Datenbank-Programmierung

Kapitel 4: Mustervergleiche mit regulären Ausdrücken

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Sommersemester 2021

http://www.informatik.uni-halle.de/~brass/dbp21/

Lernziele

Nach diesem Kapitel sollten Sie Folgendes können:

- Die wesentlichen Elemente von regulären Ausdrücken aufzählen.
- `SIMILAR TO` in SQL anwenden.

Inhalt

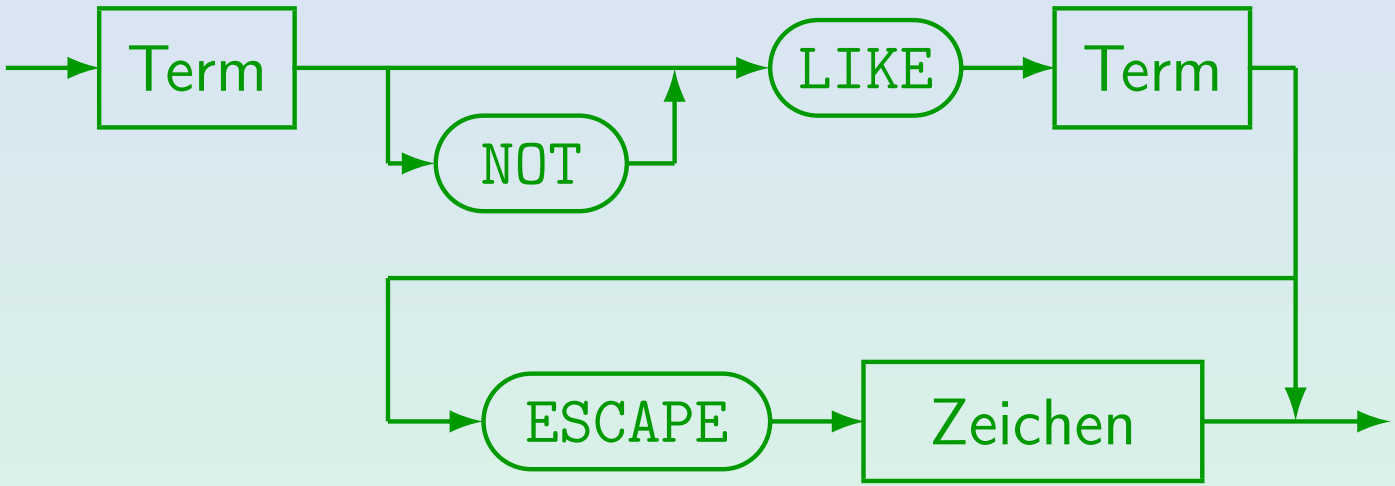
Wiederholung zu LIKE (1)

- Bereits der erste SQL-Standard (SQL-86) enthielt den `LIKE`-Operator für Vergleiche von Zeichenketten mit Mustern.
- Ein Muster ist eine Zeichenkette, die folgendermaßen interpretiert wird:
  - `%`: Passt auf beliebig lange Folge beliebiger Zeichen.
  - `_`: Passt auf ein einzelnes beliebiges Zeichen.
  - Alle anderen Zeichen passen nur auf sich selbst.
- Man kann außerdem ein „Escape“-Zeichen definieren, z.B. „`\`“. Wenn man dieses `%` oder `_` voranstellt, verlieren sie ihre besondere Bedeutung, und passen nur auf sich selbst.

  Da das Escape-Zeichen jetzt auch eine besondere Bedeutung hat, muss es sich auch vorangestellt werden (d.h. verdoppelt werden), wenn man auf genau dieses Zeichen testen will.

Wiederholung zu LIKE (2)

Bedingung (Form 3: LIKE):



- Das rechte Argument wird als Muster interpretiert.
- Z.B.: `EMAIL LIKE '%.pitt.edu'`

  Das ist für alle Email-Adressen wahr, die mit „`.pitt.edu`“ enden.

Wiederholung zu LIKE (3)

- In SQL-86 musste das Muster eine String-Konstante sein. Seit SQL-92 kann man einen beliebigen Term verwenden, um das Muster zu berechnen.

  Z.B. kann man eine andere Spalte als Muster verwenden, und ggf. mit der String-Konkatenation || vorn und hinten Wildcards '%' anfügen.

- Achtung! Punktabzug für `LIKE` statt =:
  - Ich ziehe einen halben Punkt ab, wenn Sie `LIKE` verwenden, wo ein einfaches Gleichheitszeichen = gereicht hätte.

    Also wenn die rechte Seite keine „Wildcard“ % oder _ enthält, und Sie nicht speziell begründen, dass Leerzeichen am Ende der Zeichenkette für den Vergleich wichtig sind.

    Spätere Leser Ihres Codes verstehen schneller, was sie meinen.

    Die Ausführung von = ist schneller als von `LIKE` (meist ersetzt der Optimierer aber automatisch `LIKE` durch =, wenn rechts eine Stringkonstante ohne %/_ steht — bei Spalten aber nicht).

Inhalt

SIMILAR TO (1)

- `SIMILAR TO` wird syntaktisch wie `LIKE` verwendet, unterstützt aber viel mächtigere Muster (eine Variante von regulären Ausdrücken).
  - `%`: Passt auf beliebig lange Folge beliebiger Zeichen.
  - `_`: Passt auf einzelnes, beliebiges Zeichen.
  - $r_1 \mid \cdots \mid r_n$: Passt auf alles, auf das mindestens ein r_i passt.
  - `[`c`]`: Passt auf Zeichen in der Spezifikation c, z.B. `[0-9]`.
  - r`*`: Passt auf 0 oder mehr Vorkommen von r.
  - r`+`: Passt auf 1 oder mehr Vorkommen von r.
  - `(`r`)`: Passt auf regulären Ausdruck r.

    Man benutzt Klammern, um Struktur des Ausdrucks deutlicher zu machen, bzw. um die impliziten Vorrangregeln zu überschreiben (s.u.).

SIMILAR TO (2)

Mengen von Zeichen [...]:

- Normalerweise ist der Inhalt von [...] eine Folge von Zeichen oder Zeichen-Bereichen c_1-c_2 (auch gemischt), z.B.
  - [abc]: Passt auf a, auf b und auf c.
  - [0-9]: Passt auf alle Zeichen, die in der Sortierreihenfolge zwischen 0 und 9 liegen (inklusive der Grenzen).
- Mit [^c] kann umgekehrt Zeichen ausschließen:
  - [^abc]: Passt auf jedes Zeichen außer $\{a, b, c\}$.
- Schließlich gibt es noch einige vordefinierte Zeichen-Mengen: [:ALPHA:], [:UPPER:], [:LOWER:], [:DIGIT:], [:ALNUM:].

SIMILAR TO (3)

Syntax, Vorrang-Regeln:

- Auf äußerster Ebene ist ein regulärer Ausdruck eine durch | getrennte Folge von regulären Termen: $r_1 \mid \cdots \mid r_n$.

  D.h. | bindet am schwächsten. Wenn die Alternativen irgendwie Teil eines größeren Ausrucks sind, müssen sie in Klammern gesetzt werden: $(r_1 \mid \cdots \mid r_n)$.

- Ein regulärer Term besteht aus mehreren regulären Faktoren hintereinander: $r_1 \cdots r_n$.

  Dies passt auf eine Zeichenkette, die sich aus Stücken $s_1 \ldots s_n$ zusammensetzt, so dass r_i jeweils auf s_i passt.

- Ein regulärer Faktor ist ein regulärer Primär-Ausdruck, optional gefolgt von * oder +.

  D.h. wenn die Iterationsmöglichkeit von * oder + auf einen regulären Ausdruck angewendet werden soll, der mehr als ein Zeichen oder Zeichenklasse `[...]` ist, muss man den Operanden in Klammern schreiben: `(...)*` oder `(...)+`.

SIMILAR TO (4)

Syntax, Vorrang-Regeln (Forts.):

- Ein regulärer Primär-Ausdruck ist
  - ein Zeichen (ggf. escaped, s.u.),
  - ein Prozent-Zeichen `%`,
  - ein Unterstrich `_`,
  - eine Zeichenmenge `[...]` oder
  - ein geklammerter regulärer Ausdruck `(...)`.
- Ein Escape-Zeichen (z.B. `\`) muss explizit deklariert werden. Es darf nur vor den Spezial-Zeichen verwendet werden: `[`, `]`, `(`, `)`, `|`, `^`, `-`, `+`, `*`, `_`, `%` und dem Escape-Zeichen selbst.

  Der Ausdruck `\`*c* passt auf das Zeichen *c* (wobei hier `\` als Escape-Zeichen angenommen wird). Auch die Spezialzeichen in `[...]` können escaped werden.

SIMILAR TO (5)

Beispiele:

- `ID SIMILAR TO '[:UPPER:][:DIGIT:]+'`
  - Dieses Muster passt auf einen Großbuchstaben gefolgt von beliebig vielen Ziffern (mindestens einer).
  - Wahr z.B. für: `'A123'`, `'Z1'`.
  - Falsch z.B. für: `'AB123'`, `'A'`, `'123'`, `'123A'`.
  - Leider funktioniert das so nicht in PostgreSQL.

    Das Muster `'[[:upper:]][[:digit:]]+'` würde funktionieren, das wäre aber nach dem Standard nicht korrekt.

- `ID SIMILAR TO '[ABCDEFGHIJKLMNOPQRSTUVWXYZ][0-9]+'`
  - Wie eben, funktioniert auch in PostgreSQL.

    Setzt voraus, dass Ziffern in der Sortierreihenfolge direkt hintereinander.

    Buchstaben auch `[A-Z]`, wenn keine anderen Zeichen dazwischen (abc?).

SIMILAR TO (6)

Beispiele, Forts.:

- `S SIMILAR TO 'ab*'`
  - Wahr für: `'a'`, `'ab'`, `'abb'`, ...
  - Falsch für: `'b'`, `'abab'`.
- `S SIMILAR TO '(ab)*'`
  - Wahr für: `''`, `'ab'`, `'abab'`, ...
  - Falsch für: `'a'`, `'b'`.
- `S SIMILAR TO '[ab]*'` und `S SIMILAR TO '(a|b)*'`
  - Wahr für: `''`, `'a'`, `'b'`, `'bb'`, `'bba'` ...
  - Falsch für: `'c'`.

SIMILAR TO: Erweiterung in SQL-2003

- In SQL-99 gab es nur + und *, um die Vielfachheit des vorstehenden Ausdrucks anzugeben.
- In SQL-2003 wurde dies erweitert:
  - *: Beliebig häufig.
  - +: Mindestens 1 Mal.
  - ?: 0 or 1 Mal (d.h. optional)
  - {*n*}: Exakt *n* Mal.
  - {*n*,}: Mindestens *n* Mal.
  - {*n*,*m*}: Mindestens *n* Mal und maximal *m* Mal.

    Damit kann man auch „maximal *m* Mal“ ausdrücken: {0,*m*}.
- Ausserdem wurden die Zeichenmengen erweitert um [:SPACE:] und [:WHITESPACE:].

Mustervergleich in PostgreSQL (1)

- PostgreSQL hat `SIMILAR TO` mit den SQL-2003 Erweiterungen, aber:
  - Der Rückwärtsschrägstrich „\“ ist als Escape-Zeichen vordefiniert.

    Nach dem Standard gibt es ein Escape-Zeichen nur dann, wenn man die `ESCAPE`-Klausel verwendet. In PostgreSQL braucht man die Escape-Klausel nur, wenn man ein anderes Escape-Zeichen definieren will. Um standard-konform zu programmieren, kann man natürlich auch in PostgreSQL `ESCAPE '\'` schreiben, wenn man das Escape-Zeichen braucht.
  - Die Zeichenmengen `[...]` sind von den „POSIX regular expressions“ übernommen.

    In der Basis sind sie gleich, aber die benannten Zeichenmengen müssen anders geschrieben werden (s.o.), außerdem gibt es Erweiterungen.

Mustervergleich in PostgreSQL (2)

- PostgreSQL hat außerdem „POSIX regular expressions“ mit dem Operator `~` und verschiedenen Varianten.

  `~*`: „passt case-insensitiv auf“, `!~`: „passt nicht auf“, `!~*`: „passt case-insensitiv nicht auf“.

  [https://www.postgresql.org/docs/current/functions-matching.html]

- Ein wesentlicher Unterschied ist, dass hier auch ein Teilstring passen kann, z.B. `'abbbc' ~ 'b*'`.

  Wenn man das nicht will, muss man mit ^ explizit den Anfang markieren, und mit $ explizitd as Ende.

- Außerdem haben „_“ und „%“ keine Spezialbedeutung mehr.

  Man schreibt „`.`“ für ein beliebiges Zeichen, und „`.*`“ für eine beliebige Folge beliebiger Zeichen.

Mustervergleich in PostgreSQL (3)

- Die PostgreSQL-Anleitung warnt davor, reguläre Ausdrücke von möglicherweise bösartigen Benutzern zu übernehmen.

  Z.B. aus Web-Formularen. Siehe auch „SQL Injection“ später.

- Man kann reguläre Ausdrücke schreiben, die extrem viel Laufzeit und Hauptspeicher benötigen.

  Bei der in PostgreSQL verwendeten, recht üblichen Implementierung.
  In der theoretischen Informatik lernt man, zu einem regulären Ausdruck einen deterministischen endlichen Automaten zu erstellen, der die Sprache dann natürlich in Linearzeit erkennt. Die Umwandlung vom leicht zu erstellenden nichtdeterministischen Automaten in den deterministischen kann wegen der Teilmengenkonstruktion zu exponentiellem Aufwand führen (in der Anzahl Zustände des nichtdeterministischen Automaten). Deswegen simuliert den nichtdeterministischen Automaten. Wenn man das aber mit Backtracking macht, kann die Laufzeit ganz übel werden.
  Siehe: [https://swtch.com/ rsc/regexp/regexp1.html]

Mustervergleich in anderen Systemen (1)

- Fast jedes moderne DBMS unterstützt reguläre Ausdrücke. Leider meist nicht mit `SIMILAR TO`.
- MySQL/MariaDB hat `REGEXP` (Synonym: `RLIKE`).

  [https://mariadb.com/kb/en/regexp/]

  [https://dev.mysql.com/doc/refman/8.0/en/regexp.html]

  Die Syntax der regulären Ausdrücke ist etwas ähnlich zu den POSIX regulären Ausdrücken in PostgreSQL: Statt % muss man .* schreiben (und entsprechend „.“ statt _). Außerdem zählt auch ein Treffer im Innern der Zeichenkette, man muss ggf. explizit mit ^ und $ fordern, dass der Treffer sich bis zum Anfang bzw. Ende erstreckt. Bei „\“ ist zu bedenken, dass MySQL es in Strings interpretiert (\\).
- Microsoft SQL Server versteht auch bei gewöhnlichem `LIKE` Zeichenmengen `[...]` und `[^...]`. Das ist aber alles.

  [https://docs.microsoft.com/en-us/sql/t-sql/language-elements/like-transact-sql?view=sql-server-ver15]

Mustervergleich in anderen Systemen (2)

- In Oracle heißt es `REGEXP_LIKE(s,r)`.

  [https://docs.oracle.com/en/database/oracle/oracle-database/21/sqlrf/Pattern-matching-Conditions.html]

  [. . . //Oracle-Regular-Expression-Support.html]

  Die regulären Ausdrücke *r* sind im wesentlichen POSIX-kompatibel, also mit „`.`“ für beliebige Zeichen (außer Newline) und „`.*`“ für eine beliebige Zeichenfolge (innerhalb einer Zeile). Es gibt noch eine Variante mit einem dritten Parameter für Optionen, darin kann man mit `'n'` festlegen, dass „`.`“ auch auf ein Zeilenende passt, entsprechend mit `'m'` („multi-line“), das sich `^` und `$` auf jede einzelne Zeile beziehen, nicht auf die Zeichenkette als Ganzes. Das ist wichtig, da diese Funktion auch für Zeichenketten *s* vom Typ `CLOB` benutzt werden kann, und damit ganze Dateien durchsuchen. Im dritten Parameter kann man auch mit `'i'` den case-insensitiven Vergleich verlangen, bzw. mit `'c'` den case-sensitiven (der Default ist über einen Konfigurationsparameter festgelegt).

Mustervergleich in anderen Systemen (3)

- DB2 ist ähnlich zu Oracle: `REGEXP_LIKE(s, r)`

  [https://www.ibm.com/support/producthub/db2/docs/content/SSEPGG_11.5.0/com.ibm.db2.luw.sql.ref.doc/doc/r0061494.html?pos=2]

  [. . . /SSEPGG_11.5.0/com.ibm.db2.luw.sql.ref.doc/doc/r0061533.html]

- SQLite3 hat keine regulären Ausdrücke eingebaut, aber
  - man kann aber eine Funktion `regexp` hinzubinden, und dann würde der `REGEXP` Operator funktionieren.

    [https://www.sqlite.org/lang_expr.html#regexp]
  - Es gibt außerdem einen Operator `GLOB` (wie `LIKE` zu verwenden), der dem „file globbing“ von UNIX Shells entspricht, und Zeichenklassen `[A-Z]` versteht.

    Dort ist „*“ äquivalent zu „%“ im `LIKE`, und dient nicht zur Iteration des davorstehenden Ausdrucks. Entsprechend steht „?“ für ein beliebiges Zeichen. [https://en.wikipedia.org/wiki/Glob_(programming)]

SQL-2008

- Im SQL-2008 Standard wurde eine Bedingung

  $$s \texttt{ LIKE\_REGEX } r$$

  eingeführt, mit einer „XQuery Regular Expression“ r.

  Dieses Prädikat wurde also im Zuge der Erweiterung von SQL um XML eingeführt. Der Typ von s ist aber „character string“, nicht ein spezieller XML Typ. Die Semantik scheint nach dem Standard im wesentlichen „implementation defined“ zu sein, andererseits bezieht es sich ja klar auf XPath/XQuery (und damit auf `pattern` in den XML Schema Datentypen):
  [https://www.w3.org/TR/xpath-functions-31/#func-matches]
  [https://www.w3.org/TR/xpath-functions-31/#regex-syntax]
  [https://www.w3.org/TR/xmlschema-2/#regexs]

- PostgreSQL 13 versteht dieses Prädikat nicht.

  DB2 11.1 auch nicht, obwohl DB2 stolz auf seine XML-Unterstützung ist.

Inhalt

SUBSTRING in SQL-2003

- Normalerweise werden bei `SUBSTRING` Positionen angegeben:
  - `SUBSTRING(`*s* `FROM` *p*`)`: Rest von *s* ab Position *p*.
  - `SUBSTRING(`*s* `FROM` *p* `FOR` *n*`)`: Teilstück von *s* ab Position *p*, Länge *n*.
- Es gibt aber auch eine Variante mit regulärem Ausdruck:
  - `SUBSTRING(`*s* `SIMILAR` *r* `ESCAPE` *e*`)`: Der reguläre Ausdruck *r* muss genau zwei Vorkommen von *e*" (z.B. \") haben. Wenn *s* auf den regulären Ausdruck ohne die *e*" passt, wird das Teilstück zwischen diesen Markierungen geliefert.

    Tatsächlich werden mit *r* also drei reguläre Ausdrücke spezifiziert, wobei das *e*" als Trennzeichen verwendet wird: Der Ausdruck r_1 passt auf den Präfix s_1 von *s* vor dem gelieferten Stück, der Ausdruck r_2 passt auf das gelieferte Teilstück s_2 von *s* und der Ausdruck r_3 passt auf den Suffix s_3 von *s* nach dem Ergebnis s_2. Wenn es nicht passt: Ergebnis `NULL`.

Einige String-Funktionen in PostgreSQL (1)

- PostgreSQL hat `SUBSTRING` aus dem SQL-99 Standard. Dort gab es aber offenbar einen Copy&Paste Fehler, die Schlüsselworte heißen wie bei der Variante mit numerischen Positionen: `SUBSTRING(s FROM r FOR e)`

  Auch in PostgreSQL 13 werden `SIMILAR` und `ESCAPE` hier nicht verstanden.

- Man kann in PostgreSQL aber auch eine Syntax ohne Schlüsselworte wählen: `substring(s, r, e)`.

  Diese Schreibweise ist im SQL-Standard nicht vorgesehen.

- Eine Variante mit zwei Parametern, `substring(s from r)`, verwendet die POSIX regulären Ausdrücke.

  Der reguläre Ausdruck *r* braucht nur auf ein Stück von *s* zu passen, und dieses Stück wird geliefert. Falls *r* aber Klammern enthält, wird das Stück geliefert, was auf den ersten geklammerten Ausdruck passt (mit der ersten `(`). Bei Bedarf kann man den ganzen Ausdruck klammern.

Einige String-Funktionen in PostgreSQL (2)

- `regexp_replace(s, r, x)` ersetzt in *s* das erste Vorkommen einer Teilzeichenkette, die auf den regulären Ausdruck *r* passt, durch *x*.
- Dabei kann man sich im Ersetzungstext *x* mit `\1` bis `\9` auf den Teil von *s* beziehen, der entsprechend auf den *n*-ten geklammerten Teilausdruck in *r* gepasst hat.
- Außerdem steht `\&` für die ganze Teilzeichenkette von *s*, die auf *r* gepasst hat.
- `regexp_replace(s, r, x, 'g')` ersetzt entsprechend alle Vorkommen, nicht nur das erste.

  Es gibt noch weitere solche Optionen, z.B. `'i'` für case-insensitiven Vergleich.

Einige String-Funktionen in PostgreSQL (3)

- `regexp_split_to_table(s, r)` spaltet einen String *s* in Stücke auf, die durch Teilzeichenketten getrennt werden, die auf den regulären Ausdruck *r* passen.
- Das Ergebnis ist eine Tabelle, die unter `FROM` benutzt werden kann, man kann darüber dann also mit SQL iterieren:

```
SELECT *
FROM   regexp_split_to_table('rot, grün, blau',
                             ', *')
```

regexp_split_to_table
rot
grün
blau

Literatur/Quellen

- PostgreSQL Doumentation: 9.7 Pattern Matching [https://www.postgresql.org/docs/9.3/functions-matching.html]
- Oracle Database, SQL Language Reference, 21c, Feb. 2021 (ab Seite 6-19). [https://docs.oracle.com/en/database/oracle/oracle-database/21/sqlrf/sql-language-reference.pdf]
- Cornel Brücher: Oracle: Regular Expressions. Informatik Aktuell, 28. Mai 2019. [https://www.informatik-aktuell.de/betrieb/datenbanken/oracle-regular-expressions.html]
- ANSI/ISO/IEC International Standard (IS): Database Language SQL — Part 2: Foundation (SQL/Foundation), September 1999
- International Standard ISO/IEC 9075-2, Second edition 2003-12-15 Information technology — Database languages — SQL — Part 2: Foundation
- INCITS/ISO/IEC 9075-2:2016 (2017): Information technology — Database languages — SQL — Part 2: Foundation (SQL/Foundation)
- The Open Group Base Specifications Issue 7, 2018 edition: 9. Regular Expressions [https://pubs.opengroup.org/onlinepubs/9699919799/]
- Russ Cox: Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, Python, Ruby, . . .) [https://swtch.com/ rsc/regexp/regexp1.html]