Pl/Sql blijft een interessante taal

Maar alleen als het echt moet...?

Het eerste dat je leert als Oracle-ontwikkelaar is Oracle SQL met vlak daarna Pl/Sql. Pl/Sql is sinds Oracle 7 de programmeertaal in de Oracle Database. In die tijd een van de belangrijkste toevoegingen in de Database. Sinds Oracle 8i kennen we ook Java in de database. Maar hoewel buiten de database Java een grote vlucht heeft genomen, in de database heeft het Pl/Sql niet kunnen verdringen. Toch heb ik altijd wel het idee gehad dat ontwikkelaars om mij heen Pl/Sql als een verplicht opleidingsnummer zagen, maar zo snel mogelijk doorstoomden naar Forms/Designer en later Java/ADF. Pl/Sql doe je alleen als het echt moet.

Wat er aan Pl/Sql geprogrammeerd wordt, is meestal ook nog op een Oracle 7.x manier: hele lappen code, bij voorkeur met zo weinig mogelijk commentaar. Eerlijk waar, ik zie het nog regelmatig. Toch investeert Oracle nog heel veel in Pl/Sql en dat maakt het nog steeds tot een hele interessante taal.
Sinds Oracle 8 kent Pl/Sql Object Types. Die implementatie was nog wel heel basaal, al maakte het de taal al heel wat krachtiger. Het belangrijkste gemis vond ik toen het ontbreken van fatsoenlijke constructors. Wel kon je al functies definiëren die collecties van Object Types retourneerden. En middels een aparte table-functie kon je zelfs al queries op zo'n collection uitvoeren. Dat is handig, want je kunt dan met zo'n functie allerlei informatie bij elkaar verzamelen met de exception handling van Pl/Sql en op het resultaat van de functie een SQL-query schrijven. Je hebt dan een view met als bron een functie. Echt briljant.
In Oracle 9i zijn de mogelijkheden een heel stuk uitgebreid. Met als belangrijkste toevoegingen Constructors en Pipe-lined functions. Het gaat nog wat te ver om te zeggen dat daarmee Pl/Sql een object geörienteerde taal is. Die discussie laat ik hier liggen.
Het is en blijft een 3-GL met Object-toevoegingen. Dus voor de OO-purist: je hebt wat mij betreft bij voorbaat al gelijk. Maar object-typen dat maakt het leven van de Pl/Sql programmeur wel een heel stuk leuker.

Een object met een constructor

Een object declaratie gaat als volgt::

```
create or replace type car_car_t as object
(
  -- Attributes
    license varchar2(10)
  , category number(1)
  , year number(4)
  , brand varchar2(20)
  , model varchar2(30)
  , city varchar2(30)
  , country varchar2(30)
  -- Member functions and procedures
  , constructor function car_car_t(p_license in varchar2)
    return self as result
  , member function dayly_rate(p_date date)
    return number
  , member procedure print
)
/
create or replace type body car_car_t is
  -- Member procedures and functions
  constructor function car_car_t(p_license in varchar2)
  return self as result
  is
  begin
    select license
    ,     category
    ,     year
    ,     brand
    ,     model
    ,     city
    ,     country
    into self.license
    ,     self.category
    ,     self.year
    ,     self.brand
    ,     self.model
    ,     self.city
    ,     self.country
    from cars
    where license = p_license;
    return;
  end;
  member function dayly_rate(p_date date)
    return number
  is
    l_rate number;
    cursor c_cae( b_license in varchar2
                , b_date in date)
    is select cae.dailyrate
    from carsavailable cae
```

```
    where b_date between cae.date_from and nvl(cae.date_to, b_date)
    and   cae.car_license = b_license
    order by cae.date_from;
    r_cae c_cae%rowtype;
  begin
    open c_cae( b_license => self.license
              , b_date => p_date);
    fetch c_cae into r_cae;
    close c_cae;
    l_rate := r_cae.dailyrate;
    return l_rate;
  end;
  member procedure print
  is
    l_daily_rate number;
  begin
    dbms_output.put_line( 'License : '||self.license);
    dbms_output.put_line( 'Category : '||self.category);
    dbms_output.put_line( 'Year : '||self.year);
    dbms_output.put_line( 'Brand : '||self.brand);
    dbms_output.put_line( 'Model : '||self.model);
    dbms_output.put_line( 'City : '||self.city);
    dbms_output.put_line( 'Country : '||self.country);
    l_daily_rate := daily_rate(p_date => sysdate);
    if l_daily_rate is not null
    then
      dbms_output.put_line('Daily Rate: '||l_daily_rate);
    else
      dbms_output.put_line('No cars available');
    end if;
  end;
end;
/
```

Ik ga hier geen college objectoriëntatie geven, maar eigenlijk is al direct te zien dat een objecttype een soort losstaand record-type is, met naast attributen ook uitvoerbare toevoegingen: methods. Methods zijn functies en procedures die op de eigen data werken.
Vanaf Oracle 9i is de belangrijkste toevoeging de mogelijkheid om eigen constructors te definiëren. Dat gaat dus als boven is uitgewerkt. Een constructor is feitelijk een speciale method die begint met het keyword constructor en is altijd een functie die het eigen-object type als resultaat terug geeft. Ook heeft de constructor altijd dezelfde naam als het object type. Impliciet is er altijd één constructor die alle attributen als parameter van het object-type meekrijgt. Dit was al zo in Oracle 8, maar krijg je in Oracle 9i/10g nog steeds gratis. Die hoeft dus niet gedeclareerd te worden, tenzij het gedrag bijgestuurd moet worden. Maar naast de impliciete constructor zijn er nu ook meerdere zelf te definiëren. Hierdoor kan een object zichzelf instantiëren op basis van bijvoorbeeld een primary key waarde. Of door het uitlezen van een bestand. Of parameter-loos zodat gewoon een leeg dummy-object is te instantiëren.

Meestal voeg ik ook een print methode of een 'to_xml'-methode toe. Hierin neem ik de attributen, eventuele methods, voor zover het resultaat afdrukbaar is, en aanroepen naar de print of 'to_xml'-methoden van child-objects op. Met child-objects bedoel ik hier objecten die als attribuut zijn opgenomen. Wanneer dat collections van objecten zijn (waarover hierna meer) dan is er uiteraard een loopje nodig. Hierdoor is een object meteen makkelijk te testen, door na de instantiatie gewoon de print method van het parent object aan te roepen:

```
declare
  -- Local variables here
  l_car car_car_t;
begin
  -- Test statements here
  l_car := car_car_t(:license);
  l_car.print;
end;
```

Collections

Een object komt vaak niet alleen. Dat geldt ook voor Object-instanties. We werken nog steeds vanuit een database en we hebben dan ook vaak meer dan één instantie van een bepaalde entiteit. Een verzameling object-instanties noemen we een collection. En die wordt als volgt:gedefinieerd:

```
create or replace type car_cars_t as table of car_car_t;
```

Het is dus eigenlijk een tabel van objecten van een bepaald type. Merk overigens op dat er nu een referentie is naar, of anders gezegd een afhankelijkheid met het betreffende object-type. Dit betekent dat de object-specificatie van in dit geval 'car_car_t' niet meer gewijzigd kan worden, zonder alle referenties er naar toe te droppen. De 'body', of wel de programmacode, kan wel opnieuw gecompileerd worden. Dit is wel belangrijk, want de specificatie legt de definitie van het object vast (de class) en daarvan zijn de andere objecten te zeer afhankelijk. Met name als het gaat om tabel definities (in de database) waarin een objecttype als kolomdatatype wordt opgenomen. Immers, een fysieke tabel kan niet invalid worden. Wat zou er dan met de data moeten gebeuren? Die zou dan ook ineens 'onbepaald' worden. Collections zijn voor de rest te beschouwen als een Pl/Sql table, vergelijkbaar met een 'index by binary_integer'-table. Met dit verschil dat een collection zelf ook een object is, waarin objecten zitten. Dit betekent dat om een Collection te gebruiken deze wel geïnstantieerd moet worden.
Je kunt de collection expliciet instantiëren, als volgt:

```
declare
  l_cars car_cars_t;
begin
  l_cars := car_cars_t();
  for r_car in (select license from cars)
  loop
    l_cars.extend;
    l_cars(l_cars.count) := car_car_t(r_car.license);
  end loop;
  if l_cars.count > 0
    then
    for l_idx in l_cars.first..l_cars.last
    loop
      dbms_output.put_line('Car '||l_idx||':');
```

```
      l_cars(l_idx).print;
    end loop;
  end if;
end;
```

Wat hier gebeurt, is dat in de eerste regel de collection wordt geïnstantieerd. Vervolgens wordt in een loop op basis van een select op de primary-key van de tabel, de collection steeds uitgebreid (extend). En elke rij krijgt dan een instantiatie van het car_car_t object. In de tweede loop zie je hoe eenvoudig je de collection kunt verwerken. In dit geval worden de attributen van elk rijobject even geprint.

Je kunt bovenstaande ook impliciet doen, bijvoorbeeld door middel van een query:

```
select cast(multiset(
  select license
  , category
  , year
  , brand
  , model
  , city
  , country
  from cars
  ) as car_cars_t)
from dual
```

Wat hier feitelijk gebeurt, is dat de result-set van de select op de tabel cars wordt geherdefinieerd als een Collection. De Multiset-functie geeft aan dat wat er terug wordt gegeven een gegevensset is van 0 of meerdere rijen. De Cast-functie wordt gebruikt om aan te geven als welk datatype/objecttype de multiset moet worden beschouwd. Er wordt dus een collection-laagje over de resultset heen gelegd. Wat is het verschil tussen deze query en de for-cursor-loop? Door het collection-sausje overheen is een willekeurige resultset te behandelen als ware het een Pl/Sql-tabel in geheugen. Je kunt het in een 'execute immediate' constructie uitvoeren. En dan is eigenlijk geen 'ref-cursor' constructies meer nodig. Verder is dit een valide SQL-constructie, die als sub-queries in andere SQL statements kan worden opgenomen.

Object-Functions en Views

Het opbouwen van een collection kan natuurlijk ook in een functie worden verwerkt:

```
create or replace function get_cars return car_cars_t is
  l_cars car_cars_t;
begin
  select cast(multiset (select license
                        ,category
                        ,year
                        ,brand
                        ,model
                        ,city
                        ,country
                from cars) as car_cars_t)
  into l_cars
  from dual;
  return(l_cars);
exception
  when no_data_found then
    l_cars := car_cars_t();
    return l_cars;
end get_cars;
```

Ook hier in de functie wordt weer een collection-sausje over een resultset gelegd. Maar andersom kan ook: een collection kan ook met SQL worden bevraagd:

```
select car.license
      ,car.category
      ,car.year
      ,car.brand
      ,car.model
      ,car.city
      ,car.country
      ,car.daily_rate(sysdate) daily_rate
from table(get_cars) car
```

Het is de 'table'-functie die het hem doet. Die zegt eigenlijk: beschouw de collection als een resultset. In een query zijn ook de methods beschikbaar, mits het een functie betreft. Overigens zijn in dit voorbeeld de attributen en het functie-resultaat van enkelvoudige datatypen, maar dat kunnen ook objecten of collections zijn. Ook die zijn te benaderen in de query. Van objectattributen zijn met de '.'-notatie ook weer de onderliggende attributen op te vragen. Met andere woorden: hiërarchisch dieper liggende attributen zijn zo als kolom-waarde te benoemen.

Hier gebruiken we een functie als basis voor de query. In dat geval is het ook mogelijk om er een view overheen te leggen. Zolang de functie en de object-typen die worden terug gegeven maar 'zichtbaar' zijn voor het schema dat eigenaar is van de view en/of gebruik maakt van de view.

Maar het gaat nog een stapje verder. Niet alleen het resultaat van een functie kan worden gebruikt als basis van een functie. Ook een locale variabele of package-variabele kan als bron voor een query worden gebruikt:

```
declare
  l_cars car_cars_t;
  l_rate number;
begin
  l_cars := get_cars;
  select car.daily_rate(sysdate - 365) daily_rate
  into l_rate
  from table(l_cars) car
  where license = 'JR-JP-09';
  dbms_output.put_line(l_rate);
end;
```

Dat is toch heel wat makkelijker dan door een pl/sql-tabel door lopen op zoek naar net die ene rij!
Let wel: dit veroorzaakt feitelijk wel een full-table scan. Maar

omdat die full-table scan volledig in het geheugen plaats vindt is die toch heel snel. En laten we eerlijk zijn: wie heeft er al eens een pl/sql-table opgebouwd van een gigabyte? Wanneer je een goed gevulde database hebt, is dat met bovenstaande voorbeelden natuurlijk wel zo gedaan. Dus een beetje performance-bewust programmeren is wel verstandig.

Pipelining

In de vorige paragraaf is performance al aan gehaald. Met de collection-function-methode van hierboven was in Oracle 8i eigenlijk al 'External Tables ' uit te programmeren. Het was bijvoorbeeld mogelijk om in een Pl/Sql-function met UTL_File een bestand uit te lezen, verwerken tot een collection en deze terug te geven. Door er vervolgens met de table-functie een view overheen te definiëren is het mogelijk queries te doen op een bestand!
Een belangrijk nadeel van deze methode is dat de functie als logisch/functioneel geheel wordt uitgevoerd. Het hele bestand wordt ingelezen waarmee de collection wordt opgebouwd en als geheel aan het aanroepende proces teruggegeven. Dat betekent dat als je een select op die function doet, de hele functie eerst wordt uitgevoerd voordat je pas resultaat terug krijgt. Dat is natuurlijk vooral vervelend als de nabewerking op het resultaat van die functie ook nog eens tijdrovend is. Daarvoor is in Oracle 9i 'pipelining' geïntroduceerd. De verschillende stadia van zo'n proces die voorheen in per stadium in het geheel sequentieel werden uitgevoerd, kunnen dan parallel worden uitgevoerd.
Een pipelined function is functioneel identiek aan de collection-returning-function. Het belangrijkste verschil is dat in de declaratie wordt aangegeven dat het om een pipelined function gaat, maar vooral dat tussenresultaten worden 'ge-piped' (oud-Nederlands woord) of te wel teruggegeven worden zodra ze beschikbaar zijn.

Het ziet er als volgt uit:

```
create or replace function get_cars_piped(p_where in varchar2 default
null)
  return car_cars_t
  pipelined is
  l_car car_car_t;
  type t_car is record(
     license cars.license%type);
  type t_cars_cursor is ref cursor;
  c_car   t_cars_cursor;
  r_car   t_car;
  l_query varchar2(32767);
begin
  l_query := 'Select license from cars ' || p_where;
  open c_car for l_query;
  fetch c_car
    into r_car;
  while c_car%found
  loop
    l_car := car_car_t(p_license => r_car.license);
    pipe row(l_car);
    fetch c_car
      into r_car;
  end loop;
  close c_car;
  return;
end get_cars_piped;
```

Merk in de specificatie van de functie het keyword 'pipelined' op. In de loop wordt nu elk afzonderlijk object middels 'pipe row' naar buiten wordt gebracht.
Daarnaast geheel gratis en voor niks in de functie nog even een voorbeeld van het gebruik van een ref-cursor. Hierdoor is het mogelijk om een flexibele cursor op te bouwen waarvan de select is bij te stellen.
De functie is dan als volgt aan te roepen:

```
select *
from table(get_cars_piped('where license != ''TR-XP-81'''))
```

Pas met dit soort ref-cursor constructies overigens wel op vanwege het niet gebruiken van bind-variabelen.
Het is me gebleken dat het niet mogelijk is om deze functie in een pl/sql-block direct aan te
roepen. Eigenlijk is dat ook wel logisch. Wat hierboven gebeurt is dat de sql-engine de pl/sql-functie aanroept en elke rij tussentijds terug krijgt en meteen kan verwerken. Hierdoor is het mogelijk om het uitvoeren van de functie en het verwerken van het resultaat van de functie parallel te verwerken. Pl/Sql in zichzelf ondersteunt geen threads of pipe-lines. Pl/Sql verwacht bij een functie-aanroep het resultaat als geheel terug en kan pas verder met verwerken als de gehele aangeroepen functie klaar is. Uiteraard is de functie wel te gebruiken in een Cursor-select en het resultaat op die manier in een cursor-loop te verwerken.

Object Views

Nu hebben we gezien hoe een Collection sausje over een result-set is te leggen en hoe een Collection weer met Select-statements is uit te vragen. Een andere belangrijke toevoeging sinds Oracle 9i zijn ook de zogenaamde object views. Dit zijn views die object-instanties terug geven. Dit in tegenstelling tot gewone views die rijen met kolommen terug geven. Een object view ziet er als volgt uit:

```
create or replace view car_ov_cars
of car_car_t
with object oid (license)
as
select license
, category
, year
, brand
, model
, city
, country
from cars
```

Typisch aan een object view is dat je aangeeft wat het object-type is waar de view op is gebaseerd en wat de object-identifier OID is. Dat is feitelijk het attribuut of de serie attributen die gelden als de primary-key van het object.
Je kunt deze view bevragen als een normale view, maar de kracht zit hem in het kunnen ophalen van een rij als een object. Dit gaat met de functie 'value':

```
declare
  l_car car_car_t;
begin
  select value(t) into l_car from car_ov_cars t where license = '79-JF-
VP';
  l_car.print;
end;
```

Je hebt nu zonder moeilijk te doen een object-instantie uit de view.

References

Wanneer je een uitgebreid objectmodel hebt, dan loop je er tegen aan dat een object als attribuut een of meerdere collections heeft. Die collections kunnen weer meerdere instanties van een ander objecttype bevatten. Dit kan nogal geheugenintensief worden. Daarnaast komt het wel eens voor dat je circulaire-referenties wilt implementeren. Bijvoorbeeld een afdeling heeft een manager en dat is een employee die een of meerdere andere empoyees onder zich heeft. Het zou kunnen dat je dat wilt modelleren als een employee met een attribuut van een collectiontype op het employeetype. Het is dan handig als je een losse koppeling kunt hebben tussen objecten.
Daarvoor zijn References in het leven geroepen. In feite is een reference niets anders dan een pointer naar een object-instantie. En die neemt natuurlijk minder geheugen in beslag dan het object zelf. Een reference verwijst naar een object-instantie in een object-tabel of naar een object-view. En dan komt die object-identifier in de vorige paragraaf van pas.
Een collection van references ziet er als volgt uit:

Een reference is niets anders dan een pointer naar een objectinstantie

```
create or replace type car_cars_ref_t as table of ref car_car_t;
```

Het verschil met de eerder genoemde collection declaratie is het keyword 'ref'. Het gaat daardoor om een collectie van verwijzingen naar objecten.
Deze collectie is te vullen met de 'make_ref' functie.

```
declare
  l_cars car_cars_ref_t;
  l_car  car_car_t;
begin
  -- Bouw collectie met references op
  select cast(multiset (select make_ref(car_ov_cars, cae.car_license)
              from carsavailable cae) as car_cars_ref_t)
  into l_cars
  from dual;
  -- Verwerk collection
  if l_cars.count > 0
  then
    for l_idx in l_cars.first .. l_cars.last
    loop
      dbms_output.put_line('Car ' || l_idx || ': ');
      -- Haal object-value op basis van reference op
      select deref(l_cars(l_idx)) into l_car from dual;
      -- Druk object af
      l_car.print;
    end loop;
  end if;
end;
```

Je ziet dat de make_ref functie een verwijzing naar een object-view nodig heeft en een opgave van de betreffende objectidentifier. De onderliggende query levert dan de verwijzing naar objecten die behandeld moeten worden. Die query kan dus anders zijn dan de query van de objectview.
Waar het op neer komt is dat je eerst bepaalt welke objecten in aanmerking komen. Van die objecten bepaal je een referentie/pointer ten opzichte van een objectview. En vervolgens kun je in een later stadium aan de hand van die referentie het uiteindelijke object ophalen. Dat laatste gaat met behulp van de 'deref'-functie. Die 'deref'-functie verwacht een referentie en geeft het bijbehorende object terug. De 'deref' is er overigens alleen in de SQLfunctie smaak, je kunt hem niet direct in Pl/Sql gebruiken.

Onder water wordt de 'select deref()'-query vertaald naar een select op de objectview. Het is dan ook van belang om je objectmodel en je objectview zodanig te ontwerpen dat de uiteindelijke query op die objectview voldoende geïndexeerd is. De ervaring leert dat het vrij lastig te achterhalen is waarom de optimizer wel of niet een index gebruikt bij derefs. Daarin is de deref een lastige extra abstractie laag.

Het keyword 'ref' uit de ref-collection declaratie, is ook in de declaratie van objectattributen te gebruiken. Wanneer een object als attribuut in een ander object wordt opgenomen, bijvoorbeeld een object car in het object garage, dan is met het keyword ref aan te geven dat niet het object zelf maar een referentie op wordt bedoeld.

```
create or replace type car_garage_t as object
(
```

```
  car ref car_car_t
)
```

Daarbij bestaat er ook een ref functie die referenties naar afzonderlijke objecten creëert:

```
select ref(car) reference
, license
from car_ov_cars car
```

Deze functie is dus eigenlijk een tegenhanger van de value-functie.

Het verschil tussen de functies ref en make_ref is eigenlijk dat 'ref' als parameter het object krijgt waarvoor een referentie bepaald moet worden. Make_ref is daarentegen gebaseerd op een objectview of objecttable en bepaalt de referentie op basis van de primary-key of objectid in de objectview of -tabel. De ref-functie wordt gebruikt als een referentie naar een object die direct het resultaat is van een query op de object-view nodig is. Maar als op basis van een query de primary-keys van te behandelen objecten worden bepaald, dan is make_ref handig. Want dan worden de primary-keys van de te behandelen objecten apart aangeleverd en make_ref bepaalt dan op basis van de objectview en de primary-key waarden de referenties.

MAP en Order methods

Het kan voorkomen dat objecten moeten worden geordend. Welke is nu groter of kleiner en hoe sorteer ik objecten? Dat is natuurlijk van belang bij het vergelijken van objecten maar ook bij het bevragen van objectviews en objecttabellen.
Voor het vergelijken van objecten kun is een map method aan te maken.

```
map member function car_size
return number
is
begin
  return 1000; -- of een berekening van de inhoud of waarde van de auto
end;
```

Hierin kan een berekening op basis van attributen van het object worden gemaakt. Het resultaat moet van een scalaire datatype zijn (number, date, varchar2) en 'maatgevend' voor het object zijn ten opzichte van andere objecten van hetzelfde object-type. De map-method wordt dan door Oracle gebruikt om vergelijkingen te doen als l_car1 > l_car2, en vergelijkingen die worden geïmpliceerd in select-clausules als: DISTINCT, GROUP BY, en ORDER BY.
Ook kan gebruik worden gemaakt van een Order methode:

```
order member function car_order(p_car car_car_t) return number is
  l_order number := 0;
  c_kleiner constant number := -1;
  c_groter  constant number := 1;
begin
  if licence < p_car.license
  then
    l_order := c_kleiner;
  elsif licence > p_car.license
  then
    l_order := c_groter;
  end if;
  return l_order;
end;
```

Het verschil met de map-method is dat de map-method een waarde terug geeft die uitsluitend iets zegt over het eigen object. De impliciete parameter is alleen het 'self'-object. Oracle bepaalt twee te vergelijken objecten de het resultaat van de map-methods en vergelijkt die twee resultaten.
Bij de order-method geeft Oracle het ene object als parameter naar de order-method van het andere object. De order method heeft daarom altijd een extra parameter naast de impliciete self-parameter. In de functie moet dan een vergelijking tussen de twee objecten worden gecodeerd. En die kan natuurlijk een stuk complexer zijn dan hierboven. Vervolgens geef wordt een negatieve waarde teruggegeve als het self-object kleiner is als het meegegeven object en een positieve waarde als het self-object groter blijkt. Een waarde van 0 geeft een gelijkheid van de twee objecten weer.
De Order-method wordt bij l_car1 > l_car2 vergelijkingen gebruikt en moet altijd een numerieke return datatype hebben. Een object mag maar 1 map-method en 1 order-method hebben.

Conclusie

Het werken met Object Types komt initieel misschien erg omslachtig over. De meeste functionaliteit die je bouwt krijg je ook wel op de Oracle 7 manier voor elkaar. Terwijl je voor vergelijkbare functionaliteit wel eerst Object Types moet declareren.

Object Types zijn echter vaak gewoon te genereren. Bijvoorbeeld met behulp van XSLT op basis van tabel definities in XML. En bepaalde oplossingen worden ineens een stuk krachtiger als ze met behulp van object-typen worden uitgewerkt. Door objecttypes wordt Pl/Sql een stuk krachtiger en zijn er weer meer handvatten om sommige netelige performance-problemen op te lossen. Of stukjes functionaliteit die op de Oracle 7 manier toch echt niet voor elkaar zijn te krijgen.

Het meeste hier werkt al in Oracle 9i. In Oracle 10g en 11g zal het door de performance optimalisaties van de Pl/Sql-engine een stuk sneller werken.

Martien van den Akker is Technisch Architect bij Darwin IT-Professionals.