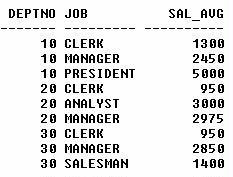
« [New Book Shipment is in – recent titles on Web, Java and Oracle](http://technology.amis.nl/2006/05/23/new-book-shipment-is-in-recent-titles-on-web-java-and-oracle/)

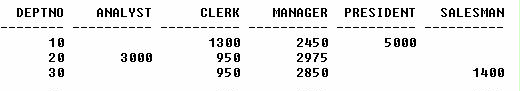
[The long awaited Oracle Warehouse Builder 10g R2 is now available](http://technology.amis.nl/2006/05/24/the-long-awaited-oracle-warehouse-builder-10g-r2-is-now-available/) »

**Dynamic SQL Pivoting – Stealing Anton’s Thunder**

I have a colleague who has a stroke of genius. He can create things in SQL and PL/SQL that I find very hard to fathom, let alone could have come up with myself. His name is Anton and his recent focus has been Pivoting in SQL. The process of turning a Query Result upside down to present it differently. Turning the set of values in one of the columns into the column headings for the entire query.

Like taking the following query result:



and turning it into this structure:  
  
Anton recently wrote a brief, somewhat cryptic article on Pivoting SQL Queries on this weblog; [see:http://technology.amis.nl/blog/?p=1197](http://technology.amis.nl/blog/?p=1197) . In this article, Anton introduced an approach to pivoting that does not require us to hard-code all the values that we want to see appear in our column-headings (such as the values for JOB in the example above). However, the OBJECT TYPE that he uses in this article contains the hardcode query itself. It was already quite good, but he went where man did not go before and created an implementation that is completely dynamic. We simply pass in the query we want to have pivoted, and the function Anton created obliges.

In this article I will show a few examples of what we can do with this amazing gem as well as make the code itself available for download. However, and this is the last time I will blow his trumpet, it is not my code. I am good at writing long-winded articles, while Anton forges some source.

The first query result in the article was created using a pretty straightforward query:

select deptno

, job

, avg(sal) sal\_avg

from emp

group

by deptno

, job

/

The second, pivoted result was created using the same query, however this time passing it as a string to the pivot function I will introduce in more detail later on:

select \*

from table( pivot(' select deptno

, job

, avg(sal) sal\_avg

from emp

group

by deptno

, job

')

)

We have seen the stunning result of this call already. To prove it is really working I can add two rows to our EMP table:

insert into emp

( empno, ename, job, deptno, sal)

values

( 1111, 'ANTON', 'WIZARD', 10, 1991)

/

insert into emp

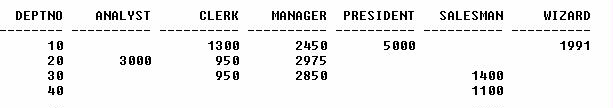
( empno, ename, job, deptno, sal)

values

( 1112, 'TOBIAS', 'SALESMAN', 40, 1100)

/

Now if we rerun the last query, the result will be:



So we have picked up the new records and extracted the new JOB value (WIZARD) that was added as a new column in our query result. Note: sometimes new column values are not picked up so readily. In those cases, we have to cheat a little by enticing the CBO into re-evaluating our query (or so it seems, we have not quite figured it out what it is). If we add a space somewhere in the query-string or make another change, the new column values are picked up.

**Generating a Distance Matrix**

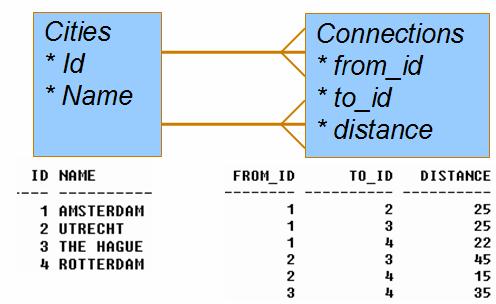
Your average roadmap will have a grid-like figure, a distance matrix, that lists cities and the distances between them. Using Anton’s Pivot function, it turns out to be pretty simple to create such a Distance Matrix. Let’s review an hypothetical example (the distances are by no means correct, but that’s not the point) for the main cities in The Netherlands:



The western part of The Netherlands is called De Randstad. This is where almost half the population lives and most economical activity is concentrated. The map for this piece (blue on the map because it is below sea level) of our country:



We now want to build ourselves a little distance matrix for the four main cities in De Randstad. We create a small database structure for the city-and-connection details:



To have easier access to our data, we create a utility view:

create or replace view city\_connections

as

select from\_city.name from\_city

, to\_city.name to\_city

, conn.distance distance

from cities from\_city

, connections conn

, cities to\_city

where conn.from\_id = from\_city.id

and conn.to\_id = to\_city.id

union

select from\_city.name from\_city

, to\_city.name to\_city

, conn.distance distance

from cities from\_city

, connections conn

, cities to\_city

where conn.to\_id = from\_city.id

and conn.from\_id = to\_city.id

/

A simple query against this view renders these results:

FROM\_CITY TO\_CITY DISTANCE

------------ ------------------

AMSTERDAM ROTTERDAM 22

AMSTERDAM THE HAGUE 25

AMSTERDAM UTRECHT 25

ROTTERDAM AMSTERDAM 22

ROTTERDAM THE HAGUE 35

ROTTERDAM UTRECHT 15

THE HAGUE AMSTERDAM 25

THE HAGUE ROTTERDAM 35

THE HAGUE UTRECHT 45

UTRECHT AMSTERDAM 25

UTRECHT ROTTERDAM 15

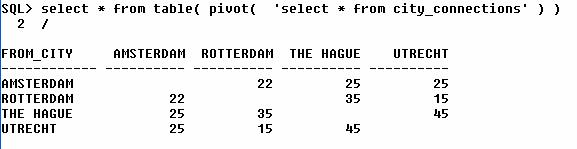
UTRECHT THE HAGUE 45

The pivot-query we need is the following:

select \* from table( pivot( 'select \* from city\_connections' ) )

/

and its results



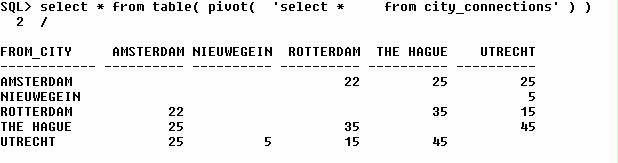
Of course we feel the need to prove the pudding once again, by adding the fifth city in the country – Nieuwegein (well, it not really the fifth city but it is where we (AMIS) are located as a company:

<code>insert into cities ( id, name) values ( 5, 'NIEUWEGEIN') / insert into connections ( from\_id , to\_id , distance ) values ( 5, 2, 5) / </code>

Note only a single connection, between Utrecht and Nieuwegein. On the map, this looks like:



Now will our pivot-query smoothly extend to include Nieuwegein?



And it does! (of course I knew it would otherwise I would not have raise the subject).

**Using that magical pivot function**

You may have grasped how we make use of that pivot function. It is pretty simple really: we pass it a varchar2 that contains a query. The query needs to query at least three columns. More is allowed. The columns can have any data type – NUMBER, VARCHAR2, DATE, INTERVAL, TIMESTAMP etc. Not sure about BLOB though – probably not a good idea.

The last column in the query you feed the pivot function turns up in the cells of the matrix. The one but last column provides the values for the columns in the matrix. The other columns are (all) used as row-label.

So for one more example: we would like to see which roles have privileges on which tables. In our schema, we have the tables CAREER, CITIES, DEPT, EMP and POINTS. We have granted access privileges to various roles and we would like an overview. We know that such information is available from the Data Dictionary View USER\_TAB\_PRIVS. This view has columns GRANTEE and TABLE\_NAME that are useful to us. We want Roles in the Rows of our matrix and Tables in the columns. So the query we offer up for pivoting should select three columns of which the role is the first one, the table\_name the second and the third should be whatever we want to display in the cells:

select \* from table( pivot( 'select grantee role

, table\_name

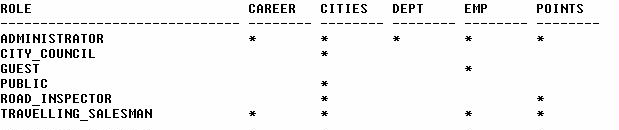
, '' \* ''

from user\_tab\_privs

')

)

/



Note: in this case we simply mark the cell if the role has a (any) privilege on the table. We could refine this to show the entire CRUD palet (for CREATE, RETRIEVE, UPDATE and DELETE privileges).

**How it works**

I really couldn’t tell you. Well, I can tell you some of it. But I am also trying to get my head round it.

The function pivot is defined as follows:

create or replace

function pivot

( p\_stmt in varchar2

, p\_fmt in varchar2 := 'upper(@p@)'

, dummy in number := 0

) return anydataset pipelined using PivotImpl;

/

The essence of the pivoting is clearly not in the function, but in the PivotImpl Type it leverages. The function is defined as pipelined, spitting out records one by one instead of as a whole bunch, and returns an AnyDataSet. Because of that return type, we can regard this function as a Table Function that we can use with the TABLE operator in a FROM clause: select \* from TABLE( TABLE\_FUN()).

The Pipelined Table Function is implemented with the Interface Approach, requires the user to supply a type that implements a predefined Oracle interface consisting of start, fetch, and close operations. See for details: [Chapter 13 – Pipelined Table Functions in the Oracle Data Cartridge Guide](http://www.stanford.edu/dept/itss/docs/oracle/10g/appdev.101/b10800/dcitblfns.htm#i1004689).

The interface approach requires the user to supply a type that implements a predefined Oracle interface consisting of start, fetch, and close operations. The type is associated with the table function when the table function is created. During query execution, the fetch method is invoked repeatedly to iteratively retrieve the results. With the interface approach, the methods of the implementation type associated with the table function can be implemented in any of the supported internal or external languages (including PL/SQL, C/C++, and Java).

**Resources**

Download the sources for this article: [AntonsPivoting.zip](http://technology.amis.nl/wp-content/uploads/images/antonsPivoting.zip). Connected as SCOTT, first run pivotFun.sql, then either pivot.sql or pivot\_distances.sql.

Also check out the discussion on AskTom (September 2012) on this article: http://asktom.oracle.com/pls/apex/f?p=100:11:0::::P11\_QUESTION\_ID:4843682300346852395#5394721000346803830.

[Oracle® Data Cartridge Developer’s Guide 10g Release 1 (10.1) – Chapter 13](http://www.stanford.edu/dept/itss/docs/oracle/10g/appdev.101/b10800/dcitblfns.htm#g1016436) Using Pipelined and Parallel Table Functions

[Oracle Data Cartridge Developer’s Guide – Chapter 17](http://www.stanford.edu/dept/itss/docs/oracle/10g/appdev.101/b10800/dcitblfnxemp.htm) Pipelined Table Functions: Interface Approach Example

Related posts:

1. [Pivot dynamic data with pipelined function](http://technology.amis.nl/2006/05/16/pivot-dynamic-data/)
2. [Pivoting in SQL using the 10g Model Clause](http://technology.amis.nl/2004/12/16/pivoting-in-sql-using-the-10g-model-clause/)
3. [Pop-quiz: VPD policy that depends on a table with a policy…](http://technology.amis.nl/2005/09/26/pop-quiz-vpd-policy-that-depends-on-a-table-with-a-policy/)
4. [Parallelizing Table Functions (instead of paralysing)](http://technology.amis.nl/2004/12/12/parallelizing-table-functions-instead-of-paralysing/)
5. [How Rownum can help you tune a Query](http://technology.amis.nl/2005/12/09/how-rownum-can-help-you-tune-a-query/)