PL/SQL Performance – Best Practices für Laufzeitoptimierung

Jan Gorkow
SD&C Solutions Development & Consulting GmbH
Berlin

Schlüsselworte
PL/SQL, PL/SQL Development, PL/SQL Performance Tuning

Einleitung
Der Vortrag befasst sich mit Techniken zur Umsetzung von hoch-performanten Oracle Datenbankapplikationen, bei denen eine möglichst geringe Laufzeit von entscheidender Bedeutung ist. Relevant ist dies bspw. im Bereich rechenintensiver Web-Applikationen, wo sich bereits geringfügig zu lange Wartezeiten direkt negativ auf die Benutzerakzeptanz auswirken können. Eingegangen wird in diesem Zusammenhang auf Caching Mechanismen, Native Kompilierung, die Verwendung optimierter Datentypen sowie Möglichkeiten der Parallelisierung.

Der Function Result Cache
Der Function Result Cache wurde mit Oracle 11g eingeführt und ist seitdem eines der wichtigsten Features zur Steigerung der Performance von PL/SQL-Programmen. Er ermöglicht die Ablage von beliebig einfach oder komplex strukturierten Funktionsergebnissen in der SGA der Instanz, wodurch die gecachten Werte grundsätzlich Session-übergreifend zur Verfügung stehen. Der Cache Key wird hierbei gebildet aus sämtlichen Parameterwerten, die an die Funktion übergeben wurden. Oracle invalidiert automatisch gecachte Ergebnisse wenn die entsprechende Funktion bzw. das Package, in dem sich diese befindet, neu kompiliert wird oder sich die Datengrundlage in den referenzierten Tabellen ändert. In Oracle 11g Release 1 mussten hierfür die relevanten Tabellenabhängigkeiten noch umständlich manuell über die RELIES_ON – Klausel gepflegt werden. Diese Klausel ist seit Oracle 11g Release 2 obsolet. Nun findet die Verwaltung und Prüfung der abhängigen Tabellen automatisch statt, was den Einsatz des Function Result Caches erheblich einfacher, robuster und komfortabler macht. Die Nutzung wird über die RESULT_CACHE-Klausel im Funktionsheader aktiviert:

```
FUNCTION get_stock_price_rc (isin IN stock_price.isin%TYPE)
   RETURN stock_price.price%TYPE
   RESULT_CACHE
IS
…
```

Der Result Cache selbst wird primär über die folgenden beiden Parameter konfiguriert:

`RESULT_CACHE_MAX_SIZE`
gibt die Größe des Result Caches innerhalb der SGA an. Hierbei wird nicht zwischen dem PL/SQL Function Result Cache und dem Query Result Cache unterschieden.

`RESULT_CACHE_MAX_RESULT`
ist der maximale Anteil eines einzelnen Results innerhalb des Result Caches in Prozent.

Selbstverständlich macht der Einsatz des Function Result Caches nur für deterministische Funktionen Sinn, die bei gleicher Parametrisierung und Datengrundlage auch identische Ergebnisse produzieren, d.h. bspw. intern nicht von SYSDATE abhängig sind. Dies muss der Entwickler selbst sicherstellen, da eine Prüfung diesbezüglich nicht durch Oracle vorgenommen wird. Ferner ist Vorsicht bei der

Verwendung von dynamischem SQL geboten, da Oracle die sich hieraus ergebenden Abhängigkeiten nicht berücksichtigen kann!

Eines der effektivsten Einsatzgebiete sind Lookup-Routinen, welche ihre benötigten Informationen aus Tabellen laden. Beim Zugriff auf bereits gecachte Ergebnisse entfällt durch die Nicht-Ausführung der Query auch der entsprechende Context Switch zwischen der PL/SQL- und der SQL-Engine.
Der Performancevorteil, der sich durch die Nutzung des Function Result Caches ergibt ist abhängig von der Komplexität bzw. Laufzeit der spezifischen Funktion sowie der Häufigkeit der Aufrufe auf Basis identischer Parameter, aber ebenso auch die Häufigkeit der Invalidierung der Cache Results aufgrund von Änderungen an den zugrunde liegenden Tabellendaten.

Werden Funktionsergebnisse oder allgemeiner Daten im Rahmen einer Verarbeitung vielfach benötigt, so macht es ggf. Sinn, die Werte in PL/SQL-Variablen, z.B. assoziativen Arrays, in der lokalen Session zu halten. Der Zugriff auf Variablen innerhalb der eigenen Session ist aufgrund ihrer Ablage in der PGA noch einmal um einiges schneller, als der Zugriff auf den zentralen Function Result Cache. Dieses sogenannte PGA Caching kann somit eine Alternative aber auch eine Ergänzung zum Einsatz des Function Result Cache sein. Besondere Beachtung muss in diesem Fall dem erhöhten Speicherverbrauch geschenkt werden, da identische Daten ggf. in den PGAs verschiedener Server Prozesse vorgehalten werden.

Native Kompilierung
Die Standard-Parametrisierung einer Oracle Datenbank sieht vor, dass jeglicher PL/SQL Code interpretiert kompiliert wird. Hierbei wird der Source Code beim Einspielen in eine Art Zwischencode umgewandelt, welcher im System Katalog hinterlegt wird. Wird dieser dann aufgerufen, so werden die einzelnen Anweisungen interpretiert und die in den Oracle Libraries hinterlegte Funktionalität ausgeführt.
Bei nativer Kompilierung wird der PL/SQL Code beim Einspielen direkt nativen Code umgewandelt, welcher wiederum im System Katalog hinterlegt wird. Bei dessen Ausführung entfällt die Interpretierung, wodurch eine schnellere Abarbeitung ermöglicht wird. Hierbei ist jedoch zu beachten, dass sich die Art der gewählten Kompilierung ausschließlich auf PL/SQL-Anweisungen auswirkt. Eingebettete SQL-Anweisungen müssen stets interpretiert werden. Daher profitiert PL/SQL-Code umso mehr von nativer Kompilierung, je weniger SQL enthalten ist. Ferner ist zu beachten, dass ausschließlich interpretierter PL/SQL Code debugt werden kann, weshalb die native Kompilierung auf Entwicklungssystemen eher selten zum Einsatz kommt. Nicht zuletzt ist die native Kompilierung aufwendiger und dauert damit etwas länger und der so erzeugte Code benötigt zur Laufzeit mehr Shared Memory.

Die Entscheidung für interpretierte bzw. native Kompilierung kann auf Instanz-, Session- oder Statement-Ebene getroffen werden und wird über den Parameter PLSQL_CODE_TYPE gesteuert:

```
ALTER SYSTEM SET PLSQL_CODE_TYPE = [INTERPRETED | NATIVE];

ALTER SESSION SET PLSQL_CODE_TYPE = [INTERPRETED | NATIVE];

ALTER PACKAGE [name] COMPILE BODY PLSQL_CODE_TYPE = [INTERPRETED | NATIVE];
```

Die native Kompilierung kann neben dem benutzerdefinierten Code auch auf die Built In Packages angewendet werden. Hierfür stellt Oracle entsprechende Skripte zur Rekompilierung der Packages zur Verfügung. Wird die Kompilierungsart geändert, so sollte auch der Parameter PLSQL_CODE_TYPE auf System-Ebene geändert werden, um den Standard für zukünftige Kompilierungen entsprechend anzupassen.

Verwendung optimierter Datentypen
Muss man sich bei der Entwicklung eines PL/SQL-Programms für einen numerischen Datentyp entscheiden, so fällt die Wahl nicht selten automatisch auf den Datentyp NUMBER, nicht zuletzt, da dieser in der Oracle Datenbank auch der gängigste numerische SQL Datentyp ist. Darüber hinaus stehen jedoch eine ganze Reihe weiterer Typen zur Verfügung, z.B. INTEGER, BINARY_DOUBLE, BINARY_FLOAT sowie die mit Oracle 11g eingeführten SIMPLE-Varianten hiervon. Doch wodurch unterscheiden sich diese Typen voneinander?

Zunächst einmal lassen sich alle Typen danach unterscheiden, in welcher Art ihre Arithmetik umgesetzt wird. NUMBER und INTEGER verwenden Softwarearithmetik, d.h. ihre Rechenlogiken sind in Softwarebibliotheken hinterlegt, was sie sehr genau und robust macht. So werden sie bei jeder Operation auf Over- bzw. Underflow sowie auf NULL geprüft. Allerdings sind die Rechenoperationen hierdurch auch langsamer als bei der Verwendung von Datentypen, deren Arithmetik direkt durch die Hardware umgesetzt wird. Dies ist bspw. bei den Typen PLS_INTEGER, BINARY_FLOAT und BINARY_DOUBLE der Fall. Bei BINARY_FLOAT und BINARY_DOUBLE handelt es sich um Fließkommatypen mit einer Größe von 4 bzw. 8 Byte. Da sie keine implizite Over- bzw. Underflow-Prüfung haben, sind sie zwar schneller, aber dafür muss sich der Programmierer selbst um die ggf. erforderlichen Gültigkeitsprüfungen kümmern. Mit der Version 11g hat Oracle die Datentypen SIMPLE_INTEGER, SIMPE_FLOAT und SIMPLE_DOUBLE eingeführt. Diese basieren auf den Typen PLS_INTEGER, BINARY_FLOAT und BINARY_DOUBLE und sind mit einem NOT NULL Constraint versehen. Da hierdurch die NULL-Prüfung im Rahmen arithmetischer Operationen entfallen kann, sind sie noch einmal performanter.

Um die Performanceunterschiede aufzuzeigen, wurde ein Testfall entwickelt, welche eine Variable mit einem Ausgangswert von Null 100.000.000 mal um den Wert Eins erhöht. Dieser Testfall wurde für diverse Datentypen und sowohl interpretiert als auch nativ kompiliert ausgeführt. Der folgenden Tabelle sind die gemessenen Laufzeiten zu entnehmen.

Datentyp	**Arithmetik**	**Interpretiertes PL/SQL**	**Natives PL/SQL**	**Verbesserung durch native Kompilierung**
NUMBER	Software	2,094s	1,578s	25%
INTEGER	Software	3,984s	3,391s	15%
PLS_INTEGER	Hardware	0,704s	0,344s	51%
SIMPLE_INTEGER	Hardware	0,641s	0,156s	76%
BINARY_DOUBLE	Hardware	1,094s	0,281s	74%
SIMPLE_DOUBLE	Hardware	1,094s	0,266s	76%

Zwei Punkte sind bei der Betrachtung der Werte besonders beachtenswert:

- Der INTEGER-Datentyp ist als ein von NUMBER abgeleiteter Datentyp aufgrund der zusätzlichen Constraints bei Berechnungen nur halb so performant, wie NUMBER selbst.

- Die direkt von der Hardware umgesetzten Datentypen profitieren überproportional von einer nativen Kompilierung des PL/SQL Codes.

Bei der Verwendung der aufgrund ihrer Hardware-Unterstützung hoch-performanten Datentypen sollten mindestens die folgenden Punkte betrachtet bzw. bedacht werden:

- Genügt der jeweils verfügbare Wertebereich?
- Kann die Zuweisung von NULL-Werten ausgeschlossen werden?
- Können Over- bzw. Underflows ausgeschlossen werden bzw. wo muss auf diese geprüft werden?
- Kann beim Einsatz der Fließkommatypen mit den resultierenden Ungenauigkeiten dennoch gearbeitet werden?

Insbesondere der letzte Punkt soll an dieser Stelle aufgrund seiner Wichtigkeit noch einmal genauer betrachtet werden. Man muss sich bei der Verwendung von Fließkommatypen stets darüber bewusst sein, dass die recht große Genauigkeit von Software-basierten Datentypen hier schlichtweg nicht zwingend gegeben ist. Dies soll an folgendem Beispiel verdeutlicht werden:

```
DECLARE
   v_value1   SIMPLE_DOUBLE := 0.1;
   v_value2   SIMPLE_DOUBLE := 0.2;
   v_value3   SIMPLE_DOUBLE := 0.3;
BEGIN
   DBMS_OUTPUT.put_line (
      'v_value1: ' || TO_CHAR (v_value1, '0D99999999999999999'));
   DBMS_OUTPUT.put_line (
      'v_value2: ' || TO_CHAR (v_value2, '0D99999999999999999'));
   DBMS_OUTPUT.put_line (
      'v_value3: ' || TO_CHAR (v_value3, '0D99999999999999999'));
   DBMS_OUTPUT.put_line ('');
   DBMS_OUTPUT.put_line (
         'v_value1 + v_value2: '
      || TO_CHAR (v_value1 + v_value2, '0D99999999999999999'));
   DBMS_OUTPUT.put_line ('');
   DBMS_OUTPUT.put_line (
         'v_value1 + v_value2 = v_value3: '
      || CASE
            WHEN v_value1 + v_value2 = v_value3 THEN 'TRUE'
            ELSE 'FALSE'
         END);
END;
/
```

Die Ausführung dieses Codes erzeugt folgende Ausgabe:

```
v_value1:  0.10000000000000001
v value2:  0.20000000000000001
v value3:  0.29999999999999999
v value1 + v value2:  0.30000000000000004
v_value1 + v_value2 = v_value3: FALSE
```

Parallelisierung
So optimiert der Quellcode im Hinblick auf die bis dato angesprochenen Aspekte auch ist: Ohne eine parallelisierte Verarbeitung wird die Software kaum die seitens der heute gängigen Hardware bereitgestellten Ressourcen nutzen. Zwar ist das Oracle DBMS selbst über die Anzahl der vorhandenen CPUs bzw. CPU-Cores hochgradig skalierbar, allerdings wird der PL/SQL Code einer Datenbank Session zunächst mal nur durch einen Prozess bzw. einen Thread und somit streng sequentiell abgearbeitet.

Eine Parallelisierung muss folgerichtig durch eine Aufteilung der anfallenden Rechenschritte auf mehrere Datenbank Server Prozesse erfolgen. Im einfachsten Fall lässt sich die komplexe Rechenlogik einer Anwendung in mehrere unabhängige Teilberechnungen splitten, deren Einzelergebnisse am Ende wieder zusammengeführt werden. Bauen hingegen einzelne Rechenschritte aufeinander auf, die jeweils nur auf einem kleinen Teil einer zu verarbeitenden Gesamtdatenmenge basieren, so wäre ein Pipeline-Ansatz denkbar, bei dem das Ergebnis eines Prozesses an einen weiteren Prozess übergeben wird und diesem dann als Eingangsdatenmenge für dessen Teilschritt dient.

Wie auch immer die Gesamtaufgabe für die Aufteilung auf mehrere Teilprozesse geschnitten wird: Schnell wird klar, dass technisch ein Datenaustausch zwischen den beteiligten Prozessen erforderlich ist, sei es zur tatsächlichen Weitergabe von Daten oder lediglich zur Status-Rückmeldung im Rahmen einer Prozesssynchronisation. Hierfür bietet sich die Nutzung das Packages DBMS_PIPE an. Im Gegensatz zu den denkbaren Alternativen Advanced Queuing oder dem Datenaustausch über normale Tabellen erfordert der Nachrichtenaustausch via Pipe keine Transaktionen, welche verhältnismäßig teure schreibende und lesende Plattenzugriffe zur Folge hätten. Bei Pipes handelt es sich um Queues, welche ausschließlich in der SGA, also im Arbeitsspeicher einer Oracle-Instanz gehalten werden.

Wie können nun aber innerhalb der Datenbank per PL/SQL zusätzliche Verarbeitungsprozesse gestartet werden? Hierfür sollen im Folgenden zwei Ansätze betrachtet werden, zum einen die Verwendung des Packages DBMS_PARALLEL_EXECUTE und zum anderen die Nutzung des Schedulers (Package DBMS_SCHEDULER).

Das Package DBMS_PARALLEL_EXECUTE ist seit Oracle 11g verfügbar und realisiert intern die Parallelisierung ebenfalls unter zur Hilfenahme des Schedulers. Hintergrund der Einführung war die Schaffung einer Möglichkeit der einfachen Parallelisierung von Update-Anweisungen auf sehr großen Datenmengen, z.B. in Data Warehouses. Es ist jedoch flexibel genug, dass sich mit ihm auch PL/SQL-Anweisungen parallelisiert ausführen lassen, was an folgendem Code-Beispiel verdeutlicht werden soll:

```
…
DBMS_PARALLEL_EXECUTE.create_task (task_name => :task_name);
DBMS_PARALLEL_EXECUTE.create_chunks_by_sql (
  task_name   => :task_name,
  sql_stmt    =>    'WITH instances (instance)'
                 || '        AS (SELECT 1 AS instance FROM DUAL'
                 || '            UNION ALL'
                 || '            SELECT instances.instance + 1 AS'
                 || '             instance'
                 || '              FROM instances'
                 || '             WHERE instances.instance < '
                 || TO_CHAR (:count_instances)
                 || ') '
                 || 'SELECT instances.instance AS start_id,'
                 || '       instances.instance AS end_id'
                 || '  FROM instances',
```

```
  by_rowid    => FALSE);
DBMS_PARALLEL_EXECUTE.run_task (
  task_name        => :task_name,
  sql_stmt         =>    'DECLARE'
                      || '   v_start_id PLS_INTEGER := :start_id;'
                      || '   v_end_id   PLS_INTEGER := :end_id;'
                      || 'BEGIN'
                      || '   prc_workload (instance => v_start_id);'
                      || 'END;',
  language_flag    => DBMS_SQL.native,
  parallel_level   => :count_instances);
DBMS_PARALLEL_EXECUTE.drop_task (task_name => :task_name);
…
```

Zunächst ist ein Task zu erzeugen, welcher einen DB-weit eindeutigen Namen haben muss. Anschließend sind auf Basis einer SQL-Abfrage die erforderlichen Chunks zu definieren, welche die Teil-Arbeitspakete abgrenzen. Diese Abfrage muss zwingend die Felder START_ID und END_ID liefern. Im dritten Schritt wird der Task ausgeführt. Erst hier wird der Parallelisierungsgrad definiert und auch der auszuführende Code übergeben, welcher wiederum zwingend die Bind-Variablen :START_ID und :END_ID enthalten muss. Die RUN_TASK-Routine kommt erst zurück, wenn alle definierten Chunks abgearbeitet wurden, d.h. sie beinhaltet selbst eine Synchronisierung. Im letzten Schritt ist der Aufruf der DROP_TASK-Routine für das House Keeping erforderlich.

Soll hingegen der Scheduler zur Parallelisierung eingesetzt werden, so empfiehlt es sich, für die parallel auszuführenden PL/SQL-Routinen zunächst Scheduler-Programme inkl. der erforderlichen Argumente zu definieren. Dies ist erforderlich, um im Scheduler LIGHTWEIGHT-Jobs erzeugen zu können. Im Vergleich zu REGULAR-Jobs führen LIGHTWEIGHT-Jobs weniger Metadaten mit und sind somit weniger Overhead-behaftet, was ihr Handling etwas schneller macht. Ist dies geschehen, so kann eine Parallelisierung bspw. wie folgt aussehen:

```
…
v_result := DBMS_PIPE.create_pipe (pipename => :pipename,
                                   private  => TRUE);
v_job_definition_array := NEW job_definition_array ();
v_job_definition_array.EXTEND (:count_instances);

FOR i IN 1 .. :count_instances
LOOP
  v_job_definition_array (i) :=
    NEW job_definition (
          job_name     => 'JOB_EXECUTE_PARALLEL_' || TO_CHAR (i),
          job_style    => 'LIGHTWEIGHT',
          program_name => 'WORKSHOP.PRC_WORKLOAD_CALLBACK',
          arguments    => NEW jobarg array (
                            NEW jobarg (
                                  arg_position => 1,
                                  arg_value    => TO_CHAR (i))),
          enabled        => TRUE);
END LOOP;

DBMS_SCHEDULER.create_jobs (jobdef_array       => v_job_definition_array,
                            commit_semantics   => 'TRANSACTIONAL');

FOR v_instance IN 1 .. :count_instances
LOOP
  v_result :=
```

```
      DBMS_PIPE.receive_message (pipename   => :pipename,
                                 timeout    => const_timeout);
END LOOP;

V_result := DBMS_PIPE.remove_pipe (pipename => :pipename);
…
```

In diesem Fall muss man sich um die Prozesssynchronisierung selbst kümmern. Daher wird zunächst eine private Pipe angelegt, welche von den gestarteten Parallelprozessen zur Rückmeldung nach jeweiliger Erledigung der Arbeit verwendet wird. Anschließend wird ein Array mit der gewünschten Anzahl an Job-Definitionen erstellt, welches dem Scheduler zur Erzeugung der Jobs übergeben wird. Da den einzelnen Jobs kein Ausführungszeitpunkt mitgegeben wurde, werden sie direkt nach Erzeugung auch gestartet. Anschließend muss auf die Rückmeldungen der einzelnen Jobs gewartet werden, was in diesem Fall über eine Schleife geschieht. Zu guter Letzt wird die Pipe wieder entfernt.

Ein Laufzeitvergleich beider Varianten zeigt, dass der direkte Einsatz des Schedulers erheblich effizienter ist. In Tests wurden hierbei nur Overhead-Zeiten von wenigen Millisekunden gemessen, während bei der DBMS_PARALLEL_EXECUTE-Variante jeweils mindestens eine Sekunde mehr als die eigentliche Workload-Rechenzeit verging. Zu erklären ist dies damit, dass erheblich weniger Metadaten zu verwalten sind, und somit intern auch weniger Transaktionen ausgelöst werden. Darüber hinaus ist der Pipe-basierte Synchronisierungsmechanismus der Scheduler-Variante nahezu Overhead-los. Was nicht vergessen werden darf ist, dass DBMS_PARALLEL_EXECUTE ursprünglich nicht zur Parallelisierung von PL/SQL-Code und schon gar nicht für das Umfeld von OLTP-Anwendungen entwickelt wurde. Die Robustheit des Mechanismus, z.B. im Hinblick auf Wiederanlauffähigkeit nach Instanz-Neustarts oder auch die Möglichkeiten, Tasks zu stoppen und wieder zu starten, all das hat selbstverständlich seinen Preis. Sofern aus Sicht der Anwendung die auf diese Weise erreichbare Performance ausreicht, spricht nichts gegen den Einsatz dieses Packages. Ist hingegen das Verhältnis von Parallelisierungsoverhead zu tatsächlicher Rechenzeit zu ungünstig oder soll tatsächlich die minimal mögliche Laufzeit erreicht werden, so sollte die Wahl auf die Parallelisierung via Scheduler fallen.

Kontaktadresse:
Jan Gorkow
SD&C Solutions Development & Consulting GmbH
Mauerstr. 79
10117 Berlin

Telefon:	+49 (0) 30-44 32 32 0
Fax:	+49 (0) 30-44 32 32 99
E-Mail	jan.gorkow@sd-c.de
Internet:	www.sd-c.de