12*c*-New-Features im Praxis-Einsatz

Roger Troller, Trivadis AG

Zwei Jahre sind vergangen, seit Oracle die Version 12*c* ihrer Datenbank ausgeliefert hat. Die ersten Unternehmen haben den Schritt zum produktiven Einsatz der jüngsten Datenbank-Version gewagt und es ist an der Zeit, ein erstes Fazit zu den neuen Funktionalitäten im Bereich „SQL & PL/SQL" zu ziehen.

Die folgende Zusammenstellung ist eine persönliche Einschätzung des Autors zu den neuen Funktionalitäten, basierend auf Erfahrungen in verschiedenen Projekten.

Invisible Columns

Mit der Möglichkeit, Spalten zu verstecken, wird ein älteres Feature von Oracle (virtuelle Spalten) enorm aufgewertet. Bisher musste man beim Einsatz von virtuellen Spalten mit verschiedenen Problemen kämpfen wie zum Beispiel:

- Virtuelle Spalten sind in „%ROWTYPE"-Variablen enthalten, wodurch keine „ROW"-Operationen möglich sind *(siehe Listing 1 und 2)*
- Insert ohne Angabe der Spalten (Ausschluss der virtuellen Spalten) führt zu Fehlern, wenn die Ziel-Tabelle virtuelle Spalten enthält

Die Möglichkeit, eine virtuelle Spalte unsichtbar zu definieren, umgeht diesen Fehler.

Ein wenig störend an den unsichtbaren Spalten ist, dass man keine „%TYPE"-Deklaration auf einer solchen Spalte durchführen kann *(siehe Listing 3)*.

```
ALTER TABLE emp ADD total_income
   GENERATED ALWAYS AS (nvl(sal,0) + nvl(comm,0));

CREATE PROCEDURE upd_emp (in_emp_row IN emp%rowtype)
is
BEGIN
   UPDATE emp
      SET ROW = in_emp_row
   where empno = in_emp_row.empno;
END upd_emp;
/

DECLARE
   r_emp emp%rowtype;
BEGIN
   SELECT *
      INTO r_emp
      FROM emp
      WHERE empno = 7839; -- King

   r_emp.sal := r_emp.sal * 1.1;

   upd_emp(in_emp_row => r_emp);
end;
/
ORA-54017: UPDATE operation disallowed on virtual columns
```

Listing 1: „UPDATE SET ROW" mit einer virtuellen sichtbaren Spalte

Row Limiting Clause

Die Row Limiting Clause stellt eine wesentliche Vereinfachung im Bereich der Paginierung und der „Top-N"-Abfragen dar. Es ist nicht so, dass diese Problemstellungen vorher nicht lösbar waren, aber die mehrstufigen Ansätze unter Verwendung von „ROWNUM" *(siehe Listing 4)* oder analytischen Funktionen *(siehe Listing 5)* werden durch die Row Limiting Clause überflüssig.

Ein Vergleich herkömmlicher Lösungen, um die zweite Seite eines Produktkatalogs (Produkt 11-20, nach „product_id" sortiert) anzuzeigen, veranschaulicht mit einer 12*c*-Implementation, wie viel einfacher sich solche Abfragen zukünftig mit dem neuen Feature definieren lassen. Mit der Row Limiting Clause fällt die Notwendigkeit der geschachtelten Abfragen weg. Es werden eine Sortierung, eine Anzahl zu überspringende Datensätze sowie eine Anzahl der zu lesenden Datensätze definiert *(siehe Listing 6)*.

Neben der verbesserten Lesbarkeit des SQL-Statements erhalten wir mit der Row Limiting Clause auch die Möglichkeit, prozentuale „Top-N"-Abfragen (die ersten 5 Prozent der Datensätze anzeigen, *siehe Listing 7)* sowie eine Behandlung von Duplikaten *(siehe Listing 8)* an der Anzeigegrenze durchzuführen.

Zusammengefasst vereinfacht die Row Limiting Clause die Formulierung von „Top-N"- und Paginierungs-Abfragen wesentlich und erhöht dadurch sowohl die Wart- wie auch die Lesbarkeit solcher Queries.

Pattern Matching

Die neue Mustererkennungs-Funktionalität hat angesichts der Tatsache, dass in den Daten an verschiedensten Orten Muster zu finden sind, das Potenzial, eine Nische zu besetzen.

Beispiele für ein Muster in den Daten sind:

- *Finanzsektor*
  Aktienkurse, Fraud-Detection, Geldwäsche
- *Handel*
  Kaufgewohnheit, Preisentwicklung
- *Telekom*
  Netzaktivität, Ticketing
- *Verkehr*
  Sensorketten, Prozessketten

Viele der über die „Pattern Matching“-Klausel lösbaren Fragestellungen werden heute mit Joins, Subqueries, Pipelined Functions etc. gelöst. Dies dürfte erfahrungsgemäß auch für einige Jahre so bleiben, da neue Ansätze es immer schwer haben, Nutzer und Fürsprecher zu finden. Ein Einsatzgebiet der „Pattern Matching“-Klausel ist das Suchen nach fortlaufenden Bereichen beziehungsweise das Auffinden von Lücken. *Listing 9* zeigt die herkömmliche „Tabibitosan“-Lösung und *Listing 10* die gleiche Problemstellung, gelöst über die neue „MATCH_RECOGNIZE“-Klausel.

Beim Pattern Matching sind die Muster mit dem Keyword „PATTERN“ beschrieben. Im Beispiel wird definiert, dass einem Start „0 – n“ Weiterführungen folgen „[PATTERN (strt cont*)]“. Die Beschreibung des Musters ist syntaktisch an reguläre Ausdrucke angelehnt. Später wird in der „MATCH_RECOGNIZE“-Klausel definiert, woran man eine Weiterführung erkennt „[DEFINE cont AS id = PREV(id) + 1]“. Die Aussage hinter dieser Definition ist, dass eine Weiterführung dann vorliegt, wenn der aktuelle Wert von „id“ den Wert der „id“-Spalte des vorhergegangenen Datensatzes um „1“ übersteigt. Ebenfalls kann definiert werden, welche Werte („MEASURES“) ermittelt und wie viele Resultatzeilen pro gefundenes Muster zurückgegeben werden sollen „[ONE ROW PER MATCH]“.

Seine wahren Stärken kann „MATCH _RECOGNIZE“ ausspielen, wenn es darum geht, komplexere Muster zu erkennen und zusammenzuführen. Mittelfristig dürfte diese Klausel jedoch ein Schattendasein führen, wie etwa die „Model“- oder „Pivot“-Klauseln, die zwar vorhanden, aber kaum bekannt sind.

```
ALTER TABLE emp ADD total_income INVISIBLE
   GENERATED ALWAYS AS (nvl(sal,0) + nvl(comm,0));

DECLARE
   r_emp emp%rowtype;
BEGIN
      SELECT *
          INTO r_emp
          FROM emp
      WHERE empno = 7839; -- King

   r_emp.sal := r_emp.sal * 1.1;

   upd_emp(in_emp_row => r_emp);
END;
/
anonymous block completed
```

Listing 2: „UPDATE SET ROW“ mit einer virtuellen unsichtbaren Spalte

```
ALTER TABLE emp MODIFY comm invisible;

DECLARE
   l_comm emp.comm%type;
BEGIN
   NULL;
END;
/

Error report -
ORA-06550: line 2, column 15:
PLS-00302: component 'COMM' must be declared
```

Listing 3: „%TYPE“-Deklaration gegen eine unsichtbare Spalte

```
SELECT y.product_id, translated_name
   FROM (SELECT ROWNUM rn
              ,x.*
          FROM (SELECT product_id
                      ,translated_name
                  FROM product_descriptions
              WHERE language_id = 'US'
              ORDER BY product_id) x
      ) y
 WHERE y.rn BETWEEN 11 AND 20
/
```

Listing 4: Paginierung mittels „ROWNUM“

```
SELECT y.product_id, translated_name
   FROM (SELECT product_id
          ,translated_name
          ,row_number() OVER (ORDER BY product_id) rn
      FROM product_descriptions
      WHERE language_id = 'US') y
 WHERE y.rn BETWEEN 11 AND 20
 /
```

Listing 5: Paginierung durch analytische Funktion

```
SELECT product_id
      ,translated_name
  FROM product_descriptions
 WHERE language_id = 'US'
 ORDER BY product_id
 OFFSET 10 ROWS FETCH NEXT 10
ROWS ONLY
/
```

Listing 6: Pagination mit Row Limiting Clause

```
SELECT product_id
      ,translated_name
  FROM product_descriptions
 WHERE language_id = 'US'
 ORDER BY product_id
FETCH FIRST 5 PERCENT ROWS ONLY
/
```

Listing 7: Prozentuale „Top-N“-Abfrage

UTL_CALL_STACK

Mit „UTL_CALL_STACK" erhalten wir die Möglichkeit, auf den Namen der Program Unit und der darin aufgerufenen Prozedur/Funktion bis hinunter zu lokalen Prozeduren und Funktionen zuzugreifen. Die Verwaltung geschieht innerhalb von Oracle in einem mehrdimensionalen Feld, wobei auf der Y-Achse die Aufrufsequenz (letzte aufgerufene Program Unit auf Position 1, die vorherigen mit einer entsprechend höheren Nummer) und auf der X-Achse die Program-Unit-Struktur abgelegt ist, in der wir uns befinden *(siehe Listing 11)*. In diesem Beispiel würde der Call-Stack zu dem Zeitpunkt, zu dem wir uns in der lokalen Prozedur befinden, wie in *Listing 12* aussehen.

Wichtig dabei ist, dass sich die aktuelle Position innerhalb des Call-Stacks immer auf Position 1 befindet. Dadurch lässt sich einfach eine Library-Funktion erzeugen, die es erlaubt, die aktuelle Position als String zurückzugeben *(siehe Listing 13)*. Als aktuelle Position wird dabei der Ort zurückgegeben, an dem die Library-Funktion aufgerufen wird (somit Level 2).

Diese Funktion macht sich das Wissen zunutze, dass sich der Aufrufer der Funktion im Call-Stack auf Level 2 befindet. Dadurch kann nun innerhalb der eigenen Program Units diese kleine Library-Funktion aufgerufen werden, um die aktuelle Position in Form eines Strings zu erhalten *(siehe Listing 14)*.

Der Call-Stack funktioniert auch auf gewrapptem PL/SQL-Code und leistet gute Dienste, wenn es darum geht, ein Logging oder ein applikationsspezifisches Tracing durchzuführen, bei dem der Zugriff auf den Namen der aktuell ausgeführten Program Unit erforderlich ist.

ACCESSIBLE_BY Clause

Im Oracle Magazine, Ausgabe 1/2015, hat Steven Feuerstein aufgezeigt, wie die „ACCESSIBLE_BY"-Klausel verwendet werden kann, wenn man bestehenden PL/SQL-Code („Packages") restrukturieren muss. In dem Artikel „When Packages Need to Lose Weight" beschrieb er die Trennung eines Package in zwei – unter Beibehaltung der zuvor privaten Methoden, die nun neu von zwei verschiedenen (neuen) Packages benutzt, aber nur einmal definiert und gleichzeitig vor anderen unbe-

```
SELECT product_id
      ,translated_name
  FROM product_descriptions
 WHERE language_id = 'US'
 ORDER BY product_id
FETCH FIRST 5 PERCENT ROWS ONLY
/

Listing 7: Prozentuale „Top-N"-Abfrage

SELECT first_name, last_name, salary
  FROM employees
 ORDER BY salary DESC
 FETCH FIRST 2 ROWS WITH TIES
/
FIRST_NAME           LAST_NAME                 SALARY
-------------------- ------------------------- ----------
Steven               King                      24000
Neena                Kochhar                   17000
Lex                  De Haan                   17000
```

Listing 8: Behandlung von Duplikaten, Anzeige von drei Datensätzen bei „FIRST 2"

```
WITH data (id) AS (SELECT column_value
                    FROM TABLE(nc(1,2,3,5,6,7,11,12,13)))
SELECT MIN(id) start_of_range
      ,MAX(id) end_of_range
  FROM (SELECT id
              ,id - ROW_NUMBER() OVER (ORDER BY id) distance
          FROM data)
 GROUP BY distance
 ORDER BY distance
/
```

Listing 9: „Tabibitosan"-Lösung (Quelle: https://community.oracle.com/thread/1007478?start=0&tstart=0)

```
WITH DATA (ID) AS (SELECT COLUMN_VALUE
                     FROM TABLE(nc(1,2,3,5,6,7,11,12,13)))
SELECT *
  FROM data
      MATCH_RECOGNIZE(ORDER BY id
                     MEASURES FIRST(id) start_of_range
                             ,LAST(id)end_of_range
                     ONE ROW PER MATCH
                     PATTERN (strt cont*)
                        DEFINE cont AS id = PREV(id)+1
                        )
/
START_OF_RANGE END_OF_RANGE
-------------- ------------
             1            3
             5            7
            11           13
```

Listing 10: Pattern Matching Clause für fortlaufende Nummern

```
Anonymer Bock
  Packaged Procedure (TEST_PCK.
PROC1)
      Local Procedure    (TEST_
PCK.PROC1.LOCAL_PROC)
```

Listing 11: Beispiel einer Aufrufsequenz

```
Level Element 1  Element 2  Ele-
ment 3
1  TEST_PCK    PROC1    LOCAL_PROC
2  TEST_PCK    PROC1
3  __anonymous_block
```

Listing 12: Call-Stack für Aufrufsequenz von Listing 11

rechtigten Zugriffen geschützt werden sollen. Da dies zur Folge hat, dass die ehemals privaten Methoden neu öffentlich in einem Library-Package deklariert sein müssen, kommt die „ACCESSIBLE_BY"-Klausel zum Einsatz, um unberechtigte Zugriffe auf das neue Library-Package zu verhindern.

```
CREATE OR REPLACE PACKAGE BODY TVDUTIL
IS
   FUNCTION whocalled RETURN VARCHAR2
   IS
   BEGIN
       RETURN sys.utl_call_stack.concatenate_subprogram(
                        sys.utl_call_stack.subprogram(2));
   END whocalled;
END TVDUTIL;
```

Listing 13: Library-Funktion für die Rückgabe der aktuellen Position im Call-Stack

```
CREATE OR REPLACE PACKAGE BODY test_pck
IS
   PROCEDURE proc1 IS
       PROCEDURE local_proc IS
       BEGIN
           sys.dbms_output.put_line('[2] : '
                                         || tvdutil.whocalled());
       END local_proc;
   BEGIN
       sys.dbms_output.put_Line('[1] : '
                                         || tvdutil.whocalled());
       local_proc();
        END proc1;
END;
/

BEGIN
   test_pck.proc1();
END;
/

[1] : TEST_PCK.PROC1
[2] : TEST_PCK.PROC1.LOCAL_PROC
```

Listing 14: Verwendung der Library-Funktion

```
CREATE TABLE employees (
   emp_id     NUMBER DEFAULT ON NULL emp_seq.nextval NOT NULL
   ,last_name VARCHAR2(30)
);
```

Listing 15: DEFAULT ON NULL

```
CREATE TABLE employees (
   emp_id     NUMBER GENERATED ALWAYS AS IDENTITY
   ,last_name VARCHAR2(30)
);
```

Listing 16: Identity Column (hier „generated always")

Identity Columns/Default ON NULL

Wichtiger als die neuen „Identity Columns" ist der Umstand, dass es nun möglich ist, als Default-Wert einer Spalte eine Sequenz zu hinterlegen und zu steuern, dass der Default-Wert auch dann verwendet werden soll, wenn „NULL" für eine Spalte übergeben wird. Diese Möglichkeit wertet die Default-Klausel extrem auf. Dadurch kann man öfter die Default-Zuweisung über diese Klausel statt über einen Trigger erfolgen lassen *(siehe Listing 15)*.

Den größten Nutzen, den wir aus den Identity Columns und der Möglichkeit, Sequenzen als Default-Wert zu verwenden, ziehen können, ist der Zeitgewinn gegenüber der herkömmlichen Lösung über Sequenzen und Datenbank-Trigger. Dieser ist dem Umstand geschuldet, dass die Context Switches zwischen SQL und PL/SQL (Trigger) wegfallen. Zudem reduziert sich der Implementationsaufwand, da weder ein Trigger noch eine Sequenz erstellt werden muss *(siehe Listing 16)*.

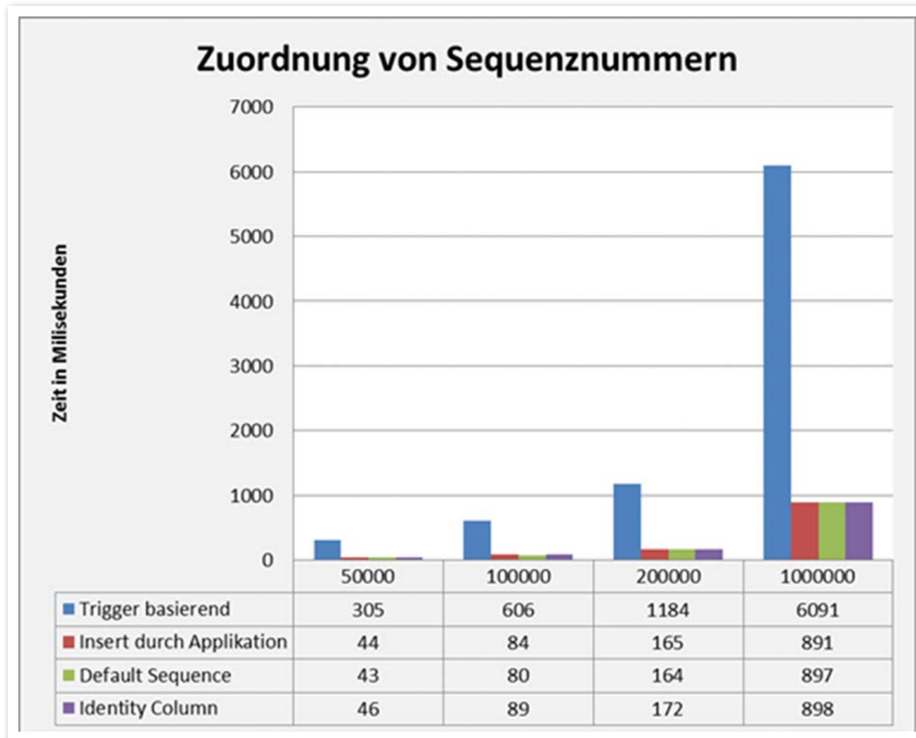
Der folgende Vergleich zeigt vier verschiedene Szenarien dafür, wie ein Datenbank-Feld mit einem Sequenz-Wert gefüllt werden kann:

- *Trigger-Lösung*
  In einem „PRE INSERT"-Trigger auf Row-Ebene wird die Sequenz gelesen und dem Datenbank-Feld zugewiesen
- *Applikatorischer Insert*
  Die Applikation liest beim Einfügen den nächsten Sequenzwert und liefert diesen beim Insert mit
- *Sequenz als Default-Wert*
  Die Sequenz ist als „Default ON NULL" in der Datenbank-Spalte hinterlegt
- *Identity Column*
  Wir bedienen uns einer Identity Column (technisch identisch mit Punkt 3)

Beim vorliegenden Test wurden 50.000, 100.000, 200.000 und eine Million Datensätze erzeugt. Die Auswertung zeigt, dass die Varianten 2 bis 4 vergleichbare Performance erreichen, während die Variante 1 (Trigger) deutlich hinterherhinkt *(siehe Abbildung 1)*.

Fazit

Auch wenn die Datenbank-Version 12*c* nicht als Entwickler-Release in die Geschichte eingehen wird, sind etliche Features enthalten, die wir in Zukunft nützen können. Neben den hier aufgeführten gäbe es viele weitere interessante Neuerungen, die es wert wären, genauer beleuchtet zu werden:



	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