Object geörienteerd Pl/Sql

Pl/Sql, da's toch niks man! Java is veel krachtiger, daar kun je alles mee! Een beetje overdreven denk je misschien, maar soms bekruipt me het gevoel dat veel Oracle-consultants binnen en vooral buiten Oracle zo denken. Pl/Sql gebruiken ze alleen als het echt moet. En als ze al wat aan Pl/Sql doen, dan voornamelijk op een Oracle 7.x manier: hele lappen code (liefst met zo weinig mogelijk commentaar). Misschien dat na dit artikel je vingers weer wat beginnen te kriebelen voor Pl/Sql.

Sinds Oracle 8 kent Pl/Sql Object Typen. Die implementatie was nog wel heel basaal, al maakte het de taal al aardig wat krachtiger. Het belangrijkste gemis vond ik toen het ontbreken van fatsoenlijke constructors.
In Oracle 9i zijn de mogelijkheden een heel stuk uitgebreid. 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 bijvoorbaat al gelijk. Maar object-typen dat maakt het leven van de Pl/Sql programmeur wel een stuk leuker.

Met dit artikel hoop ik je een stuk op weg te helpen. We beginnen gewoon maar even basaal.

Een object met een constructor

Een object maak je 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 object orientatie geven, maar je ziet eigenlijk al direct dat een object-

type een soort los-staand record-type is maar dan naast attributen ook uitvoerbare toevoegingen: methods. Functies en procedures die op de eigen data werken.

Zoals gezegd is een handige toevoeging de mogelijkheid om eigen constructors te definieëren. Dat gaat dus als boven is uitgewerkt. Een constructor 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 typen. Impliciet is er altijd 1 constructor die als parameter alle attributen van het object-type meekrijgt. Dit was al zo in Oracle 8, maar krijg je in Oracle 9i/10g nog steeds gratis. Maar daarnaast zijn er nu dus ook meerdere zelf te definieren. Hierdoor kun je een object zichzelf laten instantieren op basis van bijvoorbeeld een primary key waarde. Of door het uitlezen van een bestand. Of parameter-loos zodat je gewoon een leeg dummy-object kunt instantieren.

Meestal voeg ik ook een print methode toe. Hierin neem ik de attributen, eventuele methods voor zover het resultaat afdrukbaar en aanroepen naar de printmethoden van child-objects, objecten die als attribuut zijn opgenomen. Wanneer het een collection van objecten zijn (waarover hierna meer) dan is er uiteraard een loopje nodig. Hierdoor is een object meteen makkelijk te testen:

```
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 niet vaak alleen. Dat geldt vaak ook voor Object-instanties. We zijn tenslotte een database-club en we hebben dan ook vaak meer dan één instantie in zitten van een bepaalde entiteit. Een verzameling object-instanties noemen we een collection. En die definieer je als volgt:

```
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 programma-code, 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 je een object-type kunt opnemen. Immers, een fysieke tabel kan niet invalid worden. Wat zou er dan met de data moeten gebeuren? Die zou dan ook in eens 'onbepaald' worden.

Collections kun je voor de rest beschouwen als een Pl/Sql table, vergelijkbaar met een "index by binary_integer"-table. Met dit verschil dat het wel een object is, waarin objecten zitten. Dit betekent dat om een Collection te gebruiken deze wel geïnstantieerd moet worden. Dit kun je bijvoorbeeld impliciet doen 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 gegevens-set 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 laag je over de resultset heen gelegd. Ik heb het nog nooit kunnen testen, maar ik ben wel benieuwd naar de performance effecten. Wat zou het verschil zijn tussen deze query en bijvoorbeeld een for-cursor-loop? Nu er een collection-sausje overheen ligt kun je die resultset immers behandelen als ware het een Pl/Sql-tabel in geheugen.

Je kunt de collection ook expliciet instantieren, 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 je hier dan ziet gebeuren 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 elke rij-object even geprint.

Object-Functions en Views

Het opbouwen van een collection kun je natuurlijk ook in een functie stoppen:

```
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;
```

De grap is dan dat je het resultaat van die functie weer kunt gebruiken als bron voor een query. Je ziet hier in de functie weer dat je een collection-sausje over een resultset kunt leggen. Maar anders om kan ook: je kunt een collection ook be-queriën:

```
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 result set. Je ziet dat in een query dus ook de methods bruikbaar zijn. Uiteraard alleen als 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 kun je benaderen in de query. Van object-attributen kun je in de query daarbij met de '.'-notatie ook weer de onderliggende attributen opvragen. Met andere woorden: hiërarchisch dieper liggende attributen kun je zo als kolom-waarde terug geven.

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 de user/het schema die/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 = '79-JF-VP';
  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! Nu denk je misschien: maar dat levert toch een full-table scan? Ja inderdaad. Maar die fulltable scan vindt volledig in het geheugen plaats en is daardoor heel snel. En laten we eerlijk zijn: wie heeft er al eens een pl/sql-table opgebouwd van een gigabyte? Al 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 kon je in Oracle 8i al 'External Tables' uit programmeren. Je kon bijvoorbeeld in een Pl/Sql-function met UTL_File een bestand uitlezen en verwerken tot een collection en deze terug geven. Vervolgens een view er overheen definiëren met de table-functie en je kunt een query doen op een bestand!
Een belangrijk nadeel van deze methode is dat de functie als logisch/functioneel geheel wordt uitgevoerd. Dus het hele bestand wordt ingelezen, de hele collection wordt opgebouwd en als geheel naar de aanroep 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. Vandaar dat er in Oracle 9i pipelining is geïntroduceerd.

Een pipelined function is functioneel identiek aan de collection-returning-function als ik hierboven heb beschreven. Het belangrijkste verschil is dat is aangegeven dat het om een pipelined function gaat (duh!), maar vooral dat tussen-resultaaten worden ge-piped (sorry, heb ik ook niet bedacht) 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;
```

Je ziet inderdaad in de specificatie van de functie het keyword 'pipelined', en vervolgens dat in de loop 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 != ''15-DF-HJ'''))
```

Het is me gebleken dat het niet mogelijk is om deze functie in een pl/sql-block direct aan te roepen. Als je er over nadenkt 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 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.

Object Views

Je hebt nu kunnen zien hoe je een Collection sausje over een resultset kunt leggen en hoe je een Collection weer met Select-statements kunt uitvragen. Een belangrijke toevoeging sinds Oracle 9i zijn echter 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 object model 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 object type bevatten. Dit kan nogal geheugen intensief 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 collection type op het employee-type. 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. Je kunt die leggen 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:

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

Deze 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 object identifier. De onderliggende query levert dan de verwijzing naar objecten die behandeld moeten worden. Die query kan dus anders zijn dan de query van de object-view. 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 object-view. 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 SQL-functie smaak, je kunt hem niet direct in Pl/Sql gebruiken. Onder water wordt de 'select deref()'-query vertaald naar een select op de object-view. Het is dan ook van belang om je object model en je object view zodanig te ontwerpen dat de uiteindelijke query op die object view 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 keywordje ref dat je terug ziet in de ref-collection declaratie, kun je in de declaratie van object attributen ook gebruiken. Wanneer je een object als attribuut in een ander object wilt opnemen, bijvoorbeeld een object car in het object garage, dan kun je door het keyword ref aangeven dat je niet het object zelf maar een referentie op wilt nemen.

```
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 waar voor een referentie bepaald moet worden. Make_ref is daarentegen gebaseerd op een object-view of object-table en bepaalt de referentie op basis van de primary-key of object-id in de object-view of -tabel. De ref-functie gebruik je als je een referentie wilt maken naar een object die direct het resultaat is van een query op de object-view. Maar als je nu op basis van een query op andere tabellen en/of views de primary keys wilt bepalen van objecten die je wilt behandelen dan is make_ref handig. Want dan lever je de primary-keys van de te behandelen objecten apart aan en make_ref bepaalt dan op basis van de object-view en de primary-key waarden de referenties.

MAP en Order methods

Het kan voorkomen dat je objecten wilt ordenen. 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 querien op object-views en object-tabellen.
Voor het vergelijken van objecten kun je een map method aanmaken.

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

Hier in kun je een berekening maken op basis van attributen van het object. 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 geimpliceerd in select-clausules als: DISTINCT, GROUP BY, en ORDER BY.

Ook kun je gebruik maken 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 codeer je dan zelf een vergelijking tussen de twee objecten. En die kan natuurlijk een stuk complexer zijn dan hierboven. Vervolgens geef je een negatieve waarde terug 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

Misschien duizelt het je allemaal. Als je helemaal tot hier bent gekomen met lezen dan ben ik onder de indruk. Het is misschien saaie kost. En als je er mee aan de slag gaat dan komt het initieel allemaal misschien erg omslachtig over. De meeste functionaliteit die je bouwt krijg je ook wel op de Oracle 7 manier voor elkaar. Maar bepaalde oplossingen worden in eens een stuk krachtiger als je dat met behulp van object-typen uitwerkt. Ik maak er al een tijd dankbaar gebruik van. Maar ik ben dan ook iemand die eenzelfde probleem een volgende keer graag op

een andere manier oplost.

Door object-typen wordt Pl/Sql een stuk krachtiger en heb je weer meer handvatten om sommige netelige performance-problemen op te lossen. Of stukjes functionaliteit die je op de Oracle 7 manier toch echt niet voor elkaar krijgt.

Het meeste hier werkt al onder Oracle 9i. Onder Oracle 10g zal het door de performance optimalisaties van de Pl/Sql-engine een stuk sneller lopen.

Eigenlijk was dit niet een verhaal over Object Geörienteerd Pl/Sql. Ik heb het bijvoorbeeld niet gehad over super en sub-typen. Lees dat maar eens na in Hoofdstuk 12 van de Pl/Sql User's Guide en Reference van Oracle 10g. Maar ik wilde je aan de slag helpen met Object Typen, laten zien wat je er mee kunt en hoe krachtig Pl/Sql hierdoor is.

Veel plezier met Pl/Sql, want zo is Pl/Sql echt leuk! En mocht je met bovenstaande voorbeelden aan de slag gaan, mail dan even naar de redactie of post een topic op het Snapshot-forum. Dan zorgen we er voor dat de scripts voor het aanmaken van het datamodel dat ik gebruikt heb op de Snapshot-site komen te staan.

Martien van den Akker
Development Specialist