Mehr Ergebnisse:
Linguistische Funktionen und Ähnlichkeitssuche mit SQL

Carsten Czarski
ORACLE Deutschland B.V. & Co KG
München

Einleitung

Jede Suche in den Tabellen im Data Warehouse ist eine SQL-Abfrage mit einer WHERE-Klausel - das ist bekannt. Ebenfalls bekannt ist, dass immer exakt gesucht wird (sieht man von SQL LIKE einmal ab). Häufig muss jedoch nach diakritischen Zeichen (ü,ä,ö oder gar den osteuropäischen Varianten) gesucht werden - und hier beginnt für den Anwender das Problem: Das gesuchte Zeichen ist gar nicht auf der Tastatur - wie soll man also suchen?

Dieser Artikel und der Vortrag auf der DOAG BI-Konferenz zeigt, wie man dieses Problem mit den Bordmitteln der Datenbank, nämlich durch Einsatz linguistischer SQL-Funktionen lösen kann. Es wird vorgeführt, wie die linguistische Suche aktiviert und Tabellen mit den nötigen Indizes versehen werden, so dass man danach Case- und Akzent-Insensitiv suchen kann.

Der Anfang (ganz einfach): Case-Insenstive Suche mit SQL

Der erste Schritt zu einer fehlertoleranten Suche ist mit Sicherheit die Case-Insensitive Suche. In SQL ist eine solche Abfrage schnell formuliert.

```
select cust_first_name, cust_last_name, cust_city
from sh.customers
where upper(cust_last_name) = upper('baer');

CUST_FIRST_NAME      CUST_LAST_NAME       CUST_CITY
-------------------- -------------------- --------------
Bryan                Baer                 Walsall
Bryan                Baer                 Evinston
Bryan                Baer                 Noma
Bryan                Baer                 Montara
```

Diese Abfrage funktioniert sofort – und wenn sie aus einer Anwendung heraus aufgerufen wird, findet die Suche tatsächlich völlig Case-Insensitiv statt. Allerdings hat diese Abfrage Auswirkungen auf den Ausführungsplan ...

```
------------------------------------------------------------------
|   0 | SELECT STATEMENT  |           |       |       |   423 (100)|
|*  1 |  TABLE ACCESS FULL| CUSTOMERS |   555 | 13875 |   423   (1)|
------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter(UPPER("CUST_LAST_NAME")='BAER')
```

Der normale Index passt natürlich nicht, denn dieser ist ja Case-Sensitiv – bei größeren Tabellen dürfte eine solche Suche also nicht besonders performant sein. Es braucht einen *funktionsbasierten Index*, damit die Tabelleninhalte im Upper-Case in den Index geschrieben werden.

```
create index fix_upper_cust_lastname on customers(upper(cust_last_name));

Index wurde erstellt.
```

Man beachte den Funktionsaufruf von **UPPER** im **CREATE INDEX**-Kommando – neben **UPPER** können hier natürlich auch andere (und auch eigene) SQL-Funktionen verwendet werden – allerdings muss eine eigene Funktion als **DETERMINISTIC** deklariert werden. Mit dieser Deklaration "versichert" der Entwickler dem Optimizer, dass die Funktion für gleiche Eingabeparametern auch gleiche Ergebnisse zurückliefert. Nachdem der Index erstellt wurde (ggfs. noch die Statistiken aktualisieren), sieht der Ausführungsplan so aus, wie es sein soll.

```
----------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |                         |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| CUSTOMERS               |
|*  2 |   INDEX RANGE SCAN                  | FIX_UPPER_CUST_LASTNAME |
----------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access(UPPER("CUST_LAST_NAME")='BAER')
```

Übrigens: Im Gegensatz zu früheren Versionen ist das QUERY REWRITE-Privileg für einen funktionsbasierten Index nicht mehr nötig.

Zweiter Schritt: Diakritische Zeichen

Als nächstes geht es an die in Westeuropa durchaus üblichen diakritischen Zeichen – hierunter fallen alle Zeichen, auf denen Bögen, Punkte, Striche oder ähnliche Markierungen angebracht werden, um die Betonung oder die Aussprache zu verändern. In der deutschen Sprache wären das die Umlaute, aber bekanntlich sind solche Zeichen auch in anderen Sprachen präsent.

Diakritische Zeichen sind für eine Suche meist dann problematisch, wenn eine andere Sprache vorliegt. Zum einen befinden sich die nötigen Zeichen meist nicht auf der Tastatur des Anwenders, zum anderen fehlen oft die Sprachkenntnisse, um das *richtige* diakritische Zeichen auszuwählen – man denke nur an die Akzente im Französischen. Speziell für diese Fälle sind schon seit langem *linguistische Funktionen* in die Datenbank eingebaut – diese werden mit zwei Session-Parametern gesteuert.

NLS_COMP legt fest, ob linguistische Funktionen grundsätzlich für Zeichenvergleiche oder Sortierungen genutzt werden sollen. Standardmäßig ist dieser auf **BINARY** – mit **LINGUISTIC** aktiviert man die sprachspezifischen Funktionen.

```
alter session set NLS_COMP = 'LINGUISTIC'
```

NLS_SORT stellt die konkrete Sprache, sowie den Umgang mit diakritischen Zeichen und Groß- und Kleinschreibung ein. **GERMAN_AI** legt demnach fest, dass die Sortierung nach deutschen Standards erfolgen und dass diakritische Zeichen und die Groß- und Kleinschreibung bei Vergleichen ignoriert werden sollen. Lässt man das **_AI** weg, so erfolgt *nur* die Sortierung nach deutschen Standards.

Gerade bei internationalen Daten wäre das Suffix **_AI** schon sehr hilfreich; auf eine konkrete Sprache möchte man sich jedoch nicht festlegen. In diesem Fall stellt man **NLS_SORT** am besten auf **BINARY_AI** ein.

```
alter session set NLS_COMP = 'BINARY_AI'
```

Nach dem Einstellen der Session-Parameter lassen wir die Eingangsabfrage nochmals laufen – die SQL-Funktion UPPER lassen wir nun aber weg – denn **NLS_COMP=BINARY_AI** legt ja schon fest, dass Case-Insensitiv gesucht werden soll.

```
select cust_first_name, cust_last_name, cust_city
from sh.customers
where cust_last_name = 'bäèr';

CUST_FIRST_NAME       CUST_LAST_NAME        CUST_CITY
-------------------- -------------------- --------------
Bryan                 Baer                  Walsall
Bryan                 Baer                  Evinston
Bryan                 Baer                  Noma
Bryan                 Baer                  Montara
```

Die Abfrage ist erfolgreich - ein Blick auf den Ausführungsplan zeigt aber, dass wiederum kein Index mehr genutzt wird.

```
-------------------------------------------------------------------
|   0 | SELECT STATEMENT  |           |   555 | 26085 |   423   (1)|
|*  1 |  TABLE ACCESS FULL| CUSTOMERS |   555 | 26085 |   423   (1)|
-------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter(NLSSORT(UPPER("CUST_LAST_NAME"),'nls_sort=''BINARY_AI''')=
              HEXTORAW('6261657200'))
```

Das ist naheliegend, denn der eingangs erzeugte Index, der auf der SQL-Funktion **UPPER** basiert, passt jetzt nicht mehr. Es braucht einen anderen funktionsbasierten, einen *linguistischen* Index ...

```
create index fidx_nls_customers
on customers (nlssort(cust_last_name, 'NLS_SORT=BINARY_AI'));
```

Jetzt, wo der Index passt, wird er auch genutzt.

```
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT                       |                     |   555 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| CUSTOMERS            |   555 |
|*  2 |   INDEX RANGE SCAN                   | FIDX_NLS_CUSTOMERS |   222 |
---------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access(NLSSORT("CUST_LAST_NAME",'nls_sort=''BINARY_AI''')=HEXTORAW
       ('6261657200'))
```

Allein mit diesen Einstellungen sollte sich die Suche für den Endanwender schon wesentlich verbessern lassen: Egal, mit welchen diakritischen Zeichen ein Name oder Begriff geschrieben wird – für die Suche reicht die Grundform des Zeichens aus. Es geht aber noch weiter ...

Eigene SQL-Funktionen ...

Verwendet man eigene anstelle der eingebauten Funktionen, so lässt sich dieses Beispiel noch erweitern: Eine LIKE-Suche mit einem Wildcard *auf der linken Seite* soll *dennoch* indexunterstützt ablaufen.

Da eine LIKE-Abfrage mit **%SUCHWORT** niemals indexunterstützt ablaufen kann, muss man sich etwas überlegen: Man könnte die Strings umdrehen – als **CUST_LAST_NAME** wäre anstelle von **"Baer"** dann **"reaB"** gespeichert – und eine LIKE-Suche mit **%er** könnte dann als **CUST_LAST_NAME like {REVERSE}(%er)** umgeschrieben werden – und damit besteht wieder die Grundlage für einen funktionsbasieren Index.

Es braucht also eine PL/SQL-Funktion, die eine Zeichenkette "umdreht" und danach die Groß-/Kleinschreibung sowie die diakritischen Zeichen eliminiert. Letzteres müssen wir nun aber "manuell" machen, da die Kombination eines linguistischen Index mit einem eigenen funktionsbasierten Index nicht funktioniert. Als erstes stellen wir also die Session-Parameter **NLS_COMP** und **NLS_SORT** wieder zurück.

```
alter session set NLS_COMP = 'BINARY';
alter session set NLS_SORT = 'BINARY';
```

Nun werden die PL/SQL-Funktionen erstellt: **STRING_REVERSE** "dreht" eine Zeichenkette um (sie ist aber nur eine Hilfsfunktion für die beiden Folgenden).

```
create or replace function string_reverse (
  p_string in varchar2
) return varchar2 deterministic is
  v_revstring varchar2(32767) := '';
begin
  for i in reverse 1..length(p_string) loop
    v_revstring := v_revstring || substr(p_string, i, 1);
  end loop;
  return v_revstring;
end;
```

PREP_STRING eliminiert diakritische Zeichen und die Groß/Kleinschreibung.

```
create or replace function prep_string(
  p_string in varchar2
) return varchar2 deterministic is
begin
  return utl_raw.cast_to_varchar2(
    nlssort(p_string, 'nls_sort=binary_ai')
  );
end prep_string;
```

PREP_STRING_REV "dreht" eine Zeichenkette mit der Hilfsfunktion **STRING_REVERSE** um und eliminiert danach diakritische Zeichen und die Groß/Kleinschreibung.

```
create or replace function prep_string_rev(
  p_string in varchar2
) return varchar2 deterministic is
begin
  return utl_raw.cast_to_varchar2(
    nlssort(string_reverse(p_string), 'nls_sort=binary_ai')
  );
end prep_string_rev;
```

Nun werden mit diesen beiden SQL-Funktionen zwei funktionsbasierte Indizes erstellt – einer für Queries mit einer Wildcard rechts (Normalfall) und einer für Queries mit einer Wildcard links.

```
create index fidx_nls_customers
on customers (PREP_STRING(CUST_LAST_NAME));


create index fidx_nls_customers_rev
on customers (PREP_STRING_REV(CUST_LAST_NAME));
```

Nun können die Indizes genutzt werden. Allerdings muss man die Funktionen **PREP_STRING** und **PREP_STRING_REV** nun in die SQL-Abfragen einbauen. Zunächst die "normale" LIKE-Abfrage mit einer Wildcard rechts ..

```
select cust_first_name, cust_last_name, cust_city
from sh.customers
where prep_string("CUST_LAST_NAME") like prep_string('Bäé%');

-----------------------------------------------------------------
|   0 | SELECT STATEMENT                    |                    |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| CUSTOMERS          |
|*  2 |   INDEX RANGE SCAN                  | FIDX_NLS_CUSTOMERS |
-----------------------------------------------------------------
```

… danach die Abfrage mit einer Wildcard links:

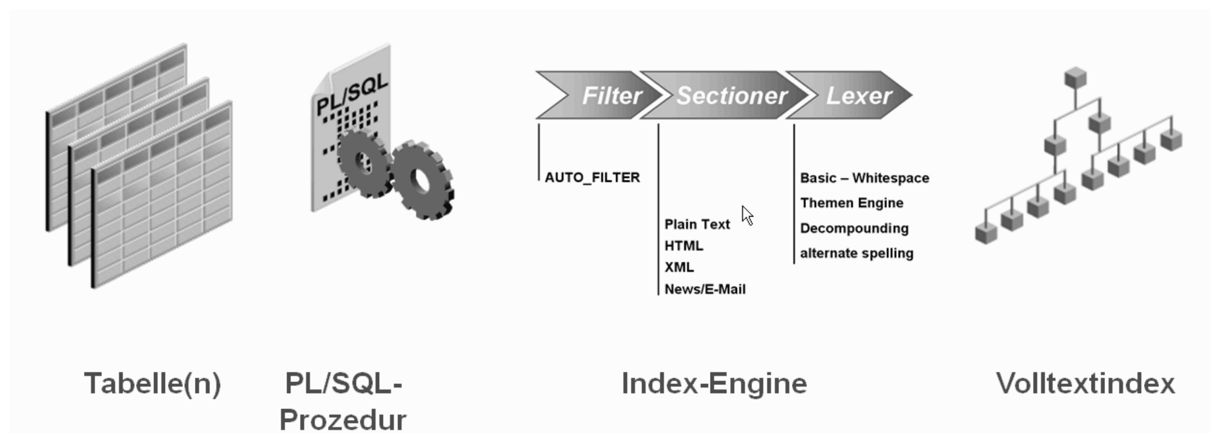
```
select cust_first_name, cust_last_name, cust_city
from sh.customers
where prep_string_rev("CUST_LAST_NAME") like prep_string_rev('%äér');

--------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |                        |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| CUSTOMERS              |
|*  2 |   INDEX RANGE SCAN                  | FIDX_NLS_CUSTOMERS_REV |
--------------------------------------------------------------------
```

Wie man sieht, nutzt der Optimizer in beiden Fällen einen funktionsbasierten Index – und in beiden Fällen sucht er sowohl Case- als auch Akzent-Insensitiv. Zwar muss man nun die SQL-Abfragen anpassen und das ganze ist damit nicht mehr transparent; allerdings kann die Unabhängigkeit von Session-Parametern auch wieder ein Vorteil sein ...

Allerdings hat auch diese Technik Grenzen – eine Abfrage mit einem Wildcard *sowohl links als auch rechts* wie **%ea%** kann mit normalen Mitteln nicht indexunterstützt ablaufen.

Ein Index für alles: Volltextsuche auf strukturierten Tabellen

Zum Abschluß sei vorgestellt, wie mit Hilfe der in der Datenbank enthaltenen Volltextengine Oracle TEXT eine sehr mächtige Suche auf Datenbeständen realisiert werden kann. Oracle TEXT ist prinzipiell zur Suche in unstrukturierten Daten (Dokumente) vorgesehen, wie die folgende Abbildung zeigt, kann es jedoch auch genutzt werden, um in strukturierten Daten in "normalen" Tabellen zu suchen.



In diesem Fall wird eine PL/SQL-Prozedur zwischen die zu indizierenden Tabellen und den Textindex gestellt. Anhand der Daten in den Tabellen generiert die PL/SQL Prozedur transiente Dokumente, die hernach von Oracle TEXT indiziert werden. Die Vorteile liegen auf der Hand:

Mit nur einem Index (Oracle TEXT Index) können mehrere Tabellenspalten in mehreren Tabellen indiziert werden. Alle linguistischen Funktionen von Oracle TEXT, vor allem die Fuzzy-Suche, stehen zur Verfügung.

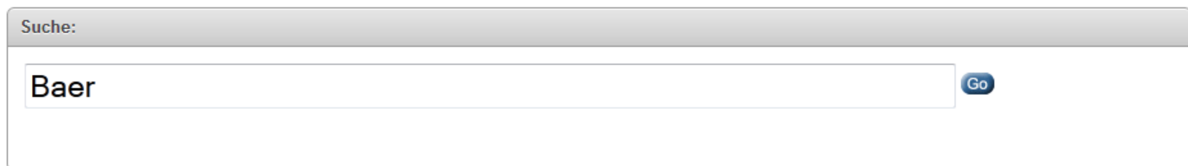
Die Beschreibung der exakten Vorgehensweise würde den Rahmen dieses Dokuments sprengen; unter **Weitere Informationen** findet sich ein Link auf ein Online-Howto [2], welches die Schritte detailliert beschreibt. Nachdem der Index erzeugt wurde, werden Abfragen mit der SQL-Funktion **CONTAINS** eingeleitet – innerhalb von **CONTAINS** lassen sich dann komplexe Suchausdrücke verwenden. Das folgende Beispiel sucht wieder in der Spalte **CUST_LAST_NAME**, nutzt nun aber die Fuzzy-Suche von Oracle TEXT.

```
select cust_first_name, cust_last_name, cust_city
from customers
where contains(cust_street_address, '?Beer WITHIN CUST_LAST_NAME') > 0
```

Das nächste Beispiel sucht das Wort **Baer** in *allen* indizierten Tabellenspalten.

```
select cust_first_name, cust_last_name, cust_city
from customers
where contains(cust_street_address, 'Baer') > 0
```

Mit diesen technischen Grundlagen kann man nun, auch im eigenen Unternehmen (und mit dem DWH), die beim Nutzer sicherlich beliebsteste Form der Suchmaske realisieren:



Weitere Informationen

Viele der hier vorgestellten Beispiele sind der deutschen APEX Community und dem Blog "SQL und PL/SQL in Oracle" entnommen. Darüber hinaus gibt es mittlerweile einige deutschsprachige Informationsquellen rund um die Oracle-Datenbank.

[1] Case- und Umlautsensitive Suche in Datenbeständen mit SQL LIKE
https://apex.oracle.com/pls/apex/GERMAN_COMMUNITIES.SHOW_TIPP?P_ID=281

[2] Suchen in strukturierten Daten mit Oracle TEXT
http://oracle-text-de.blogspot.co.uk/2012/03/nochmal-userdatastore-ein-umfassendes.html

[3] Ähnlichkeitssuche auf Deutsch: "Kölner Phonetik" mit SQL
https://apex.oracle.com/pls/apex/GERMAN_COMMUNITIES.SHOW_TIPP?P_ID=1502

[4] Blog "SQL und PL/SQL in Oracle"
http://sql-plsql-de.blogspot.com

[5] Blog "Oracle TEXT in deutscher Sprache"
http://oracle-text-de.blogspot.com

Kontaktadresse:

Carsten Czarski
ORACLE Deutschland B.V. & Co KG
Riesstr. 25, 80992 München

Telefon:	+49 (0) 89 1430 2116
E-Mail	carsten.czarski@oracle.com
Internet:	www.oracle.de
Blog des Autors	http://sql-plsql-de.blogspot.com
Twitter	@cczarski