Oracle's Eleven

11 nieuwe PL/SQL en SQL features in Oracle 11g

In dit artikel behandelen Ton Elie en Erwin Groenendal 11 nieuwe PL/SQL en SQL features in Oracle 11g. De selectie is gemaakt vanuit het perspectief van een ontwikkelaar. Nieuwe functionaliteit in Oracle 11g met betrekking tot beheer, beschikbaarheid en schaalbaarheid vallen buiten de scope van dit artikel. De geselecteerde features zijn, uiteraard, niet de enige nieuwe PL/SQL en SQL features in Oracle 11g, en ook niet de 11 belangrijkste. Wel zijn ze stuk voor stuk interessant en goed toepasbaar. Behalve misschien de laatste...

Enkele nieuwe features met betrekking tot ondersteuning van XML in de database zijn al besproken in het artikel 'XML en Oracle11g RDBMS' (Optimize nr. 4/ 2007). Hierin is ingegaan op het nieuwe opslagformaat voor XML data 'binary XML' en het nieuwe type index 'XMLIndex' dat het oude Oracle Text indextype 'CTXXPath' vervangt. Deze twee nieuwe features tellen we in dit artikel niet mee, wel komen er nog twee andere nieuwe XML features aan bod.

Pivot en Unpivot

Oracle 10g had veel nieuwe features op datawarehousing-gebied. In Oracle 11g worden hier de PIVOT en UNPIVOT clauses aan toegevoegd. Met de pivot clause kan de output van een query in matrixvorm gezet worden. Hierbij worden waarden uit een rij in het resultaat van de query in een kolom in de matrix gezet.

De syntax is als volgt:

```
SELECT ...
FROM <table-expr> AS <alias>
PIVOT
(
aggregate-function(<value-column>)
FOR <pivot-column> IN ([<value1>], [<value2>],..., [<valuen>])
) AS <alias>
WHERE ...
```

Er moet altijd gebruik worden gemaakt van een aggregatiefunctie (MAX, MIN, STD, AVG et cetera) op een groepering van rijen. Indien niet zo'n functie gebruikt wordt, levert dit de error 'ORA-54032: expected aggregate function inside pivot operation' op. De waarden die de functie retourneert zijn de waarden die in de matrix gezet kunnen worden.

De <table-expr> zal in de regel een inline view zijn om enkel de kolommen te selecteren die nodig zijn en om de juiste groepering te krijgen. Een voorbeeld maakt deze functionaliteit duidelijker. Als we op de 'stoffige', maar bekende, tabellen 'emp' en 'dept' de volgende (normale) query uitvoeren.

```
SELECT deptno
,      job
,      max(sal)
FROM emp
GROUP BY deptno
,         job
ORDER BY deptno
,         job
```

levert dit de onderstaande output op.

```
    DEPTNO JOB          MAX(SAL)
---------- --------- ----------
        10 CLERK             1300
        10 MANAGER           2450
        10 PRESIDENT         5000
        20 ANALYST           3000
        20 CLERK             1100
        20 MANAGER           2975
        30 CLERK              950
        30 MANAGER           2850
        30 SALESMAN          1600
```

Om de output in matrixvorm te krijgen, met het afdelingsnummer (DEPTNO) in de eerste kolom en een kolom per job-naam (JOB) met het maximumsalaris (voor de betreffende afdeling en functie) als matrixwaarden kan de pivot clause

gebruikt worden. Dit wordt gedaan in onderstaande query.

```
SELECT deptno
,      clerk_sal
,      nvl(salesman_sal,0)    AS salesman_sal
,      manager_sal
,      nvl("'ANALYST'_SAL",0) AS analyst_sal
,      president_sal
FROM  (SELECT deptno, sal, job
       FROM scott.emp
      )
PIVOT (max(sal) AS sal FOR job IN ( 'CLERK'     AS clerk
                                  , 'SALESMAN'  AS salesman
                                  , 'MANAGER'   AS manager
                                  , 'ANALYST'
                                  , 'PRESIDENT' AS president
                                  )
             )
ORDER BY deptno
```

Het resultaat van deze query is als volgt.

```
    DEPTNO  CLERK_SAL SALESMAN_SAL MANAGER_SAL ANALYST_SAL PRESIDENT_
SAL
---------- ---------- ------------ ----------- ----------- ------------
        10       1300            0        2450           0
5000
        20       1100            0        2975        3000
        30        950         1600        2850           0
```

Een aantal opmerkingen over bovenstaande query:

- de IN clause bepaalt welke nieuwe 'kolommen' beschikbaar zijn
- in de (buitenste) select list van de query wordt verwezen naar deze 'kolomnamen' (bijvoorbeeld 'manager_sal') via een alias die is opgebouwd uit de alias ('manager') gebruikt in de IN clause en daarachter een underscore ('_') en de alias van aggregatie ('sal').
- omdat voor de job-naam 'ANALYST' geen gebruik is gemaakt van een alias in de IN clause moet naar deze 'kolom' op een andere manier verwezen worden, de opgebouwde alias bevat namelijk een enkele quote (apostrophe) voor en achter de job-naam, in de select list moet daarom gebruik worden gemaakt van dubbele quotes zodat naar de juiste alias verwezen kan worden, het is daarom handig om altijd aliassen te gebruiken in de IN clause
- de 'nvl' functies in de select list zorgen er voor dat de waarde '0' in de matrix gezet wordt voor niet voorkomende combinaties van afdeling en functie, voor de job-naam 'PRESIDENT' is dit niet gedaan

Vóór Oracle 11g kon een dergelijke output alleen verkregen worden door gebruik te maken van een gekunstelde constructie met decode's.

Unpivot

Met de unpivot clause kan het omgekeerde van de pivot clause gedaan worden. Met unpivot kunnen kolommen in de tabel rijen worden in de output van de query.
Stel dat tabel 'DEPT_MAX_SAL' het resultaat van bovenstaande query als inhoud heeft. Dan kan met de onderstaande query een rij verkregen worden per afdeling en functie combinatie.

```
SELECT *
FROM   dept_max_sal
UNPIVOT (salary FOR job_title IN ( clerk_sal AS clerk
                                 , salesman_sal AS salesman
                                 , manager_sal AS manager
                                 , analyst_sal AS analyst
                                 , president_sal AS president
                                 ))
```

De output van deze query is:

```
DEPTNO               JOB       SALARY
-------------------- --------- ----------------------
30                   CLERK     950
30                   SALESMAN  1600
30                   MANAGER   2850
30                   ANALYST   0
20                   CLERK     1100
20                   SALESMAN  0
20                   MANAGER   2975
20                   ANALYST   3000
10                   CLERK     1300
10                   SALESMAN  0
10                   MANAGER   2450
10                   ANALYST   0
10                   PRESIDENT 5000
```

Een aantal opmerkingen over deze query:

- de IN clause bepaalt voor welke kolommen een rij in de output komt, omdat de kolom DEPTNO niet in de IN clause staat komt deze kolom in elke rij terug
- met de literal 'job_title' wordt aangegeven hoe de kolom in de output heet, de aliassen in de IN clause bepalen welke waarden hierin komen
- de matrixwaarden in de tabel (maximum salaris) komt in de ouput terug in een kolom waarvan de naam bepaalt wordt met de literal 'salary'
- de query levert 13 rijen op; precies het aantal (not null) matrixwaarden in de tabel

Virtuele kolommen

In Oracle 11g is het mogelijk om zogenaamde virtuele kolommen te creëren in een tabel. In deze kolommen kunnen waarden afgeleid van de inhoud van andere kolommen (in hetzelfde record) geplaatst worden. Vervolgens kunnen ze als 'gewone" kolommen benaderd worden in queries en is het zelfs mogelijk om er indexen en constraints op te definiëren.

```
alter table employees add (full_name varchar2(50)
as upper(last_name)||', '||upper(first_name))
```

In het bovenstaande statement wordt een kolom toegevoegd waarin de full name van de employee wordt gezet. In voorgaande versies van de Oracle database zou een echte kolom toegevoegd moeten worden, plus een trigger die de afleiding doet. Stel dat de full name in het bovenstaande voorbeeld uniek moet zijn, dan kan eenvoudigweg een unieke index gedefinieerd worden op deze virtuele kolom. Vóór 11g zou dit afgedwongen kunnen worden met een function based index. In 11g is dat veel gemakkelijker en beter onderhoudbaar geworden.
Virtuele kolommen zijn ook erg handig om data uit een XMLType kolom toegangkelijk te maken voor indexeren of constraints. In het onderstaande statement wordt de afkorting van een staat afgeleid en wordt deze als primary key gedefinieerd.

```
CREATE TABLE cxd_xml_data_states
( xmldata    XMLTYPE        NOT NULL
, v_state    VARCHAR2(2) AS (EXTRACTVALUE(xmldata,
                             'state/abbreviation'))
, CONSTRAINT cxd_xml_states_pk PRIMARY KEY (v_state)
)
```

Jammer genoeg kunnen in de afleiding van een virtuele kolom enkel standaard SQL functies gebruikt worden. Wordt een zelfgescheven (deterministic) functie gebruikt dan resulteert dat in een fout: 'ORA-54016: Invalid column expression was specified. In dat geval zal teruggevallen moeten worden op oude, meer omslachtige, methodes.

'Onzichtbare' indexen

Indexen kunnen 'onzichtbaar' gemaakt worden voor de optimizer. Deze mogelijkheid is zeer handig voor het tunen van een applicatie. Pré 11g moesten indexen gedropt worden om niet meer door de optimizer te worden meegenomen bij het berekenen van het beste executieplan. Het opnieuw creëren van indexen kan een tijdrovende bezigheid zijn. In 11g kan dus snel getest worden wat de performance zou zijn als de index niet (meer) zou bestaan.

Het onzichtbaar maken van een index gebeurt met het volgende statement:

ALTER INDEX <index> INVISIBLE;

maar kan even snel weer worden teruggedraaid met het statement:

ALTER INDEX <index> VISIBLE;

Houd er wel rekening mee dat de index onzichtbaar voor de optimizer maar bij inserts, updates en deletes wel bijgewerkt wordt.

Read-only tables

Tabellen kunnen in 11g read-only gemaakt worden met statement:

ALTER TABLE <table name> READ ONLY;

Na het uitvoeren van het volgende statement kan weer in de tabel geschreven worden:

ALTER TABLE <table name> READ WRITE;

In vorige versies kon wel op tablespace niveau afgedwongen worden dat er geen DML kan plaatsvinden. Nu kan dit op de tabel zelf afgedwongen worden.

Naast een functionele reden om tabellen in read only mode te zetten kan er ook een technische reden zijn. Voor tabellen die in read-only mode staan hoeft er geen redo log informatie bijgehouden te worden. Als dit niet gebeurt zal dat voordelig zijn voor de performance en zullen 'snapshot too old' meldingen niet optreden.

PL/SQL function result cache

Met deze functionaliteit kan het resultaat van een PL/SQL-functie in een result cache geplaatst worden. Deze cache is beschikbaar over sessies heen. Wordt een result cache functie met dezelde parameterwaarden aangeroepen dan zal de returnwaarde uit de cache gehaald worden.
In vorige versies van de Oracle database zou cache functionaliteit zelf gemaakt kunnen worden door in globale package variabelen resultaten op te slaan. De cache is dan echter sessiegebonden, waardoor invalidatie door andere sessies niet mogelijk is. Invalidatie is noodzakelijk wanneer wijzigen plaatsvinden in

Advertentie

de kolomwaarden, waarvan het functieresultaat afhankelijk is wijzigen.
Daarnaast is vanwege het stateless karakter tussen middle tier en database in webapplicaties een sessieafhankelijke cache niet bruikbaar.

De PL/SQL function result cache functionaliteit van 11g biedt uitkomst door een sessie onafhankelijk cache te bieden die automatisch geschoond (invalidatie) wordt.

Invalidatie result cache

De RELIES_ON clause bepaalt wanneer de cache geschoond wordt. De gespecificeerde objecten zullen bij wijzigingen de cache schonen. Dus als in het voorbeeld de EMP tabel wordt aangepast zal de result cache van de functie NAME leeggemaakt worden.
Inconsistentie tussen de RELIES_ON clause en de functionaliteit van de functie kan dus tot verkeerde resultaten leiden. Nauwkeurigheid is hier vereist.

```
CREATE OR REPLACE FUNCTION name(p_empno IN NUMBER)
RETURN VARCHAR2
RESULT_CACHE RELIES_ON (emp)
IS
   l_name VARCHAR2(200);
BEGIN
   SELECT ename
   INTO l_name
   FROM emp
   WHERE empno = p_empno
   ;
   RETURN l_name;
END;
```

SQL

In SQL kan de result cache geactiveerd worden door de optimizer hint /*+ result_cache */. Invalidatie van de cache gebeurt automatisch als de tabellen waarop de query gebasseerd is gemuteerd worden.
De bovenstaande functie is dus functioneel gelijk aan de volgende functie.

```
CREATE OR REPLACE FUNCTION name(p_empno IN NUMBER)
RETURN VARCHAR2
IS
l_name VARCHAR2(200);
BEGIN
   SELECT /*+ result_cache */  ename
   INTO l_name
   FROM emp
   WHERE empno = p_empno
   ;
   RETURN l_name;
END;
```

De systeem- of sessieparameter RESULT_CACHE_MODE stuurt de SQL result cache functionaliteit. Deze kan de waarde 'FORCE', 'MANUAL' of 'AUTO' hebben.

Door deze parameter op FORCE te zetten met

alter session set RESULT_CACHE_MODE = FORCE;

zal altijd gebruik gemaakt worden van de result cache. In deze mode kan de result cache voor een statement uitgezet worden met de hint /*+ no_result_cache */.

Gedetailleerder dependencies model

In 11g is het dependency model verfijnd zodat niet onnodig objecten geinvalideerd hoeven te worden bij een alter statement of hercreatie van een package. Zo worden bijvoorbeeld views niet meer invalid als aan een tabel (waarop de view gebaseerd is) een kolom wordt toegevoegd. En worden package bodies niet meer invalid als een specificatie wordt gehercompileerd terwijl aan de gebruikte functies of procedures niets is gewijzigd.
Dit heeft met name grote voordelen voor de beschikbaarheid van productiesystemen tijdens releases. Downtime wordt zo tot een minimun beperkt. Werd je in de oude versies in sommige gevallen weleens 'gestraft' voor het gebruik van packages in plaats van losse functies en procedures, dat is dat nu eindelijk verleden tijd.

PL/SQL function calls met named parameters in SQL

Als een PL/SQL functie wordt aangeroepen in een SQL statement is het vanaf Oracle 11g ook mogelijk om named parameters te gebruiken.

```
SELECT get_full_name(p_empno => empno)
FROM employees;
```

PL/SQL function voor next value van een sequence

In 11g is er een PL/SQL implementie voor het ophalen van de nextval van een sequence (<sequence-name>.nextval;)en de currval (<sequence-name>.currval;). Hierdoor is het niet meer nodig om van dual te selecteren zoals in pré-11g versies.

Nieuwe XML functies: XMLExists en XMLCast

In 11g zijn er weer twee nieuwe functies toegevoegd aan de SQL standaard package.

De XMLExists() functie is vergelijkbaar met existsNode(). Deze nieuwe functie is onderdeel van de SQL/XML standaard van W3C. Deze is dus te prefereren boven de Oracle specifieke functie existsNode(). Met de functie XMLExists() kunnen XQuery expressies worden gebruikt.
De XMLExists() functie geeft een SQL boolean (!) terug en kan in de where clause gebruikt worden. ExistsNode() retourneert een number waardoor deze ook op andere plaatsen te gebruiken is. Door XMLExists in een CASE statement in de select list te plaatsen kan toch hetzelfde resultaat verkregen worden. Zie de tweede query hieronder.
De XMLCast() functie is vergelijkbaar met de functie extractValue. Ook deze functie is een SQL/XML standaard. Met de nieuwe functie XMLCast() kan het output datatype gespecificeerd worden, wat niet mogelijk is met de Oracle specifieke functie extractValue. ExtractValue geeft altijd VARCHAR2 terug. Hieronder een voorbeeld om het verschil te demonstreren tussen het gebruik van de "oude" functies en de nieuwe SQL/XML functies.

```
SELECT extractValue(OBJECT_VALUE, 'PurchaseOrder/ShippingInstructions/
address')
,       existsNode(OBJECT_VALUE, 'PurchaseOrder/LineItems/LineItem[@
ItemNumber="17"]') FROM    oe.PurchaseOrder
WHERE  existsNode(OBJECT_VALUE, 'PurchaseOrder/User/text()="AMCEWEN"') =
1
```

Met de nieuwe functies wordt dat:

```
SELECT XMLCast(XMLQuery('/PurchaseOrder/ShippingInstructions/address'
                        PASSING OBJECT_VALUE RETURNING CONTENT
                       )
              AS VARCHAR2(100)
              ) shipping_address
,       CASE WHEN XMLExists('$p/PurchaseOrder/LineItems/LineItem[@
ItemNumber="17"]'
                           PASSING OBJECT_VALUE AS "p"
                          )
       THEN 1 ELSE 0 END Item_17_inorder
FROM oe.PurchaseOrder
WHERE XMLExists('$b/PurchaseOrder[User="AMCEWEN"]'PASSING OBJECT_VALUE
AS "b")
```

Dit statement geef echter een fout:

```
ORA-19162: XPTY0004 - XQuery type mismatch: invalid argument types
'xs:integer', 'xs:string' for function '='
```

Dit komt door het zogenaamde static type checking wat geintroduceerd is voor de nieuwe functies. De XMLExists doet een deze static type checking op de XQuery expressie. Dit is een aanbeveling van W3C voor XQuery 1.0. Dit houdt in dat er een fout wordt gegeven als het datatype van de waarde in de expressie niet overeenkomt met zijn context.

De regel:

```
XMLExists('$p/PurchaseOrder/LineItems/LineItem[@ItemNumber="17"]'
```

moet worden:

```
XMLExists('$p/PurchaseOrder/LineItems/LineItem[@ItemNumber=17]'
```

Het onderstaande fragment van het de XML Schema definition laat inderdaad zien dat ItemNumber het datatype integer heeft.

```
   <xs:attribute name="ItemNumber" type="xs:integer"/>
</xs:complexType>
```

Static type checking in 11g is afhankelijk van twee factoren:

- de XML tabel of XML kolom moet gecreëerd zijn met een XMLSCHEMA clause
- de XML opslag is object relationeel of binary

Voor CLOB opslag gebeurt dit niet. Waarschijnlijk omdat Oracle niet kijkt naar de XML Schema definition maar naar de opslagstructuur in de database voor de static type checking.

Unlimited node size

Bij het maken van een XML document in de Oracle XDB zit er in de vorige versies van de database een limiet van 64K voor text nodes en attribuutwaarden. In 11g is aan de grootte geen restrictie meer.

BLOB support in SQL*Plus

Wie verzint een toepassing?

In oudere versies van SQL*Plus kan geen BLOB datatype gebruikt worden. Nu lijkt het representeren van een BLOB waarde niet erg nuttig. Echter applicaties die veel gebruik maken van SQL*plus scripts zouden last van deze beperking. Voorbeelden van een nuttige toepassing kunnen naar Ton Elie (ton.elie@cumquat.nl) gemaild worden.

Ton Elie is Senior Software Engineer binnen Tangelo Product Development bij Cumquat Information Technology.
E-mail: ton.elie@cumquat.nl.
Erwin Groenendal is Senior Solutions Architect bij Cumquat Information Technology. E-mail: erwin.groenendal@cumquat.nl.