Vorlesung

Informatik III

Prof. Dr. Klaus-Dieter Althoff &
Dipl.-Inf. Alexandre Hanft &
Dr. Kerstin Bach

Raum A8 Spl
Email: althoff@iis.uni-hildesheim.de

Literatur

- Standardisierungsdokumente von ANSI/ISO/IEC, Link zu frei verfügbaren SQL-Standard-Dokumenten http://www.wiscorp.com/SQLStandards.html
- Can Türker: „SQL: 1999 & SQL:2003 – Objektrelationales SQL, SQLJ & SQL/XML“; dpunkt.verlag; 2003. (anhand von IBM, Oracle, Informix, PostgreSQL)
- Kap 5 SELECT in „Einstieg in SQL“, galileocomputing.de/download/dateien/2082/galileocomputing_einstieg_in_sql.pdf
- Kurzüberblick SQL: Vossen2000, Kap. 11 bzw. Vossen 2008 Kap. 5
- http://www.torsten-horn.de/techdocs/sql.htm
  - SQL-Einführung mit Vergleichen Oracle, Access, MySQL, PostgreSQL
- http://www.sqlzoo.net/
- PostgreSQL manuals http://www.postgresql.org/docs/manuals/
- Integrierte Hilfe zu PostgreSQL, pgAdmin bei PostgreSQL Installation
- „Writing PostgreSQL Functions with PL/pgSQL“ www.onlamp.com/pub/a/onlamp/2006/05/11/postgresql-plpgsql.html
- www.postgresonline.com/journal/index.php?/archives/58-Quick-Guide-to-writing-PLPGSQL-Functions-Part-1.html
- www.postgresonline.com/journal/index.php?/archives/76-Quick-Guide-to-writing-PLPGSQL-Functions-Part-2-.html
- www.postgresonline.com/journal/index.php?/archives/83-Quick-Guide-to-writing-PLPGSQL-Functions-Part-3---NOTICES,-RECURSION,-and-more.html
- 20 CodeClips auf ‚msdn solve‘ zeigen anhand des MS SQL Server 2005 eine Einführung in die durchaus komplexen Möglichkeiten und Befehle von SQL, das Gesagte gilt aber generell für die standardisierte SQL-Syntax. http://www.microsoft.com/germany/msdn/solve/knowhow/sql/default.mspx
- „Die XML-Anfragesprache XQuery im Praxiseinsatz“ Wolfgang Lezius
  XML & Web Services Magazin, Ausgabe 2/2003
- http://entwickler.de/zonen/portale/psecom,id,205,online,363,.html
- H. Faeskorn-Woyke, B. Bertelsmeier, P. Riemer, E. Bauer: Datenbanksysteme - Theorie und Praxis mit SQL2003, Oracle und MySQL, Pearson 2007

§6 SQL

- Entwicklung von SQL und Überblick
- Sprachebene: Lexikalisch / Syntaktisch-Semantisch
- SQL Datentypen, Domain Def., Tabellen Def.
- Erzeugen und Aktualisieren von Tabelleninhalten
- Anfragen
- Views
- Sequenzen
- Trigger
- Stored Procedures

Entwicklung von SQL

- Entwicklung von IBM zu Beginn der 70er Jahre
  - E.F. Codd führte Relationenalgebra ein (um 1970)
  - SQL ← Structured English Query Language / „SEQUEL“ (IBM, 1974)
    - System R von IBM war erster Prototyp der SEQUEL2 (1976)
    - Sprache für DBMS von IBM, 1981 umbenannt in SQL
- Ab 80er Jahre: Standardisierung durch ANSI
  - SQL1/SQL-86, SQL-89, **SQL2/SQL-92,** alle jeweils auch bei ANSI
  - **SQL3/SQL-99**: ANSI/ISO/IEC 9075:1999-2001
    - Aufteilung in mehrere Teilstandards
  - **SQL-2003**:
    - Sequenzgeneratoren, MERGE-Anweisung, TABLESAMPLE, MULTISET,...
    - SQL/XML: XML-Datentyp, XML-Dokumente ←→ SQL-Daten
    - SQL/MED (Management of External Data)
  - SQL:2008

SQL Heute

- SQL = Structured (/ Standard) Query Language
- Weit verbreitet, akzeptiert in Praxis, von jeder relationalen DB unterstützt
- Viele spezifische SQL-Dialekte für einzelne DBMS
- SQL umfasst generell:
  - DQL (*data query language*): Datenbankabfrage→ SELECT
  - DML (*data manipulation language*): Datenmanipulation →UPDATE,
  - DDL (*data definition language*): Datendefinition → CREATE, DROP,
  - DCL (*data control language*): Rechteverwaltung → GRANT, REVOKE
  - Programmiersprachen-Einbettungen und Anbindungen:
    (Typ Konvertierung, Tabellenzugriff via Cursor)
    - Embedded SQL für Ada, C, COBOL, Fortran, ... nicht Basic /Java (EXEC SQL ...)
    - SQL/CLI (Call-Level Interface)
- SQL-Standard Teile
  - SQL/86, SQL/89, SQL/92, SQL/99, SQL:2003– Part 1: SQL/Framework (Aufbau des Standards)
  - ***Part 2: SQL/Foundation***
  - Part 3: SQL/CLI (*call level interface*)
  - Part 4: SQL/PSM (*persistent storage modules*)

Türker2003, S. 9-12

DDL & DML (1)

- Keine strikte Trennung zwischen DDL und DML
- Dynamisches Auswerten von DDL und DML Ausdrücken zur Laufzeit
- Unterschied zu Programmiersprachen mit statischer Bindung (z.B. Java oder C):
  - Datendeklaration (entspricht DDL) werden zur Compilezeit ausgewertet
  - Operationen (entspricht DML) werden zur Laufzeit ausgewertet.
- Vorteile:
  - Deklarationen (lokale) und Operationen können zusammengefasst werden (z.B. in einer (atomaren) Transaktion)
  - Deklarationen können zur Laufzeit durch das aufrufende Programm parametrisiert werden (z.B. Bezeichner von Tabellen, Spalten, etc.)
  - Deklarationen (z.B. Views für User) können zur Laufzeit hinzugefügt/geändert werden.

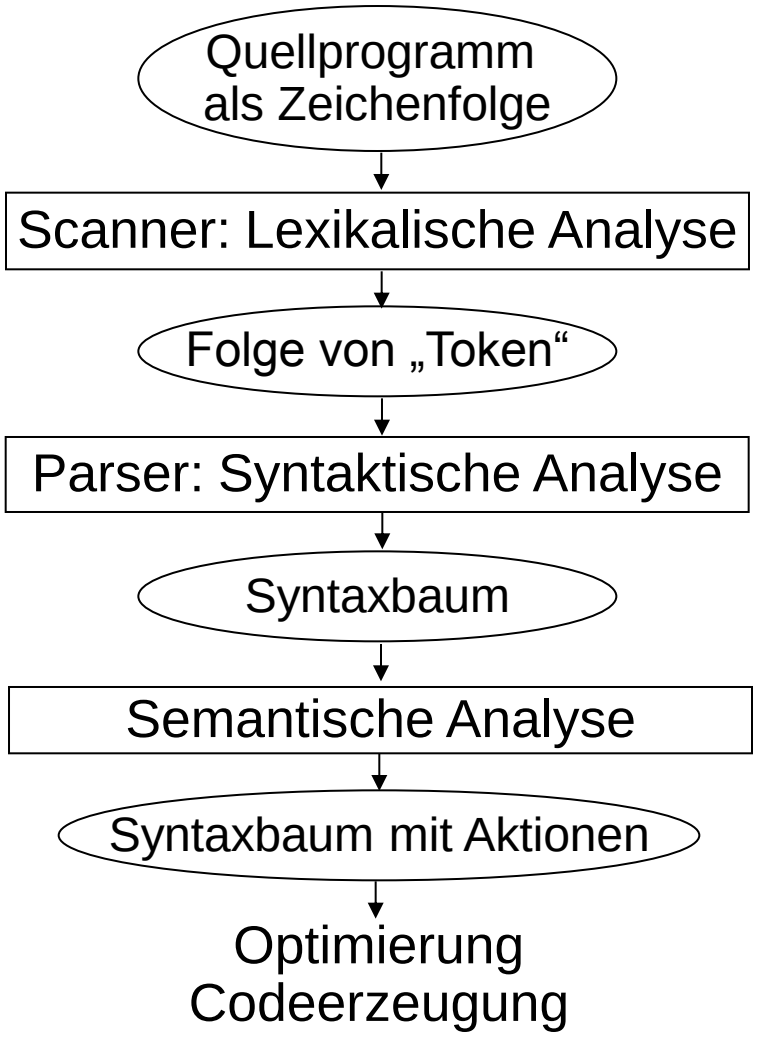
DDL & DML (2)

- Nachteile
  - Dynamische Bindung: Identifier müssen zur Laufzeit aufgelöst werden

    Beispiele für resultierende Fehler
    - Tippfehler bei Identifiern
    - Nicht deklarierte Identifier
  - Dynamische Typisierung: Kompatibilität von Ausdrücken (z.B. in Anfragen) mit den deklarierten Datenobjekten kann erst zur Laufzeit getestet werden.

    Beispiele für resultierende Fehler
    - Einzutragender Wert nicht im Wertebereich für Attribut
    - Vergleichsoperation bezieht sich auf inkompatible Typen

Schichtenmodell für Sprachanalyse / Compiler



Lexikalische Regeln

Syntaktische Regeln, z.B. in BNF

Bedeutung der Konstrukte, textuelle Beschreibung

Lexikalische Sprachebene

- Lexikalische Analyse zerlegt den SQL-Quelltext in eine Folge von Token:
  - Nicht-druckbare Steuerzeichen (z.B. Zeilenvorschub) und Kommentare werden wie Leerzeichen behandelt.
  - Klein/Grossbuchstaben werden nicht unterschieden, außer in Stringkonstanten und in „delimited identifier“; Kleinbuchstaben werden in Großbuchstaben umgewandelt.
  - Kommentare, zwei Arten
    - beginnen mit „--“ und reichen bis zum Zeilenende
    - werden in „/*“ „*/“ eingeschlossen.

Beispiel

- SQL Anweisung:
  ```
  SELECT Name FROM person_table
  WHERE Alter >= 18;   /* bestimme Volljährige */
  ```

- Besteht aus folgenden 9 Token:

SELECT	Schlüsselwort
Name	Identifier
FROM	Schlüsselwort
person_table	Identifier
WHERE	Schlüsselwort
Alter	Identifier
>=	Symbol (Operator)
18	Literal
;	SQL Special Character

Beachte: Tokenfolge ist unabhängig von der Gross/Kleinschreibung, unabhängig von Kommentaren oder Formatierung

SQL Token

- Ein Token ist:
  - Keyword
    - Reserved Keywords, Non-reserved Keywords
  - Identifier
    - Regular Identifier, Delimited Identifier
  - Literal
    - Numeric Literal, String Literal, Bit String Literal, Temporal Literal, Boolean Literal
  - SQL Special Characters oder
  - Symbol

Keywords

- Keywords sind die Basiselemente der SQL Sprachkonstrukte
- Reserved Keywords
  - SQL2011 hat mehr als 200 Reserved Keywords
  - Dürfen nicht als Regular Identifier verwendet werden
  - Beispiele: SELECT, CREATE, AND, INT, ...
- Non-Reserved Keywords
  - Dürfen als Identifier verwendet werden
  - Können durch Herstellerspezifische Keywords erweitert werden
  - Beispiele: ABS, MIN, SUM, ...

Identifier

- Identifizieren SQL Objekte wie z.B.: Attribute (Spalten), Tabellen, Schemata, Kataloge,...
- Regular Identifier:
  - Zeichenfolge aus Buchstaben (inkl. einiger Sonderzeichen, abh.von Zeichensatz), Ziffern und dem _ Zeichen
  - Muss mit Buchstaben beginnen
  - Beispiele: Namen, Patienten_Anzahl, MinWert1, ...
  - Gross/Kleinschreibung wird nicht unterschieden (interne Konvertierung in Grossbuchstaben)
- Delimited Identifier:
  - Beliebiger String in doppelten Anführungszeichen: "..."
  - Gross/Kleinschreibung wird unterschieden!
  - String darf ein Schlüsselwort sein.
  - Beispiel: "NAMEn", "SELECT", "ABS", ...
- Beachte: Namen, NAmEn, "NAMEN", bezeichnen dasselbe Objekt. "Namen" und "NAMEN" bezeichnen unterschiedliche Objekte.

PostgreSQL: akzeptiert nur Kleinschreibung

SQL Special Characters / Symbole

- SQL Special Character:
  - Zeichen mit spezieller Bedeutung
  - Diese sind folgende: ( ) " ´ % & * / + - , . : ; < > ? [ ] _ | = { } ^
- SQL Symbole:
  - spezielle Zeichenfolgen

    <> ungleich

    >= größer oder gleich

    <= kleiner oder gleich

    || Konkatenation

    -> Pfeil rechts

    => Schlüsselwortparameter

Literale (1)

- Numeric Literals
  - Darstellung eines numerischen Datenwertes
  - Festkommazahl: z.B. 65 -345 32.45 -542.34
  - Fließkommazahl: z.B. 1.234E+5
- String Literals
  - Darstellung eines Strings, in einfachen Anführungszeichen: '...'
  - z.B. 'Peter Müller'
  - Angabe von Zeichensätzen für Strings möglich durch: _<Zeichensatzname>'Zeichenfolge'
  - z.B. _LATIN1'Hello' ist der String 'Hello' im Zeichensatz LATIN1
- Bit Strings
  - Darstellung binärer Daten:
  - binäre Darstellung durch B'Bitfolge', z.B. B'00110111'
  - Hexadezimale Darstellung durch X'Hexdigitfolge', z.B. X'37fa'

Literale(2)

- Temporal Literals
  - Darstellung von Zeitangaben
  - Datumsangaben: DATE 'yyyy-mm-dd', z.B. DATE '2002-09-01', auch (in PostgreSQL): '2002-09-01' oder DATE('2002-09-01')
  - Zeitabgaben: TIME 'hh:mm:ss' oder TIME 'hh:mm:ss.nn', z.B. TIME '18:45:12.33'
  - Zeitpunktangaben: TIMESTAMP 'yyyy-mm-dd hh:mm:ss.nn' (.nn optional), z.B. TIMESTAMP '2002-09-01 18:45:12.33'
  - Zeitintervalle: INTERVAL 'yyyy' YEAR .... INTERVAL 'ss' SECOND z.B. INTERVAL '4 ' YEAR oder INTERVAL '32 ' MINUTE
    - Funktioniert erst ab PostgresSQL 8.4, vorher nur SECOND: (Angabe von MINUTE …→ 0), Abhilfe: INTERVAL '0:32'
- Boolean Literals:
  - Darstellung von Wahrheitswerten: TRUE FALSE UNKNOWN

Inhalt

Syntaktische und Semantische Sprachebene

- Wir führen syntaktische Konstrukte zusammen mit deren Bedeutung ein
- Verzicht auf vollständige Syntaxdiagramme
  - Bei Bedarf siehe SQL-2011 Buch und herstellerspezifische Spezifikation
- Betreffen die Ebenen
  - DQL,
  - DDL
  - DML

Verwendete BNF Elemente

	GROSS geschriebene Worte sind Sprachelemente, die genau so geschrieben werden müssen (Terminalsymbole)
<...>	Platzhalter für anderes syntaktisches Konstrukt (Nicht-Terminal Symbole)
[...]	Optionale Tokenfolge
...|...	Alternative
{...}	Muss einmal auftreten
{...}*	Muss einmal oder mehrfach auftreten
	Alle anderen Symbole z.B. + - * / ( ) , ; sind Sprachelemente die genau so geschrieben werden müssen (Terminalsymbole)

Beispiel:

Parameterliste (<Literal> [{ , < Literal > }*])

erlaubt z.B. (12) oder (12,3,65,754), aber nicht ()

SQL Schema

- Ein SQL Schema ist ein dynamischer Sichtbarkeitsbereich für die Namen der enthaltenen Objekte, wie Tabelle, Sichten, Triggers, ...
- Jedes Schema ist in einem Katalog abgelegt.
- Zuweisung von Namen werden explizit erzeugt (CREATE) und können wieder aufgehoben werden (DROP).

```
CREATE SCHEMA <identifier>
DROP SCHEMA <identifier> { RESTRICT | CASCADE }
```

- Die Schemas eines Katalogs müssen unterschiedliche Namen haben.
- Löschen eine Schemas löscht alle enthaltenen Objekte mit.
- Es kann Querbezüge zwischen Objekten unterschiedlicher Schemas geben.
  - RESTRICT: Schema darf keine Objekte enthalten und von keinem anderen Schema referenziert werden
  - CASCADE: Alle enthaltenen Objekte und Objekte, die andere Objekte dieses Schemas referenzieren werden gelöscht.

Beispiel

```
CREATE SCHEMA Universitaet;
        CREATE TABLE Fachbereiche .... ;
        CREATE TABLE Personen .... ;
        CREATE TABLE Studenten .... ;
        CREATE VIEW Dekan ... ;

CREATE SCHEMA Verwaltung;
        CREATE TABLE Personen .... ;
        CREATE TABLE Dezernate .... ;
...
DROP SCHEMA Universitaet CASCADE;
```

Referenzierung von Objekten

- Innerhalb eines Schemas können Objekte durch ihren Identifier bezeichnet werden.
- Ausserhalb ist eine mehrstufige Qualifizierung nötig durch:

  <schema identifier>.<objekt identifier>

  oder

  <katalog identifier>.<schema identifier>.<objekt identifier>

- Beispiel

  Verwaltung.Personen

  Universitaet.Personen

SQL Domains

- SQL besitzt vordefinierte Basisdatentypen, für die folgendes definiert ist:
  - Schreibweise für Literale (siehe Literaldefinitionen)
  - Operatoren und Auswertungsregeln für Operatoren
  - Typkompatibilitätsregeln für Ausdrücke mit mehreren Typen
  - Konvertierungsregeln zur Transformation in von einem Typ in einen anderen
  - Speichergröße, inkl. Präzision

Numerische Datentypen

- Ganze Zahlen (Bereich z.T. Herstellerabhängig)

  INTEGER INT BIGINT SMALLINT (INT=INTEGER)

- Festkommazahlen

  NUMERIC NUMERIC (<p zahl>) NUMERIC (<p zahl>, <s zahl>)

  DECIMAL DECIMAL(<p zahl>) DECIMAL (<p zahl>, <s zahl>)

  <p zahl> Ganze Zahl (precision): Gesamtzahl der Ziffern

  <s zahl> Ganze Zahl (scale): Anzahl der Ziffern hinterm Komma

  Ohne Angaben = Herstellerspezifischer Defaultgrößen.

- Gleitkommazahlen

  REAL DOUBLE PRECISION feste Zahlengrößen

  FLOAT (<p zahl>)

  <p zahl> Ganze Zahl (precision): Gesamtzahl der Bits zur Zahlendarstellung

Operatoren auf Numerischen Datentypen

- Mit Operatoren können Ausdrücke gebildet werden:
  - Binäre Operatoren: + - * / (* / haben Priorität)
  - Monadische Operatoren: - +
  - Funktionen: ABS(<operand>) MOD(<op1>,<op2>) ...
  - Binäre Vergleichsoperatoren: = < > <= >=
- Explizite Typumwandlung

  CAST (<operand> as <cast target type>)

  Typumwandlung des Ausdrucks <operand> in <cast target type>

Beispiele

-159.74 ist ein gültiges Literal für DECIMAL(5,2)

CAST(12 AS DECIMAL(6,1)) liefert 12.0

CAST(12 AS DECIMAL(3,1)) / 5 liefert 2.4

CAST(12/5 AS DECIMAL(3,1)) liefert 2.0

(meistens, abhängig vom DBMS)

MOD(5*7, 6-2) liefert 3

5+7 < 9+4 liefert TRUE

CAST(1.7 AS INTEGER) liefert 1 oder 2

je nach DBMS (rounding oder truncating)

Bitstring Datentypen

- Bitfolgen fester Länge BIT(<n zahl>)

  <n zahl> Bitanzahl

- Bitfolge variabler Länge: BIT VARYING (<n zahl>)

  <n zahl> Maximale Bitanzahl

- Literale in Binär (B´01001´) oder Hexdarstellung (X´44´)
- Operationen:
  - Konkatenation (binär): || Bitfolgen werden hintereinander geschrieben.
  - Teilfolge: SUBSTRING (<bit op> FROM <start zahl> [FOR <len zahl>])

    Extrahiert <len zahl> Bits aus <bit op> startend von <start zahl>
  - Bitanzahl: BIT_LENGTH (<bit op>)
- Typumwandlung mit CAST von
  - BIT <-> BIT VARYING (evtl. Abschneiden oder Auffüllen mit „0“ Werten)
  - BIT, BIT VARYING <-> CHARACTER Strings

CHARACTER Datentypen

- Zeichenfolgen fester Länge

  CHAR CHAR(<n zahl>)

  CHARACTER, abgekürzt CHAR

  <n zahl> Zeichenanzahl

  Bei kürzeren Texten mit „ “ aufgefüllt, aber beim Vergleich irrelevant

- Zeichenfolge variabler Länge:

  VARCHAR VARCHAR(<n zahl>)

  CHARACTER VARYING, abgekürzt CHAR VARYING, VARCHAR

  <n zahl> Maximale Zeichenanzahl, ansonsten beliebig

- Operationen (Auswahl)
  - Konkatenation: || Zeichenfolgen werden hintereinander geschrieben.
  - Vergleichsoperatoren wie bei Numbers (Lexikografische Ordnung)
  - Teilfolge: SUBSTRING (<ch op> FROM <start zahl> FOR <len zahl>)

    Extrahiert <len zahl> zeichen aus <ch op> startend von <start zahl>
  - Zeichenanzahl: CHAR_LENGTH (<bit op>)
  - Typumwandlung durch CAST in beliebige andere Datentypen

Beispiele

'abcD' || 'Fg'	liefert 'abcDFg'
SUBSTRING ('abcdefghij' FROM 3 FOR 4)	liefert 'cdef'
CHAR_LENGTH('abc')	liefert 3
CAST('12.64' AS DECIMAL(6.3))	liefert 12.640
UPPER('abc DE')	liefert ('ABC DE')
LOWER('abc DE')	liefert ('abc de')
'a' < 'b'	liefert TRUE

Temporale Basistypen

- Datums Type
  DATE
- Zeit Type
  TIME TIME(<fraction zahl>)
  <fraction zahl> Anzahl der Ziffern für Sekundenbruchteile
- Zeitpunkte bestehend aus Datum und Uhrzeit
  TIMESTAMP TIMESTAMP(<fraction zahl>)
- Zeitintervalle verschiedener Genauigkeit, z.B.
  INTERVAL <Value_as_Stringliteral> YEAR| MONTH| DAY| HOUR|MINUTE|SECOND
- Operationen
  - Vergleichsoperatoren wie bei Numbers (zeitliche Ordnung)
  - Arithmetik:
    + - Operatoren: DATE/TIME/TIMESTAMP { + | -} INTERVAL
    * / Operatoren: INTERVAL { * | / } NUMBER
  - Typumwandlung durch CAST in andere Datentypen, z.B. CHARACTERs

Beispiele

TIME '13:45:30.3952' ist ein Literal vom Typ TIME(4)

DATE '2002-01-10' + INTERVAL '22' DAY
liefert DATE '2002-02-01'

TIMESTAMP '2002-01-01 13:45:00' < TIMESTAMP '2002-02-01 10:15:39'
liefert TRUE

INTERVAL '5' SECOND * 3
liefert INTERVAL '15' SECOND liefert '00:00:15'

BOOLEAN Basistyp

- Boolean Typ

  BOOLEAN

- Operatoren
  - Vergleichsoperatoren: hier gilt TRUE > FALSE
  - Logische Verknüpfungen: AND OR NOT
  - Test auf festen Wert:

    IS TRUE IS FALSE IS UNKNOWN

    IS NOT TRUE IS NOT FALSE IS NOT UNKNOWN

- Beispiele

  (TRUE AND FALSE) IS TRUE liefert FALSE

  TRUE AND UNKNOWN liefert UNKNOWN

Nullwerte und Wahrheitswerte

- In SQL gibt es einen Nullwert NULL
- NULL ist Bestandteil jedes Basistypen
- Nullwerte können aber explizit ausgeschlossen werden, z.B. durch INTEGER NOT NULL
- NULL kann auch als Ergebnis eines Ausdruckes entstehen. Abfrage mit IS NULL und IS NOT NULL
- Ein nicht *BOOLEAN* Ausdruck ist genau dann NULL wenn der Wert *eines* Operanden NULL ist.
- Bei BOOLEAN Datentypen entspricht NULL dem UNKNOWN
- Bei der Einfügung von UNKNOWN in Boole'sche Logik spricht man auch von 3-wertiger Logik

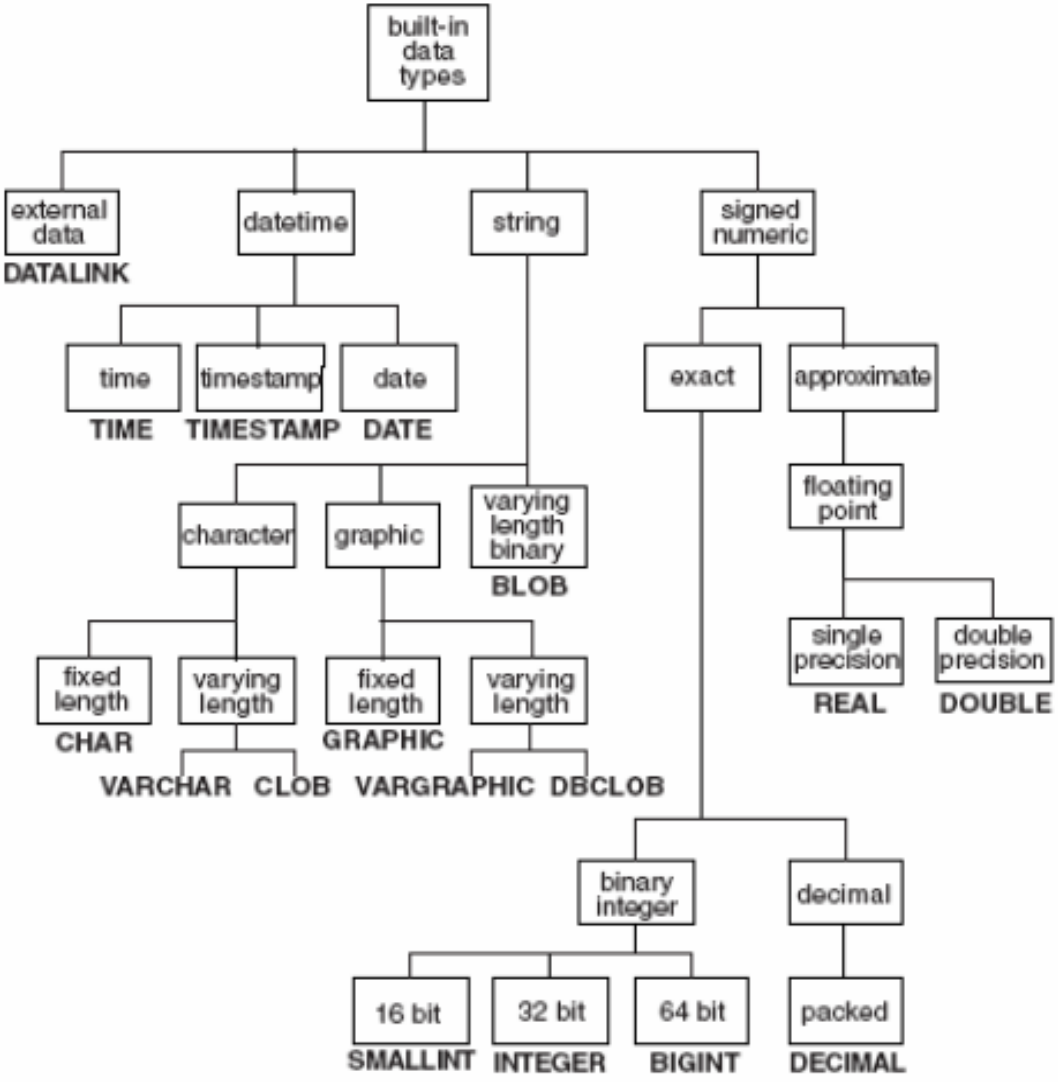
Wahrheitstabellen der 3-wertige Logik

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

X	NOT X	X IS UNKNOWN	X IS NOT UNKNOWN	X IS TRUE
TRUE	FALSE	FALSE	TRUE	TRUE
FALSE	TRUE	FALSE	TRUE	FALSE
UNKNOWN	UNKNOWN	TRUE	FALSE	FALSE

Eingebaute Datentypen (SQL Reference Vol. 1)



SQL DOMAINS -

- Benutzerdefinierte Wertebereiche:

  CREATE DOMAIN <dom identifier> [AS] <type identifier>
  [{ [CONSTRAINT <con identifier>] CHECK (<bool expr>) }*]

  – <dom identifier> ist der Name des neuen Domains
  – <type identifier> ist der Name eines der Basisdatentypen (Obermenge des neuen Wertebereiches)
  – AS ist ohne Bedeutung und kann weggelassen werden
  – Constraints schränken die Menge der zulässigen Werte weiter ein
    - mit CONSTRAINT <con identifier> bekommt ein Constraint einen Namen (nicht unbedingt nötig)
    - <bool expr> ist ein Boole'scher Ausdruck,
      für einen Wert (der zu definierenden Domain) wird der Bezeichner VALUE verwendet.

- Löschen von Domains

  DROP <dom identifier> {RESTRICT | CASCADE}

Beispiele

```
CREATE DOMAIN Name_Type AS VARCHAR(20);

CREATE DOMAIN Geb_Jahr AS DECIMAL(4)
    CHECK (VALUE > 1850 AND VALUE < 2050);

CREATE DOMAIN Geschlecht CHAR(1)
    CHECK ( VALUE IN ('m', 'w') OR
                VALUE IS NULL );

CREATE DOMAIN Vorzeichen AS INT
    CONSTRAINT Keine_Null CHECK (VALUE IS NOT NULL)
    CONSTRAINT Plus_Minus CHECK (VALUE = 1 OR VALUE = -1);
```

SQL TABLE

- SQL TABLE ist eine Datenbanktabelle und wird im Kontext eines Schemas definiert.
- Tabellendefinition hat:
  - Tabellennamen
  - Mehrere Spalten, jeweils mit Name und Typ
- Tabelle hat mehrere Datenzeilen, sind ungeordnet
- Achtung: Im Gegensatz zu einer Relation sind hier **Duplikate erlaubt** und werden unterschieden! Tabellen sind daher Multimengen.
- Beispiel:

```
CREATE  TABLE Student
    ( Name VARCHAR(30), MatrNr DECIMAL(6),
      GebDat DATE,
      PRIMARY KEY (MatrNr) );
```

CREATE TABLE Syntax(1)

```
CREATE [ { GLOBAL | LOCAL } TEMPORARY ] TABLE <tab name>
    ( { <column def> | <tab constraint> }
      [ { , { <column def> | <tab constraint> } }* ] )
```

Vererbung nur in PostgreSQL: `[ INHERITS ( parent_table [, ... ] ) ]`

- <tab name> ist der Name der Tabelle; kann auch ein qualifizierter Name sein: <schema identifier>.<tabellen identifier>
- <column def> legt eine Spalte fest (siehe nächste Folie)
- <tab constraint> legt Einschränkungen, wie Werteinschränkungen, Schlüssel und Fremdschlüssel über mehrere Spalten fest

- Tabellen sind normalerweise persistent wenn die Option [{ GLOBAL | LOCAL } TEMPORARY] nicht verwendet wird.
  Mit der Option werden temporäre Tabellen (z.B. für Zwischenergebnisse) definiert:
  - GLOBAL TEMPORARY: in allen Client Modulen bekannt
  - LOCAL TEMPORARY: nur im definierenden Client Module bekannt.

CREATE TABLE Syntax(2)

<column def> ::= <col identifier> { <type identifier> | <dom identifier> }
[DEFAULT <value> | nextval('<seq_name>')] [{ <col constraint> }*]

- < col identifier > ist der Name der Spalte (Attributname)
- der Typ der Spalte wird festgelegt durch
  - <type identifier> legt fest, dass der Typ einer der Basistypen ist. ODER
  - <dom identifier> legt fest, dass der Typ ein vorher definierter Domain ist.
- DEFAULT legt neuen Wert wenn bei INSERT nicht gegeben, als <value> oder aufgrund des nächsten Wertes der Sequenz <seq_name> an, (siehe Folie Sequenzen S.93)
- <col constraint> legt Einschränkungen, wie Werteinschränkungen, Schlüssel und Fremdschlüssel fest, die sich nur auf die aktuelle Spalte beziehen (siehe nächste Folie)

CREATE TABLE Syntax(3)

```
<col constraint> ::= [ CONSTRAINT <con identifier> ]
    { UNIQUE |
      PRIMARY KEY |
      REFERENCES <tab name> [ ( <col identifier> ) ] |
      NOT NULL |
      CHECK (<bool expr>) }
```

- mit CONSTRAINT <con identifier> bekommt ein Constraint einen Namen (nicht unbedingt nötig)
- UNIQUE: Spalte erfüllt die Schlüsselbedingung; jeder Wert kommt nur einmal vor
- PRIMARY KEY: Spalte ist Primärschlüssel
- REFERENCES: Spalte ist Fremdschlüssel aus Tabelle <tab name>; wenn das Attribut in der anderen Tabelle einen anderen Namen hat, so wird der mit <col identifier> angegeben.
- NOT NULL besagt, dass ein NULL Wert nicht zulässig ist
- CHECK legt eine beliebige Bedingung (über dem Spaltennamen) fest, die erfüllt sein muss.

CREATE TABLE Syntax(4)

```
<tab constraint> ::= [ CONSTRAINT <con identifier> ]
    { UNIQUE (<col identifier> [ { , <col identifier> }* ] ) |
      PRIMARY KEY (<col identifier> [ { , <col identifier> }* ] ) |
      FOREIGN KEY (<col identifier> [ { , <col identifier> }* ] )
         REFERENCES <tab name> [(<col identifier> [{ ,<col identifier> }*] )] |
      CHECK (<bool expr>) }
```

- Ähnliche Bedeutung wie <col constraint>.
- Hier können jedoch mehrere Spalten (Attribute) betroffen sein. Diese müssen daher explizit genannt werden, durch (<col identifier> [{ , <col identifier> }*])
- Fremdschlüssel werden hier mit FOREIGN KEY eingeleitet; zunächst stehen die Spalten der eigenen Tabelle, die Fremdschlüssel sind, danach die Tabelle auf die sie sich beziehen (und ggf. abweichende Spaltennamen) dieser Tabelle.
- CHECK legt eine beliebige Bedingung (über jeder Zeile) fest, die erfüllt sein muss.

Beispiele

```
CREATE TABLE Tab1
    (A1 INT PRIMARY KEY,
     A2 VARCHAR(20) );

CREATE DOMAIN NewDate AS DATE
     CHECK (VALUE IS NULL OR VALUE >= '2002-01-01');

CREATE TABLE Tab2
    (A3 INT REFERENCES Tab1(A1),
     A4 INT CONSTRAINT PositiveZahl CHECK (A4 > 0),
     A5 NewDate,
    PRIMARY KEY (A3,A4) );

CREATE TABLE Tab3
    (A6 INT,
     A7 INT,
     A8 VARCHAR(8),
     PRIMARY KEY (A6,A7),
     FOREIGN KEY (A6,A7) REFERENCES Tab2(A3,A4) );
```

Lieferantenbeispiel in SQL

L = Lieferantenname
O = Ort
K = Entfernung in Km
T = Teil
A = Anzahl

Lieferung = ({ L, T, A }, {{ L, T} → { L, T, A }})
Lieferant = ({ L, O }, {{L} → {L,O}})
Distanz = ({ O, K } {{O} → { O,K }})
Fremdschlüssel: Lieferung[L] ⊆ Lieferant[L] , Lieferant[O] ⊆ Distanz[O]

```
CREATE SCHEMA Lieferantenbeispiel;
    CREATE TABLE Distanz (
                Ort VARCHAR(20) PRIMARY KEY
                Distanz REAL CHECK(Distanz >= 0) );
    CREATE TABLE Lieferant (
                Lieferantenname VARCHAR(20) PRIMARY KEY,
                Ort VARCHAR(20) REFERENCES Distanz );
    CREATE TABLE Lieferung (
               Lieferantenname VARCHAR(20) REFERENCES Lieferant,
               Teil VARCHAR(15),
               Anzahl INT CHECK(Anzahl > 0),
               PRIMARY KEY (Lieferantenname , Teil) );
```

Verändern von Tabellen(strukturen)

```
ALTER TABLE <tab name>  <action [, ... ]> |
ALTER TABLE [ ONLY ] name [ * ]
        RENAME [ COLUMN ] column TO new_column |
ALTER TABLE name RENAME TO new_name |
ALTER [ COLUMN ] column TYPE type | ...
```

<action> =

ADD [COLUMN] *column type* [*column_constraint* [...]] |
DROP [COLUMN] *column* [RESTRICT | CASCADE] |
DROP CONSTRAINT *constraint_name* [RESTRICT | CASCADE] |
OWNER TO *new_owner* | ALTER [COLUMN]... | ...weitere actions

- <tab name> ist der Name der Tabelle
- Bsp: ALTER TABLE student DROP COLUMN sex;

Inhalt

Erzeugen Tabelleninhalten

- Bisher: überwiegend DDL Sprachelemente
- Erzeugen und aktualisieren von Tabelleninhalten ist in DML
- Erste Form von INSERT

```
INSERT INTO <tab name> [ ( <col identifier> [ { , <col identifier> }* ] ) ]
VALUES ( <expr> [ { , <expr> }* ] )
            [ { , (<expr> [ { , <expr> }* ] ) }* ]
```

  - mit <tab name> gibt die Tabelle an, in der eingefügt werden soll
  - mit der <col identifier> list kann eine Folge von Attributen angegeben werden, die die Reihenfolge angibt, in der die Werte zugewiesen werden; nicht alle Attribute müssen vorkommen (→ default/NULL Werte für übrige); Wenn nicht angegeben ist die Reihenfolge aus der Tabellendefinition gültig.
  - Hinter VALUES steht eine Liste von Tupeln, die in Klammern geschrieben werden. Die <expr> geben die Tupelwerte in der zuvor definierten Attributfolge an.
  - <expr> kann auch mittels eines SELECT Statements in () ermittelt werden …

Löschen von Tabelleninhalten

- DELETE Syntax:

  DELETE FROM <tab name> [WHERE <bool expr>]

  – <tab name> gibt die Tabelle an, aus der gelöscht werden soll
  – <bool expr> gibt eine Bedingung über den Attributen der Tabelle an; Es werden alle Tupel gelöscht, die diese Bedingung erfüllten.
  – In anderen DBMS auch DELETE * FROM <tab name> WHERE

Aktualisieren von Tabelleninhalten

```
UPDATE <tab name>
SET <col identifer> = <expr> [{, <col identifer> = <expr>}*]
[WHERE <bool expr>]
```

- <tab name> gibt die Tabelle an, welche aktualisiert werden soll
- <bool expr> gibt eine Bedingung über den Attributen der Tabelle an; Es werden alle Tupel aktualisiert, die diese Bedingung erfüllen.
- <col identifer> = <expr> ist die Zuweisung eines neuen Wertes an eine Spalte

Beispiele

```
CREATE TABLE Tab
   (A1 INT PRIMARY KEY, A2 INT, A3 VARCHAR(1) );

INSERT INTO Tab
   VALUES (1,2,'a'), (5,3,'b'), (6,2,'c');
INSERT INTO Tab (A3, A1)
   VALUES ('d',8), ('e',9);
```

Liefert:

Tab	A1	A2	A3
	1	2	'a'
	5	3	'b'
	6	2	'c'
	8	NULL	'd'
	9	NULL	'e'

```
DELETE FROM Tab WHERE (A3='a');
UPDATE Tab SET A1 = A1 *2
   WHERE A2 IS NULL;
```

Liefert:

Tab	A1	A2	A3
	5	3	'b'
	6	2	'c'
	16	NULL	'd'
	18	NULL	'e'

Inhalt

SQL Abfragen

- Anfragen bestehen aus Konstrukten für
  - Selektion und Projektion aus einer Tabelle
  - Join Operationen,
  - Algebraische Operationen (Vereinigung, Durchschnitt, Differenz)
  - Duplikatsbehandlung, Gruppierung und Sortierung
  - Aggregatfunktionen
  - Sichten

SELECT Statement
- erste Fassung -

- Einfache Form der SELECT Syntax

  SELECT <col identifier> [{ , <col identifier> }*]

  FROM <tab name>

  [WHERE <bool expr>]

  - Führt eine Projektion auf die Attribute <col identifier> hinter dem SELECT Schlüsselwort aus, angewendet auf Tabelle <tab name>.
  - Wenn der WHERE Zusatz enthalten ist, wird zusätzlich eine Selektion bzgl. der <bool expr> ausgeführt, d.h. es werden nur die Tupel ausgewählt, für welche die Bedingung erfüllt ist.
  - SELECT liefert eine anonyme Tabelle (Tabelle ohne Namen) zurück.
  - Auch: SFW-Block, entspricht SPJ-Ausdruck in Relationaler Algebra
- SELECT A1,A2,...,An FROM T WHERE P entspricht also: $\Pi_{A1,A2,\ldots,An}(\sigma_P(T))$

Beispiele

CREATE TABLE Tab
(A1 INT PRIMARY KEY, A2 INT, A3 VARCHAR(1));

mit folgenden Inhalt:

Tab	A1	A2	A3
	1	2	'a'
	5	3	'b'
	6	2	'c'
	8	NULL	'd'
	9	NULL	'e'

SELECT A1,A3 FROM Tab WHERE A2=2;

liefert:

A1	A3
1	'a'
6	'c'

SELECT A2 FROM Tab;

liefert:

A2
2
3
2
NULL
NULL

Beachte: Duplikate kommen hier vor!

SELECT Statement (1)
- zweite Fassung -

zu
dritter Fassung

- Erste erweiterte Form der SELECT Syntax

SELECT [ALL | DISTINCT] { <col expr> [{ , < col expr> }*] | * }
FROM <tab name> [{, <tab name>}*] [WHERE <bool expr>]

<col expr> ::= <expr> [AS <col identifier>]

  - Kombiniert Kreuzprodukt der Tabellen, mit Selektion, Projektion und Umbenennung.
  - DISTINCT eliminiert Duplikate aus dem Ergebnis; ALL (oder wenn nichts steht) belässt Duplikate in der Tabelle.
  - Stehen hinter FROM mehrere <tab name> Tabellen, so werden diese mit dem Kreuzprodukt (oder Konkatenation) unter Beibehaltung der Duplikate verknüpft.
  - Der WHERE Zusatz führt eine Selektion bzgl. <bool expr> durch.
  - AS benennt einen <col identifier> Spaltennamen um

SELECT Statement (2)
- zweite Fassung -

- <col expr> kann ein ganzer Ausdruck sein (nicht nur ein <col identifier>). Der Ausdruck wird für jedes Tupel der Tabelle berechnet und als (neues) Attribut mit dem Namen <col identifier> (hinter dem AS) zurückgegeben. Hiermit wird auch eine Umbenennung realisiert, z.B. durch A1 AS B1.
- Die <col expr> Liste kann auch ganz weggelassen werden und durch das Zeichen * ersetzt werden. Dies ist dann die Abkürzung für die Liste aller Attribute aus allen Tabellen. Hierdurch entfällt somit die Projektion.

- SELECT A1 AS B1, A2 AS B2 ,..., An AS Bn
  FROM T1,T2,..,Tm WHERE P
  entspricht also:

$$\rho_{B1,B2,\ldots,Bn \leftarrow A1,A2,\ldots,An}(\Pi_{A1,A2,\ldots,An}(\sigma_P(T1*T2*\ldots*Tm)))$$

- SELECT * FROM T1,T2,...,Tm entspricht also:
  $T1*T2*\ldots*Tm$

Beispiele (1)

```
CREATE TABLE Tab
    (A1 INT PRIMARY KEY, A2 INT, A3 VARCHAR(1) );
```

mit folgenden Inhalt:

Tab	A1	A2	A3
	1	2	'a'
	5	3	'b'
	6	2	'c'
	8	NULL	'd'
	9	NULL	'e'

```
SELECT DISTINCT A2 FROM Tab;
```

liefert:

A2
2
3
NULL

```
SELECT A1*A2 AS Produkt FROM Tab WHERE A2 IS NOT NULL;
```

liefert:

Produkt
2
15
12

```
SELECT * FROM Tab WHERE A2 IS NULL;
```

liefert:

A1	A2	A3
8	NULL	'd'
9	NULL	'e'

Beispiele (2)

```
CREATE TABLE Tab1
    (A1 INT PRIMARY KEY, A2 INT);
```

Tab1	A1	A2
	1	2
	5	3
	6	2

```
CREATE TABLE Tab2
    (A1 INT PRIMARY KEY, A3 INT);
```

Tab2	A1	A3
	1	2
	4	5

```
SELECT * FROM Tab1, Tab2;
```

liefert

Tab1.A1	A2	Tab2.A1	A3
1	2	1	2
5	3	1	2
6	2	1	2
1	2	4	5
5	3	4	5
6	2	4	5

```
SELECT Tab1.A1 AS AA, A3
FROM Tab1, Tab2
WHERE Tab1.A1 = Tab2.A1;
```

liefert

AA	A3
1	2

```
SELECT DISTINCT A2, A3 FROM Tab1, Tab2
    WHERE A2+A3<6;
```

liefert

A2	A3
2	2
3	2

SELECT Statement (1) - dritte Fassung -

Zurück zu zweiter Fassung

weiter zu vierter Fassung

- Zweite erweiterte Form der SELECT Syntax

```
SELECT [ALL | DISTINCT] { <col expr> [ { , < col expr> }* ]  | * }
FROM <tab expr> [ {, <tab expr >}* ]   [ WHERE <bool expr> ]
```

```
<tab expr> ::= { <tab name>
                    [ AS <new tab name>
                       [ (<col identifier> [ {, <col identifier>}* ] ) ]
                    ]  |
                 <join expr> }
```

  - Hinter FROM können nicht nur Tabellennamen stehen, sondern auch Join Ausdrücke <join expr>, die als Ergebnis Tabellen liefern.
  - Mit AS <new tab name> kann die Tabelle zur Bezeichnung innerhalb des SELECT Ausdruckes umbenannt werden(→Selfjoin), AS hier optional
  - Mit der Liste der <col identifier> können die Attribute zur Bezeichnung innerhalb des SELECT Ausdruckes Tabelle umbenannt werden.

SELECT Statement (2)
- dritte Fassung -

- Festlegung von JOIN Ausdrücken:

```
<join expr> ::=   { <tab expr> CROSS JOIN <tab expr> } |
                  { <tab expr> [NATURAL] [ <join type> ] JOIN <tab expr>
                        [ ON <condition> |
                              USING (<col identifier> [ {, <col identifier>}* ] ) ]  } |
                   ( <join expr> )
<join type> ::= INNER |
                { LEFT | RIGHT | FULL } [OUTER] |
                UNION
```

  - Die Joins der Tabellen werden zunächst gebildet; auf das Ergebnis werden dann die weiteren Operationen im SELECT angewandt.
  - Verschiedene Arten von Join Operationen werden unterstützt.
  - Self-Join: kein anderer Typ, nur JOIN einer Tabelle mit sich selbst

SELECT Statement (3)
- dritte Fassung -

- Kreuzprodukt: <tab expr> CROSS JOIN <tab expr>
  - Es wird das Kreuzprodukt der beiden Tabellen bestimmt.
  - Dies liefert dasselbe Ergebnis, wie SELECT ... FROM <tab expr>,<tab expr> ...
- Allgemeine JOIN Operationen
  - Es werden verschiedene **JOIN Arten** <join type> unterschieden mit denen Tabellen verknüpft werden können: INNER, OUTER, UNION
  - Verschiedene **Verknüpfungsbedingungen** können spezifiziert werden:
    - NATURAL: Tupel werden aufgrund des Übereinstimmens ALLER gleichbenannter Attribute verknüpft.
    - USING (<col identifier> [{, <col identifier>}*]): Tupel werden aufgrund des Übereinstimmens der aufgezählten Attribute <col identifier> verknüpft (müssen in beiden Tabellen vorkommen). → nur eine Spalte in Ergebnis
    - ON <condition>: Tupel der zwei Tabellen werden verknüpft, wenn die Bedingung für die Kombination beider Tupel erfüllt ist.

SELECT Statement (4)
- dritte Fassung -

- INNER Joins:
  - Zur Berechnung des Natural-, Theta- und Equi-Join der RA.
  - Ein Tupel der linken Tabelle wird mit einem Tupel der rechten Tabelle verknüpft wenn:
    - NATURAL: die Werte aller gleich bezeichneten Attribute übereinstimmen.
    - USING (<col identifier> [{, <col identifier>}*]): die Werte der aufgezählten Attribute übereinstimmen.
    - ON <condition>: die Bedingung für die Tupel wahr ist.
  - Schlüsselwort INNER kann weggelassen werden
  - R NATURAL JOIN S berechnet $R \triangleright\!\triangleleft S$
  - R INNER JOIN S USING (A) berechnet $R[R.A = S.A]S$

Beispiel

Tab1	A1	A2
	1	2
	5	3
	6	2

Tab2	A1	A3
	1	2
	4	5

```
SELECT Tab1.A1 AS AA, A2, A3
    FROM Tab1,Tab2 WHERE Tab1.A1=Tab2.A1;

SELECT Tab1.A1 AS AA, A2, A3
    FROM Tab1 CROSS JOIN Tab2 WHERE Tab1.A1=Tab2.A1;

SELECT Tab1.A1 AS AA, A2, A3
    FROM Tab1 NATURAL JOIN Tab2;

SELECT Tab1.A1 AS AA, A2, A3
    FROM Tab1 INNER JOIN Tab2 USING(A1);

SELECT Tab1.A1 AS AA, A2, A3
    FROM Tab1 INNER JOIN Tab2 ON(Tab1.A1=Tab2.A1);
```

Alle Ausdrücke sind äquivalent und liefern dasselbe Ergebnis:

AA	A2	A3
1	2	2

SELECT Statement (5)
- dritte Fassung -

- Outer Joins

  LEFT OUTER JOIN:

  – Jedes Tupel der linken Tabelle wird (falls möglich) mit allen Tupel der rechten Tabelle verknüpft.
  – Wenn es kein solches Tupel der rechten Tabelle gibt, wird das Tupel der linken Tabelle mit NULL Werten für die rechte Tabelle aufgefüllt.
  – Jedes Tupel der linken Tabelle ist im JOIN Ergebnis enthalten.
  – Beispiel

Student	Matrikel	Name
	37	Klein
	49	Lütt
	53	Klee

Info3	Matrikel	Note
	37	2
	49	3

Student LEFT OUTER JOIN Info3 USING(Matrikel) liefert alle Studenten mit Name, ggf. Info3-Note

Matrikel	Name	Note
37	Klein	2
49	Lütt	3
53	Klee	NULL

SELECT Statement (5)
- dritte Fassung -

RIGHT OUTER JOIN:

- Jedes Tupel der rechten Tabelle wird mit den Tupeln der linken Tabelle oder NULL verknüpft.

- Beispiel

Student	Matrikel	Name
	37	Klein
	49	Lütt
	53	Klee

LSFUser	Name	Login
	Lütt	luett
	Althoff	althoff
	Klee	klee

Student RIGHT OUTER JOIN LSFUser USING(Name) liefert alle LIS-Benutzer, bei Studentinnen mit MatrNr.

Name	Login	Matrikel
Lütt	luett	49
Althoff	althoff	NULL
Klee	klee	53

FULL OUTER JOIN

- Jedes Tupel der rechten Tabelle wird mit den Tupeln der linken Tabelle verknüpft und umgekehrt. Wenn es für das rechte oder das linke Tupel keinen Verknüpfungspartner gibt, wird mit NULL verknüpft.

Student FULL OUTER JOIN LSFUser USING(Name) liefert alle Personen, die als Student oder in LSF vorkommen

Name	Matrikel	Login
Klein	37	NULL
Lütt	49	luett
Klee	53	klee
Althoff	NULL	althoff

Code für vorherige Beispiele

```
CREATE TABLE student (  matrikel int4 NOT NULL, name text NOT NULL,
  CONSTRAINT student_pkey PRIMARY KEY (matrikel) );
INSERT INTO student (matrikel, name) VALUES (37, 'Klein');
INSERT INTO student (matrikel, name) VALUES (49, 'Lütt');
INSERT INTO student (matrikel, name) VALUES (53, 'Klee');

CREATE TABLE info3 ( matrikel int4 NOT NULL, note int2,
  CONSTRAINT "info3_pkey" PRIMARY KEY (matrikel),
  CONSTRAINT "fk_Student" FOREIGN KEY (matrikel) REFERENCES student (matrikel)
      ON UPDATE RESTRICT ON DELETE RESTRICT );
INSERT INTO info3(matrikel, note) VALUES (37,2); INSERT INTO info3(matrikel, note) VALUES (49,3);

CREATE TABLE lsf_user (
  name text NOT NULL,
  login text NOT NULL,
  CONSTRAINT lsf_user_pkey PRIMARY KEY (login) ) ;
INSERT INTO lsf_user (name, login) VALUES ('Lütt', 'luett');INSERT INTO lsf_user (name, login)
      VALUES ('Althoff', althoff');INSERT INTO lsf_user (name, login) VALUES ('Klee', 'klee');
```

SELECT Statement (6)
- dritte Fassung -

UNION JOIN

- Es wird kein richtiger Join durchgeführt; es gibt keine Verknüpfungsbedingung
- Es wird eine Tabelle erzeugt, die alle Zeilen und Spalten der beiden Tabellen („untereinander“) enthält
- Gleichnamige Attribute kommen auch im Ergebnis doppelt vor
- Attributwerte, die im Ursprungstupel nicht vorkommen, werden mit NULL ausgefüllt.

- Beispiel:

Tab1	A1	A2
	1	2
	5	3
	6	2

Tab2	A1	A3
	1	2
	4	5

Tab1 UNION JOIN Tab2 liefert

Tab1.A1	A2	Tab2.A1	A3
1	2	NULL	NULL
5	3	NULL	NULL
6	2	NULL	NULL
NULL	NULL	1	2
NULL	NULL	4	5

- Achtung: Es gibt auch noch UNION, siehe Folie 6-74

SELECT Statement (1) - vierte Fassung -

Ursprüngliche <tab expr> 3. Fassung

- Erweiterung der <tab expr> um <tab subquery>:

```
<tab expr> ::= { <tab name>
                    [ AS <new tab name>
                       [ (<col identifier> [ {, <col identifier>}* ] ) ]
                    ] |
                 <join expr> |
                 ( <tab subquery> )
                    [ AS <new tab name>
                       [ (<col identifier> [ {, <col identifier>}* ] ) ]
                    ]   }
```

Neuer Teil

  - Zur Bestimmung einer Tabelle kann eine Unteranfrage <tab subquery> gestellt werden.
  - Diese liefert eine Tabelle zurück, die dann in der SELECT Anfrage weiter verarbeitet werden kann.
  - Mit AS <new tab name> kann die Tabelle zur Bezeichnung innerhalb des SELECT Ausdruckes umbenannt werden.
  - Mit der Liste der <col identifier> können die Attribute umbenannt werden.

SELECT Statement (2)
- vierte Fassung -

- Aufbau von Unteranfragen:

<tab subquery> ::= <select query> |
<select query> { UNION | INTERSECT | EXCEPT }
<select query>

  - <select query> ist eine „normale“ Abfrage im SELECT Format
  - UNION bildet die Vereinigung der Tabellen aus den Selektionsergebnissen
  - INTERSECT bildet die Schnittmenge der Tabellen aus den Selektionsergebnissen
  - EXCEPT bildet die Differenz der Tabellen aus den Selektionsergebnissen, d.h. Elemente der 2. Tabellen werden aus denen der 1. Tabelle entfernt
  - Bei UNION, INTERSECT, EXCEPT müssen beide Tabellen dieselben Attribute (Namen und Typ) besitzen.

Beispiel

Tab1	A1	A2
	1	2
	5	3
	6	2

Tab2	A1	A3
	1	2
	4	5

```
SELECT * FROM Tab1 UNION SELECT A1, A3 AS A2 FROM Tab2;
```

liefert

A1	A2
1	2
5	3
6	2
1	2
4	5

```
SELECT * FROM Tab1 INNER JOIN (SELECT * FROM Tab1 EXCEPT
    SELECT A1, A3 AS A2 FROM Tab2) Tab3 ON Tab1.A1=Tab3.A1;
```

liefert

A1	A2	A1	A2
5	3	5	3
6	2	6	2

```
SELECT A1+A2 AS Summe
    FROM (SELECT * FROM Tab1 UNION SELECT A1, A3 AS A2 FROM Tab2)
    Tab3 WHERE A1+A2>6;
```

liefert

Summe
8
8
9

Bedingungen in WHERE-Klausel

Verschiedene Formen der <bool expr> in WHERE Klausel

- <col identifer> <comp op> <expr>
  - <comp op>: <,>,=, <>, >=, <=
  - Logischen Verknüpfungen: NOT AND OR (NOT bindet am stärksten)
  - Berechnungen
- <col identifer> BETWEEN value1 AND value2
- **< col identifer > LIKE <expr> für Zeichenkettenvergleiche**
  - '% ' kein o. beliebig viele Zeichen: z.B. beginnt mit W: LIKE 'W%'
  - '_' für genau ein Zeichen alle Meier: LIKE 'M__er'
- <colidentifer> IS [NOT] NULL true[false] wenn Wert in Spalte NULL
- <expr> <comp op> <Subquery> siehe folgende Folien
- <expr> [NOT] IN <Subquery>
- [NOT] { EXISTS | UNIQUE } (<tab subquery>)

SELECT Unteranfragen im WHERE Teil (1)

- In der <bool expr> im WHERE Teil des SELECT Statements können auch folgende Bedingungen vorkommen, die selbst wieder eine Unteranfrage auslösen können.

  <expr> <comp op> { ALL | ANY | SOME } (<tab subquery>)

  - <expr> ist ein Ausdruck, der einen skalaren Wert zurückliefert.
  - <comp op> ist ein beliebiger Vergleichsoperator: = <> < > <= >=
  - <tab subquery> ist eine SELECT Unteranfrage, die als Ergebnis eine Tabelle mit **einer** Spalte zurückliefern muss.
    Sonst Fehler „subquery has too many columns“
  - Variante ALL: Die Bedingung ist wahr, wenn <expr> mit ***allen Werten*** der Tabelle in der angegebenen Vergleichsrelation <comp op> steht.
  - Variante ANY/SOME (sind Synonyme): Die Bedingung ist wahr, wenn <expr> mit mind. ***einem Wert*** der Tabelle in der angegebenen Vergleichsrelation <comp op> steht.

Beispiel

Tab1	A1	A2
	1	2
	5	3
	6	2

0 < ALL (SELECT A1 FROM Tab1) liefert TRUE
5 > ALL (SELECT A1 FROM Tab1) liefert FALSE
5 > ANY (SELECT A1 FROM Tab1) liefert TRUE

Keine vollständige SQL Anfrage!
Nur eine <bool expr> in WHERE Klausel

```
SELECT * FROM Tab1 WHERE A1 >= ALL (SELECT A1 FROM Tab1);
```

Liefert

A1	A2
6	2

das Tupel, bei welchem A1 >= allen Einträgen ist (Maximum)

SELECT Unteranfragen im WHERE Teil (2)

- In der <bool expr> im WHERE Teil des SELECT Statements können auch folgende Bedingungen vorkommen, die selbst wieder eine Unteranfrage auslösen können.

  <expr> [NOT] IN { (<tab subquery>) | (<expr> [{ , <expr> }*]) }

  - <expr> ist ein Ausdruck, der einen skalaren Wert zurückliefert.
  - Die Bedingung prüft, ob der Wert des Ausdrucks in einer Menge enthalten ist. Die Menge kann entweder durch die <tab subquery> bestimmt werden **oder** in einer Liste von Werten fest angegeben sein.
  - 'IN' is äquivalent zu '= ANY‘
  - <tab subquery> ist eine SELECT Unteranfrage, die als Ergebnis eine Spalte zurückliefern muss.
  - (<expr> [{ , <expr> }*]) ist eine Liste von Werten. (nur bei IN)
  - Bedingung ist erfüllt, wenn der Wert der <expr> in der Tabelle vorkommt oder mit einem der aufgelisteten Werte übereinstimmt.
  - NOT negiert das Ergebnis, testet also auf Nicht-Enthaltensein.

SELECT Unteranfragen im WHERE Teil (3)

- In der <bool expr> im WHERE Teil des SELECT Statements können auch folgende Bedingungen vorkommen, die selbst wieder eine Unteranfrage auslösen können.

  [NOT] { EXISTS | UNIQUE } (<tab subquery>)

  – <tab subquery> ist eine SELECT Unteranfrage(auch mehrere Spalten).
  – Mit EXISTS wird geprüft, ob die Tabelle, die sich aus der SELECT Unteranfrage ergibt, mind. 1 Zeile (Tupel) enthält (Existenzquantor). In diesem Fall liefert die EXISTS Bedingung TRUE.
  – Mit UNIQUE wird geprüft, ob die Tabelle zwei (oder mehr) Zeilen (Tupel) mit demselben Inhalt enthält. Ist dies **nicht** der Fall (oder ist die Tabelle leer), so liefert die UNIQUE Bedingung TRUE.
  – Mit NOT wird jeweils die Bedingung negiert.

Beispiel

Tab1	A1	A2
	1	2
	5	3
	6	2

Tab2	A1	A3
	1	2
	4	5

EXISTS (SELECT * FROM Tab1 WHERE A1<A2) liefert TRUE

EXISTS (SELECT * FROM Tab1 WHERE A1>10) liefert FALSE

UNIQUE (SELECT * FROM Tab1) liefert TRUE

UNIQUE (SELECT A2 FROM Tab1) liefert FALSE

UNIQUE (SELECT * FROM Tab1 WHERE A1>10) liefert TRUE

Korrelation

```
SELECT A1, A2 FROM Tab1
WHERE EXISTS (SELECT * FROM Tab2 WHERE Tab1.A1=Tab2.A1);
```

liefert

A1	A2
1	2

Aggregationsfunktionen (1)

- Anstelle von <col expr> hinter SELECT sind als Ausdruck außerdem *Aggregationsfunktionen* erlaubt.
- Aggregationsfunktionen nehmen als Eingabe eine ganze Spalte und fassen diese zu einem einzelnen Wert zusammen.

```
{ COUNT | MAX | MIN | SUM | AVG }
      ( [ { DISTINCT | ALL } ] <expr>) | COUNT (*)
```

  - <expr> ist ein Ausdruck, der eine Spalte angibt. Der Ausdruck wird auf die Zwischenergebnistabelle, die durch SELECT ... FROM ... WHERE bestimmt ist, angewendet.
  - COUNT liefert die Anzahl der nicht NULL Werte,
  - MAX bzw. MIN liefert das Maximum bzw. Minimum der Werte, (auch für nicht numerische Wertebereiche, wenn vollst. totale Ordnung exisitiert)
  - AVG liefert den Durchschnitt der Werte,
  - SUM liefert die Summe der Werte, die sich nach Auswerten von <expr> ergeben

Aggregationsfunktionen (2)

- Mit ALL wird die Aggregationsfunktion auf alle nicht NULL Werte der Tabelle angewendet. Dies ist auch der Default, wenn weder ALL noch DISTINCT verwendet wird.
- Mit DISTINCT werden zuerst alle Duplikate eliminiert bevor die Aggregationsfunktion angewendet wird
- COUNT (*) zählt die Anzahl der Tupel in der Tabelle; hier werden NULL Werte mitgezählt.

Gruppierung und weitere Bedingungen

- Aggregationsfunktion nicht über alle Tupel einer Spalte, sondern über Teilgruppen von Tupeln:
- WHERE … GROUP BY <col identifier> [{, <col identifier> }*
  - <col identifier> ist/sind Gruppierungsattribut(e)
- weitere Einschränkung die für jede Tupel-Gruppe gelten muss:
- GROUP BY… [HAVING <bool_expr>]
  - <bool_expr> Selektionsbedingung bzgl. der Tupel-Gruppe
- (konzeptuelle) Ausgewertung:
  - GROUP BY Operator unterteilt Tabelle aus FROM (nach WHERE) in Tupel-Gruppen (Tupel innerhalb jeder Gruppe haben gleichen Wert für alle Gruppierungsattribut(e)), auf welche dann SELECT angewendet wird
  - Tupel-Gruppen, welche nicht <bool_expr> hinter HAVING genügen, fliegen raus
- Alle in der Select-Klausel aufgeführten Attribute müssen aggregiert werden oder in der Group By-Klausel aufgeführt werden!
- FROM → WHERE → GROUP BY → Agg → HAVING → SELECT

Beispiel

Tab1	A1	A2	A3
	1	2	7
	5	3	8
	6	2	9

```
SELECT COUNT(*) FROM Tab1 WHERE A2=2 ;
```
liefert 2

```
SELECT SUM(A2) FROM Tab1;
```
liefert 7

```
SELECT SUM(DISTINCT A2) FROM Tab1;
```
liefert 5

```
SELECT * FROM Tab1
WHERE A1 > (SELECT AVG(A1) FROM Tab1);
```
liefert

A1	A2	A3
5	3	8
6	2	9

```
SELECT AVG(A1), A2 FROM Tab1
GROUP BY A2;
```
liefert

A1	A2
5	3
3,5	2

```
SELECT AVG(A1) AS Schnitt, A2 FROM Tab1
GROUP BY A2 HAVING COUNT(*)>=2
                AND AVG(A1) >=3.5;
```
liefert

Schnitt	A2
3,5	2

```
SELECT AVG(A1), A2, A3 FROM Tab1 GROUP BY A2;   !!!
```
→ Statement fehlerhaft, da A3 weder aggregiert noch in Gruppierung!!

Sortieren

- Das Ergebnis einer SELECT Anfrage kann sortiert werden. Die SELECT Syntax wird hierzu durch folgenden Zusatz ergänzt, der hinter des WHERE Teils angefügt wird.

```
ORDER BY <col identifier> [ASC | DESC]
            [ {, <col identifier> [ASC | DESC] }* ]
```

  - ORDER BY darf nicht bei inneren SELECT Anfragen verwendet werden (Unterabfragen).
  - <col identifier> gibt einen Spaltenbezeichner der Tabelle an, nach dem sortiert wird. Werden mehrere Spalten angegeben, so wird primär nach dem ersten Spaltenbezeichner sortiert; bei Gleichheit nach dem 2. Spaltenbezeichner, usw.
  - ASC steht für aufsteigende Sortierung (ist auch Default)
  - DESC steht für absteigende Sortierung.
  - NULL ist entweder größer oder kleiner als alle anderen Werte, abhängig vom DBMS Hersteller.

Beispiel

Tab1	A1	A2
	1	2
	5	3
	6	2

SELECT * FROM Tab1 ORDER BY A2, A1;

liefert:

A1	A2
1	2
6	2
5	3

SELECT * FROM Tab1 ORDER BY A2 ASC, A1 DESC;

liefert:

A1	A2
6	2
1	2
5	3

SELECT Überblick

SELECT [ALL | DISTINCT [ON (col*expression* [, ...])]] * | col*expression* [AS *output_name*] [, ...] [
FROM *from_item* [, ...]]
[WHERE *tupelcondition*]
[GROUP BY *groupattribute* [, ...]]
[HAVING *groupcondition* [, ...]]
[{ UNION | INTERSECT | EXCEPT } [ALL] *SELECT…*]
[ORDER BY *colexpression* [ASC | DESC | USING *operator*] [, ...]]

where *from_item* can be one of:

- *table_name* [*] [[AS] *alias* [(*column_alias* [, ...])]]
- (*select*) [AS] *alias* [(*column_alias* [, ...])]
- *from_item* [NATURAL] *join_type from_item* [ON *join_condition* | USING (*join_column* [, ...])]

aus PostgreSQL Doku

Inhalt

Definition von Sichten (Views)

- Mittel der externen Ebene des ANSI 3-Ebenen-Schemas
  - Daten für bestimmten Zweck (z.B. Formular) zusammengestellt
  - Häufig für Datenaufbereitung in Data Warehouse (DWH) Applikationen u. a. IS
- Views sind virtuelle Tabellen, auch mit eigenen Benutzerrechten daran möglich
- Views gehören wie Tabellen zu SQL SCHEMAs, in DDL definiert
- Views werden über SELECT Ausdrücke definiert (vgl. RA in §5).
- Views können in SQL Anfragen wie Tabellen verwendet werden.

```
CREATE VIEW <viewname> [ ( <col identifier> [ { ,<col identifier> }* ] ) ]
AS <select query>
```

  - < viewname> ist der Name des Views
  - <col identifier> sind die Namen der Attribute; ansonsten werden die von SELECT geliefert Attributnamen der anonymen Tabelle verwendet.
  - <select query> ist ein SELECT Ausdruck, der die View definiert.
  - Änderungen in zugrundeliegenden Tabellen
    → View-Inhalt ändert sich automatisch.
- Weitherhin: Unterscheidung in virtuelle Sichten (default, s.o.) und materialisierte Sichten (Materialized View in Oracle, Summarized Table in IBM DB2)

Faeskorn-Woyke+2007, S. 226

View Beispiel

kurs	lsf	name	dozentid
	3500	Info 3	1
	3501	Info 3 Uebung	3
	3510	WBS	1
	3505	DBAE	2

PostgreSQL-spezifisch

- Einfach: `CREATE OR REPLACE VIEW kurs_nur1 AS SELECT * FROM kurs WHERE dozentid = 1;`
- mit Join:

```
CREATE OR REPLACE VIEW kurs_althoff AS
SELECT  kurs.*, dozent.name AS dozentname FROM kurs INNER JOIN
dozent ON kurs.dozentid = dozent.id WHERE dozent.name = 'Althoff';
```

Erzeugt eine virtuelle Tabelle:

kurs_althoff	lsf	name	dozentid	dozentname
	3500	'Info 3'	1	'Althoff'
	3510	'WBS'	1	'Althoff'

anschauen: `SELECT * FROM kurs_althoff;`

Wir können danach Anfragen formulieren wie:

`SELECT COUNT(*) FROM kurs_althoff;`

liefert 2

Warum Stored Procedures?

- Stored Procedures sind vorkompilierte, optimierte Blöcke von (SQL-) Anweisungen oder in anderen Sprachen (PL/…)
- Berechtigungen an UPDATE, INSERT ausschließen
  - Verhinderung SQL Injection!!
- Zwischenspeicherung von Ausführungsplänen → Geschwindigkeit erhöhen
- Netzwerkverkehr minimieren
- Output Parameter verwenden
- Logik kapseln, Übersichtlichkeit
- Festhalten, welche Spalten verwendet werden
- Quelle und Erläuterung:
  www.insidesql.de/beitraege/dev_basics/es_dynamisches_sql_fluch_und_segen.html

SQL-Injection

- bezeichnet das Einschleusen von eigenen Befehlen in eine SQL-Datenbank.
- Überprüft eine Web-Applikation Benutzereingaben nicht ausreichend oder fehlerhaft auf Funktionszeichen, ist es möglich, spezielle Zeichenketten einzuschleusen →DB verwundbar.
- Funktionszeichen: \ ‘ ;
- Mit geschickt gewählten Eingaben können dann eigene Parameter und Befehle an die Datenbank übergeben und auf deren Inhalte und sogar das System zugreifen.
- Bsp:
  - Angedacht: SELECT * FROM kunde WHERE card = '$card‘
  - Eingegeben: *';DROP TABLE KUNDE–*
  - Ergibt SELECT * FROM kunde WHERE card = '; und **DROP TABLE KUNDE--'**
- Quellen:
  - http://www.heise.de/security/artikel/43175
  - http://de.wikipedia.org/wiki/SQL-Injektion
  - http://www.unixwiz.net/techtips/sql-injection.html (SQL Injection Attacks by Example)

Sequenzen (1)

- Erzeugen einer Sequenz *name*, welche bei jedem Aufruf mit nextval einen um *incr* erhöhten Wert zurückliefert
- CREATE SEQUENCE *name*
  [INCREMENT [BY] *incr*] //default 1
  [MINVALUE *minvalue* | NO MINVALUE] //default keiner
  [MAXVALUE *maxvalue* | NO MAXVALUE] //default keiner
  [START [WITH] *start*] //default MINVALUE
  [CACHE *cache*] [[NO] CYCLE]
  - Bsp. CREATE SEQUENCE sq_kurs_id START WITH 3500;
- Abrufen des nächsten Wertes:
  - SQL:2003: NEXT VALUE FOR <sequence_name>
  - Postgresql: nextval(<sequence_name>);
    SELECT nextval('sq_kurs_id');
- Direktes Setzen des (über)nächsten Wertes:
  - setval(<sequence_name>, <value>, <incrementByNextNextval>);
  - Bsp. SELECT setval('sq_kurs_id', 3540); → nextval(...) = 3541
  - Bsp. SELECT setval('sq_kurs_id', 3540, false); → = 3540
  - setval('sq_autor_id', SELECT MAX(id) FROM author);

Türker2003, S. 96

Sequenzen (2) für Autowerte

- Für Autowert einer Tabellen-Spalte wird benötigt:
  - CREATE SEQUENCE *tablename_colname*_seq;
  - CREATE TABLE *tablename* (
    colname integer DEFAULT nextval('*tablename_colname*_seq')
    NOT NULL ,...);
- Syntactic Sugar (äquivalent):
  CREATE TABLE *tablename* (*colname* SERIAL, ...);
  - Spalte id als SERIAL im Dialog eingegeben
    → verändert SQL-Quelltext auf:
    id int4 NOT NULL DEFAULT nextval('dozent_id_seq'::regclass),
  - Kommentar "…will create implicit sequence "tablename_ colname _seq" for serial column "tablename.colname" "