkonsistent").

Beispiel: Logischer Fehler (5)

- Bei inkonsistenten Bedingungen ist die Ausgabe natürlich immer leer.

  Natürlich liegt hier ein logischer (semantischer) Fehler vor: Der Benutzer kann das nicht beabsichtigt haben.

- Die Frage, ob eine gegebene SQL-Anfrage inkonsistent ist, ist unentscheidbar, d.h.
  - ◇ es ist unmöglich, ein Programm zu schreiben, das beliebige Anfragen prüft, und immer nach endlicher Zeit das richtige Ergebnis ausgibt.

    Es gibt also prinzipielle Grenzen für das, was Computer leisten können. Für einen Teil der Anfragen ist eine solche Analyse aber schon möglich.

Beispiel: Logischer Fehler (6)

- Für die korrekte Lösung muß man sich auch auf zwei Zeilen aus der Bewertungs-Tabelle beziehen:

```
SELECT S.VORNAME, S.NACHNAME
FROM   STUDENTEN S,
       BEWERTUNGEN B1, BEWERTUNGEN B2,
       AUFGABEN A1, AUFGABEN A2
WHERE  S.SID = B1.SID AND S.SID = B2.SID
AND    B1.ATYP = A1.ATYP AND B1.ANR = A1.ANR
AND    B2.ATYP = A2.ATYP AND B2.ANR = A2.ANR
AND    A1.THEMA = 'SQL'
AND    A2.THEMA = 'ER'
```


Exkurs: QBE (1)

- Oft hilft es, sich an einem konkreten Beispiel zu überlegen, welche Eingabezeilen man braucht, um eine Ausgabezeile zu erzeugen.
- Es gab eine Anfragesprache namens QBE ("Query By Example"), die auf diesem Prinzip beruhte.
- Während QBE von SQL verdrängt ist, werden seine Ideen zum Teil noch von graphischen Schnittstellen für ungeübte Datenbank-Anwender genutzt.
- Sie sind nützlich für die Anfrageformulierung, selbst wenn man am Ende eine SQL-Anfrage braucht.

Exkurs: QBE (2)

- In QBE trägt man Beispielzeilen in Tabellengerüste ein, wobei man Beispielwerte durch einen vorangestellten "_" von tatsächlich geforderten Werten unterscheidet. "P." markiert Ausgabespalten.

STUDENTEN			
SID	VORNAME	NACHNAME	EMAIL
_101	P.	P.	

BEWERTUNGEN			
SID	ATYP	ANR	PUNKTE
_101	_H	_1	
_101	_Z	_2	

AUFGABEN			
ATYP	ANR	THEMA	MAXPT
_H	_1	SQL	
_Z	_2	ER	

Exkurs: QBE (3)

- Ein Beispielwert wie "_101" ist tatsächlich eine Variable: Das System kann dafür verschiedene Werte einsetzen, z.B. auch 102.

  Diese Variable steht für einen einzelnen Wert, nicht eine Zeile.
- Man darf nur solche Beispielwerte gleich machen, die unbedingt gleich sein müssen.
- Wenn man z.B. bei `ATYP` in beiden Zeilen den Beispielwert "_H" gewählt hätte, müßte dort in einer konkreten Lösung immer der gleiche Wert stehen.

  Eine Variablenbelegung ordnet einer Variablen nur einen Wert zu. Verschiedene Variablen können mit dem gleichen Wert belegt werden.

Exkurs: QBE (4)

- Wenn man eine QBE-Anfrage nach SQL übersetzen will, geht man wie folgt vor:
  - ◇ Man führt eine Tupelvariable für jede Beispielzeile ein.
  - ◇ Tabellenzellen, die mit dem gleichen Beispielwert gefüllt sind, werden gleichgesetzt.

    Bei drei Zellen reichen zwei Bedingungen (Transitivität).
  - ◇ Wenn eine Tabellenzelle mit einem konkreten Wert gefüllt ist (z.B. "ER"), setzt man die Spalte der betreffenden Tupelvariable gleich diesem Wert: `A2.THEMA = 'ER'`.

Exkurs: QBE (5)

- In QBE kann man auch Bedingungen wie "> 8" in die Tabellenzellen schreiben.

  Wenn das noch nicht reicht, gibt es noch eine "Condition Box", in die man wieder allgemeine Formeln schreiben kann.
- Aufgabe: Formulieren Sie folgende Anfrage in QBE: Wer hat mehr als 5 Punkte für Hausaufgabe 1?
- Man kann auch fordern, daß bestimmte Zeilen nicht existieren. Dazu stellt man der Beispielzeile "**not**" voran.

  Die Entsprechung in SQL wird erst im nächsten Kapitel behandelt.

Join-Fehler

- Fehlende Join-Bedingungen (sehr häufig)
- Unnötige Joins (machen Anfrage langsamer)
- Wenn eigentlich mehrere Tupelvariablen über einer Relation benötigt werden, und man "verschmilzt sie", so erhält man oft inkonsistente Bedingungen.
- Duplikate sind oft ein Zeichen für Fehler: Man sollte die Ursache der Duplikate verstehen und nicht einfach `DISTINCT` anwenden, um das Problem zu vermeiden (eigentlich nur zu verdecken).

Inhalt

1. Einführung: SELECT-FROM-WHERE
2. Lexikalische Syntax
3. Tupelvariablen, Terme, Bedingungen
4. Verbunde, etwas Logik
5. Mehr über Vergleiche, weitere Bedingungen
6. SELECT-Klausel, Duplikate

Beispiel-Datenbank

STUDENTEN

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

AUFGABEN

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

BEWERTUNGEN

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

Vergleiche (1)

Atomare Formel (Form 1):

→ Term → Vergleichs-Op. → Term →

- Vergleichsoperatoren: `=`, `<>`, `<`, `>`, `<=`, `>=`.
- Man kann sie sowohl für Zahlen als auch für Strings verwenden, z.B.: `PUNKTE >= 8`, `NACHNAME < ’M’`.
- “Ungleich” wird in SQL als “`<>`” geschrieben.

  Oracle, SQL Server, DB2 und MySQL verstehen auch “`!=`” (Access nicht). “`^=`” funktioniert in Oracle und DB2, aber nicht in SQL Server, Access oder MySQL.

Vergleiche (2)

- Zahlen werden anders verglichen als Zeichenketten, z.B. `3 < 20`, aber `’3’ > ’20’`.

  Strings werden Zeichen für Zeichen verglichen, bis das Ergebnis klar ist. In diesem Fall kommt “3” alphabetisch nach “2”, daher ist der Rest der Zeichenkette nicht wichtig.

- Nach dem SQL-92-Standard ist es falsch, Zeichenketten mit Zahlen zu vergleichen, z.B. `3 > ’20’`.

  Die verglichenen Werte müssen von kompatiblen Datentypen sein: Alle numerischen Typen sind kompatibel und alle String-Typen ebenfalls, aber numerische Typen sind nicht kompatibel mit String-Typen.

Vergleiche (3)

- Vergleiche zwischen Strings und Zahlen sollten vermieden werden (Ergebnis systemabhängig):
  - ◇ In SQL-92, DB2 und Access ist es ein Typfehler.
  - ◇ Oracle, MySQL, SQL Server konvertieren den String in eine Zahl und vergleichen numerisch.

    Hat der String kein numerisches Format, konvertiert ihn MySQL in 0. Z.B. ist `0 = 'abc'` in MySQL wahr. In Oracle und SQL Server erhält man in diesem Fall jedoch einen Fehler. Das kann ein Laufzeitfehler sein, wenn der String ein Spaltenwert ist.
  - ◇ Wird jedoch eine Spalte mit einer Konstanten verglichen, nimmt SQL Server den Spaltentyp.

    Aggregations-Funktionen haben noch höhere Priorität als Spalten.

Zeichenkettenvergleich (1)

- Das Ergebnis eines Vergleichs (=, <>, <, <=, >, >=) zweier Zeichenketten kann vom DBMS abhängen.

  Oder von Einstellungen innerhalb des DBMS.
- Der SQL-92-Standard definiert den Begriff "collation sequences", der Folgendes festlegt:
  - ◇ für jedes Paar X und Y von Zeichen, ob $X < Y$, $X = Y$ oder $X > Y$ und
  - ◇ ob blank-padded-Semantik (`PAD SPACE`) oder non-padded-Semantik (`NO PAD`) verwendet wird.

Zeichenkettenvergleich (2)

- `'a' < 'b'` usw. und `'A' < 'B'` usw. sollten in jedem System gelten.
- Die Systeme unterscheiden sich schon im Vergleich von Klein- und Großbuchstaben. Die Defaults sind:
  - ◇ In Oracle kommen alle Großbuchstaben vor den Kleinbuchstaben (ASCII), z.B. `'Z' < 'a'`.
  - ◇ In DB2 liegen die Großbuchstaben zwischen den Kleinbuchstaben, z.B. `'a' < 'A'`, `'A' < 'b'`.
  - ◇ SQL Server, MS Access und MySQL sind case-insensitive, z.B. `'a' = 'A'`.

Zeichenkettenvergleich (3)

- Manchmal kann man dies ändern, aber z.B. nur während der Installation (SQL Server) oder während der DB-Erstellung (Oracle, DB2).
- Ist die Reihenfolge (<, =, >) zweier Zeichen bekannt, so ist der Vergleich von Zeichenketten der gleichen Länge klar:
  - ◇ Das System vergleicht Zeichen für Zeichen und der erste Vergleich, der nicht "=" ergibt, bestimmt das Ergebnis.

    DB2 macht zwei Schritte: Es vergleicht erst character "weights" und wenn es keinen Unterschied gibt, auch die character codes.

Zeichenkettenvergleich (4)

- Für Zeichenketten veschiedener Länge gibt es
  - ◇ Non-Padded Vergleichs-Semantik:

    Z.B. `'a' < 'a '`.

    Strings werden Zeichen für Zeichen verglichen. Endet ein String und es wurde kein Unterschied gefunden, gilt der kürzere String als kleiner.
  - ◇ Blank-Padded Vergleichs-Semantik:

    Z.B. `'a' = 'a '`.

    Der kürzere String wird vor dem Vergleich mit `' '` aufgefüllt.

Zeichenkettenvergleich (5)

- DB2, SQL Server, Access und MySQL verwenden blank-padded Semantik (zumindest als Default).
- Oracle hat non-padded Semantik, wenn mindestens ein Operand des Vergleichs den Typ `VARCHAR2` hat.

  Oracle hat einen Typ `VARCHAR2`(n) eingeführt. Er ist derzeit äquivalent zu `VARCHAR`(n), aber Oracle beabsichtigt, die Vergleichs-Semantik für `VARCHAR` zu ändern, wobei die Semantik für `VARCHAR2` bleibt wie bisher. String-Konstanten in der Anfrage haben den Typ `CHAR`(n). Z.B. kann ein Vergleich von `CHAR(10)`- und `CHAR(20)`-Spalten möglicherweise wahr sein, sowie ein Vergleich dieser Spalten mit z.B. `'abc'`. Aber `CHAR(10)` und `VARCHAR(20)` können nur gleich sein, wenn der `VARCHAR` zufällig 10 Zeichen hat. Leerzeichen am Stringende in `VARCHAR2`-Spalten sind ein Problem: unsichtbar in der Ausgabe, aber können "`=`" verhindern.

Zeichenkettenvergleich (6)

- Verwendet das DBMS eine case-sensitive Semantik, bekommt man einen case-insensitiven Vergleich, indem man alles in z.B. Großbuchstaben konvertiert:

```
SELECT VORNAME, NACHNAME
FROM   STUDENTEN
WHERE  UPPER(EMAIL) = UPPER('xyz@hotmail.com')
```

- `UPPER` funktioniert in SQL-92, Oracle, SQL Server, DB2, MySQL. In Access nimmt man `UCASE`.

  `UCASE` funktioniert auch in DB2 und MySQL. Das Buch von Chamberlin über DB2 beschreibt nur `UCASE`.

Zeichenkettenvergleich (7)

- Der umgekehrt Fall (case-sensitiver Vergleich mit case-insensitivem DBMS) ist schwieriger.

  Aber auch viel seltener erforderlich.
- Z.B. kann man in MySQL einen String in einen binären String konvertieren, um einen case-sensitiven Vergleich zu machen:

```
BINARY EMAIL = 'xyz@hotmail.com'
```

- Das gleiche funktioniert auch in SQL Server:

```
  CAST(EMAIL AS VARBINARY(255))
= CAST('...' AS VARBINARY(255))
```

Zeichenkettenvergleich (8)

- Vermutet man angehängte Leerzeichen, kann man sie so sichtbar machen:

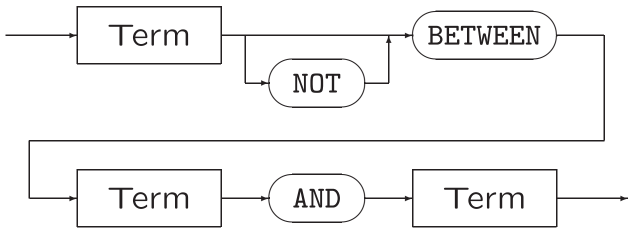
```
SELECT '"' || NACHNAME || '"' AS NACHNAME
FROM   STUDENTEN
```

- Man kann angehängte Leerzeichen auch löschen:
  - ◇ `TRIM(TRAILING ' ' FROM NACHNAME)`
    in SQL-92 (funktioniert in MySQL)

    Wird in Oracle, DB2, SQL Server, Access nicht unterstützt.
  - ◇ `RTRIM(NACHNAME)`
    in Oracle, DB2, SQL Server, MySQL, Access.

BETWEEN-Bedingungen

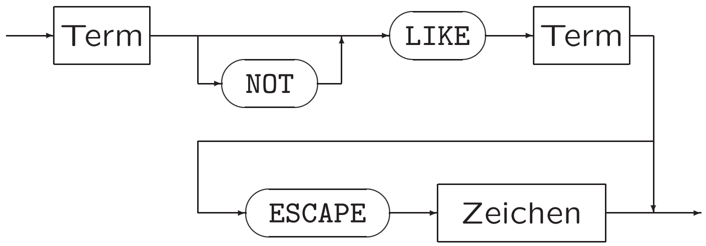
Atomare Formel (Form 2):



- `x BETWEEN y AND z` ist äquivalent zu `x >= y AND x <= z`.
- Z.B.: `PUNKTE BETWEEN 5 AND 8`

LIKE-Bedingungen (1)

Atomare Formel (Form 3):



- Z.B.: `EMAIL LIKE '%.pitt.edu'`

  Das ist für alle Email-Adressen wahr, die mit "`.pitt.edu`" enden.

LIKE-Bedingungen (2)

- Das rechte Argument wird als Muster interpretiert. In SQL-86 und in DB2 muß dies eine String-Konstante sein.

  In Oracle, SQL Server, Access und MySQL kann man jeden stringwertigen Term als Muster verwenden (z.B. auch eine andere Spalte).
- "%" im Muster ersetzt eine Folge beliebiger Zeichen (den leeren String eingeschlossen).

  Im UNIX-Shell (Kommando-Interpreter) wird "*" statt "%" verwendet.
- "_" passt auf ein beliebiges einzelnes Zeichen.

  Dies entspricht "?" im Shell.

LIKE-Bedingungen (3)

- `LIKE` muß zur Mustersuche verwendet werden. Das Gleichheitszeichen überprüft nur Zeichengleichheit.

  Auch wenn der Vergleichs-String "%" oder "_" enthält.

- Z.B. ist Folgendes in SQL legal, wird aber das falsche Ergebnis liefern (im Beispiel ∅):

```
SELECT VORNAME, NACHNAME
FROM   STUDENTEN
WHERE  NACHNAME = 'S%'
```
Falsch!

LIKE-Bedingungen (4)

- Um die Zeichen "%" und "_" ohne ihre spezielle Bedeutung im Muster zu verwenden, wird ein "Escape"-Zeichen verwendet.

  Ein Escape-Zeichen löscht die spezielle Bedeutung des darauffolgenden Zeichens. Ist z.B. "\" das Escape-Zeichen, ist "\%" nur ein Prozentzeichen, kein beliebiger String.

- Das Escape-Zeichen muß deklariert werden, z.B.:

```
PROZNAME LIKE '\_%' ESCAPE '\'
```

  Dies gibt alle Prozeduren aus, die mit "_" beginnen.

  In MySQL ist "\" der Default, wenn kein anderes Escape-Zeichen deklariert wurde. Dies verletzt jedoch den SQL-92-Standard.

LIKE-Bedingungen(5)

- `LIKE` verwendet die non-padded-Semantik.

  Oracle, DB2, MySQL, Access verwenden die non-padded-Semantik, wie im SQL-92-Standard verlangt. Man beachte, daß MySQL angehängte Leerzeichen entfernt, wenn Strings gespeichert werden. Alle Systeme füllen Werte mit Leerzeichen auf, wenn die Spalte den Typ Zeichenkette mit fester Länge hat.
  In SQL Server stimmt es evtl. auch dann überein, wenn der gespeicherte String mehr Leerzeichen als das Muster hat. Enthält das Muster mehr Leerzeichen, schlägt der Vergleich fehl.

- Z.B. ist `'a' = 'a '` in manchen DBMS wahr, aber `'a' LIKE 'a '` ist mit Sicherheit falsch.
- Die Case-Sensitivität ist die gleiche wie für gewöhnliche Vergleiche.

Reguläre Ausdrücke

- SQL Server und Access unterstützen auch Zeichenbereiche, z.B. `[a-zA-Z]` in `LIKE`-Bedingungen.

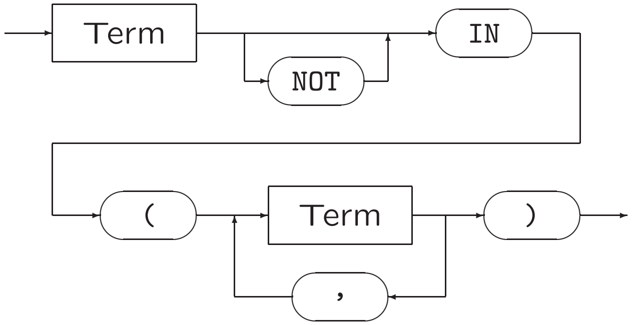
  Dies verletzt den Standard.

- MySQL hat einen zusätzlichen Operator "`RLIKE/REGEXP`", der beliebige reguläre Ausdrücke als Muster akzeptiert.
- Der SQL:1999-Standard führte ein Prädikat "`SIMILAR TO`" ein, das Vergleiche mit regulären Ausdrücken durchführt.

  In den meisten Systemen, z.B. Oracle 9i, noch nicht implementiert.

IN-Bedingungen (1)

Atomare Formel (Form 4):



IN-Bedingungen (2)

- Z.B. `ATYP IN ('Z', 'E')`
- Dies ist äquivalent zu

  `ATYP = 'Z' OR ATYP = 'E'`

- Der SQL-86-Standard erlaubt nur Konstanten in der Aufzählung der Werte.

  SQL-92, Oracle, SQL Server und DB2 erlauben beliebige Terme, aber es ist normalerweise besserer Stil, wenn man `OR` verwendet, falls die Menge keine Aufzählung von Konstanten ist.

- Man beachte, daß "(...)" hier eine "Menge" ist (obwohl in Mathematik für Intervalle verwendet).

Inhalt

SELECT-Klausel, *

- `SELECT` legt die Terme fest, die ausgegeben werden, falls die `WHERE`-Bed. wahr ist (Ergebnis-Spalten).
- `SELECT *` kann verwendet werden, um alle Spalten der Tabelle(n) unter `FROM` auszugeben, z.B. ist

```
SELECT *
FROM   STUDENTEN
```

  äquivalent zu

```
SELECT SID, VORNAME, NACHNAME, EMAIL
FROM   STUDENTEN
```

- In Programmen sollte man * vermeiden, da später manchmal Spalten zu Tabellen hinzugefügt werden.

Duplikat-Eliminierung (1)

- Ein Unterschied zwischen SQL und RA ist, daß Duplikte in SQL explizit eliminiert werden müssen.
- Z.B.: Welche Aufgaben wurden von mindestens einem Studenten gelöst?

```
SELECT ATYP, ANR
FROM   BEWERTUNGEN
```

ATYP	ANR
H	1
H	2
Z	1
H	1
H	2
Z	1
H	1
Z	1

Duplikat-Eliminierung (2)

- Die Duplikate treten auf, weil die Anfrage mit einer Schleife über die Zeilen in BEWERTUNGEN ausgeführt wird.
- Könnte eine Anfrage Duplikate enthalten und gibt es keinen Grund, diese mit auszugeben, verwendet man “SELECT DISTINCT”:

```
SELECT DISTINCT ATYP, ANR
FROM   BEWERTUNGEN
```

ATYP	ANR
H	1
H	2
Z	1

Duplikat-Eliminierung (3)

- Um zu betonen, daß es Duplikate gibt, die auch gewünscht sind, kann man “SELECT ALL” schreiben.

  “ALL” ist jedoch der Default.

- Man beachte, daß DISTINCT immer zu ganzen Zeilen gehört, nicht zu einzelnen Spalten.

  Sonst NF^2-Tabellen nötig. Mit klassischen Relationen z.B. unmöglich: eine Zeile je Student mit all seinen Resultaten. Mit Ausgabe-Formatierung aber ähnliche Ergebnisse: In SQL*Plus kann man Spaltenwerte nur ausgeben lassen, wenn sich der Wert vom vorigen unterscheidet.

- Z.B. ist Folgendes ein Syntax-Fehler:

```
SELECT ATYP, ANR, DISTINCT THEMA
FROM   AUFGABEN
```

Falsch!

Umbenennung von Spalten

- Um Ausgabe-Spalten umzubenennen:

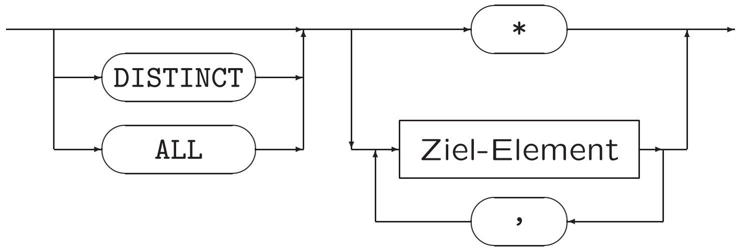
```
SELECT VORNAME AS V_Name, NACHNAME AS "Name"
FROM   STUDENTEN
```

V_NAME	Name
Lisa	Weiss
Michael	Grau
Daniel	Sommer
Iris	Winter

- Dies funktioniert in SQL-92, Oracle, SQL Server, DB2, MySQL, Access, aber nicht in SQL-86.
- “AS” kann in SQL-92 und allen obigen Systemen außer Access weggelassen werden.

SELECT-Syntax (1)

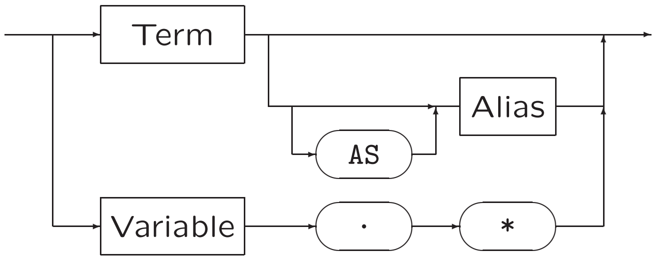
Ziel-Liste (nach SELECT):



- ALL (keine Duplikat-Elimination) ist der Default.

SELECT-Syntax (2)

Ziel-Element:



- "Variable.*" und "[AS] Alias" funktionieren in SQL-92, Oracle, SQL Server, DB2, MySQL und Access (in Access wird "AS" benötigt). Diese Konstruktionen sind im alten SQL-86-Standard nicht enthalten.