Vom Nutzen der Best Practices

Jürgen Sieben, ConDeS GmbH & Co. KG

Immer wieder wird auf die Bedeutung der Einhaltung von Best Practices verwiesen. Die Ermunterungen sind wohlfeil und – seien wir ehrlich – ermüdend. Natürlich verstehen wir alle, warum Best Practices eingeführt wurden, wir verstehen auch deren Bedeutung für die Teamarbeit und dass sie eingehalten werden müssen. Aber sie erscheinen manchmal unnötig aufwendig. Diese Folge der Kolumne beleuchtet den Vorteil einer konkreten Best Practice, allerdings kann der Autor, um die Spannung nicht gleich herauszunehmen, noch nicht verraten, um welche.

Noch steht nicht die Empfehlung im Blickpunkt, sondern ein Stück Code, das ein sehr merkwürdiges Verhalten zeigte. Der Code lief monatelang einwandfrei und auf einmal warf er einen eher suspekt wirkenden Fehler. *Listing 1* zeigt den (im Grundsatz trivialen) Code.

Eine Prozedur vereinbart einen Cursor gegen eine Remote-Datenbank und iteriert über die Ergebniszeilen, um mit diesen Zeilen irgendeine Arbeit durchzuführen. Sieht harmlos aus, oder? Der Code funktionierte, wie gesagt, monatelang auch problemlos, doch dann erschien eine Fehlermeldung, die sich anschließend immer wieder zeigte *(siehe Listing 2)*.

```
SQL> create or replace procedure do_something
   2 as
   3   cursor my_cur is
   4     select *
   5       from some_view@dblink v
   6       join some_table t
   7         on v.id = t.id;
   8 begin
   9   for rec in my_cur loop
  10     -- do_something;
  11     null;
  12   end loop;
  13 end;
  14 /
```

Listing 1: Test-Prozedur

Nanu? Zugegeben: Dieser Fehler gehört definitiv in die Kategorie „Wer weiß denn so was?“, er ist aber auch nicht aus der Welt und kann so oder sehr ähnlich immer wieder mal auftreten. Zunächst einmal ist der Code, bis auf den Datenbank-Link in der Deklaration des Cursors, ganz normal. Da andererseits das Öffnen des Cursors bereits den Fehler wirft, muss das Problem irgendwo in dieser Ecke liegen. Auch die Fehlermeldung „Bulk Bind“ weist hierauf hin, denn seit Version 11g der Datenbank wird das Öffnen eines Cursors in einer „cursor for“-Schleife durch eine Bulk-Operation optimiert (wofür der ungewollte Fehler also gleich einen schönen Beleg liefert).

```
ORA-06502: PL/SQL: numerischer oder Wertefehler: Bulk Bind: Truncated Bind
ORA-06512: in "DOAG.DO_SOMETHING", Zeile 9
```

Listing 2

Erste Anlaufstelle: die Oracle Dokumentation. Dort erfährt man, dass dieser Fehler auftritt, wenn ein „type mismatch“ vorliegt, also zum Beispiel innerhalb einer Bulk-Operation eine Variable mit einer Länge von 10 mit 11 Zeichen beladen wird. Das kann hier jedoch ganz offensichtlich nicht das Problem sein, denn der Record, der eine Zeile des Cursors aufnimmt, wird implizit als „my_cur%ROWTYPE“ definiert, entspricht also exakt der Definition des Cursors. Woher die Deklaration des Typs kommt, ist auch schnell geklärt: aus dem Data Dictionary nämlich, aus den Tabellen-Eigenschaften der zugrunde liegenden Tabellen.

Um den Fehler einzugrenzen, sollte man zunächst ermitteln, ob es irgendwelche Probleme mit den Daten des Cursors gibt. Allerdings funktioniert die „select“-Anweisung des Cursors, für sich ausgeführt, ohne Probleme. Nur im Zusammenhang mit der Prozedur wird konstant der oben gezeigte Fehler geworfen. Die nächste Aktion, um den Fehler einzugrenzen, besteht darin, die Spalte zu finden, die den Fehler wirft. Dies ist einfach, man muss nur die Cursor-Deklaration so ändern, dass der Fehler kontrolliert geworfen oder verhindert werden kann. Im Beispiel fand sich schließlich eine Zeile als Übeltäter, die einen Kommentar zu einem Dokument enthielt.

Doch was war so speziell an dieser Spalte? Vielleicht liegt es an der unüblichen Verwendung von „LONG“-Datentypen oder ähnlichen, doch in diesem Datenmodell wird „LONG“ ausschließlich noch von Oracle selbst verwendet, nicht aber von den Datenbank-Modellierern. Die Spalte war schlicht vom Typ „varchar2(40 byte)“. Warum Byte? Es mag zunächst nicht so scheinen, aber diese Frage bringt uns auf die richtige Fährte. Alternativ hätte dort „varchar2(40 char)“ stehen können.

```
-- In entfernter Datenbank
SQL> create table encoding_test(
   2   id number,
   3   text varchar2(10 byte),
   4   constraint pk_encoding_test primary key(id)
   5 ) organization index;
SQL> table ENCODING_TEST erstellt.

SQL> insert into encoding_test(id, text)
   2 select 1, 'Zehn Ohne:' from dual union all
   3 select 2, 'Zehn mit Ö' from dual;
2 Zeilen eingefügt.

SQL> commit;
festgeschrieben.
```

Listing 3

```
SQL> -- Test fuer Zeile 1
SQL> call encoding_test(1);

encoding_test 1) erfolgreich.

SQL> -- Test fuer Zeile 2
SQL> call encoding_test(2);
Fehler beim Start in Zeile : 16 in Befehl - call encoding_test(2)

Fehlerbericht -
SQL-Fehler: ORA-06502: PL/SQL: numerischer oder Wertefehler: #
                       Bulk Bind: Truncated Bind
ORA-06512: in "DOAG.ENCODING_TEST", Zeile 9
06502. 00000 -  "PL/SQL: numeric or value error%s"
```

Listing 4

Unterschied zwischen Byte- und Char-Semantik

Bei Zeichensatz-Kodierungen mit variabler Länge pro Zeichen (hier ist eigentlich immer UTF-8 gemeint) besteht durchaus ein Unterschied zwischen der Angabe „Byte“ oder „Char“, bei Zeichensatz-Kodierungen aus dem ISO-8859-Umfeld jedoch nicht, weil dort jeder Buchstabe genau ein Byte lang ist. Der Unterschied besteht darin, dass im Fall einer UTF-8-basierten Datenbank die Datenbank für eine Spalte vom Typ „varchar2(40 char)“ 160 Byte Speicherplatz vereinbart, weil ein Unicode-Zeichen maximal vier Byte lang sein kann. Im Fall der Angabe „varchar2(40 byte)“ ist es in einer Unicode-Datenbank daher möglich, dass lediglich zehn Buchstaben in diese Spalte passen, zum Beispiel, wenn es sich um chinesische Zeichen handelt.

Wenn Sie bei der Anlage der Tabelle nicht definieren, ob man „40 Byte“ oder „40 Char“ speichern möchte, wird die Entscheidung von einem Initialisierungs-Parameter mit dem Namen „NLS_LENGTH_SEMANTICS“ abhängig gemacht. Der wiederum steht standardmäßig auf dem Wert „BYTE“, wenn er nicht durch den Administrator geändert wurde. Da man nicht gern von Standard-Werten abhängig ist, hat es sich daher als Best Practice etabliert, die Angabe, ob „Byte“- oder „Char“-Semantik zur Speicherung verwendet werden soll, bei der Tabellen-Deklaration explizit anzugeben. Dann ist beides auch gemischt möglich, in jedem Fall ist es jedoch eindeutig geregelt.

Woher der Fehler kommt

Nun keimt der Verdacht für den Fehler: Wir haben einen Datenbank-Link, der auf eine fremde Datenbank zeigt. Lokal ist unsere Datenbank in UTF-8 kodiert, aber ist das auch so für die fremde Datenbank? Nein, die fremde Datenbank ist in WIN-1252 kodiert, was kompatibel zu ISO-8859-1 ist und also eine Ein-Byte-Zeichensatzkodierung darstellt. Nun wird das Problem klar:

- In der entfernten Datenbank war als Spalten-Typ „varchar2(40 byte)“ eingetragen
- Nach langer Zeit war zum ersten Mal ein langer Kommentar mit 40 Zeichen eingetragen
- In diesem Kommentar war auch noch ein Umlaut enthalten (der zwei Byte Speicherplatz benötigt)

Diese Zeile der Tabelle benötigt in der entfernten Datenbank 40 Byte Speicherplatz, in der lokalen jedoch 41 Byte. Da andererseits der Cursor die Spaltenbreite aus dem Data Dictionary der entfernten Datenbank entnommen hat, hat das lokale Kommentar-Attribut des Records die Breite 40 Byte, und dort passt die übernommene Zeichenkette nicht hinein: „Exception -06052“.

Lösungsansätze

Wie gesagt: Wer weiß denn so etwas? Bevor wir nun sozusagen zur Moral der Geschichte kommen, hier zunächst einige Lösungsansätze. Auf der entfernten Datenbank in 1-Byte-Kodierung existiert eine einfache Tabelle, um das Problem zu demonstrieren *(siehe Listing 3)*. Die Prozedur in der Multibyte-kodierten, lokalen Datenbank aus *Listing 1* wirft nun für Parameter „1“ keinen, für Parameter „2“ jedoch den Fehler „-06052“ *(siehe Listing 4)*.

Um dieses Problem zu lösen, gibt es mehrere Möglichkeiten. Zum einen wäre es möglich gewesen, die Prozedur anzuweisen, die Daten in die neue Zeichensatz-Kodierung zu konvertieren. Erfolgt dies innerhalb der „select“-Anweisung des Cursors, wird damit automatisch

```
create or replace procedure encoding_test(
  p_id in number)
as
 cursor encoding_cur(p_id in number) is
   select id, convert(text, 'AL32UTF8')
     from encoding_test@kis
    where id = p_id;
begin
  for r in encoding_cur(p_id) loop
    null;
  end loop;
end;
/
```

Listing 5: Geänderte Prozedur

```
SQL> -- Erfolgreicher Test fuer Zeile 2
SQL> call encoding_test(2);
encoding_test 2) erfolgreich.
```

Listing 6

auch die Breite des Record-Attributs angepasst. Dies ermöglicht die Funktion „CONVERT“, aufgerufen wie im Beispiel in *Listing 5*.

Als Name der Zeichensatz-Kodierung, in die übersetzt werden soll, muss der interne Oracle-Bezeichner für die Zeichensatz-Kodierungen gewählt werden, hier „AL32UTF8“. Der Fehler wird nun nicht mehr ausgelöst, die Prozedur funktioniert *(siehe Listing 6)*. Natürlich wäre es auch möglich, einen ganz einfachen, anderen Weg zu gehen: Wenn man die Spalte der entfernten Tabelle als „varchar2(40 char)“ definiert, ist der Umweg über die „CONVERT“-Funktion nicht mehr erforderlich, der Code funktioniert einfach. Natürlich wurde wieder die Testprozedur aus *Listing 1* zum Testen verwendet *(siehe Listing 7)*.

Der Grund ist, dass nun die lokale Datenbank aus der Angabe, zehn Zeichen speichern zu müssen, ableiten kann, dass sie dafür bis zu 40 Byte benötigen wird. Durch die (fehlerhafte) explizite Festlegung auf zehn Byte im Quellsystem hatte es sozusagen die Zusicherung gegeben, dass nicht mehr als diese Datenmenge in Byte übermittelt werden. Eine Zusicherung, die die entfernte Datenbank aus ihrer Sicht ja auch einhält. Durch die Einhaltung der Best Practice kann die lokale Datenbank das Problem nun auflösen.

Fazit

Es gibt viele Beispiele für eine eher entmutigende Tatsache: Die möglichen Fehlerursachen sind vielfältig und zum Teil nur schwer zu testen. Natürlich hätten entsprechend sorgfältig erstellte Testdaten dieses Problem aufdecken können, aber eigentlich nur, wenn man den Fehler bereits kennt. Wer würde sonst daran denken, in Testdaten die maximale Breite einer Spalte nicht nur auszunutzen, sondern auch noch Umlaute zu integrieren? Natürlich ist es auch immer ein guter Rat, sich weiter und weiter mit der Datenbank auseinanderzusetzen, um besser darin zu werden, die Arbeitsweise sowie mögliche Fehlerursachen frühzeitig zu erkennen. Doch eigentlich sind dies alles wohlfeile Ermahnungen.

Wirklich hilfreich sind allerdings Best Practices. Denn wieder einmal zeigt sich, dass hier ein Problem auftaucht, das nur durch einen Verstoß gegen Best Practices möglich wurde: Man verheimlicht der Datenbank, welche Daten man zu speichern gedenkt. Wer zehn Zeichen speichern können möchte, muss dies bei der Anlage der Tabelle auch hinterlegen.

Natürlich ist dies nur ein Beispiel für den Nutzen von Best Practices, aber es zeigt, dass aus diesen Vorgaben Vorteile gezogen werden können, die nicht offensichtlich sind, den Code jedoch robuster machen: Das Beispiel verhindert Fehler, von deren Existenz man möglicherweise nicht einmal wusste. Man kann es auch so ausdrücken: Best Practices sind nicht zuletzt kristallisierte Erfahrung …

```
SQL> -- Zweite Moeglichkeit: Implementierung einer Best Practice
SQL> drop table encoding_test;

table ENCODING_TEST gelöscht.

SQL> create table encoding_test(
   2   id number,
   3   text varchar2(10 char),
   4   constraint pk_encoding_test primary key(id)
   5) organization index;

table ENCODING_TEST erstellt.

SQL> insert into encoding_test(id, text)
   2 select 1, 'Zehn Ohne:' from dual union all
   3 select 2, 'Zehn mit Ö' from dual;
2 Zeilen eingefügt.

SQL> commit;
festgeschrieben.

SQL> -- Erfolgreicher Test fuer Zeile 2
SQL> call encoding_test(2);
encoding_test 2) erfolgreich.
```

Listing 7

Jürgen Sieben
j.sieben@condes.de