Wann PL/SQL nicht verwendet werden sollte

Jürgen Sieben, ConDeS GmbH & Co. KG

Diese Folge erscheint dem Autor unter dem Eindruck mehrerer Veröffentlichungen, auch im Red Stack Magazin, sinnvoll: Im Red Stack Magazin zum Beispiel fand er vor einigen Ausgaben einen Artikel über die Vermeidung von Umgebungswechseln zwischen SQL und PL/SQL. So lesenswert und richtig der Artikel auch ist, so leidet das dort verwendete Beispiel jedoch an einem verbreiteten Problem, das in dieser Kolumne verdeutlicht werden soll.

In einem Kochbuch für PL/SQL-Code, also einem Buch, das Musterlösungen für verbreitete Programmierprobleme anbietet, fand sich folgende, sehr schöne PL/SQL-Prozedur für das Problem, doppelte Einträge in einer Tabelle zu finden *(siehe Listing 1)*. Wunderbar! Bitte versuchen Sie zunächst, den Code zu verstehen, und fragen sich, was daran wohl falsch sein könnte.

Das Problem

Die Tabelle „EMPLOYEES“ gehört zum Schema „HR“ und enthält 107 Zeilen. Das ist nichts. Daher erstellen wir eine realistischere Simulation und verwenden die Daten der Tabelle „ALL_OBJECTS“ mit etwa 100.000 Zeilen für unsere Beispiele. Die Datenbank-Objekte des Benutzers „SH“ werden anschließend doppelt importiert, damit Verstöße gegen einen zu schaffenden Primärschlüssel auftreten *(siehe Listing 2)*. Dann portieren wir das Rezept auf unsere Daten *(siehe Listing 3)*. Wer eine Pause braucht, ruft die Prozedur nun auf *(siehe Listing 4)*.

Dieser Code hat offensichtlich eine ganze Menge Probleme; eines davon ist die Laufzeit, ein anderes die Tatsache, dass jede Dublette doppelt ausgegeben wird. Zeit zu analysieren, auf welche Weise die Dubletten in der Tabelle aufgespürt werden. Wir iterieren über alle Zeilen der Tabelle „TEST_TABLE“ und öffnen für jede ermittelte Zeile einen weiteren Cursor

```
declare
   cursor emp_cur is
       select *
           from employees
       order by employee_id;
   emp_count number := 0;
   total_count number := 0;
begin
   for emp in emp_cur loop
       select count(*)
           into emp_count
           from employees
           where employee_id = emp.employee_id;
       if emp_count > 1 then
           total_count := total_count + 1;
           -- do_something
       end if;
   end loop;
end;
```

Listing 1

```
SQL> create table test_table(
   2       owner varchar2(30),
   3       object_id number,
   4       object_name varchar2(30),
   5       object_type varchar2(30));

Tabelle wurde erstellt.

SQL> insert into test_table(owner, object_id, object_name, object_type)
   2       select owner, object_id, object_name, object_type
   3       from all_objects
   4       union all
   5       select owner, object_id, object_name, object_type
   6       from all_objects
   7       where owner = 'SH';

92553 Zeilen erstellt.
```

Listing 2

darauf, um die Anzahl der Zeilen zu zählen, die für diese ID vorhanden sind. Um allem die Krone aufzusetzen, ist die Tabelle „TEST_TABLE“ des äußeren Cursors zudem noch sortiert, obwohl nicht ganz klar ist, wozu das dienen soll. Der äußere Cursor wird also etwa 95.000 Zeilen lesen und im Arbeitsspeicher des Session-Kontextes sortieren, um anschließend für jede Zeile eine Auswertung auf die gleiche Tabelle durchzuführen. Sollte sich dann herausstellen, dass eine Primärschlüssel-Information doppelt vorhanden ist, wurde eine Dublette gefunden und ausgegeben.

Eines der Probleme des Codes besteht darin, dass nicht sichergestellt ist, dass die weiteren „select“-Abfragen in einem lesekonsistenten Zustand zur äußeren Abfrage beantwortet werden. Eine „select“-Abfrage sperrt keine Daten und solange der Session-Isolation-Level nicht auf „SERIALIZABLE“ gestellt wurde, wird jede „select“-Abfrage die Daten der Tabelle so sehen, wie sie zum Zeitpunkt der Abfrage sichtbar waren. Die äußere Abfrage wird in über drei Minuten ausgeführt, das heißt, dass die letzten „select“-Abfragen die gleiche Tabelle über drei Minuten später sehen als die ersten Abfragen der Schleife. Zudem werden immerhin etwa 100.000 „select“-Abfragen abgearbeitet.

Dieser Ansatz atmet prozeduralen Geist, was nicht per se schlecht, in Datenbanken aber beinahe immer falsch ist. Das Mantra lautet:

- Wenn du ein Problem hast, löse es in SQL
- Wenn das gar nicht geht, löse es in PL/SQL
- Wenn das gar nicht geht, löse es in Java
- Wenn das gar nicht geht, löse es in C
- Wenn das gar nicht geht – überlege, was für ein Problem du da hast

Das Wichtige an diesem Mantra ist die Reihenfolge der einzusetzenden Programmiersprachen. Das Problem ist, dass die vorgestellte Prozedur hier einen entscheidenden Fehler macht.

Lösungsansätze

Versuchen wir, den Code zu retten, indem wir das Problem der Lese-Konsistenz und der vielen „select“-Anweisungen angehen. Inhaltlich sollen doch wohl die

```
SQL> create or replace procedure report_duplicates
  2  as
  3      cursor object_cur is
  4          select *
  5              from test_table
  6          order by object_id;
  7      l_obj_count number := 0;
  8      l_total_count number := 0;
  9  begin
 10      for obj in object_cur loop
 11          select count(*)
 12              into l_obj_count
 13              from test_table
 14              where object_id = obj.object_id;
 15          if l_obj_count > 1 then
 16              dbms_output.put_line(
 17                  obj.object_type || ' ' ||
 18                  obj.object_name || ' existiert doppelt.');
 19          end if;
 20      end loop;
 21  end report_duplicates;
 22  /

Prozedur wurde erstellt.
```

Listing 3

```
SQL> set serveroutput on;
SQL> call report_duplicates();

TABLE COSTS existiert doppelt.
TABLE COSTS existiert doppelt.
TABLE PARTITION COSTS existiert doppelt.
TABLE PARTITION COSTS existiert doppelt.
TABLE PARTITION COSTS existiert doppelt.
TABLE PARTITION COSTS existiert doppelt.
TABLE PARTITION COSTS existiert doppelt.
...

Aufruf wurde abgeschlossen.
Abgelaufen: 00:03:14:51
```

Listing 4

```
SQL> select object_id, object_type, object_name, count(*) anzahl
  2   from test_table
  3   group by object_id, object_type, object_name
  4   having count(*) > 1;

OBJECT_ID OBJECT_TYPE              OBJECT_NAME               ANZAHL
--------- ------------------------ ------------------------  ----------
   92618  TABLE PARTITION          COSTS                     2
   92613 TABLE PARTITION           COSTS                     2
   92660 TABLE PARTITION           SALES                     2
   92694 TABLE                     TIMES                     2
   92850 INDEX                     CHANNELS_PK               2
   92891 INDEX PARTITION           COSTS_TIME_BIX            2
...
310 Zeilen ausgewählt.

Abgelaufen: 00:00:00.48
```

Listing 5

Daten, die nicht als Dublette vorhanden sind, ignoriert werden. Wir wollen also lediglich die Daten sehen, die doppelt auftauchen. Das klingt sehr nach der „select"-Anweisung wie in *Listing 5*.

Eine erste Verbesserung, das ist klar; man achte auf die Zeit. Nun könnte man den Eindruck haben, die Abfrage würde Dubletten doppelt ausgeben (die ersten beiden Zeilen), doch wir sehen an der Objekt-ID, dass dies nicht der Fall ist, sondern hier tatsächlich zwei Partitionen unter dem Namen der Tabelle vorliegen. Diese Abfrage würde sich gut für die Verwendung im Cursor der Prozedur eignen, zumal sich dadurch die innere Abfrage komplett erledigt hätte *(siehe Listing 6)*.

Doch die Frage ist, was wir eigentlich tun möchten. Wollten wir nur das Ergebnis auf den Bildschirm schreiben? Dann hätten wir die Text-Konkatenation in „dbms_output.put_line()" auch direkt in der „select"-Abfrage schreiben können. Wahrscheinlicher ist jedoch, dass wir die Dubletten nur einfach loswerden möchten. Welche Optionen hier zur Verfügung stehen, hängt vom Szenario ab. Spielen wir einmal einige durch.

```
SQL> create or replace procedure report_duplicates
  2  as
  3    cursor object_cur is
  4      select object_id, object_type, object_name, count(*) anzahl
  5        from test_table
  6       group by object_id, object_type, object_name
  7      having count(*) > 1;
  8  begin
  9    for obj in object_cur loop
 10      dbms_output.put_line(
 11        obj.object_type || ' ' ||
 12        obj.object_name || ' existiert doppelt.');
 13    end loop;
 14  end report_duplicates;
 15  /

Prozedur wurde erstellt.

SQL> call remove_duplicates();
TABLE PARTITION COSTS existiert doppelt.
TABLE PARTITION COSTS existiert doppelt.
TABLE PARTITION SALES existiert doppelt.
TABLE TIMES existiert doppelt.
...
Aufruf wurde abgeschlossen.

Abgelaufen: 00:00:00.45
```

Listing 6

Daten aus einer CSV-Datei importieren

Das wahrscheinliche Szenario ist, dass wir die guten Daten in eine Ziel-Tabelle kopieren und die schlechten verwerfen möchten. Das ist ohne PL/SQL machbar *(siehe Listing 7)*.

Die „insert"-Anweisung ignoriert die Dubletten beim Import. Die Lösung nutzt die analytische Funktion „rank()", die eine Reihenfolge über ein Gruppierungskriterium nach einem Sortierkriterium bildet. Da die doppelten Zeilen gleich sind, benötigen wir ein Kriterium, um uns zwischen den beiden zu entscheiden. Hier bietet sich zum Beispiel die „ROWID" an. In der äußeren Abfrage werden anschließend alle Zeilen gefiltert, die keine Dubletten sind, denn diese haben einen „Rang = 1". Die Liste der „ROWID" der verbleibenden Zeilen wird dann zum Identifizieren der Zeilen verwendet, die eingefügt werden sollen. Es ist zu beachten, in diesem Fall keine „row limiting"-Klausel ab Version 12c einzusetzen, da sich diese nicht (nach der Objekt-ID) partitionieren lässt.

```
SQL> create table target_table
  2   as select *
  3   from test_table
  4   where null is not null;

Tabelle wurde erstellt.

SQL> alter table target_table add constraint pk_target_table
  2   primary key(object_id);

Tabelle wurde geändert.

SQL> insert into test_table(owner, object_id, object_name, object_type)
  2   select owner, object_id, object_name, object_type
  3       from test_table
  4   where rowid in (
  5           select row_id
  6               from (select rowid row_id,
  7                      rank() over (
  8                          partition by object_id order by rowid) rang
  9                  from test_table)
  10              where rang = 1);

92243 Zeilen erstellt.
Abgelaufen: 00:00:00.17
```

Listing 7

Daten bereinigen, um einen Primärschlüssel einzurichten

Sind die Daten bereits in der Tabelle enthalten und sollen die Dubletten gelöscht werden, ist auch dies ohne PL/SQL möglich *(siehe Listing 8)*.

Auch hier gilt es, auf die Ausführungszeit zu achten. Die Lösung nutzt wieder die analytische Funktion „rank()“. In der äußeren Abfrage werden nur Dubletten gefiltert, denn diese haben einen „Rang > 1“. Die Liste der „ROWID“ dieser Zeilen wird anschließend zum Bereinigen der Tabelle verwendet.

Die guten ins Töpfchen, die schlechten ins Kröpfchen

Ist das Ziel die Separierung der beiden Datenmengen, stehen ebenfalls mehrere Wege zur Verfügung. Einerseits könnte man eine „insert“-Anweisung in die Ziel-Tabelle mit einer „log errors“-Klausel ausstatten, die bei einem Primärschlüssel-Verstoß die verstoßende Zeile in eine Fehler-Tabelle schreiben wird *(siehe Listing 9)*.

Alternativ könnte der Weg über eine „multi-table-insert“-Anweisung laufen. Bei dieser Variante würden die Daten durch eine entsprechende „select“-Anweisung mit einem Rang ausgestattet und beim Einfügen der Daten eine Fallunterscheidung nach dem Rang ausgeführt *(siehe Listing 10)*. Die Aufbereitung der Daten ist bekannt, hier entfällt nur die Filterung über die Dubletten, da diese ja für die Fallunterscheidung benötigt werden.

```
SQL> delete from test_table
  2   where rowid in (
  3       select row_id
  4           from (select rowid row_id,
  5               rank() over (
  6                   partition by object_id order by rowid) rang
  7               from test_table)
  8           where rang > 1);

310 Zeilen gelöscht.
Abgelaufen: 00:00:00.06
```

Listing 8

```
SQL> call dbms_errlog.create_error_log('TARGET_TABLE', 'TARGET_ERR');

Aufruf wurde abgeschlossen.

SQL> insert into target_table
  2   select *
  3       from test_table
  4           log errors into target_err
  5   reject limit unlimited;

92243 Zeilen erstellt.

Abgelaufen: 00:00:00.23

SQL> select count(*)
  2   from target_err;

  COUNT(*)
----------
       310
```

Listing 9

```
SQL> insert first
  2       when rang = 1 then
  3       into target_table(owner, object_id, object_name, object_type)
  4           values(owner, object_id, object_name, object_type)
  5       else into target_err(owner, object_id, object_name, object_type)
  6           values(owner, object_id, object_name, object_type)
  7   select t.*, rank() over (partition by object_id order by rowid) rang
  8       from test_table t;

92553 Zeilen erstellt.

Abgelaufen: 00:00:00.14
```

Listing 10

Fazit

Das Hauptproblem der Programmierung von Datenbanken ist, der Datenbank möglichst wenig im Weg zu stehen. Genau das passiert allerdings gerade routinierten Anwendungsprogrammierern häufig. Zu sehr ist man den prozeduralen Ansatz gewohnt, als dass man sich auf die Suche nach einem mengenorientierten Ansatz macht. SQL hat viele Anstrengungen unternommen, prozedurale Programmierung unnötig zu machen und Standardprobleme direkt aus SQL heraus zu lösen.

Sollten die eingesetzten Techniken unbekannt gewesen sein („log errors“-Klausel, „multi-table-insert“, „analytische Funktionen“ etc.), kann man das doch bitte als Einladung verstehen, sich wieder einmal mit SQL auseinanderzusetzen. Die Lösungen sind im Regelfall kürzer, schneller, besser zu warten und benötigen keine weitreichenden Unit-Tests. Genug Argumente also, PL/SQL auch einmal im Schrank zu lassen.

Jürgen Sieben
j.sieben@condes.de