Datenbank-Programmierung

Kapitel 2: Tabellendeklarationen

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Sommersemester 2024

http://www.informatik.uni-halle.de/~brass/dbp24/

Lernziele

Nach diesem Kapitel sollten Sie Folgendes können:

- `CREATE TABLE`-Anweisungen in SQL schreiben.
- Integritätsbedingungen in SQL definieren.

  `NOT NULL`, Schlüssel, Fremdschlüssel und `CHECK`.
- für ein Attribut einen Datentyp wählen.

  Sie sollten die üblichen Datentypen, die es in jedem DBMS geben sollte, aufzählen und erklären können. Sie sollten die Parameter von `NUMERIC`, `CHAR` und `VARCHAR` erklären können. Sie sollten auch wissen, welche weiteren Datentypen es noch geben könnte (für die Einzelheiten können Sie dann in der Anleitung Ihres DBMS nachschauen).
- wissen, dass es ein `ALTER TABLE` Statement gibt.

Inhalt

Beispiel-Datenbank (1)

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

Beispiel-Datenbank (2)

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

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

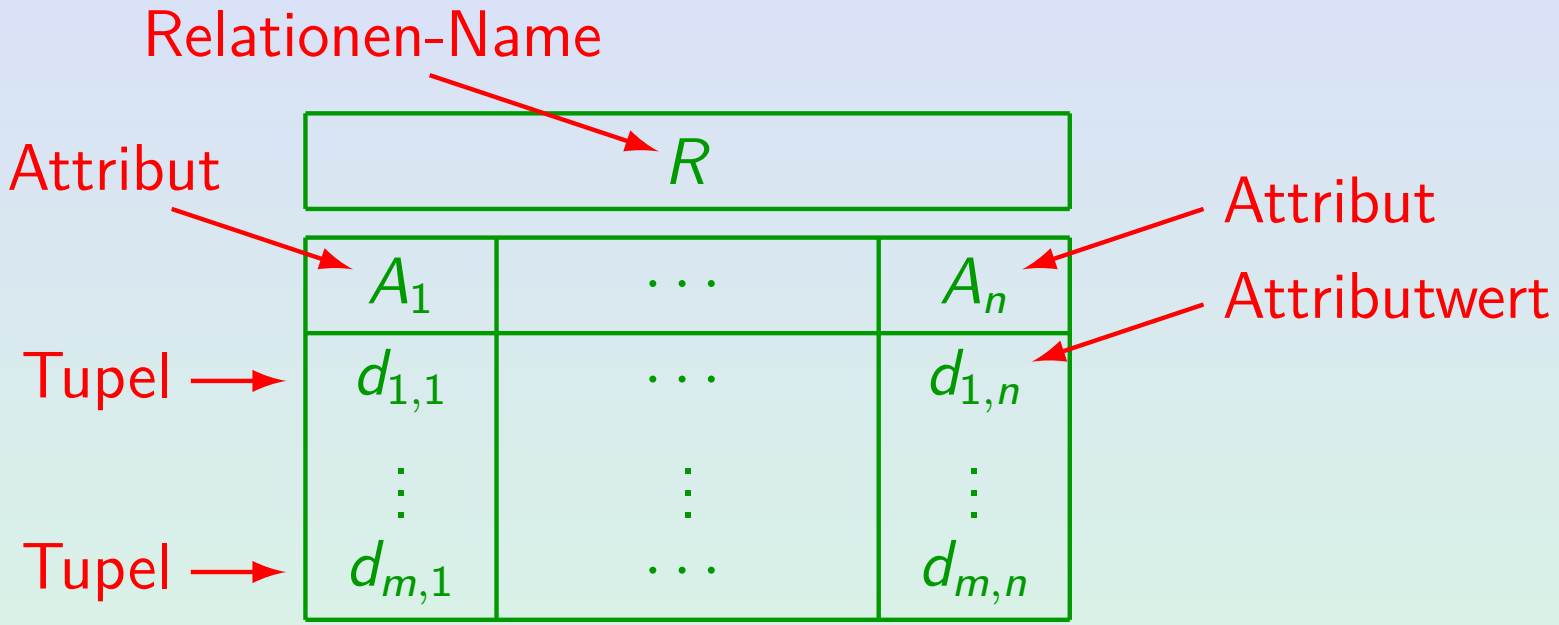
Beispiel-Datenbank (3)

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

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

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

Grundbegriffe des relationalen Modells



Synonyme: Relation und Tabelle (entsprechen Klassen).
Tupel, Zeile und Record (entsprechen Objekten).
Attribut, Spalte, Feld.
Attributwert, Spaltenwert, Tabelleneintrag.

Datentypen und Nullwerte

- Im Datenbank-Schema muss man für jede Spalte einen Datentyp festlegen.
- Alle Tabellenzeilen können in dieser Spalte nur Werte des festgelegten Datentyps enthalten, oder alternativ einen Nullwert (sofern der nicht ausgeschlossen wird).
- Ein Nullwert ist ein leerer Tabelleneintrag, verschieden von allen normalen Werten des Datentyps, also insbesondere nicht die Zahl 0 und auch nicht der leere String.

  In Oracle ist es der leere String, das verletzt den SQL Standard.
  Bei Ein- und Ausgabe sind Nullwert und leerer String aber nur umständlich zu unterscheiden, so dass man sie nicht für die gleiche Spalte erlauben sollte.

- Vergleiche mit Nullwerten geschehen in dreiwertiger Logik.

Vergleich mit Java-Klasse

```
class Student {
    private int sid;
    private String vorname;
    private String nachname;
    private String email;

    Student(int s, String vn, String nn, String e) {
        if(s < 0) // CHECK(sid >= 0)
            throw new IllegalArgumentException("...");
        this.sid = s;
        ...
    }

    int getSid() { return this.sid; }
    ...
}
```

Schlüssel (1)

- Ein Schlüssel einer Relation R ist eine Spalte A, die die Tupel/Zeilen in R eindeutig identifiziert.
- Wenn z.B. SID als Schlüssel von STUDENTEN deklariert wurde, ist dieser DB-Zustand verboten:

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

- Schlüssel sind eine Art von Integritätsbedingungen.

Bedingungen, die jeder Datenbank-Zustand erfüllen muss.

Schlüssel (2)

- Ein Schlüssel kann auch aus mehreren Attributen bestehen („zusammengesetzter Schlüssel“).

  Wenn A und B zusammen einen Schlüssel bilden, ist es verboten, dass es zwei Zeilen t und u gibt, die in beiden Attributen übereinstimmen (d.h. $t.A = u.A$ und $t.B = u.B$). Zwei Zeilen können in einem Attribut übereinstimmen, aber nicht in beiden.

- Der Schlüssel „VORNAME, NACHNAME“ ist hier erfüllt:

STUDENTEN			
SID	VORNAME	NACHNAME	EMAIL
101	Lisa	Weiss	...
102	Michael	Weiss	...
103	Michael	Grau	...

Schlüssel (3)

- Eine Relation kann mehr als einen Schlüssel haben.
- Z.B. ist `SID` ein Schlüssel von `STUDENTEN` und „`VORNAME, NACHNAME`“ ggf. ein weiterer Schlüssel.
- Ein Schlüssel wird zum „Primärschlüssel“ ernannt.

  Der Primärschlüssel sollte möglichst aus einem einzigen kurzen Attribut bestehen, das möglichst nie verändert wird (durch Updates).
  Der Primärschlüssel wird in anderen Tabellen verwendet, die sich auf Zeilen dieser Tabelle beziehen. In manchen Systemen ist Zugriff über Primärschlüssel besonders schnell. Ansonsten ist die Wahl des Primärschlüssels egal.
- Die anderen sind „Alternativ-/Sekundär-Schlüssel“.

  SQL verwendet den Begriff `UNIQUE` für alternative Schlüssel.

Fremdschlüssel (1)

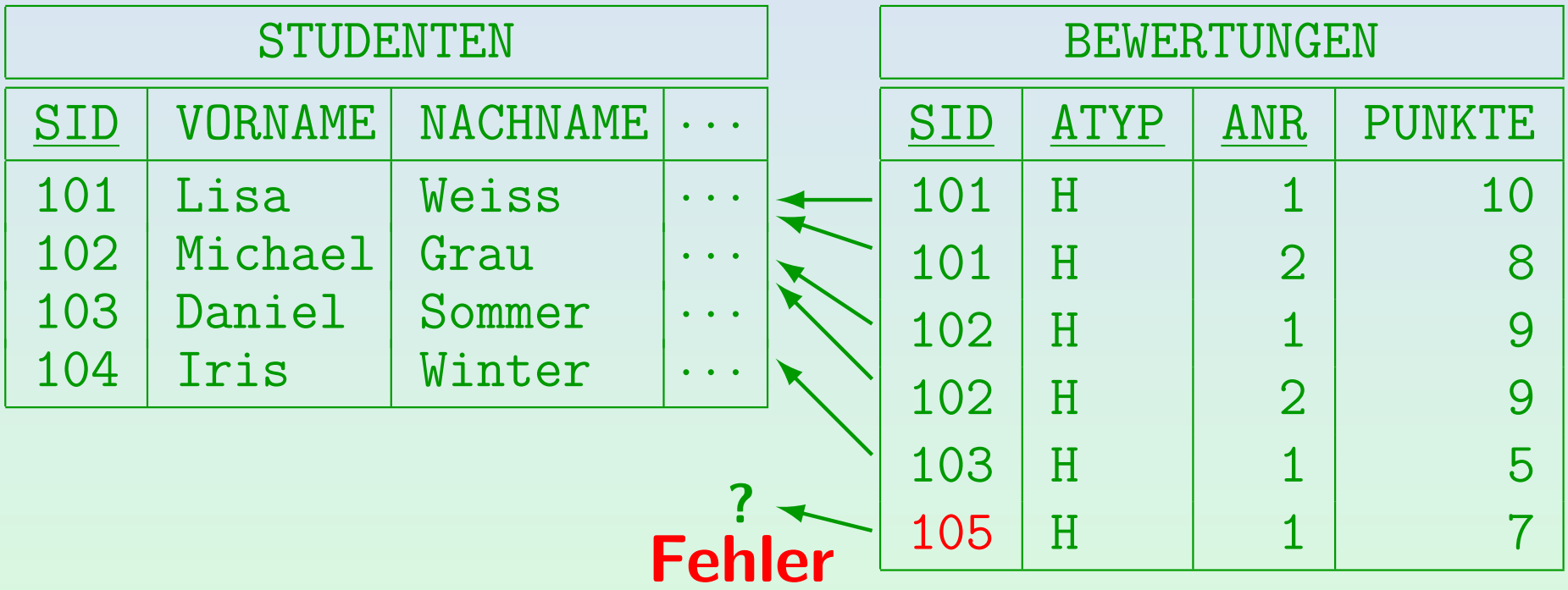
- Das relationale Modell hat keine expliziten Verweise (Zeiger, Referenzen) oder Beziehungen zwischen Tupeln.
- Schlüsselattributwerte identifizieren ein Tupel.

  Sie sind „logische Adressen“ der Tupel.
- Um sich in einer Relation S auf Tupel von R zu beziehen, fügt man den Primärschlüssel von R zu den Attributen von S hinzu. In S heißen die Attribute „Fremdschlüssel“.

  Solche Attributwerte sind „logische Zeiger“ auf Tupel in R.
- Die Fremdschlüssel-Bedingung besagt, dass Werte bzw. Wertkombinationen in Fremdschlüssel-Spalten auch tatsächlich in der referenzierten Tabelle als Schlüsselwerte vorkommen.

Fremdschlüssel (2)

SID in BEWERTUNGEN ist ein Fremdschlüssel, der STUDENTEN referenziert:



Die hier benötigte Bedingung ist, dass jeder SID-Wert in BEWERTUNGEN auch in STUDENTEN auftaucht.

Fremdschlüssel (3)

- Fremdschlüssel sind selbst keine Schlüssel!

  Die Fremdschlüssel-Bedingung hat nichts mit einer Schlüsselbedingung zu tun. Für manche Autoren ist jedoch jedes Attribut, das Tupel identifiziert (nicht unbedingt in der gleichen Tabelle), ein Schlüssel. Dann wären Fremdschlüssel Schlüssel, aber normale Schlüssel brauchen dann immer einen Zusatz („Primär-/Alternativ-“).

- Es ist Zufall, dass im Beispiel die Fremdschlüssel gleichzeitig Teil eines Schlüssels sind. Das muss nicht so sein.

- Nur Schlüssel einer Relation können referenziert werden, keine beliebigen Attribute.

- Besteht der Schlüssel der referenzierten Relation aus zwei Attributen, muss der Fremdschlüssel auch aus zwei Attributen passenden Datentyps bestehen.

Schema-Notation (1)

- In der hier verwenden Schema-Kurznotation schreibt man den Tabellennamen und dann die Spaltennamen in Klammern:

```
BEWERTUNGEN(SID → STUDENTEN,
            (ATYP, ANR) → AUFGABEN, PUNKTE)
STUDENTEN(SID, VORNAME, NACHNAME, EMAIL°)
AUFGABEN(ATYP, ANR, THEMA, MAXPT)
```

- Primärschlüssel-Attribute werden unterstrichen.

  In ASCII stellt man den Schlüsselspalten ein „#“ voran.

- Fremdschlüssel werden mit einem Pfeil und dem Namen der referenzierten Tabelle gekennzeichnet.

  Bei zusammengesetzten Fremdschlüsseln sind Klammern nötig.

- Attribute, die Nullwerte enthalten können, sind mit „°“ oder „?“ gekennzeichnet.

Schema-Notation (2)

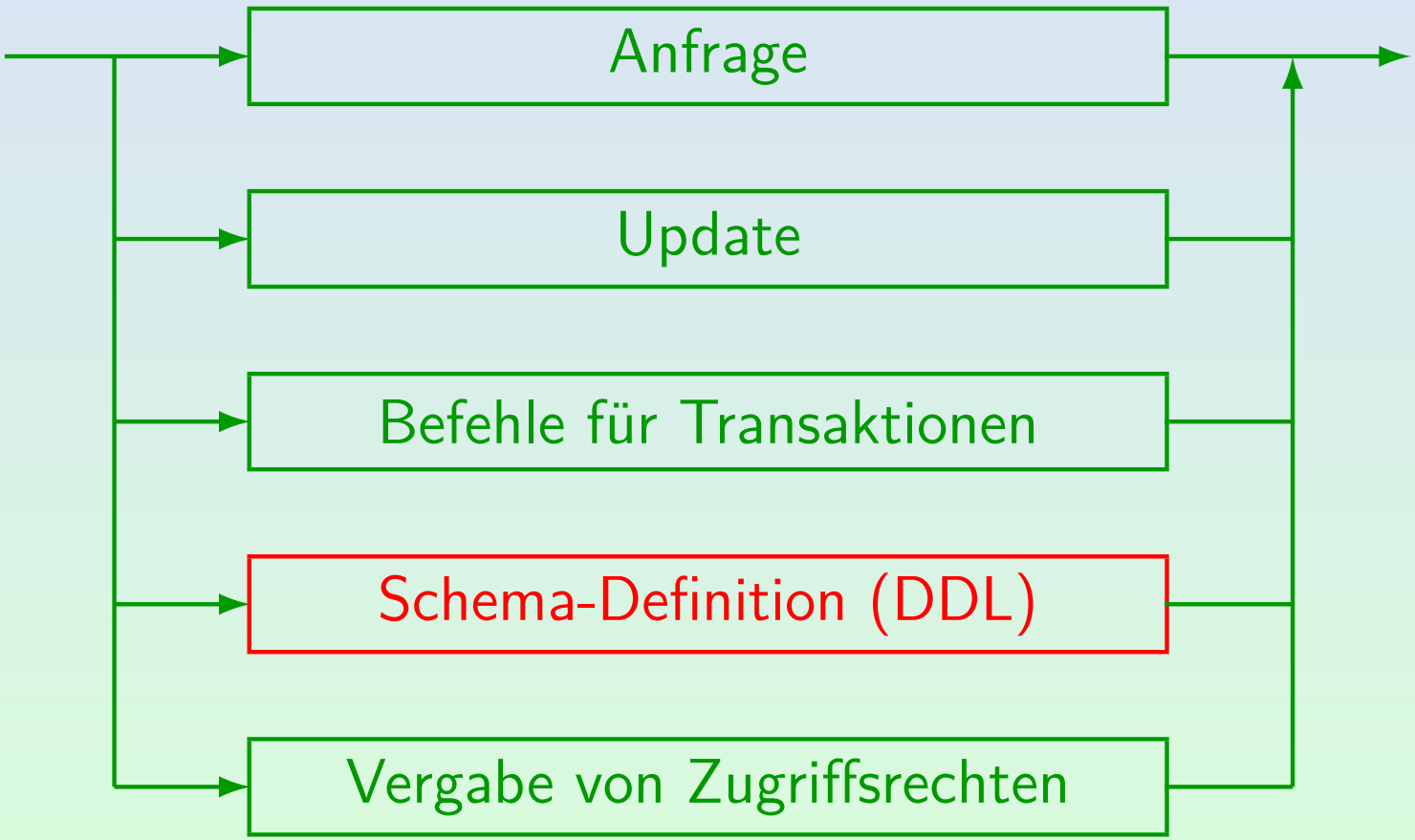
- Letztendlich benötigt man ein CREATE TABLE Statement:

```
CREATE TABLE BEWERTUNGEN(
                SID             NUMERIC(3)    NOT NULL,
                ATYP            CHAR(1)       NOT NULL,
                ANR             NUMERIC(2)    NOT NULL,
                PUNKTE          NUMERIC(4,1) NOT NULL,
                PRIMARY KEY(SID, ATYP, ANR),
                FOREIGN KEY(SID)
                                REFERENCES STUDENTEN,
                FOREIGN KEY(ATYP,ANR)
                                REFERENCES AUFGABEN)
```

- Solche Anweisungen sollen in diesem Kapitel ausführlich besprochen werden.

SQL-Befehle: Übersicht

SQL-Befehl (Auswahl):



Inhalt

Zeichenketten (1)

`CHARACTER(`*n*`)`:

- Zeichenkette fester Länge mit *n* Zeichen.
- Daten, die in einer Spalte mit diesem Datentyp gespeichert werden, werden mit Leerzeichen bis zur Länge *n* aufgefüllt.

  Also wird immer Plattenspeicher für *n* Zeichen benötigt. Variiert die Länge der Daten stark, sollte man `VARCHAR` verwenden, siehe unten.
- `CHARACTER(`*n*`)` kann als `CHAR(`*n*`)` abgekürzt werden.
- Wird keine Länge angegeben, wird 1 angenommen.

  Somit erlaubt „`CHAR`“ (ohne Länge) das Speichern einzelner Zeichen.
- Das maximale *n* ist je nach System begrenzt.
  $n < 255$ sollte sehr portabel sein.

  Der Datentyp `CHAR(`*n*`)` war bereits im SQL-86-Standard enthalten.

Zeichenketten (2)

`VARCHAR(n)`:

- Zeichenkette variabler Länge mit bis zu n Zeichen.

  Es wird nur Speicherplatz für die tatsächliche Länge der Zeichenkette benötigt. Die maximale Länge n dient als Beschränkung, beeinflusst aber normalerweise das Dateiformat auf der Festplatte nicht. Nach dem SQL-Standard (und in PostgreSQL) ist n die Anzahl Zeichen, bei MS SQL Server ist es die Anzahl Bytes, bei Oracle kann man wählen (meist `BYTE`).

- Dieser Datentyp wurde im SQL-92-Standard hinzugefügt (im SQL-86-Standard nicht enthalten).
- Er wird jedoch von allen modernen DBMS unterstützt.
- $n < 255$ sollte absolut portabel sein, $n \leq 4000$ geht meist.
- Der volle Name ist „`CHARACTER VARYING(n)`“.

Zeichenketten (3)

- Jedes moderne System erlaubt auch längere Zeichenketten bis zu ganzen Dateien (als einen Tabelleneintrag).
- Die Details sind aber systemabhängig.
- In MySQL erlaubt der Typ `TEXT` bis zu 65 535 ($2^{16} - 1$) Byte an Daten.

  Ggf. weniger Zeichen, wenn ein Zeichen als mehrere Bytes codiert ist. Für größere Texte gibt es `MEDIUMTEXT` (max. 16 MB) und `LONGTEXT` (max. 4 GB).
- `TEXT` gibt es auch in PostgreSQL (max. 1 GB) und Microsoft SQL Server (max. 2 GB).
- Oracle hat den Typ `CLOB` (max. 128 TB).
- Die Nutzung dieser Werte in Anfragen ist aber meist eingeschränkt.

Zahlen (1)

- `NUMERIC(p,s)`: Vorzeichenbehaftete Zahl mit insgesamt p Ziffern (s Ziffern hinter dem Komma).

  Wird auch Festkommazahl/Fixpunktzahl genannt, da das Komma immer an der gleichen Stelle steht (im Gegensatz zu Gleitkommazahlen).

- Z.B. erlaubt `NUMERIC(3,1)` die Werte `-99.9` bis `99.9`.

  MySQL erlaubt Werte von `-99.9` bis `999.9` (falsch).

- `NUMERIC(p)`: Ganze Zahl mit p Ziffern.

  `NUMERIC(p)` ist das gleiche wie `NUMERIC(p,0)`. „`NUMERIC`“ ohne p verwendet ein implementierungsabhängiges p.

- „`NUMERIC(p[,s])`“ war bereits in SQL-86 enthalten.

  Es wird nicht in Access unterstützt, aber in allen anderen betrachteten DBMS. Oracle verwendet `NUMBER(p,s)` und `NUMBER(p)`, versteht aber auch `NUMERIC`/`DECIMAL` als Synonyme. Keins der anderen Systeme versteht `NUMBER`.

Zahlen (2)

- Die maximale Anzahl Ziffern ist begrenzt.
  $p < 15$ sollte sehr portal sein, $p \leq 28$ ziemlich portabel.

  MySQL erlaubt die Speicherung vieler Ziffern, aber Arithmetik nur mit ca. 15 Ziffern Genauigkeit.

- Der Parameter s muss $s \geq 0$ und $s \leq p$ erfüllen.

  In Oracle muss $-84 \leq s \leq 127$ gelten (egal, wie groß p ist).

- `DECIMAL(`p`,`s`)`: fast das Gleiche wie `NUMERIC(`p`,`s`)`.

  Hier sind größere Wertemengen möglich. Z.B. muss das DBMS bei `NUMERIC(1)` einen Fehler ausgeben, wenn man versucht, 10 einzufügen. Bei `DECIMAL(1)` kann das DBMS evtl. den Wert speichern (wenn sowieso ein ganzes Byte für die Spalte verwendet wird). Übrigens gibt MySQL nie einen Fehler aus, es nimmt einfach den größtmöglichen Wert.

- `DECIMAL` kann mit „`DEC`“ abgekürzt werden.

Zahlen (3)

- `INTEGER`: Vorzeichenbehaftete ganze Zahl, dezimal oder binär gespeichert, Wertebereich ist implementierungsabhängig.

  PostgreSQL, DB2, SQL Server, MySQL und Access verwenden 32 Bit-Binärzahlen: -2 147 483 648 (-2^{31}) .. +2 147 483 647 ($2^{31}-1$).
  D.h. der Wertebereich in diesen DBMS ist etwas größer ist als `NUMERIC(9)`, aber der SQL-Standard garantiert dies nicht. Oracle: Synonym für `NUMBER(38)`.

- `INT`: Abkürzung für `INTEGER`.

- `SMALLINT`: Wie oben, Wertebereich evtl. kleiner.

  PostgreSQL, DB2, SQL Server, MySQL und Access verwenden 16 Bit-Binärzahlen: -32 768 (-2^{15}) .. +32 767 ($2^{15}-1$).
  Somit ist der Wertebereich in diesen Systemen größer als `NUMERIC(4)`, aber kleiner als `NUMERIC(5)`. In Oracle wieder `NUMBER(38)`.

- `BIGINT`: Wie oben, Wertebereich ggf. größer (seit SQL:2003).

Zahlen (4)

INT vs. NUMERIC:

- Wenn die Spezifikation eine bestimmte Anzahl Dezimalziffern verlangt, geht eigentlich nur `NUMERIC`.

  Der Wertebereich von `INT` ist implementierungsabhängig, obwohl 32 Bit sehr typisch ist. Wenn man sich auf die 32-Bit verlassen will, wäre bis 9 Dezimalstellen der Typ `INT` zusammen mit einem `CHECK`-Constraint (s.u.) für das Intervall eine Option. Die Größe des Wertebereichs (nützliche Information für Optimierer) und die benötigte Ausgabebreite wären aber bei `NUMERIC(`p`)` klarer.

- PostgreSQL und einige andere DBMS verwenden bei `INT` die Integer-Division (automatische Abrundung).

  Fragen Sie sich, ob Sie den Wert jemals durch 2 oder durch 10 teilen wollen (mit mathematisch korrektem Ergebnis). Wenn ja, geht nur `NUMERIC`. Wenn eine Spalte `A` den Wert 7 enthält, und vom Typ `INT` ist, liefert PostgreSQL für `A/2` den Wert 3. Bei `NUMERIC(`p`)` käme `3.5` heraus.

Zahlen (5)

INT vs. NUMERIC, Forts.:

- In vielen Systemen verbraucht `INT` weniger Speicherplatz als `NUMERIC`, und Rechnungen mit `INT` gehen schneller.

  Das würde für `INT` sprechen (wenn man nicht dividieren will). Z.B. würden Kundennummern (und andere IDs) ja sicher nicht dividiert.

- In gespeicherten Prozeduren von PostgreSQL erfordern Zuweisungen von `NUMERIC` (auch ohne Nachkommastellen) an `INT` eine explizite Typ-Konvertierung.

  PostgreSQL fasst die Parameter p und s von `NUMERIC` als Integritätsbedingungen auf, intern und in gespeicherten Prozeduren gibt es nur den Typ `NUMERIC` (ohne Parameter). Der PL/pgSQL-Compiler (für Trigger und gespeicherte Prozeduren) versteht nicht, dass eine Spalte vom Typ `NUMERIC(3)` problemlos für einen Parameter vom Typ `INT` übergeben werden kann. Das würde (zumindest bei PostgreSQL) für `INT` sprechen, wenn es möglich ist.

Zahlen (6)

- `FLOAT(`p`)`: Gleitkommazahl $M * 10^E$ mit mindestens p Bits Präzision für M $(-1 < M < +1)$.
- `REAL`, `DOUBLE PRECISION`: Abkürzungen für `FLOAT(`p`)` mit implementierungsabhängigen Werten für p.
- Typisch nur zwei Implementierungen:
  - `FLOAT(`p`)`, $1 \leq p \leq 24$, verwendet 4 Bytes.

    6–7 Ziffern Präzision (Wertebereich `-3.40E+38` bis `3.40E+38`).

    `REAL` bedeutet `FLOAT(24)`. Mit Exponent und Vorzeichen 32 Bit.
  - `FLOAT(`p`)`, $25 \leq p \leq 53$, verwendet 8 Bytes.

    15 Ziffern Präzision (Wertebereich `-1.79E+308` bis `+1.79E+308`).

    `DOUBLE PRECISION` bedeutet `FLOAT(53)`. Insgesamt 64 Bit.
- Rundungsfehler sind nicht wirklich kontrollierbar.
  Man sollte diese Typen nicht für Geldbeträge verwenden!

Datumsangaben

- Datums-, Zeit- und Intervall-Datentypen sind systemabhängig.
- Viele Systeme haben einen Typ `DATE` für Datumsangaben.
  - Oft können die Werte als Strings im Format `'YYYY-MM-DD'` eingegeben werden.
  - SQL-92 verlangt ein Typ-Schlüsselwort: `DATE '2019-03-30'`

    In Oracle enthält `DATE` auch eine Uhrzeit, und das String-Format ist abhängig von den landesspezifischen Einstellungen.
- Es gibt auch die Typen `TIME` für Uhrzeiten und `TIMESTAMP` für eine Kombination aus Datum und Uhrzeit.

  Im SQL Standard kann man Nachkommastellen für die Sekunden festlegen, sowie die Speicherung einer Zeitzone verlangen, z.B. `TIME(3) WITH TIME ZONE`. Bei SQL Server ist `TIMESTAMP` eine eindeutige Nummer, und man muss `DATETIME` verwenden. `TIME` hat dort immer Nachkommastellen.

Datentypen in verschiedenen DBMS

- PostgreSQL:
  [https://www.postgresql.org/docs/9.5/datatype.html]
- MariaDB:
  [https://mariadb.com/kb/en/library/data-types/]
- MySQL:
  [https://dev.mysql.com/doc/refman/8.0/en/data-types.html]
- Oracle:
  [https://docs.oracle.com/cd/B28359_01/server.111/b28318/datatype.htm]
- Microsoft SQL Server:
  [https://docs.microsoft.com/en-us/sql/t-sql/data-types/data-types-transact-sql]
- MySQL, SQL Server, Access (W3Schools):
  [https://www.w3schools.com/sql/sql_datatypes.asp]

Inhalt

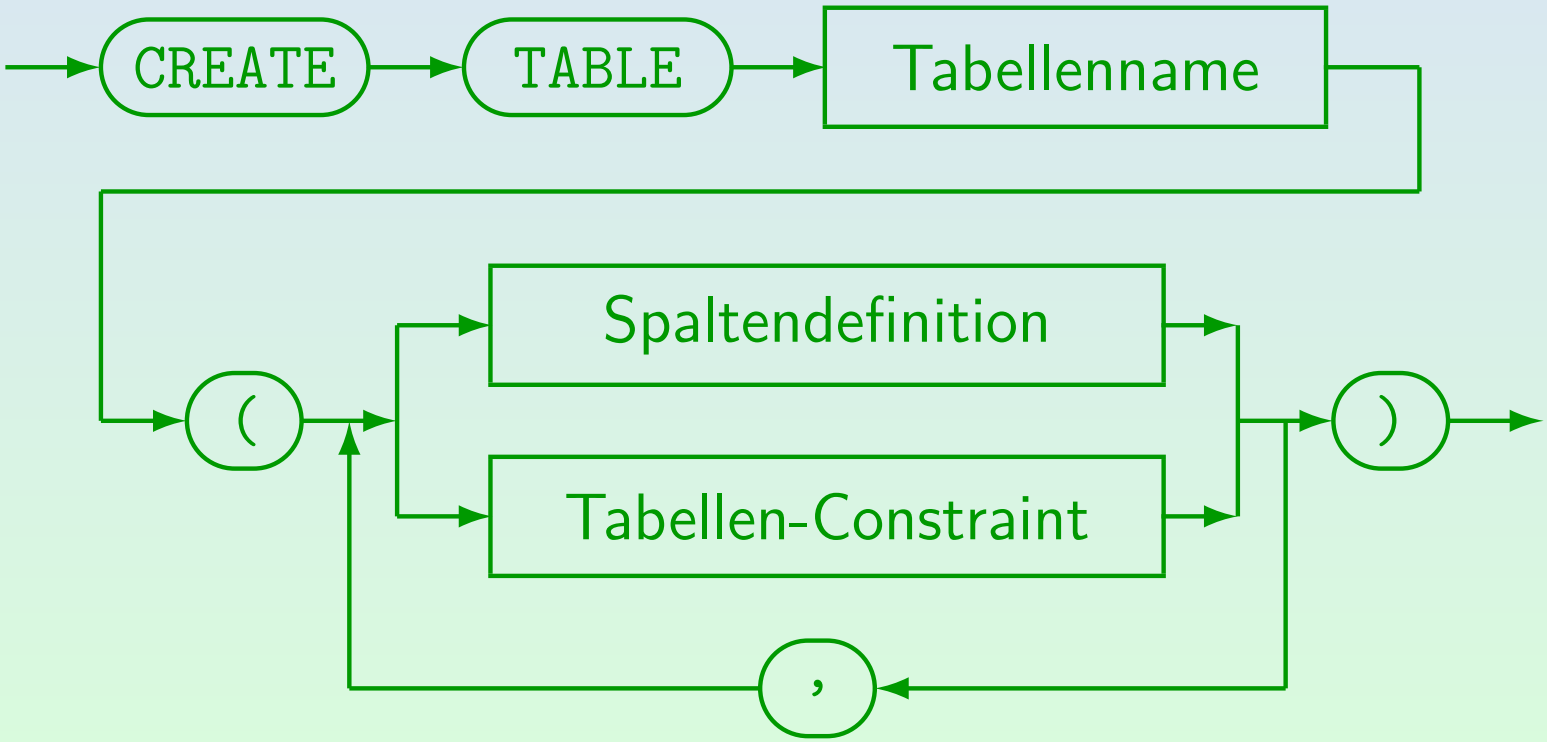
Beispiel

- Das `CREATE TABLE`-Statement in SQL definiert:
  - Tabellenname
  - Spalten und ihre Datentypen
  - Constraints (`NOT NULL`, (Fremd-)Schlüssel, `CHECK`)
- Z.B. `STUDENTEN(SID, VORNAME, NACHNAME, EMAIL°)`:

```
CREATE TABLE STUDENTEN (
       SID      NUMERIC(3)  NOT NULL PRIMARY KEY
                            CHECK(SID > 0),
       VORNAME  VARCHAR(20) NOT NULL,
       NACHNAME VARCHAR(20) NOT NULL,
       EMAIL    VARCHAR(128),
       UNIQUE(VORNAME, NACHNAME) )
```

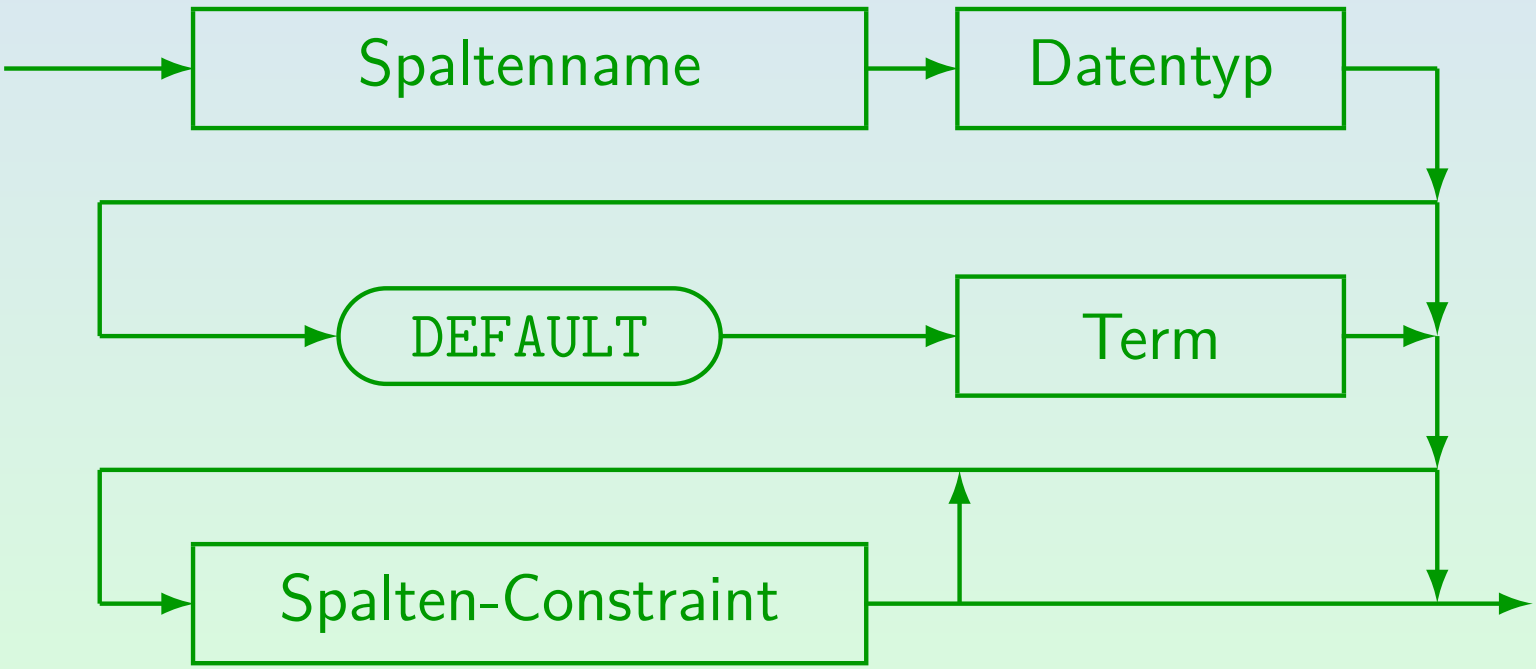
CREATE TABLE: Syntax (1)

CREATE TABLE-Statement:



CREATE TABLE: Syntax (2)

Spaltendefinition:



Constraints: Überblick (1)

- In SQL kann man Constraints als Spalten-Constraints oder Tabellen-Constraints definieren.
- Spalten-Constraints sind Bedingungen, die sich nur auf eine Spalte beziehen.
- Tabellen-Constraints können sich auf mehrere Spalten beziehen.
- Spalten-Constraints (außer vielleicht `NOT NULL`) können auch als „Tabellen-Constraints“ formuliert werden, aber die Syntax von Spalten-Constraints ist etwas einfacher.

Intern werden Spalten-Constraints in Tabellen-Constraints übersetzt.

Constraints: Überblick (2)

- Spalten-Constraints werden in der Spaltendefinition festgelegt (nur durch Leerzeichen getrennt).
- Spalten-Constraints im Beispiel:

```
CREATE TABLE STUDENTEN (
        SID       NUMERIC(3)   NOT NULL   PRIMARY KEY
                                CHECK(SID>0),
        VORNAME   VARCHAR(20)  NOT NULL,
        NACHNAME  VARCHAR(20)  NOT NULL,
        EMAIL     VARCHAR(128),
        UNIQUE(VORNAME, NACHNAME) )
```

Constraints: Überblick (3)

- Tabellen-Constraints werden durch ein Komma von den Spaltendefinitionen und voneinander getrennt:
- Tabellen-Constraint im Beispiel:

```
CREATE TABLE STUDENTEN (
        SID       NUMERIC(3)   NOT NULL PRIMARY KEY
                                CHECK(SID>0),
        VORNAME   VARCHAR(20)  NOT NULL,
        NACHNAME  VARCHAR(20)  NOT NULL,
        EMAIL     VARCHAR(128),
        UNIQUE(VORNAME, NACHNAME) )
```

Constraints: Überblick (4)

- Gleiches Beispiel, aber Spalten-Constraints (außer `NOT NULL`) durch Tabellen-Constraints ersetzt:

```
CREATE TABLE STUDENTEN (
        SID       NUMERIC(3)   NOT NULL,
        VORNAME   VARCHAR(20)  NOT NULL,
        NACHNAME  VARCHAR(20)  NOT NULL,
        EMAIL     VARCHAR(128),
        PRIMARY KEY (SID),
        CHECK(SID > 0),
        UNIQUE(VORNAME, NACHNAME) )
```

Spalten-Constraints (1)

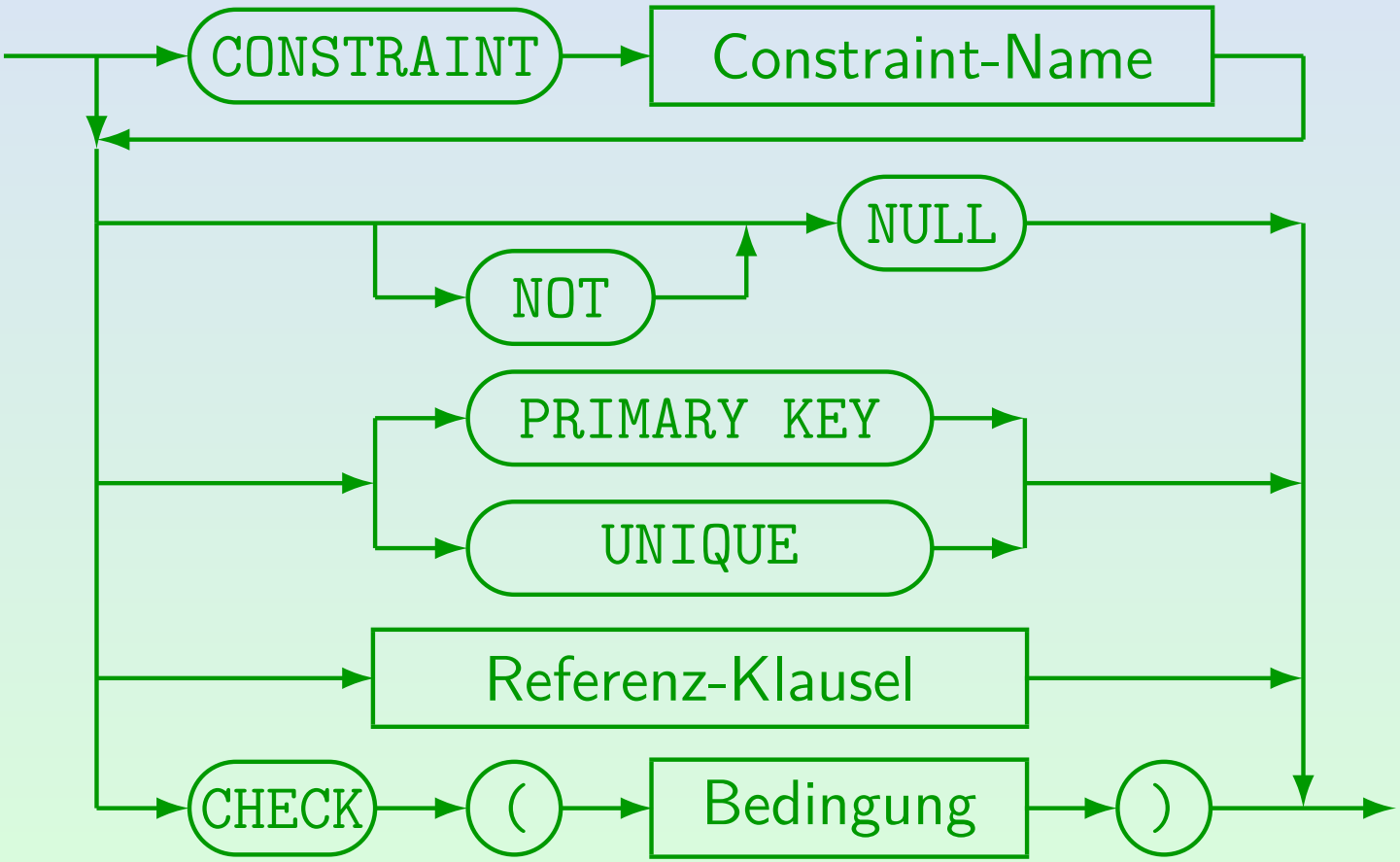
- Es gibt fünf Arten von Spalten-Constraints:
  - `NOT NULL`: Keine Nullwerte in dieser Spalte.
  - `PRIMARY KEY`: Diese Spalte ist der Primärschlüssel der Tabelle.
  - `UNIQUE`: Diese Spalte ist ein Alternativschlüssel.
  - `REFERENCES` T: Die Spalte ist ein Fremdschlüssel.

    D.h. Werte dieser Spalte müssen in der Primärschlüssel-Spalte der Tabelle T auftauchen.
  - `CHECK (`C`)`: Werte der Spalte müssen C erfüllen.

    C ist eine Bedingung wie in der `WHERE`-Klausel, nur mit gewissen Einschränkungen (siehe unten).

Spalten-Constraints (2)

Spalten-Constraint:



Spalten-Constraints (3)

- Man kann „`NULL`“ als Spalten-Constraint schreiben, um zu betonen, dass Nullwerte erlaubt sind.

  Das ist jedoch keine richtige Bedingung und ist sowieso der Default.

- `PRIMARY KEY` impliziert `NOT NULL`.

  DB2 verlangt jedoch, dass `NOT NULL` zusätzlich festgelegt wird.
  Während ich sonst die explizite Angabe implizierter Constraints für falsch halte, ist dies eine Ausnahme: Geben Sie `NOT NULL` auch für Primärschlüssel an (portabler und klarer). Man kann `NOT NULL` auch als Teil des Datentyps sehen, wenn es das formal auch nicht ist.

- Es darf nur einen Primärschlüssel je Tabelle geben.

- `UNIQUE` impliziert `NOT NULL` nicht.

  In DB2 kann `UNIQUE` nur zusammen mit `NOT NULL` verwendet werden.

Spalten-Constraints (4)

- SQL-86 unterstützte nur `[NOT NULL [UNIQUE]]`.
- Außerdem war `UNIQUE` in vielen Systemen nicht implementiert.
- In älterem Code wurde oft „`CREATE UNIQUE INDEX`“ verwendet, um Schlüsselbedingungen zu erzwingen.

  Siehe Kapitel 18 über physischen Entwurf (u.a. Indexe).
- 1989 wurde der Standard um ein optionales „Integrity Enhancement Feature“ erweitert (SQL-89).

  Dies enthielt die obigen Konstrukte.

CHECK-Constraints (1)

- Die Bedingung C eines `CHECK`-Constraints sieht wie eine `WHERE`-Bedingung ohne Unteranfragen aus.

  In Spalten-Constraints kann nur diese eine Spalte verwendet werden. Ansonsten verwendet man Tabellen-Constraints (siehe unten).

- Dabei sind Funktionen wie z.B. `CURRENT_DATE`, die sich später ändern können, ausgeschlossen.

- Grundidee effizienter Constraint-Überprüfung: Bedingung war vor dem Update erfüllt.

- Das DBMS prüft nur geänderte/eingefügte Zeilen.

  Da solche Constraints im leeren DB-Zustand gelten und das System sicherstellt, dass Updates deren Gültigkeit nicht zerstören, folgt per Induktion, dass sie in jedem DB-Zustand gelten. Das erklärt auch, warum z.B. `CURRENT_DATE` verboten ist: Constraints könnten ohne Update ungültig werden.

CHECK-Constraints (2)

- CHECK-Constraints werden von MySQL nicht unterstützt.

  Das CREATE TABLE mit CHECK-Constraint wird akzeptiert, aber Werte, die die CHECK-Bedingung verletzen, werden auch akzeptiert.

- Oracle, PostgreSQL, SQL Server und DB2 schließen Unteranfragen in CHECK-Constraints aus.

  Somit ist eine grundlegende Einschränkung in CHECK-Constraints, dass sie für jedes einzelne Tupel getrennt auswertbar sein müssen.

- SQL-92 erlaubt Unteranfragen in CHECK-Constraints, aber dann ist die Integritätsprüfung schwierig/ineffizient.

  Wird eine in der Unteranfrage verwendete Tabelle geändert, ist es im allgemeinen schwierig herauszufinden, für welche der Zeilen die CHECK-Bedingung erneut überprüft werden muss. Eine einfache Lösung wäre, alle Zeilen zu überprüfen, aber dann führt jede kleine Änderung zu einer langen Integritätsprüfung.

CHECK-Constraints (3)

- Wären Unteranfragen unter `CHECK` implementiert, könnte ein Fremdschlüssel so formuliert werden:

```
CREATE TABLE BEWERTUNGEN(
    SID NUMERIC(3)
        CHECK(SID IN (SELECT SID FROM STUDENTEN)),
    ... )
```

Nicht implementiert!

- Wenn ein `STUDENTEN`-Tupel t gelöscht oder seine `SID` geändert wird, betrifft dies nur Tupel in `BEWERTUNGEN`, die die (alte) SID von t haben.

  Es wäre dann also nicht nötig, die `CHECK`-Bedingung in `BEWERTUNGEN` für alle Tupel zu überprüfen.

Constraints und Nullwerte

- Integritätsbedingungen gelten als erfüllt, wenn das Ergebnis den Wahrheitswert „unbekannt“ hat.

  Die Definitionen für Schlüssel und Fremdschlüssel, die aus mehreren Spalten bestehen, von denen nur einige Null sind, sind kompliziert und systemabhängig.

- Z.B. kann bei dieser Deklaration die E-Mail-Adresse Null sein. Ist sie es nicht, muss sie „@“ enthalten:

```
CREATE TABLE STUDENTEN(
    ...,
    EMAIL VARCHAR(128)
          CHECK(EMAIL LIKE '%@%'))
```

Constraint-Namen (1)

- Einem Constraint kann ein Name gegeben werden, indem man „`CONSTRAINT` ⟨`Name`⟩“ voranstellt.

  MySQL erlaubt Constraint-Namen nur für Tabellen-Constraints. Es scheint sie jedoch sowieso gleich wieder zu vergessen.

- Constraint-Namen müssen innerhalb eines Schemas eindeutig sein, wohingegen Spaltennamen nur eindeutig für eine Tabelle sein müssen.

  In DB2 müssen Constraint-Namen nur für eine Tabelle eindeutig sein. In DB2 können `NOT NULL`-Constraints nicht benannt werden.

- Definieren Sie immer Namen (außer für `NOT NULL`). Sonst wählt das System Namen wie „`SYS_C036`“.

Constraint-Namen (2)

- Wenn ein Constraint verletzt ist, wird der Name ausgegeben: System-generierte Namen ergeben unverständliche Fehlermeldungen.

  Die Fehlermeldung für Not Null-Constraints ist meist klar:

  ```
  ORA-01400: cannot insert NULL into (USER.TABLE.COLUMN)
  ```

  Gibt es mehrere Schlüssel, ist dies schon unklar:

  ```
  ORA-00001: unique constraint (BRASS.SYS_C007916) violated
  ```

  Für einen Fremdschlüssel bekommt man diese Fehlermeldung:

  ```
  ORA-02291: integrity constraint (BRASS.SYS_C007914) violated -
             parent key not found.
  ```

  Für Check-Constraints erhält man diese Nachricht:

  ```
  ORA-02290: check constraint (BRASS.SYS_C007915) violated.
  ```

- Namen erleichtern das Löschen von Constraints.

Constraint-Namen (3)

- Beispiel mit Constraint-Namen:

```
CREATE TABLE STUDENTEN (
       SID       NUMERIC(3) NOT NULL
                 CONSTRAINT SID_MUSS_POSITIV_SEIN
                            CHECK(SID > 0)
                 CONSTRAINT STUDENTEN_SCHLUESSEL
                            PRIMARY KEY,
       VORNAME   VARCHAR(20) NOT NULL,
       NACHNAME  VARCHAR(20) NOT NULL,
       EMAIL     VARCHAR(128),
       CONSTRAINT STUDENTENNAMEN_EINDEUTIG
                    UNIQUE(VORNAME, NACHNAME) )
```

Tabellen-Constraints (1)

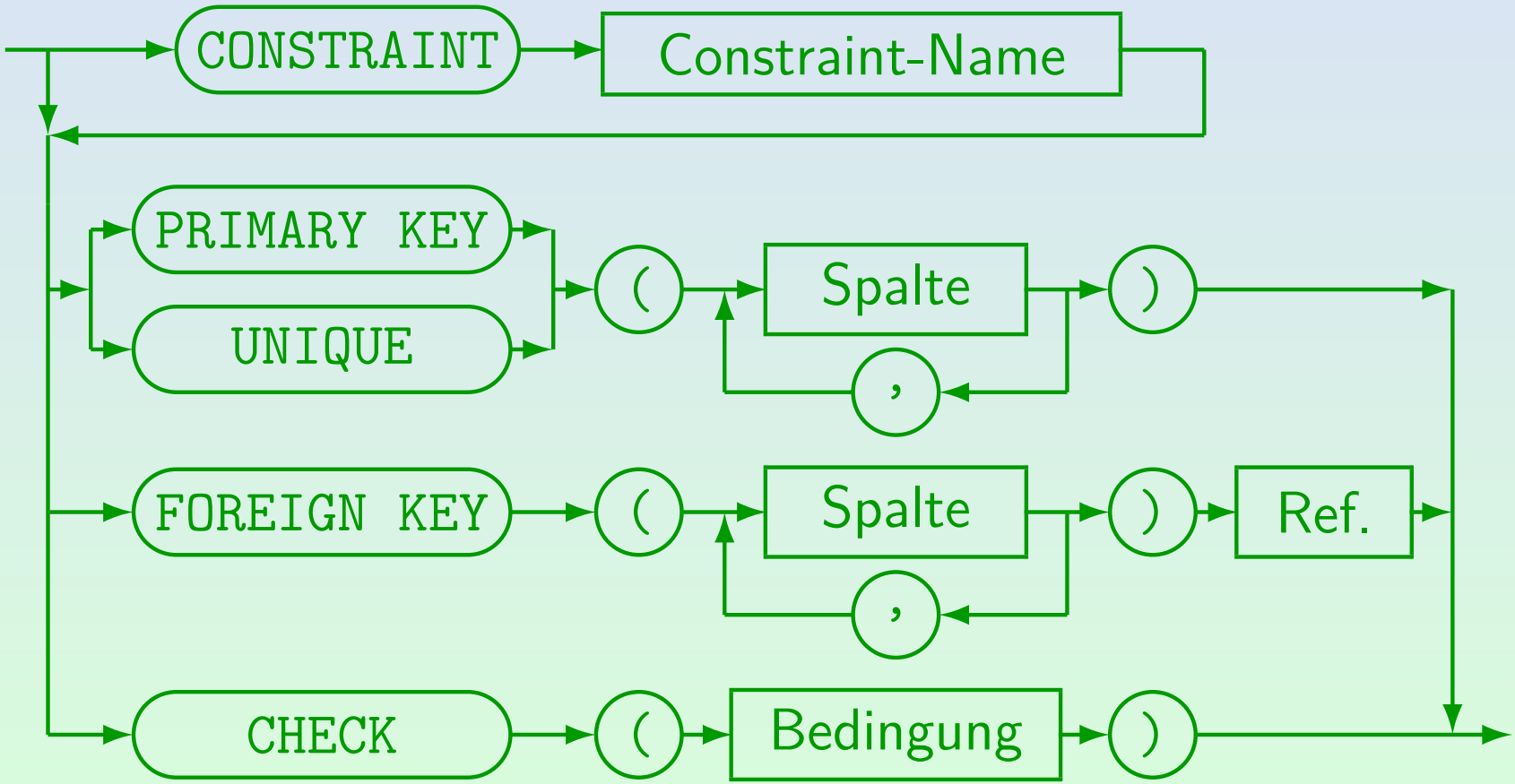
- Tabellen-Constraints werden benötigt, wenn ein (Fremd-)Schlüssel mehrere Spalten hat oder sich eine `CHECK`-Bedingung auf mehrere Spalten bezieht.

  Constraints, die sich nur auf eine Spalte beziehen, können als Spalten- oder Tabellen-Constraint definiert werden.

- Die vier Arten von Tabellen-Constraints sind:
  - `PRIMARY KEY(`A_1`, ..., `A_2`)`
  - `UNIQUE(`A_1`, ..., `A_2`)`
  - `FOREIGN KEY(`A_1`, ..., `A_2`) REFERENCES `R
  - `CHECK(`C`)`

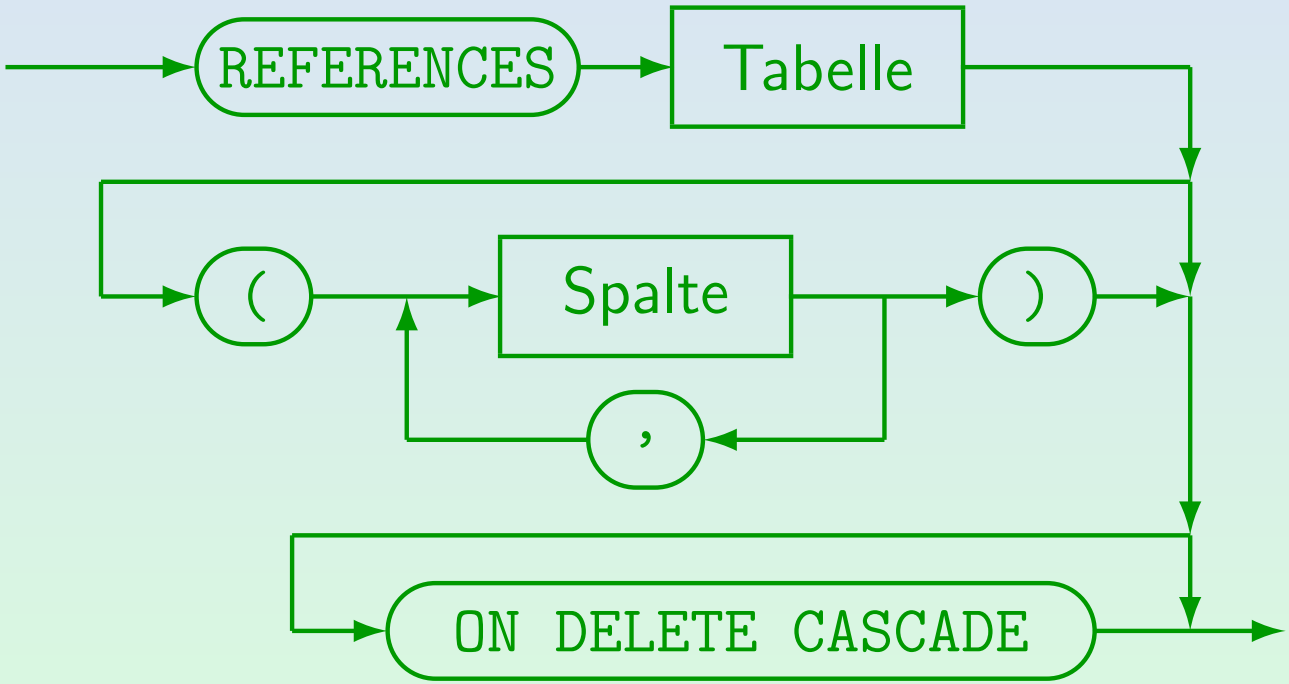
Tabellen-Constraints (2)

Tabellen-Constraint:



Fremdschlüssel (1)

Referenz-Klausel (Ref.):



Fremdschlüssel (2)

- Die Referenz-Klausel steht in Spalten-Constraints hinter der Spalte und in Tabellen-Constraints hinter der `FOREIGN KEY`-Deklaration.
- Es ist möglich, die referenzierten Spaltennamen anzugeben, z.B. `REFERENCES STUDENTEN(SID)`.

  Es können jedoch nur Schlüssel (`PRIMARY KEY` oder `UNIQUE`) referenziert werden.

  Werden keine Spalten genannt, wird der Primärschlüssel referenziert.

  Es macht selten Sinn, Alternativschlüssel zu referenzieren.
- Fremdschlüssel und referenzierter Schlüssel müssen die gleiche Spaltenanzahl und zusammengehörige Spalten den gleichen Datentyp haben.

Fremdschlüssel (3)

- Im Prinzip prüfen MySQL/MariaDB inzwischen Fremdschlüssel, wenn die „Storage Engine“ InnoDB ist.

  Man teste es mit einem unzulässigen Wert. Bei MariaDB 5.5.64 scheint die Kurzform (Spalten-Constraint) ignoriert zu werden. Beim Tabellenconstraint bekommt man den Fehler 1005 (HY000) „Can't create table“, wenn man nach `REFERENCES` nur den Tabellennamen angibt — man muss die Spalten des Primärschlüssels mit angeben.

- Beispiel: `BEWERTUNGEN(SID`$\rightarrow$`STUDENTEN, ...)`
  - In SQL-92 und vielen DBMS (u.a. PostgreSQL, Oracle) kann man verlangen, dass die Bewertungen eines Studenten automatisch mitgelöscht werden, wenn man ihn löscht:

    ```
    REFERENCES STUDENTEN ON DELETE CASCADE
    ```

  - Sonst lehnt das DBMS die Löschung eines Studenten ab, von dem es noch Bewertungen gibt (vgl. `rmdir` vs. `rm -r`).

Fremdschlüssel (4)

- Gegeben sei eine DB mit Vorlesungsdaten: `VORLESUNGEN(..., DOZENT°→PROFESSOREN, ...)`.
- In SQL-92 und vielen DBMS (z.B. PostgreSQL, Oracle) dass `DOZENT` auf Null gesetzt wird, wenn der Professor gelöscht wird. Die Vorlesungen bleiben dann in der DB.
- Dazu schreibt man: „`ON DELETE SET NULL`“.
  Mögliche „Referential Actions“ in SQL-92:
  - `NO ACTION` (der Default: Fehlermeldung)
  - `CASCADE`
  - `SET NULL`
  - `SET DEFAULT`

Fremdschlüssel (5)

- In SQL-99 kam noch `RESTRICT` dazu (ähnlich `NO ACTION`).

  In einigen Systemen wird `NO ACTION` erst am Ende des Statements geprüft, ggf. auch nach Triggern. Möglicherweise sind die Probleme dann beseitigt. `RESTRICT` ist schärfer und führt zu sofortigem Abbruch.

- Seit SQL-92 kann man auch die Reaktion auf Änderungen des Schlüssels von `PROFESSOREN` festlegen (Namensänderung).

- Dazu schreibt man „`ON UPDATE ...`“, z.B.:

```
CREATE TABLE VORLESUNGEN(
        ...
        DOZENT VARCHAR(25) REFERENCES PROFESSOREN
                           ON DELETE CASCADE
                           ON UPDATE CASCADE)
```

`ON UPDATE` wird seltener unterstützt, z.B. nicht in Oracle. PostgreSQL hat es.

Default-Spaltenwerte (1)

- Im Befehl zum Erstellen neuer Zeilen (`INSERT`) muss man nicht für alle Spalten Werte definieren.

  `INSERT` wird in Kapitel 3 erläutert. Wird eine Zeile über eine Sicht eingefügt, kann es sein, dass man gar nicht für alle Spalten Werte angeben kann.

- Fehlende Spalten werden normalerweise auf `NULL` gesetzt.
- Es ist jedoch möglich, einen Default-Wert festzulegen, der in Spalten, für die kein Wert gegeben ist, gespeichert wird.
- Z.B. sollen alle Aufgaben `MAXPT` = 10 haben, wenn der `INSERT`-Befehl keinen Wert für `MAXPT` enthält:

```
CREATE TABLE AUFGABEN(
       ...
       MAXPT NUMERIC(2) DEFAULT 10 NOT NULL
                        CHECK(MAXPT >= 0))
```

Default-Spaltenwerte (2)

- Manchmal können mehrere Nutzer neue Zeilen in eine Tabelle einfügen. Mit „`DEFAULT USER`“ wird der Name des aktuellen Nutzers in einer Spalte gespeichert.

  Z.B. in einer Spalte „`EINGEGEBEN_VON`“.

- Auf diese Weise kennt man den Nutzer, der für das Einfügen einer bestimmten Zeile verantwortlich ist.

  „`USER`“ war schon im SQL-86-Standard enthalten und müsste sehr portabel sein.

  MySQL: `CURRENT_USER`. In PostgreSQL geht beides, in Oracle nur `USER`.

- „`DEFAULT CURRENT_TIMESTAMP`“ speichert aktuelles Datum und Uhrzeit in der Spalte (Datum der Zeilenerstellung).

  So in SQL-92 und z.B. PostgreSQL, SQL Server, inzwischen auch Oracle.

  In Oracle klassisch „`SYSDATE`“ (vom Typ `DATE`, intern mit Uhrzeit!).

  In DB2: „`CURRENT TIMESTAMP`“. In MySQL nur eine Spalte vom Typ `TIMESTAMP` pro Tabelle möglich, automatisch mit aktuellem Zeitstempel als Default.

Default-Spaltenwerte (3)

- `INSERT`-Rechte können selektiv für bestimmte Spalten an Datenbank-Benutzer vergeben werden.

  Zugriffsrechte sind Gegenstand von Kapitel 6.

- Der Benutzer kann dann für die anderen Spalten bei der Tupel-Einfügung keine Werte angeben.

- Daher kann er/sie mit diesen Rechten den `DEFAULT`-Wert nicht überschreiben (durch explizite Wertangabe im `INSERT`).

  Natürlich muss für die `UPDATE`-Rechte die gleiche Einschränkung gelten.
  Der Benutzername oder das aktuelle Datum werden dann immer so gespeichert, wie es im Default-Wert definiert ist.

Default-Spaltenwerte (4)

- `DEFAULT` war in SQL-86 nicht enthalten.
- In SQL-92 darf der Default-Wert nur sein:
  - eine Konstante bzw. das Schlüsselwort `NULL`, oder
  - eine Funktion ohne Argumente.

    Genauer: `USER`, `CURRENT_USER`, `SESSION_USER`, `SYSTEM_USER`, `CURRENT_DATE`, `CURRENT_TIME[(...)]`, `CURRENT_TIMESTAMP[(...)]`

  Z.B. Oracle und PostgreSQL akzeptieren jeden Term ohne Spaltennamen.
- „`DEFAULT NULL`“ wird verwendet, wenn kein Default-Wert explizit definiert ist.

  Somit ist es unmöglich, eine Zeile ohne festgelegten Wert für eine Spalte, die `NOT NULL` ist und keinen definierten Default-Wert hat, einzufügen.
  In MySQL wird in diesem Fall einfach der leere String bzw. `0` eingefügt (macht Fehlererkennung schwieriger).

Temporäre Tabellen

- SQL-92 und einige DBMS ermöglichen die Deklaration temporärer Tabellen, die
  - automatisch am Ende der Transaktion oder Sitzung gelöscht werden,

    Je nach System kann es sein, dass die Tabelle selbst nicht gelöscht wird, sondern nur all ihre Zeilen.
  - unsichtbar für andere (parallele) Sitzungen sind.

    Tabellen anderer Benutzer sind normalerweise unsichtbar (außer der Nutzer hat Zugriffsrechte erteilt). Bei temporären Tabellen haben aber auch parallele Sitzungen des gleichen Nutzers verschiedene Kopien.
- Für Details siehe Handbuch des jeweiligen DBMS.

  [https://www.postgresql.org/docs/12/sql-createtable.html]
  [https://docs.oracle.com/cd/B28359_01/server.111/b28286/statements_7002.htm]

Speicherparameter

- In den meisten DBMS hat das „`CREATE TABLE`“-Statement viele Speicherparameter, die verändert werden können.
- Diese Parameter gehören zum physischen/internen Schema, nicht zum konzeptuellen Schema.

  Es ist etwas unglücklich, dass hier beide Dinge vermischt werden.
- Der SQL-Standard enthält keine Definition, die zu dem physischen Schema gehört.

  Diese Dinge sind stark systemabhängig.
- Ist die Performance wichtig, sollte man die Bedeutung der Parameter verstehen und diese verändern.

  Es sei auf die Vorlesung „Datenbanken II B: DBMS-Implementierung“ verwiesen.

Einschränkungen

- Die Anzahl der Spalten pro Tabelle ist begrenzt (z.B. 255 in ACCESS, 500 in DB2, 1000 in Oracle).
- Die Gesamtlänge von Tabellenzeilen ist begrenzt, dabei zählen aber Spalten von einem „Large Object“-Typ nicht mit (z.B. 2000 Bytes in Access, 8060 Bytes in SQL Server).

  Typischerweise können sich Zeilen nicht über mehrere Datenbank-Blöcke erstrecken, damit ist ihre Länge durch die Blockgröße begrenzt. Die ist aber häufig konfigurierbar (z.B. 8 KB bis 64 KB). Oracle kann Zeilen über mehrere Blöcke aufspalten, allerdings leidet dann die Performance.
- Die Anzahl der Spalten pro Schlüssel ist begrenzt (z.B. 10 in ACCESS, 16 in DB2/SQL Server, 32 in Oracle).

  Die Gesamtlänge eines Schlüssels kann auch begrenzt sein, z.B. 255 Bytes in einer alten DB2 Version. Bei Oracle sind sie auf 40% der Blockgröße begrenzt.

Inhalt

Eindeutige Zahlen (1)

- Oft muss man eindeutige Zahlen generieren, z.B. die `SID` für Studenten (künstlicher Primärschlüssel).
- SQL-92 hatte keinen Mechanismus dafür.
- Theoretisch könnte man das aktuelle Maximum in der Tabelle abfragen, und eins dazu addieren.

  Alternativ könnte man auch eine Tabelle mit einer Zeile und Spalte erstellen, die das Maximum enthält. Das ist evtl. etwas schneller, obwohl man das Maximum auch mit einem B-Baum-Index schnell findet.
- Aber bei gleichzeitigen Nutzern wird dies schwierig, siehe Kapitel 4. Daher haben viele DBMS einen Mechanismus zum Generieren von eindeutigen Zahlen.

Eindeutige Zahlen (2)

SQL-99, PostgreSQL (ab Ver. 10), DB2:

- In SQL-99 wurden Sequenz-Generatoren eingeführt:

```
SID NUMERIC(3)
    GENERATED ALWAYS AS IDENTITY(START WITH 101)
    NOT NULL PRIMARY KEY
```

- „`GENERATED ALWAYS`“ heißt, dass man keinen Wert für diese Spalte im `INSERT`-Befehl festlegen kann.

  Die Alternative ist `GENERATED BY DEFAULT`. Beide Fälle schließen ein `DEFAULT`-Statement aus. Es gibt weitere Parameter für den Generator eindeutiger Zahlen, z.B. `IDENTITY(START WITH 101, INCREMENT BY 1)`. Es gibt auch `MINVALUE` *n*, `MAXVALUE` *n* und `CYCLE` bzw. `NO CYCLE`.

- Es sind auch berechnete Spalten möglich mit `GENERATED ALWAYS AS (`⟨`Term`⟩`)`.

Eindeutige Zahlen (3)

Oracle/PostgreSQL (seit Ver. 7.1 oder früher):

- Diese DBMS haben ein DB-Objekt „sequence“ (Sequenz):

  ```
  CREATE SEQUENCE STUD_IDS START WITH 101
  ```

  Ähnliche Sequenzen wurden ebenfalls in SQL-99 eingeführt.

- Man kann folgendermaßen eine Zahl abrufen und den in der Sequenz gespeicherten Wert erhöhen:
  - `SELECT NEXTVAL('STUD_IDS')` (PostgreSQL)

    Man kann `NEXTVAL('STUD_IDS')` als Default-Wert für eine Spalte vom Typ `INT` festlegen. Das tut der PostgreSQL-Typ `SERIAL` intern.
  - `SELECT STUD_IDS.NEXTVAL FROM DUAL` (Oracle)

    `DUAL` ist eine Dummy-Tabelle mit einer Zeile. In Oracle kann man `STUD_IDS.NEXTVAL` nicht als `DEFAULT` festlegen, aber man kann es im `INSERT`-Befehl verwenden. SQL-99: `NEXT VALUE FOR` ⟨Sequenz⟩.

Eindeutige Zahlen (4)

SQL Server:

- In SQL Server kann man das Wort `IDENTITY` zur Deklaration einer Integer-Spalte hinzufügen:

```
SID NUMERIC(3) IDENTITY NOT NULL PRIMARY KEY
```

- Werte dieser Spalte kann man nicht festlegen, der nächste Wert wird automatisch eingefügt.

  Sogar wenn man keine Spalten-Liste unter `INSERT` verwendet, wird die Spalte in der `VALUES`-Liste übersprungen. Besser: Spalten festlegen.

  `IDENTITY` kann nicht zusammen mit `DEFAULT` verwendet werden.

- Man kann Startwert und Schrittweite festlegen:

```
SID NUMERIC(3) IDENTITY(100,1) PRIMARY KEY
```

Eindeutige Zahlen (5)

MySQL:

- In MySQL kann man das Wort `AUTO_INCREMENT` zur Deklaration einer Integer-Spalte hinzufügen:

```
SID INT AUTO_INCREMENT PRIMARY KEY
```

- Dies funktioniert nur mit binären Integer-Typen.

  Z.B. funktioniert es nicht mit `NUMERIC(3)`. `UNSIGNED`-Typen sind möglich.

  Die Spalte muss als Schlüssel deklariert sein (`PRIMARY KEY` oder `UNIQUE`).

- Wird mit `INSERT` für diese Spalte `NULL` oder `0` festgelegt, wird stattdessen die nächste Zahl eingefügt.

  Ein deklarierter Default-Wert wird einfach ignoriert.

PostgreSQL:

- PostgreSQL auch auch Typen `SERIAL` und `BIGSERIAL`.

  Abkürzung für `INT` bzw. `BIGINT` mit zugehöriger Sequenz.

Inhalt

CREATE SCHEMA (1)

- In Oracle und SQL Server entsprechen sich DB-Accounts und Schemas 1:1.

  Benötigt ein Nutzer mehr als ein Schema, müssen mehrere Accounts für die gleiche Person erstellt werden. Tabellen werden im System global durch ihren Namen und ihren Eigentümer identifiziert. In SQL Server: Server, Datenbank, Eigentümer, Name.

- In PostgreSQL sind Nutzer und Schemata getrennte Konzepte. Tabellen können als `Schema.Tab` angesprochen werden.
  - Beliebig viele Nutzer können Lese- und/oder Schreib-Zugriff auf ein Schema haben.
  - Ein Nutzer kann auf beliebig viele Schemata Zugriff haben.

    Es gibt einen Suchpfad für Schemata, der bestimmt, welches Schema benutzt wird, wenn nicht explizit eins angegeben ist.

CREATE SCHEMA (2)

- Es gibt einen `CREATE SCHEMA`-Befehl:

```
CREATE SCHEMA AUTHORIZATION ⟨Account⟩
    CREATE TABLE ABT(ABTNR NUMERIC(2) PRIMARY KEY,
        CHEF NUMERIC(4) REFERENCES ANG, ...)
    CREATE TABLE ANG(ANGNR NUMERIC(4) PRIMARY KEY,
        ABTNR NUMERIC(2) REFERENCES ABT, ...);
```

- Insbesondere ist es damit möglich, Tabellen anzulegen, die sich gegenseitig referenzieren (nicht in PostgreSQL).
- Schlägt ein Statement fehl, wird nichts ausgeführt.
- Man beachte, dass die einzelnen `CREATE TABLE`-Statements nicht durch „;“ getrennt werden.

[https://www.postgresql.org/docs/12/sql-createschema.html]

CREATE SCHEMA (3)

- MySQL, Access unterstützen `CREATE SCHEMA` nicht.
- In Oracle, DB2, SQL Server kann `CREATE SCHEMA` die Befehle `CREATE TABLE`, `CREATE VIEW`, `GRANT` enthalten.

  In DB2 sind auch `COMMENT ON` und `CREATE INDEX` erlaubt.
- Da PostgreSQL und DB2 mehrere Schemata pro Nutzer zulassen, lautet die Syntax dort

  `CREATE SCHEMA` ⟨Name⟩ `AUTHORIZATION` ⟨Nutzer⟩

  Der Schema-Name und die Autorisierungs-Klausel sind beide optional, aber mindestens eines der beiden wird benötigt.
- Bei diesen Systemen kann man das Schema zunächst auch ohne Tabellen anlegen, und die Tabellen später im Schema anlegen: `CREATE TABLE` ⟨Schema⟩.⟨Tab⟩ ...

Tabellen löschen (1)

- Tabellen werden mit folgendem Befehl gelöscht:

  `DROP TABLE ⟨Tabellenname⟩`

- Natürlich löscht dies auch alle Zeilen in der Tabelle.

  Man muss vorsichtig sein! In Oracle wird die aktuelle Transaktion automatisch beendet, wenn eine Tabelle gelöscht wird. Somit ist es nicht möglich, das Löschen oder vorangehende Aktionen rückgängig zu machen.

- Was passiert, wenn die Tabelle in Fremdschlüssel-Bedingungen referenziert wird?

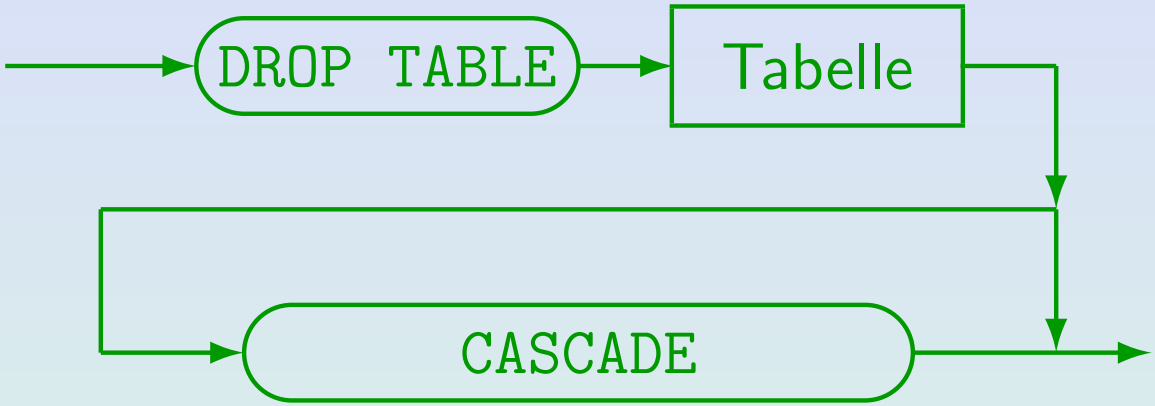
  In SQL Server muss die referenzierende Tabelle zuerst gelöscht werden.

  In DB2 wird der Fremdschlüssel automatisch gelöscht.

  In Oracle kann man mit „`CASCADE CONSTRAINTS`“ Fremdschlüssel mit löschen.

- In SQL-92 und PostgreSQL kann man „`CASCADE`“ hinzufügen, um die Fremdschlüssel mit zu löschen.

Tabellen löschen (2)



- In Oracle muss man „`CASCADE CONSTRAINTS`“ schreiben.

Beispiele:

- `DROP TABLE STUDENTEN CASCADE`
- `DROP TABLE BEWERTUNGEN`

  Löscht man zuerst die Tabelle `BEWERTUNGEN` (enthält Fremdschlüssel) und dann `STUDENTEN`, so ist kein `CASCADE CONSTRAINTS` notwendig.

Inhalt

ALTER TABLE (1)

- Mit dem `ALTER TABLE`-Befehl kann man das Schema einer existierenden Tabelle verändern.
- Im Prinzip könnte man die Daten temporär woanders speichern, die Tabelle löschen, sie verändert neu erstellen und die Daten zurückkopieren. Aber
  - Kopieren ist unpraktisch bei großen Tabellen.
  - Tabelleneinträge könnten von Fremdschlüsseln referenziert werden.

    Dann muss man evtl. die gesamte DB neu erstellen. Auch Indexe, Grants, Sichten, Trigger etc. referenzieren Tabellen. Manches davon wird verlorengehen, wenn die Tabelle gelöscht wird.

ALTER TABLE (2)

- Beispiele für Änderungen eines Tabellen-Schemas:
  - Neue Spalten können hinzugefügt werden.

    Daher ist es sicherer, in Anwendungsprogrammen statt `SELECT *` Spaltennamen anzugeben.
  - Die Breite von Spalten kann erhöht werden.

    Z.B. von `VARCHAR(20)` zu `VARCHAR(30)`.
    Das ist eigentlich im SQL-92-Standard nicht möglich, aber in allen fünf DBMS (Oracle, DB2, SQL Server, Access, MySQL).
  - Constraints kann man hinzufügen/entfernen.

    In manchen Systemen kann man eine Bedingung (Constraint) auch deaktivieren, so dass sie nicht mehr überprüft wird, aber noch im System gespeichert ist. Später kann man sie wieder aktivieren.

ALTER TABLE (3)

- `ALTER TABLE` war nicht in SQL-86 enthalten.
- Es ist im SQL-92-Standard enthalten, aber DBMS-Implementierungen unterscheiden sich stark in der Syntax und darin, was geändert werden kann.
- Der SQL-92-Standard hat folgende Möglichkeiten:
  - Spalten können hinzugefügt oder entfernt werden.
  - Der Default-Wert einer Spalte kann geändert werden, aber der Datentyp nicht.
  - Constraints kann man hinzufügen/entfernen.

ALTER TABLE (4)

SQL-92 (Spalten hinzufügen):

- Z.B. Spalte „ZUSATZPKT“ zu „STUDENTEN“ hinzufügen:

```
ALTER TABLE STUDENTEN
      ADD COLUMN ZUSATZPKT NUMERIC(4,1)
                           CHECK(ZUSATZPKT >= 0)
```

- Das Schlüsselwort „COLUMN“ ist optional.
- Die neue Spalte hat zunächst in allen Zeilen einen Nullwert.
- Der Wert kann anschließend verändert werden.

  Es kann jedoch zu Effizienzproblemen kommen, wenn die Zeilen viel länger werden, als sie beim Einfügen waren.

ALTER TABLE (5)

SQL-92 (Spalten hinzufügen, fortgesetzt):

- Wird ein Default-Wert festgelegt, kann der neue Wert `NOT NULL` sein:

```
ALTER TABLE STUDENTEN
  ADD ZUSATZPKT NUMERIC(4,1) DEFAULT 0 NOT NULL
```

- Die neue Spalte wird die letzte (rechts) in der Tabelle.

  Es ist nicht möglich, sie woanders einzufügen.

- Sichten (gespeicherte Anfragen) werden nicht beeinflusst, weil `SELECT *` durch eine explizite Spaltenliste ersetzt wird, wenn die Sicht gespeichert wird.

ALTER TABLE (6)

SQL-92 (Spalten entfernen):

- Spalten können aus Tabellen entfernt werden:

```
ALTER TABLE STUDENTEN
      DROP COLUMN ZUSATZPKT RESTRICT
```

- `RESTRICT` heißt, dass `ALTER TABLE` fehlschlägt, wenn die Spalte in Constraints/Sichten referenziert wird.

  Constraints, die sich nur auf diese Spalte beziehen, zählen nicht: Sie werden automatisch gelöscht.

- Die Alternative ist `CASCADE`: Das referenzierende DB-Objekt wird mit gelöscht.

  Es gibt keinen Default: Man muss `RESTRICT` oder `CASCADE` angeben.

ALTER TABLE (7)

SQL-92 (Spalten verändern):

- Die einzige erlaubte Veränderung einer Spalte ist die Änderung des Default-Wertes:

```
ALTER TABLE AUFGABEN
ALTER COLUMN MAXPT SET DEFAULT 12
```

- Der Default kann mit dieser Syntax auch auf Null gesetzt werden:

```
ALTER TABLE AUFGABEN
ALTER COLUMN MAXPT DROP DEFAULT
```

- In SQL-92 ist es nicht möglich, den Datentyp einer Spalte zu verändern.

ALTER TABLE (8)

SQL-92 (Constraints verändern):

- Eine Bedingung hinzufügen:

```
ALTER TABLE STUDENTEN
      ADD CONSTRAINT ZUSATZPKT_DEF
      CHECK(ZUSATZPKT IS NOT NULL)
```

Es können nur Tabellen-Constraints hinzugefügt werden, aber Spalten-Constraints sind sowieso nur syntaktische Abkürzungen.

- Einen benannten Constraint entfernen:

```
ALTER TABLE STUDENTEN
      DROP CONSTRAINT ZUSATZPKT_DEF RESTRICT
```

Date/Darwen behaupten, dass der Standard verlangt, `RESTRICT` oder `CASCADE` festzulegen, obwohl das nur bei Schlüsseln Sinn macht, die in Fremdschlüsseln referenziert werden.

Literatur/Quellen

- Elmasri/Navathe:Fundamentals of Database Systems, 3. Auflage, 1999.
- Kemper/Eickler: Datenbanksysteme, 4. Auflage, Oldenbourg, 1997.
- Lipeck: Skript zur Vorlesung Datenbanksysteme, Univ. Hannover, 1996.
- Date/Darwen: A Guide to the SQL Standard, 4. Auflage, Addison-Wesley, 1997.
- van der Lans: SQL, Der ISO-Standard, Hanser, 1990.
- Melton/Simon: Understanding the New SQL. Morgan Kaufman, 1993.
- Oracle 8i SQL Reference, Release 2 (8.1.6), Dez. 1999, Part No. A76989-01.
- Oracle 8i Concepts, Release 2 (8.1.6), Dez. 1999, Part No. 76965-01.
  Kapitel 12: Built-in Datatypes.
- Chamberlin: A Complete Guide to DB2 Universal Database. Morgan Kaufmann, 1998.
- Microsoft SQL Server Books Online: Accessing and Changing Data.
- Microsoft Jet Database Engine Programmer's Guide, 2. Auflage
  (Teil der MSDN Library Visual Studio 6.0). Microsoft Access 2000 Online-Hilfe.
- DuBois: MySQL. New Riders Publishing, 2000, ISBN 0-7357-0921-1, 756 Seiten.
- MySQL-Handbuch für Version 3.23.53.