Vergessene Schätze: Datenbankfeatures, die nicht jeder nutzt

Carsten Czarski
ORACLE Deutschland B.V. & Co KG
München

Einleitung

Die Version 12 der Oracle-Datenbank steht vor der Tür - und schon jetzt kann man eines mit Sicherheit sagen: Es wird wieder eine Fülle neuer Features und Funktionen geben. Aber gerade weil das jüngste Release in Artikeln, Blogs und im OTN stets im Vordergrund steht, geraten die in früheren Versionen eingeführten Funktionen häufig in Vergessenheit ...

Dieser DOAG2012-Vortrag stellt eine Auswahl an Funktionen und Features vor, welche allesamt schon seit längerer Zeit verfügbar sind - und manche davon können mit nur wenigen Handgriffen eine Menge Programmieraufwand sparen.

SQL: INSERT, UPDATE, DELETE ... und mehr.

SQL RETURNING Klausel

Möchte man wissen, welche Tabellenzeilen von einer DELETE- oder UPDATE-Anweisung betroffen sind, so ist die **RETURNING-Klausel** sehr hilfreich. Listing 1 zeigt die Anwendung.

```
declare
  type t_numlist is table of number index by binary_integer;
  la_empno t_numlist;
begin
  delete from emp where deptno = 20
  returning empno bulk collect into la_empno;

  for i in la_empno.first..la_empno.last loop
    dbms_output.put_line('deleted row with EMPNO ' || la_empno(i));
  end loop;
end;
/

deleted row with EMPNO 7369
deleted row with EMPNO 7566
deleted row with EMPNO 7788
deleted row with EMPNO 7876
deleted row with EMPNO 7902
```

Listing 1: DELETE-Anweisung mit RETURNING-Klausel

Für SQL UPDATE wird RETURNING analog eingesetzt - hier werden allerdings nicht die "alten" Werte, sondern die von der UPDATE-Anweisung veränderten "neuen" Werte zurückgegeben.

MERGE-Anweisung

Recht häufig kommt es vor, dass man INSERT und UPDATE miteinander kombinieren möchte; ist eine Zeile mit einem bestimmten Primärschlüssel schon vorhanden, so soll ein UPDATE stattfinden, andernfalls ein INSERT. Das lässt sich zum einen mit PL/SQL realisieren: Man beginnt mit einem INSERT, fängt dabei die Exception **DUP_VAL_ON_INDEX** ab und führt im *Exception Handler* ein UPDATE aus.

Alternativ dazu lässt sich die seit Oracle9*i* verfügbare MERGE Anweisung verwenden. Diese sieht spezielle Syntax für den UPDATE-Fall (**WHEN MATCHED**) und den INSERT-Fall (**WHEN NOT MATCHED**) vor. PL/SQL-Coding ist dann nicht mehr nötig. Seit Oracle10*g* sieht MERGE sogar eine DELETE-Klausel vor - MERGE kann also Zeilen einfügen, verändern oder sogar löschen. Listing 2 zeigt ein Anwendungsbeispiel.

```
merge into emp_new dest
using (select * from emp) src
on (src.empno = dest.empno)
when matched then update set
  dest.ename    = src.ename,
  dest.sal      = src.sal,
  dest.hiredate = src.hiredate,
  dest.job      = src.job,
  dest.deptno   = src.deptno,
  dest.comm     = src.comm,
  dest.mgr      = src.mgr
  -- Lösche Zeilen, wenn DEPTNO = 30
  delete where deptno = 30
when not matched then
  insert (empno, ename, sal, comm, mgr, job, hiredate, deptno)
  values (src.empno, src.ename, src.sal, src.comm, src.mgr, src.job,
src.hiredate, src.deptno)
  -- Füge nur ein, wenn DEPTNO != 30
  where not src.deptno = 30
/
```

Listing 2: Anwendungsbeispiel für SQL MERGE

Leider unterstützt SQL MERGE keine RETURNING-Klausel - die vom UPDATE bzw. DELETE betroffenen Zeilen können also nicht zurückgegeben werden.

Zeilen generieren mit hierarchischem SQL

Möchte man mit SQL Zeilen "generieren", das wird recht häufig im DWH-Umfeld für das Erzeugen von Dimensionstabellen benötigt, so bietet sich die Nutzung eines "Tricks" mit den hierarchischen SQL-Funktionen **START WITH ... CONNECT BY** an. Listing 3 generiert Zeilen für die nächsten 100 Tage, ausgehend vom aktuellen Datum.

```
select sysdate + level as datum
from dual
connect by level <= 100;
```

Listing 3: Tabellenzeilen generieren - ohne eigene Table Function

Auch mit der in Oracle11g R2 eingeführten, standardkonformen Syntax für rekursive Abfragen (*recursive WITH clause*) kann dieser Effekt erzielt werden (Listing 4).

```
with generate_date (tage, datum) as (
  select 1 as tage, sysdate as datum from dual
  union all (
    select tage + 1, sysdate + tage datum from generate_date
    where tage < 100
  )
)
select datum from generate_date;
```

Listing 4: Zeilen generieren mit der standardkonformen "recursive WITH clause"

SQL MODEL Klausel

Bereits mit Oracle10*g* wurde die *SQL Model Clause* eingeführt – sie macht es möglich, im Ergebnis einer SQL-Abfrage so mit Formeln zu rechnen, wie man es von einem Tabellenkalkulationsprogramm gewohnt ist.

Die SQL Model Clause betrachtet die Ergebnismenge einer Abfrage wie ein Arbeitsblatt einer Tabellenkalkulation. Nach dem Schlüsselwort **model** werden die Dimensionen (**dimensions**) und die Werte (**measures**) deklariert. Die Dimensionen dienen zum "Ansprechen" der Werte, die selbst durch die nachfolgenden Rules verändert werden können. Listing 5 zeigt zunächste die Struktur einer SQL-Abfrage mit der MODEL-Klausel.

```
select zeile, a, b, c, d, e, f from emp
model
 dimension by (rownum zeile)
 measures (
  empno a, ename b, hiredate c, sal d,
  comm e, cast(null as number) f
 )
 rules upsert (
   -- Hier werden "Formeln" als "Rules" eingebeben!
   b[2] = 'SCHMIDT',
   f[1] = (d[1] / d[2] - 1) * 100,
   d[15] = avg(d)[ANY]
 )
order by zeile
```

Listing 5: SQL MODEL Klausel in Aktion

Anstelle des Kommentars in den Klammern von RULES UPSERT können nun "Formeln" ähnlich zur Nutzung in einer Tabellenkalkulation angegeben werden. Und natürlich bezieht sich das "RULES UPSERT" allein auf die Ergebnismenge der SQL-Abfrage - in der Tabelle werden keine Daten geändert. Einige Beispiele

In der zweiten Zeile der Ergebnismenge soll der Name (Spalte "B") nach SCHMIDT geändert werden:
b[2] = 'SCHMIDT'

Die Spalte "F" ist zunächst leer (SQL NULL). Dort soll nun für die erste Zeile berechnet werden, um wie viel Prozent das Gehalt (Spalte "D") über dem Gehalt der zweiten Zeile liegt.
f[1] = (d[b="KING"] / d[2] - 1) * 100

Schließlich soll eine neue Zeile angefügt werden, die in der Spalte "D" den Durchschnitt über alle Gehälter enthält.
d[15] = avg(d)[ANY]

Das Ergebnis dieser SQL-Abfrage sieht dann wie folgt aus:

```
ZEILE      A B          C                          D     E     F
----- ----- ---------- ------------------- ----- ----- -----
    1  7369 SMITH      17.12.1980 00:00:00   800          -50
    2  7499 SCHMIDT    20.02.1981 00:00:00  1600   300
    3  7521 WARD       22.02.1981 00:00:00  1250   500
    4  7566 JONES      02.04.1981 00:00:00  2975
    5  7654 MARTIN     28.09.1981 00:00:00  1250  1400
:
   13  7902 FORD       03.12.1981 00:00:00  3000
   14  7934 MILLER     23.01.1982 00:00:00  1300
   15                                       2073

15 Zeilen ausgewählt.
```

Listing 6: Ergebnismenge der SQL-Abfrage mit MODEL-Klausel

Linguistische Indizes

In Datenbeständen wird häufig mit LIKE-Abfragen gesucht. Eine Standard-SQL-Abfrage arbeitet aber stets *Case-Sensitiv*; die Groß- und Kleinschreibung muss in der Suche also genau so eingegeben werden, wie die Daten in der Tabelle stehen. Werden Umlaute oder andere Sonderzeichen verwendet, so muss man diese ebenfalls exakt eingeben.

Funktionsbasierte und linguistische Indizes können hier Abhilfe schaffen. Ein linguistischer Index "normalisiert" alle Werte vor der Indizierung, das bedeutet, es wird alles kleingeschrieben indiziert und diakristische Zeichen werden auf ihre Grundformen reduziert ("ä" wird zu "a").

Damit der linguistische Index richtig genutzt wird, müssen die SESSION-Parameter **NLS_COMP** und **NLS_SORT** korrekt eingestellt werden.

```
create index idx_daten on daten (
  NLSSORT(col, 'NLS_SORT = German_AI')
);

alter session set nls_comp=linguistic;

alter session set nls_sort=german_AI;

select * from daten where col='MULLER';

COL
--------------------------------------------------------------------
Müller
Muller
```

Listing 7: Erstellen und nutzen eines linguistischen Index

Nützliche PL/SQL - Packages

Mit PL/SQL arbeitet man in der Oracle-Datenbank nahezu ständig. Die von Oracle mitgelieferte Funktionsbibliothek in Form der eingebauten Packages erweitert sich ebenfalls mit jedem Release. Im folgenden sind einige der weniger bekannten Packages vorgestellt.

HTTPURITYPE

HTTPURITYPE ist eigentlich kein Package, sondern ein Objekttyp, hinter dem sich aber die sehr nützliche Funktion eines "Kommandozeilen-Browsers" verbirgt. Der Inhalt einer Webseite kann wie folgt abgerufen werden:

```
select httpuritype('www.doag.org').getclob() from dual;

HTTPURITYPE('WWW.DOAG.ORG').GETCLOB()
--------------------------------------------------------------------
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html
     PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
:
```

Listing 8: Abruf einer Webseite mit HTTPURITYPE

Besonders interessant ist das natürlich zum Abrufen und direkten Verarbeiten von XML.

```
select x.termin
from
 xmltable(
  '//item'
  passing
    httpuritype('http://www.doag.org/rss/doag_termine.xml').getxml()
  columns
    termin varchar2(50) path '/item/title/text()'
 ) x;

TERMIN
--------------------------------------------------
04.09.2012: SIG MySQL - fällt leider aus!
05.09.2012: Regionaltreffen Berlin Brandenburg
07.09.2012: DOAG Leitungssitzung
:
```

Listing 9: XML-Abruf per HTTPURITYPE und direkt Verarbeitung mit SQL-Funktionen

Befindet sich die Datenbank hinter einer Firewall (was meistens der Fall sein dürfte), so muss der Proxy-Server vorher mit **UTL_HTTP.SET_PROXY** eingestellt werden.

DBMS_FREQUENT_ITEMSETS

Dieses, sehr mächtige Paket erlaubt das Finden von häufig vorkommenden Zeilenkombinationen. Angenommen, man hat eine Tabelle **SALES**, in der Verkaufsvorgänge gespeichert sind - und nun möchte man herausfinden, welche Produkte besonders häufig gemeinsam gekauft werden.

```
select
  column_value,
  support,
  length,
  total_tranx
from table(
  dbms_frequent_itemset.fi_transactional(
    cursor(
      select
        cust_id || ' ' || time_id,
        prod_name
      from sh.sales join sh.products using (prod_id)
    ),
    0.05,
    2,
    2
  )
) fi, table(fi.itemset)
```

Listing 10: Finden häufig vorkommender Zeilenkombinationen mit DBMS_FREQUENT_ITEMSETS

Neben einem Cursor als Datenquelle müssen noch Parameter übergeben werden für ...

- Ab wie viel Prozent (der Gesamtmenge) gilt eine Kombination als "häufig"
- Aus wie vielen Zeilen besteht eine Kombination mindestens
- Aus wie vielen Zeilen besteht eine Kombination höchstens

Es können zusätzlich Cursor übergeben werden, die bestimmte Merkmal explizit einbeziehen oder ausschließen

DBMS_APPLICATION_INFO

Dies ist eines der für den Entwickler wichtigsten Pakete überhaupt. Egal, ob man mit PL/SQL, Java, .NET oder einer anderen Programmierumgebung arbeitet: DBMS_APPLICATION_INFO sollte jeder Entwickler nutzen.

Mit DBMS_APPLICATION_INFO werden, kurz gesagt, die Spalten **MODULE**, **ACTION** und **CLIENT_INFO** der Dictionary View **V$SESSION** befüllt. Der Zweck des Pakets ist also, den DBA über die Verwendung der Datenbanksession zu informieren. Gerade in den heute typischen 3-Schichten-Architekturen, bei denen der Application Server mit der Datenbank über einen"technischen" User verbunden ist, sieht der DBA typischerweise nur "n JDBC-Datenbanksitzungen". Was die Sessions im einzelnen tun, welche wichtig und welche weniger wichtig ist, ist nicht erkennbar.

Mit DBMS_APPLICATION_INFO kann der Entwickler wie folgt Abhilfe schaffen:

```
begin
  dbms_application_info.set_client_info(
    client_info => 'MEINE_PLSQL_PROZEDUR'
  );
  dbms_application_info.set_module(
    module_name => 'MODUL 1',
    action_name => 'BERECHNE KENNZAHLEN'
  );
end;
```

Listing 11: Nutzen von DBMS_APPLICATION_INFO

Wenn der DBA sich nun per **V$SESSION** in seiner Datenbank umsieht, erhält er folgende Information:

```
select sid, serial#, client_info, module, action
from v$session where username='SCOTT'

       SID    SERIAL# CLIENT_INFO          MODULE     ACTION
---------- ---------- -------------------- ---------- --------------
        39       9780 MEINE_PLSQL_PROZEDUR MODUL 1    BERECHNE_KENN
         :          : :                    :          :
```

Listing 12: DBA-Sicht auf die V$SESSION bei Nutzung von DBMS_APPLICATION_INFO

DBMS_ERRLOG

Dieses Paket wird benötigt, um das DML Error Logging Feature zu verwenden. Angenommen, man möchte mit einem INSERT ... SELECT eine Tabelle mit sehr großen Datenmengen füllen - es wird also erwartet, dass das Kommando sehr lang läuft.

Und in der Praxis ist es nun gerade so, dass (natürlich kurz vor dem Ende) eine Zeile generiert wird, die nicht in die Tabelle passt - ein VARCHAR2-Wert mag zu lang sein, ein Constraint mag verletzt werden, es wird vielleicht auch ein ungültiges Datum generiert.

Ein "normales" DML-Kommando bricht mit einer Fehlermeldung ab und alle bislang gemachten Änderungen werden wieder zurückgerollt. Man muss also den ganzen langlaufenden Vorgang nochmals starten. Das DML Error Logging Feature schafft Abhilfe:

Zunächst wird mit dem Paket DBMS_ERRLOG eine neue Tabelle angelegt, in welche die aufgetretenen Fehler geschrieben werden können. Dabei wird der Name der "Ziel-Tabelle", also der Tabelle, in die die neuen Zeilen geschrieben werden sollen, angegeben.

```
begin
  dbms_errlog.create_error_log (dml_table_name => 'EMP');
end;
```

Listing 13: Erzeugen der DML Error Logging Tabelle

Das "Error Log" ist eine Tabelle mit dem Namen **ERR$_{Tabellenname}**; hier also **ERR$_EMP**. Sie enthält neben den Spalten der Zieltabelle auch einige Systemspalten, in denen Informationen zum aufgetretenen Fehler gespeichert werden. Wird nun beim DML-Kommando die Klausel **log errors into** verwendet, so läuft das DML-Kommando auch bei auftretenden Fehlern weiter.

```
insert into EMP (
  select a.spalte1 as empno, b.spalte2 as ename, ...
  from quelltabelle1 a, quelltabelle2 b
  where a.id = b.reference
)
log errors into err$_EMP('DML_OP') reject limit unlimited;
```

Listing 14: Nutzung des "DML Error Logging"

Das INSERT .. SELECT wird nun auch bei auftretenden Fehlern weiterlaufen. Im Fehlerfall werden alle Spaltenwerte mitsamt Informationen über den aufgetretenen Fehler in die Tabelle **ERR$_EMP** abgespeichert und können im Nachgang bearbeitet werden. Die langlaufende Operation kann aber auf jeden Fall durchlaufen und ein ggfs. vorhandenes Zeitfenster ausgenutzt werden.

Die Datenbank erweitern

Die Oracle-Datenbank ist an sehr vielen Stellen erweiterbar. Das geht weit über das Erstellen von Funktionen hinaus: So kann die Datenbank mit zusätzlichen SQL-Operatoren, Indextypen oder neuen Aggregatsfunktionen ausgestattet werden.

Benutzerdefinierte Aggregatsfunktionen

Eine *User Defined Aggregate Function* wird anders implementiert als eine normale PL/SQL-Funktion - dafür leistet sie auch wesentlich mehr. So kann sie als Aggregatsfunktion in einer SQL-Abfrage im

Zusammenspiel mit **GROUP BY** verwendet werden. Sogar in der analytischen Variante mit der Klausel **OVER()** sind sie nutzbar.

Ein Beispiel findet sich auf dem Blog "SQL und PL/SQL in Oracle" unter http://sql-plsql-de.blogspot.co.uk/2011/02/sql-aggregatsfunktion-product-fehlt.html. Wird der Code für die benutzerdefinierte Aggregatsfunktion **AGG_PRODUCT** eingespielt, so kann diese danach wie folgt genutzt werden.

```
select deptno, agg_product(sal) product_sal from emp
group by deptno;

    DEPTNO PRODUCT_SAL
---------- ------------------------------
        10 15925000000
        20 23562000000000000
        30 101531250000000000000
```

Listing 15: Die benutzerdefinierte Aggregatsfunktion AGG_PRODUCT in Aktion

Gerade in Bereichen der Finanzmathematik können eigene Aggregatsfunktionen sehr hilfreich sein.

Java in der Datenbank

Die in der Datenbank seit Oracle8*i* enthaltene Java-Engine kann für vielfältige Zwecke verwendet werden; besonders interessant wird Java Code, wenn es für den jeweiligen Zweck kein PL/SQL-Paket gibt. Beispiele:

ZIP Archive ein und auspacken
http://sql-plsql-de.blogspot.co.uk/2010/11/zip-archive-einpacken-und-auspacken-das.html

Email-Client für SQL und PL/SQL
http://plsqlmailclient.sourceforge.net

Betriebssystem-Kommandos mit PL/SQL ausführen
http://plsqlexecoscomm.sourceforge.net

FTP-Client für PL/SQL
http://www.oracle.com/webfolder/technetwork/de/community/apex/tipps/ftpclient/index.html

Weitere Informationen

Viele der hier vorgestellten Beispiele sind der deutschen APEX Community und dem Blog "SQL und PL/SQL in Oracle" entnommen. Darüber hinaus gibt es mittlerweile einige deutschsprachige Informationsquellen rund um die Oracle-Datenbank.

[1] Deutschsprachige APEX Community
http://tinyurl.com/apexcommunity

[2] Deutschsprachige DBA Community
http://tinyurl.com/dbacommunity

[3] Blog "SQL und PL/SQL in Oracle"
http://sql-plsql-de.blogspot.com

Kontaktadresse:

Carsten Czarski
ORACLE Deutschland B.V. & Co KG
Riesstr. 25

80992 München

Telefon:	+49 (0) 89 1430 2116
E-Mail	carsten.czarski@oracle.com
Internet:	www.oracle.de
Blog des Autors	http://sql-plsql-de.blogspot.com
Twitter	@cczarski