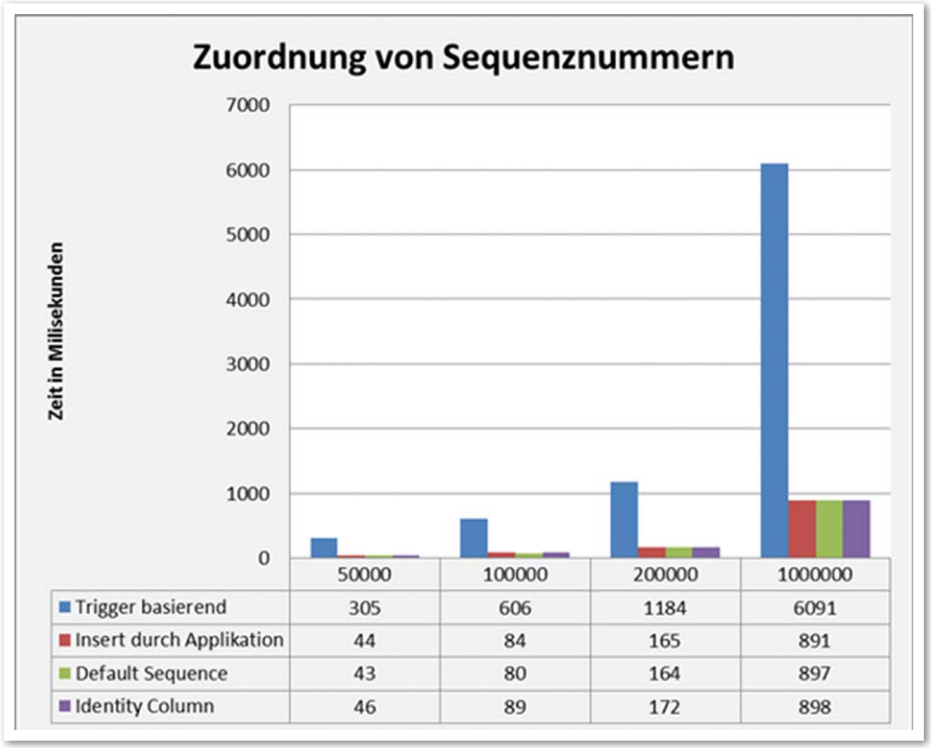


	50000	100000	200000	1000000
Trigger basierend	305	606	1184	6091
Insert durch Applikation	44	84	165	891
Default Sequence	43	80	164	897
Identity Column	46	89	172	898

Abbildung 1: Gegenüberstellung der Performance

- Erweiterungen im Bereich der Partitionierung
- „WITH"-Klausel mit PL/SQL-Support
- Integration von PL/SQL-Datentypen in dynamisches SQL
- In-Memory-Funktionalität

Roger Troller
roger.troller@trivadis.com

Umgebungswechsel vermeiden

Jürgen Sieben, ConDeS GmbH & Co. KG

Performance-Tuning ist ein weites Feld mit vielen Facetten. Doch es gibt Best Practices als Grundlage für weitere Tuning-Maßnahmen. Oft werden diese nicht ausreichend berücksichtigt, sondern die Hoffnung auf eher exotische Optimierungen gelegt. Eine besondere Rolle fällt in diesem Zusammenhang den Umgebungswechseln zwischen SQL und PL/SQL zu. Sie sind – meist unbemerkt – für erhebliche Einbußen der Performance verantwortlich. Dieser Artikel zeigt die grundlegenden Mechanismen auf und erläutert, wie ungewollte Umgebungswechsel erkannt und vermieden werden können.

Seit vielen Versionen der Datenbank enthält die Programmiersprache PL/SQL keine eigene SQL-Implementierung mehr, sondern ruft die SQL-Implementierung der Datenbank auf, wenn entsprechende Anweisungen im Code auftreten. Wechselt die Kontrolle von PL/SQL zu SQL oder umgekehrt, entsteht ein Umgebungswechsel, der aufgrund der Einrichtung der Umgebungsvariablen etc. einen erheblichen Aufwand für die Datenbank darstellt.

Diese Umgebungswechsel haben enormen Einfluss auf die Gesamtleistung der Anwendung und sollten daher so selten wie möglich auftreten. Vor der Vermeidung steht jedoch das Erkennen von Umgebungswechseln und das kann sehr einfach, aber auch sehr vertrackt sein. Zunächst die einfachen Beispiele. Beim Umgebungswechsel von SQL nach PL/SQL sind es vor allem zwei Szenarien, die immer wieder anzutreffen sind:

- Aufruf von PL/SQL-Funktionen aus SQL
- Zeilen-Trigger auf Tabellen

Listing 1 zeigt ein einfaches Beispiel für einen Funktionsaufruf aus SQL. Stellvertretend für alle PL/SQL-Funktionen wird hier

```
select *
  from emp
  where deptno = v('P10_DEPTNO');
```

Listing 1

```
select *
  from emp
  where deptno = (select v('P10_DEPTNO') from dual);
```

Listing 2

```
  with params as(
      select v('P10_DEPTNO') deptno,
          v(('P10_JOB') job
      from dual)
select e.*
  from emp e natural join params p;
```

Listing 3

```
create or replace trigger trg_emp_briu
before insert or update on emp
for each row
declare
  l_now date := sysdate;
begin
  :new.empno := coalesce(:new.empno, emp_seq.nextval);
  :new.ename := upper(:new.ename);
  if updating then
      :new.empno := :old.empno;
  end if;
end;
/
```

Listing 4

die Funktion „v“ gezeigt. Sie ist im Schema „APEX_nnnnn“ definiert und bietet dem Entwickler Zugriff auf Informationen, die sich im Session State von Apex befinden. Der Parameter „P10_DEPTNO“ bezeichnet ein Element auf einer Apex-Anwendungsseite, er ist eine Konstante und die Funktion mit Sicherheit für die Dauer der Abfrage deterministisch, das heißt, sie wird für den gleichen Eingangsparameter im Kontext dieser „select“-Abfrage das gleiche Ergebnis liefern. Sollte SQL nicht so clever sein, die Funktion nur einmal aufzurufen anstatt für jede Zeile?

Die Antwort lautet: Nein. Die Funktion selbst ist nicht deterministisch: Ein anderer Benutzer mit anderem Session-Identifier wird einen anderen Wert für das Element „P10_DEPTNO“ erhalten können, und auch zwischen zwei Abfragen innerhalb der gleichen Benutzersession können die Ergebnisse für den Funktionsaufruf mit diesem Parameter differieren. Die Datenbank weiß also nicht, welches Ergebnis die Funktion liefert, sondern muss es aktuell erfragen. Daher hätte noch nicht einmal PL/SQL die Möglichkeit, das Ergebnis im Cache zwischenzuspeichern.

Viel wichtiger ist jedoch, dass SQL nicht weiß, welche Werte die Funktion zurückliefern wird. Daher muss für jede Zeile die Funktion erneut aufgerufen und von PL/SQL neu berechnet werden. Dies ist besonders fatal, weil die Funktion in der „where“-Klausel der Abfrage verwendet wird, denn sie wird in jedem Fall für jede Zeile der Tabelle aufgerufen: Beinhaltet die Tabelle eine Million Zeilen, von denen nach der Filterung nur noch hundert übrigbleiben, hat die Anfrage dennoch eine Million Umgebungswechsel durchführen müssen. Was also tun? Die Lösung liegt darin, den Aufruf der Funktion in einer skalaren Unterabfrage zu schachteln *(siehe Listing 2)*.

Das sieht auf den ersten Blick zwar komisch aus, hat aber eine Reihe von Vorteilen:

- Weil eine skalare Unterabfrage von SQL wie eine Konstante behandelt wird, erfolgt die Berechnung nur einmal für jeden unterschiedlichen Parameter der Funktion, in unserem Fall also genau einmal.
- Es treten nach der ersten Berechnung keine weiteren Umgebungswechsel mehr auf.
- Diese Optimierung funktioniert mit deterministischen Funktionen ebenso wie mit nicht deterministischen Funktionen.

Gerade der letzte Punkt ist interessant: Die Denkweise ist, dass SQL eine Unterabfrage lesekonsistent zum Zeitpunkt der Abfrage einmal pro unterschiedlichem Parameter beantwortet und das Ergebnis der Abfrage nachfolgend als Konstante betrachtet. Daher ist es nicht erforderlich, dass die Funktion „v“ deterministisch ist; wir verwenden das Ergebnis, das zum Zeitpunkt der Abfrage geliefert wurde.

Da wir mit diesem Aufruf nun eine Million Umgebungswechsel einsparen, sinkt die Ausführungszeit der Abfrage auf die gleichen Werte (nahezu) wie bei Verwendung einer Konstanten in der Abfrage, sie wird förmlich pulverisiert und sinkt unter den sinnvoll messbaren Bereich. Ein erweiterter Lösungsansatz besteht darin, mehrere Funktionsaufrufe in einer faktorisierten Unterabfrage der eigentlichen Abfrage voranzustellen *(siehe Listing 3)*.

Das ist auch einmal eine sinnvolle Anwendung des Natural Join, der ja Spalten gleichen Namens zweier Tabellen in eine automatische „Inner Join“-Beziehung einbezieht. Da man die „PARAMS“-Abfrage über Aliase im Griff hat, kann man sich eine explizite Join-Klausel ersparen. Doch ist dieser Aufwand nötig? Haben wir nicht von „RESULT_CACHE“, „QUERY_RESULT_CACHE“ (für SQL-Abfragen) und in Version 12*c* vom Pragma „UDF“ (User Defined Function) gelesen, das speziell für Funktionen da ist, die in SQL verwendet werden sollen? Was ist mit dem Pragma „DETERMINISTIC“?

Alle diese Lösungen beziehen sich auf die jeweilige Umgebung, also auf PL/SQL oder auf SQL, und können dort unnötige Neuberechnungen ersparen. Viele dieser Optionen sind zudem an die Enterprise Edition gebunden und haben keine Auswirkung auf SE oder XE. Zudem kann keine dieser Optimierungen unnötige Umgebungswechsel minimieren; das kann nur die Kapselung in einer Unterabfrage, die zudem auch noch in jeder Datenbank-Version und -Edition funktioniert.

Grund genug für die erste Best Practice: „Funktionsaufrufe in SQL gehören in eine skalare Unterabfrage. Jedenfalls dann, wenn Sie nicht für jede Zeile ein unterschiedliches Ergebnis benötigen (zum Beispiel „dbms_random"). Diese Funktionsaufrufe müssen pro Zeile gerechnet werden und dürfen daher nicht in eine skalare Unterabfrage."

Zeilen-Trigger

Ein beliebtes Thema und für viele die Einstiegsdroge in die Programmierung von PL/SQL sind Trigger auf Tabellen, um zum Beispiel Primärschlüssel-Spalten zu berechnen, Großschreibung von Namen zu garantieren oder sonstige Datenprüfungen vorzunehmen. *Listing 4* zeigt einen einfachen Zeilen-Trigger auf die Tabelle „EMP".

Zeilen-Trigger, die also für jede betroffene Zeile einer DML-Anweisung einmal ausgeführt werden („Klausel for each row"), haben für jede Zeile einen Umgebungswechsel zwischen SQL und PL/SQL zur Folge: Der Trigger-Körper ist ein anonymer PL/SQL-Block. Wann der Trigger ausgelöst wird und auf welche Tabelle er sich bezieht, ist in SQL definiert. Die Logik wird aber in PL/SQL implementiert und die Schnittstelle zwischen beiden ist der Aufruf eines anonymen PL/SQL-Blocks für jede Zeile, für die der Trigger ausgelöst wird.

Was tun wir dagegen? Die Antwort ist einfach, die Umsetzung nicht: Den Zeilen-Trigger weglassen. Ab Version 12*c* gibt es mit der „column identity"-Klausel (endlich) einen Weg, Trigger zur Generierung neuer Primärschlüssel zu ersetzen. Das muss dann auch konsequent gemacht werden, wenn man auf Version 12*c* umsteigt und keine Rückwärts-Kompatibilität benötigt.

Wie ersetzt man den Rest? Wer die Zeilen-Trigger weglässt, verlagert die Logik in ein PL/SQL-Package. Toll, dann haben wir halt die Umgebungswechsel umgekehrt. Richtig und auch wieder nicht, denn wie der entsprechende Abschnitt zeigt, bietet ein Package immer die Möglichkeit, solche Arbeiten in einer Mengenverarbeitungsroutine in PL/SQL zunächst aufzubereiten und dann mit wenigen Umgebungswechseln als Menge an SQL zu übergeben.

Doch ein Vorteil des Triggers ist, dass an ihm keiner vorbeikommt. Ist jederzeit bekannt, welche Teile des extern programmierten Anwendungscodes auf die Tabelle schreiben? Falls nicht, aber dennoch sichergestellt werden muss, dass immer ein Primär-Schlüssel berechnet und der Name in Großbuchstaben gespeichert wird, scheint man um den Trigger nicht herumzukommen. Das könnte man nur verhindern, wenn nicht jeder beliebig auf die Tabelle schreiben, sondern nur ein bestimmtes Package nutzen darf.

Hier beginnt das Problem: Wenn man sagt: „Danke, das war's dann mit diesem Konzept, das kriegen wir nie durch", lässt sich das zwar gut nachvollziehen, es ist aber auch ein Indiz für eine zu starke Kopplung zwischen Datenbank und Anwendungscode. Sagen wir so: Es ist eine architektonische Entscheidung. Wenn man die nicht treffen möchte oder kann, sind die durch die Trigger verursachten Umgebungswechsel eine unausweichliche Folge und die damit verbundenen Performance-Probleme nur äußerst schwer in den Griff zu bekommen.

Fassen wir also zusammen: „Zeilentrigger sind mit einer performanten Anwendung nicht in Einklang zu bringen, wenn sie in Bewegungs-Tabellen verwendet werden. Je größer die Transaktionslast auf einer Tabelle, desto stärker die Begründung, auf Zeilentrigger zu verzichten und alternative Programmiermodelle zu etablieren."

Umgebungswechsel von PL/SQL nach SQL

Auch umgekehrt sind Umgebungswechsel von PL/SQL nach SQL möglich. Dazu ein Beispiel mit einem Aufruf einer DML-An-

```
create or replace procedure sql_
performance_test_A
as
begin
   for i in 1..10000 loop
       insert into test_table
values(i);
   end loop;
   commit;
end sql_performance_test_A;
/
```

Listing 5

```
create or replace procedure sql_
performance_test_B
as
   type value_table_type is table
of pls_integer index by binary_
integer;
   l_value_table value_table_
type;
   l_iterations integer := 10000;
begin
   for i in 1..l_iterations loop
       l_value_table(i) := i;
   end loop;
   forall idx in 1 .. l_itera-
tions
   insert into test_table
values(l_value_table(idx));
   commit;
end sql_performance_test_B;
/
```

Listing 6

```
scott@condes> begin
   2       Compare_Implementation('A', 'B');
   3   end;
   4   /
Run1 ran in 307 hsecs
Run2 ran in 4 hsecs
run 1 ran in 7675% of the time

Run1 latches total versus runs -- difference and pct
Run1          Run2           Diff    Pct
120,409       6,072      -114,337   1,983.02%
PL/SQL procedure successfully completed.
```

Listing 7

weisung in einer Loop. Einfach zu sehen ist dies in der Prozedur, die 10.000 Zeilen in eine Tabelle schreibt *(siehe Listing 5)*.

Diese Prozedur macht für jeden Schleifen-Durchlauf einen Umgebungswechsel. Ärgerlich daran ist, dass die elegante und einfache Art der Programmierung eigentlich suggeriert, dass man alles richtig gemacht hat. Um die Aufgabe ohne unnötige Umgebungswechsel zu lösen, wird die mengenorientierte Programmierung eingesetzt, wie im folgenden Beispiel *(siehe Listing 6)*.

Zunächst wird die PL/SQL-Tabelle „value_table" mit Daten geladen, was einen erhöhten Speicherverbrauch (schließlich müssen statt einer alle 10.000 Zeilen im Arbeitsspeicher gehalten werden), nicht aber Umgebungswechsel zur Folge hat. Anschließend wird in einer „FORALL"-Anweisung die lokal gefüllte Variable an SQL übergeben und dort in einer einzigen SQL-Anweisung in die Tabelle eingefügt.

Der Autor hat die beiden Prozeduren gegeneinander ins Rennen geschickt. Er hat dabei nicht nur die Zeit der Ausführung, sondern auch die Locks und Latches, die beide Prozeduren innerhalb der Datenbank allokieren, mit Tom Kytes „RUN_STATS"-Skript gemessen. *Listing 7* zeigt das beeindruckende Ergebnis.

Der Vergleich zeigt, dass die mengenorientierte Arbeitsweise um den Faktor 75 schneller ist, die Anzahl der Locks und Latches sinkt um den Faktor 20 (und damit steigt die Skalierbarkeit der Anwendung um diesen Faktor). Nachteil ist, wie gesagt, der erhöhte Speicherverbrauch dieser Lösung. Um diesen zu reduzieren, kann man die Anzahl der Zeilen steuern, die im Bulk an SQL übergeben werden, allerdings erfolgen nun mehr Umgebungswechsel.

Wo ist hier also die richtige Balance? Hier gibt es gute Nachrichten, denn bereits kleine Zeilenmengen haben erheblichen Einfluss auf die Geschwindigkeit. *Abbildung 1* zeigt den obigen Vergleich für die Bulk-Größen von 1 bis 128 Zeilen.

Die Ausführungszeit sinkt anfangs rapide (Angaben in hsec) und reduziert sich ab etwa hundert Zeilen nicht mehr relevant weiter. Daher ist diese Zeilenzahl ein guter Start für eigene Versuche. *Listing 8* zeigt, wie man die Bulk-Größe kontrolliert.

Auch hier die Zusammenfassung als Best Practice: „Datenbanken werden

```
create or replace procedure sql_bulk_test(
   p_bulk_size in integer)
as
   type value_table_type is table of pls_integer index by binary_inte-
ger;
   l_value_table value_table_type;
   c_iterations constant integer := 100000;
   l_idx integer := 1;
   l_value integer;
   l_time integer;
begin
   l_time := dbms_utility.get_time;
   while l_idx <= ceil(c_iterations / p_bulk_size) + 1 loop
       -- Daten in PL/SQL-Tabelle laden
       l_value_table.delete;
       for i in 1..p_bulk_size loop
           l_value := ((l_idx - 1) * p_bulk_size) + i;
           if l_value <= c_iterations then
           l_value_table(i) := l_value;
       else
       exit;
       end if;
       end loop;
       -- BULK-Insert in die Datenbank
       forall idx in 1 .. l_value_table.count
           insert /*+ append */ into test_table values(l_value_
table(idx));
       l_idx := l_idx + 1;
   end loop;
   l_time := dbms_utility.get_time - l_time;
   dbms_output.put_line('Bulk size: ' || p_bulk_size || ', Time: ' ||
l_time || 'hsec');
   execute immediate 'truncate table test_table';
end sql_bulk_test;
/
```

Listing 8

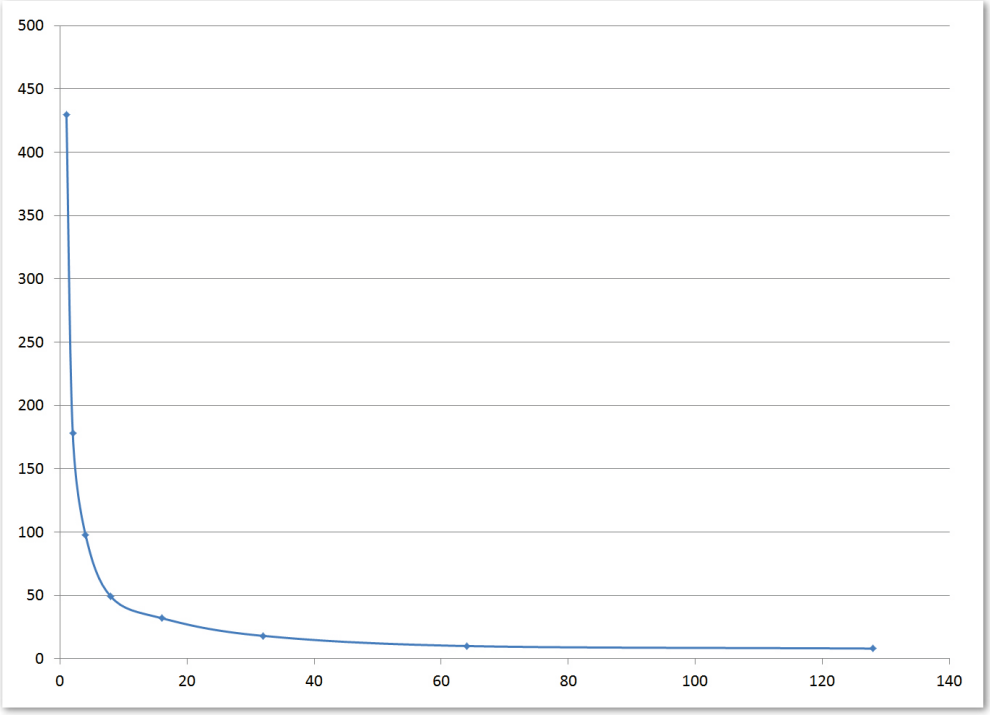


Abbildung 1: Zeitverbrauch in Abhängigkeit von der Bulk-Größe

grundsätzlich mengenorientiert programmiert und nicht zeilenorientiert. Bei einer Bearbeitung von Daten Zeile für Zeile (Tom Kyte übersetzt „row by row“ durch „slow by slow“) führen die häufigen Umgebungswechsel zu schlechter Performance. Selbst höherer Programmieraufwand ist gerechtfertigt, wenn dadurch mengenorientiert gearbeitet werden kann.“

Öffnen eines Cursors

Ebenso offensichtlich ist, dass von PL/SQL nach SQL gewechselt werden muss, wenn ein Cursor geöffnet wird. Auch hier gilt die Empfehlung, mit Mengenverarbeitung (beim Lesen von Daten allerdings mit der Anweisung „bulk collect into“) zu arbeiten, falls lediglich Daten-Strukturen in PL/SQL-Variablen umkopiert werden sollen.

Arbeitet man mit „cursor for“-Schleifen nd ist der Parameter „PLSQL_OPTIMIZE_LEVEL“ mindestens auf den Wert „2“ eingestellt, wird der Compiler beim Kompilieren des PL/SQL-Codes diese automatisch zu „bulk collect into“-Schleifen umbauen, ebenfalls mit einer reduzierten Bulk-Größe von hundert Zeilen. Ein Beispiel dafür können wir sparen, denn es handelt sich um das Standardverfahren zum Arbeiten mit Datenmengen in PL/SQL.

Interessant wird es allerdings, wenn die Situation nicht ganz so einfach ist. Man stelle sich folgenden Klassiker vor: Es wird ein Cursor benötigt, der über alle Abteilungen iteriert, um für eine Abteilung in einem zweiten Cursor alle Mitarbeiter zu bearbeiten. *Listing 9* zeigt den Code dafür.

```
declare
   cursor dept_cur is
       select deptno, dname, loc
           from dept;
   cursor emp_cur(p_deptno in number) is
       select empno, ename, job, sal
       from emp
       where deptno = p_deptno
       order by ename;
begin
   for dept in dept_cur loop
       dbms_output.put_line('Abteilung ' || dept.dname);
       for emp in emp_cur(dept.deptno) loop
           dbms_output.put_line(
           '. ' || emp.ename || ', ' || emp.job);
       end loop;
   end loop;
 end;
/
```

Listing 9

Dieser Code hat gleich zwei massive Probleme: Zum einen, das ist aus dem bisher Gesagten klar, werden nun pro Durchlauf der äußeren Schleife Umgebungswechsel für das Öffnen jedes inneren Cursors notwendig. Das allein wäre nur lästig beziehungsweise langsam. Schlimmer ist jedoch, dass der innere Cursor nicht lesekonsistent zum äußeren Cursor ist: Ändern sich die Zeilen der Tabelle „EMP“ während des Laufs des PL/SQL-Codes, sehen nachfolgende Durchläufe der inneren Schleife die geänderten Daten. Dies liegt daran, dass „select“-Anweisungen keine Sperren in der Datenbank hinterlassen und daher nicht den Regeln der Lesekonsistenz unterliegen.
Ein rabiater Ausweg wäre, vor der Ausführung des Codes den Serialization Level der Session auf „serializable“ zu stellen und anschließend explizit eine Transaktion anzufordern. Wenn jemand die Anweisungen dafür nicht kennt, dann wohl auch deshalb, weil dies bei Oracle so gut wie nie unbedingt nötig ist.

Was aber ist der Ausweg aus dieser Situation? Wieder wird die Programmierung etwas umfänglicher, aber eben auch sowohl lesekonsistent als auch schnell. Wir benötigen einen Cursor-Ausdruck in der „select“-Anweisung *(siehe Listing 10)*.

Man übernimmt explizit die Kontrolle über den Cursor („cursor for“-Schleifen können mit Cursor-Ausdrücken nicht genutzt werden) und deklariert im Cursor mit einem Cursor-Ausdruck die „n“-Seite der Abfrage. Auf diese Weise wird die gesamte Ergebnismenge in einem Roundtrip zur SQL-Seite ermittelt und lesekonsistent zur Verfügung gestellt.

Die Ergebnisse der Unterabfrage werden durch den Cursor-Ausdruck in der Schleife in einen lokalen Cursor vom Typ „SYS_REFCURSOR“ übernommen (andere Varianten wären möglich, aber komplexer) und in der äußeren Schleife in eine Variable dieses Typs. Über diesen Cursor „L_EMP_CUR“ kann nun eine zweite Schleife iterieren und die Werte ausgeben.

Die Lehre aus der Geschichte: „Ein Cursor ist ein unvermeidbarer Umgebungswechsel zwischen SQL und PL/SQL und daher nicht schlimm. Wenn aber geschachtelte Cursor verwendet werden, ist es erforderlich, die Daten in einem Roundtrip zu generieren und mit Cursor-Ausdrücken auszuwerten, um nicht in Lesekonsistenz-Probleme und schlechte Performance zu schlittern.“

Verdeckte Umgebungswechsel

Etwas weniger durchsichtig wird es, wenn Umgebungswechsel nicht offensichtlich sind. Wie auch schon in den gezeigten Beispielen liegt die Gefahr darin, dass die Programmierung verdeckter Umgebungswechsel mit elegant erscheinendem Code möglich ist und daher für richtig erachtet wird.

Dabei fällt vor allem die Instanziierung von Objekttypen beziehungsweise die Verwendung von „member“-Funktionen dieser Typen auf. Hat man eigene Objekt-Typen erstellt und hierfür Konstruktor- oder „member“-Funktionen angelegt, wird der Gebrauch dieser Funktionen zu Umgebungswechseln führen. Das ist schwer zu verhindern, am ehesten noch dadurch, dass man die Objekte in PL/SQL erstellt und mit „bulk“-Operationen an SQL übergibt. Ob der Aufwand hierfür gerechtfertigt ist, ist allerdings von Fall zu Fall zu entscheiden.

Etwas hinterhältiger verhält sich PL/SQL. Zum einen gibt es Funktionen, die eine Fallback-Lösung auf eine SQL-Abfrage besitzen und daher potenziell für Umgebungswechsel sorgen können. Dazu als Beispiel einmal die Implementierung der Funktion „sysdate“ im Package „STANDARD“ *(siehe Listing 11)*.

Die Funktion ist als externe Funktion „PESSDT“ in C implementiert. Sollte dies aber aus irgendeinem Grund nicht gehen

```
declare
   cursor dept_cur is
       select dname,
              cursor(
              select ename, job
                  from emp e
              where e.deptno = d.deptno
              order by e.ename) emp_cur
          from dept d;
   l_dname dept.dname%type;
   l_emp_cur sys_refcursor;
   l_ename emp.ename%type;
   l_job emp.job%type;
begin
   open dept_cur;
   loop
       fetch dept_cur into l_dname, l_emp_cur;
       exit when dept_cur%notfound;
       dbms_output.put_line('Abteilung ' || l_dname);
       loop
           fetch l_emp_cur into l_ename, l_job;
           exit when l_emp_cur%notfound;
           dbms_output.put_line(
           '. ' || l_ename || ', ' || l_job);
       end loop;
   end loop;
end;
/
```

Listing 10

```
function pessdt return DATE;
      pragma interface (c,pessdt);
   -- Bug 1287775: back to calling ICD.
   -- Special: if the ICD raises ICD_UNABLE_TO_COMPUTE, that means we
should do
   -- the old 'SELECT SYSDATE FROM DUAL;' thing.      This allows us to
do the
   -- SELECT from PL/SQL rather than having to do it from C (within the
ICD.)
   function sysdate return date is
       d date;
   begin
       d := pessdt;
       return d;
   exception
       when ICD_UNABLE_TO_COMPUTE then
           select sysdate into d from sys.dual;
       return d;
   end;
```

Listing 11

(„exception ICD_UNABLE_TO_COMPUTE“), erfolgt „the old 'SELECT SYSDATE FROM DUAL' thing“, also ein Umgebungswechsel zu SQL. Das ist vielleicht etwas exotisch, aber es spricht dafür, auch „sysdate“ nicht in einer Schleife zu verwenden, sondern lieber eine lokale Variable „l_now date := sysdate;“ zu vereinbaren und in der Methode zu benutzen. Auf diese Weise ist auch sichergestellt, dass alle Iterationen einer Schleife zur gleichen Zeit ausgeführt werden (so dies denn nicht ausdrücklich vermieden werden soll).

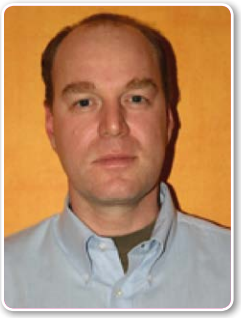
Wichtiger ist sicher noch das Beispiel der Funktion „USER“, die viele für eine Pseudo-Spalte halten. Das ist allerdings eine „select“-Abfrage gegen die Datenbank, wie ein Blick in das Package „STANDARD“ zeigt. Daher verbietet sich der Einsatz von „USER“ in Schleifen, zumal es doch eher unwahrscheinlich ist, dass sich der angemeldete Datenbank-Benutzer während eines laufenden PL/SQL-Blocks ändert. Auch hier lautet die Empfehlung, sich den aktuell angemeldeten Benutzer als Konstante in den Code zu holen, vielleicht in einem Konstanten-Package, das den Namen beim Initialisieren einmalig erfragt.

Fazit

Umgebungswechsel sind ein ganz wesentliches Problem der Programmierung in PL/SQL und der Abfrage-Gestaltung in SQL. Daher sollte man einen Blick für mögliche Umgebungswechsel entwickeln. Umgebungswechsel sind natürlich auch über die bekannten Hilfsmittel „TKPROF“ oder den hierarchischen Profiler „HPROF“ zu finden.

Die in diesem Artikel zusammengetragenen Best Practices sind nicht umfassend, nicht einmal bezüglich der Umgebungswechsel, stellen jedoch einen wesentlichen Teil der Strategien dar, die ein PL/SQL-Entwickler im Hinterkopf haben sollte, um grundsätzlich performanten und skalierbaren Code zu schreiben.

Performance-Tuning beginnt ja nicht erst, wenn ein Problem existiert, sondern manifestiert sich zunächst einmal darin, dass die groben und bekannten Fehler von vornherein vermieden werden. Dies gilt insbesondere für Code, von dem man annimmt, dass er nicht Performance-kritisch sei und von daher ja auch nachlässiger programmiert werden könnte. Erfahrungsgemäß lebt solcher Code ewig und zwar nach Murphys Gesetz genau dort, wo er erheblich im Wege steht.



Jürgen Sieben
j.sieben@condes.de