Tien manuals voor PL/SQL ontwikkelaar

Overzicht nieuwe features in Oracle 9i

Een PL/SQL ontwikkelaar heeft niet alleen maar met de taal PL/SQL te maken. De basis van de functionaliteit die een PL/SQL ontwikkelaar tot zijn beschikking heeft, is uiteraard de database zelf: de database objecten die kunnen worden aangemaakt en de functionaliteit die deze objecten bieden. Daarnaast zijn er de standaard (supplied) database packages en types. En natuurlijk SQL, de taal waarvan PL/SQL de procedurele extensie is (PL = procedural language). Voor een PL/SQL ontwikkelaar is het daarom belangrijk om de ontwikkelingen op deze gebieden ook goed in de gaten te houden. In dit artikel ligt de focus echter op de nieuwe mogelijkheden in PL/SQL zelf.

In de tijd van Oracle 7.3.4 had je als PL/SQL ontwikkelaar maar één manual echt nodig: de Application Developer's Guide. De mogelijkheden van de database zelf, SQL en PL/SQL waren beperkt en kende je uit je hoofd. Deze manual was de enige *reference* die je nodig had en bevatte onder andere informatie over alle supplied packages. In de versies 8.0, 8i en 9i is er enorm veel functionaliteit toegevoegd. De PL/SQL ontwikkelaar kan inmiddels gebruik maken van object *extensions* (geïntroduceerd in Oracle 8 en "voltooid" in 9i release 2), *native* ondersteuning van XML (Oracle9i release 2) en veel andere mogelijkheden. De introductie van Java in Oracle8i biedt de PL/SQL ontwikkelaar ook een nieuwe wereld aan mogelijkheden via Java Stored Procedures die vanuit PL/SQL gebruikt kunnen worden. De Application Developer's Guide is inmiddels vervangen door een reeks met meer dan 10 manuals (je hebt dus niet meer genoeg aan dat ene boek in je koffertje). Ook zijn SQL en PL/SQL dusdanig uitgebreid dat de betreffende manuals regelmatig als reference zullen moeten worden geraadpleegd.

New features

De laatste versie van 9i is "9i release 2", oftewel versie 9.2. De versie daarvoor was 9.0.1. Bij alle new features zal worden aangegeven in welke versie zij geïntroduceerd zijn: 9.0.1 of 9.2. De volgende nieuwe features zullen aan de hand van voorbeelden behandeld worden.

- Searched CASE statement en expressie (9.0.1)
- Associative arrays (9.2)
- INSERT en UPDATE met record variabelen (9.2)
- BULK COLLECT in table of records (9.2)
- Verbeteringen foutafhandeling bulk binding (9.0.1)
- Cursor expressies (9.0.1)
- PIPELINED table functions (9.0.1)
- MERGE statement (9.0.1)

Daarnaast zijn er nog een aantal andere interessante nieuwe mogelijkheden die het noemen waard zijn. De belangrijkste hiervan is de integratie van de SQL en PL/SQL parsers (9.0.1) waardoor bijvoorbeeld functies die alleen in SQL gebruikt konden worden nu ook in PL/SQL beschikbaar zijn. Sinds 9.0.1 kan PL/SQL code *natively compiled* worden in C om de performance van code waarin intensieve berekeningen worden uitgevoerd te "boosten". Over dit onderwerp is een paper te vinden op de website van Cumquat (www.cumquat.nl) waarmee Martijn Hinten de Editor's Choice Award heeft gewonnen op de ODTUG conferentie in 2002. Ook nieuw in 9.0.1 is de ondersteuning voor multi-level collections, de ondersteuning voor bulk binding in native dynamic SQL, nieuwe datatypes voor date en time waarden en verbeteringen aan de UTL_FILE supplied database package.

Searched CASE statement en expressie

Het CASE statement werd al eerder geïntroduceerd als handig alternatief voor geneste IF statements (en omdat het een taalelement is dat in de meeste programmeertalen voorkomt, maar nog ontbrak in PL/SQL). Nieuw in 9.0.1 is het *searched*

In de tijd van Oracle 7.3.4 had je als PL/SQL ontwikkelaar maar één manual echt nodig

CASE statement. Hieronder staat een voorbeeld van het typische gebruik van een *simple* CASE statement. Afhankelijk van de uitkomst van de expressie achter het CASE keyword (in het onderstaande voorbeeld is de expressie simpelweg een variabele) worden de statements achter de betreffende WHEN … THEN uitgevoerd.

```
DECLARE
   l_point_des_code   VARCHAR2(4) := 'ISLD';
BEGIN
   CASE l_point_des_code
   WHEN 'HUT' THEN
      dbms_output.put_line('HUT: Simple rural building for shelter');
   WHEN 'ISLD' THEN
      dbms_output.put_line('ISLAND: Body of land surrounded by water');
    WHEN 'ISTH' THEN
       dbms_output.put_line('ISTHMUS: Neck of land joining two larger
areas');
   ELSE
      dbms_output.put_line('ERROR: Unknown point description code.');
   END CASE;
END;
```

De beperking van het simple CASE statement is dat er slechts één expressie gebruikt kan worden. Een genest IF statement met verschillende expressies kan niet vereenvoudigd worden met een simple CASE statement. Met het searched CASE statement kan dit wel, zoals in onderstaand voorbeeld geïllustreerd wordt.

```
DECLARE
   l_foo    NUMBER(3,0) := 4;
   l_foo2   VARCHAR2(3) := 'XYZ';
   l_foo3   VARCHAR2(3);
BEGIN
   CASE
   WHEN l_foo < 4 AND l_foo2 = 'ABC' THEN
      l_foo3 := 'XYZ';
   WHEN l_foo = 4 AND l_foo2 = 'XYZ' THEN
      l_foo3 := 'ABC';
   WHEN l_foo > 4 AND l_foo2 not in ('ABC', 'XYZ') THEN
      l_foo3 := '___';
   ELSE
      l_foo3 := '???';
   END CASE;
   dbms_output.put_line(l_foo3);
END;
```

De expressies worden van bovenaf geëvalueerd. Van de eerste "gevonden" (vandaar de term "searched") expressie die true oplevert worden de statements uitgevoerd. Het CASE statement is beter leesbaar en sneller uitvoerbaar dan een IF statement, maar geeft een PL/SQL ontwikkelaar geen echte nieuwe mogelijkheden. Dit is wel het geval bij de CASE expressie. Dit is een handige verkorte manier om een veel voorkomend pattern te coderen, namelijk het toekennen van een waarde afhankelijk van de uitkomst van een expressie. Hieronder staat een voorbeeld waarin het eerder gegeven voorbeeld van een simple CASE statement als een CASE expressie herschreven is. De variabele l_point_des_name krijgt nu een waarde afhankelijk van de uitkomst van de expressie l_point_des_code.

```
DECLARE
   l_point_des_code   VARCHAR2(4) := 'ISLD';
   l_point_des_name   VARCHAR2(40);
BEGIN
   l_point_des_name := CASE l_point_des_code
      WHEN 'HUT' THEN
         'HUT: Simple rural building for shelter'
      WHEN 'ISLD' THEN
         'ISLAND: Body of land surrounded by water'
      WHEN 'ISTH' THEN
         'ISTHMUS: Neck of land joining two larger areas'
      ELSE
         'ERROR: Unknown point description code.'
      END;
   dbms_output.put_line(l_point_des_name);
END;
```

CASE expressies kunnen ook gebruikt worden in een SQL statement als alternatief voor de veel geprezen, maar vaak veel "trucs" vereisende, DECODE functie. In het onderstaande voorbeeld staan twee SQL query die hetzelfde resultaat opleveren, de eerste maakt gebruik van DECODE, de tweede van een CASE expressie. De CASE expressie is veel begrijpelijker en maakt dit soort SQL queries beter onderhoudbaar.

```
SELECT DECODE(SIGN(LENGTH(name) - 10), 1, code, name)
FROM   land_districts

SELECT CASE
       WHEN LENGTH(name) > 10 THEN code
       ELSE name
       END
FROM   land_districts
```

In 9.0.1 zijn ook twee interessante functies toegevoegd die specifieke patterns in CASE expressies implementeren: COALESCE en NULLIF.

Associative arrays

PL/SQL tables, of index-by tables, kunnen al gebruikt worden sinds versie 7. Een index-by table is een array waarbij een numerieke waarde gebruikt wordt als subscript waarde. De subscript waarde duidt de plek (of positie) aan in de array waar een waarde is opgeslagen. De twee datatypes die gebruikt kunnen worden voor de subscript waarde zijn BINARY_INTEGER en PLS_INTEGER. Deze datatypes hebben allebei hetzelfde bereik,

maar verschillen iets in semantiek. Omdat bewerkingen met PLS_INTEGER sneller zijn is het aanbevolen om van dit datatype gebruik te maken.

In 9.2 is het nu ook mogelijk om VARCHAR2 (of een sub-type hiervan) als datatype voor de subscript waarde te gebruiken. De plek waar een waarde in de array wordt opgeslagen kan dan worden aangeduid door middel van een string. Op deze manier kan een, uit Java bekende, hash table, worden geïmplementeerd. In het voorbeeld hieronder worden de namen van de land districts in de associative array geplaatst op een plek aangeduid met de code van het betreffende land district. De waarden kunnen vervolgens opgevraagd worden via deze code.

```
DECLARE
   TYPE t_land_districts_tab IS
   TABLE OF VARCHAR2(20)
   INDEX BY VARCHAR2(2);
   l_land_districts_tab   t_land_districts_tab;
BEGIN
   l_land_districts_tab('AK') := 'North Auckland';
   l_land_districts_tab('BM') := 'Marlborough';
   …
   l_land_districts_tab('WN') := 'Wellington';
   dbms_output.put_line(l_land_districts_tab('DN'));
   dbms_output.put_line(l_land_districts_tab('GS'));
   dbms_output.put_line(l_land_districts_tab('NP'));
END;
```

In het bovenstaande voorbeeld is een table gebaseerd op één waarde (VARCHAR2(20)) gebruikt, maar de array kan uiteraard ook gebaseerd zijn op een record.

INSERT en UPDATE met record variabelen

Bij het selecteren van een rij met een SELECT INTO statement of met een FETCH (uit een cursor) kon altijd al gebruik worden gemaakt van een record variabele om de kolomwaarden van de rij in op te slaan. Bij het gebruik van een cursor wordt de record variabele meestal gedeclareerd met behulp van %ROWTYPE, zoals in onderstaand voorbeeld.

```
DECLARE
   l_land_district_row   land_districts%ROWTYPE;
BEGIN
   SELECT code
   ,      name
   INTO   l_land_district_row
   FROM   land_districts
   WHERE  code = 'AK';
   dbms_output.put_line(l_land_district_row.code);
   dbms_output.put_line(l_land_district_row.name);
END;
```

In 9.2 is nu ook mogelijk om bij een INSERT en UPDATE gebruik te maken van een record variabele. Hieronder staat een voorbeeld van een INSERT.

```
DECLARE
   l_land_district_row   land_districts%ROWTYPE;
BEGIN
   l_land_district_row.code := 'AK';
   l_land_district_row.name := 'North-Auckland';
   INSERT INTO land_districts
   VALUES l_land_district_row;
END;
```

Het datatype van de record variabele moet uiteraard overeenkomen met de tabel waarop de INSERT wordt uitgevoerd. Waar het voorheen noodzakelijk was om na VALUES iedere afzonderlijk waarde binnen het record expliciet uit te schrijven (l_land_district_row.code, l_land_district_row.name) volstaat het nu dus om de record variabele aan te geven. Niet-geïnitialiseerde waarden in de record variabele worden als een null-waarde gezien. Bij een UPDATE wordt gebruik gemaakt van SET ROW om een bestaande rij de waarden uit een record variabele te geven.

```
DECLARE
   l_land_district_row   land_districts%ROWTYPE;
BEGIN
   l_land_district_row.code := 'AK';
   l_land_district_row.name := 'North Auckland';
   UPDATE land_districts
   SET ROW = l_land_district_row
   WHERE code = 'AK';
END;
```

BULK COLLECT in table of records

Ook nieuw in 9.2 is de mogelijkheid om in één keer (= BULK) een set, of collection, van waarden te selecteren (= COLLECT) in een table variabele gebaseerd op een record. BULK COLLECT was al eerder geïntroduceerd maar vereiste een aparte table variabele (gebaseerd op één waarde, zoals NUMBER of DATE) per kolom in de select list. In het onderstaande voorbeeld worden de 12 rijen in de tabel in één keer geselecteerd in de table variabele l_land_districts_tab.

In het voorbeeld vindt er maar één context switch plaats: de twaalf rijen worden in één keer toegevoegd aan de tabel

```
DECLARE
  CURSOR c_land_districts IS
  SELECT dst.code
  ,      dst.name
  FROM   land_districts  dst;
  TYPE land_districts_tab_t IS
  TABLE OF land_districts%ROWTYPE;
  l_land_districts_tab  land_districts_tab_t;
BEGIN
  OPEN c_land_districts;
  FETCH c_land_districts
  BULK COLLECT INTO l_land_districts_tab;
  CLOSE c_land_districts;
  FOR i IN 1..l_land_districts_tab.LAST LOOP
    dbms_output.put_line(l_land_districts_tab(i).name);
  END LOOP;
END;
```

In het voorbeeld hierboven wordt de BULK COLLECT gedaan in een FETCH statement. Ook in een SELECT INTO statement en in de RETURNING clause van een UPDATE of DELETE statement kan van BULK COLLECT gebruik worden gemaakt.

Verbeteringen foutafhandeling bulk binding

Al langer kan gebruik worden gemaakt van het FORALL statement om bulk binding toe te passen bij een INSERT, UPDATE of DELETE op basis van waarden in table variabelen. Met bulk binding worden kostbare context switches voorkomen tussen PL/SQL en SQL wanneer in een loop telkens waarden uit table variabelen worden gehaald (PL/SQL context) om in een SQL statement te gebruiken (SQL context). In het voorbeeld hieronder worden de codes en namen van de land districts eerst in twee afzonderlijke (opmerking: dit is nu nog vereist) table variabelen geplaatst en vervolgens met FOR ALL via bulk binding geINSERT. In dit voorbeeld vindt er maar één context switch plaats: de twaalf rijen worden in één keer toegevoegd aan de tabel.

```
DECLARE
   TYPE dst_code_tab_t IS
   TABLE OF VARCHAR2(2)
   INDEX BY BINARY_INTEGER;
   TYPE dst_name_tab_t IS
   TABLE OF VARCHAR2(20)
   INDEX BY BINARY_INTEGER;
   l_dst_code_tab   dst_code_tab_t;
   l_dst_name_tab   dst_name_tab_t;
BEGIN
   l_dst_code_tab(1) := 'AK';
   l_dst_name_tab(1) := 'North Auckland';
   l_dst_code_tab(2) := 'BM';
   l_dst_name_tab(2) := 'Marlborough';
   …
   l_dst_code_tab(12) := 'WN';
   l_dst_name_tab(12) := 'Wellington';
   FORALL i IN l_dst_code_tab.FIRST..l_dst_code_tab.LAST
      INSERT INTO land_districts
      VALUES (l_dst_code_tab(i), l_dst_name_tab(i));
END;
```

Indien er een DML-fout optreedt bij bulk binding (bijvoorbeeld bij het toevoegen van de vijfde rij) dan wordt er een rollback uitgevoerd waarmee de eerder uitgevoerde DML-acties binnen het FOR ALL statement worden teruggedraaid (bijvoorbeeld de eerste vier INSERTs). Dit gedrag is erg vervelend omdat de rijen waarmee niets aan de hand is niet verwerkt kunnen worden.

In 9.0.1 wordt hier een oplossing voor geboden. Met SAVE EXCEPTIONS kan in het FORALL statement aangegeven worden dat na een eventuele DML-fout voor een bepaalde rij de verwerking van de overige rijen gewoon verder moet gaan. De rijen waarvoor een fout optreedt worden automatisch bewaard in de impliciete collection SQL%BULK_COLLECT. Deze collection kan na de verwerking van alle rijen worden benaderd om na te gaan voor welke rijen een DML-fout is opgetreden. In het onderstaande voorbeeld is het hierboven gebruikte voorbeeld aangepast en bevatten nu de rijen (in de twee afzonderlijk table variabelen) 2, 6 en 12 fouten (respectievelijk een te lange naam en schendingen van de primary key en unique key constraints op de tabel). De exception e_dml_errors is toegevoegd en geïnitialiseerd, de SAVE EXCEPTIONS clause is in het FORALL statement opgenomen en er is nu een exception handler aanwezig.

```
DECLARE
   …
   e_dml_errors   EXCEPTION;
   PRAGMA EXCEPTION_INIT(e_dml_errors, -24381);
BEGIN
   l_dst_code_tab(1) := 'AK';
   l_dst_name_tab(1) := 'North Auckland';
   l_dst_code_tab(2) := 'BM';
   l_dst_name_tab(2) := 'Marlborough_____________';
   …
   l_dst_code_tab(5) := 'GS';
   l_dst_name_tab(5) := 'Gisborne';
   l_dst_code_tab(6) := 'GS';
   …
   l_dst_name_tab(11) := 'Taranaki';
   l_dst_code_tab(12) := 'WN';
   l_dst_name_tab(12) := 'Taranaki';
   FORALL i IN l_dst_code_tab.FIRST..l_dst_code_tab.LAST
   SAVE EXCEPTIONS
      INSERT INTO land_districts
      VALUES (l_dst_code_tab(i), l_dst_name_tab(i));
   EXCEPTION
      WHEN e_dml_errors THEN
        FOR i in 1..SQL%BULK_EXCEPTIONS.COUNT LOOP
           dbms_output.put_line(  'Error in record '

||TO_CHAR(SQL%BULK_EXCEPTIONS(i).ERROR_INDEX)
                                  ||': '
                                  ||SQLERRM(-
SQL%BULK_EXCEPTIONS(i).ERROR_CODE));
        END LOOP;

END;
```

In de exception handler wordt de SQL%BULK_COLLECT table doorlopen en de hierin opgeslagen informatie wordt getoond. Dit geeft het volgende resultaat:

```
Error in record 2: ORA-01401:
  Ingevoegde waarde is te groot voor kolom.
Error in record 6: ORA-00001:
  Schending van UNIQUE-voorwaarde (constraint .).
Error in record 12: ORA-00001:
  Schending van UNIQUE-voorwaarde (constraint .).
```

Met de mogelijkheid om deze informatie te achterhalen kan de gewenste foutafhandeling gerealiseerd worden.

Cursor expressies

Voor 9i was het al mogelijk om in SQL gebruik te maken van een cursor expressie:

```
SELECT d.dname
,      CURSOR(SELECT e.ename
              ,      e.job
              FROM   emp e
              WHERE  e.deptno = d.deptno) employees
FROM   dept d
```

In SQL*Plus is wordt het resultaat van deze query wat vreemd getoond, maar wat de query in feite oplevert zijn rijen waarbij de tweede kolom een collection is. Dat wil zeggen, de tweede kolom bevat zelf weer een aantal rijen, namelijk de namen en functies van de werknemers binnen de afdeling. De cursor-expressie wordt ook vaak een sub-query genoemd, maar dan dus een sub-query in de select list van de SQL query en niet een sub-query in de WHERE clause. In SQL kon de cursor expressie dus al voor 9i gebruikt worden. In PL/SQL was er echter nog geen ondersteuning. Het was dus niet mogelijk om een een cursor te maken van een query met een cursor expressie en hier rijen uit te selecteren in PL/SQL variabelen. In 9.0.1 wordt dit wel ondersteund. Het onderstaande voorbeeld laat dit zien.

CASE expressies kunnen ook gebruikt worden in een SQL statement als alternatief voor de vaak veel "trucs" vereisende DECODE functie

```
DECLARE
   CURSOR c_departments
   IS
   SELECT d.dname
   ,      CURSOR(SELECT e.ename
                 ,      e.job
                 FROM   emp e
                 WHERE  e.deptno = d.deptno) employees
   FROM   dept d;
   l_department_name   dept.dname%TYPE;
   l_employees          SYS_REFCURSOR;
   l_employee_name     emp.ename%TYPE;
   l_employee_job      emp.job%TYPE;
BEGIN
   OPEN c_departments;
   LOOP
      FETCH c_departments
      INTO  l_department_name, l_employees;
      EXIT WHEN c_departments%NOTFOUND;
      dbms_output.put_line(l_department_name);
      LOOP
         FETCH l_employees
         INTO  l_employee_name, l_employee_job;
         EXIT WHEN l_employees%NOTFOUND;
         dbms_output.put_line(l_employee_name||'
('||l_employee_job||')');
      END LOOP;
   END LOOP;
   CLOSE c_departments;
END;
```

De cursor c_departments wordt geopend en in een loop wordt voor iedere geselecteerde rij de naam van de afdeling in variabele l_department_name gezet en de collection (met namen en functies van de werknemers van de afdeling) in de variabele l_employees. Deze laatste variabele is van het standaard REF CURSOR type SYS_REFCURSOR. Dit is zogenaamde weak REF CURSOR die voor cursors gebruikt kan worden van verschillende types (dat wil zeggen, voor cursors met verschillende aantallen kolommen en datatypes). Door van dit standaard weak REF CURSOR type gebruik te maken hoeft er geen type gedefinieerd te worden voor de collection. Er wordt een REF CURSOR type gebruikt omdat de collection benaderd moet worden als een cursor variabele. De informatie over de werknemers wordt dan ook in een loop doorlopen. Hierbij hoeft de cursor variabele niet expliciet geopend te worden, deze is al geopend. Het bovenstaande voorbeeld geeft het volgende resultaat:

```
ACCOUNTING
CLARK (MANAGER)
KING (PRESIDENT)
MILLER (CLERK)
RESEARCH
SMITH (CLERK)
JONES (MANAGER)
```

```
SCOTT (ANALYST)
ADAMS (CLERK)
FORD (ANALYST)
SALES
ALLEN (SALESMAN)
WARD (SALESMAN)
MARTIN (SALESMAN)
BLAKE (MANAGER)
TURNER (SALESMAN)
JAMES (CLERK)
OPERATIONS
```

Het vorige voorbeeld laat zien dat de cursor expressie een geopende cursor variabele (REF CURSOR) oplevert. Omdat het mogelijk is om stored procedures of functions te maken die een REF CURSOR als parameter hebben, zou het mogelijk moeten zijn om een aanroep van een dergelijke procedure of function te doen met een cursor expressie, zoals hieronder.

```
BEGIN
   do_something(CURSOR(SELECT ename, job FROM EMP));
END;
```

Dit is echter alleen toegestaan (ook in 9.2) in de aanroep van functions in top-level SQL statements. Het onderstaande voorbeeld laat dit zien. De function COUNT_CLERKS telt het aantal werknemers met functie "CLERK" in de collection die als cursor variabele wordt doorgegeven.

```
CREATE OR REPLACE
PACKAGE hrm
IS
   TYPE t_emp_cursor
   IS REF CURSOR RETURN emp%ROWTYPE;
   FUNCTION count_clerks( p_employees   IN   t_emp_cursor)
   RETURN NUMBER;
END;

CREATE OR REPLACE
PACKAGE BODY hrm
IS
   FUNCTION count_clerks( p_employees   IN   t_emp_cursor)
   RETURN NUMBER
   IS
     r_employee       emp%ROWTYPE;
     l_clerk_count    NUMBER := 0;
   BEGIN
     LOOP
       FETCH p_employees INTO r_employee;
       EXIT WHEN p_employees%NOTFOUND;
       IF r_employee.job = 'CLERK' THEN
         l_clerk_count := l_clerk_count + 1;
       END IF;
     END LOOP;
     RETURN l_clerk_count;
   END count_clerks;
END;
```

Deze function kan vervolgens gebruikt worden in de volgende SQL query:

```
SELECT d.dname
FROM   dept d
WHERE  hrm.count_clerks(CURSOR(SELECT *
                               FROM   emp e
                               WHERE  e.deptno = d.deptno)) > 1
```

Uiteraard kan deze query, die alle afdeling selecteert met meer dan één CLERK, ook (en veel beter) helemaal in SQL geschreven worden, maar dit voorbeeld laat wel goed zien wat de mogelijkheden zijn. Indien een complexe conditie niet in SQL kan worden uitgedrukt, of wanneer dit te ingewikkeld wordt, kan de conditie in PL/SQL geprogrammeerd worden. Een dergelijke function kan gebruikt worden in de WHERE clause, in de ORDER BY clause of in een table function in de FROM clause.

PIPELINED table functions

Ook table functions stonden al ter beschikking van de PL/SQL ontwikkelaar voor 9i. Een table function is een stored function die een collection retourneert en gebruikt kan worden in een SQL query. De table function in het voorbeeld hieronder retourneert de twaalf, inmiddels bekende, land districts als een waarde van het table type land_districts_table_type.

```
CREATE OR REPLACE
TYPE land_district_row_type
AS OBJECT (code VARCHAR2(2), name VARCHAR2(20))

CREATE OR REPLACE
TYPE land_districts_table_type
AS TABLE OF land_district_row_type

CREATE OR REPLACE
FUNCTION get_land_districts
RETURN land_districts_table_type DETERMINISTIC
IS
   l_land_districts   land_districts_table_type;
BEGIN
   l_land_districts := land_districts_table_type(
      land_district_row_type('AK', 'North Auckland')
    , land_district_row_type('BM', 'Marlborough')
    …
    , land_district_row_type('WN', 'Wellington'));
   RETURN l_land_districts;
END;
```

Deze stored function kan op de volgende manier gebruikt worden in een SQL query:

```
SELECT name
FROM   TABLE(get_land_districts)
WHERE  code like '_N'
ORDER BY length(name)
```

Indien er een DML-fout optreedt bij bulk binding wordt met een rollback de eerder uitgevoerde DML-acties binnen het FOR ALL statement teruggedraaid

Een verzameling (collection) gegevens die is opgebouwd in PL/SQL kan dus met SQL bevraagd worden, wat interessante en krachtige mogelijkheden biedt. Het keyword DETERMINISTIC geeft aan dat de resultaten van de function in de cache opgeslagen mogen worden. Hierbij is het de verantwoordelijkheid van de programmeur dat de resultaten inderdaad niet zullen veranderen in de loop van een session.

Nieuw in 9.0.1 zijn PIPELINED table functions. Bij een normale table function wordt de verzameling gegevens die in de function wordt opgebouwd pas doorgegeven aan de SQL engine als de complete verzameling in zijn geheel is opgebouwd. Bij een PIPELINED table function kan een afzonderlijke rij in de verzameling gegevens alvast doorgegeven worden, zodra deze rij is opgebouwd. Het doorgeven van een rij gebeurd met PIPE ROW zoals in het onderstaande voorbeeld wordt getoond.

```
CREATE OR REPLACE
FUNCTION get_land_districts
RETURN land_districts_table_type PIPELINED
IS
   l_land_district_row   land_district_row_type;
BEGIN
   FOR i IN 1..12 LOOP
      l_land_district_row := CASE i
         WHEN  1 THEN land_district_row_type('AK', 'North Auckland')
         WHEN  2 THEN land_district_row_type('BM', 'Marlborough')
         …
         WHEN 12 THEN land_district_row_type('WN', 'Wellington')
         END;
         PIPE ROW(l_land_district_row);
   END LOOP;
   RETURN;
END;
```

Opvallend is dat hoewel de function aangeeft dat een waarde van het datatype land_districts_table_type wordt geretourneerd (net als de niet-PIPELINED versie), er in feite geen enkele waarde geretourneerd wordt met het RETURN statement (zonder expressie). In plaats daarvan worden dus de afzonderlijke rijen met PIPE ROW doorgegeven als een waarde van het type l_land_district_row. In dit gekunstelde voorbeeld zal de performancewinst waarschijnlijk niet merkbaar zijn, maar in veel andere situaties kan de PIPELINED table function goed uitkomst bieden. Denk bijvoorbeeld aan een situatie waar de rijen worden verkregen uit een XML document dat met een webservice wordt opgehaald. Een table function kan ook parallel enabled worden zodat geprofiteerd kan worden van de parallel query ondersteuning op multi-processor machines.

MERGE statement

Nieuw in 9.0.1 is het MERGE statement. Dit statement combineert een INSERT en UPDATE statement en vindt zijn toepassing vooral in datawarehousing scenario's. Om het gebruik van het MERGE statement te illustreren wordt eerst de volgende tabel aangemaakt waarin de bonussen van werknemers worden opgeslagen:

```
CREATE TABLE bonuses
( empno    NUMBER(4,0)   NOT NULL
, job      VARCHAR2(9)   NOT NULL
, bonus    NUMBER(7,2)   NOT NULL
, CONSTRAINT bns_pk PRIMARY KEY (empno)
)
```

Vervolgens wordt aan CLERKs een bonus van 500 gegeven en aan ANALYSTs een bonus van 1000. De vulling van de tabel is dan als volgt:

```
EMPNO JOB              BONUS
_____ _____ _____

      7369 CLERK             500
      7788 ANALYST          1000
      7876 CLERK             500
      7900 CLERK             500
      7902 ANALYST          1000
      7934 CLERK             500
```

De volgende stap is om iedere werknemer in afdeling 30 een bonus te geven ter grootte van de helft van zijn of haar salaris. Dit betekent dat voor werknemers die al een bonus hebben de bonus opgehoogd moet worden (UPDATE) en dat voor de overige werknemers een bonus moet worden aangemaakt (INSERT). Een dergelijke actie zou voorheen procedureel in PL/SQL geprogrammeerd moeten worden, maar kan nu dus met een enkel MERGE statement afgehandeld worden. Het onderstaande statement voert deze actie uit.

```
MERGE INTO bonuses b
USING (SELECT empno
       ,      job
       ,      sal
       FROM   emp
       WHERE  deptno = 30) e
ON (b.empno = e.empno)
WHEN MATCHED THEN UPDATE SET b.bonus = b.bonus + e.sal / 2
WHEN NOT MATCHED THEN INSERT VALUES (e.empno, e.job, e.sal/ 2)
```

Na MERGE INTO wordt de tabel aangegeven waarop de actie moet plaatsvinden. De USING clause bevat de query die de gegevens moet leveren voor de UPDATE of INSERT actie.
In het voorbeeld worden in deze query de benodigde gegevens opgehaald van de werknemers in afdeling 30. Vervolgens wordt na het ON keyword de join conditie gegeven om de rijen in de tabel waarop de MERGE wordt uitgevoerd te "matchen" met de rijen in de query. Indien er een "match" is (dat wil zeggen, indien er al een rij bestaat) moet er een UPDATE actie worden uitgevoerd, anders een INSERT actie. Bij het uitvoeren van het bovenstaande statement geeft SQL*Plus de volgende feedback:

```
6 rijen zijn samengevoegd.
```

De tabel bevat daarna deze data:

```
EMPNO JOB              BONUS
_____ _____- _____

      7369 CLERK             500
      7788 ANALYST          1000
      7876 CLERK             500
      7900 CLERK             975
      7902 ANALYST          1000
      7934 CLERK             500
      7499 SALESMAN          800
      7521 SALESMAN          625
      7654 SALESMAN          625
      7698 MANAGER          1425
      7844 SALESMAN          750
```

Er zijn dus vijf rijen toegevoegd en er is een rij geUPDATE voor de CLERK 7900.

Erwin Groenendal
is technisch directeur van Cumquat Information Technology. Cumquat levert oplossingen op het gebied van Self-Service Applications, Portals, Web Services en B2B Integration en maakt daarbij gebruik van Oracle, XML en Java technologie. Erwin heeft meer dan 10 jaar ervaring met Oracle en is bereikbaar via erwin.groenendal@cumquat.nl.