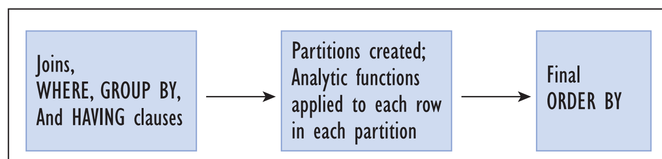
DBA

Oracle Analytische Functies

De keuze: één functie of 1000 regels code

Met Oracle Analytische SQL Functies kun je data vanuit verschillende rijen gelijktijdig ophalen, zonder dat daar een self join voor nodig is. Het is mogelijk om een rangvolgorde aan te geven binnen een groep van waarden. Veel query's worden een stuk eenvoudiger en eleganter. Dit betekent echter niet dat Analytische Functies eenvoudig zijn. Soms lijkt het alsof je hersenen een flikflak achterover met een hele schroef maken, maar wat een voldoening als je de meer uitdagende programmeerpuzzels kan oplossen. De performance is buitengewoon!

Het verwerken van een query met Analytische Functies bestaat uit drie stappen. Allereerst worden de Joins, Where condities, Group By en Having clausules toegepast. Dan worden de Analytische Functies losgelaten op de resultaatset die de eerste stap oplevert. Als allerlaatste wordt de ORDER BY toegepast.



Figuur 1. Verwerkingsvolgorde van een Query.

Partities

Op de resultaatset worden de Partities gecreëerd. Een Partitie is een groep van logisch bij elkaar horende rijen, zoals bijvoorbeeld alle werknemers van eenzelfde afdeling. De grootte van een partitie is variabel. Het kan zo klein zijn als een enkele rij of zo groot als de totale resultaatset. Dit voorbeeld toont het gemiddelde salaris voor alle werknemers binnen dezelfde afdeling met dezelfde baan:

```
avg(sal) over (partition by deptno, job)
```

Als de partitieclausule wordt weggelaten, dan is de gehele resultaatset de partitie. Per partitie wordt de Analytische Functie 'op nul gezet'. Stel je wilt een overzicht krijgen van alle salarissen met subtotaal per afdeling: sum(sal) over (partition by deptno). In dit geval wordt de teller op nul gezet per afdeling.

Windows

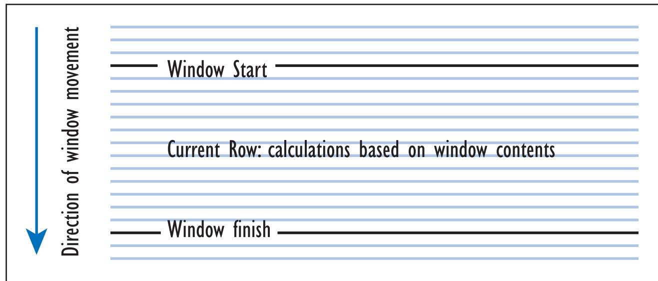
Binnen een partitie is het mogelijk om een window te definiëren. Het Window is altijd omsloten door een partitie. Het Window bepaalt het aantal rijen in de huidige Partitie waarvoor de bewerking moet gelden. Analytische Functies worden altijd uitgevoerd ten opzichte van de huidige rij. De huidige rij is dan ook het referentiepunt voor het Window.
Default is het Window RANGE UNBOUNDED PRECEDING. Dit houdt in dat het Window met iedere rij steeds groter wordt. De eerste rij in het Window is dan de eerste rij van de Partitie en de laatste rij is de huidige rij. Hoewel het nu een beetje cryptisch klinkt, word het duidelijk in de volgende paragrafen.
Windows komen in twee smaken. De eerste is de zogenaamde anchored window, de tweede een sliding window. Default is de anchored window. Dit type window begint bij de eerste rij van de partitie en eindigt bij de huidige rij. Het wordt een anchored window genoemd omdat de eerste vast geankerd is. Bij een sliding window is dit niet het geval. Een sliding window heeft, in tegenstelling tot een anchored window geen vast beginpunt. Het beginpunt wordt bepaald door óf een numerieke offset óf een aantal rijen voor de huidige rij. De eerste variant wordt een range window genoemd, de tweede een row window.
Een voorbeeld van een range window: toon alle rijen met werknemers waarvoor geld dat het salaris binnen 200 euro van de huidige rij valt:

```
over (order by sal range between 200 preceding and
200 following)
```

Dit type window werkt alleen op numbers of dates. Voor dates geldt dan het aantal dagen. Bij het andere type sliding window, de row window, geeft je aan hoeveel rijen je voor- of achteruit wilt kijken. Bijvoorbeeld: bereken over (niet meer dan) twee

rijen voor tot de huidige rij en niet meer dan twee rijen na de huidige rij, over alle rijen gesorteerd op salaris;

```
over (order by sal rows between 2 preceding and 2
following)
```



Figuur 2. Windows.

Analytische Functies

Een aantal Analytische Functies bekend uit de 'gewone' SQL hebben ook een Analytische tegenhanger. Bijvoorbeeld SUM, COUNT of AVG. De Analytische tegenhanger is te herkennen aan het reserved word OVER. Een groot aantal van de Analytische Functies zijn veel cryptischer: STDDEV_SAMP, VAR_POP of CUME_DIST.
Er is ook nog een groepje Analytische Functies dat in eerste instantie een beetje vreemd overkomt, maar eigenlijk in de dagelijkse werkzaamheden goed te gebruiken is. Denk bijvoorbeeld aan ROW_NUMBER. Je kunt deze beschouwen als de analytische tegenhanger van ROWNUM. Met ROW_NUMBER kun je een nummer toekennen aan een rij per partitie. Dit is een goede functie om vragen te beantwoorden als: "Toon mij de drie grootste verdieners per afdeling" of "Wie heeft het twee na hoogste salaris?"

> ***Zoals voor alle nieuwe features geldt: het is niet alleen maar rozengeur***

RANK en DENSE_RANK zijn zogenaamde ranking-functies, zij geven een waarde aan in een rij binnen een partitie. LAG en LEAD geven de mogelijkheid om waardes uit andere rijen te benaderen.
Laten we eens gaan kijken naar een aantal van de meest voorkomende gebruiken van Analytische Functies. Dit zijn slechts een aantal voorbeelden om een beetje de smaak te pakken te krijgen. Het is natuurlijk ook mogelijk om vergelijkbare query's te schrijven met 'huis-tuin en keuken'-SQL, maar met behulp van Analytische Functies ziet de query er een stuk eenvoudiger uit. Een leuke bijkomstigheid is dat het vaak ook nog een betere performance geeft.
In de navolgende voorbeelden zal ik gebruik maken van de EMP en DEPT-tabellen. De informatie die wordt opgeslagen heeft betrekking op Medewerkers en Afdelingen. Ze zijn deel van het SCOTT-schema, één van de demo-schema's die al heel wat Oracle-versies meegaat.

Subtotalen

Een meelopend subtotaal is verassend eenvoudig te maken met behulp van Analytische Functies. Om hetzelfde resultaat te bewerkstelligen op een traditionele manier kan een aardige uitdaging zijn. Dit is een stukje voorbeeldcode:

```
SQL> select empno
  2      , ename
  3      , sal
  4      , Sum (sal) over (order by empno) overall_total
  5   from emp
  6  order by empno
  7  /
```

```
EMPNO ENAME             SAL OVERALL_TOTAL
----- ---------- ---------- -------------
 7369 SMITH             800           800
 7499 ALLEN            1600          2400
 7521 WARD             1250          3650
 7566 JONES            2975          6625
 7654 MARTIN           1250          7875
 7698 BLAKE            2850         10725
 7782 CLARK            2450         13175
 7788 SCOTT            3000         16175
 7839 KING             5000         21175
 7844 TURNER           1500         22675
 7876 ADAMS            1100         23775
 7900 JAMES             950         24725
 7902 FORD             3000         27725
 7934 MILLER           1300         29025
```

De SUM functie die in de bovenstaande code staat is de Analytische Functie. Na de OVER staan de condities waar deze Analytische Functie voor geldt. Hier ontbreekt het Partition sleutelwoord. In dit geval is de totale resultaat de partitie. Er is echter wel een window. Dit vloeit voort uit de ORDER BY in regel 4. Het –impliciete- window is RANGE UNBOUNDED PRECEDING, het default voor de Window clausule. Dit houdt in dat alle voorgaande salarissen opgeteld worden bij het salaris van de huidige rij. Hierdoor krijg je dus een meelopend subtotaal. Als we nu naar WARD kijken (met medewerkernummer 7521) zien we dat de overall_total kolom een waarde toont van 3650. Dit is een optelling van alle voorgaande salarissen, inclusief zijn eigen salaris. (800 + 1600 + 1250).
Indien je niet geïnteresseerd bent in subtotalen van het hele

bedrijf, maar alleen in de subtotalen per afdeling, dan hoef je alleen maar een Partition-clausule toe te voegen, zoals in het volgende voorbeeld.

```
SQL> select empno
  2    , ename
  3    , sal
  4    , deptno
  5    , Sum (sal) over (partition by deptno
  6                     order by empno
  7                    ) department_total
  8   from emp
  9   order by deptno, empno
 10  /
```

```
 EMPNO ENAME             SAL     DEPTNO DEPARTMENT_TOTAL
------ ---------- ---------- ---------- ----------------
  7782 CLARK            2450         10             2450
  7839 KING             5000         10             7450
  7934 MILLER           1300         10             8750
  7369 SMITH             800         20              800
  7566 JONES            2975         20             3775
  7788 SCOTT            3000         20             6775
  7876 ADAMS            1100         20             7875
  7902 FORD             3000         20            10875
  7499 ALLEN            1600         30             1600
  7521 WARD             1250         30             2850
  7654 MARTIN           1250         30             4100
  7698 BLAKE            2850         30             6950
  7844 TURNER           1500         30             8450
  7900 JAMES             950         30             9400
```

Als we wederom kijken naar Ward (7521), dan zien we dat Department_total een waarde heeft van 2850. Dit is een optelling van alle voorgaande salarissen binnen zijn eigen afdeling (1600 + 1250). Het is overigens niet nodig om de ORDER BY in regel 9 gelijk te houden aan de ORDER BY van de Analytische Functie. Dit is enkel gedaan om het eindresultaat overzichtelijk te maken. Het kan nogal verwarrend zijn als de sorteringen anders zijn. Om de Windows inzichtelijk te maken kan het gebruik van FIRST_VALUE en LAST_VALUE helpen. Deze tonen respectievelijk de eerste of de laatste rij uit het Window.

```
SQL> select ename
  2    , Sum (sal) over (partition by deptno
  3                     order by empno
  4                    ) dept_total
  5    , First_Value (ename) over (partition by deptno
  6                               order by empno
  7                              ) fv
  8    , Last_Value (ename) over (partition by deptno
  9                              order by empno
 10                             ) lv
 11   from emp
 12   where deptno = 20
 13   order by deptno
 14          , empno
 15  /
```

```
ENAME      DEPT_TOTAL FV         LV
---------- ---------- ---------- -------
SMITH             800 SMITH      SMITH
JONES            3775 SMITH      JONES
SCOTT            6775 SMITH      SCOTT
ADAMS            7875 SMITH      ADAMS
FORD            10875 SMITH      FORD
```

Hier zie je dat het Window per rij groter wordt. Als we naar Scott kijken, dan is te zien dat het Window begint bij Smith en eindigt bij de huidige rij (Scott).

Ranking: Top N

Om een Top 3 van meest verdienenden per afdeling te maken is niet veel nodig. De werkwijze is als volgt. We delen het hele bedrijf in Partities per afdeling, vervolgens geven we iedereen een volgnummer gebaseerd op zijn/haar salaris. Als laatste filteren we hier de nummers één, twee en drie uit. Om de nummers één, twee en drie uit te delen, zijn er drie verschillende Analytische Functies die in aanmerking komen. RANK, DENSE_RANK en ROW_NUMBER. Allen delen een nummer uit op basis van de ORDER BY clausule binnen de Partitie. Ze doen het ieder op hun eigen manier. Het verschil zit hem in de manier hoe ze met gelijkspel omgaan. RANK staat toe dat er nummers worden overgeslagen, DENSE_RANK staat dit niet toe. ROW_NUMBER deelt een willekeurig nummer als het niet mogelijk is om een onderscheid te maken op basis van de ORDER BY clausule. Dit laatste is te vergelijken met ROWNUM.
Om het een en ander te verduidelijken een voorbeeld. Hier zijn de verschillende manieren van Ranken naast elkaar gezet. Er zijn twee personen met hetzelfde hoge salaris, te weten Scott en Ford. Beide verdienen 3000, en hebben volgens RANK en DENSE_RANK recht op de eerste plaats. Jones, die 2975 verdient heeft een RANK van 3, terwijl de DENSE_RANK zegt dat het 2 is. RANK heeft de tweede plaats simpelweg overgeslagen; we delen geen zilveren medailles uit als er al twee gouden vergeven zijn. De laatste kolom in dit voorbeeld toont ROW_NUMBER. Deze deelt een willekeurig nummer uit aan de twee met het hoogste salaris.

```
SQL>  select ename
  2    , deptno
  3    , sal
  4    , Rank() over (partition by deptno
  5                  order by sal desc
  6                 ) rk
  7    , Dense_Rank() over (partition by deptno
  8                        order by sal desc
  9                       ) dr
 10    , Row_Number() over (partition by deptno
 11                        order by sal desc
 12                       ) rn
```

```
13  from emp
14  where deptno = 20
15  order by deptno
16              , sal desc;
17
```

```
ENAME          DEPTNO        SAL         RK         DR         RN
---------- ---------- ---------- ---------- ---------- ----------
SCOTT              20       3000          1          1          1
FORD               20       3000          1          1          2
JONES              20       2975          3          2          3
ADAMS              20       1100          4          3          4
SMITH              20        800          5          4          5
```

Aangezien een Analytische Functie niet in de Where-clausule gebruikt mag worden, is het noodzakelijk om deze te verbergen in een in-line view. De uiteindelijke query is dan als volgt:

```
SQL> select ename
  2     , deptno
  3     , sal
  4     , rn
  5  from (select ename
  6           , deptno
  7           , sal
  8           , Row_Number() over (partition by deptno
  9                                order by sal desc
 10                               ) rn
 11        from emp
 12       )
 13  where rn <= 3
 14  /
```

```
ENAME          DEPTNO        SAL         RN
---------- ---------- ---------- ----------
KING               10       5000          1
CLARK              10       2450          2
MILLER             10       1300          3
SCOTT              20       3000          1
FORD               20       3000          2
JONES              20       2975          3
BLAKE              30       2850          1
ALLEN              30       1600          2
TURNER             30       1500          3
```

Pivot

Nu je begrijpt hoe ranking functies werken, is het ook erg leuk om er een andere draai aan te geven. In dit geval bedoel ik het eindresultaat. In plaats van de Top 3 te tonen zoals hierboven, kunnen we het eindresultaat draaien op zo'n manier dat het van links naar rechts gaat in plaats van boven naar beneden. Om dit voor elkaar te krijgen nemen we de query van hierboven en wijzigen deze licht tot:

```
SQL> select deptno
  2     , Max (Decode (rn, 1, ename)) "Top 1"
  3     , Max (Decode (rn, 2, ename)) "Top 2"
  4     , Max (Decode (rn, 3, ename)) "Top 3"
  5  from (select ename
  6           , deptno
  7           , Row_Number() over (partition by deptno
  8                                order by sal desc
  9                               ) rn
 10        from emp
 11       )
 12  where rn <= 3
 13  group by deptno
 14  /
```

```
DEPTNO Top 1      Top 2      Top 3
------- ---------- ---------- -------
     10 KING       CLARK      MILLER
     20 SCOTT      FORD       JONES
     30 BLAKE      ALLEN      TURNER
```

Hoewel het niet altijd nodig is om een Analytische Functie te gebruiken om dit resultaat te krijgen, is het voor een Top 3 zoals hierboven heel eenvoudig.

Andere rijen in resultaatset

Soms is het nodig om waarden van andere rijen in de resultaatset te gebruiken. Bijvoorbeeld om de toename ten opzichte van de voorganger te tonen. Dat is precies waar de functies LAG en LEAD voor zijn. LAG kijkt naar waarden in voorgaande

> ***Het duurt even voor je gewend bent aan Analytische Functies***

rijen: rijen die op grond van de ORDER BY clause in de Analytische expressie eerder in de partitie staan dan de huidige rij. LEAD kijkt naar de rijen die nog moeten komen. De huidige rij is altijd het uitgangspunt bij de bepaling van het aantal rijen dat je voor- of achteruit wilt kijken. De noodzaak voor een self-join is hiermee in veel gevallen verdwenen. Zoals altijd zegt een stukje code meer dan duizend woorden, dus laten we eens een voorbeeld bekijken. We zijn geïnteresseerd in de eerstvolgende – lead(ename,1) medewerker die met dezelfde job – partition by job - na een medewerker in dienst is gekomen – order by hiredate.:

```
SQL> select ename, job
  2     , hiredate
  3     , Lead (ename) over (partition by job
  4                          order by hiredate
  5                         ) next_hiree_in_job
  6  from emp
  7  order by job
  8        , hiredate;
```

```
ENAME      JOB       HIREDATE  NEXT_HIREE
---------- --------- --------- ----------
FORD       ANALYST   03-DEC-81 SCOTT
SCOTT      ANALYST   09-DEC-82
SMITH      CLERK     17-DEC-80 JAMES
JAMES      CLERK     03-DEC-81 MILLER
MILLER     CLERK     23-JAN-82 ADAMS
ADAMS      CLERK     12-JAN-83
JONES      MANAGER   02-APR-81 BLAKE
BLAKE      MANAGER   01-MAY-81 CLARK
CLARK      MANAGER   09-JUN-81
KING       PRESIDENT 17-NOV-81
ALLEN      SALESMAN  20-FEB-81 WARD
WARD       SALESMAN  22-FEB-81 TURNER
TURNER     SALESMAN  08-SEP-81 MARTIN
MARTIN     SALESMAN  28-SEP-81
```

In dit voorbeeld heb ik de LEAD functie gebruikt om naar opvolgende rijen te kijken. We hebben de resultaatset op grond van JOB in partities verdeeld en iedere partitie op Hiredate gesorteerd. Als we kijken naar bijvoorbeeld de Managers dan zie je dat voor Blake de eerstvolgende collega Clark is. Er zijn geen managers meer in dienst gekomen na Clark, dus de Next_Hiree voor Clark is null.
Het is ook mogelijk verder naar voren of achteren te kijken binnen het window. LAG en LEAD hebben nog twee optionele parameters, naast de eerste verplichte parameter die expressie beschrijft die voor de gerefereerde rij moet worden geëvalueerd. De eerste geeft het aantal rijen aan dat naar voren of achteren wordt gekeken. De default is 1. De andere is de waarde die moet worden opgeleverd als LAG of LEAD aan een niet bestaande rij refereert – zoals in het voorbeeld de next_hiree voor Clark.

Addertjes onder het gras

Zoals voor alle nieuwe features geldt: het is niet alleen maar rozengeur. Zodra je de syntax van de Analytische Functies onder de knie hebt, zijn de mogelijkheden onbegrensd. Er is echter wel een aantal aandachtspunten:
Analytische Functies kunnen niet in de WHERE clause of de afsluitende ORDER BY worden gebruikt. Deze beperking kan vaak makkelijk worden omzeild met een in-line view die de Analytische Functie bevat en met een Kolom Alias in de Order By clause. Je moet ook de ordening van NULL values in de gaten houden en de invloed daarvan op de ORDER BY clause van het Window van de Analytische Functie. We zullen later nog een voorbeeld zien. Natuurlijk is performance een belangrijk aandachtspunt. Hoewel Analytische Functies in veel omstandigheden de meest elegante en vaak ook best performende oplossing bieden, neemt dat niet weg dat het gebruik van meerdere verschillende windows en order by definities snel leidt tot flink wat sorteer- en filter-operaties. Deze kunnen een aanmerkelijke invloed hebben op de totale query-performance. Ook voor Analytische Functies geldt dat je de code terdege moet testen – onder productieomstandigheden - voor je deze in productie neemt.
Tenslotte een waarschuwing vanuit mijn eigen ervaring. Als je vertrouwd raakt met Analytische Functies is het voor je het weet een hamer die in alle SQL-uitdagingen een spijker ziet. Even een voorbeeldje van hoe je daarin gemakkelijk te ver kunt gaan. We willen graag een lijstje krijgen van de namen die meerdere keren voorkomen in een EMP-tabel. Je kunt dat met je Analytische hamer als volgt aanpakken:

```
SQL> select distinct
  2      ename
  3   from (select ename
  4          , Row_Number() over (partition by ename
  5                               order by null
  6                              ) rn
  7       from big_emp
  8      )
  9   where rn > 1;

ENAME
----------
ADAMS
ALLEN
BLAKE
```

De meer traditionele aanpak – even de hamer aan de kant leggen – is zoiets als dit;

```
SQL> select ename
  2   from big_emp
  3   group by ename

  4  having Count(*) > 1;

ENAME
----------
ADAMS
ALLEN
BLAKE
```

Bij een test van deze beide query's op tabellen met een slordige 200.000 rijen bleek dat de traditionele aanpak een iets betere performance heeft en aanzienlijk minder resources gebruikt.

Daad bij het woord

Ik werd recent op een project door een college aangesproken. Hij liet me de query zien die hij net had afgerond. Na een paar minuten aandachtig studeren had ik nog steeds geen flauw idee wat de query zou moeten doen. Het commentaar in de code - "Haal het huidige contract op" - was maar beperkt nuttig. Gelukkig kon ik de ontwikkelaar vragen me uit te leggen wat nu precies de specificatie was waarvoor hij de query had geschreven.

Deze luidde: "Retourneer het huidige contract. Je kunt de huidige contracten vinden door naar de EIND_DATUM te kijken. Het huidige contract heeft de meest recente einddatum of zelfs helemaal geen waarde voor einddatum. Als er meerdere contracten zijn met dezelfde einddatum (of er zijn er meerdere

De huidige rij is altijd het uitgangspunt bij de bepaling van het aantal rijen dat je voor- of achteruit wilt kijken

met een niet ingevulde einddatum), dan moet je kijken naar de begindatum: het contract met de meest recente begindatum is namelijk in dat geval het huidige contract. Een lege begindatum is de verst in het verleden liggende begindatum. Als alle einddatums en begindatums gelijk zijn – nulls inbegrepen – dan maakt het niet uit welk contract je teruggeeft. Doe maar die met de hoogste ID waarde. Oh, trouwens, dit alles moet worden gedaan per categorie van contracten."
Hij had geprobeerd te realiseren door tweemaal dezelfde tabel te queriën en gebruik te maken van verschillende geneste DECODE statements om de einddatum van de hoofdquery te vergelijken met die in de geneste query. Ook voor de vergelijking van de begindatum gebruikte hij verschillende geneste decodes. En nogmaals ditzelfde voor de hoogste ID-waarde. In zekere zin bouwde hij zijn eigen variant op de LAG en LEAD functies die we eerder hebben gezien.
Het duurde even voor ik het allemaal door had. Neem dus ook gerust je tijd. Pas toen hij zei, tegen het eind: 'per categorie' viel het kwartje: dat klonk als een partition clause voor een Analytische Functie! Natuurlijk is alleen een Partition Clause niet voldoende voor een geldige Analytische expressie, we moeten ook nog de functie zelf hebben.
Vanwege zijn uitleg van de specificatie met 'zoek de hoogste waarde' en 'vergelijk deze einddatum met de volgende' waren mijn volgende gedachtes om de Analytische MAX te gebruiken, samen met LAG en LEAD om de vergelijking uit te voeren.
Na enig gestoei met deze functies en een uur en enkele bakken sterke koffie verder, drong het tot me door dat ik iets heel anders nodig had. Ik was bezig de oplossing van mijn collega over te bouwen, alleen nu met Analytische functies.
Het bleek goed mogelijk om de specificatie te formuleren als: "zoek de top 1 van de contracten per categorie". Dit klinkt meer als een ROW_NUMBER of RANK uitdaging! Omdat ID the primary key is, maak ik gebruik van ROW_NUMER.
Daarmee kwam ik uit op de volgende query:

```
select id
    , cat
    , start_date
    , end_date
  from (select id
          , cat
          , start_date
          , end_date
          , Row_Number() over (partition by cat
                        order by end_date desc nulls
first
                              , start_date desc
nulls last
                              , id desc
                        ) rn
        from contract
      )
  where rn = 1
```

De Top 1 wordt bepaald door de meest recente einddatum, vervolgens de meeste recente begindatum en uiteindelijk de hoogste ID-waarde. Met DESC in het sorteren van elke partitie sorteert de waarden van hoog naar laag. De NULLS FIRST en NULLS LAST zijn geïntroduceerd in Oracle 8i; we gebruiken ze om aan te geven hoe NULL-voorkomens tijdens sorteren moeten worden behandeld: helemaal vooraan of juist helemaal achteraan. Overigens kunnen ze ook in gewone ORDER BY clausules worden gebruikt, en dus niet alleen binnen Analytische Expressies.

Winst- en verliesrekening

Het ziet er zo simpel en elegant uit. Ik weet wel welk van de twee voorgaande query's ik het liefst zou willen onderhouden… en het is niet de eerste! Hoe zit het nu met de performance? Theoretisch zou de tweede query het veruit moeten winnen van de eerste. Om dat vast te kunnen stellen, moeten we echter erst wat hard bewijsmateriaal verzamelen.
De eerste testen waren op onze testdata – niet meer dan zestig rijen. Ik was enigszins verrast door de resultaten: met TIMING ON in SQL*Plus, duur de eerste query 00:00:00.01, terwijl de Analytische query 00:00:00.03 nodig had. Hé, zouden die Analytische dingen niet veel sneller moeten zijn? Een beetje teleurgesteld ging ik eens grasduinen in de statistieken. Daar vond ik beter nieuws: de tweede, Analytische Query deed veel minder IO dan de oorspronkelijke, traditionele.
Misschien was de set van testdata gewoon te klein om een goed beeld te geven. En inderdaad, het bleek dat de testdata op geen enkele manier representatief waren voor de productiedatabase. Ik heb de ontwikkelomgeving wat uitgebreid voor een betere testset. Vervolgens bleek inderdaad dat de Analytische Query veruit superieur was ten opzichte van de oorspronkelijke query. In de laatste test heb ik 1,9 miljoen rijen in de tabel geladen. De Analytische Query was daar in pakweg negen seconden mee klaar. Hoe lang de oorspronkelijke query er

over deed weet ik niet: na drie uur heb ik de query afgebroken en ben ik naar huis gegaan. Ik wist genoeg.

De tweede, Analystische Query deed veel minder IO dan de oorspronkelijke, traditionele query

Gelooft u mij echter niet zomaar. Download het demonstratiescript van de AMIS Technology Blog en probeer het zelf eens uit: http://technology.amis.nl/blog/?p=506.

Conclusie

Het duurt even voor je gewend bent aan Analytische Functies. Het moeilijkst daarbij is het denken in groepen rijen (partities, windows) in plaats van individuele rijen. Als je eenmaal die stap hebt gemaakt wordt het snel gemakkelijker. Een belangrijk onderdeel van het schrijven van query's is af en toe wat afstand nemen en je realiseren dat er meerdere wegen naar Rome leiden. En soms ook kortere. Sta open voor alternatieve aanpakken. Analytische functies bieden een heel elegante manier om je query's te schrijven. Ze zijn vaak gemakkelijk te doorgronden en zijn daardoor goed te onderhouden. De goede performance , de compacte code en de lol om er mee te ontwikkelen rechtvaardigen dagelijks gebruik. Pas alleen op: niet alles is een spijker, ook al is de Analytische Hamer nog zo veelbelovend.

Alex Nuijten is Oracle Specialist bij AMIS Services in Nieuwegein, met meer dan zes jaar Oracle-ervaring. Hij is binnen AMIS actief als coördinator en veelvuldig presentator van het Kennis Centrum Oracle Server Development. Query performance en tuning zijn naast PL/SQL ontwikkeling specialismen van Alex. Daarnaast is hij een actieve weblogger (op de AMIS Technology Weblog op http://technology.amis.nl/blog). Dit artikel is gebaseerd op een paper en presentatie die Alex in juni hield tijdens de ODTUG 2006 Conferentie in Washington. Voor vragen of opmerkingen is Alex bereikbaar via alex.nuijten@amis.nl.

Advertentie

OraVision bouwt Oracle-oplossingen waarin documenten, transacties en bestaande systemen samenwerken.
OraVision staat bekend als *the mid-office company*.
OraVision bouwt vanuit haar geheel eigen visie: kwaliteit staat centraal.

Kwaliteit in kennis
OraVision beschikt over enorme ervaring in Oracle-, Java- en integratietechnologieën. Bij ons staat de techniek echter nooit op zichzelf. Juist bij mid-office en document-integratie toepassingen laten we de technologie tot volle bloei komen.

Kwaliteit in werk
Klanten geven OraVision al jaren het vertrouwen om geavanceerde ICT-toepassingen te realiseren die tegelijk gebruikersvriendelijk zijn. Onze mid-office oplossingen bevinden zich immers in het hart van elke bedrijfsvoering.

Kwaliteit in samenwerking
Bij OraVision staat niet alleen technische kwaliteit hoog in het vaandel, ook onze stijl is onderscheidend. Vanuit onze Limburgse basis investeren we nadrukkelijk in persoonlijke relaties en genieten van het goede leven.

Geïnteresseerd in de visie van OraVision op Oracle, Java, integratie en mid-office? Bezoek *www.oravision.com* en abonneer u gratis op de OraVisionair.