Teil 5: 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 6 behandelt.

- Fehler und unnötige Komplikationen vermeiden.

  Z.B sollten Sie das Konzept einer inkonsistenten Bedingung erklären können. Sie sollten auch überprüfen können, ob eine Anfrage Duplikate produzieren könnte (zumindest in einfachen Fällen).

- Anfragen auf Fehler oder Äquivalenz überprüfen.
- Die Portabilität von Sprachkonstrukten erläutern.

Inhalt

SQL

- Heute ist SQL die einzige DB-Sprache für relationale DBMS (Industriestandard).

  Andere Sprachen wie QUEL sind ausgestorben (die Sprache QBE hat zumindest einige graphische Schnittstellen zu Datenbanken inspiriert (z.B. in Access) und die Sprache Datalog hat SQL:1999 beeinflusst und wird noch in der Forschung studiert und weiterentwickelt). Jedes kommerzielle RDBMS muß heute eine SQL-Schnittstelle haben. Es kann zusätzlich noch andere (z.B. graphische) Schnittstellen geben.

- SQL wird verwendet für:
  - ◇ Interaktive "Ad-hoc"-Befehle und
  - ◇ Anwendungsprogrammentwicklung (in andere Sprachen wie C, Java, HTML eingebettet).

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

Basis-SQL-Anfragen (1)

Einfache Ein-Tabellen-Anfragen (SQL für Anfänger):

- Eine einfache SQL-Anfrage hat folgende Form

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

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

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

ANR	PUNKTE
1	10
2	8

Basis-SQL-Anfragen (2)

Von relationaler Algebra zu SQL:

- Ein Ausdruck in relationaler Algebra

$$\pi_{A_1,\ldots,A_n}(\sigma_F(R_1 \times \cdots \times R_m))$$

wird in SQL geschrieben als

$$\begin{array}{ll} \texttt{SELECT} & A_1\texttt{,}\ \ldots\texttt{,}\ A_n \\ \texttt{FROM} & R_1\texttt{,}\ \ldots\texttt{,}\ R_m \\ \texttt{WHERE} & F \end{array}$$

- Haben verschiedene R_i Attribute mit gleichem Namen A, schreibt man $R_i.A$, um den Bezug eindeutig zu machen.
- Duplikate können auftreten (→ `SELECT DISTINCT`).

Basis-SQL-Anfragen (3)

Von relationaler Algebra zu SQL, fortgesetzt:

- Man kann aber auch Umbenennungen verwenden:

$$\pi_{A_1,\ldots,A_n}(\sigma_F(\rho_{X_1}(R_1) \times \cdots \times \rho_{X_m}(R_m)))$$

wird in SQL geschrieben als

$$\begin{array}{ll} \texttt{SELECT} & A_1\texttt{,}\ \ldots\texttt{,}\ A_n \\ \texttt{FROM} & R_1\ X_1\texttt{,}\ \ldots\texttt{,}\ R_m\ X_m \\ \texttt{WHERE} & F \end{array}$$

- SQL hat einen `UNION`-Operator, um Ergebnisse verschiedener `SELECT`-Ausdrücke zu kombinieren.
- Obwohl es `EXCEPT` für Mengendifferenzen gibt, verwendet man dafür meist Unteranfragen (s. Teil 6).

Basis-SQL-Anfragen (4)

Von Logik zu SQL:

- SQL ist sehr ähnlich zum relationalen Tupelkalkül.

  Siehe Kapitel 3. (Logik 1. Stufe, Relationen werden als Sorten behandelt, und Attribute als Funktionen, in Punktnotation geschrieben). SQL erlaubt keine neuen Recordtypen (und braucht dafür `UNION`).

- Z.B. alle Hausaufgabenpunkte von Studentin 101:

  $$\{\texttt{X.anr},\ \texttt{X.punkte}\ [\texttt{bewertungen X}] \mid \texttt{X.atyp} = \texttt{'H'} \land \texttt{X.sid} = 101\}$$

- Nur eine syntaktische Variante der SQL-Anfrage:

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


Basis-SQL-Anfragen (5)

Von Logik zu SQL, fortgesetzt:

- Enthält die Bedingung F keine Quantoren, so wird

  $$\{t_1, \ldots, t_n\ [R_1\ X_1, \ldots, R_m\ X_m] \mid \exists S_1\ Y_1 \ldots \exists S_k\ Y_k\colon F\}$$

  in SQL geschrieben als

  SELECT t_1, ..., t_n
  FROM $R_1\ X_1$, ..., $R_m\ X_m$, $S_1\ Y_1$, ..., $S_k\ Y_k$
  WHERE F

- Es kann kleine Unterschiede mit Duplikaten geben.

  Wenn nötig, verwendet man `SELECT DISTINCT`, um Duplikate zu eliminieren.

- Für Quantoren in F braucht man Unteranfragen.

Bezug zur Theorie

- SQL basiert auf Logik (Tupelkalkül).

  Die Übersetzung von Quantoren in Unteranfragen ist immer noch sehr direkt (wenn die Variable an eine Relation gebunden ist). SQL hat viele Erweiterungen, z.B. Aggregationen und vieles mehr.

- SQL enthält auch eine wachsende Anzahl von Konstrukten der relationalen Algebra.
- Bei der Entwicklung von SQL wurde versucht, nah an der natürlichen Sprache (Englisch) zu bleiben.

  In SQL kann man eine Bedingung oft auf mehrere Arten ausdrücken. Viele Varianten wurden nur eingeführt, weil sie sich besser lesen lassen.

- In SQL sind Duplikate signifikant.

Geschichte

- SEQUEL, eine frühere Version von SQL, wurde von Chamberlin, Boyce et al. bei IBM Research, San Jose (1974) entwickelt.

  SEQUEL steht für "Structured English Query Language" ("Strukturierte englische Anfragesprache"). Manche Leute Sprechen SQL noch heute auf diese Art aus. Der Name wurde aus rechtlichen Gründen geändert (SEQUEL war bereits eine eingetragene Marke). Codd war auch in San Jose, als er das relationale Modell erfand.

- SQL war die Sprache des System/R (1976/77).

  System/R war ein sehr einflussreicher Forschungs-Prototyp.

- Ersten SQL-unterstützenden kommerziellen Systeme waren Oracle (1979) und IBM SQL/DS (1981).

Standards (1)

- Erster Standard 1986/87 (ANSI/ISO).

  Das war sehr spät, weil es schon viele SQL-Systeme auf dem Markt gab. Der Standard war der "kleinste gemeinsame Nenner". Er enthielt im wesentlichen nur den Schnitt der SQL-Implementierungen.

- Erweiterung für Fremdschüssel usw. '89 (SQL-89).

  Diese Version wird auch SQL-1 genannt. Alle kommerziellen Implementierungen unterstützen heute diesen Standard, aber jede hat viele Erweiterungen. Der Standard hatte 120 Seiten.

- Größere Erweiterung: SQL2 oder SQL-92 (1992).

  626 Seiten, aufwärts kompatibel zu SQL-1. Der Standard definiert drei Kompatibilitätsgrade: "Entry", "Intermediate", "Full" (+ später "Transitional" zwischen ersten beiden). Oracle 8.0 und SQL Server 7.0 sind "Entry Level"-kompatibel, höhere Konstrukte zum Teil.

Standards (2)

- Weitere große Erweiterung: SQL:1999.

  Zuerst wurde SQL:1999 als eine Vorgänger-Version des SQL3-Standards angekündigt, aber nun scheint es SQL3 zu sein. Bis 12/2000 erschienen die Bände 1–5 und 10 des SQL:1999-Standards. Sie haben zusammen 2355 Seiten. Anstelle von Levels definiert der Standard nun "Core SQL" und eine Vielzahl von Paketen von Sprach-Features.

- Wichtige neue Möglichkeiten in SQL:1999:
  - ◇ Nutzer-definierte Datentypen, strukturierte Typen.

    Man kann nun "distinct types" definieren, die Domains sehr ähnlich sind, aber ein Vergleich zwischen verschiedenen "distinct types" ist nun ein Fehler. Es gibt auch Typ-Konstruktoren "`ARRAY`" und "`ROW`" für strukturierte Attributwerte und "`REF`" für Zeiger auf Zeilen.

Standards (3)

- Wichtige neue Möglichkeiten in SQL:1999, Forts.:
  - ◇ OO-Fähigkeiten, z.B. Vererbung/Untertabellen.
  - ◇ Rekursive Anfragen.
  - ◇ Trigger, persistent gespeicherte Module.
- Kleinere Veränderung: SQL:2003.

  Größte Neuerungen: Unterstützung von XML, Verwaltung von externen Daten. Außerdem gibt es nun Generatoren für eindeutige Nummern und einen "`MULTISET`" type constructor. Ansonsten ist es nur die 2. Auflage des SQL:1999-Standards. Auch die OLAP-Elemente (On-Line Analytical Processing), die als Zusatz zum SQL:1999-Standard veröffentlicht wurden, sind nun im SQL:2003-Standard integriert.

Standards (4)

- Bände des SQL:2003-Standards:
  - ◇ Teil 1: Framework (SQL/Framework) [84 S.]
  - ◇ Teil 2: Foundation (SQL/Foundation) [1268 S.]
  - ◇ Teil 3: Call-Level Interface (SQL/CLI) [406 S.]

    Dies spezifiziert eine Menge von Prozeduren, die verwendet werden können, um SQL-Anweisungen auszuführen und die Ergebnisse einzulesen (ähnlich zu ODBC).
  - ◇ Teil 4: Persistent Stored Modules (SQL/PSM) [186 S.]

    Programmiersprache für gespeicherte Prozeduren. Es macht SQL berechnungsuniversell und ist ähnlich zu PL/SQL in Oracle oder Transact SQL in Microsoft SQL Server und Sybase.

Standards (5)

- Bände des SQL:2003-Standards, fortgesetzt:
  - ◇ Teil 9: Management of External Data (SQL/MED)
  - ◇ Teil 10: Object Language Bindings (SQL/OLB) [404 Seiten]

    Dies definiert, wie SQL-Befehle in Java-Programme eingebettet werden können.
  - ◇ Teil 11: Information and Definition Schemas (SQL/Schemas) [298 Seiten]

    Dies definiert ein Standard-Data-Dictionary (System-Katalog).

Standards (6)

- Bände des SQL:2003-Standards, fortgesetzt:
  - ◇ Teil 13: SQL Routines and Types using the Java Programming Language (SQL/JRT) [206 S.]

    Dies legt fest, wie Javas statistische Methoden in SQL-Statements verwendet werden können und wie Java-Klassen als strukturierte Typen in SQL verwendet werden können.
  - ◇ Teil 14: XML-Related Specifications (SQL/XML) [268 S.]
- Der offizielle Name von Teil n des Standards ist [INCITS/] ISO/IEC 9075-n-2003: Information technology — Database Languages — SQL.

Standards (7)

- Die fehlenden Teile des SQL:2003-Standards werden wahrscheinlich nie erscheinen.

  Teil 5 ist in SQL:1999 "object language bindings" beschrieben. Dies wurde in Teil 2 integriert. Teil 6 sollte die X/OPEN XA-Spezifikation (Transaktionen) behandeln. Dieser Teil wurde gelöscht. Teil 7 sollte SQL/Temporal spezifizieren. Es wurde gestoppt, weil das Komitee große Meinungsverschiedenheiten hatte. Teil 8 sollte die erweiterten Objekte/abstrakten Datentypen behandeln und wurde in Teil 2 integriert. Teil 12 soll Replication behandeln.
- 2006 erschien eine neue Version des SQL-Standards (SQL:2006), in dem aber nur der XML-Teil erweitert wurde.

  Insbesondere Kombination von XQuery und SQL.

Standards (8)

- Aktuelle Version des Standards ist SQL:2008.

ISO/IEC 9075-1:2008	Framework (SQL/Framework) Korrektur ISO/IEC 9075-1:Cor 1:2010
ISO/IEC 9075-2:2008	Foundation (SQL/Foundation)
ISO/IEC 9075-3:2008	Call-Level Interface (SQL/CLI)
ISO/IEC 9075-4:2008	Persistent Stored Modules (SQL/PSM)
ISO/IEC 9075-9:2008	Management of External Data (SQL/MED)
ISO/IEC 9075-10:2008	Object Language Bindings (SQL/OLB)
ISO/IEC 9075-11:2008	Information and Definition Schemas (SQL/Schemata)
ISO/IEC 9075-13:2008	SQL Routines and Types Using the Java TM Programming Language (SQL/JRT)
ISO/IEC 9075-14:2008	XML-Related Specifications (SQL/XML)

  Teil 1, 2, 11: minimale Anforderungen, andere Teile: Erweiterungen.
- Die Unterschiede zum früheren Stand sind klein.

  Z.B. `TRUNCATE` jetzt im Standard, sowie `ORDER BY` auch in Unteranfragen, zur Unterstützung von `FETCH FIRST`. Auch `INSTEAD OF` Trigger.

Standards (9)

- Die Standards sind bei `webstore.ansi.org` billiger als bei `www.iso.org`.

  Bei der ANSI gibt es die Variante INCITS/ISO/IEC für 30$ pro Band. `http://www.wiscorp.com/sql200n.zip`: Kostenlose Vorversion.

- Es gibt einen verwandten Standard ISO/IEC 13249: "SQL multimedia and application packages".

  Teil 1: Framework, Teil 2: Full-Text, Teil 3: Spatial, Teil 5: Still Image, Teil 6: Data Mining.

- Es gibt außerdem noch den technischen Bericht ISO/IEC TR 19075-1:2011, der reguläre Ausdrücke in SQL beschreibt.

Syntax-Formalismus

- In dieser Vorlesung wird die Syntax von SQL-Anfragen mit "Syntax-Graphen" definiert.

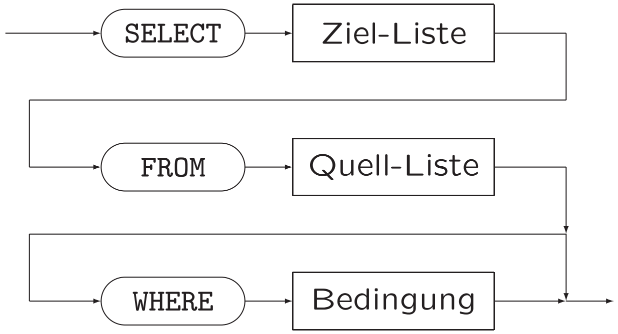
  Beispiel auf nächster Folie. Alternative zu kontextfreien Grammatiken (definiert dieselbe Klasse von Sprachen). In Anhang A genauer.

- Um eine Zeichenfolge syntaktisch richtig zu erstellen, muß man einen Pfad vom Start bis zum Ziel durch den Graphen verfolgen.

  Wörter in Ovalen werden direkt in die Ausgabe geschrieben, Kästen sind "Aufrufe" anderer Graphen: An diesem Punkt muß man einen Pfad durch den Graphen finden, dessen Name in dem Kasten steht. Anschließend kehrt man zu der Kante zurück, die den Kasten verlässt. Das Oracle-SQL-Referenz-Handbuch enthält auch Syntax-Graphen, aber dort ist die Bedeutung von Ovalen und Kästen umgekehrt.

Basis-Anfrage-Syntax (1)

SELECT-Ausdruck (vereinfacht):



Basis-Anfrage-Syntax (2)

- 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 ($\sim$ 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:
  - ◇ 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 "*/"

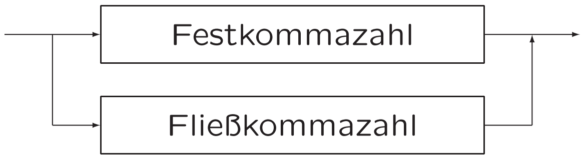
    Nur in Oracle, SQL Server und MySQL unterstützt: weniger portabel.

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.

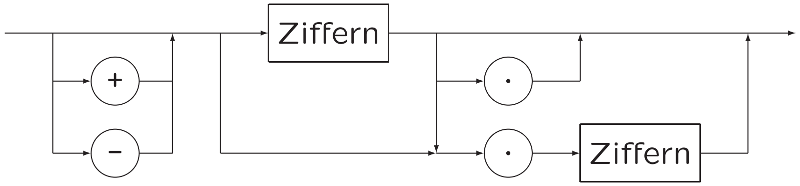
Zahlen (1)

- 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)

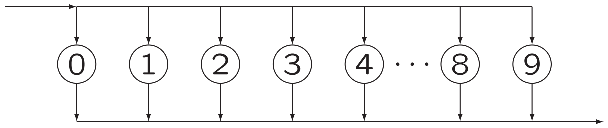
- Festkommazahl ("Exact Numeric Literal")



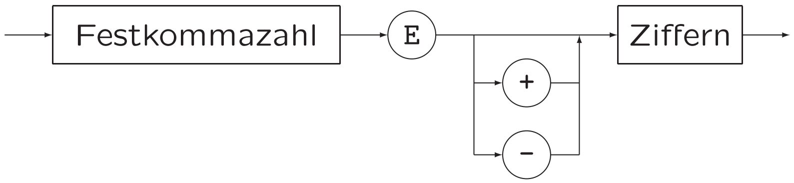
- Ziffern (vorzeichenlose ganze Zahl):

Zahlen (3)

- Ziffer:



- Fließkommazahl ("Approximate Numeric Literal"):



Zeichenketten (1)

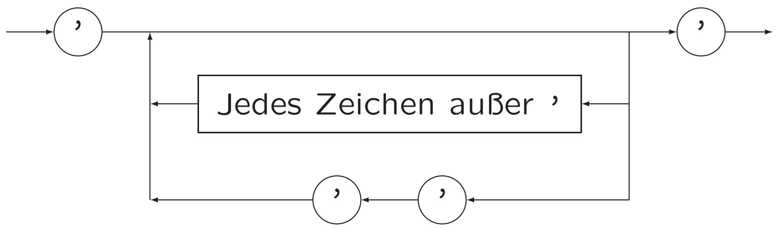
- Ein Zeichenketten-Literal ist eine Folge von Zeichen, eingeschlossen in Hochkommas, z.B.
  - ◇ `'abc'`
  - ◇ `'Dies ist ein String.'`
- Hochkommas in Zeichenketten müssen verdoppelt werden, z.B. `'John''s book'`.

  Der tatsächliche Wert der Zeichenkette ist `John's book` (mit einfachem Hochkomma). Das Verdoppeln ist nur eine Art, es einzugeben.
- Natürlich ist auch die leere Zeichenkette erlaubt: `''`.

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

Zeichenketten (2)

- String Literal:



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 10):
  - ◇ 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.

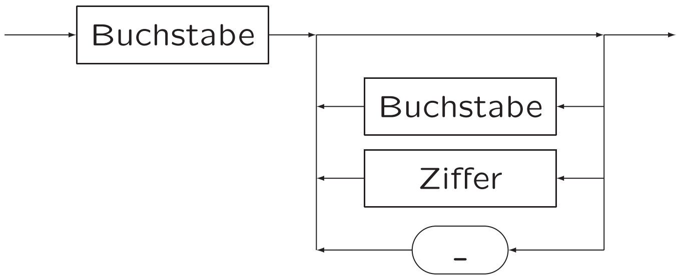
Andere Konstanten (2)

- 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)

- Bezeichner:



- Z.B. `Dozenten_Name`, `X27`, aber nicht `_XYZ`, `12`, `2BE`.
- U.a. als Tabellen- und Spaltennamen verwendet.

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 möglich, nationale Zeichen 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.

- Bezeichner/Schlüsselwörter 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.

Bezeichner (4)

- Betrachten Sie z.B. die Anfrage

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

- Folgende Anfrage ist vollkommen äquivalent:

```
select X . Vorname ,
           x.NachNaMe
       From Studenten X
```

(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 - 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]

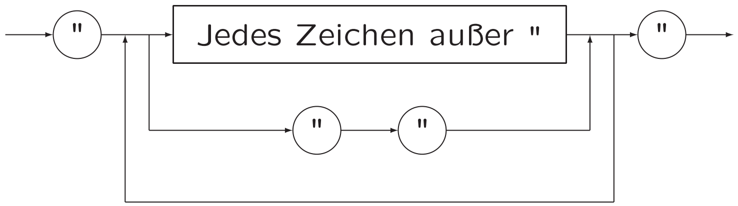
Reservierte Wörter - SQL (7)

— 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 jedes SQL-Statement 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 zum SQL-Statement.

  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	···

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

Tupelvariablen (1)

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

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

- Diese Anfrage kann wie folgt ausgewertet werden:

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

- A steht hier für eine Zeile in AUFGABEN (die Schleife weist der Variablen nacheinander jedes Tupel zu).

Tupelvariablen (2)

- 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`".)

Tupelvariablen (3)

- 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`".

- Bezieht man sich auf ein Attribut A einer Tupelvariablen X, ist es möglich, einfach A statt $X.A$ zu schreiben, falls X die einzige Tupelvariable ist, die das Attribut A hat.

Das wird später näher erklärt. Im Beispiel könnte man einfache "`ANR`" für das Attribut schreiben, egal, ob eine Tupelvariable explizit deklariert wurde oder nicht.

Verbunde/Joins (1)

- Gegeben sei eine Anfrage mit zwei Tupelvariablen:

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

- Dann wird `S` über die 4 Tupel in `STUDENTEN` laufen und `B` über die 8 Tupel in `BEWERTUNGEN`. Im Prinzip werden alle $4 * 8 = 32$ Kombinationen betrachtet:

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

Verbunde/Joins (2)

- Gute DBMS wenden evtl. einen besseren Auswertungsalgorithmus an (hängt von Bedingung C ab).

Wenn z.B. C die Verbundbedingung `S.SID = B.SID` enthält, könnte das DBMS alle Tupel in `BEWERTUNGEN` durchgehen und jeweils das zugehörige Tupel in `STUDENTEN` mit Hilfe eines Indexes über `STUDENTEN.SID` finden (die meisten Systeme erstellen einen Index über die Schlüsselattribute automatisch).

- Aber um die Bedeutung der Anfrage zu verstehen, reicht der einfache Algorithmus.

Der Anfrageoptimierer kann jeden Algorithmus verwenden, der das gleiche Ergebnis hat, eventuell in einer anderen Reihenfolge (SQL legt die Reihenfolge der Ergebnistupel nicht fest).

Verbunde/Joins (3)

- Der Verbund muß explizit in der WHERE-Bedingung angegeben werden:

```
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'
```

- 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!

Verbunde/Joins (4)

- 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'
```


Verbunde/Joins (5)

- 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:

- 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.

Unnötige Joins (1)

- Verbinden Sie nicht mehr Tabellen als nötig.

  Anfragen laufen langsamer: Viele Optimierer entfernen keine Joins.

- Z.B 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
```

- Kann diese Anfrage je ein anderes Ergebnis liefern?

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


Unnötige Joins (2)

- 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
```

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:

Falsch!

```
SELECT S.VORNAME, S.NACHNAME
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.

Übung

- Erkennen Sie das Problem in dieser Anfrage? Ziel ist es, alle Studenten auszugeben, die eine Aufgabe über SQL und eine über ER-Entwurf gelöst haben.

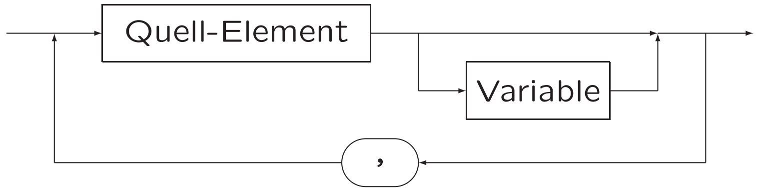
```
SELECT S.VORNAME, S.NACHNAME
FROM   STUDENTEN S, BEWERTUNGEN B,
       AUFGABEN A1, AUFGABEN A2
WHERE  S.SID = B.SID
AND    B.ATYP = A1.ATYP AND B.ANR = A1.ANR
AND    B.ATYP = A2.ATYP AND B.ANR = A2.ANR
AND    A1.THEMA = 'SQL'
AND    A2.THEMA = 'ER'
```


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).

FROM-Syntax (1)

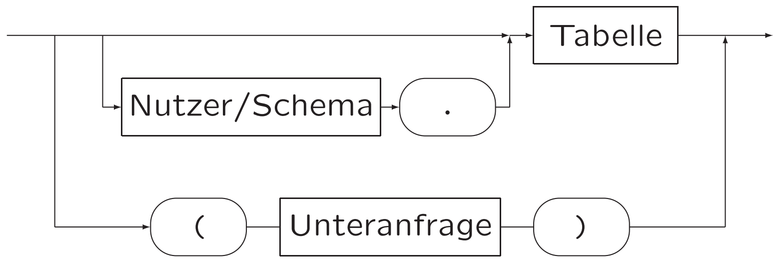
Quell-Liste (nach FROM):



- 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 Unteranfragen erst ab Version 4.1.
- 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.

Inhalt

Terme (1)

- Ein Term bezeichnet ein Datenelement.

  Der Begriff "Term" wird in der Logik verwendet. In Programmiersprachen sagt man "Ausdruck". Der SQL-Standard verwendet "skalarer Ausdruck", weil es dort auch "Tabellenausdrücke" gibt.

- Terme sind:
  - ◇ Attribut-Referenzen, z.B. `STUDENTEN.SID`.
  - ◇ Konstanten ("Literale"), z.B. `'Lisa'`, `1`.
  - ◇ Zus.gesetzte Terme (mit Datentyp-Operatoren wie +, -, *, / (für Zahlen), || (String-Konkatenation) und Datentyp-Funktionen wie `0.9 * MAXPT`).
  - ◇ Aggregations-Terme, z.B. `MAX(PUNKTE)`: s. Teil 6.

Terme (2)

- 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

Attribut-Referenzen (1)

- Auf Attribute kann man in dieser Form zugreifen:

  `Variable.Attribut`

- Hat nur eine Variable das Attribut, kann der Variablenname fehlen. 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.

Attribut-Referenzen (2)

- Gegeben sei diese Anfrage:

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

Falsch!

- 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 genau.

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 10 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 Standard-Vorrangregeln, z.B. bedeutet `A+B*C`:

  `A+(B*C)`,

  und nicht

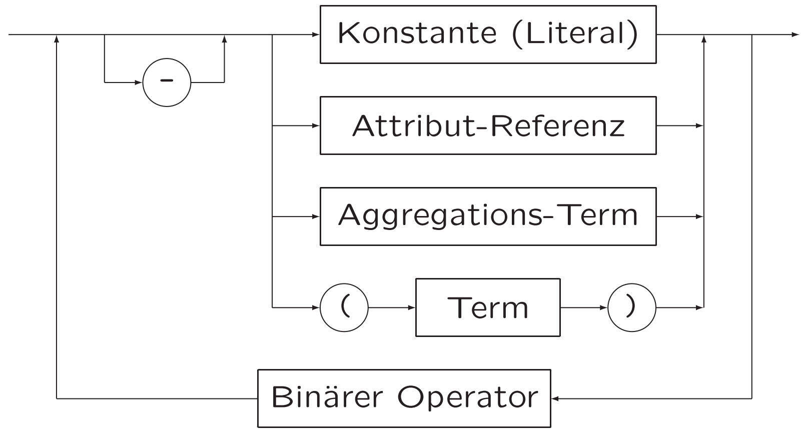
  `(A+B)*C`.

- 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).

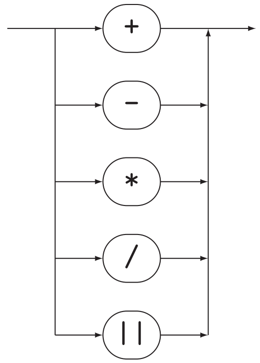
Terme: Syntax (1)

Term (Skalarer Ausdruck, Wert-Ausdruck):



Terme: Syntax (2)

Binärer Operator:



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

Terme mit Nullwerten (1)

- Wie in Kapitel 3 erwähnt, können Attribute einen Nullwert enthalten (falls nicht mit `NOT NULL` in der Tabellendeklaration ausgeschlossen).
- 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 ernste Verletzung des SQL-Standards (Oracle listet dies als "nicht-übereinstimmend" in einem Anhang seines SQL-Referenz-Handbuches auf). Da es jedoch in Anwendungsprogrammen verwendet werden könnte und es existierende DB-Dateien gibt, die nicht zwischen beiden unterscheiden, ist es schwierig zu ändern.

Terme mit Nullwerten (2)

- Ist eines der Argumente Null, so liefern Datentyp-Funktionen auch Null. Ist z.B. `A` Null, so ist `A+B` ebenfalls Null.
- Das Schlüsselwort `NULL` ist selbst kein Term (Ausdruck), obwohl es an vielen Stellen verwendet werden kann, an denen ein Term verlangt wird.

Terme mit Nullwerten (3)

- `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).

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

Bedingungen (1)

- Bedingungen bestehen aus atomaren Formeln, z.B.

  `PUNKTE >= 8`,

  verbunden mit "`AND`", "`OR`", "`NOT`".

- `AND` bindet stärker als `OR`, somit wird

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

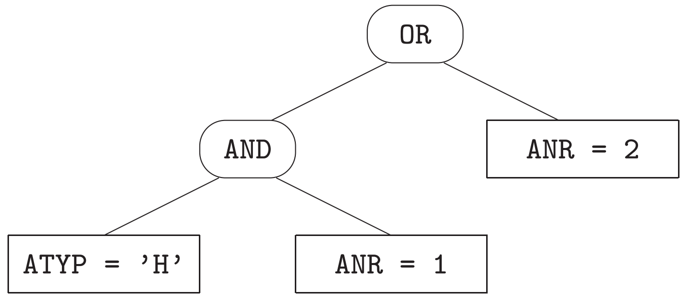
  implizit so geklammert:

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

- In diesem Beispiel ist dies jedoch nicht gewünscht.

Bedingungen (2)

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



Bedingungen (3)

- `NOT` bindet am stärksten, d.h. es gilt nur für die direkt folgende Bedingung (atomare Formel).
- Klammern (...) können verwendet werden, um die Bindungsstärken / Prioritäten der Operatoren aufzuheben.
- 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 (4)

- 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 (5)

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

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

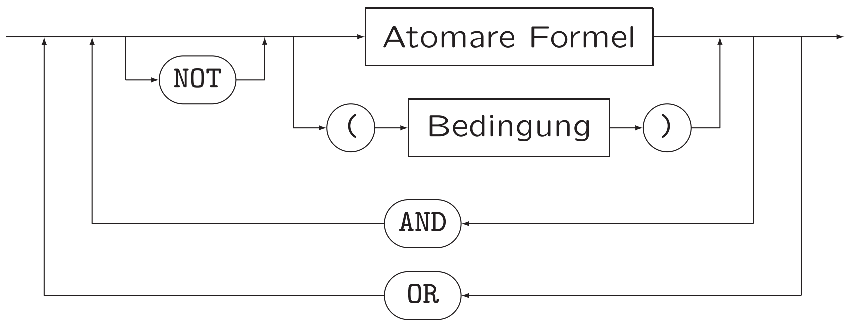
STUDENTEN		
SID	VORNAME	NACHNAME
101	Lisa	Weiss
102	Michael	Grau
103	Daniel	Sommer
104	Iris	Winter

VORN.='Lisa'	VORN.='Iris'	WHERE
wahr	falsch	falsch
falsch	falsch	falsch
falsch	falsch	falsch
falsch	wahr	falsch

- Die obige Bedingung ist inkonsistent. Hier muß "OR" verwendet werden, nicht "AND".

Bedingungen (6)

Bedingung:



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

Bedingungen (7)

- 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.

Vergleiche (1)

Atomare Formel (Form 1):

- 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
```

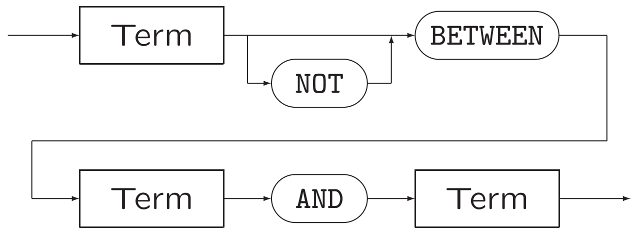
- Man kann angehängte Leerzeichen auch löschen:
  - ◇ `TRIM(TRAILING ' ' FROM NACHNAME)` in SQL-92 (funktioniert in MySQL)

    Wird in Oracle, DB2, SQL Server, Access nicht unterstützt.

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

BETWEEN-Bedingungen

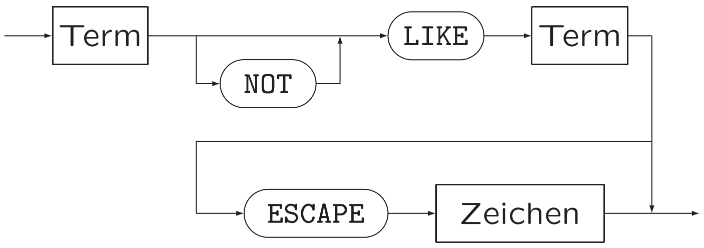
Atomare Formel (Form 2):



- `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 $\emptyset$):

```
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.

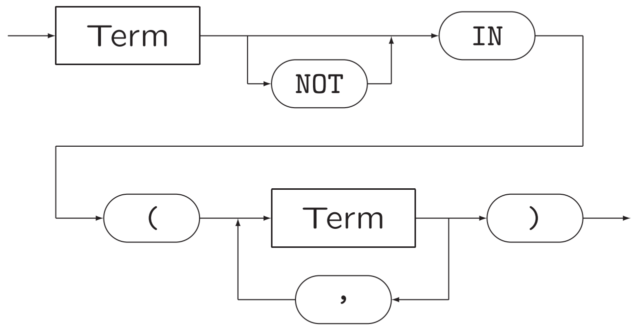
  Dies verletzt den Standard.

- 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).

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.

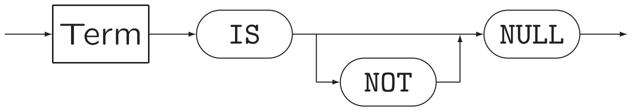
- "`AND`" / "`OR`" leiten den Wahrheitswert "`unbekannt`" weiter, außer das Ergebnis ist klar:
  Z.B. "`wahr OR unbekannt = wahr`".

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)

- Übung: 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?

- `CHECK`-Integritätsbedingungen sind erfüllt, wenn die Bedingung den Wert "unbekannt" hat.

  Sie sind nur verletzt, wenn die Bedingung falsch ist.

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” ($\exists X: \ldots$) die Verwendung der normalen logischen Äquivalenzen erlauben.

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!

Ist DISTINCT nötig? (1)

Hinreichende Bedingung für überflüssiges DISTINCT:

- Sei $\mathcal{K}$ die Menge der Attribute, die als Ausgabe-Spalten unter SELECT auftreten.

  Die Elemente von $\mathcal{K}$ haben die Form "Tupelvariable.Attribut". $\mathcal{K}$ ist die Menge von Attributen, die einen eindeutigen Wert für eine gegebene Ausgabe-Zeile haben.
- Füge alle Attribute A zu $\mathcal{K}$ hinzu, für die $A = c$ in der WHERE-Bedingung auftaucht.

  Natürlich wird eine Bedingung $c = A$ genauso behandelt. Dieser Test nimmt an, daß die Bedingung eine Konjunktion ist. Bedingungen in Unteranfragen zählen nicht (Unteranfragen werden vor dem Test entfernt).

Ist DISTINCT nötig? (2)

Test für überflüssiges DISTINCT, fortgesetzt:

- Solange sich etwas ändert, mache Folgendes:
  - ◇ Füge zu $\mathcal{K}$ Attribute A hinzu, wobei $A = B$ in der WHERE-Bedingung auftaucht und $B \in \mathcal{K}$.
  - ◇ Enthält $\mathcal{K}$ einen Schlüssel einer Tupelvariable, füge alle Attribute dieser Tupelvariable hinzu.
- Enthält $\mathcal{K}$ von jeder Tupelvariable unter FROM einen Schlüssel, so ist DISTINCT überflüssig.

  Enthält die Anfrage GROUP BY, prüft man stattdessen, ob alle GROUP BY-Spalten in $\mathcal{K}$ enthalten sind.

Ist DISTINCT nötig? (3)

Beispiel:

- Betrachten Sie folgende Anfrage:

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

- Annahme: VORNAME, NACHNAME bilden zusammen einen alternativen Schlüssel für STUDENTEN.
- $\mathcal{K} := \{$S.VORNAME, S.NACHNAME, B.ANR, B.PUNKTE$\}$.
- $\mathcal{K} := \mathcal{K} \cup \{$B.ATYP$\}$ wegen Bedingung B.ATYP = 'H'.

Ist DISTINCT nötig? (4)

Beispiel, fortgesetzt:

- $\mathcal{K} := \mathcal{K} \cup \{$S.SID, S.EMAIL$\}$, da $\mathcal{K}$ einen Schlüssel von STUDENTEN S enthält (S.VORNAME und S.NACHNAME).
- $\mathcal{K} := \mathcal{K} \cup \{$B.SID$\}$ wegen B.SID = S.SID.
- Nun ist auch ein Schlüssel von BEWERTUNGEN B in $\mathcal{K}$ enthalten (B.SID, B.ATYP, B.ANR): DISTINCT unnötig.
- Wären VORNAME, NACHNAME nicht ein Schlüssel von STUDENTEN, dann wären Duplikate möglich.

  In diesem Fall könnte es jedoch sinnvoll sein Duplikate auszugeben, da Studenten in der realen Welt durch ihren Namen identifiziert werden.

DISTINCT vs. GROUP BY

- Duplikate sollten mit DISTINCT eliminiert werden, obwohl es auch mit GROUP BY funktioniert:

```
SELECT   ATYP, ANR      Schlechter Stil!
FROM     BEWERTUNGEN
GROUP BY ATYP, ANR
```

  Dies teilt die Tabelle in Gruppen von Tupeln auf: jede Gruppe enthält Tupel, die in den GROUP BY-Attributen ATYP, ANR übereinstimmen. Für jede Gruppe wird nur ein Tupel ausgegeben. Normalerweise verwendet, um Aggregationsfunktionen (SUM, COUNT) für jede Gruppe auszuwerten.

- Ich sehe dies als Missbrauch von GROUP BY an.

  GROUP BY ist jedoch flexibler als DISTINCT, wenn man nur manche Duplikate eliminieren möchte. Alte Versionen von MySQL unterstützten kein DISTINCT. Dann musste man GROUP BY verwenden.

Umbenennung von Spalten

- Um Ausgabe-Spalten umzubenennen:

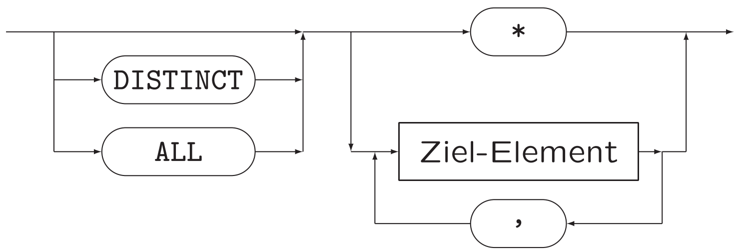
```
SELECT VORNAME AS V_Name, NACHNAME AS "Name"
FROM   STUDENTEN
```

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)

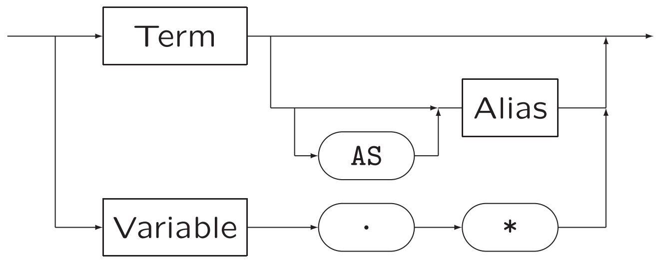
Ziel-Liste (nach SELECT):



- 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.