Einführung in Datenbanken

Kapitel 4: SQL: Datentypen

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Wintersemester 2018/19

http://www.informatik.uni-halle.de/~brass/db18/

Inhalt

Logik: Motivation (1)

- Der Kern von SQL kann als leichte syntaktische Variante der Prädikatenlogik erster Stufe gesehen werden.

  Vermutlich haben viele Teilnehmer dieser Vorlesung zumindest ein wenig Logik in früheren Vorlesungen gehabt.

- Allgemein geht es in mathematischer Logik wie in Datenbanken darum,
  - Wissen zu formalisieren, und
  - mit diesem Wissen zu arbeiten.
- Ein Zugang zu SQL kann es sein, es als Spezialfall bzw. praktische Anwendung der Logik zu erklären.

  Mir ist bewusst, das verschiedene Teilnehmer der Vorlesung verschiedene Zugänge benötigen. Z.B. kann man eine Sprache auch über viele Beispiele und die Syntaxdiagramme lernen.

Logik: Motivation (2)

- Vorteile der Logik:
  - Die Konzepte der Logik sind präzise mathematisch definiert.
  - In SQL gibt es häufig viele syntaktische Varianten, das Gleiche auszudrücken.
    Logik erlaubt die Konzentration auf das Wesentliche.
  - Logik kann ein allgemeiner Rahmen sein, um auch andere Datenmodelle und Datenbank-Sprachen zu verstehen.
  - Wenn zwei `WHERE`-Bedingungen logisch äquivalent sind, spielt es keine große Rolle, welche man wählt.
  - Begriffe aus der Logik wie Inkonsistenz (“immer falsch”) sind nützlich, um Fehler in SQL-Anfragen zu verstehen.
- In der Klausur wird nur SQL verlangt, keine logischen Formeln.

Alphabet (1)

Definition:

- Sei *ALPH* eine unendliche, aber abzählbare Menge von Elementen, die Symbole genannt werden.

  Die Elemente von *ALPH* sind die Wortsymbole der lexikalischen Syntax, z.B. ist jeder Bezeichner oder jede Zahlkonstante ein Element von *ALPH*.

- *ALPH* muss zumindest die logischen Symbole enthalten, d.h. $LOG \subseteq ALPH$, wobei
  $LOG = \{(,), ,, \top, \bot, =, \neg, \wedge, \vee, \leftarrow, \rightarrow, \leftrightarrow, \forall, \exists, :\}$.

- Zusätzlich muss *ALPH* eine unendliche Teilmenge $VARS \subseteq ALPH$ enthalten, die Menge der Variablen. Diese ist disjunkt zu *LOG* (d.h. $VARS \cap LOG = \emptyset$).

  Die Menge der Variablen muss unendlich sein, damit es immer noch eine weitere Variable gibt, zusätzlich zu denen, die in einer gegebenen Formel vorkommen.

Alphabet (2)

- Z.B. kann das Alphabet bestehen aus
  - den speziellen logischen Symbolen *LOG*,
  - Bezeichnern: Folgen von Buchstaben, Ziffern, "_".

    In der Praxis sind Variablen oft Bezeichner. In Logik-Lehrbüchern wird üblicherweise angenommen, dass die Variablen disjunkt zu den Bezeichnern sind, die für Prädikate und Funktionen verwendet werden. Das vereinfacht die Definitionen, aber die Beispiele sehen dann komisch aus, wenn z.B. alle Nicht-Variablen klein geschrieben werden müssen, oder man nur x, y, z (mit Index) als Variablen verwenden kann. Damit die Definitionen leicht auf SQL übertragbar sind, müssen auch die Namen von parametrisierte Datentypen wie "`NUMERIC(2)`" als ein Element von *ALPH* gesehen werden.
  - Operator-Symbolen wie `+`, `<`,
  - Datentyp-Literalen (Konstanten) wie `123`, `'abc'`.

Alphabet (3)

- Man beachte, dass Wörter wie "STUDENTEN" als Symbole angesehen werden (Elemente des Alphabets).

  Wenn man ein Alphabet aus 26 Buchstaben gewöhnt ist, erscheint es zunächst komisch, dass das Alphabet hier unendlich ist. Es ist aber bei Programmiersprachen auch üblich, die einzelnen Zeichen zunächst zu Wortsymbolen (Token) zusammenzufassen (durch die lexikalische Analyse). Die eigentliche Grammatik der Programmiersprache beschreibt dann, wie Programme als Folgen solcher Wortsymbole zu bilden sind (gewissermaßen "Sätze"). Nichts anderes geschieht hier.

- In der Theorie sind die exakten Symbole unwichtig.

  Auch die logischen Symbole müssen nicht unbedingt so aussehen wie auf Folie 5 (Beispiele für Alternativen siehe Folie 8).

Alphabet (4)

Mögliche Alternativen für logische Symbole:

Symbol	Alternative	Alt2	Name	SQL
$\top$	`true`	`T`		
$\bot$	`false`	`F`		
$\neg$	`not`	`~`	Negation	NOT
$\wedge$	`and`	`&`	Konjunktion	AND
$\vee$	`or`	`|`	Disjunktion	OR
$\leftarrow$	`if`	`<-`		
$\rightarrow$	`then`	`->`		
$\leftrightarrow$	`iff`	`<->`		
$\exists$	`exists`	`E`	Existenzquantor	s.u.
$\forall$	`forall`	`A`	Allquantor	

In SQL wird der Existenzquantor mit einer Unteranfrage ausgedrückt:

`EXISTS(SELECT * FROM ... WHERE ...)`.

Worte über einem Alphabet

- Für eine Menge A ist A^* die Menge der endlichen Folgen $a_1 \dots a_n$ von Elementen $a_i \in A$.

  Man nennt A^* auch die Menge der Worte über dem Alphabet A.

- ϵ ist die leere Folge (Folge der Länge 0).

  Man nennt ϵ auch das leere Wort.

- Formeln sind Elemente von $ALPH^*$, z.B. ist

  $$\forall\, \texttt{int}\, \texttt{X}\colon\ \neg(\texttt{X} < \texttt{X})$$

  eine Formel bestehend aus den Symbolen $\forall$, `int`, `X`, u.s.w.

  Es wird hier eine mehrsortige Logik verwendet, also eine Logik mit Datentypen. Deswegen ist hier angegeben, dass die Variable `X` über den ganzen Zahlen läuft. In SQL laufen Variablen über den Zeilen einer Tabelle also nur einem endlichen Bereich (siehe Kapitel 5).

Signaturen (1)

Definition:

- Eine Signatur $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F})$ enthält:
  - Eine nichtleere Menge $\mathcal{S}$. Die Elemente heißen Sorten (Datentypnamen).
  - Für jedes $\alpha = s_1 \dots s_n \in \mathcal{S}^*$, eine endliche Menge $\mathcal{P}_\alpha \subseteq \mathit{ALPH} \setminus \mathit{LOG}$ (Prädikatssymbole).
  - Für jedes $\alpha \in \mathcal{S}^*$ und $s \in \mathcal{S}$, eine Menge $\mathcal{F}_{\alpha,s} \subseteq \mathit{ALPH} \setminus \mathit{LOG}$ (Funktionssymbole).

    Für jedes $\alpha \in \mathcal{S}^*$ und $s_1, s_2 \in \mathcal{S}$, $s_1 \neq s_2$ muss $\mathcal{F}_{\alpha,s_1} \cap \mathcal{F}_{\alpha,s_2} = \emptyset$ gelten (Beschränkung des Überladens von Funktionssymbolen: Name und Argumentsorten bestimmen Ergebnissorte).
    Außerdem muss $\mathit{VARS} \setminus \cup_{s \in \mathcal{S}} \mathcal{F}_{\epsilon,s}$ noch unendlich sein (ausreichend Variablen nach Auflösung von Namenskonflikten mit Konstanten).

Signaturen (2)

- Sorten sind Datentyp-Namen, z.B. `int`, `NUMERIC(2)`.

  Später auch für Zeilen von Tabellen (Tupel-Typen), z.B. `STUDENTEN`.

- Prädikate liefern für gegebene Eingabewerte wahr oder falsch, z.B. ist $<$ ein Prädikat.

  In Kapitel 2 gab es auch ein Beispiel mit dem Prädikat `LIKE` für den Mustervergleich in SQL.

- Ist $p \in \mathcal{P}_\alpha$, und $\alpha = s_1 \dots s_n$, dann werden $s_1, \dots, s_n$ die Argumentsorten von p genannt.

  s_1 ist der Typ des ersten Arguments, s_2 der des zweiten, usw.

- Die Anzahl der Argumentsorten (Länge von α) nennt man Stelligkeit des Prädikatsymbols, z.B. ist $<$ ein Prädikatsymbol der Stelligkeit 2.

Signaturen (3)

- Das gleiche Symbol p kann Element verschiedener $\mathcal{P}_\alpha$ sein (überladenes Prädikat), z.B.
  - $< \in \mathcal{P}_{\texttt{int}\,\texttt{int}}$
  - $< \in \mathcal{P}_{\texttt{string}\,\texttt{string}}$
    (lexikographische/alphabetische Ordnung)
- Es kann also mehrere verschiedene Prädikate mit gleichem Namen geben.

  Die Möglichkeit überladener Prädikate ist in der Theorie nicht wichtig. Man kann stattdessen auch verschiedene Namen verwenden, etwa `lt_int` und `lt_string`. Überladene Prädikate komplizieren die Definition und sind in der mathematischen Logik normalerweise ausgeschlossen. Sie erlauben aber natürlichere Formulierungen und sind z.B. auch in Programmiersprachen üblich.

Signaturen (4)

- Eine Funktion liefert für gegebene Eingabewerte einen Ausgabewert. Beispiele: `+`, `-`, `*`, `/`, `SIN`, `SQRT`.

  Es sei hier angenommen, dass Funktionen für alle Eingabewerte definiert sind. Das muss nicht sein, z.B. ist 5/0 ist nicht definiert. Man könnte das mit einer mehrwertigen Logik behandeln (wahr, falsch, Fehler).

- Ein Funktionssymbol in $\mathcal{F}_{\alpha,s}$ hat die Argumentsorten α und die Ergebnissorte s, z.B. $\texttt{+} \in \mathcal{F}_{\texttt{int}\,\texttt{int},\,\texttt{int}}$.

- Informatiker würden eher schreiben:

  `+(int, int): int`.

  Die exakte Notation ist nicht besonders wichtig. Das Konzept eines Funktionssymbols, von Argumentsorten (oder "Datentypen der Argumente") und Ergebnissorte schon. Wenn man (wie hier) das Überladen braucht, bietet sich die mathematische Formalisierung als indizierte Menge an. Für die meisten α, s ist $\mathcal{F}_{\alpha,s}$ allerdings leer.

Signaturen (5)

- Eine Funktion ohne Argumente heißt Konstante.

  Das ist nur ein technischer Trick, um eine weitere Komponente der Signatur zu vermeiden.

- Beispiele für Konstanten:
  - $1 \in \mathcal{F}_{\epsilon,\texttt{int}}$ `1: int`.
  - $\texttt{'Birgit'} \in \mathcal{F}_{\epsilon,\texttt{string}}$ `'Birgit': string`.
- Bei Datentypen (z.B. `int`, `string`) kann üblicherweise jeder mögliche Wert durch eine Konstante bezeichnet werden.

  Im Allgemeinen sind die Menge der Werte und die der Konstanten dagegen verschieden.

Signaturen (6)

- Natürlich kann man eine unendliche Konstantenmenge nicht durch Aufzählung definieren.
- Mathematisch ist das kein Problem, $\mathcal{F}_{\epsilon,\texttt{int}}$ ist eben die Menge der ganzen Zahlen in Dezimalnotation.
- Praktisch ist das auch kein Problem, da die Datentypen bereits in das DBMS eingebaut sind (s.u.): $\mathcal{F}_{\epsilon,\texttt{int}}$ ist durch ein Programm definiert.
- Die letztendlich in SQL verwendete Signatur hat zwei Teile:
  - Die vom DBMS vorgegebene Signatur der Datentypen.
  - Anwendungsspezifische Symbole, die über das DB-Schema eingeführt werden (eine Sorte für die Tupel jeder Tabelle).

Signaturen (7)

Definition:

- Eine Signatur $\Sigma' = (\mathcal{S}', \mathcal{P}', \mathcal{F}')$ ist eine Erweiterung der Signatur $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F})$, falls
  - $\mathcal{S} \subseteq \mathcal{S}'$,
  - für jedes $\alpha \in \mathcal{S}^*$: $\mathcal{P}_\alpha \subseteq \mathcal{P}'_\alpha$,
  - für jedes $\alpha \in \mathcal{S}^*$ und $s \in \mathcal{S}$: $\mathcal{F}_{\alpha,s} \subseteq \mathcal{F}'_{\alpha,s}$.
- D.h. eine Erweiterung von Σ fügt nur neue Symbole zu Σ hinzu.

  Die Signatur, die letztendlich für Anfragen und Integritätsbedingungen verwendet wird, ist eine Erweiterung der durch das DBMS vorgebenen Datentyp-Signatur.

Inhalt

Interpretationen (1)

Definition:

- Sei die Signatur $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F})$ gegeben.
- Eine Σ-Interpretation $\mathcal{I}$ definiert:
  - eine Menge $\mathcal{I}(s)$ für jedes $s \in \mathcal{S}$ (Wertebereich),
  - Eine Relation $\mathcal{I}(p, \alpha) \subseteq \mathcal{I}(s_1) \times \cdots \times \mathcal{I}(s_n)$ für jedes $p \in \mathcal{P}_\alpha$ und $\alpha = s_1 \ldots s_n \in \mathcal{S}^*$.

    Im Folgenden wird teilweise $\mathcal{I}(p)$ statt $\mathcal{I}(p, \alpha)$ geschrieben, wenn α für die gegebene Signatur Σ eindeutig ist (d.h. p nicht überladen ist).
  - eine Funktion $\mathcal{I}(f, \alpha)\colon\ \mathcal{I}(s_1) \times \cdots \times \mathcal{I}(s_n) \to \mathcal{I}(s)$ für jedes $f \in \mathcal{F}_{\alpha,s}$, $s \in \mathcal{S}$ und $\alpha = s_1 \ldots s_n \in \mathcal{S}^*$.
- Im Folgenden schreiben wir $\mathcal{I}[\ldots]$ anstatt $\mathcal{I}(\ldots)$.

Interpretationen (2)

Beachte:

- Leere Wertebereiche verursachen Probleme, deshalb werden sie normalerweise ausgeschlossen.

  Einige Äquivalenzen gelten nicht, wenn die Wertebereiche leer sein können. Zum Beispiel kann die Pränex-Normalform nur unter der Annahme erreicht werden, dass die Wertebereiche nicht leer sind.

- In Datenbanken kann dies aber vorkommen.

  Z.B. Menge von Studenten (Tupel der Tabelle `STUDENTEN`), wenn die Datenbank gerade erst erstellt wurde. Auch bei SQL kann es Überraschungen geben, wenn eine Tupelvariable über einer leeren Relation deklariert ist.

- Die Wertebereiche der Datentypen wie `int` sind nicht leer.
- Statt Wertebereich sagt man auch Individuenbereich, Universum oder Domäne (engl. Domain).

Interpretationen (3)

- Die Relation $\mathcal{I}[p]$ wird auch die Extension von p genannt (in $\mathcal{I}$).
- Formal gesehen sind Prädikat und Relation nicht gleiche, aber isomorphe Begriffe.

  Ein Prädikat ist eine Abbildung auf die Menge $\{\mathit{true}, \mathit{false}\}$ boolescher Werte. Eine Relation ist eine Teilmenge des kartesischen Produkts $\times$.
- Z.B. ist $\mathtt{X} < \mathtt{Y}$ genau dann wahr in $\mathcal{I}$, wenn $(\mathtt{X}, \mathtt{Y}) \in \mathcal{I}[<]$.

  Im Folgenden werden die Wörter "Prädikatsymbol" und "Relationssymbol" austauschbar verwendet.

Datenbanken und Logik (1)

- Das DBMS definiert eine Datentyp-Signatur $\Sigma_{\mathcal{D}}$ und eine zugehörige Interpretation $\mathcal{I}_{\mathcal{D}}$, die Folgendes festlegen:
  - Namen für die Datentypen (Sorten) und jeweils einen zugehörigen (nichtleeren) Wertebereich.
  - Für jede Sorte Namen für Datentyp-Konstanten (wie `123`, `'abc'`), interpretiert durch Elemente des entsprechenden Wertebereichs.
  - Namen für Datentyp-Funktionen (wie `+`, `SIN`) mit Argument- und Ergebnissorten, interpretiert durch entsprechende Funktionen auf den Wertebereichen.
  - Namen für Datentyp-Prädikate (wie `<`, `LIKE`), interpretiert durch entsprechende Relationen auf den Wertebereichen.

    Statt Relation alternativ auch Funktion, die einen Wahrheitswert liefert.

Datenbanken und Logik (2)

- Ein Datenmodell (wie z.B. das relationale Modell) wird normalerweise (so wie in Kapitel 2) ohne direkte Zuhilfenahme der Logik definiert:
  - Natürlich nimmt man Datentypen (Signatur und Interpretation) als gegeben an,
  - aber dann werden DB-Schemata und Zustände durch Anwendung üblicher mathematischer Formalismen (wie Mengen, Funktionen) definiert.
  - Spätestens bei der Anfragesprache werden aber viele Konstruktionen der mathematischen Logik dupliziert.
- Es ist hilfreich, die durch ein DB-Schema gegebene Signatur zu definieren, und die Teilmenge der Interpretationen, die als DB-Zustände zulässig sind.

Datenbanken und Logik (3)

- Jedes DB-Lehrbuch auf Uni-Niveau enthält zwei formale Anfragesprachen für das relationale Modell:
  - Tupelkalkül (mit Variablen für ganze Tupel) und
  - Bereichskalkül (mit Variablen für Datenwerte).
- Beide basieren sehr stark auf der mathematischen Logik (es sind eigentlich Spezialfälle).
- Wenn man aber z.B. nur den Tupelkalkül definiert (nicht allgemeine Prädikatenlogik), muss man anschließend bei der Definition des Bereichskalküls vieles neu definieren (ähnlich, aber nicht identisch).
- Logik ist auch für andere Datenmodelle nützlich (z.B. ER-Modell).

Inhalt

Datentypen (1)

- Jede Spalte kann nur Werte eines bestimmten Datentyps speichern (unter `CREATE TABLE` definiert).
- Das relationale Modell hängt nicht von einer bestimmten Auswahl an Datentypen ab.
- Verschiedene DBMS bieten unterschiedliche Datentypen, aber Strings und Zahlen verschiedener Länge und Genauigkeit sind immer verfügbar.
- Moderne (objektrelationale) Systeme bieten auch benutzer-definierte Datentypen (Erweiterbarkeit).

  PostgreSQL, DB2, Oracle und SQL Server unterstützen Benutzer-definierte Typen.

Datentypen (2)

- Datentypen definieren neben der Menge der möglichen Werte auch die Operationen auf den Werten.
- Im SQL-86-Standard gab es nur die Datentypfunktionen `+`, `-`, `*`, `/`.

  Die Existenz dieser Funktionen kann man in jedem DBMS erwarten.
- Andere Funktionen können sich von DBMS zu DBMS unterscheiden.

  Die Verwendung kann also zu Portabilitätsproblemen führen. Z.B. ist der Stringkonkatenations-Operator `||` im SQL-92-Standard enthalten, aber SQL Server und Access verwenden stattdessen "`+`" und in MySQL heißt es "`concat(...)`".

Datentypen (3)

Kategorien von Datentypen:

- Relativ standardisiert:
  - Zeichenketten (feste Länge, variable Länge)
  - Zahlen (Integer, Fest- und Gleitkommazahl)
- Unterstützt, aber in jedem DBMS verschieden:
  - Lange Zeichenketten
  - Binäre Daten
  - Zeichenketten in nationalem Zeichensatz
  - Datums- und Zeitwerte
- Benutzer-definierte und DBMS-spezifische Typen.

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. In Access scheint `CHAR` ohne Länge wie `CHAR(255)` behandelt zu werden.

Zeichenketten (2)

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

  Natürlich wird er von allen fünf Systemen unterstützt (Oracle, SQL Server, DB2, Access und MySQL).

- Die Systeme unterscheiden sich im maximalen Wert für die Länge n.

DBMS	Maximales n
Oracle 8.0	2000
DB2 UDB 5	254
SQL Server 7	8000
Access	255
MySQL	255
PostgreSQL	10485760

Zeichenketten (3)

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.

- Dieser Datentyp wurde im SQL-92-Standard hinzugefügt (im SQL-86-Standard nicht enthalten).

- Er wird jedoch wohl von allen modernen DBMS unterstützt.

  Er wird von allen in dieser Vorlesung besprochenen DBMS unterstützt (PostgreSQL, Oracle, DB2, SQL Server, Access, MySQL).

Zeichenketten (4)

- Offiziell heißt der Typ `CHARACTER VARYING(`*n*`)`, aber der Standard erlaubt die Abkürzung `VARCHAR`.
- Die Systeme unterscheiden sich im maximalen Wert für die maximale Länge *n*:

DBMS	Maximales *n*
Oracle 8.0	4000
DB2 UDB 5	254/4000
SQL Server 7	8000
Access	255
MySQL	255
PostgreSQL	10485760

Ist in DB2 *n* größer als 254, ist für diese Spalte keine Sortierung möglich (einschließlich `ORDER BY`, `GROUP BY`, `DISTINCT`).

Zeichenketten-Funktionen (1)

Zeichenketten-Funktionen in Oracle:

- $s_1 || s_2$, CONCAT(s_1, s_2).
- LENGTH(s), INSTR(s, x), INSTR(s, x, n, m).
- INITCAP(s), LOWER(s), UPPER(s), TRANSLATE(s, x, y).
- LPAD(s, n), LPAD(s, n, p), LTRIM(s), LTRIM(s, x), RPAD(s, n), RPAD(s, n, p), RTRIM(s), RTRIM(s, x).
- SUBSTR(s, m), SUBSTR(s, m, n), REPLACE(s, x, y).
- ASCII(s), CHR(n).
- SOUNDEX(s).

Zeichenketten-Funktionen (2)

Zeichenketten-Funktionen im SQL-92-Standard:

- $s_1 || s_2$.
- `CHARACTER_LENGTH`(s), `OCTET_LENGTH`(s).

  `CHARACTER_LENGTH`(s) kann mit `CHAR_LENGTH`(s) abgekürzt werden.
- `LOWER`(s), `UPPER`(s).
- `POSITION`(s_1 `IN` s_2).
- `SUBSTRING`(s `FROM` m `FOR` n).
- `TRIM`(s), `TRIM`(`LEADING` c `FROM` s),
  `TRIM`(`TRAILING` c `FROM` s), `TRIM`(`BOTH` c `FROM` s).

Zeichenketten-Funktionen (3)

Zeichenketten-Funktionen in DB2:

- $s_1 || s_2$, $\texttt{CONCAT}(s_1, s_2)$.
- $\texttt{LENGTH}(s)$.
- $\texttt{LOCATE}(s_1, s_2)$, $\texttt{LOCATE}(s_1, s_2, n)$, $\texttt{POSSTR}(s_1, s_2)$.
- $\texttt{LTRIM}(s)$, $\texttt{RTRIM}(s)$.
- $\texttt{INSERT}(s_1, n, l, s_2)$, $\texttt{REPLACE}(s, f, t)$, $\texttt{LEFT}(s, n)$, $\texttt{RIGHT}(s, n)$, $\texttt{SUBSTR}(s, m)$, $\texttt{SUBSTR}(s, m, n)$.
- $\texttt{LCASE}(s)$, $\texttt{UCASE}(s)$, $\texttt{TRANSLATE}(s)$, $\texttt{TRANSLATE}(s, t, f)$, $\texttt{TRANSLATE}(s, t, f, p)$.

Zeichenketten-Funktionen (4)

Zeichenketten-Funktionen in DB2, fortgesetzt:

- `ASCII`(s), `CHR`(n).
- `DIFFERENCE`(s_1, s_2), `SOUNDEX`(s).
- `GENERATE_UNIQUE`(), `REPEAT`(s, n), `SPACE`(n).

Zeichenketten-Funktionen (5)

Zeichenketten-Funktionen in SQL Server:

- s_1+s_2 (Konkatenation).
- LEN(s).
- ASCII(s), CHAR(n), UNICODE(s), NCHAR(n).
- LOWER(s), UPPER(s).
- LTRIM(s), RTRIM(s).
- LEFT(s, n), RIGHT(s, n), SUBSTRING(s, m, n).
- CHARINDEX(s_1, s_2), CHARINDEX(s_1, s_2, n), PATINDEX(s_1, s_2).

Zeichenketten-Funktionen (6)

Zeichenketten-Funktionen in SQL Server, fortgesetzt:

- `REPLACE`(s, x, y), `STUFF`(s, n, m, x).

  `REPLACE` ersetzt jedes Auftreten von x in s durch y. `STUFF` ersetzt Zeichen mit Position n bis $n + m$ in s durch x.

- `DIFFERENCE`(s_1, s_2), `SOUNDEX`(s).
- `QUOTENAME`(s), `QUOTENAME`(s, q).
- `REPLICATE`(s, n), `SPACE`(n).
- `REVERSE`(s).
- `STR`(x), `STR`(x, l), `STR`(x, l, d).

  Konvertiert eine Zahl in einen String. l ist die Output-Länge.

Zeichenketten-Funktionen (7)

Zeichenketten-Funktionen in MySQL:

- `CONCAT`$(s_1, s_2, \ldots)$, `CONCAT_WS`$(s, s_1, s_2, \ldots)$.
- `LENGTH`(s), `OCTET_LENGTH`(s), `CHAR_LENGTH`(s), `CHARACTER_LENGTH`(s).
- `LOCATE`(s_1, s_2), `POSITION`$(s_1$ `IN` $s_2)$, `LOCATE`(s_1, s_2, n), `INSTR`(s, x).
- `LCASE`(s), `LOWER`(s), `UCASE`(s), `UPPER`(s).
- `LPAD`(s, n, p), `LTRIM`(s), `RPAD`(s, n, p), `RTRIM`(s), `TRIM`(s), `TRIM(LEADING` c `FROM` $s)$, `TRIM(TRAILING` c `FROM` $s)$, `TRIM(BOTH` c `FROM` $s)$.

Zeichenketten-Funktionen (8)

Zeichenketten-Funktionen in MySQL, fortgesetzt:

- `LEFT`(s, n), `RIGHT`(s, n), `SUBSTRING`(s, m, n), `SUBSTRING`$(s$ `FROM` m `FOR` $n)$, `MID`(s, n), `SUBSTRING`(s, m), `SUBSTRING`$(s$ `FROM` $m)$, `SUBSTRING_INDEX`(s, d, n).
- `INSERT`(s_1, n, m, s_2).
- `ASCII`(s), `ORD`(s), `CHAR`$(n_1, \ldots)$.
- `SOUNDEX`(s).
- `SPACE`(n), `REPEAT`(s, n).

Zeichenketten-Funktionen (9)

Zeichenketten-Funktionen in MySQL, fortgesetzt:

- `REVERSE`(s).
- `CONV`(s, b_1, b_2), `CONV`(n, b_1, b_2), `BIN`(n), `OCT`(n), `HEX`(n).
- `ELT`$(n, s_1, s_2, \ldots)$, `FIELD`$(s, s_1, s_2, \ldots)$.
- `FIND_IN_SET`(s_1, s_2), `MAKE_SET`$(n, s_1, s_2, \ldots)$, `EXPORT_SET`(n, s_1, s_2, s_3, m).
- `LOAD_FILE`(f).

Zeichenketten-Funktionen (10)

Zeichenketten-Funktionen in Access:

- ASC(s), ASCB(s), ASCW(s), CHR(n), CHRB(n), CHRW(n).
- CHOOSE($n, s_1, s_2, \ldots$).
- CURDIR(), CURDIR(c), DIR(p), DIR(p, a).
- ENVIRON(v), ENVIRON(n).
- ERROR(), ERROR(n).
- FORMAT(x), FORMAT($x, f, \ldots$), FORMATCURRENCY($x, \ldots$), FORMATDATETIME($x, \ldots$), FORMATNUMBER($x, \ldots$), FORMATPERCENT($x, \ldots$).

Zeichenketten-Funktionen (11)

Zeichenketten-Funktionen in Access, fortgesetzt:

- HEX(n), OCT(n), STR(n), CSTR(n).
- INSTR(s_1, s_2), INSTR(n, s_1, s_2), INSTR(n, s_1, s_2, c), INSTRB(...), INSTRREV($s_1, s_2, \ldots$), REPLACE($s, s_1, s_2, \ldots$).
- LCASE(s), UCASE(s), STRCONV(s, c, l).
- LEFT(s, n), LEFTB(s, n), MID(s, n), MID(s, n, m), RIGHT(s, n), RIGHTB(s, n).
- LEN(s), LENB(s).

Zeichenketten-Funktionen (12)

Zeichenketten-Funktionen in Access, fortgesetzt:

- LTRIM(s), RTRIM(s), TRIM(s).
- MONTHNAME(n), MONTHNAME(n, b). WEEKDAYNAME($n, \ldots$).
- SPACE(n), STRING(n, c).
- STRCOMP(s_1, s_2), STRCOMP(s_1, s_2, c).
- STRREVERSE(s).
- TYPENAME(x).

Zeichenketten-Funktionen (13)

Zeichenketten-Funktionen in PostgreSQL:

- PostgreSQL unterstützt die SQL-92 Funktionen und hat darüber hinaus eine große Zahl eigener Funktionen.

  `||`, `bit_length(`s`)`, `char_length(`s`)`, `octet_length(`s`)`,

  `lower(`s`)`, `upper(`s`)`,

  `trim([leading|trailing|both] [`c`] from `s`)`,

  `trim([leading|trailing|both] [from] `s` [, `c`])`,

  `position(`s_1 `in` s_2`)`,

  `substring(`s `[from` n`] [for` m`])`,

  `substring(`s `from` p`)` (liefere Substring von s, der auf einen POSIX regulären Ausdruck p passt),

  `substring(`s `from` p `for` e`)` (wenn s auf den SQL regulären Ausdruck p passt, wird darin der mit e`"...`e`"` markierte Teil extrahiert, dabei ist e das "Escape-Zeichen"),

  `overlay(`s_1 `placing` s_2 `from` n `[for` m`])` (ersetze Substring), ...

Zeichenketten-Funktionen (14)

Zeichenketten-Funktionen in PostgreSQL, Forts.:

- [https://www.postgresql.org/docs/10/functions-string.html]

  Es gibt dort noch viele weitere Funktionen außer denen, die oben angegeben sind.

- Hinweis: `RPAD(s, n, [c])`
  - füllt s bis zur Länge n mit Leerzeichen (bzw. c) auf.
  - Falls s länger ist als n, wird der Rest abgeschnitten!
- Mit `CHR(n)` kann man jedes Unicode-Zeichen erhalten, und dann mit `||` in eine Zeichenkette einfügen.

  Wenn man die UTF8 Codierung hat, sonst nur ASCII-Zeichen.

Inhalt

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 den anderen vier DBMS.

Zahlen (2)

- `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.
- Wie `NUMERIC` gab es auch `DECIMAL` schon in SQL-86.
- Oracle verwendet `NUMBER(p,s)` und `NUMBER(p)`, versteht aber auch `NUMERIC`/`DECIMAL` als Synonyme.

  Keines der anderen vier Systeme versteht `NUMBER`.

Zahlen (3)

- Die Präzision p (Gesamtanzahl der Ziffern) kann zwischen 1 und einem gewissen Maximum liegen.

DBMS	Maximales p	
Oracle 8.0	38	
DB2 UDB 5	31	
SQL Server 7	28/38	
MySQL	253/254	(arith. ca. 15)
PostgreSQL	1000	

In SQL Server muss der Server mit der Option `/p` gestartet werden, um bis zu 38 Ziffern zu unterstützen (sonst 28). MySQL speichert `NUMERIC(`p`,`s`)` als String von p Ziffern und Zeichen für "-", ".". Aber MySQL macht Berechnungen mit `DOUBLE` (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).

Zahlen (4)

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

  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. In Oracle: Synonym für `NUMBER(38)`.

- `INT`: Abkürzung für `INTEGER`.

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

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

Zahlen (5)

- Zusätzliche, nicht standardisierte Integertypen:
  - `BIT`: In SQL Server (0,1), Access (-1, 0).

    In MySQL gelten `BIT` und `BOOL` als Synonyme für `CHAR(1)`.
  - `TINYINT`: In MySQL $(-128 \,..\, 127)$, in SQL Server $(0 \,..\, 255)$.

    In Access kann der Typ `BYTE` die Werte $0 \,..\, 255$ speichern.
  - `BIGINT`: In DB2 und MySQL $(-2^{63} \,..\, 2^{63}-1)$.

    Der Wertebereich ist größer als `NUMERIC(18)`.
  - MySQL unterstützt z.B. auch `INTEGER UNSIGNED`.

    In MySQL kann man auch eine Ausgabe-Weite definieren (z.B. `INTEGER(5)`) und `ZEROFILL` hinzufügen, um festzulegen, dass z.B. 3 als 0003 dargestellt wird, wenn die Ausgabe-Weite 4 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.
- Z.B. SQL Server (DB2 und MySQL ähnlich):
  - `FLOAT(`p`)`, $1 \leq p \leq 24$, verwendet 4 Bytes.

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

    `REAL` bedeutet `FLOAT(24)`.
  - `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)`.

Zahlen (7)

- Oracle verwendet `NUMBER` (ohne Parameter) als Datentyp für Gleitkommazahlen.

  Oracle versteht auch `FLOAT(`p`)`. `NUMBER` erlaubt das Speichern von Werten zwischen $1.0 * 10^{-130}$ und $9.9\ldots * 10^{125}$ mit 38 Ziffern Präzision.

- Access versteht `REAL`, `FLOAT` (ohne Parameter) und `DOUBLE` (ohne Schlüsselwort `PRECISION`).

- `NUMERIC`, `DECIMAL` etc. sind exakte numerische Datentypen. `FLOAT` ist ein gerundeter numerischer Typ: Rundungsfehler sind nicht wirklich kontrollierbar.

  Z.B. sollte man für Geld nie `FLOAT` verwenden.

Zahlen(8)

Operationen für Zahlen in Oracle:

- $x + y$, $x - y$, $x * y$, x / y, $-x$, $+x$.
- ABS(x), SIGN(x).
- SIN(x), SINH(x), ASIN(x), COS(x), COSH(x), ACOS(x), TAN(x), TANH(x), ATAN(x), ATAN2(x, y).
- CEIL(x), FLOOR(x), ROUND(x), ROUND(x, n), TRUNC(x, n).
- EXP(x), LN(x), LOG(b, x), POWER(x, y),
- MOD(m, n).
- SQRT(x).

Zahlen (9)

Operationen für Zahlen im SQL-92-Standard:

- $x + y$, $x - y$, $x * y$, x / y, $-x$, $+x$.

Operationen für Zahlen in DB2:

- $x + y$, $x - y$, $x * y$, x / y, $-x$, $+x$.
- `ABS`(x), `SIGN`(x).
- `SIN`(x), `ASIN`(x), `COS`(x), `ACOS`(x), `TAN`(x), `COT`(x), `ATAN`(x), `ATAN2`(x, y).
- `CEIL`(x), `FLOOR`(x), `ROUND`(x, n), `TRUNC`(x, n).

Zahlen (10)

Operationen für Zahlen in DB2, fortgesetzt:

- `EXP`(x), `LN`(x), `LOG10`(x), `POWER`(x, y).
- `MOD`(m, n).
- `SQRT`(x).
- `RAND()`.
- `DEGREES`(x).

Zahlen (11)

Operationen für Zahlen in SQL Server:

- $x + y$, $x - y$, $x * y$, x / y, $x \,\%\, y$ (modulo), $-x$, $+x$.
- $x \,\&\, y$, $x \mid y$, $x \,\hat{}\, y$, $\tilde{}\, x$ (Bit-Operationen).
- `ABS`(x), `SIGN`(x).
- `SIN`(x), `ASIN`(x), `COS`(x), `ACOS`(x), `TAN`(x), `ATAN`(x), `ATN2`(x, y), `COT`(x).
- `CEILING`(x), `FLOOR`(x), `ROUND`(x, n), `ROUND`$(x, n, 1)$ (schneidet ab).

Zahlen (12)

Operationen für Zahlen in SQL Server, fortgesetzt:

- `EXP`(x), `LOG`(x), `LOG10`(x), `POWER`(x, y).
- `SQRT`(x), `SQUARE`(x).
- `DEGREES`(x), `RADIANS`(x), `PI`().
- `RAND`(), `RAND`(n).

Zahlen (13)

Operationen für Zahlen in MySQL:

- $x + y$, $x - y$, $x * y$, x / y, $-x$.
- ABS(x), SIGN(x).
- SIN(x), ASIN(x), COS(x), ACOS(x), TAN(x), ATAN(x), ATAN(x, y), ATAN2(x, y), COT(x).
- CEIL(x), FLOOR(x), ROUND(x), ROUND(x, n), TRUNCATE(x, n).
- EXP(x), LOG(x), LOG10(x), POW(x, y), POWER(x, y).
- MOD(m, n), $n \% m$.

Zahlen (14)

Operationen für Zahlen in MySQL, fortgesetzt:

- `SQRT`(x).
- `PI`(), `DEGREES`(x), `RADIANS`(x).
- `RAND`(), `RAND`(n).

  Es gibt Einschränkungen in der Verwendung von `RAND`.

Zahlen (15)

Operationen für Zahlen in Access:

- $x + y$, $x - y$, $x * y$, x / y, $-x$, $+x$.
- ABS(x), SGN(x).
- SIN(x), COS(x), TAN(x), ATN(x).
- ROUND(x), ROUND(x, n), FIX(x), INT(x).
- EXP(x), LOG(x).
- n MOD m.
- SQR(x).

Außerdem gibt es Funktionen, um Zinszahlungen zu berechnen etc.

Zahlen (16)

Operationen für Zahlen in PostgreSQL:

- $x + y$, $x - y$, $x * y$, x / y, $- x$, $+ x$.

  Ganzzahlige Division schneidet Nachkommastellen ab.

- $x \,\%\, y$: Modulo (Divisionsrest)
- $x \,\hat{}\, y$: Exponentiation: x^y.
- $|/ x$: Quadratwurzel: $\sqrt{x}$, $||/ x$: Kubikwurzel: $\sqrt[3]{x}$.
- $n \,\&\, m$: Bit-UND Verknüpfung.

  Entsprechend | für Bit-ODER, # für Bit-XOR, ~ n für bitweise Negation, $n << m$ für Links-Shift, >> für Rechts-Shift.

- $n\,!$: n Fakultät (alternative Schreibweise: $!\,!\, n$).
- $@\, n$: Absolutwert von n.

Zahlen (17)

Operationen für Zahlen in PostgreSQL, Forts.:

- `ABS`(x), `SIGN`(x), `SCALE`(x).
- `CEIL`(x), `CEILING`(x), `FLOOR`(x), `ROUND`(x), `ROUND`(x, n), `TRUNC`(x), `TRUNC`(x, n).
- `SIN`(x), `COS`(x), `TAN`(x), `ASIN`(x), `ACOS`(x), `ATAN`(x), `ATAN2`(x, y), `COT`(x).

  Diese Funktionen arbeiten mit Bogenmaß (Radians). Es gibt jeweils noch eine Funktion mit Suffix `D` für Grad (Degrees), z.B. `SIND`(x).
- `PI`$()$, `RADIANS`(x), `DEGREES`(x).
- `EXP`(x), `LN`(x), `LOG`(x), `LOG`(b, x), `POWER`(x, y).
- `DIV`(n, m), `MOD`(n, m).

Zahlen (18)

Operationen für Zahlen in PostgreSQL, Forts.:

- `SQRT`(x), `CBRT`(x).
- `RANDOM`(), `SETSEED`(x).
- Außerdem gibt es Funktionen für Histogramme.
- [https://www.postgresql.org/docs/10/functions-math.html]

Datentypen in SQL-86

- CHAR[ACTER][(*n*)] [...] markiert optionale Teile.
- NUMERIC[(*p*[,*s*])]
- DEC[IMAL][(*p*[,*s*])]
- INT[EGER], SMALLINT
- FLOAT[(*p*)], REAL, DOUBLE PRECISION
- Diese Typen sollten sehr portabel sein.

  Access unterstützt NUMERIC, DECIMAL, FLOAT(*p*), DOUBLE PRECISION nicht.
- Alle untersuchten Systeme unterstützen zusätzlich VARCHAR, was nicht im SQL-86-Standard enthalten war.

Inhalt

Lange Zeichenketten (1)

Oracle:

- `LONG`: Zeichenketten von bis zu 2GB Länge.
- Die Verwendung von `LONG`-Spalten ist sehr eingeschränkt. Im wesentlichen ist nur möglich, eine Datei in der DB zu speichern und sie wieder abzurufen, aber man kann sie nicht in Bedingungen oder Berechnungen in SQL verwenden.

  Z.B. können `LIKE`, `||`, `LENGTH` und andere Zeichenketten-Funktionen nicht für `LONG`-Werte verwendet werden. Eine Tabelle darf maximal eine Spalte vom Typ `LONG` haben. Die Eingabe/Ausgabe von `LONG`-Werten ist wie gewohnt möglich, z.B. mit `SELECT`. In SQL*Plus legt `SET LONG` *n* die maximale Ausgabelänge fest.

Lange Zeichenketten (2)

Oracle, fortgesetzt:

- Wird für einen langen Text `LIKE` benötigt, muss er in Zeilen/Abschnitte geteilt werden, die separat als `VARCHAR`-Werte gespeichert werden.
- `CLOB`: Character large object ("großes Zeichenobjekt"), bis zu 4GB.

  Das ist wie eine in der DB gespeicherte Datei mit einer eigenen Identität (LOB-Locator). Es ist `LONG` sehr ähnlich.
- `CLOB` ist neu in Oracle8. Oracle7 hatte nur `LONG`.

  Wahrscheinlich wird `LONG` nur für Abwärts-Kompatibilität unterstützt.

Lange Zeichenketten (3)

Oracle, fortgesetzt:

- Unterschiede zwischen `CLOB` und `LONG` sind z.B.:
  - Die Programmierschnittstelle erlaubt wahlfreien Zugriff auf die `CLOB`-Daten (beliebiger Teilstring).

    `LONG`-Werte können nur sequentiell gelesen werden.
  - Tabelle kann mehrere `CLOB`-Spalten enthalten.
  - `CLOB`-Werte können an anderen Orten gespeichert werden als die Tabelle, in der sie auftauchen.
- `CLOB`-Werte darf man nur über PL/SQL-Prozeduren (im Paket `DBMS_LOB`) in Bedingungen verwenden.

Lange Zeichenketten (4)

DB2:

- Auch DB2 hat character large objects (bis zu 2GB).

  Character large objects werden mit einer maximalen Größe versehen, z.B. `CLOB(1M)`, was die Länge des Deskriptors beeinflusst, der verwendet wird, um auf die eigentlichen Daten zu zeigen. Diese Daten werden separat von den Tabellenzeilen gespeichert.

- Schon `VARCHAR`-Spalten der Länge > 254 kann man nicht in `ORDER BY`, `GROUP BY`, `DISTINCT` verwenden.

  Alles, was Sortierung benötigt, ist ausgeschlossen.

- Außerdem können `CLOB(`n`)`-Spalten nicht mit =, <>, <, <=, >, >=, `IN`, `BETWEEN` verwendet werden.

Lange Zeichenketten (5)

DB2, fortgesetzt:

- `CLOB(`n`)`-Spalten können jedoch mit `LIKE` verwendet werden.
- Es gibt auch einen Datentyp `LONG VARCHAR` (bis zu 32 700 Bytes), der wegen Abwärtskompatibilität beibehalten wurde.

Lange Zeichenketten (6)

SQL Server:

- SQL Server hat einen Datentyp "`TEXT`", der bis zu 2GB speichern kann.

  Eigentlich ist die maximale Größe $2^{31} - 1$, d.h. 2 147 483 647.

- `TEXT`-Spalten können in der `WHERE`-Klausel nur mit `LIKE` oder `IS NULL` verwendet werden.

  Z.B. können =, <>, <, <=, >, >=, `IN`, `BETWEEN` nicht verwendet werden.

- `TEXT`-Spalten können nicht mit `DISTINCT`, `ORDER BY`, `GROUP BY` verwendet werden.

Lange Zeichenketten (7)

SQL Server, fortgesetzt:

- `TEXT`-Spalten sind nicht als Argumente für + (String-Konkatenation) erlaubt, aber es gibt einige Datentyp-Funktionen wie `DATALENGTH` oder `SUBSTRING`, die mit `TEXT` funktionieren.

Lange Zeichenketten (8)

MySQL:

- `BLOB`: "Binary large object" von bis zu 64 KBytes.

  Für binäre Daten. Genaue maximale Größe: $2^{16} - 1 = 65\,535$ Bytes.

- `TEXT`: Zeichenkette von bis zu 64 KBytes.

  Der einzige Unterschied zwischen `BLOB` und `TEXT` ist, dass Vergleiche für `BLOB` case-sensitive und für `TEXT` case-insensitive sind.

- `MEDIUMBLOB`/`MEDIUMTEXT`: Max. 16 MB.

  Die genaue maximale Länge ist $2^{24} - 1 = 16\,777\,215$.

- `LONGBLOB`/`LONGTEXT`: Max. 4 GB.

  Die genaue maximale Länge ist theoretisch $2^{32} - 1 = 4\,294\,967\,295$. Die derzeitige Version von MySQL hat jedoch eine Grenze von 16 MB.

Lange Zeichenketten (9)

MySQL, fortgesetzt:

- Im Allgemeinen werden `BLOB` und `TEXT` wie `VARCHAR(`n`)` mit großem n behandelt.

  MySQL entfernt jedoch Leerzeichen am Ende, wenn `VARCHAR`-Werte gespeichert werden, tut dies aber nicht für `BLOB` und `TEXT`.

- Sortierung (`GROUP BY`, `ORDER BY`, `DISTINCT`) funktioniert nur für die ersten 1024 Bytes.

  Diese Grenze kann mit dem Parameter `max_sort_length` erhöht werden. Seit Version 3.23.2 kann man Indexe auf `BLOB`- und `TEXT`-Spalten haben.

- Empfehlung: `SUBSTRING`-Funktion verwenden, um einen Wert zu extrahieren, der sortiert werden kann.

Lange Zeichenketten (10)

Access:

- `MEMO`: Lange Textdaten.

  Die maximale Größe ist 65.535 Zeichen, wenn man die Daten über das Nutzer-Interface eingibt, und 1 GB, wenn man die Daten über das Programm-Interface eingibt. Sogar in den Handbüchern werden verschiedene Werte erwähnt: 64.000 Zeichen, 65.535 Zeichen, 1.2 GB und 2.14 GB. `LONGTEXT` ist ein Synonym für diesen Datentyp.

- `OLEOBJECT`: Z.B. Microsoft Word-Dokument.

  Maximale Größe ist 1 GB. `LONGBINARY` ist ein Synonym für diesen Typ. Man kann `DISTINCT`, `GROUP BY` oder `ORDER BY` für Werte dieses Typs nicht verwenden. Das Handbuch erwähnt die gleichen Einschränkungen für `MEMO`-Daten, aber es schien dort zu funktionieren (in Access 2000). `OLEOBJECT`-Werte werden als "Long binary data" (lange Binärdaten) ausgegeben.

Bitfolgen in SQL-92 (1)

- `BIT(`*n*`)`: Bitfolgen mit genau *n* Bits.

  Konstanten von `BIT(`*n*`)` werden entweder binär, z.B. `B'11000101'`, oder hexadezimal, z.B. `X'C5'`, geschrieben. Es gibt auch `BIT VARYING(`*n*`)`.

- Bitfolgen waren in SQL-86 nicht enthalten und werden von keinem der fünf Systeme unterstützt.

  Jedes System erlaubt jedoch binäre Daten und SQL Server und Access haben sogar einen Typ `BIT` (ohne Länge).

Bitfolgen in SQL-92 (2)

- Der SQL-92-Standard hat keinen booleschen Datentyp.

  SQL Server hat `BIT`, Access hat `YESNO`. Oracle und DB2 haben keine spezielle Unterstützung für boolesche Werte. Meist wird `CHAR(1)` zusammen mit der Bedingung, dass in dieser Spalte nur `'J'` und `'N'` erlaubt sind, verwendet. MySQL behandelt `BIT` und `BOOL` als Synonyme für `CHAR(1)` (ohne Bedingung). In Access und MySQL können die booleschen Spalten als Bedingungen verwendet werden.

Binäre Daten (1)

Oracle:

- `RAW(`n`)`: Binäre Daten mit der Länge von n Bytes.

  n muss zwischen 1 und 2000 liegen.

- Binäre Daten werden z.B. für Grafiken verwendet.

  Immer wenn die Bedeutung der Daten von einem externen Programm interpretiert wird und die Datenbank die Bedeutung nicht kennt.

- Gewöhnlich konvertiert Oracle die Daten, wenn der Nutzer (Client, z.B. PC) einen anderen Zeichensatz als der Server hat. Das wird bei `RAW`-Daten nicht gemacht.

Binäre Daten (2)

Oracle, fortgesetzt:

- `RAW`-Daten werden in SQL-Statements als Zeichenketten mit hexadezimalen Ziffern geschrieben.
- `LONG RAW`: Binäre Daten variabler Länge, bis zu 2GB.
- `BLOB`: Binary large object (großes binäres Objekt), bis zu 4GB.

Binäre Daten (3)

DB2:

- Man kann für String-Spalten festlegen, dass sie binäre Daten enthalten, z.B. (Spalte `ENCRKEY`):

```
ENCRKEY VARCHAR(100) FOR BIT DATA
```

- Dies stellt sicher, dass
  - Vergleiche die exakten binären Codes benutzen,

    Es wird also die "collation sequence" nicht benutzt, die Kleinbuchstaben mit den entsprechenden Großbuchstaben identifizieren könnte.
  - keine Konvertierung zwischen verschiedenen Zeichensätzen ("code pages") erfolgt.

Binäre Daten (4)

DB2, fortgesetzt:

- String-Konstanten können in hexadezimaler Notation geschrieben werden, z.B. `X'FFFF'`.
- Es gibt auch große binäre Objekte, `BLOB(`n`)`, die binäre Daten bis zu 2GB enthalten können.

Binäre Daten (5)

SQL Server:

- `BINARY(n)`: Binäre Daten fester Länge von n Bytes.

  Die Größe n darf maximal 8000 sein.

  Eingabedaten werden mit `0x00` Bytes bis zur Länge n aufgefüllt.

- `VARBINARY(n)`: Binäre Daten variabler Länge.

  n ist die maximale Länge in Bytes. Es kann maximal 8000 sein.

- Konstanten werden in der Form `0xFF1C` geschrieben.

- `IMAGE` kann binäre Daten bis zu 2GB speichern.

  Das ist der BLOB-Typ von SQL Server. Trotz seines Namens interpretiert SQL Server die enthaltenen Daten nicht, es speichert z.B. kein Grafik-Format für das Bild (GIF, JPEG, PNG, usw.).

Binäre Daten (6)

MySQL:

- `CHAR(n) BINARY`: String fester Länge von n Bytes.

  Wird `BINARY` an den String-Datentyp angehängt, werden Vergleiche durchgeführt, indem die Byte-Codes auf exakte Gleichheit verglichen werden. Ohne `BINARY` werden Vergleiche case-insensitiv durchgeführt (Groß- und Kleinbuchstaben werden identifiziert).

- `VARCHAR(n) BINARY`: String variabler Länge $\leq n$ Bytes.
- `BLOB`, `MEDIUMBLOB`, `LONGBLOB`: siehe oben.
- Konstanten kann man hexadezimal schreiben.

  Z.B. `0x612D7A`. Ansonsten müssen beim Einfügen binärer Daten die Bytes 0 (ASCII NUL), 34 (ASCII "), 39 (ASCII ’), 92 (ASCII \) mit Escape-Sequenzen codiert werden: (\0, \", \’, \\).

Binäre Daten (7)

Access:

- `BINARY`, `BINARY(`n`)`: Byte-String.

  In Access 2000 muss $n \leq 510$ gelten.

- Vergleiche mit diesem Typ sind case-sensitiv.
- `OLEOBJECT`, `LONGBINARY`: siehe oben.
- Hexadezimale Konstanten der Form `0x61002D007A00` sind möglich.

  Da Access Unicode verwendet, muss man zwei Bytes pro Zeichen schreiben. Das obige Beispiel ist der String 'a-z'. Man beachte, dass die Bytes eines 16-Bit-Integers "vertauscht" sind. Die Bedingung `'a'=0x6100` wird als wahr ausgewertet.

Datums- und Zeit-Typen (1)

SQL-92:

- DATE: Ein Wert zwischen 0001-01-01 (1. Jan. 0001) und 9999-12-31 (31. Dez. 9999).

  Natürlich sind ungültige Daten wie 1999-02-29 ausgeschlossen.

  DATE-Konstanten werden als Zeichenkette der Form YYYY-MM-DD geschrieben, gekennzeichnet mit dem Schlüsselwort DATE, z.B. `DATE '1965-06-26'`.

- TIME: Zeit (von 00:00:00 bis 23:59:59).

  Man kann auch Bruchteile einer Sekunde speichern. Z.B. erlaubt TIME(3) das Speichern von Werten wie 16:20:31.001. Der Sekunden-Anteil kann bis 61.9 gehen (für Schaltsekunden). TIME-Konstanten werden als Zeichenketten der Form HH:MM:SS[.SSS] geschrieben, wobei das Wort "TIME" vorangestellt wird, z.B. `TIME '09:30:00'`.

- SQL-92 unterstützt auch verschiedene Zeitzonen.

Datums- und Zeit-Typen (2)

SQL-92, fortgesetzt:

- `TIMESTAMP`: `DATE` und `TIME(6)` zusammen.

  Z.B.: `TIMESTAMP '1999-03-23 18:30:00.000000'`.

- `INTERVAL DAY(`p`)`: Zeitintervall in Tagen.

  Werte sind n Tage, $-10^p < n < 10^p$. `INTERVAL DAY(3)` ist eine Differenz zwischen zwei `DATE`-Werten (positiv oder negativ), die 999 Tage nicht überschreiten kann. Eine Konstante ist z.B. `INTERVAL '14' DAY`.

- `INTERVAL HOUR(`p`) TO SECOND`: Differenz zwischen `TIME`-Werten in Stunden ($< 10^p$), Minuten, Sekunden.

  Eine Konstante ist z.B. "`INTERVAL '2:12:35' HOUR TO SECOND`". Anstelle von "`HOUR TO SECOND`" kann man z.B. "`DAY TO MINUTE`" oder eine beliebige andere Genauigkeit angeben.

Datums- und Zeit-Typen (3)

DB2:

- DB2 unterstützt `DATE`, `TIME` und `TIMESTAMP`.

  `TIME` ist immer in Sekunden, eine Präzision kann man nicht festlegen. `TIMESTAMP` hat jedoch Mikrosekunden. Es gibt keine spezifischen Konstanten für Datums- und Zeit-Werte (z.B. wird `TIME '09:30:00'` nicht verstanden), aber Zeichenketten bestimmter Formate werden automatisch konvertiert.

  `DATE`: `'2000-03-27'`, `'03/27/2000'`, `'27.03.2000'`.

  `TIME`: `'09:30:00'`, `'9:30'`, `'09.30.00'`, `'9:30 AM'`.

  `TIMESTAMP`: `'2000-03-27-09.30.00.000000'`.

- DB2 hat keinen `INTERVAL`-Typ.

  Aber gewisse Intervalle können als Argumente von + und - verwendet werden. Z.B. funktioniert `DUE_DATE + 21 DAYS < CURRENT DATE`, aber `CURRENT DATE - DUE_DATE > 21 DAYS` ist ungültig.

Datums- und Zeit-Typen (4)

Oracle:

- Oracle unterstützt die SQL-92-Typen nicht.
- Oracle hat einen Typ für Zeitstempel: `DATE`.

  Trotz seines Namens speichert `DATE` auch die Zeit (in Stunden, Minuten, Sekunden). Wird nur ein Datum festgelegt, geht Oracle von der Uhrzeit 00:00:00am (Mitternacht, Tagesbeginn) aus.

- Es gibt keine spezifischen `DATE`-Konstanten, aber Oracle konvertiert Strings automatisch.

  Strings der Form `'DD-MON-YY'` (z.B. `'23-JAN-99'`) werden akzeptiert, wenn Datumswerte benötigt werden. Man verwende `TO_DATE` u. `TO_CHAR` für andere Formate (einschließlich Zeit). Das Default-Format hängt von `NLS_DATE_FORMAT` ab: In Deutschland wird `'DD.MM.YY'` verwendet.

Datums- und Zeit-Typen (5)

SQL Server:

- `DATETIME`: Vom 1. Jan 1753 bis zum 31. Dez 9999.

  Datum und Zeit (wie Oracles `DATE`). Genauigkeit: 0.003s.

- `SMALLDATETIME`: Vom 1. Jan 1900 bis 6. Juni 2079.

  Genauigkeit: 1 Minute. Benötigt 4 Bytes (`DATETIME`: 8 Bytes).

- Es gibt keine spezifischen `DATETIME`-Konstanten, aber Strings werden automatisch transformiert.

  Das Default-Ausgabeformat ist `'2000-03-29 18:00:00'`. SQL Server versteht jedoch auch andere Formate, z.B. `'March 29, 2000'` (fehlt die Zeit, wird `00:00` angenommen), `'29-MAR-2000 12:00'`, `'03/27/00 9:00 PM'`, `'14:30:00'` (fehlt das Datum, wird `01.01.1900` angenommen).

Datums- und Zeit-Typen (6)

MySQL:

- `DATETIME`: Datum und Zeit (sekundengenau).

  Von `'1000-01-01 00:00:00'` bis `'9999-12-31 23:59:59'`. Das normale Format für Konstanten ist `'YYYY-MM-DD HH:MM:SS'`. Einige alternative Formate werden unterstützt (aber Jahr immer zuerst). MySQL lässt ungültige Daten wie `'2002-02-31 00:00:00'` zu und Jahr, Monat und Tag können Null sein (was eine Art partiellen Nullwert ergibt).

- `DATE`: Datum (von `'1000-01-01'` bis `'9999-12-31'`).

- `TIME`: Uhrzeit und Zeitintervall.

  Von `-838:59:59` bis `838:59:59` (mehr als 34 Tage rückwärts bis 34 Tage vorwärts). Man muss Sekunden in `TIME`-Konstanten einfügen, z.B. wird `'06:15'` als `'00:06:15'` verstanden. Formate sind z.B. `'HH:MM:SS'`, `'HHH:MM:SS'` oder `'DD HH:MM:SS'` (`DD` sind Tage zwischen 0 und 33).

Datums- und Zeit-Typen (7)

MySQL, fortgesetzt:

- `YEAR`

  Von 1901 bis 2155 (1 Byte). Das normale Format ist `'YYYY'`. Zweistellige Jahreszahlen von 70 bis 99 werden in 1970 bis 1999 konvertiert und 00 bis 69 werden als 2000 bis 2069 verstanden.

- `TIMESTAMP`: Datum und Zeit (in Sekunden).

  Werte zwischen 1970 und irgendwann im Jahr 2037. MySQL behandelt eine Spalte dieses Typs auf besondere Art: Sie hat als Default-Wert automatisch das aktuelle Datum/Zeit (bei mehreren `TIMESTAMP`-Spalten gilt dies nur für die erste). Somit wird in der `TIMESTAMP`-Spalte das Datum und die Zeit der Erstellung einer neuen Zeile gespeichert, wenn kein anderer Wert für diese Spalte festgelegt wurde. `TIMESTAMP`-Werte werden als Integer dargestellt, z.B. im Format `YYYYMMDDHHMMSS`. Man kann eine Ausgabe-Größe festlegen, z.B. `TIMESTAMP(8)`: `YYYYMMSS`.

Datums- und Zeit-Typen (8)

Access:

- `DATETIME`: Datum und Zeit zwischen den Jahren 100 und 9999 (sekundengenau).

  Als Gleitkommazahl gespeichert: Der ganzzahlige Teil ist die Anzahl der Tage seit dem 30. Dez. 1899. Der gebrochene Teil ist die Anzahl der Sekunden seit Mitternacht.
  Wahrscheinlich sind Jahre vor 100 ausgeschlossen, um zweistellige Jahresangaben erkennen zu können.

- Konstanten werden z.B. in der Form `#MM-DD-YYYY#` oder `#MM-DD-YYYY HH:MM:SS#` geschrieben.

  Man kann auch "/" anstelle von "-" verwenden oder das Jahr zweistellig angeben.

Nationale Sprachen (1)

- Oracle kann spezielle deutsche Buchstaben wie ä, ö, ü oder ß speichern und es kann z.B. auch mit japanischen Zeichen arbeiten.

  Oracle transformiert Zeichen zwischen verschiedenen Kodierungsschemata, z.B. in einer Client/Server-Umgebung.

- Oracle kann Fehlermeldungen etc. in verschiedenen Sprachen ausgeben.

  Auch Dinge wie eine richtige Sortierungs-Reihenfolge und das Format für Datumswerte etc. kann an nationale Standards angepasst werden.

- Manche Entscheidungen, wie der DB-Zeichensatz, müssen während der Installation getroffen werden.

Nationale Sprachen (2)

SQL-92:

- Der SQL-92-Standard enthält auch einen großen Abschnitt über nationale Zeichensätze (der sich von Oracle unterscheidet).

SQL Server:

- Während der Installation von SQL Server kann ein Zeichensatz für `CHAR`, `VARCHAR`, `TEXT` gewählt werden.
- `NCHAR`, `NVARCHAR`, `NTEXT` speichern Strings in Unicode (2 Bytes je Zeichen). Konstanten z.B. N'aäb'

Nationale Sprachen (3)

DB2:

- Das `CREATE DATABASE`-Statement hat die Optionen `CODESET` und `TERRITORY`.
- Auf diese Weise wird ein Zeichensatz (ggf. auch mit 2 Bytes pro Zeichen) festgelegt.
- Die Typen `GRAPHIC`, `VARGRAPHIC`, `LONG VARGRAPHIC` und `DBCLOB` Zeichen mit 2 Byte Codes.

Access:

- Access verwendet Unicode für Text-Typen.

Andere Datentypen (1)

Oracle:

- `BFILE`: Referenz zu einer Betriebssystem-Datei.

  Die Datei selbst wird nicht in der DB gespeichert, nur ihr Name. Im Gegensatz zu `CLOB` und `BLOB` findet keine Transaktionsverwaltung statt. Externe Dateien können über die Datenbank nur gelesen werden.

- `ROWID`: Physischer Zeiger auf eine bestimmte Zeile.

  Die `ROWID` spezifiziert Datei, Block und Tupelnummer. Zugriffe über die `ROWID` sind sehr schnell. Jede Tabelle hat eine "Pseudo-Spalte" `ROWID` (sie kann wie eine normale Spalte unter `SELECT` und `WHERE` verwendet werden). `ROWID`-Komponenten werden z.B. mit `DBMS_ROWID.ROWID_BLOCK_NUMBER(ROWID)` angezeigt.

- Nutzer-definierte PL/SQL Datentypen.

Andere Datentypen (2)

SQL Server:

- `MONEY` und `SMALLMONEY`: Zahlen mit einer Genauigkeit von 1/10 000 einer Geldeinheit.

  `SMALLMONEY` sind 32-Bit-Zahlen von -214 748.3648 bis +214 748.3647, `MONEY` sind 64-Bit (bis zu 922 Mrd.). Eine Konstante ist z.B. `$12.34`.

- `TIMESTAMP`: DB-weit eindeutige Zahl, automatisch erzeugt und bei jedem Update der Zeile geändert.
- `UNIQUEIDENTIFIER`: Global eindeutiger Bezeichner.
- `CURSOR`: Referenz zu einem Cursor.

  Das ist eine SQL-Anfrage (die möglicherweise gerade ausgeführt wird) oder ein Anfrage-Resultat.

Andere Datentypen (3)

DB2:

- `DATALINK`: Referenz zu einer Datei, die außerhalb der DB gespeichert ist (URL).

Access:

- `CURRENCY`: Zahlen mit 4 Stellen nach dem Komma.

  $-922\,337\,203\,685\,477.5808$ bis $922\,337\,203\,685\,477.5807$

  (64 Bit-Zahl). Mit Währungssymbol angezeigt (z.B. $\$10.25$).

- `GUID`: Vom System generierte eindeutige Zahl.

  16 Bytes. Man sollte nicht in Spalten dieses Typs schreiben.

Andere Datentypen (4)

MySQL:

- `ENUM(`$v_1, v_2, \ldots$`)`: Einer der Werte v_i.

  Die Werte werden als String-Konstanten, z.B. `ENUM('MO','DI',...)`, geschrieben. Ein Aufzählungstyp kann bis zu 65 535 verschiedene Werte haben. Wird ein ungültiger Wert (nicht in $\{v_1, v_2, \ldots\}$) eingefügt, fügt MySQL stattdessen den leeren String ein. Die Werte sind durchnummeriert, der leere String hat die Zahl 0. MySQL erlaubt Vergleiche mit Zahlen, man kann auch Operationen für Zahlen anwenden (z.B. +). MySQL sortiert Aufzählungs-Werte nach ihrem numerischen Wert, d.h. in der Reihenfolge, in der sie deklariert wurden.

- `SET(`$v_1, v_2, \ldots$`)`: Jede Teilmenge von $\{v_1, v_2, \ldots\}$.

  Werte sind wieder String-Konstanten. Eine Menge kann max. 64 Elemente haben. Mengenkonstanten werden als String mit den Element-Namen (durch Kommas getrennt) geschrieben (in v_i keine Kommas).