Texte suchen und finden mit SQL

Inhalt:

1. Einleitung

Eine der Hauptaufgaben von Datenbanken ist es, Datensätze schnell aufzufinden. Moderne DBMS stellen hierzu mehr Funktionen zur Verfügung, als allgemein bekannt ist. Dieses Tutorial zeigt, welche Möglichkeiten der Text-Suche es in SQL grundsätzlich gibt und welche Möglichkeiten speziell MySQL und PostgreSQL dafür zur Verfügung stellen.
Im Folgenden steht "mysql>" für einen Query für MySQL, "pgsql>" für PostgreSQL, "db2>" für DB2, "mssql>" für MS-SQL Server und "oracle>" für Oracle. Verwendet habe ich für meine Tests folgende Versionen: MySQL 4.1.5gamma, PostgreSQL 8.0.0beta2, DB2 Server 8.2, Microsoft SQL Server 2005 sowie Oracle 10g XE.

2. Der LIKE-Operator

2.1. Arbeitsweise

Mit **LIKE** lassen sich einfache Patterns definieren. Wenn keine besonderen Anforderungen an die Suchmuster gestellt werden, so bietet LIKE den einfachsten und schnellsten Zugriff auf Textfelder. SQL-Syntax:

```
text [NOT] LIKE pattern [ESCAPE 'escape-char']
```

Die LIKE-Patterns können dabei zwei verschiedene *Wildcards* enthalten:

%	Ein beliebiger String (kein oder eine beliebige Anzahl an Zeichen)
_	Genau ein Zeichen

Angemerkt sei noch, dass die meisten Datenbanksysteme die Performance optimieren können (z.B über den Zugrif auf einen Index, soweit vorhanden), wenn der Pattern *nicht* mit einem Wildcard beginnt.

2.2. Beispiele

a) Finde alle Datensätze, in deren Text-Feld das Wort "Verschlüsselung" vorkommt:

```
mysql> SELECT * FROM testtab WHERE textfeld LIKE '%Verschlüsselung%';
```

b) Finde alle Datensätze, die nicht mit dem Wort "Verschlüsselung" beginnen:

```
pgsql> SELECT * FROM testtab WHERE textfeld NOT LIKE 'Verschlüsselung%';
```

c) Finde alle Datensätze zu Geschäftspartnern, deren Namen Meier oder Maier ist:

```
oracle> SELECT * FROM kunden WHERE nachname LIKE 'M_ier';
```

2.3. Wildcards und Escape-Character

Sollen % und _ nicht als Wildcards verwendet werden, sondern als normale Zeichen

interpretiert werden, muss ihnen bei MySQL der Escape-Character "\" (Backslash) vorangestellt werden. Wenn wir zum Beispiel alle Datensätze finden wollen, in denen die Zeichenfolge "mysql_query" vorkommt, schreiben wir:

```
mysql> SELECT * FROM testtab WHERE textfeld LIKE '%mysql\_query%';
```

Der Escape-Character wird damit selbst zum Sonderzeichen. Wollen wir bspw. einen Backslash als normales Zeichen finden, z.B. in der Zeichenkette "\Windows", schreiben wir:

```
mysql> SELECT * FROM testtab WHERE textfeld LIKE '%\\Windows%';
```

Mit der **ESCAPE**-Klausel können wir bei MySQL auch einen eigenen Escape-Character festlegen, z.B. '#'. Bei Oracle, DB2 und MS-SQL ist der Backslash allerdings nicht automatisch ein Escape-Character, deswegen *müssen* wir hier einen solchen explizit bestimmen:

```
mysql> SELECT * FROM testtab WHERE textfeld LIKE '%mysql#_query%' ESCAPE '#';
db2> SELECT * FROM testtab WHERE textfeld LIKE '%mysql\_query%' ESCAPE '\';
```

2.4. Unterscheidung von Groß- und Kleinschreibung

Einen gewichtigen Unterschied gibt es, was die *Behandlung von Groß- und Kleinschreibung* angeht: während MySQL und MS-SQL grundsätzlich Groß- und Kleinschreibung ignorieren, wird sie von PostgreSQL, DB2 und Oracle berücksichtigt.
In MySQL wird der **LIKE BINARY**-Operator verwendet, um eine Berücksichtigung von Groß- und Kleinschreibung zu erzwingen. Im folgenden Beispiel soll zwar "Verschlüsselung", nicht aber "verschlüsselung" gefunden werden:

```
mysql> SELECT * FROM testtab WHERE textfeld LIKE BINARY '%Verschlüsselung%';
```

In PostgreSQL wird der **ILIKE**-Operator verwendet, um die Berücksichtigung von Groß- und Kleinschreibung zu deaktivieren. Folgender Query findet sowohl "Verschlüsselung" als auch "verschlüsselung":

```
pgsql> SELECT * FROM testtab WHERE textfeld ILIKE '%Verschlüsselung%';
```

DB2 und Oracle bieten keinen eigenen Operator oder Modifikator, um das Verhalten von LIKE zu ändern. Es gibt aber eine andere Möglichkeit, um die Unterscheidung von Groß- und Kleinschreibung zu umgehen. Man verwendet im Suchpattern nur Kleinbuchstaben und konvertiert den Text in der Spalte, über die gesucht wird, mit der Funktion **LOWER** ebenfalls in Kleinbuchstaben. Folgender Query findet sowohl "Verschlüsselung" als auch "verschlüsselung":

```
db2> SELECT * FROM testtab WHERE LOWER(textfeld) LIKE '%verschlüsselung%';
```

In Oracle heißt diese Funktion **NLS_LOWER**:

```
oracle> SELECT * FROM testtab WHERE NLS_LOWER(textfeld) LIKE
'%verschlüsselung%';
```

Übrigens funktioniert dies auch analog mit einem Vergleich nur zwischen Großbuchstaben. Die entsprechenden Funktionen zur Konvertierung heißen **UPPER** bzw. **NLS_UPPER**.

2.5. Spezielle Erweiterungen

Eine MySQL-Erweiterung zu SQL-99 ist die Fähigkeit, LIKE auf numerische Typen anwenden zu können. Im folgenden Beispiel sollen alle Artikel-Nummern gefunden werden, die mit 2 beginnen:

```
mysql> SELECT * FROM artikel WHERE artikel_nr LIKE '2%';
```

MS-SQL erlaubt in Verbindung mit dem LIKE-Operator die Verwendung einer Liste der erlaubten Zeichen. Diese wird mit [ ] begrenzt, bspw. findet [a-eAB] die Kleinbuchstaben a,b,c,d,e und die Großbuchstaben A,B. Das Zeichen ^ dient der Negation. [^a-eAB] findet alle außer der angegebenen Zeichen.

Als Beispiel sollen die Einträge gefunden werden, die eine durch runde Klammern umfasste dreistellige Zahl enthalten:

```
mssql> SELECT * FROM testtab WHERE textfeld LIKE '%([0-9][0-9][0-9])%';
```

Um die Funktion der Negativliste zu veranschaulichen, sollen nun alle Einträge gefunden werden, die nicht mit dem Buchstaben M beginnen (beachte, dass Groß- und Kleinschreibung bei der Suche ignoriert werden, also sowohl m als auch M als Treffer gezählt werden):

```
mssql> SELECT * FROM testtab WHERE textfeld LIKE '[^m]%';
```

3. Reguläre Ausdrücke nach POSIX-Standard

3.1. Grundlagen

Die meisten DBMS sollten auch in der Lage sein, reguläre Ausdrücke (Regular Expressions) nach dem *POSIX*-Standard zu interpretieren. Damit lassen sich viel genauere Suchmuster bestimmen, als dies mit LIKE möglich ist. MySQL, PostgreSQL und Oracle halten sich ebenfalls an diesen Standard. Für eine ausführliche Beschreibung der Funktionsweise regulärer Ausdrücke sei auf andere Tutorials und einschlägige Fachliteratur verwiesen. Hier soll eine Auflistung der wichtigsten Konstrukte mit kurzer Beschreibung genügen.

Die wichtigsten *Sonderzeichen* regulärer Ausdrücke:

^	Der Anfang einer Zeichenkette
$	Das Ende einer Zeichenkette

.	Ein beliebiges Zeichen
[ ]	umschließen eine Liste erlaubter Zeichen, bspw. findet [a-eAB] die Kleinbuchstaben a,b,c,d,e und die Großbuchstaben A,B. Das Zeichen ^ dient der Negation. [^a-eAB] findet alle außer der angegebenen Zeichen
[:character-class:]	eine Character Class ist eine Abkürzung für eine Zeichenliste. Die Klassen sind von der Locale-Einstellung abhängig (lokaler Zeichensatz). Eine Auflistung findet sich weiter unten.
|	ein logisches "oder", das Alternativen kennzeichnet: "pi|e" findet sowohl "pi" als auch "e"
( )	fasst mehrere Zeichen zu einer Einheit (Gruppe) zusammen, um sie z.B. zusammen mit einem Quantifizierer verwenden zu können
*	die vorausgehende Einheit kann beliebig oft (auch keinmal) hintereinander vorkommen
+	die vorausgehende Einheit kann einmal oder mehrmals vorkommen (aber nicht keinmal)
?	die vorausgehende Einheit kann einmal oder keinmal vorkommen, aber nicht mehrmals
{n}	die vorausgehende Einheit kommt genau n-mal vor
{n,m}	die vorausgehende Einheit kommt mindestens n-mal und höchstens m-mal vor
{n,}	die vorausgehende Einheit kommt mindestens n-mal vor

Eine Auflistung der wichtigsten *Character-Klassen*:

[[:upper:]]	Alle Großbuchstaben
[[:lower:]]	Alle Kleinbuchstaben
[[:alpha:]]	Alle Buchstaben
[[:alnum:]]	Alle Buchstaben und Ziffern
[[:digit:]]	Alle Ziffern
[[:space:]]	Leerzeichen

PostgreSQL und Oracle bieten mächtigere reguläre Ausdrücke als MySQL. Zusätzlich zu den POSIX-Ausdrücken werden Erweiterungen unterstützt, die sich auch in Perl-kompatiblen regulären Ausdrücken wiederfinden, z.B. gierige und nicht-gierige Quantifizierer, Backreferences, Shorthands für bestimmte Charakter-Klassen u.a.
Quantifizierer sind in PostgreSQL grundsätzlich gierig, solange ihnen kein ? nachgestellt wird. Beispielsweise sind "*?" und "{3,}?" die nicht-gierigen Versionen der Quantifizierer "*" und "{3,}". Backreferences, die sich auf vorausgehende Gruppierungen beziehen (in runde Klammern eingefasst), werden ebenfalls in der üblichen Notation geschrieben: \1 für die erste Gruppierung, \2 für die zweite usw. Die wichtigsten Shorthands sind \d (alle Ziffern), \w (Buchstaben, Ziffern und _) und \s (Leerzeichen). Diese Shorthands werden negiert, indem sie groß geschrieben werden: \D (keine Ziffern), \W (keine Wortzeichen), \S (keine Leerzeichen).
Beachte, dass ein Backslash für PostgreSQL ein Sonderzeichen ist und deswegen in diesen Fällen seinerseits durch einen Backslash "entwertet" werden muss, man also "\\1" für "\1" und "\\d" für "\d" schreiben muss. Oracle erkennt aus dem Kontext, welchen Zweck der Backslash erfüllt, deswegen ist dies hier nicht nötig (siehe Beispiele unten). Weiterhin

scheint Oracle die Shorthands für Character Classes (z.B. \d, \w) nicht innerhalb einer mit [ ] begrenzten Liste erlaubter Zeichen zu akzeptieren. In diesem Fall muss man auch hier die Charakterklasse ausschreiben (z.B. [:digit:], [:alnum:], siehe dazu das Beispiel e) weiter unten).

3.2. Implementation regulärer Ausdrücke

In Verbindung mit regulären Ausdrücken verwenden MySQL, PostgreSQL und Oracle allerdings verschiedene Operatoren bzw. Funktionen. MySQL kennt die Operatoren **REGEXP** und **RLIKE**. Sie sind gleichwertig. SQL-Syntax:

```
text [NOT] REGEXP [BINARY] pattern
text [NOT] RLIKE [BINARY] pattern
```

Mit **NOT** kann ein regulärer Ausdruck wie bei LIKE negiert werden. **BINARY** bewirkt eine Suche, die Groß- und Kleinschreibung beachtet.

PostgreSQL kennt kein REGEXP und kein RLIKE. Statt dessen wird in bester Tradition (vgl. die Syntax von Perl) der **~**-Operator (Tilde) verwendet. Seine Anwendung:

PostgreSQL	**Bedeutung**	**Entsprechung in MySQL**
text ~ pattern	text muss auf pattern passen	REGEXP BINARY
text !~ pattern	text darf nicht auf pattern passen	NOT REGEXP BINARY
text ~* pattern	text muss auf pattern passen, Groß-/Kleinschreibung unbeachtet	REGEXP
text !~* pattern	text darf nicht auf pattern passen, Groß-/Kleinschreibung unbeachtet	NOT REGEXP

Oracle kennt reguläre Ausdrücke erst seit der Version 10g. Dafür steht die Funktion **REGEXP_LIKE** zur Verfügung:

```
[NOT] REGEXP_LIKE(text, pattern [, suchoptionen])
```

Der dritte Parameter ist optional und kann einen oder mehrere der folgenden Optionen enthalten:

c	Suche beachtet Groß-/Kleinschreibung (Voreinstellung)
i	Suche beachtet Groß-/Kleinschreibung nicht
n	Erlaubt Operator für beliebige Zeichen inkl. Zeilenumbruch (Punkt)
m	Behandelt Text als Mehrzeiler, falls Zeilenumbruch vorhanden

3.3. Beispiele

a) Die Namen Meier, Maier und Mayr finden (in Großschreibung):

```
mysql> SELECT * FROM kunden WHERE nachname REGEXP BINARY '^M(eie|aie|
ay)r$';
pgsql> SELECT * FROM kunden WHERE nachname ~ '^M(eie|aie|ay)r$';
oracle> SELECT * FROM kunden WHERE REGEXP_LIKE (nachname, '^M(eie|aie|
ay)r$');
```

b) Datensätze finden, in denen eine drei- oder mehrstellige Zahl in runden Klammern enthalten ist:

```
mysql> SELECT * FROM testtab WHERE textfeld RLIKE '\([[:digit:]]{3,}\)';
pgsql> SELECT * FROM testtab WHERE textfeld ~* '\(\\d{3,}\)';
oracle> SELECT * FROM testtab WHERE REGEXP_LIKE (textfeld, '\(\d{3,}\)');
```

c) Datensätze finden, die "DESede" oder "DES-ede" enthalten:

```
mysql> SELECT * FROM testtab WHERE textfeld RLIKE BINARY 'DES-?ede';
pgsql> SELECT * FROM testtab WHERE textfeld ~ 'DES-?ede';
oracle> SELECT * FROM testtab WHERE REGEXP_LIKE (textfeld, 'DES-?ede');
```

d) Datensätze suchen, die keine Leerzeichen enthalten:

```
mysql> SELECT * FROM testtab WHERE textfeld REGEXP '^[^[:space:]]*$';
pgsql> SELECT * FROM testtab WHERE textfeld ~* '^\\S*$';
oracle> SELECT * FROM testtab WHERE REGEXP_LIKE (textfeld, '^\S*$');
```

e) Datensätze finden, die eine E-Mail mit .com-Domain enthalten:

```
mysql> SELECT * FROM testtab WHERE textfeld REGEXP '[[:alnum:]\._-]
+@[[:alnum:]\._-]+\.com';
pgsql> SELECT * FROM testtab WHERE textfeld ~* '[\\w\.-]+@[\\w\.-]+\.com';
oracle> SELECT * FROM testtab WHERE REGEXP_LIKE (textfeld, '[[:alnum:]._-]
+@[[:alnum:]._-]+.com', 'i');
```

f) Datensätze finden, die kursiven oder unterstrichenen Text in HTML enthalten:

```
mysql> SELECT * FROM testtab WHERE textfeld RLIKE '<u>.*<\/u>' OR textfeld RLIKE
'<i>.*<\/i>';
pgsql> SELECT * FROM testtab WHERE textfeld ~* '<(u|i)>.*?<\/\\1>';
oracle> SELECT * FROM testtab WHERE REGEXP_LIKE (textfeld, '<(u|i)>.*?</\1>', 'in');
```

3.4. Weitere Funktionen mit regulären Ausdrucken (Oracle, PostgreSQL)

Oracle stellt Funktionen bereit, die reguläre Ausdrücke nicht nur zum Auffinden von passenden Textstellen verwenden, sondern auch zur Ausgabeformatierung.
Um die Position einer bestimmten Textstelle in einem Text anzuzeigen, wird die Funktion **REGEXP_INSTR** verwenden. Rückgabewert ist eine positive Ganzzahl bzw. 0, wenn keine passende Textstelle gefunden wird. Syntax:

```
REGEXP_INSTR(text, pattern [, position [, vorkommen [, rueckgabe [, suchoptionen]]]])
```

Nur die ersten beiden Parameter sind Pflichtangaben. Der Parameter "position" ist eine positive Ganzzahl (> 0) und gibt an, an welcher Stelle (Anzahl der Zeichen) die Suche beginnen soll. Der Parameter "vorkommen" erwartet ebenfalls eine positive Ganzzahl oder 0 und bestimmt, das wievielte Vorkommen gezählt werden soll. Mit dem Parameter "rueckgabe" kann man festlegen, ob der Anfang oder das Ende des gefundenen Vorkommens bei der Positionsbestimmung herangezogen werden soll (0 = Anfang, 1 = Ende). Die erlaubten Suchoptionen hingegen entsprechen denen von REGEXP_LIKE.

Als Beispiel soll ermittelt werden, an welcher Position das zweite Mal ein durch HTML unterstrichener oder kursiver Text beginnt. Beachte, dass der Text selbst erst *nach* dem HTML-Tag beginnt, wir also den Parameter "rueckgabe" auf 1 setzen, um die Position genau nach dem HTML-Tag zu erhalten:

```
oracle> SELECT titel, REGEXP_INSTR(textfeld, '<(u|i)>', 1, 2, 1, 'i') AS position FROM
testtab;
```

Die Funktion **REGEXP_SUBSTR** dient dazu, die Textelemente, die aufgrund des Pattern gefunden werden, zu extrahieren und den Rest zu verwerfen. Wird keine passende Textstelle gefunden, gibt die Funktion NULL zurück. Syntax:

```
REGEXP_SUBSTR(text, pattern [, position [, vorkommen [, suchoptionen]]])
```

Die Parameter entsprechen in ihrer Funktionsweise den gleichnamigen Parametern der Funktion REGEXP_INSTR. Nun ein kleines Beispiel: es soll anhand der Mailadressen in der Kundentabelle ermittelt werden, welchen Domains die Adressen insgesamt zugeordnet sind. Dazu müssen wir die zweite Zeichenkette extrahieren, die nicht @ enthält. Dadurch bekommen wir die Zeichenkette hinter dem @:

```
oracle> SELECT DISTINCT REGEXP_SUBSTR(email, '[^@]+', 1, 2) AS domains FROM
kunden;
```

Die Funktion **SUBSTRING** in PostgreSQL erlaubt das Extrahieren einer Zeichenkette nicht nur mit genauen Positionsangaben für Anfang und Ende, sondern auch mit Hilfe eines Patterns. Sie entspricht damit in etwa der Oracle-Funktion REGEXP_SUBSTR. Syntax:

```
SUBSTRING(text FROM pattern [FOR escape_character])
```

Der Parameter "pattern" ist auch hier ein regulärer Ausdruck. Optional kann nach FOR ein alternativer Escape-Character angegeben werden. Auch hier sollen als Beispiel die Domains der Mailadressen der Kunden zurückgegeben werden:

```
pgsql> SELECT DISTINCT SUBSTRING(email FROM '[^@]+$') AS domains FROM
kunden;
```

Schließlich können wir mit der Funktion **REGEXP_REPLACE** bestimmte Textstellen durch andere ersetzen. Dabei bleibt der Text, der nicht auf den Pattern passt, erhalten und wird nicht fallengelassen, wie das bei REGEXP_SUBSTR der Fall ist. Wird keine passende Textstelle zur Ersetzung gefunden, wird der ursprüngliche Text zurückgegeben. Syntax:

```
REGEXP_REPLACE(text, pattern [, ersetzung [, position [, vorkommen
[, suchoptionen]]]])
```

Bis auf den dritten Parameter läuft alles wie gehabt. Der Parameter "ersetzung" gibt die Zeichenkette an, durch die alle passenden Textstellen ersetzt werden sollen. Dabei sind auch Backreferences erlaubt. Wird als Zeichenkette zur Ersetzung ein leerer String übergeben oder der Parameter ganz weggelassen, werden alle gefundenen Textstellen einfach gestrichen.

Als Beispiel sollen in HTML-Dokumenten die Textformatierungen "kursiv" und "unterstrichen" durch Fettdruck ersetzt werden.

```
oracle> SELECT REGEXP_REPLACE(textfeld, '<(i|u)>(.*?)</\1>', '<b>\2<b>', 1, 0, 'in')
AS newtext FROM testtab;
```

4. Reguläre Ausdrücke nach SQL-99 (PostgreSQL)

PostgreSQL implementiert reguläre Ausdrücke nach *SQL-99*. Dabei handelt es sich um reguläre Ausdrücke, die Charakteristiken des LIKE-Operators und gewöhnlichen regulären Ausdrücken, ohne aber die Flexibilität letzterer zu erreichen. Diese Art von regulären Ausdrücken steht mit dem **SIMILAR TO**-Operator. SQL-Syntax:

```
text [NOT] SIMILAR TO pattern [ESCAPE 'escape-char']
```

Folgende *Sonderzeichen* (und nur diese) sind definiert:

%	ersetzt keins oder eine beliebige Anzahl an Zeichen
_	ersetzt genau ein Zeichen
|	ein logisches "oder", das Alternativen kennzeichnet
[ ]	umschließen eine Liste erlaubter Zeichen wie in Regulären Ausdrücken (POSIX)
( )	fasst mehrere Zeichen zu einer Einheit (Gruppe) zusammen
*	die vorausgehende Einheit kann beliebig oft (auch keinmal) hintereinander vorkommen
+	die vorausgehende Einheit kann einmal oder mehrmals vorkommen (aber nicht keinmal)

Beachte, dass der Pattern genau wie bei LIKE auf den gesamten Text oder String passen muss (während reguläre Ausdrücke von sich aus ein Vorkommen irgendwo im Text finden).

Einige einfache Beispiele sollen an dieser Stelle genügen:

a) Die Namen Meier, Maier und Mayr finden (in Großschreibung):

```
pgsql> SELECT * FROM kunden WHERE nachname SIMILAR TO 'M(eie|aie|ay)r';
```

b) Datensätze finden, die kursiven oder unterstrichenen Text in HTML enthalten:

```
pgsql> SELECT * FROM testtab WHERE textfeld SIMILAR TO '%<i>%</i>%' OR textfeld SIMILAR TO '%<u>%</u>%';
```

c) Datensätze finden, die nur aus den Kleinbuchstaben a-r und Ziffern bestehen dürfen:

```
pgsql> SELECT * FROM testtab WHERE textfeld SIMILAR TO '[[:digit:]a-r]+';
```

5. Volltextsuche mit MySQL

5.1. Anlegen eines Volltextindex

MySQL bietet eine besondere Art der Volltextsuche, welche auf *Relevanzkalkulation* basiert. Um diese Art der Suche nutzen zu können, müssen die abgefragten Spalten mit einem **FULLTEXT**-Index versehen sein. Damit beschränkt sich die Möglichkeit einer Volltext-Suche leider auf MyISAM-Tabellen.

Legen wir für die später folgenden Beispiele eine Tabelle "dbnews" an mit einen FULLTEXT-Index auf allen Feldern, die Texte oder Strings enthalten, dazu einige Beispieleinträge.

```
mysql> CREATE TABLE dbnews (id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY, titel VARCHAR(200) NOT NULL, inhalt TEXT NOT NULL, FULLTEXT(titel,inhalt));
mysql> INSERT INTO dbnews VALUES
    -> (NULL, 'MySQL 4.1 vor Release', 'Das beliebte DBMS MySQL erscheint bald in der Version ...'),
    -> (NULL, 'Neue C-API in MySQL', 'Bald gehören die alten mysql-Funktionen dem alten Eisen an ...'),
    -> (NULL, 'MySQL vs. PostgreSQL', 'In diesem Vergleich zweier frei erhältlicher Datenbanken ...'),
    -> (NULL, 'Neue ODBC-Treiber', 'Diese neuen Treiber bringen Verbesserungen in der Performance ...'),
    -> (NULL, 'Tutorial Stored Procedures', 'Diese Tutorial zeigt den Einsatz von Stored Procedures in MySQL ...'),
    -> (NULL, 'Übersicht DBMS', 'Hier wollen wir verschiedene Datenbanken vergleichen, darunter MySQL ...');
```

5.2. Ausführen einer Volltextsuche

Die eigentliche Suche wird mit **MATCH AGAINST** durchgeführt. SQL-Syntax:

```
MATCH (spalte1 [, spalte2, ...]) AGAINST (ausdruck [IN BOOLEAN MODE | WITH QUERY EXPANSION])
```

Das Statement liefern eine positive Fließkommazahl (Typ FLOAT) zurück, die die Relevanz ausdrückt (wobei 0 keine Übereinstimmung bedeutet). Verwendet innerhalb einer WHERE-Klausel bewirkt dies, dass die Ausgabe der Treffer *bereits sortiert* erfolgt, mit dem Datensatz, der die größte Relevanz aufweist, am Anfang. Beachte, dass die Auflistung der Spalten hinter MATCH *exakt* mit den FULLTEXT-Spalten der abgefragten

Tabelle übereinstimmen muss.
Suchen wir nun nach dem (exakten) Wort "Datenbanken" in allen indizierten Feldern:

```
mysql> SELECT * FROM dbnews WHERE MATCH(titel, inhalt)
AGAINST('Datenbanken');
```

5.3. Suche mit "blind query expansion"

Nun stellt sich das Problem, dass ein Benutzer, der nach "Datenbanken" sucht, auch solche Einträge finden will, in denen ein DBMS explizit mit Namen genannt wird, z.B. "MySQL", "PostgreSQL" usw. Um dies zu bewerkstelligen, unterstützt MySQL seit Version 4.1.1 den Zusatz **WITH QUERY EXPANSION**. Dies wird möglich durch eine doppelte interne Suche: findet MySQL beim ersten Durchlauf "Datenbanken" zusammen mit Wörtern wie "MySQL" oder "PostgreSQL", wird im zweiten Durchlauf nach ebendiesen Wörtern gesucht. Dabei weist MySQL einem Wort, dass in fast jedem Datensatz vorkommt, eine sehr niedrige Relevanz zu, während ein seltenes Wort höhere Relevanz hat. Richtig wirksam wird dieses Feature erst bei größeren Tabellen. Ein grundsätzliches Problem bleibt aber bestehen: es kann viel Unnützes zurückgegeben werden, wodurch sich diese sog. "blind query expansion" nur bei kurzen Suchausdrücken anbietet.
Verbessern wir nun unsere Abfrage:

```
mysql> SELECT * FROM dbnews WHERE MATCH(titel, inhalt) AGAINST('Datenbanken'
WITH QUERY EXPANSION);
```

5.4. Der boolesche Suchmodus

Wenn wir nun gezielt nach "MySQL" suchen, werden wir eine Überraschung erleben:

```
mysql> SELECT * FROM dbnews WHERE MATCH(titel, inhalt) AGAINST('MySQL');
```

Obwohl "MySQL" in 5 von 6 Einträgen enthalten ist, bekommen wir ein leeres Ergebnis zurück. Das ist eine Art Schutzmechanismus von MySQL: kommt ein Wort in mehr als der Hälfte aller Einträge vor, gibt MySQL kein Ergebnis zurück. Verständlich, wenn man sich eine Tabelle mit 200000 Einträgen vorstellt. Um dieses Verhalten abzustellen, verwenden wir die Erweiterung **IN BOOLEAN MODE** (verfügbar seit MySQL 4.0.1, nicht zusammen mit WITH QUERY EXPANSION verwendbar):

```
mysql> SELECT id, MATCH(titel, inhalt) AGAINST('MySQL' IN BOOLEAN MODE)
FROM dbnews WHERE MATCH(titel, inhalt) AGAINST('MySQL' IN BOOLEAN MODE);
```

Da die Rückgabe von MATCH AGAINST keine Gleitkommazahl mehr ist, sondern ein boolescher Wert, können die Ergebnisse nicht mehr sortiert werden. Außerdem kann eine boolesche Volltext-Suche auch auf Tabellen und Felder angewendet werden, die keinen FULLTEXT-Index enthalten. Dann ist sie aber um einiges langsamer.

5.5. Gewichtung und Verknüpfung einzelner Begriffe

Zusätzlich werden folgende *Operatoren* im Suchausdruck unterstützt:

+	vor ein Wort gesetzt: dieses Wort *muss* enthalten sein
-	vor ein Wort gesetzt: dieses Wort darf *nicht* enthalten sein
>	vor ein Wort gesetzt: dieses Wort wird in der Relevanz höher gewertet
<	vor ein Wort gesetzt: dieses Wort wird in der Relevanz niedriger gewertet
~	(Tilde) vor ein Wort gesetzt: dieses Wort bekommt einen negativen Relevanzwert (wird aber nicht komplett ausgeschlossen)
*	am Ende eines Wortes gesetzt: maskiert verschiedene Endungen dieses Wortes
"	Zeichenketten in Anführungszeichen müssen *genau in dieser Schreibweise* vorkommen
( )	runde Klammern fassen Wörter zu Einheiten zusammen, um auf sie z.B. andere Operatoren anzuwenden

5.6. Beispiele

a) Datensätze finden, die entweder "MySQL" oder "PostgreSQL" enthalten:

```
mysql> SELECT * FROM dbnews WHERE MATCH(titel, inhalt) AGAINST('MySQL PostgreSQL' IN BOOLEAN MODE);
```

b) Datensätze finden, die sowohl "MySQL" als auch "PostgreSQL" enthalten:

```
mysql> SELECT * FROM dbnews WHERE MATCH(titel, inhalt) AGAINST('+MySQL +PostgreSQL' IN BOOLEAN MODE);
```

c) Datensätze finden, die zwar "MySQL" enthalten, aber nicht "PostgreSQL":

```
mysql> SELECT * FROM dbnews WHERE MATCH(titel, inhalt) AGAINST('+MySQL -PostgreSQL' IN BOOLEAN MODE);
```

d) Datensätze finden, die "MySQL" enthalten. Falls auch "PostgreSQL" enthalten ist, sollen sie höher gewertet werden (erreicht so den Effekt, den man ohne boolesche Suche hätte):

```
mysql> SELECT * FROM dbnews WHERE MATCH(titel, inhalt) AGAINST('+MySQL PostgreSQL' IN BOOLEAN MODE);
```

e) Datensätze finden, die die exakte Zeichenkette "kostenlose Datenbank" enthalten (nicht aber z.B. "kostenlose kommerzielle Datenbank"):

```
mysql> SELECT * FROM dbnews WHERE MATCH(titel, inhalt) AGAINST(' "kostenlose Datenbank" ' IN BOOLEAN MODE);
```

f) Datensätze finden, die "MySQL" und "Datenbank" (bzw. alle Wörter, die mit "Datenbank" beginnen) enthalten oder "PostgreSQL" und "Datenbank", wobei PostgreSQL höher gewertet wird:

```
mysql> SELECT * FROM dbnews WHERE MATCH(titel, inhalt) AGAINST('+(<MySQL >PostgreSQL) +Datenbank*' IN BOOLEAN MODE);
```

6. Volltextsuche mit Microsoft SQL Server

Microsoft hat seinem SQL Server ein Volltextmodul spendiert, das noch flexiblere Volltextsuchen erlaubt, als das mit MySQL möglich ist. Zusätzlich zur Verknüpfung einzelner Suchbegriffe mit logischen Operatoren (AND, OR, NOT) und der Gewichtung der einzelnen Suchbegriffe findet der SQL Server dank leistungsfähiger, sprachabhängiger Wortstammerkennung und eines Thesaurus auch gebeugte Formen eines Wortes oder sinnäquivalente Begriffe. Leider kommt die kostenlose Express-Version des SQL Servers ohne Volltextmodul. Die Volltextsuche steht somit nur bei den kommerziellen Versionen zur Verfügung.

6.1. Anlegen eines Volltextindex

Die Volltextsuche unterliegt nicht der Beschränkung auf ein bestimmtes Tabellenformat, wie das bei MySQL der Fall ist, sondern kann grundsätzlich auf alle CHAR-, VARCHAR- und TEXT-Felder angewendet werden, sofern die Tabelle einen eindeutigen Index besitzt. Dies ist in Form eines Primärschlüssels aber in der Regel gegeben.
Bevor wir loslegen, müssen wir noch überprüfen, ob das Volltextmodul bereits installiert ist und der Dienst gestartet wurde. Gegebenenfalls muss es nachinstalliert werden. Als nächstes muss überprüft werden, ob die Datenbank, mit der wir arbeiten werden (in meinem Beispiel wieder "test"), für die Volltextindizierung aktiviert wurde. Mit folgendem Query kann dies abgefragt werden:

```
mssql> SELECT DATABASEPROPERTY('test', 'IsFullTextEnabled');
```

Wenn die Volltextindizierung aktiviert ist, liefert der Query 1 zurück. Andernfalls müssen wir die Volltextindizierung von Hand aktivieren (wir gehen ab jetzt davon aus, dass die Datenbank "test" angewählt wurde):

```
mssql> sp_fulltext_database enable;
```

Die später folgenden Beispiele benutzen wieder eine Tabelle "dbnews" und können anhand folgender Beispieldatensätze nachvollzogen werden. Dem Primärschlüssel habe ich einen explititen Namen zugewiesen. Den Grund dafür werden wir später sehen.

```
mssql> CREATE TABLE dbnews(id INT NOT NULL, titel VARCHAR(200) NOT NULL, inhalt TEXT NOT NULL, CONSTRAINT myIndex PRIMARY KEY(id));
mssql> INSERT INTO dbnews VALUES
    -> (1, 'Einführung in SQLJ', 'Anfang des Monats fand eine Schulung zu SQLJ statt'),
    -> (2, 'Neue JDBC-Treiber', 'Die neuen Treiber zum Aufbau einer Verbindung via Java gibt es jetzt zum Download'),
    -> (3, 'DB2 Express', 'Das beliebte DBMS von IBM erscheint in einer kostenlosen Express-Version'),
    -> (4, 'MySQL vs. PostgreSQL', 'Unser Volontär vergleicht zwei frei erhältliche Datenbanken'),
    -> (5, 'Vergleichstest', 'Der Chefredakteur verglich viele verschiedene Datenbanken wie DB2 und Oracle');
```

Der genaue Aufbau des Volltextmoduls kann in der Onlinedokumentation nachgelesen werden. Uns interessiert im Moment nur die Tatsache, dass jeder Volltextindex einem Volltextkatalog zugeordnet sein muss. Sofern noch kein Volltextkatalog existiert, muss man mit **CREATE FULLTEXT CATALOG** einen neuen anlegen. Die (verkürzte) SQL-Syntax:

```
CREATE FULLTEXT CATALOG katalogname [AS DEFAULT]
```

Die genaue Syntax und weitere Optionen sind für uns jetzt nicht relevant und können in der Onlinedokumenation nachgelesen werden. Wir merken uns nur, dass jeder Volltextkatalog einen eindeutigen Namen braucht. Die Option AS DEFAULT kann auch weggelassen werden. Wird sie angegeben, bewirkt dies, dass ein Volltextindex, der keinem Volltextkatalog explizit zugewiesen wird, standardmäßig diesem Katalog zugeordnet wird.
In folgendem Beispiel wird ein Volltextkatalog namens "myCatalog" angelegt:

```
mssql> CREATE FULLTEXT CATALOG myCatalog AS DEFAULT;
```

Der Volltextindex selbst wird mit **CREATE FULLTEXT INDEX** angelegt. Die (verkürzte) Syntax dazu:

```
CREATE FULLTEXT INDEX ON tabellenname (spalte1 [LANGUAGE 'sprache'] [, spalte2 [LANGUAGE 'sprache'], ...]) KEY INDEX indexname [ON katalogname]
```

Auch hier begnüge ich mich mit den wichtigsten Optionen und verweise ansonsten auf die Dokumentation. Angegeben werden muss außer der Tabelle und einer oder mehrerer Spalten (durch Kommata getrennt) der Name des eindeutigen Index der Tabelle. Optional kann zu jeder Spalte die Sprache angegeben werden, wenn sie von der Standardeinstellung abweicht (auf deutschen Systemen ist der Standard in der Regel 'german'). Außerdem lässt sich der Volltextkatalog spezifizieren, dem der Volltextindex zugeordnet werden soll. Wird kein Volltextkatalog genannt, wird der Index dem Katalog zugeordnet, der als letztes mit AS DEFAULT angelegt wurde.
Jetzt wird auch klar, warum ich dem Primärschlüssel beim Anlegen der Tabelle "dbnews" einen expliziten Namen zugewiesen habe. Wir erstellen nun einen Volltextindex über der Tabelle "dbnews" mit dem Schlüssel "myIndex" und beziehen die Spalten "titel" und "inhalt" in den Volltextindex mit ein. Zur Demonstration weisen wir den Volltextindex auch ausdrücklich dem Volltextkatalog "myCatalog" zu, obwohl wir uns das sparen könnten, da "myCatalog" unser Default-Volltextkatalog ist.

```
mssql> CREATE FULLTEXT INDEX ON dbnews (titel, inhalt) KEY INDEX myIndex ON myCatalog;
```

Volltextindex und Volltextkatalog können mit **ALTER FULLTEXT INDEX** bzw. **ALTER FULLTEXT CATALOG** geändert und mit **DROP FULLTEXT INDEX** bzw. **DROP FULLTEXT CATALOG** gelöscht werden. Genaueres findet man in der Dokumentation.

6.2. Der Thesaurus

Im normalen Sprachgebrauch werden "Datenbank" und "Datenbankmanagementsystem" (DBMS) oft gleichgesetzt, obwohl dies nicht ganz richtig ist. Eine "Datenbank" repräsentiert den konkreten Datenbestand, während als DBMS die Software zur

Verwaltung der Datenbanken und zur Steuerung der Benutzertransaktionen bezeichnet wird. Auch die von mir angegebenen Beispieldatensätze enthalten diese Ungenauigkeit. Um es dem Benutzer, der den Unterschied nicht so genau kennt oder gar nicht weiß, was ein DBMS ist, trotzdem zu ermöglichen, auch Einträge mit "DBMS" zu finden, wenn er nur "Datenbank" eingibt, können wir den Thesaurus des SQL Servers bearbeiten. Der Thesaurus wird bei Suchanfragen herangezogen, die auch Synonyme berücksichtigen sollen (siehe Kap. 6.3.3.). Durch den Thesaurus ist das Ergebnis natürlich besser kontrollierbar und genauer, als eine Abfrage mit "blind query expansion" jemals sein könnte.
Der Thesaurus selbst liegt im XML-Format vor. Wenn der SQL Server in C:\Programme installiert wurde, findet man die Thesaurusdateien unter *C:\Programme\Microsoft SQL Server\MSSQL.1\MSSQL\FTData*\. Die Datei mit dem deutschen Thesaurus heißt *tsDEU.xml*, die englische tsENG.xml usw. Nun fügen wir zwischen <thesaurus> und </thesaurus> folgenden Abschnitt ein:

```
<expansion>
  <sub>Datenbank</sub>
  <sub>DBMS</sub>
</expansion>
```

Damit erkennt der Thesaurus "Datenbank" und "DBMS" als gleichwertig an. Beachte, dass der Thesaurus in der Datei nicht auskommentiert sein darf. Nach einem Neustart des Servers werden die Änderungen wirksam.

6.3. Ausführen einer Volltextsuche

Die Volltextsuche selbst wird mit dem Prädikat **CONTAINS** in der WHERE-Klausel ausgeführt. Syntax:

```
CONTAINS (spalte | (spaltenliste) | *, 'suchausdruck' [, LANGUAGE 'sprache'])
```

Als erstes Argument wird die Spalte, die abgefragt werden soll, oder eine Liste von Spalten übergeben. In letzterem Falle muss die Liste durch runde Klammern begrenzt und die einzelnen Spalten durch Kommata getrennt werden. Bei Angabe eines Asterisk (*) werden alle für die Volltextsuche registrierten Spalten der Tabelle durchsucht. Als zweites Argument wird der eigentliche Suchausdruck erwartet. Er wird immer (!) durch einfache Hochkommata begrenzt. Optional lässt sich als drittes Argument wieder die Sprache angeben. Wir richten unser Augenmerk im Folgenden auf den Suchausdruck.

6.3.1. Suche nach einfachen Begriffen

Einfache Wörter oder Wortverbindungen, nach denen gesucht werden soll, werden immer in doppelte Hochkommata eingefasst. Beachte, dass dadurch die einfachen Hochkommata, durch die der komplette Suchausdruck begrenzt wird, nicht entfallen! Als Beispiel suchen wird nach dem Wort "Datenbanken" in den Spalten "titel" und "inhalt" und geben zur Demonstration nochmals die Sprache Deutsch an:

```
mssql> SELECT * FROM dbnews WHERE CONTAINS( (inhalt,titel), ' "Datenbanken" ',
LANGUAGE 'german');
```

Wenn nur nach dem Beginn eines Wortes gesucht werden soll, schreibt man statt dem möglichen Ende des Wortes einen Asterisk. Dieser muss direkt vor dem abschließenden Hochkomma stehen und maskiert beliebige Wortzeichen. Gefunden werden alle Wörter oder Wortverbindungen, die mit dem Text beginnen, der vor dem Asterisk steht.
Im folgenden Beispiel werden alle Datensätze gesucht, die in der Spalte "inhalt" ein Wort enthalten, das mit "Ver" oder "ver" beginnt (Groß- und Kleinschreibung werden ignoriert):

```
mssql> SELECT * FROM dbnews WHERE CONTAINS(inhalt, ' "Ver*" ');
```

6.3.2. Verknüpfung mehrerer Begriffe

Mehrere Wörter oder Wortverbindungen können mit den logischen Operatoren **AND**, **OR**, **AND NOT** verknüpft werden. Die Kombination mit der Präfixsuche ist möglich.
Als Beispiel sollen alle Datensätze gefunden werden, die "Datenbank" oder "DBMS" in allen indizierten Spalten (bei uns "titel" und "inhalt") enthalten:

```
mssql> SELECT * FROM dbnews WHERE CONTAINS(*, ' "Datenbank" OR
"DBMS" ');
```

Außerdem lassen sich zwei oder mehr Begriffe mit **NEAR** oder **~** (Tilde) verbinden. Die damit verknüpften Begriffe müssen nahe beieinander stehen, wobei mit der Nähe der Übereinstimmungsgrad steigt.
Suchen wir nun nach dem Wort "Schulung" in der Nähe von "SQLJ":

```
mssql> SELECT * FROM dbnews WHERE CONTAINS(inhalt, ' "Schulung" NEAR
"SQLJ" ');
```

6.3.3. Suche nach Varianten

Die besondere Stärke des SQL Servers liegt darin, auf einfache Weise auch Ableitungen eines Wortes oder Sinnäquivalente zu finden. Dies kann erreicht werden, indem im Suchausdruck ein Term mit dem Schlüsselwort **FORMSOF** verwendet wird. Syntax:

```
FORMSOF (INFLECTIONAL | THESAURUS, "wort" [, "wort2", ...])
```

INFLECTIONAL wird verwendet, wenn Ableitungen eines Wortes (Formen, die durch Deklination eines Substantivs oder Konjugation eines Verbs entstehen) ebenfalls in der Suche berücksichtigt werden sollen. Bei Angabe von **THESAURUS** hingegen wird der interne Thesaurus herangezogen, um Synonyme zu finden.
Im folgenden Beispiel sollen die Verben "vergleichen" und "erscheinen" in allen Zeiten und Personen gefunden werden, also auch "erscheint", "vergleicht" und "verglich" etc.:

```
mssql> SELECT * FROM dbnews WHERE CONTAINS(inhalt, ' FORMSOF
(INFLECTIONAL, "vergleichen", "erscheinen") ');
```

Verwenden wir jetzt den Thesaurus, um alle Einträge mit dem Wort "Datenbank" oder Synonymen zu finden:

```
mssql> SELECT * FROM dbnews WHERE CONTAINS(inhalt, ' FORMSOF
(THESAURUS, "Datenbank") ');
```

Eine Möglichkeit, alle Varianten in einer Abfrage zu finden, also sowohl Flexionsformen als auch Synonyme, bietet das Prädikat **FREETEXT**. Syntax:

```
FREETEXT (spalte | (spaltenliste) | *, 'woerter' [, LANGUAGE 'sprache'])
```

Das erste und das dritte Argument entsprechen denen des Prädikats CONTAINS. Einen Unterschied gibt es beim zweiten Argument. Hier werden in einfachen Hochkommata nur ein Wort oder mehrere durch Leerzeichen getrennte Wörter erwartet. Die Wörter selbst werden nicht in doppelte Hochkommata eingefasst, wie dies bei CONTAINS der Fall ist. Auch andere Operatoren wie AND, OR usw. sind nicht erlaubt bzw. werden als sog. "Noise-Words" aussortiert. Die als zweites Argument übergebenen Wörter werden anhand ihrer Leerzeichen aufgetrennt und zu jedem Wort sowohl Flexionsformen gebildet als auch Synonyme anhand des Thesaurus gesucht. Die Suche mit FREETEXT ist aber insgesamt etwas ungenauer als die mit CONTAINS.
Als Beispiel wollen wir alle Datensätze finden, die "vergleichen" oder "Datenbank" oder irgendeine Variante dieser beiden Wörter enthalten. Die Angabe der Sprache dient auch hier nur der Verdeutlichung.

```
mssql> SELECT * FROM dbnews WHERE FREETEXT(inhalt, 'vergleichen Datenbank',
LANGUAGE 'german');
```

6.4. Suche mit Relevanzkalkulation

Die Prädikate CONTAINS und FREETEXT geben boolesche Werte (TRUE oder FALSE) zurück, können nur in einer WHERE- oder HAVING-Klausel verwendet werden und eignen sich daher nicht für ein Ranking nach Relevanz. Soll auch die Relevanz zu jedem Treffer berechnet werden, muss man auf zwei Funktionen, **CONTAINSTABLE** und **FREETEXTTABLE**, zurückgreifen. Ihre Syntax ist der von CONTAINS bzw. FREETEXT sehr ähnlich:

```
CONTAINSTABLE (tabelle, spalte | (spaltenliste) | *, 'suchausdruck'
[, LANGUAGE 'sprache'] [, top])
FREETEXTTABLE (tabelle, spalte | (spaltenliste) | *, 'woerter'
[, LANGUAGE 'sprache'] [, top])
```

Neu ist als erster Parameter der Name der Tabelle, die abgefragt werden soll, und als optionaler vierter bzw. dritter (wenn LANGUAGE nicht angegeben wird) Parameter eine Ganzzahl, die angibt, wieviele Treffer zurückgegeben werden sollen. Dabei werden die Treffer berücksichtigt, die die höchste Relevanz haben.
Das eigentlich Besondere an diesen beiden Funktionen ist aber ihr Rückgabewert: eine Tabelle mit den Spalten "key" und "rank". Die Spalte "rank" gibt die Relevanz mit einer Zahl zwischen 0 und 1000 an, wobei 1000 höchste Relevanz bedeutet. Wichtig ist, dass dies keine absoluten Werte sind, sondern in der Relation der einzelnen Treffer gesehen werden müssen. Die Spalte "key" enthält das Schlüsselattribut des Treffers.
Die beiden Funktionen können nun so verwendet werden: da sie ein Resultset (also eine Tabelle) zurückgeben, müssen sie in der FROM-Klausel angegeben werden. Über die Spalte "key" kann sie mit der Tabelle, die die eigentlichen Daten enthält gejoined werden.
Ein Beispiel soll dies verdeutlichen: wir wollen die Titel der drei Artikel aus "dbnews"

ausgeben, die bei der Suche nach dem Wort "Datenbank" die höchste Relevanz aufweisen.

```
mssql> SELECT dbnews.titel, ct.rank FROM dbnews, CONTAINSTABLE(dbnews, inhalt,
' "Datenbank" ', LANGUAGE 'german', 3) AS ct WHERE dbnews.id = ct.[key];
```

In der Funktion CONTAINSTABLE haben wir angegeben, dass in der Tabelle "dbnews" die Spalte "inhalt" durchsucht werden soll, und zwar nach dem Wort "Datenbank". Neben der Sprache Deutsch wird noch spezifiziert, dass nur die drei relevantesten Titel ausgegeben werden sollen. Dies erzeugt gleichzeitig eine absteigende Sortierung nach "rank".
Die von CONTAINSTABLE zurückgegebene Tabelle erhält den Aliasnamen ct, um sie ohne Probleme in der Join-Bedingung verwenden zu können. Gejoined wird die Spalte "key" der Tabelle "ct" mit der Spalte "id" (Primärschlüssel!) von "dbnews". Weil "key" eigentlich ein reserviertes Wort ist, muss es hier in eckige Klammern gesetzt werden.
Noch ein Beispiel mit der Funktion FREETEXTTABLE: hier sollen die Artikel gefunden werden, die das Wort "Datenbank" oder eine Variante davon enthalten. Das Ergebnis soll absteigend nach dem Rang sortiert werden. Die Konstruktion ist analog und bedarf nun keiner weiteren Erklärung:

```
mssql> SELECT dbnews.titel, ft.rank FROM dbnews, FREETEXTTABLE(dbnews, inhalt,
'Datenbank') AS ft WHERE dbnews.id = ft.[key] ORDER BY ft.rank DESC;
```

In der Onlinedokumentation werden die genauen Formeln genannt, nach denen CONTAINSTABLE und FULLTEXTABLE die Relevanz berechnen. In meiner Arbeit mit der Volltextsuche waren die Rangfolgen – ähnlich der Volltextsuche mit MySQL – leider nicht immer ganz nachvollziehbar.

6.5. Verwenden gewichteter Begriffe

Um einen oder mehrere Begriffe bei der Suche zu gewichten, muss das Schlüsselwort **ISABOUT** im Suchausdruck der Funktion CONTAINSTABLE verwenden. Syntax:

```
ISABOUT('begriff' [WEIGHT(gewichtung)] [, 'begriff2' [WEIGHT(gewichtung2)], ...])
```

Als "begriff" kann wiederum ein vollständiger Suchausdruck – also einfache Wörter, mehrere verknüpfte Wörter, Suche nach Präfixen oder nach Varianten – angegeben werden. Es können beliebig viele Begriffe durch Kommata getrennt spezifiziert werden. Die einzelnen Begriffe werden implizit mit OR verknüpft, d.h. es muss nur zu einem der Begriffe eine Übereinstimmung gefunden werden und nicht zu allen.
Die Angabe einer Gewichtung zu jedem Begriff ist optional und wird über das Schlüsselwort **WEIGHT** bewerkstelligt. Jeder Begriff kann mit einer Gleitkommazahl zwischen 0 und 1 gewichtet werden, wobei größere Werte einem Begriff größeres Gewicht beimessen. Die Schreibweise der Kommazahl unterliegt lokalen Servereinstellungen. Auf einem englischsprachigen System wird als Dezimaltrenner ein Punkt verwendet, auf einem deutschen hingegen ein Komma. Da das Komma normalerweise Elemente einer Liste trennt, sind dann noch zusätzlich doppelte Anführungszeichen notwendig. Wird WEIGHT weggelassen, erhält der Begriff automatisch die höchste Gewichtung (1).
Zum Abschluss ein etwas komplexeres Beispiel. Wir wollen die Artikel finden, die "Java", "Tutorial" oder ein Symonym dafür oder Wörter, die mit "Schulung" beginnen, enthalten. Dabei soll "Tutorial" eine höhere Gewichtung erhalten als "Schulung". Bei der Ausgabe

interessieren uns nur die Top-10.

```
mssql> SELECT dbnews.titel, ct.rank FROM dbnews,
CONTAINSTABLE(dbnews, inhalt, 'ISABOUT("Java", FORMSOF(THESAURUS,
"Tutorial") WEIGHT("0,7"), "Schulung*" WEIGHT("0,2"))', 10) AS ct
WHERE dbnews.id = ct.[key];
```

7. Auswahl der geeigneten Methode

Wir haben nun die wichtigsten Techniken zur effektiven Suche in SQL kennen gelernt. Letztlich stellt sich die Frage, welches für welche Art von Suche am Besten geeignet ist. Diese Frage ist allerdings sehr schwierig zu beantworten.
Generell gilt: je einfacher das verwendete Verfahren, desto performanter wird es sein. So ist, sofern es die Umstände zulassen, eine Suche mit LIKE derjenigen mit REGEXP vorzuziehen. Letztere sollte dann zum Einsatz kommen, wenn die Suchmuster sehr genau definiert werden sollen. Sie liefert dann brauchbarere und korrektere Ergebnisse, was den Verlust an Performance mehr als wettmachen kann.
Bei großen Datenbanken mit vielen Millionen Datensätzen führt hingegen kaum ein Weg an der Volltextsuche vorbei. Auf Anfragen mit LIKE oder gar mit regulären Ausdrücken kann man unter Umständen Minuten warten. Die Volltextsuche bleibt hingegen relativ performant und ist pflegeleichter als ein von Hand gewarteter Wortindex in einer Lookup-Tabelle oder ähnliche Scherze. Interessant ist auch die Möglichkeit, sich die Ergebnisse nach Relevanz sortieren zu lassen. Allerdings liefert die Volltextsuche nicht immer so genaue Ergebnisse wie eine Anfrage mit LIKE oder regulären Ausdrücken. Bei MySQL kommt hinzu, dass sie auf das MyISAM-Tabellenformat beschränkt ist.
Die beste Methode wird kaum ex ante zu ermitteln sein. Der Datenbankentwickler sollte sich aber über seine Entscheidung Rechenschaft ablegen. Letztendlich wird man ein Datenbanksystem ohnehin nicht (nur) nach den Suchmöglichkeiten auswählen, sondern nach anderen Kriterien wie Integration in die bestehende IT usw., und muss dann mit dem auskommen, was einem das Wahl-DBMS anbietet.

Christoph Bichlmeier
E-Mail: chris 'at' bichlmeier 'dot' info
Website: http://www.bichlmeier.info
erstellt am: 29.10.2004, Update: 5.5.2006