Oracle 11g R2 voor ontwikkelaars - deel 1

Lucas Jellema, Alex Nuijten - AMIS Services BV

Introductie
Vlak voor Oracle Open World 2009 was de release van het nieuwste vlaggenschip van Oracle: Database 11*g*, Release 2. In eerste instantie alleen op Linux (1 september) maar geleidelijk aan op andere platforms (Solaris, HP-UX, AIX) en in de loop van februari waarschijnlijk ook Windows. De moeite waard, die nieuwe release van 11g? Of is Release 2 alleen maar een verzameling van bugfixes van Release 1?
Omdat AMIS Services betrokken is bij het beta testen van de Oracle database hadden we al in een vroeg stadium de nieuwste versie beschikbaar. Diverse collega's zijn betrokken bij het testen van de database, ieder vanuit zijn eigen specialisme. In een aantal artikelen over 11gR2 zullen we proberen die vraag of 11gR2 de moeite waard is te beantwoorden - en je ondertussen bijpraten over de vele nieuwe en verbeterde features, de speerpunten in de marketing van Oracle rondom deze release en de aspecten die ons erg aanspreken.

In deze aflevering gaan we het vooral hebben over features op het vlak van database ontwikkeling, features die zeker de moeite waard zijn, maar die misschien in andere media geen of weinig aandacht hebben gekregen. Iedereen praat tenslotte over de "killer feature" van de Oracle 11*g* R2 database: Edition Based Redefinition. De volgende aflevering staat geheel in het teken van Edition Based Redefinition. Nu eerst: wat biedt Oracle 11gR2 voor database ontwikkelaars.

Afscheid van Connect By

Vroeger, in de jaren '90, was Oracle niet zo bezig met standaarden. Oracle SQL was toch immers de standaard voor SQL? In de loop van het afgelopen decennium is daar het nodige in veranderd. De 9i release van de database bracht ons ondermeer de ANSI SQL Join syntax die het einde inluidde van de (+) notatie (hoewel veel Oracle ontwikkelaars daar nog niet aan toe (b)lijken te zijn). Ook in 9iR2 de CASE expressie - onderdeel van de industrie standaard voor SQL. Oracle probeert haar SQL/XML implementatie na wat omzwervingen zoveel mogelijk in lijn te brengen met de standaarden en heeft zelfs meer obscure standaard-elementen als COALESCE en NULLIF geïmplementeerd.

De ANSI SQL standaard bevat ook al geruime tijd een syntax voor hierarchische queries.

Queries die pig-ears of self-joins bewandelen en tree-structuren en netwerk-afhankelijkheden doorzoeken. De allereerste versie van Oracle - release 2 - bevatte al de CONNECT BY operator die samen met START WITH, PRIOR en LEVEL de hierarchische query-toolkit vormde. In latere versies zijn aan die toolkit geavanceerde functies toegevoegd, zoals SYS_CONNECT_BY_PATH, CONNECT_BY_ROOT, ORDER SIBLINGS BY en CONNECT_BY_IS_LEAF. Het oer-voorbeeld van de hierarchische query ziet er als volgt uit:

```
 select lpad(' ', level *3)||ename name
   from emp
  start with mgr is null
connect by prior empno = mgr
```

Deze query toont de employees in een soort eenvoudig organigram - een boom die start met KING en waar onder iedere tak de ondergeschikten hangen.

De ANSI SQL syntax voor hierarchische queries die Oracle introduceert in 11g Release 2 is totaal afwijkend van de connect by structuur. Onder de pakkende naam Recursive Subquery Factoring krijgen we een speciale variant op de WITH clause die recursieve queries mogelijk maakt waarmee stapsgewijs hierarchische- en netwerk-structuren kunnen worden afgelopen. Een waarschuwing vooraf - en een geruststelling: in eerste instantie ziet de syntax er weinig intuïtief of aanlokkelijk uit. Je eerste reactie zal er waarschijnlijk niet eentje zijn van groot enthousiasme. Ons advies: geef het een kans! Net als met de ANSI Join syntax moet je over een drempel, maar het is het waard. Het geeft je niet alleen de bestaande functionaliteit met een een weliswaar standaard maar niet vertrouwde en initieel dus complexe syntax, maar voegt ook functionaliteit toe.

Bovenstaande hierarchische query oude stijl wordt in 11gR2 herschreven tot:

```
with employees ( name, empno, hierlevel) as
( select ename
       , empno
       , 1
    from emp
   where mgr is null
   union all
  select e.ename
       , e.empno
       , m.hierlevel + 1
    from emp e
         join
         employees m
         on (m.empno = e.mgr)
)
select lpad(' ', hierlevel *3)||name name
  from employees
```

De recursive subquery - hier is dat employees - bestaat uit twee delen. Het eerste deel kan je vergelijken met de *start with* clause in de oude stijl hierarchische query. Dit deel bepaalt de basis-set van rijen, de uitkomst van de eerste iteratie in de recursieve query. In dit voorbeeld bestaat deze basisset uit rijen met (ename, empno,1) uit table EMP waar de kolom MGR geen waarde bevat. Alle volgende iteraties worden beschreven door de tweede

helft van de recursive subquery, het deel na de UNION ALL. Dat deel heeft een referentie naar de subquery als geheel - in het voorbeeld is dat alias m - en wordt recursief uitgevoerd, met als vertrekpunt de rijen die in de vorige iteratie zijn toegevoegd. Die rijen worden via de self-reference benaderd. De eerste keer dat dit tweede, recursieve deel wordt uitgevoerd in het voorbeeld is de inhoud van employees dus het resultaat van de eerste query: alle medewerkers zonder manager. De recursieve iteraties stoppen zodra een iteratie geen nieuwe rijen heeft opgeleverd.

De recursive subquery stelt geen specifieke eisen aan de beiden queries. Deze kunnen bijvoorbeeld prima verschillende tabellen als basis gebruiken. Zolang de beide queries maar evenveel kolommen van gelijke types opleveren is het goed.

Een voordehandliggende vraag bij de introductie van deze nieuwe syntax zou zijn: het heeft van Oracle 2 tot en met Oracle 10 geduurd alvorens we de lijst van geavanceerde functies rondom hierarchische queries op het huidige niveau hadden; moeten we nu weer zo'n periode gaan wachten met de recursive subquery - of worden er goodies meegeleverd? *Er is goed en slecht nieuws. Het slechte nieuws: er zijn geen opvolgers voor functies als* SYS_CONNECT_BY_PATH en CONNECT_BY_ROOT. De helft van het goede nieuws: de nieuwe syntax maakt deze functies overbodig. Zie maar eens hoe eenvoudig we SYS_CONNECT_BY_PATH(ename, '/') - een lijstje van de hierarchie boven een medewerker - en CONNECT_BY_ROOT(job) - de functie van de hoogste baas van een medewerker - implementeren in de recursieve wereld:

```
with employees ( name, empno, hierlevel, hierpath, top_dog_job) as
( select ename
       , empno
       , 1
       , '/'||ename
       , job
    from emp
   where mgr is null
   union all
  select e.ename
       , e.empno
       , m.hierlevel + 1
       , m.hierpath||'/'||e.ename
       , m.top_dog_job
    from emp e
         join
         employees m
         on (m.empno = e.mgr)
)
select lpad(' ', hierlevel *3)||name name
     , hierpath
     , top_dog_job
  from employees
```

De andere helft van het goede nieuws is dat er voor sommige van de geavanceerde syntax wel en soms zelfs rijkere opvolgers zijn. Bijvoorbeeld NOCYCLE en CONNECT_BY_ISCYCLE - functies om loops in hierarchische queries (waar een medewerker indirect zijn eigen manager is) te voorkomen en te detecteren worden opgevolgd door de CYCLE clause

waarmee een extra kolom aan het query resultaat wordt toegevoegd die voor de rij waar de loop wordt gedetecteerd een vlaggetje bevat. De order siblings by heeft een rijkere opvolger gekregen in de vorm van de SEARCH DEPTH FIRST en SEARCH BREADTH FIRST clausules. Deze beide opdrachten voegen ook een extra kolom toe aan het resultaat van de recursieve query. Die kolom bevat voor iedere rij een volgnummer. Dat volgnummer wordt bepaald door in de recursieve query steeds eerst tot op het diepste niveau de "kinderen" op te halen (depth first) dan wel eerst alle nodes van hetzelfde niveau te verwerken (breadth first) alvorens een niveau of iteratie dieper te gaan. Het toegekende volgnummer kan in de uiteindelijke order by clause naar believen worden toegepast. Door een order by te gebruiken in de beide queries in de recursive subquery wordt de eerste voorzet voor de uiteindelijke ordening gegeven.

Om de tree te tonen - depth first - met siblings op alfabetische volgorde van naam is deze query nodig:

```
with employees ( name, empno, hierlevel) as
( select ename
       , empno
       , 1
    from emp
   where mgr is null
   union all
  select e.ename
       , e.empno
       , m.hierlevel + 1
    from emp e
         join
         employees m
         on (m.empno = e.mgr)
)
search depth first by name desc set seq
select lpad(' ', hierlevel *3)||name name
  from employees
 order by seq
```

Merkwaardig genoeg biedt 11gR2 geen vervanger voor de op zich nuttige functie CONNECT_BY_ISLEAF die aangeeft voor een rij of deze wel of geen kinderen oplevert. Met een beetje puzzelen blijkt dat we daar ook achter kunnen komen met een search depth first in combinatie met een CASE statement:

```
with employees ( name, empno, hierlevel) as
...
)
search depth first by name desc set seq
select lpad(' ', hierlevel *3)||name name
,      case
       when (hierlevel - lead(hierlevel) over (order by seq)) < 0
       then 0
       else 1
       end is_leaf
```

```
  from employees
 order by seq
```

De crux van deze oplossing zit hem in search depth first (ga eerst op zoek naar kind-rijen) en de vergelijking met lead tussen het huidige niveau in de hierarchie en het niveau van de eerstvolgende node. Als een node kinderen heeft (en dus geen leaf is) is het niveau van de eerstvolgende node dieper dan de huidige node.

Rij-generatie, Sudoku's oplossen en andere toepassingen

In onze ervaring is er geen uitdaging die we met connect by konden bedwingen die met de recursive subquery niet is op te lossen- en in de meeste gevallen was de 11gR2 recursieve oplossing eleganter. Om een voorbeeld te geven waar de functionaliteit wel maar die elegantie niet werd bereikt: hierarchische queries werden nogal eens toegepast om een bepaald aantal rijen te genereren. Bijvoorbeeld voor domweg een bepaald aantal rijen om mee te joinen of meer specifief alle maanden van het jaar of bijvoorbeeld alle letters van het alfabet:

```
select chr(level+64) letter_in_alphabet
   from dual
connect by level < 27
```

De recursieve tegenhanger van deze query:

```
with alphabet (letter, pos) as
(  select 'A'
        , ascii('A')
     from dual
    UNION ALL
   select chr(alphabet.pos + 1)
        , alphabet.pos + 1
     from alphabet
    where alphabet.pos < ascii('Z')
)
select letter
  from alphabet
```

of iets compacter en cryptischer:

```
with num (pos) as
(  select 1
     from dual
    UNION ALL
   select num.pos + 1
     from num
    where num.pos < 26
)
select chr(pos + 64)
  from num
```

Een heel ander belangwekkend terrein: Onze collega Anton heeft een heel specifieke hobby: SQL queries ontwikkelen die Sudoku's oplossen. Hij heeft een bijzondere oplossing op zijn naam staan op basis van de SQL Model clause. Met behulp van de Recursive Subquery is hij gekomen tot een nog elegantere en beter performende oplossing voor dit nijpende probleem.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Deze en iedere willekeurige 9x9 Sudoku wordt opgelost door Anton's query van minder dan 25 regels, waarin grappig genoeg omwille van de compactheid wel enkele malen een CONNECT BY wordt gebruikt! Anton's Sudoku-oplossing is door Tom Kyte omarmd en wordt door hem regelmatig in zijn presentaties opgevoerd. Zie het blog-artikel van Anton op: http://technology.amis.nl/blog/6404/oracle-rdbms-11gr2-solving-a-sudoku-using-recursive-subquery-factoring voor de bijzonderheden.

Lijstjes, lijstjes, lijstjes

Op het Oracle Technology Network Forum voor SQL en PL/SQL is een regelmatig terugkerende vraag: Hoe kan ik alle namen als een string samenvoegen, gescheiden door een comma? Een voorbeeld om deze vraag te verduidelijken:

```
    DEPTNO STRAGG

---------- ------------------------------------------------------------
        10 CLARK,KING,MILLER
        20 SMITH,JONES,SCOTT,ADAMS,FORD
        30 ALLEN,WARD,MARTIN,BLAKE,TURNER,JAMES
```

Per afdeling willen we een lijstje met namen krijgen van werknemers, gescheiden door een comma.

In oudere versies van de database was het ook altijd wel mogelijk om dit voor elkaar te krijgen, maar niet altijd zonder slag of stoot. Heel vroeger, denk Oracle 8i, kon je gebruik maken van de Oracle Data Cartridge Interface om dit te doen. Tom Kyte, Oracle goeroe bij uitstek, heeft ooit een keer met behulp van de Oracle Data Cartridge Interface een utility geschreven om dit te doen genaamd STRAGG (van String Aggregate). Deze utility is nog steeds goed te gebruiken, maar kent ook zijn beperkingen. Een van de bekende beperkingen is het scheidingsteken, wat als je nu een punt comma wilt in plaats van een comma. Natuurlijk is dit wel aan te passen, en als je een beetje handig bent met Google

dan is er vast wel een versie van STRAGG te vinden waarbij het scheidingsteken is op te geven.

Sinds Oracle 9i is het mogelijk om hetzelfde resultaat te krijgen door gebruik te maken van Analytische Functies in combinatie met een hierarchische query. Zoals in het ondestaande voorbeeld is te zien, een eenvoudige methode is het niet.

```
SQL> select deptno
  2       , ltrim (sys_connect_by_path (ename, ','), ',') scbp
  3    from (select deptno
  4                , ename
  5                , row_number() over (partition by deptno
  6                                         order by empno
  7                                    ) rn
  8             from emp
  9         )
 10   where connect_by_isleaf = 1
 11   start with rn = 1
 12   connect by rn = prior rn + 1
 13      and deptno = prior deptno
 14  /

    DEPTNO SCBP
---------- --------------------------------------------------
        10 CLARK,KING,MILLER
        20 SMITH,JONES,SCOTT,ADAMS,FORD
        30 ALLEN,WARD,MARTIN,BLAKE,TURNER,JAMES
```

In Release 10 van de Database werd de Collect aggregatie operator geïntroduceerd die iets vergelijkbaars doet - maar een collection en geen gedelimiteerde string oplevert. Deze is net zo te gebruiken als min, max, avg en andere aggregatoren - zowel met group by als op een analytische manier - maar werkt met stringwaarden. Collect heeft als resultaat een collection van varchar2 waarden - van een type dat je zelf gedefinieerd hebt met een *create type X as table of VARCHAR2* of een type dat door de database on the fly wordt aangemaakt.

```
SQL> select deptno
  2       , ename
  3       , collect(ename) over (partition by deptno ) colleagues
  4    from emp
  5  /
```

Als je de Collect gebruikt voor gewone aggregatie - dus niet analytisch - kan je ook een order by gebruiken om aan te geven op welke volgorde de string in de collectie moeten worden gezet:

```
SQL> select deptno
  2       , cast (collect (ename order by empno) as my_strings) employees
  3    from emp
```

```
  4   group by deptno
  5  /

    DEPTNO EMPLOYEES
---------- --------------------------------------------------------------------
        10 MY_STRINGS('CLARK', 'KING', 'MILLER')
        20 MY_STRINGS('SMITH', 'JONES', 'SCOTT', 'ADAMS', 'FORD')
        30 MY_STRINGS('ALLEN', 'WARD', 'MARTIN', 'BLAKE', 'TURNER', 'JAMES')
```

En dan is er nog de ongedocumenteerde manier om het gewenste resultaat te krijgen, uiteraard niet aan te raden omdat het ongedocumenteerd is.

```
SQL> select deptno
  2       , wm_concat (ename) stragg
  3    from emp
  4   group by deptno
  5  /

    DEPTNO STRAGG
---------- ------------------------------------
        10 CLARK,MILLER,KING
        20 SMITH,FORD,ADAMS,SCOTT,JONES
        30 ALLEN,JAMES,TURNER,BLAKE,MARTIN,WARD
```

Oracle 11g R2 biedt een eenvoudige, volledig gedocumenteerde methode om het gewenste resultaat te verkrijgen.

```
SQL> select deptno
  2       , listagg (ename, ',') within group (order by empno) stragg
  3    from emp
  4   group by deptno
  5  /
```

Zoals je kunt zien is dit een heel eenvoudige manier om het gewenste effect te krijgen. De LISTAGG functie heeft maximaal twee argumenten. Het eerste argument is hetgene je als string terug wilt krijgen. Het tweede argument is het scheidingsteken, deze mag je achterwege laten. Het is mogelijk een meerdere karakters als scheidingsteken te gebruiken. In de WITHIN GROUP geeft je aan op welke manier de lijst gesorteerd dient te worden. Deze nieuwe functie is ook als analytische functie te gebruiken, in dat geval komt er de OVER clause bij.

Analytische Functies 2.0

Over analytische functies gesproken, ook op dit vlak biedt Oracle 11g R2 een paar interessante uitbereidingen. Analytische Functies bestaan sinds Oracle 8.1.6. Enterprise Edition, maar zijn sinds Oracle 9*i* deel van de Standard Edition. Zoals zojuist in de laatste paragraaf genoemd, de LISTAGG functie is ook in de analytische variant te gebruiken. Meteen maar een klein voorbeeldje om dit te demonstreren.

```
SQL> select listagg (ename, ', ') within group (order by empno)
  2         over (partition by deptno) employees
```

```
  3    from emp
  4  /

EMPLOYEES
------------------------------------------------------------
CLARK, KING, MILLER
CLARK, KING, MILLER
CLARK, KING, MILLER
SMITH, JONES, SCOTT, ADAMS, FORD
SMITH, JONES, SCOTT, ADAMS, FORD
SMITH, JONES, SCOTT, ADAMS, FORD
SMITH, JONES, SCOTT, ADAMS, FORD
SMITH, JONES, SCOTT, ADAMS, FORD
ALLEN, WARD, MARTIN, BLAKE, TURNER, JAMES
ALLEN, WARD, MARTIN, BLAKE, TURNER, JAMES
ALLEN, WARD, MARTIN, BLAKE, TURNER, JAMES
ALLEN, WARD, MARTIN, BLAKE, TURNER, JAMES
ALLEN, WARD, MARTIN, BLAKE, TURNER, JAMES
ALLEN, WARD, MARTIN, BLAKE, TURNER, JAMES
```

Het grote verschil met de gewone variant van LISTAGG is dat er nu een resultaat wordt gegeven voor iedere rij in de resultaat set, net wat je van analytische functies kan verwachten.

De analytische functies FIRST_VALUE en LAST_VALUE hebben een generieker broertje erbij gekregen, de Nth_VALUE. Waar de FIRST_VALUE de eerste, en de LAST_VALUE de laatste waarde uit de Partition van de resultaatset ophalen, haalt de Nth_VALUE iedere gewenste waarde op. Stel dat je wilt weten wie het meeste verdient binnen een afdeling, dan zou je zo'n query kunnen schrijven.

```
SQL> select deptno
  2       , ename
  3       , first_value (ename) over (partition by deptno
  4                                       order by sal desc
  5                                  ) most_earning_colleague
  6    from emp
  7  /

    DEPTNO ENAME      MOST_EARNI
---------- ---------- ----------
        10 KING       KING
        10 CLARK      KING
        10 MILLER     KING
        20 FORD       FORD
        20 SCOTT      FORD
        20 JONES      FORD
        20 ADAMS      FORD
        20 SMITH      FORD
        30 BLAKE      BLAKE
```

```
        30 ALLEN      BLAKE
        30 TURNER     BLAKE
        30 MARTIN     BLAKE
        30 WARD       BLAKE
        30 JAMES      BLAKE
```

Indien juist gewenst is om te weten wie er op-een-na het meest verdient, dan is het niet mogelijk om de FIRST_VALUE functie te gebruiken, de Nth_VALUE echter wel.

```
nth_value (ename, 2) from first
         over (partition by deptno
               order by sal desc
              )
```

Bij de Nth_VALUE geef je als tweede argument aan welke waarde je uit de resultaat set wilt hebben, van welke rij. In gewoon Nederlands staat er: Toon mij de naam van de medewerker die op de tweede rij staat als ik sorteer op salaris van hoog naar laag. Op gelijke wijze is iedere rij te benaderen, uiteraard ook de eerste en de laatste. De nieuwe variant van de FIRST_VALUE ziet er dan als volgt uit:

```
nth_value (ename, 1) from first
         over (partition by deptno
               order by sal desc
              )
```

De nieuwe variant van de LAST_VALUE wordt dan:

```
nth_value (ename, 1) from last
         over (partition by deptno
                    order by sal desc
              )
```

Wat overigens niet hetzelfde is als:

```
nth_value (ename, 1) from first
         over (partition by deptno
                    order by sal
              )
```

Deze laatste variant van de Nth_Value heeft een andere Windowing Clause (de ORDER BY).

```
SQL> select deptno
  2       , ename
  3       , sal
  4       , nth_value (ename, 1) from last
  5           over (partition by deptno
  6                      order by sal desc
  7                 ) anchor
  8       , nth_value (ename, 1) from first
  9           over (partition by deptno
```

```
 10                     order by sal
 11                 ) lowest_sal
 12    from emp
 13  /

    DEPTNO ENAME             SAL ANCHOR     LOWEST_SAL
---------- ---------- ---------- ---------- ----------
        10 KING             5000 KING       MILLER
        10 CLARK            2450 CLARK      MILLER
        10 MILLER           1300 MILLER     MILLER
        20 FORD             3000 SCOTT      SMITH
        20 SCOTT            3000 SCOTT      SMITH
        20 JONES            2975 JONES      SMITH
        20 ADAMS            1100 ADAMS      SMITH
        20 SMITH             800 SMITH      SMITH
        30 BLAKE            2850 BLAKE      JAMES
        30 ALLEN            1600 ALLEN      JAMES
        30 TURNER           1500 TURNER     JAMES
        30 MARTIN           1250 WARD       JAMES
        30 WARD             1250 WARD       JAMES
        30 JAMES             950 JAMES      JAMES
```

De LAST_VALUE functie had in Oracle 10*g* al een IGNORE NULLS clause, maar tegenwoordig hebben de LAG en de LEAD functies deze clausule ook. Uiteraard is het ook bij de Nth_VALUE mogelijk om de IGNORE NULLS clausule te gebruiken. Deze clausule doet precies wat je ervan zou verwachten: NULLs worden genegeerd bij het gebruik van de LAG en LEAD functies.

```
SQL> select ename
  2       , comm
  3       , lag (comm) over (order by empno) regular
  4       , lag (comm) ignore nulls over (order by empno) ignoring
  5    from emp
  6   where deptno = 30
  7   order by empno
  8  ;

ENAME            COMM    REGULAR IGNORING
---------- ---------- ---------- --------
ALLEN             300
WARD              500        300 300
MARTIN           1400        500 500
BLAKE                       1400 1400
TURNER              0           1400
JAMES                          0 0
```

In bovenstaande voorbeeld is het effect van de IGNORE NULLS clause te zien. Als we kijken naar de rij voor Turner, dan is te zien dat er in de kolom met de naam "REGULAR" een NULL staat. Dit wordt veroorzaakt doordat de rij voorafgaand aan Turner geen COMM heeft. Kijken we naar de kolom "IGNORING" dan is de waarde 1400 te zien. In het geval van de

IGNORE NULLS clause wordt er net zolang in de resultaat set gekeken totdat er een NOT NULL waarde wordt gevonden.

Wil je nu heel expliciet zijn, dan is er ook een RESPECT NULLS, deze clausule houdt wel rekening met NULL. De default is dan ook RESPECT NULLS.

Overige functionaliteit

Het moge ondertussen duidelijk zijn, tot nu toe hebben we nog geen hele schokkende dingen laten zien wat er in Oracle 11g R2 toegevoegd is. In de volgende Optimize staat Edition Based Redefinition op het programma. Edition Based Redefinition is dé reden om te upgraden naar Oracle 11g R2.

Dat dus in het volgende nummer. In dit artikel willen we op de valreep nog kort een paar andere 11gR2 vernieuwingen doornemen - niet in detail maar wel zodat je weet van het bestaan.

Met het package DBMS_PARALLEL_EXECUTE kan je doen wat al diverse manieren - dbms_job, pipelined table functions, meerdere sessies die via pipes synchroniseren - werd nagestreefd: parallelle executie van PL/SQL code of SQL statements. Een statement wordt aan het package aangeboden om op een set waarden of records te worden uitgevoerd. Daarbij kunnen we aangeven hoe die set rijen in parallel te verwerken blokken moet worden opgedeeld.

Met de introductie van de FORCE toevoeging in het CREATE or REPLACE TYPE statement wordt het leven voor database administrators een stuk makkelijker. Voor 11gR2 was het erg moeilijk om user defined of abstract data types waar dependencies op bestonden te wijzigen. Het kwam er meestal op neer dat alle types moesten worden gedropped en vervolgens weer allemaal aangemaakt.

Aan een External Table kan een zogenaamde preprocessor worden toegevoegd: een O/S script dat de file waarop de tabel is gebaseerd gaat verwerken - unzippen, opschonen,... - alvorens de database engine deze gaat lezen. Deze constructie kan worden 'misbruikt' om vanuit de database scripts te starten op het operating system - zonder dat er ook echt gelezen moet worden uit de external table.

Met het package dbms_scheduler, en meer specifiek de procedure create_file_watcher, kan de database geinstrueerd worden om een specifieke PL/SQL procedure aan te roepen wanneer er een file wordt geschreven naar de geconfigueerde directory. De procedure kan vervolgens overgaan tot het verwerken van die file. Dit doet in de verte wat denken aan de File Adapter van de Oracle SOA Suite.

De Function Result Cache die in 11gR1 werd geintroduceerd is een tikkeltje slimmer geworden: de database bekijkt welke tabel afhankelijkheden de functie heeft en wanneer er een DML operatie plaatsvindt op een van deze tabellen wordt de function result cache automatische gereset. In 11gR1 moest je expliciet opgeven welke tabellen aan de basis stonden van de functie resultaten.

De tamelijk onopgemerkt gebleven functionaliteit in het package DBMS_FLASHBACK om een transactie ongedaan te maken - procedure TRANSACTION_BACKOUT - is uitgebreid met de intelligentie om een CASCADE back out te doen voor transacties die via Foreign Key afhankelijkheden geraakt worden voor de initiele transactie rollback. Stel je voor: T1 creeert een nieuw department. T2 doet een update van een employee en koppelt hem aan dat

nieuwe department. Als T1 ongedaan wordt gemaakt zou T2 in een incorrecte staat - verwijzing naar een dan niet meer bestaande department - achterblijven. Het package onderkent deze situatie nu en in tegenstelling tot in 11gR1 kan de cascade automatische worden uitgevoerd voor afhankelijke transacties.

Het moge ondertussen duidelijk zijn, tot nu toe hebben we nog geen hele schokkende dingen laten zien wat er in Oracle 11g R2 toegevoegd is. In het volgende nummer staat Edition Based Redefinition op het programma. Edition Based Redefinition is dé reden om te upgraden naar Oracle 11g R2.

Dat dus in het volgende nummer. In dit artikel willen we op de valreep nog kort een paar andere 11gR2 vernieuwingen doornemen - niet in detail maar wel zodat je weet van het bestaan.
Met het package DBMS_PARALLEL_EXECUTE kan je doen wat al diverse manieren - dbms_job, pipelined table functions, meerdere sessies die via pipes synchroniseren - werd nagestreefd: parallelle executie van PL/SQL code of SQL statements. Een statement wordt aan het package aangeboden om op een set waarden of records te worden uitgevoerd. Daarbij kunnen we aangeven hoe die set rijen in parallel te verwerken blokken moet worden opgedeeld.

Met de introductie van de FORCE toevoeging in het CREATE or REPLACE TYPE statement wordt het leven voor database administrators een stuk makkelijker. Voor 11gR2 was het erg moeilijk om user defined of abstract data types waar dependencies op bestonden te wijzigen. Het kwam er meestal op neer dat alle types moesten worden gedropped en vervolgens weer allemaal aangemaakt.
Aan een External Table kan een zogenaamde preprocessor worden toegevoegd: een O/S script dat de file waarop de tabel is gebaseerd gaat verwerken - unzippen, opschonen,... - alvorens de database engine deze gaat lezen. Deze constructie kan worden 'misbruikt' om vanuit de database scripts te starten op het operating system - zonder dat er ook echt gelezen moet worden uit de external table.
Met het package dbms_scheduler, en meer specifiek de procedure create_file_watcher, kan de database geinstrueerd worden om een specifieke PL/SQL procedure aan te roepen wanneer er een file wordt geschreven naar de geconfigueerde directory. De procedure kan vervolgens overgaan tot het verwerken van die file. Dit doet in de verte wat denken aan de File Adapter van de Oracle SOA Suite.

De Function Result Cache die in 11gR1 werd geintroduceerd is een tikkeltje slimmer geworden: de database bekijkt welke tabel afhankelijkheden de functie heeft en wanneer er een DML operatie plaatsvindt op een van deze tabellen wordt de function result cache automatische gereset. In 11gR1 moest je expliciet opgeven welke tabellen aan de basis stonden van de functie resultaten.
De tamelijk onopgemerkt gebleven functionaliteit in het package DBMS_FLASHBACK om een transactie ongedaan te maken - procedure TRANSACTION_BACKOUT - is uitgebreid met de intelligentie om een CASCADE back out te doen voor transacties die via Foreign Key afhankelijkheden geraakt worden voor de initiele transactie rollback. Stel je voor: T1 creeert een nieuw department. T2 doet een update van een employee en koppelt hem aan dat nieuwe department. Als T1 ongedaan wordt gemaakt zou T2 in een incorrecte staat - verwijzing naar een dan niet meer bestaande department - achterblijven. Het package onderkent deze situatie nu en in tegenstelling tot in 11gR1 kan de cascade automatische worden uitgevoerd voor afhankelijke transacties.

Tot zover de eerste kennismaking met de nieuwe faciliteiten voor ontwikkelaars in Oracle Database 11g Release 2. In het volgende nummer van Optimize gaan we verder - over Edition Based Redefinition.