Einführung in Datenbanken

Kapitel 4: SQL: Datentypen

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Wintersemester 2019/20

http://www.informatik.uni-halle.de/~brass/db19/

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 Logik fast nur in SQL-Syntax verlangt.

  Klassische logische Formeln vielleicht als Integritätsbedingungen.

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 $a_1 \ldots a_n$ 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

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

  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 6).

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 $\alpha = s_1 \ldots s_n \in \mathcal{S}^*$, 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.

  SQL hat auch ein Prädikat `LIKE` für einfache Mustervergleiche und `SIMILAR TO` für den Mustervergleich mit regulären Ausdrücken.

- Ist $p \in \mathcal{P}_\alpha$, und $\alpha = s_1 \ldots s_n$, dann werden $s_1, \ldots, s_n$ 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:

  - $\texttt{1} \in \mathcal{F}_{\epsilon,\texttt{INT}}$ $\qquad$ `1: INT`.
  - $\texttt{'Lisa'} \in \mathcal{F}_{\epsilon,\texttt{string}}$ $\qquad$ `'Lisa': string`.

- Bei Datentypen (z.B. `INT`, `VARCHAR(10)`) 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 Datentyp-Funktionen `+`, `-`, `*`, `/`.

  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:
  - Datums- und Zeitwerte
  - Lange Zeichenketten
  - Binäre Daten
  - Zeichenketten in nationalem Zeichensatz
- 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 Typ `CHAR(`*n*`)` war bereits im SQL-86-Standard enthalten.

  Natürlich wird er von allen normalen RDBMS unterstützt. Die Systeme unterscheiden sich darin, ob *n* in Bytes oder Zeichen gemessen wird.
  Nach dem Standard kann man `CHARACTERS` oder `OCTETS` hinter der Zahl angeben. Wenn nichts angegeben ist, sind es `CHARACTERS`. In realen DBMS sind es oft aber Bytes.

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

DBMS	Maximales *n*	
Oracle 11	2000	Bytes
DB2 11.1	255	Bytes
SQL Server 2019	8000	Bytes
MySQL	255	Zeichen
PostgreSQL	10485760	Zeichen

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. Wie bei `CHAR` ist systemabhängig, ob *n* in Bytes oder Zeichen gemessen wird. Es müssten eigentlich Zeichen sein, sind aber oft Bytes. Ggf. kann man die Einheit explizit wählen.

- 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 11	4000	Bytes
DB2 11.1	32672	Bytes
SQL Server 2019	8000	Bytes
MySQL	65535	Bytes
PostgreSQL	10485760	Zeichen

Bei MySQL sind die Zeilen aber auf 65535 Bytes begrenzt. Vermutlich gibt es auch bei anderen Systemen Grenzen für die Zeilenlänge insgesamt. Außerdem sind bei MySQL Index-Einträge auf 767 Bytes begrenzt, lange Spalten können daher nicht als Schlüssel deklariert werden.

Zeichenkettenvergleich (1)

- Die Prädikate für Zeichenketten-Typen in SQL sind:
  - s_1 `=` s_2: s_1 ist gleich s_2.

    Man beachte, dass in vielen Systemen sich die Zeichenketten z.B. in Leerzeichen am Ende unterscheiden können, und trotzdem als "gleich" gelten. Es ist also nicht die strenge mathematische Gleichheit.
  - s_1 `<>` s_2: s_1 ist verschieden von s_2.
  - s_1 `<` s_2: s_1 kommt in der Sortierreihenfolge vor s_2.
  - s_1 `<=` s_2: s_1 kommt vor s_2 oder ist gleich s_2.
  - s_1 `>` s_2: s_1 kommt in der Sortierreihenfolge nach s_2.
  - s_1 `>=` s_2: s_1 kommt nach s_2 oder ist gleich s_2.
  - s_1 `LIKE` s_2: s_1 passt auf das Muster s_2.
  - s_1 `SIMILAR TO` s_2: s_1 passt auf den regulären Ausdruck s_2.

Zeichenkettenvergleich (2)

- Das Ergebnis eines Vergleichs (=, <>, <, <=, >, >=) zweier Zeichenketten hängt von der explizit oder implizit gewählten "Collation" (oder "Collation Sequence") ab.

  Übersetzungen von "Collation" sind u.a. "Vergleich", "Zusammenstellung".

- Nach dem SQL Standard kann man für jede Spalte einzeln Zeichensatz und Collation wählen, z.B.

```
NAME VARCHAR(20) CHARACTER SET UTF8
                 COLLATE UCS_BASIC
```

  Die verfügbaren Zeichensätze und Collations sind aber stark systemabhängig. Ein Zeichensatz kann verschiedene Collations erlauben. Im Normalfall sollte man für die ganze Datenbank den gleichen Zeichensatz wählen. Der Default wird beim Anlegen der Datenbank bzw. bei der Installation des DBMS festgelegt. Die genaue Vorgehensweise ist systemabhängig. Man teste mindestens deutsche Umlaute. Emojis, falls nötig, stellen besondere Anforderungen, weil sie im Unicode mehr als 16 Bit haben.

Zeichenkettenvergleich (3)

- `'a' < 'b'` usw. und `'A' < 'B'` usw. sollte immer gelten.
- Die voreingestellten Collations unterscheiden sich schon im Vergleich von Klein- und Großbuchstaben:
  - Z.B. gilt in Oracle und DB2: `'B' < 'a'` (binäre Sortierung).
  - In PostgreSQL liegen die Großbuchstaben zwischen den Kleinbuchstaben: `'a' < 'A' < 'b' < 'B'`.
  - SQL Server und MariaDB/MySQL sind case-insensitive, z.B. `'a' = 'A'`.
- Man kann aber natürlich jeweils explizit Collations wählen.

  Jeder Zeichensatz hat eine Default Collation, aber zu einem Zeichenvorrat kann es mehrere mögliche Collations geben. Bei manchen Systemen (z.B. PostgreSQL) kann man den Zeichensatz nur beim Anlegen der Datenbank wählen, aber die Collation für jede Spalte und jeden einzelnen Vergleich.

Zeichenkettenvergleich (4)

- Ist die Reihenfolge (<, =, >) zweier Zeichen bekannt, so ist der Vergleich von Zeichenketten der gleichen Länge klar:
  - Das System vergleicht Zeichen für Zeichen und der erste Vergleich, der nicht "=" ergibt, bestimmt das Ergebnis.

    Es ist aber auch möglich, dass z.B. ß wie ss behandelt wird.

    Dies passt nicht in das Schema eines Zeichen-für-Zeichen Vergleichs.
- Für Zeichenketten veschiedener Länge gibt es
  - Non-Padded Vergleichs-Semantik: Z.B. `'a' < 'a '`.

    Strings werden Zeichen für Zeichen verglichen. Endet ein String und es wurde kein Unterschied gefunden, gilt der kürzere String als kleiner.
  - Blank-Padded Vergleichs-Semantik: Z.B. `'a' = 'a '`.

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

    Die Festlegung "`NO PAD`" oder "`PAD SPACE`" ist Teil der Collation.

Zeichenkettenvergleich (5)

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

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

LIKE

- Mit `LIKE` kann man auf einfache Muster vergleichen.
- In dem Muster (rechter Operand) steht
  - `%` für eine beliebige Folge beliebiger Zeichen.
  - `_` für ein einzelnes beliebiges Zeichen.
- Beispiel:

  ```
  EMAIL LIKE '%@acm.org'
  ```

  Die EMail-Adresse liegt in der Domain `acm.org`,
  d.h. `@acm.org` ist ein Suffix der Zeichenkette.
- Beispiel: Gesucht ist SQL als Teilstring im Aufgaben-Thema:

  ```
  THEMA LIKE '%SQL%'
  ```

- Für `LIKE` sind Leerzeichen am Ende signifikant.

Zeichenketten-Funktionen (1)

- s_1 || s_2: Konkatenation zweier Zeichenketten.

  Dies ist die Syntax im SQL-Standard seit SQL-92, und funktioniert in vielen Systemen, z.B. Oracle, PostgreSQL, DB2, SQLite. Aber es gibt leider wichtige Ausnahmen. Es funktioniert nicht in MS SQL Server, da muss man s_1 + s_2 schreiben. Es funktioniert auch nicht in MariaDB/MySQL, da muss man `CONCAT(`s_1`,`s_2`)` schreiben. Dies geht in MariaDB/MySQL auch mit mehr als zwei Strings, z.B. `CONCAT('a','b','c')`. PostgreSQL, Oracle und DB2 verstehen `CONCAT` auch, aber bei Oracle und DB2 nur mit zwei Argumenten.

- Beispiel: Tabellenzeile mit HTML-Tags erzeugen:

```
SELECT '<tr>' || '<td>' || ANR || '</td>' ||
       '<td>' || Punkte || '</td>' || '</tr>'
FROM   BEWERTUNGEN
WHERE  SID = 101 AND ATYP = 'H'
```

Zeichenketten-Funktionen (2)

- `CHARACTER_LENGTH(`*s*`)`, `CHAR_LENGTH(`*s*`)`:
  Länge einer Zeichenkette in Zeichen.

  Ist im SQL-Standard seit SQL-92. Wird verstanden in: PostgreSQL, MariaDB/MySQL, DB2 (ab Ver. 11, vorher zweiter Parameter für die Einheit nicht optional). Wird nicht verstanden in: Oracle, MS SQL Server. In Oracle schreibt man `LENGTH(`*s*`)` (wird auch in DB2, PostgreSQL, MariaDB/MySQL verstanden), in MS SQL Server `LEN(`*s*`)`.

- `OCTET_LENGTH(`*s*`)`: Länge in Bytes.

  Ist im SQL-Standard seit SQL-92. Wird verstanden in: PostgreSQL, MariaDB/MySQL, DB2. Wird nicht verstanden in: Oracle, MS SQL Server. In Oracle schreibt man `LENGTHB(`*s*`)`, in MS SQL Server `DATALENGTH(`*s*`)`.

- Beispiel: Länge des längsten Namens (für `MAX` siehe Kap. 10):

```
SELECT MAX(CHAR_LENGTH(NACHNAME))
FROM   STUDENTEN
```

Zeichenketten-Funktionen (3)

- `LOWER(s)`: Zeichenkette in Kleinbuchstaben umgewandelt.

  Ist im SQL-Standard seit SQL-92 (zusammen mit `UPPER` unter "fold"). Wird verstanden in: Oracle, PostgreSQL, MariaDB/MySQL, DB2, MS SQL Server. Viele Systeme verstehen alternativ auch `LCASE(s)`.

- `UPPER(s)`: Zeichenkette in Großbuchstaben umgewandelt.

  Ist im SQL-Standard seit SQL-92. Wird verstanden in: Oracle, PostgreSQL, MariaDB/MySQL, DB2, MS SQL Server. Viele Systeme verstehen alternativ auch `UCASE(s)`. Manche DBMS (u.a. Oracle) haben `INITCAP(s)`, was den ersten Buchstaben jedes Wortes in einen Großbuchstaben umwandelt.

- Beispiel: Case-Insensitiver Vergleich:

```
SELECT *
FROM   STUDENTEN
WHERE  UPPER(VORNAME) = UPPER('lisa')
```

  Natürlich könnte man rechts auch einfach `'LISA'` schreiben.

Zeichenketten-Funktionen (4)

- `POSITION(`s_1 `IN` s_2`)`:
  Position des ersten Vorkommens von s_1 in s_2.

  Ist im SQL-Standard seit SQL-92. Wird verstanden in: PostgreSQL, MariaDB/MySQL, DB2. Wird nicht verstanden in: Oracle, MS SQL Server. In Oracle schreibt man `INSTR(`s_2`,`s_1`)` (hier ist der Suchstring an zweiter Position). `INSTR(`s_2`,`s_1`,`p`,`n`)` beginnt die Suche an Position p und liefert das n-te Vorkommen. In MS SQL Server schreibt man für Suchstrings aus einem Zeichen `CHARINDEX(`s_1`,`s_2`)`, sonst `PATINDEX('%`s_1`%', `s_2`)`

- Die Positionen zählen von 1 ab.
- Wenn s_1 in s_2 nicht vorkommt, wird 0 geliefert.
- Beispiel: Position des @-Zeichens in der EMail-Adresse:

```
SELECT EMAIL, POSITION('@' IN EMAIL)
FROM   STUDENTEN
```

Zeichenketten-Funktionen (5)

- `SUBSTRING(`*s* `FROM` *p* `FOR` *n*`)`:
  Teilzeichenkette der Länge *n* von *s*, die an Position *p* beginnt.

  Z.B. liefert `substring('abcde',2,3)` das Ergebnis `'bcd'`.

  Ist im SQL-Standard seit SQL-92. Wird verstanden in: PostgreSQL, MariaDB/MySQL, DB2. Wird nicht verstanden in: Oracle, MS SQL Server.

  In Oracle schreibt man `SUBSTR(`*s*`,`*p*`,`*n*`)`. In MS SQL Server `SUBSTRING(`*s*`,`*p*`,`*n*`)`.

- `SUBSTRING(`*s* `FROM` *p*`)`: Ganzer Rest von *s* ab Position *p*.

  Wie oben. In Oracle schreibt man `SUBSTR(`*s*`,`*p*`)`. Beim MS SQL Server muss man ein hinreichend großes *n* wählen (ggf. mit `LEN` berechnen).

- Beispiel: Domain-Anteil der EMail-Adresse:

```
SELECT SUBSTRING(EMAIL FROM
                 POSITION('@' IN EMAIL)+1)
FROM   STUDENTEN
```

Zeichenketten-Funktionen (6)

- `TRIM(`*s*`)`: Teilzeichenkette von *s*, ohne Leerzeichen am Anfang oder Ende.

  Ist im SQL-Standard seit SQL-92. Wird verstanden in: PostgreSQL, Oracle, MariaDB/MySQL, DB2, MS SQL Server.

- `TRIM(LEADING` *c* `FROM` *s*`)`: Zeichen *c* am Anfang von *s* entfernen. Entsprechend `TRAILING` oder `BOTH`.

  Wenn man das Positions-Schlüsselwort weglässt, ist `BOTH` gemeint. Wenn man *c* weglässt, wird das Leerzeichen angenommen. Ist im SQL-Standard seit SQL-92. Wird verstanden in: PostgreSQL, MariaDB/MySQL, DB2.
  Wird nicht verstanden in: Oracle, MS SQL Server.
  In Oracle gibt es `LTRIM(`*s*`)`, `LTRIM(`*s*`,`*c*`)` `RTRIM(`*s*`)`, `RTRIM(`*s*`,`*c*`)`.
  In MS SQL Server `LTRIM(`*s*`)`, `RTRIM(`*s*`)` und `TRIM(`*c* `FROM` *s*`)`.
  Nach dem SQL Standard ist es ein Fehler, wenn *c* nicht genau ein Zeichen enthält. Viele Systeme erlauben es und verhalten sich aber unterschiedlich (beliebige Zeichen aus *c* oder nur vollständiger String wird entfernt).

Zeichenketten-Funktionen (7)

- Leerzeichen am Ende sind bei der Ausgabe “unsichtbar”.

  In einigen Systemen (z.B. Oracle) sind sie aber für den Vergleich wichtig.

  Andere Systeme ignorieren beim Vergleich Leerzeichen am Ende.

- Man kann Leerzeichen z.B. auf folgende Art sichtbar machen:

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

- Man kann sie auch für den Vergleich entfernen:

```
SELECT *
FROM   STUDENTEN
WHERE  TRIM(NACHNAME) = 'Weiss'
```

- Am besten entfernt man sie vor der Einfügung in die DB.

Zeichenketten-Funktionen (8)

Einige weitere Funktionen (nicht im Standard):

- RPAD(*s*,*n*): Auffüllen mit Leerzeichen bis zur Länge *n*.

  Wenn *s* länger als *n* ist, wird der Rest abgeschnitten!

  LPAD(*s*,*n*): Das gleiche auf der linken Seite (vorne).

- SOUNDEX(*s*): String, der den Klang von *s* codiert.

  Man könnte mit `SOUNDEX(NACHNAME) = SOUNDEX('Meier')` verschiedene Schreibvarianten finden, wenn es für deutsche Aussprache gemacht wäre.

- ASCII(*c*)/UNICODE(*c*)/ORD(*c*): Code des Zeichens *c*.
- CHR(*n*)/CHAR(*n*): Zeichen mit Code *n*.
- TRANSLATE(*s*,*x*,*y*): Ersetze in *s* das *i*-te Zeichen von *x* durch das *i*-te Zeichen von *y* (bzw. lösche es, falls *y* kürzer).
- REPLACE(*s*,*x*,*y*): Ersetze Zeichenkette *x* durch *y* in *s*.

Weitere Informationen

- PostgreSQL

  [https://www.postgresql.org/docs/9.1/functions-string.html]

- Oracle

  [https://docs.oracle.com/database/121/SQLRF/functions002.htm]

- MySQL

  [https://dev.mysql.com/doc/refman/8.0/en/string-functions.html]

- Microsoft SQL Server

  [https://docs.microsoft.com/en-us/sql/t-sql/functions/
  string-functions-transact-sql]

- IBM DB2

  [https://www.ibm.com/support/knowledgecenter/en/SSEPGG_11.1.0/
  com.ibm.db2.luw.sql.ref.doc/doc/c0000767.html]

- Vergleich

  [https://en.wikibooks.org/wiki/SQL_Dialects_Reference/
  Functions_and_expressions/String_functions]

Datentypen in SQL ausprobieren

- Sie können ausprobieren, ob Ihr DBMS einen Typ T versteht, indem Sie ihn in einer Tabellendeklaration verwenden:

  `CREATE TABLE TEST(A` T`);`

- Sie können dann auch ausprobieren, ob eine Konstante c einen Wert hat, der für den Typ akzeptabel ist:

  `INSERT INTO TEST VALUES(`c`);`

- Zu Sicherheit sollten Sie sich die Tabelle nochmal anzeigen lassen, denn eventuell wurde der Wert bei der Einfügung umgewandelt/verändert:

  `SELECT * FROM TEST;`

  Z.B. gibt MySQL öfters keine Fehlermeldung aus, obwohl der Wert bei der Einfügung völlig verändert wird.

Datentyp-Funktionen in SQL ausprobieren

- Viele DBMS erlauben `SELECT` ohne `FROM`, dann können Sie Datentyp-Funktionen z.B. so ausprobieren:

```
SELECT UPPER('abc');
```

- Sie können sich aber auch eine Dummy-Tabelle machen:

```
CREATE TABLE DUMMY(A CHAR(1));
```

  Oracle hat schon eine DUMMY-Tabelle `DUAL`.

- Es ist wichtig, dass die Tabelle genau eine Zeile hat:

```
INSERT INTO DUMMY VALUES ('x');
```

- Dann können Sie die Funktion so testen:

```
SELECT UPPER('abc') FROM DUMMY;
```

  Bei manchen Datentypen zeigt das Standard-Format nicht alle Informationen über den Datenwert an. In Oracle müssen Sie z.B. `TO_CHAR(SQRT(2))` schreiben, wenn Sie alle Nachkommastellen sehen wollen.

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.

Prädikate für Zahlen

- Die Prädikate für Zahl-Datentypen in SQL sind:
  - x = y: x ist gleich y ($x = y$).

    Man beachte, dass Gleitkomma-Zahlen immer mit einer gewissen Ungenauigkeit behaftet sind (sie heißen ja auch "approximate numeric types"). Ein Vergleich auf Gleichheit oder Ungleichheit ist daher problematisch. Man testet bei diesen Typen besser auf ein kleines Intervall um den wahren Wert.
  - x <> y: x ist verschieden von y ($x \neq y$).
  - x < y: x ist kleiner als y ($x < y$).
  - x <= y: x is kleiner oder gleich y ($x \leq y$).
  - x > y: x ist größer als y ($x > y$).
  - x >= y: x is größer oder gleich y ($x \geq y$).

Funktionen für Zahlen (1)

- Im SQL-86/89 Standard gab es nur die vier Grundrechenarten: $x + y$, $x - y$, $x * y$, x / y.

  Plus und Minus können auch monadisch verwendet werden: $+x$. $-x$.

- In SQL-92 änderte sich daran nichts.

  "Numeric value functions": `CHAR_LENGTH`, `OCTET_LENGTH`, `POSITION` (alle für Zeichenketten), `EXTRACT` (Zugriff auf Komponenten von Datums/Zeitwerten).

- SQL-99 brachte zwei neue Funktionen:

  - `ABS(`x`)`: Absolutwert von x.

    Funktioniert in: PostgreSQL, Oracle, MariaDB/MySQL, DB2, MS SQL.

  - `MOD(`n`,`m`)`: Rest bei Division von n durch m.

    Funktioniert in: PostgreSQL, Oracle, MariaDB/MySQL, DB2.

    Funktioniert nicht in: MS SQL Server. Dort schreibt man n `%` m.

    Das verstehen auch PostgreSQL, MySQL (nicht Oracle, DB2).

Funktionen für Zahlen (2)

- SQL 2003 hatte folgende neuen Funktionen:
  - `LN(`x`)`: Natürlicher Logarithmus: $\ln(x)$.

    In MS SQL Server: `LOG`.
  - `EXP(`x`)`: Exponentialfunktion: e^x.
  - `POWER(`x`,`y`)`: Potenz: x^y.
  - `SQRT(`x`)`: Quadratwurzel: $\sqrt{x}$.
  - `FLOOR(`x`)`: Abrundung: $\lfloor x \rfloor$.
  - `CEILING(`x`)`, `CEIL(`x`)`: Aufrundung: $\lceil x \rceil$.

    In Oracle nur `CEIL`, in MS SQL Server nur `CEILING`.
  - `WIDTH_BUCKET(`x`,`y`,`z`,`n`)`: In welchem von n Teilintervallen des Intervalls $[y, z]$ liegt x?

    Falls $x < y$, ist das Ergebnis 0. Falls $x \geq z$, ist das Ergebnis $n + 1$.

    Das Ergebnis 1 bedeutet z.B., dass $y \leq x < y + (z - y)/n$.

    Funktioniert in PostgreSQL, Oracle, DB2. Nicht MySQL, MS SQL.

Funktionen für Zahlen (3)

- Erst SQL-2016 brachte:
  - `SIN(`x`)`, `COS(`x`)`, `TAN(`x`)`, `SINH(`x`)`, `COSH(`x`)`, `TANH(`x`)`, `ASIN(`x`)`, `ACOS(`x`)`, `ATAN(`x`)`: Trigonometrische Funktionen.

    Winkel werden in Bogenmaß (`rad`) gemessen: $180° = \pi = 3.14159$.
    PostgreSQL, MariaDB/MySQL und MS SQL Server kennen nicht `SINH`, `COSH`, `TANH`. Die anderen Funktionen sind problemlos.
  - `LOG(`b`,`x`)`: Logarithmus zur Basis b: $\log_b(x)$.

    Bei MS SQL Server sind die Argumente vertauscht: `LOG(`x`,`b`)`.
    DB2 kennt `LOG` nur mit einem Argument, und dann ist es der Logarithmus zur Basis $e = 2.71828$.
  - `LOG10(`x`)`: Logarithmus zur Basis 10: $\log_{10}(x)$.

    PostgreSQL und Oracle kennen nicht `LOG10`. Bei PostgreSQL liefert `LOG(`x`)` den Logarithmus zur Basis 10.

Funktionen für Zahlen (4)

- Viele der mathematischen Funktionen gab es in realen DBMS schon weit vor der Aufnahme in den SQL Standard.
- Viele Systeme haben auch:
  - `ROUND(`x`)`: x auf die nächste ganze Zahl gerundet.
  - `ROUND(`x`,`n`)`: x auf n Stellen nach dem Komma gerundet.
  - `DEGREES(`x`)`, `RADIANS(`x`)`: Umrechnung zwischen Grad und Bogenmaß.
  - `ATAN2(`x`,`y`)`: Polarwinkel des Punktes (x, y).
  - `PI()`: π (Kreiszahl).
  - `RAND()`: Zufallszahl.

Hinweis zur Division

- Einige Systeme machen eine "Integer Division", wie etwa aus Java bekannt, wenn man ganze Zahlen teilt.

  Das Ergebnis der Division ist dann wieder eine ganze Zahl.

- Z.B. ist dann $3/2 = 1$.

  Diese `WHERE`-Bedingung ist wahr in: PostgreSQL, DB2, MS SQL Server. Sie ist falsch in: Oracle, MariaDB/MySQL.

- Oft macht es einen Unterschied, ob eine Spalte vom Typ `NUMERIC(`n`)` oder vom Typ `INT` am Vergleich beteiligt ist.

  Z.B. Spalte `A` vom Typ `NUMERIC(1)` und `B` vom Typ `INT`. Beide enthalten den Wert 3. Bei PostgreSQL, DB2 und MS SQL Server ist `A/2=1.5` und `B/2=1`. Bei Oracle und MariaDB/MySQL sind beide Ergebnisse `1.5`.

Weitere Informationen

- PostgreSQL

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

- Oracle (wie oben, nur eine Seite für alle Funktionen)

  [https://docs.oracle.com/database/121/SQLRF/functions002.htm]

- MySQL

  [https://dev.mysql.com/doc/refman/8.0/en/numeric-functions.html]

- Microsoft SQL Server

  [https://docs.microsoft.com/en-us/sql/t-sql/functions/mathematical-functions-transact-sql]

- IBM DB2 (wie oben)

  [https://www.ibm.com/support/knowledgecenter/en/SSEPGG_11.1.0/...]

- Vergleich

  [https://en.wikibooks.org/wiki/SQL_Dialects_Reference/Math_functions/Numeric_functions]

  [http://en.wikibooks.org/.../Math_functions/Trigonometric_functions]

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)

- Die maximale Länge von `VARCHAR` is üblicherweise stark beschränkt (systemabhängig, z.B. 4000 Zeichen).
- Will man ganze Textdateien als Tabelleneinträge erlauben, so verwendet man den Typ `CLOB` ("Character Large Object").

  Man kann optional eine Maximallänge angeben, z.B. `CLOB(16M)`.
- Die Verwendung dieser Datentypen in Anfragen ist oft stark eingeschränkt.

  Z.B. kann man oft nicht in Bedingungen verwenden, also z.B. nicht mit `LIKE` nach Teilzeichenketten suchen. Auch Duplikateliminierung oder Sortierung wird nicht unterstützt. Ebenso String-Konkatenation. Man kann nur die Datei in der Datenbank abspeichern, und sie wieder herausholen, oft nur über eine Programmierschnittstelle. Die genauen Einschränkungen sind systemabhängig.

Lange Zeichenketten (2)

- Wenn man eine Suche mit `LIKE` ermöglichen will, wäre eine Lösung, Zeilen einzeln abzuspeichern.
- In einigen DBMS heißt der Datentyp für lange Strings "`TEXT`".

  In MySQL erlaubt `TEXT` nur 64 KByte, es gibt aber `MEDIUMTEXT` bis 16 MByte und `LONGTEXT` bis 4 GByte.

  In MS SQL Server erlaubt `TEXT` bis 2 GByte. Man kann auch `VARCHAR(MAX)` schreiben für Zeichenketten bis 2 GByte Länge.
- Natürlich könnte man in der Datenbank auch nur den Dateinamen speichern, und die eigentlichen Daten außerhalb der Datenbank in normalen Betriebssystem-Dateien.
- Dann verliert man aber die Transaktionsverwaltung der Datenbank (z.B. Backup/Recovery) und es besteht die Gefahr von "Broken Links".

Binäre Daten

- Bei Oracle findet eine automatische Zeichensatz-Konvertierung zwischen Client und Server statt. Für Textdaten ist das recht nützlich, aber binäre Daten werden dadurch zerstört.
- Man muss deswegen einen Datentyp zu wählen, bei dem das DBMS versteht, dass es sich um binäre Daten handelt.
- Der SQL Standard hat die Typen `BINARY`, `BINARY(`*n*`)`, `VARBINARY(`*n*`)`, `BLOB`, `BLOB(`*n*`)`.

  `BLOB` steht für "Binary Large Object". Oracle hat auch den Typ `RAW(n)` für kurze Bytefolgen fester Länge. Einige Systeme verwenden einen Zusatz zu einem Zeichen-Datentyp, z.B. DB2: "`VARCHAR(100) FOR BIT DATA`", MySQL: "`CHAR(`*n*`) BINARY`", "`VARCHAR(`*n*`) BINARY`". MS SQL Server kennt `BINARY(`*n*`)`, `VARBINARY(`*n*`)`, sowie `VARBINARY(MAX)` als BLOB-Typ (2 GByte). Konstanten können in vielen System hexadeximal geschrieben werden, z.B. `X'FFFF'` in DB2 (so auch im Standard) oder `0xFFFF` in MySQL.

Boolesche Werte

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

  Z.B. Oracle 11g kennt auch kein `BOOLEAN`.

- Der Typ `BOOLEAN` wurde erst in SQL-99 eingeführt.

  Einige moderne Systeme, z.B. PostgreSQL, verstehen `BOOLEAN`.
  MS SQL Server hat den Typ `BIT`. MySQL versteht `BOOLEAN` als Abkürzung für `TINYINT`, und erlaubt daher die Werte -128 bis $+127$.

- Die möglichen Werte sind `TRUE` und `FALSE`,
  plus (falls nicht ausgeschlossen) der Nullwert, für den es auch die Konstante `UNKNOWN` (vom Typ `BOOLEAN`) gibt.

- Klassisch wird meist der Typ `CHAR(1)` verwendet, zusammen mit der Integritätsbedingung, dass in dieser Spalte nur `'J'` und `'N'` erlaubt sind.

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 Zeichensätze

- Bei manchen Systemen wird der Datenbank-Zeichensatz bei der Installation des DBMS festgelegt, und kann anschließend nicht geändert werden.

  Man sollte also rechtzeitig prüfen, dass man alle benötigten Zeichen speichern kann. Allerdings darf man bei modernen Systemen hoffen, dass der Default Unicode z.B. in der UTF-8 Codierung ist.

- Nach dem SQL-Standard (und z.B. in MySQL) kann man Zeichensatz und Sortierreihenfolge für jede Spalte einzeln festlegen.

- Nach dem SQL-Standard gibt es String-Typen mit einem implementierungsabhängigen "nationalen" Zeichensatz.

  `NATIONAL CHARACTER(`*n*`)` oder kurz `NCHAR(`*n*`)` und `NCHAR VARYING(`*n*`)`.
  Konstanten werden z.B. `N'abc'` geschrieben. Teils wird das dann für Unicode verwendet, falls der normale Zeichensatz nicht Unicode ist.

Weitere Datentypen

- Referenzen zu externen Dateien (URIs).

  Oracle: `BFILE`, DB2: `DATALINK`.

- Physischer Zeiger auf eine bestimmte Zeile.

  Oracle: `ROWID`.

- Geldbeträge

  SQL Server: `MONEY`, `SMALLMONEY`. Access: `CURRENCY`.

- Eindeutige Zahlen.

  SQL Server: `TIMESTAMP`, `UNIQUEIDENTIFIER`. Access: `GUID`.

- Aufzählungstypen und Mengen.

  MySQL: `ENUM`$(v_1, v_2, \ldots)$, `SET`$(v_1, v_2, \ldots)$.