PL/SQL 101 – WITH CLAUSE

Author: BluShadow
Last Updated: 19th January 2017

CONTENTS

1. Introduction – The Power WITHin

What is that mysterious WITH clause? This is something that still seems to take some people by surprise, yet it's been around for a few database versions now (as well as other database platforms).

One problem has been trying to find it in the documentation. Searching the docs for "WITH" is as good as searching the docs for the word "the" ... absolutely pointless, as it's usually not indexed by the search engine. You may hear some people refer to the WITH clause as a CTE or "Common Table Expression", however that is typically a term used by other database vendors. In Oracle this is better known as the "Subquery Factoring clause" and you'll find it in the documentation, quite logically when you consider it, under the SQL SELECT syntax (as that is where it gets used):

https://docs.oracle.com/database/121/SQLRF/statements_10002.htm#i2077142

In fact, for the 12c documentation you can now find the WITH clause by searching for "with_clause" on the database documentation. That doesn't work with the 11g or 10g documentation, but in 12c the documentation has been updated. In 10g and 11g the whole "WITH {subquery}" was considered the subquery_factoring_clause, but in 12c the "WITH" keyword is considered the "with_clause", and the "{subquery}" is considered the "subquery_factoring_clause". I'm not sure if the distinction makes it any better, but it's been introduced due to the new feature of 12c that allows for more than just subqueries... we can now include "plsql_declaration" which could be a pl/sql function or procedure.... yes you can now embed PL/SQL code directly in your queries... I guess they should call it SQL/PL for consistency, so we can have "WITH {subquery factoring clause}" and "WITH {plsql_declaration}". However, the plsql_declaration is a topic for another article. ☺

For this article I'm going to take a look at some of the main aspects of the WITH clause, available since 10g, and the enhancements in 11g.

2. Subquery Factoring – Why Create A Table?

In its simplest form, the WITH clause can be used to supply a query with some test data. In the 'old days'... yes I was there!... we created a table on our database and wrote insert statements, cluttering our schema up with more tables we'd forget to remove until the database (or DBA) complained the tablespace was full. To avoid doing that we moved on to using subqueries in our main query to create data on-the-fly ...

```
select e.empno
      ,e.ename
      ,e.mgr
      ,e.sal
      ,d.deptno
      ,d.dname
from  ( -- Employees
       select 1 as empno, 5 as mgr, 'BOB' as ename, 1000 as sal, 10 as deptno from dual union all
       select 2, 5, 'JOHN', 2000, 10 from dual union all
       select 3, 6, 'JIM', 3000, 20 from dual union all
       select 4, 6, 'TOM', 2500, 20 from dual union all
       select 5, 7, 'FRED', 5000, 30 from dual union all
       select 6, 7, 'JUDY', 5500, 30 from dual union all
       select 7, null, 'SCOTT', 10000, 30 from dual
      ) e
      join
      ( -- Departments
        select 10 as deptno, 'CONSULTANT' as dname from dual union all
        select 20, 'SALES' from dual union all
        select 30, 'MANAGEMENT' from dual
      ) d
      on (e.deptno = d.deptno)
/

     EMPNO ENAME        MGR        SAL     DEPTNO DNAME
---------- ----- ---------- ---------- ---------- ----------
         1 BOB            5       1000         10 CONSULTANT
         2 JOHN           5       2000         10 CONSULTANT
         3 JIM            6       3000         20 SALES
         4 TOM            6       2500         20 SALES
         5 FRED           7       5000         30 MANAGEMENT
         6 JUDY           7       5500         30 MANAGEMENT
         7 SCOTT                 10000         30 MANAGEMENT

7 rows selected.
```

Now that's all fine, until we decided we wanted, for example, the managers name shown for each employee. With tables, that would be a simple self-join to the employees table, however, using subqueries, we'd have had to repeat the whole subquery again...

```
select e.empno
      ,e.ename
      ,e.mgr
      ,m.ename as mgrname
      ,e.sal
      ,d.deptno
      ,d.dname
from  ( -- Employees
       select 1 as empno, 5 as mgr, 'BOB' as ename, 1000 as sal, 10 as deptno from dual union all
       select 2, 5, 'JOHN', 2000, 10 from dual union all
       select 3, 6, 'JIM', 3000, 20 from dual union all
       select 4, 6, 'TOM', 2500, 20 from dual union all
       select 5, 7, 'FRED', 5000, 30 from dual union all
       select 6, 7, 'JUDY', 5500, 30 from dual union all
       select 7, null, 'SCOTT', 10000, 30 from dual
      ) e
      join
      ( -- Departments
        select 10 as deptno, 'CONSULTANT' as dname from dual union all
        select 20, 'SALES' from dual union all
        select 30, 'MANAGEMENT' from dual
      ) d
      on (e.deptno = d.deptno)
      left outer join
      ( -- Copy of Employees for getting managers
       select 1 as empno, 5 as mgr, 'BOB' as ename, 1000 as sal, 10 as deptno from dual union all
       select 2, 5, 'JOHN', 2000, 10 from dual union all
       select 3, 6, 'JIM', 3000, 20 from dual union all
       select 4, 6, 'TOM', 2500, 20 from dual union all
       select 5, 7, 'FRED', 5000, 30 from dual union all
       select 6, 7, 'JUDY', 5500, 30 from dual union all
       select 7, null, 'SCOTT', 10000, 30 from dual
      ) m
      on (m.empno = e.mgr)
/

     EMPNO ENAME        MGR MGRNA        SAL     DEPTNO DNAME
---------- ----- ---------- ----- ---------- ---------- ----------
         2 JOHN           5 FRED        2000         10 CONSULTANT
         1 BOB            5 FRED        1000         10 CONSULTANT
         4 TOM            6 JUDY        2500         20 SALES
         3 JIM            6 JUDY        3000         20 SALES
         6 JUDY           7 SCOTT       5500         30 MANAGEMENT
         5 FRED           7 SCOTT       5000         30 MANAGEMENT
         7 SCOTT                       10000         30 MANAGEMENT

7 rows selected.
```

Obviously that's really annoying if we need to make some changes to our example data to try and reflect different data situations, especially if our query is quite complex, and we have a lot of test data and need lots of self joins. So, we want our subqueries to act as if they are tables, so we only have to specify them content of them once, but still without actually creating tables on the database. If only there was a way we could factorise those subqueries (take out the common factors) so we can reference them multiple times.

Hang on, that word "factorise" sounds familiar... what did we call that WITH clause again? Ah yes! the "Subquery Factoring Clause". It really does do what it says... it factors out the subqueries from our main query. Let's factor our example employees and departments data out of our query and put it outside the main query using the WITH clause.

```
WITH e as
      ( -- Employees
       select 1 as empno, 5 as mgr, 'BOB' as ename, 1000 as sal, 10 as deptno from dual union all
       select 2, 5, 'JOHN', 2000, 10 from dual union all
       select 3, 6, 'JIM', 3000, 20 from dual union all
       select 4, 6, 'TOM', 2500, 20 from dual union all
       select 5, 7, 'FRED', 5000, 30 from dual union all
       select 6, 7, 'JUDY', 5500, 30 from dual union all
       select 7, null, 'SCOTT', 10000, 30 from dual
      )
    ,d as
      ( -- Departments
        select 10 as deptno, 'CONSULTANT' as dname from dual union all
        select 20, 'SALES' from dual union all
        select 30, 'MANAGEMENT' from dual
      )

And now let's see how we can write our query against those factored subqueries...

WITH e as
      ( -- Employees
       select 1 as empno, 5 as mgr, 'BOB' as ename, 1000 as sal, 10 as deptno from dual union all
       select 2, 5, 'JOHN', 2000, 10 from dual union all
       select 3, 6, 'JIM', 3000, 20 from dual union all
       select 4, 6, 'TOM', 2500, 20 from dual union all
       select 5, 7, 'FRED', 5000, 30 from dual union all
       select 6, 7, 'JUDY', 5500, 30 from dual union all
       select 7, null, 'SCOTT', 10000, 30 from dual
      )
    ,d as
      ( -- Departments
        select 10 as deptno, 'CONSULTANT' as dname from dual union all
        select 20, 'SALES' from dual union all
        select 30, 'MANAGEMENT' from dual
      )
select e.empno
      ,e.ename
      ,e.mgr
      ,m.ename as mgrname
      ,e.sal
      ,d.deptno
      ,d.dname
from  e
                join d   on (e.deptno = d.deptno)
      left outer join e m on (m.empno = e.mgr)
/

     EMPNO ENAME        MGR MGRNA        SAL     DEPTNO DNAME
---------- ----- ---------- ----- ---------- ---------- ----------
         2 JOHN           5 FRED        2000         10 CONSULTANT
         1 BOB            5 FRED        1000         10 CONSULTANT
         4 TOM            6 JUDY        2500         20 SALES
         3 JIM            6 JUDY        3000         20 SALES
         6 JUDY           7 SCOTT       5500         30 MANAGEMENT
         5 FRED           7 SCOTT       5000         30 MANAGEMENT
         7 SCOTT                       10000         30 MANAGEMENT

7 rows selected.
```

So here we've been able to reference our employees table twice without having to repeat the actual data, just as if they were tables on the database (our second reference being aliased as "m" for managers). If we need to change our test data, we only need to change it in one place, and it'll be just like we had changed data on a single table, yet we've not had to create any tables on the database to write and test our query.

Likewise, if we already had tables on the database that, for example, contained millions of rows of data, and we just wanted to write and test a query (perhaps to troubleshoot a specific issue?) we could still use the WITH clause to selectively give us just some rows of data to work with... (for this I've created tables called "myemps" and "mydepts" on the database with my example data)

```
WITH e as
      ( -- Select just employees I want to test with
        select * from myemps where empno between 4 and 7
      )
    ,d as
      ( -- Select just departments I want to test with
        select * from mydepts where deptno between 10 and 30
      )
select e.empno
      ,e.ename
      ,e.mgr
      ,m.ename as mgrname
      ,e.sal
      ,d.deptno
      ,d.dname
from  e
                join d   on (e.deptno = d.deptno)
      left outer join e m on (m.empno = e.mgr)
/

     EMPNO ENAME        MGR MGRNA        SAL     DEPTNO DNAME
---------- ----- ---------- ----- ---------- ---------- ----------
         4 TOM            6 JUDY        2500         20 SALES
         6 JUDY           7 SCOTT       5500         30 MANAGEMENT
         5 FRED           7 SCOTT       5000         30 MANAGEMENT
         7 SCOTT                       10000         30 MANAGEMENT
```

The WITH clause lets us simulate or test with subsets of data that already exist or that we want to create on-the-fly, that we can then reference multiple times in our main query. It can also make our code easier to read as we can put the focus of our data in the WITH clause, or even have one subquery process a previous subquery further, before we put the crux of our processing in our main query.

Many members on the OTN Community will use WITH clauses to provide example data when supplying solutions to issues, so that they don't have to create the data directly on their database. The questioner can then use the same main query of the solution against their own database tables, without the unnecessary WITH clause subqueries.

3. Recursive Subquery Factoring (11gR2)

From Oracle 11gR2 onwards, the subquery factoring (WITH) clause was enhanced to include recursive processing. That means that the factored subquery could reference itself. If you tried to do that in 10g you'd get an exception... *ORA-32031: illegal reference of a query name in WITH clause*. Not very descriptive, but at least more indicative than it just saying the syntax is invalid, but basically telling you that you'd be ok if you references other previous WITH clause queries, but not the same as the query itself.

Recursion

If you know what recursion is already, you can skip this section, though I know some people have probably heard the term (or not) but don't have a clue what it really is.

Recursion is something I first came across when I was having interviews to attend University. One of the Universities gave a small introductory presentation to all the prospective students, which included a mention, and brief description of Recursion, and then in our individual interviews, they asked us to explain Recursion to them, I guess to test that a) we'd been listening and b) we has some aptitude to understand what they were talking about. It wasn't until I was actually studying at University that I found the usefulness of recursion, when writing a language parser to check the syntax of code for a language interpreter that we were required to write as an assignment. (Language parsing is most definitely another topic if you don't know what I'm referring to).

So, what are the basics of recursion... Let's start with a simple example, written in PL/SQL...

```
create or replace procedure recursion(startValue in number
                                      ,endValue   in number) as
begin
  if startValue < endValue then
    recursion(startValue+1, endValue);
  end if;
  dbms_output.put_line(startValue);
end;
/

SQL> exec recursion(1,5);
5
4
3
2
1

PL/SQL procedure successfully completed.
```

Now, it may seem confusing at first, why have my values been displayed in reverse, starting at 5 and ending in 1. Consider what the code is actually doing...

- We call our code with Start=1, End=5
- Our code tests the values and then calls itself with Start=2, End=5
- Our code tests the values and then calls itself with Start=3, End=5
- Our code tests the values and then calls itself with Start=4, End=5
- Our code tests the values and then calls itself with Start=5, End=5
- Our code tests the values, has no need to call itself, and outputs the Start value of 5
- Execution returns to the calling code (where Start=4, End=5), which outputs the Start value of 4
- Execution returns to the calling code (where Start=3, End=5), which outputs the Start value of 3
- Execution returns to the calling code (where Start=3, End=5), which outputs the Start value of 2
- Execution returns to the calling code (where Start=3, End=5), which outputs the Start value of 1
- Execution returns to the calling code… our EXEC statement, and completes

Each time the procedure code calls itself, it's creating a separate instance of the procedure in memory. Like when we call any procedure, once execution completes for that procedure instance the execution returns to the calling code (the previous instance of the procedure). Without some condition to stop the code calling itself, we would cause Oracle to eventually run out of memory, or the code to die some horrible death. So, for recursion to work, there must be some condition in the code that eventually stops the code calling itself and start returning back up the call stack.

How about a slightly more complex example. Let's take the standard Oracle EMP table, which has employees and their managers in a hierarchy, and create a recursive procedure that drills down the hierarchy from a given manager…

```
create or replace procedure emp_hierarchy(mgr in number := 7839
                                         ,lvl in number := 1
                                         ) is
  type tEmps is table of number;
  Emps tEmps;
begin
  -- Show our Manager record
  dbms_output.put_line('['||to_char(lvl,'fm9')||']'||lpad(' ',lvl,' ')||to_char(mgr));
  -- obtain the direct employees of the provided manager
  select empno
  bulk collect into Emps
  from   emp
  where  mgr = emp_hierarchy.mgr
  order by empno;
  -- Process each of the employees
  for i in 1 .. Emps.count
  loop
    emp_hierarchy(Emps(i), lvl+1);
  end loop;
end;
/

SQL> exec emp_hierarchy;
[1] 7839
[2]  7566
[3]   7788
[4]    7876
[3]   7902
[4]    7369
[2]  7698
[3]   7499
[3]   7521
[3]   7654
[3]   7844
[3]   7900
[2]  7782
[3]   7934

PL/SQL procedure successfully completed.
```

In this example, the code takes our "manager" and displays their details, then obtains a list of all the direct employees for that manager, and for each one calls the same procedure with that employee as the "manager".

As you can see, the "level" (lvl) indicates the depth of procedures calls in the execution stack, with 0 being the first call, 1 being the calls made by 0, and so on. Interestingly, you can see that the calls go down and back up and back down etc. until all of the hierarchical tree has been traversed. This is a "**depth first**" recursive hierarchy search.

Just out of interest, compare that with an SQL hierarchical query using CONNECT BY…

```
select '['||to_char(level)||']'||lpad(' ',level,' ')||to_char(empno) as result
from   emp
connect by mgr = prior empno
start with mgr is null
order siblings by empno
/

RESULT
------------------
[1] 7839
[2]  7566
[3]   7788
[4]    7876
[3]   7902
[4]    7369
[2]  7698
[3]   7499
[3]   7521
[3]   7654
[3]   7844
[3]   7900
[2]  7782
[3]   7934

14 rows selected.
```

We get the same result. CONNECT BY queries also have a “level” pseudo value, which likewise indicates the level of depth within its own recursive nature for traversing a hierarchy.

So, what are the key features of recursion?

1. Recursive processes start with some initial value or values. (START WITH)
2. Recursive processes reference themselves, usually with new values based on current values (CONNECT BY)
3. Recursive processes have some ‘exit’ condition to prevent further recursion taking place (implied by CONNECT BY)

How does this relate to the WITH clause and “recursive subquery factoring”… let’s find out.

RECURSING THE WITH

Based on our features of recursion, let’s start to build up a WITH clause that can recurs the EMP hierarchy we did above. We’ll start with the START WITH data.

```
with emp_hier(mgr, lvl) as (
       select empno, 1 from emp where mgr is null
     )
select '['||to_char(lvl)||']'||lpad(' ',lvl,' ')||to_char(mgr) as result
from emp_hier
/

RESULT
--------
[1] 7839
```

Ok, that’s simple enough, and is just a regular WITH clause for a subquery.

Note: For Recursive Subquery Factoring, the alias names of the columns should be specified in brackets directly after the query name, whilst this wasn’t required for the query above, it will be required as we progress, otherwise an exception would be raised.

In Recursive Subquery Factoring terms, the above "starting" data is referred to as the **Anchor Member.**
Now we want to have this WITH subquery reference itself, so how do we do that?

Quite simply we include a second query (referred to as the **Recursive Member**) inside the same WITH clause, and to join those two queries together we use a UNION ALL statement.

```
with emp_hier(mgr, lvl) as (
       -- ANCHOR QUERY
       select empno, 1 from emp where mgr is null
       union all
       -- RECURSIVE QUERY
       select emp.empno
             ,emp_hier.lvl+1
       from   emp
              join emp_hier on (emp.mgr = emp_hier.mgr)
      )
select '['||to_char(lvl)||']'||lpad(' ',lvl,' ')||to_char(mgr) as result
from emp_hier
/
```

What this Recursive Member query is doing is querying data from the EMP table, where the manager is supplied by the data in the emp_hier subquery. Oracle is clever enough to understand that, in the first instance, the data is from the anchor member query, and then subsequent iterations come from the data that is unioned to that, so it will keep on going for as long as there is further EMP records. The exit condition is implied by the lack of data when we reach employees who are not managers of anybody.

Let's look at the results:

```
RESULT
--------------------------------------
[1] 7839
[2]  7566
[2]  7698
[2]  7782
[3]   7499
[3]   7521
[3]   7654
[3]   7788
[3]   7844
[3]   7900
[3]   7902
[3]   7934
[4]    7369
[4]    7876

14 rows selected.
```

Well, that's close. It has all the employee records, and they are at the right "level" in the hierarchy, however each level has been produced in its entirety before going on to the next level.

This is known as a "**breadth first**" search. It's searching across each level first before going down.

Each time the Recursive Member query runs it unions all the records of that query before then recursing again to get the employees of all of those that have not yet been processed.

So, can we ask our Recursive Query to do a "depth first" search instead?

Yes we can…

We do this by adding a "search_clause" to the subquery; just after the closing bracket.

This search clause defines whether we want depth or breadth searching, by which column we want to order the search (and like a regular ORDER BY clause, whether we want it ascending or descending and whether nulls should come first or last), and a mandatory ordering column.

e.g.

```
search depth first by mgr set rn
```

In this case we tell it we want depth first searching, in the order of the mgr values, setting a overall ordering value in a new column called "rn". We can call that the ordering column any appropriate column name we like, and this is automatically added in to our results, so that our final query can order the data to get it in the order we desired.

Let's look at the full query…

```
with emp_hier(mgr, lvl) as (
       -- ANCHOR QUERY
       select empno, 1 from emp where mgr is null
       union all
       -- RECURSIVE QUERY
       select emp.empno
             ,emp_hier.lvl+1
       from   emp
              join emp_hier on (emp.mgr = emp_hier.mgr)
      ) search depth first by mgr set rn
select '['||to_char(lvl)||']'||lpad(' ',lvl,' ')||to_char(mgr) as result
from emp_hier
/

        RN RESULT
---------- ------------------------------
         1 [1] 7839
         2 [2]  7566
         3 [3]   7788
         4 [4]    7876
         5 [3]   7902
         6 [4]    7369
         7 [2]  7698
         8 [3]   7499
         9 [3]   7521
        10 [3]   7654
        11 [3]   7844
        12 [3]   7900
        13 [2]  7782
        14 [3]   7934

14 rows selected.
```

In this case, because I'm just selecting the data straight out of the query and not doing anything else, the order is coming out correctly. To be sure of the order though we should always include an "ORDER BY rn" (or whatever you called the ordering column) on our main query.

So, there we have it! A recursive subquery that can process data in a hierarchy.

For full details of recursive subquery factoring, its requirements and limitations, ensure you read the documentation:

https://docs.oracle.com/cd/E11882_01/server.112/e41084/statements_10002.htm#i2077142

4. So What Use Is It?

A recursive subquery factoring clause can have many uses, aside from hierarchical data processing; far more than I could imagine and describe here; but here's just a few examples. (*Note: not all examples will meet all situations, but are given as ideas of what can be achieved*)

Splitting Data

We can use recursive subquery factoring to split strings of data, whether it is delimited values of data (e.g. comma separated) or whether we just want to do something like word wrap some text to a certain number of characters…

```
with t(pk, str) as (select 1, 'This is a sentence we want to word wrap on 20 characters.' from dual
   union all select 2, 'And here we have another sentence.' from dual
   union all select 3, 'And one with the supercalfragilisticexpialidociously long word.' from dual
   )
    ,r(pk, str, ln, ln_text) as (
       -- Anchor -- obtain first portion up to 20 characters
       select pk
             ,substr(str
                    ,case when instr(substr(str,1,21),' ',-1) = 0 then 21
                     else      instr(substr(str,1,21),' ',-1)+1
                     end
                    ) as str
             ,1 -- first line we extract is line 1
             ,substr(str
                    ,1
                    ,case when instr(substr(str,1,21),' ',-1) = 0 then 20
                     else      instr(substr(str,1,21),' ',-1)-1
                     end
                    ) as ln_text
       from   (select pk, trim(str)||' ' as str from t)
       union all
       -- Recursive -- Do the same process for the remaining string
       select pk
             ,substr(str
                    ,case when instr(substr(str,1,21),' ',-1) = 0 then 21
                     else      instr(substr(str,1,21),' ',-1)+1
                     end
                    ) as str
             ,ln+1 -- with each line produced we increase the line number
             ,substr(str
                    ,1
                    ,case when instr(substr(str,1,21),' ',-1) = 0 then 20
                     else      instr(substr(str,1,21),' ',-1)-1
                     end
                    ) as ln_text
       from   r
       -- exit condition - where there is no remaining string
       where  r.str is not null
      )
select pk, ln, ln_text
from r
order by pk, ln
/

        PK         LN LN_TEXT
---------- ---------- --------------------------------------------------------------------
         1          1 This is a sentence
         1          2 we want to word wrap
         1          3 on 20 characters.
         2          1 And here we have
         2          2 another sentence.
         3          1 And one with the
         3          2 supercalfragilistice
         3          3 xpialidociously long
         3          4 word.

9 rows selected.
```

Multiple Replacement

When we want to replace several things in a string, we cannot use a single replace statement, and we cannot even use something more powerful, such as a the regular expression "regexp_replace", as each requires multiple function calls, nested within each other, each having separate search criteria and replacement criteria. What if we just want to supply a single set of criteria and have something that can effectively loop through that criteria and perform multiple replacements. Whilst it still can't be done in a single function call, we can use recursion to act as a looping mechanism, allowing us to just supply a single set of replacement criteria.

```
with t(str, repl) as (
       select 'Dear ~, As of ~ you have a credit limit of ~ ~.'
             ,'Mr Smith~12th January 2017~10,000~GBP'
       from   dual
      )
    ,r(str, repl) as (
       -- Anchor
       select regexp_replace(str
                             ,'~'
                             ,substr(repl,1,instr(repl||'~','~')-1)
                             ,1 -- from the start of string
                             ,1 -- only the first occurrence
                             ) as str
             ,substr(repl
                    ,instr(repl||'~','~')+1
                    ) as repl
       from   t
       union all
       -- Recursive
       select regexp_replace(str
                             ,'~'
                             ,substr(repl,1,instr(repl||'~','~')-1)
                             ,1 -- from the start of string
                             ,1 -- only the first occurrence
                             ) as str
             ,substr(repl
                    ,instr(repl||'~','~')+1
                    ) as repl
       from   r
       where  repl is not null
      )
select str
from   r
where  repl is null
/

STR
--------------------------------------------------------------------------------
Dear Mr Smith, As of 12th January 2017 you have a credit limit of 10,000 GBP.
```

Obviously, storing data in delimited format isn't recommended in the first place (and there are limits on the size of strings!), but this could be an example of "mail merge" where the data is being retrieved from an external table of delimited data (e.g. a CSV file).

One advantage of this method is that it doesn't matter how many replacements are needed; which you'd need to know if you were doing static replacements; as long as you ensure you have sufficient data for the number of replacements required.

CONVERTING DATA

In reality this can be just another form of multiple replacement. For example if we have a binary string and we want to get the Hexadecimal value for that string (oracle already provides a way of doing NUMBER to Hex using TO_CHAR, but not for binary values)…

```
with hx(val, hx) as ( -- Lookup table for binary to hex - 16 digits
        select substr('0000000100100011010001010110011110001001101010111100110111101111'
                     ,((level-1)*4)+1,4
                     ) as val
              ,substr('0123456789ABCDEF',level,1) as hx
        from dual
        connect by level <= 16
       )
    ,bin as (select '&binary_number' as bin from dual) -- User input
    ,hx2bin(bin, hx, final) as (
       -- Anchor
       select bin, cast(null as varchar2(20)) as hx, 0 as final
       from bin
       union all
       -- Recursive
        select substr(hx2bin.bin,1,length(hx2bin.bin)-4) as bin
              ,hx.hx||hx2bin.hx as hx
              ,case when length(hx2bin.bin) <= 4 then 1 else 0 end as final
        from   hx2bin
               join hx on (hx.val = lpad(substr(hx2bin.bin
                                                ,-least(4,length(hx2bin.bin))
                                                )
                                         ,4,'0'))
       )
select hx
from   hx2bin
where  final = 1
/

Enter value for binary_number: 10100101010100100101
old   9:     ,bin as (select '&binary_number' as bin from dual) -- User input
new   9:     ,bin as (select '10100101010100100101' as bin from dual) -- User input

HX
--------------------
A5525
```

This is taking 4 binary digits at a time, from the right hand side and looks up the hexadecimal digit for those, building up the hex value. However, knowing that binary is just multiples of 2, and that we can use TO_CHAR to convert numbers to Hex, we could also do…

```
SQL> with bin as (select '&binary_number' as bin from dual) -- User input
  2      ,hx2bin(bin, val) as (
  3         select bin, 0
  4         from   bin
  5         union all
  6         select substr(bin,2), (val*2)+to_number(substr(bin,1,1))
  7         from   hx2bin
  8         where  bin is not null
  9       )
 10  select val
 11        ,to_char(val, 'fmXXXXXXXX') as hx
 12  from   hx2bin
 13  where  bin is null
 14  /
Enter value for binary_number: 10100101010100100101
old   1: with bin as (select '&binary_number' as bin from dual) -- User input
new   1: with bin as (select '10100101010100100101' as bin from dual) -- User input

       VAL HX
---------- ---------
    677157 A5525
```

However, this is limited by the maximum value a NUMBER datatype can hold, whereas the first method is only limited by the size of the binary string being supplied.

Expression Parsing

Language parsers (as used by compilers) often use recursive techniques, and typically rely on tokenising the language into component parts. You may (or may not) have noticed when trying to compile some code languages, such as PL/SQL., with errors in it, the error reported is often the one lowest in the code and as you fix that error, the next error above that is reported on the subsequent compilation attempt. This is partially due to the "depth first" recursive nature of language parsers, which break the syntax down to the bottom first and then validate things on the way back up. In case you haven't noticed before, here's an example…

```
SQL> create or replace procedure bob as
  2  begin
  3    for i in 1 .. fred()  -- non existent function
  4    loop
  5      dbms_output.put_line('Here's another error with quotes');
  6    end loop;
  7  end;
  8  /

Warning: Procedure created with compilation errors.

SQL> sho err
Errors for PROCEDURE BOB:

LINE/COL ERROR
-------- -----------------------------------------------------------------
5/32     PLS-00103: Encountered the symbol "S" when expecting one of the
         following:
         ) , * & = - + < / > at in is mod remainder not rem =>
         <an exponent (**)> <> or != or ~= >= <= <> and or like like2
         like4 likec as between from using || multiset member
         submultiset
```

This procedure has two errors in it. Firstly we have a call to a function "fred" that I know doesn't exist, and secondly, I haven't escaped my quotes in the string on the dbms_output line. The first error the compiler reports to us, is the issue with the quotes on line 5, and it's not yet telling us about "fred". If we correct the quotes issue, we now get…

```
SQL> create or replace procedure bob as
  2  begin
  3    for i in 1 .. fred()  -- non existent function
  4    loop
  5      dbms_output.put_line(q'[Here's a fixed error with quotes]');
  6    end loop;
  7  end;
  8  /

Warning: Procedure created with compilation errors.

SQL> sho err
Errors for PROCEDURE BOB:

LINE/COL ERROR
-------- -----------------------------------------------------------------
3/3      PL/SQL: Statement ignored
3/17     PLS-00201: identifier 'FRED' must be declared
```

The compiler tells us about "fred", even though that was an issue that appeared earlier in the code.
This shows us that the PL/SQL compiler does its language parsing in a recursive fashion.

Now, whilst full language parsers are typically coded as a lot of separate recursive procedures (and a full example would far exceed the scope of this PL/SQL 101 article), can we use SQL's recursive subquery factoring to break up some simple expressions into their component parts i.e. tokenize them. Of course we can… (and for this I'll use a real example that appeared on the OTN Community as a question a while ago)…

If we take an expression:

```
((({A}&{B})||({C}||{D})&({E}||({F}&{G})))
```

Can we break this down to the component parts that are defined by pairs of parentheses (brackets)?

We start with a recursive subquery that parses the "tokens" of the string (individual parentheses "(" and ")", values/variables defined by surrounding braces "{" and "}", and operators "&" and "||")…

```
with t as (select '(({A}&{B})||({C}||{D})&({E}||({F}&{G})))' as exp from dual)
    ,e(exp, l, v, rn) as
      (select exp, 0, cast(null as varchar2(3)) as v, 0 as rn from t
       union all
       select case when substr(e.exp, 1, 1) = '{' then substr(e.exp,4)
                   when substr(e.exp, 1, 1) = '|' then substr(e.exp,3)
              else substr(e.exp,2)
              end as exp
             ,case when substr(e.exp, 1, 1) = '(' then e.l+1
                   when e.v = ')' then e.l-1
              else e.l
              end as l
             ,case when substr(e.exp, 1, 1) = '{' then substr(e.exp,1,3)
                   when substr(e.exp, 1, 1) = '|' then substr(e.exp,1,2)
              else substr(e.exp,1,1)
              end as v
             ,e.rn+1 as rn
       from   e
       where  e.exp is not null
      )
select * from e

EXP                                        L V           RN
------------------------------------------ ---------- --- ----------
(({A}&{B})||({C}||{D})&({E}||({F}&{G})))   0              0
({A}&{B})||({C}||{D})&({E}||({F}&{G})))    1 (            1
{A}&{B})||({C}||{D})&({E}||({F}&{G})))     2 (            2
&{B})||({C}||{D})&({E}||({F}&{G})))        2 {A}          3
{B})||({C}||{D})&({E}||({F}&{G})))         2 &            4
)||({C}||{D})&({E}||({F}&{G})))            2 {B}          5
||({C}||{D})&({E}||({F}&{G})))             2 )            6
({C}||{D})&({E}||({F}&{G})))               1 ||           7
{C}||{D})&({E}||({F}&{G})))                2 (            8
||{D})&({E}||({F}&{G})))                   2 {C}          9
{D})&({E}||({F}&{G})))                     2 ||          10
)&({E}||({F}&{G})))                        2 {D}         11
&({E}||({F}&{G})))                         2 )           12
({E}||({F}&{G})))                          1 &           13
{E}||({F}&{G})))                           2 (           14
||({F}&{G})))                              2 {E}         15
({F}&{G})))                                2 ||          16
{F}&{G})))                                 3 (           17
&{G})))                                    3 {F}         18
{G})))                                     3 &           19
)))                                        3 {G}         20
))                                         3 )           21
)                                          2 )           22
                                           1 )           23
```

Here you can see the tokenization taking place; the string is stripped down, and a level "L" is determined based on the opening and closing parentheses, whilst each token value "V" is extracted, and a row number "RN" used to keep track of the order of everything.

From this we can see our tokens and the "depth" at which they exist, and because of the order we can see from that depth how things should be grouped together.

There are various methods we could use to re-aggregate those groups of things, but as we're talking about recursive subquery factoring, let's use that method to give us a set of results that makes sense…

```
with t as (select '(({A}&{B})||({C}||{D})&({E}||({F}&{G})))' as exp from dual)
    ,e(exp, l, v, rn) as
      (select exp, 0, cast(null as varchar2(3)) as v, 0 as rn from t
       union all
       select case when substr(e.exp, 1, 1) = '{' then substr(e.exp,4)
                   when substr(e.exp, 1, 1) = '|' then substr(e.exp,3)
              else substr(e.exp,2)
              end as exp
             ,case when substr(e.exp, 1, 1) = '(' then e.l+1
                   when e.v = ')' then e.l-1
              else e.l
              end as l
             ,case when substr(e.exp, 1, 1) = '{' then substr(e.exp,1,3)
                   when substr(e.exp, 1, 1) = '|' then substr(e.exp,1,2)
              else substr(e.exp,1,1)
              end as v
             ,e.rn+1 as rn
       from   e
       where  e.exp is not null
      )
   ,j(rn, lvl, v, start_l) as
     (select rn, l, v, 1 from e where v = '('
      union all
      select e.rn
            ,e.l
            ,cast(j.v as varchar2(100))||e.v as v
            ,case when e.v = '(' then j.start_l + 1
                  when e.v = ')' then j.start_l - 1
             else j.start_l
             end as start_l
      from   e join j on (e.rn = j.rn+1)
      where  j.start_l > 0 or (j.start_l = 0 and j.v = '(')
     ) search depth first by rn set seq
select lvl, v
from j
where start_l = 0
/

       LVL V
---------- ----------------------------------------
         1 (({A}&{B})||({C}||{D})&({E}||({F}&{G})))
         2 ({A}&{B})
         2 ({C}||{D})
         2 ({E}||({F}&{G}))
         3 ({F}&{G})
```

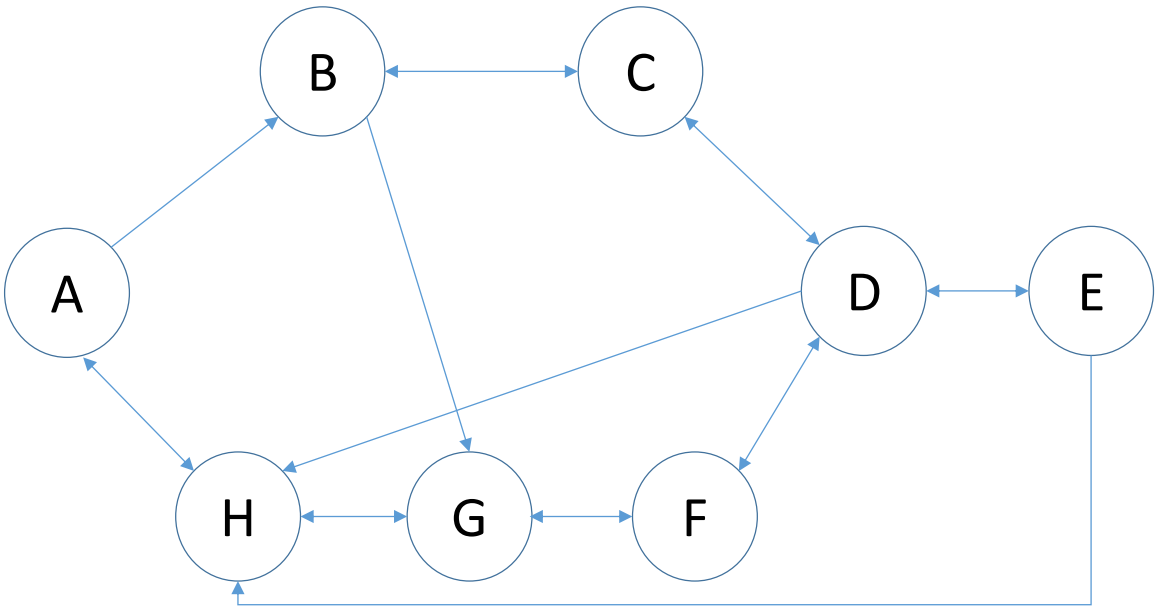
And there we have it, the whole expression broken down into its components expressions as defined by the parentheses. Level 1 is the whole expression, Level 2 shows each of the expressions within Level 1, and Level 3 shows the expressions within any Level 2 expression; in this case the "`({F}&{G})`" expression that is within the "`({E}||({F}&{G}))`".

Note in the above recursive subquery that the Anchor member supplies several starting rows of data i.e. all the rows that have an opening parenthesis "(". This gives it the start of each (sub) expression and by counting the opening and closing parenthesis as it joins the data back together, it can determine when it's concluded on a closing parenthesis ")".

SHORTEST PATH

All the previous examples have essentially come down to using recursive subquery factoring to process strings; and we've seen in earlier sections how it can also be used for hierarchical queries. In this last example, we'll do something a little different; finding the shortest path between 'nodes' in a network. This is the sort of processing where you may have, for example, flights between different places and want to find the flight with the least changes that gets you from A to B.

Let's define what our data is going to look like. Some of our 'flights' will only go in one direction, and some can go in both directions. For simplicity we'll just refer to our places as A,B,C,D,… etc.



We'll create the data as a table on our database to allow us to focus on the query without having to repeat the data in each step.

```
create table flights (
  pk number
 ,src varchar2(1) /* Source of Flight */
 ,dst varchar2(1) /* Destination of Flight */
 )
/
insert into flights
  select  1, 'A', 'B' from dual union all
  select  2, 'A', 'H' from dual union all
  select  3, 'B', 'C' from dual union all
  select  4, 'B', 'G' from dual union all
  select  5, 'C', 'B' from dual union all
  select  6, 'C', 'D' from dual union all
  select  7, 'D', 'C' from dual union all
  select  8, 'D', 'E' from dual union all
  select  9, 'D', 'F' from dual union all
  select 10, 'D', 'H' from dual union all
  select 11, 'E', 'D' from dual union all
  select 12, 'E', 'H' from dual union all
  select 13, 'F', 'D' from dual union all
  select 14, 'F', 'G' from dual union all
  select 15, 'G', 'F' from dual union all
  select 16, 'G', 'H' from dual union all
  select 17, 'H', 'A' from dual union all
  select 18, 'H', 'G' from dual
/
commit;
```

So consider what we need to traverse the paths on this network of flights:

1) We need to track the path of what airports (nodes) we've been to already so we don't go back there
2) We need a record of each flight in order that they're taken, so we can show the results at the end
3) We need to know how many changes are made to reach the destination, as we're looking for the least changes
4) We need to know where each flight is taking us (the individual destinations) so we can pick up what flights are available from there when we get there in the next recursive step.
5) We need to know if we've reached our final destination.

We also need to know where our traveller wants to get from and to. In that case we'll start by prompting the user for their starting location and their desired location:

```
with req(src, dst) as (select '&source', '&destination' from dual) -- User Input
```

This is a useful use of the WITH clause to obtain values just once which we can then use throughout any subsequent queries (we also used the same to prompt for our binary number in an earlier example)

Now let's start our recursive subquery, by creating our Anchor member:

```
with req(src, dst) as (select '&source', '&destination' from dual) -- User Input
    ,r(path, flights, changes, flight_dst, dst_reached) as (
       -- ANCHOR - all flights from the source
       select cast(to_char(flights.src) as varchar2(50)) as path
             ,cast(flights.src||'->'||flights.dst as varchar2(150))as flights
             ,0 as changes
             ,flights.dst as flight_dst
             ,case when flights.dst = req.dst then 'Y' else 'N' end as dst_reached
       from   flights
              join req on (flights.src = req.src)
      )
select *
from   r

Enter value for source: A
Enter value for destination: E
old   1: with req(src, dst) as (select '&source', '&destination' from dual) -- User Input
new   1: with req(src, dst) as (select 'A', 'E' from dual) -- User Input

PATH                           FLIGHTS                                    CHANGES F D
------------------------------ ---------------------------------------- ---------- - -
A                              A->B                                            0 B N
A                              A->H                                            0 H N
```

This query introduces the 5 elements we identified as needed above. Our "path" starts at the desired source, and we have both possible flights from that source. At this points there are zero changes, we've recorded the destination of those flights and we've identified if those destinations are the final desired destination or not (in this case neither are).

To add the recursive part of our query we want it to pick any flights that go from the previous destinations, and which do not go back to (any of) the airports we have already been to (which we're recording in the "path"). With each progression through the recursive levels we increase our number of "changes" and likewise determine if we've reached the final destination. Let's see how that code looks…

```
with req(src, dst) as (select '&source', '&destination' from dual) -- User Input
    ,r(path, flights, changes, flight_dst, dst_reached) as (
       -- ANCHOR - all flights from the source
       select cast(to_char(flights.src) as varchar2(50)) as path
             ,cast(flights.src||'->'||flights.dst as varchar2(150))as flights
             ,0 as changes
             ,flights.dst as flight_dst
             ,case when flights.dst = req.dst then 'Y' else 'N' end as dst_reached
       from   flights
              join req on (flights.src = req.src)
       union all
       -- RECURSIVE
       select r.path||'#'||to_char(flights.src) as path
             ,r.flights||', '||flights.src||'->'||flights.dst as flights
             ,r.changes+1 as changes
             ,flights.dst as flight_dst
             ,case when flights.dst = req.dst then 'Y' else 'N' end as dst_reached
       from   r
              join flights on (    flights.src = r.flight_dst
                               -- Prevent going back to an airport we have been to already
                               -- Acts as an exit condition and prevents cycling
                               and instr('#'||r.path||'#', '#'||to_char(flights.dst)||'#') = 0
                              )
              cross join req
       -- Only continue following path where destination not already reached
       where  r.dst_reached = 'N'
      )
select *
from   r
/

Enter value for source: A
Enter value for destination: E
old   1: with req(src, dst) as (select '&source', '&destination' from dual) -- User Input
new   1: with req(src, dst) as (select 'A', 'E' from dual) -- User Input

PATH                           FLIGHTS                                    CHANGES F D
------------------------------ ---------------------------------------- ---------- - -
A                              A->B                                              0 B N
A                              A->H                                              0 H N
A#B                            A->B, B->C                                        1 C N
A#B                            A->B, B->G                                        1 G N
A#H                            A->H, H->G                                        1 G N
A#B#C                          A->B, B->C, C->D                                  2 D N
A#H#G                          A->H, H->G, G->F                                  2 F N
A#B#G                          A->B, B->G, G->F                                  2 F N
A#B#G                          A->B, B->G, G->H                                  2 H N
A#B#C#D                        A->B, B->C, C->D, D->E                            3 E Y
A#B#C#D                        A->B, B->C, C->D, D->F                            3 F N
A#B#C#D                        A->B, B->C, C->D, D->H                            3 H N
A#B#G#F                        A->B, B->G, G->F, F->D                            3 D N
A#H#G#F                        A->H, H->G, G->F, F->D                            3 D N
A#H#G#F#D                      A->H, H->G, G->F, F->D, D->C                      4 C N
A#B#G#F#D                      A->B, B->G, G->F, F->D, D->C                      4 C N
A#H#G#F#D                      A->H, H->G, G->F, F->D, D->E                      4 E Y
A#B#G#F#D                      A->B, B->G, G->F, F->D, D->E                      4 E Y
A#B#G#F#D                      A->B, B->G, G->F, F->D, D->H                      4 H N
A#B#C#D#F                      A->B, B->C, C->D, D->F, F->G                      4 G N
A#B#C#D#H                      A->B, B->C, C->D, D->H, H->G                      4 G N
A#H#G#F#D#C                    A->H, H->G, G->F, F->D, D->C, C->B                5 B N
A#B#C#D#H#G                    A->B, B->C, C->D, D->H, H->G, G->F                5 F N
A#B#C#D#F#G                    A->B, B->C, C->D, D->F, F->G, G->H                5 H N

24 rows selected.
```

Our “path”, as we can see, builds up a (delimited) list of airports we’ve been to as we recurs the network of flights, and this is used to check that we don’t try and return to any airport we’ve already been to, and thus prevents us from cycling the data.

All possible paths are followed, until we have exhausted the possible paths through the network. Some of those paths are clearly identified as having reached the desired destination whilst others haven’t (but they had nowhere else to go). If we now tidy up our query to just show the data we require (the flights and number of changes) and filter it so that we only retrieve those that reach the desired destination…

```
with req(src, dst) as (select '&source', '&destination' from dual) -- User Input
    ,r(path, flights, changes, flight_dst, dst_reached) as (
       -- ANCHOR - all flights from the source
       select cast(to_char(flights.src) as varchar2(50)) as path
             ,cast(flights.src||'->'||flights.dst as varchar2(150))as flights
             ,0 as changes
             ,flights.dst as flight_dst
             ,case when flights.dst = req.dst then 'Y' else 'N' end as dst_reached
       from   flights
              join req on (flights.src = req.src)
       union all
       -- RECURSIVE
       select r.path||'#'||to_char(flights.src) as path
             ,r.flights||', '||flights.src||'->'||flights.dst as flights
             ,r.changes+1 as changes
             ,flights.dst as flight_dst
             ,case when flights.dst = req.dst then 'Y' else 'N' end as dst_reached
       from   r
              join flights on (    flights.src = r.flight_dst
                                   -- Prevent going back to an airport we have been to already
                                   -- Acts as an exit condition and prevents cycling
                                   and instr('#'||r.path||'#', '#'||to_char(flights.dst)||'#') = 0
                              )
              cross join req
       -- Only continue following path where destination not already reached
       where  r.dst_reached = 'N'
      )
select flights, changes
from   r
where  dst_reached = 'Y'
/

Enter value for source: A
Enter value for destination: E
old   1: with req(src, dst) as (select '&source', '&destination' from dual) -- User Input
new   1: with req(src, dst) as (select 'A', 'E' from dual) -- User Input

FLIGHTS                                  CHANGES
---------------------------------------- ----------
A->B, B->C, C->D, D->E                            3
A->H, H->G, G->F, F->D, D->E                      4
A->B, B->G, G->F, F->D, D->E                      4
```

We get all possible paths in the network from our source to our destination.

Let's check out some other paths to check it's working ok (compare it to the network diagram). How about a return flight from E to A for starters?

```
SQL> /
Enter value for source: E
Enter value for destination: A
old   1: with req(src, dst) as (select '&source', '&destination' from dual) -- User Input
new   1: with req(src, dst) as (select 'E', 'A' from dual) -- User Input

FLIGHTS                                    CHANGES
---------------------------------------- ----------
E->H, H->A                                        1
E->D, D->H, H->A                                  2
E->D, D->F, F->G, G->H, H->A                      4
E->D, D->C, C->B, B->G, G->H, H->A                5
```

There are more options for flying back home, and even one route that only requires 1 change, whereas we had to change 3 times at least to go on our outward journey.

Let's pick another "holiday", how about H to E?

```
SQL> /
Enter value for source: H
Enter value for destination: E
old   1: with req(src, dst) as (select '&source', '&destination' from dual) -- User Input
new   1: with req(src, dst) as (select 'H', 'E' from dual) -- User Input

FLIGHTS                                    CHANGES
---------------------------------------- ----------
H->G, G->F, F->D, D->E                            3
H->A, A->B, B->C, C->D, D->E                      4
H->A, A->B, B->G, G->F, F->D, D->E                5
```

Again it's going to take at least 3 changes to get to the destination.

And the return journey…?

```
SQL> /
Enter value for source: E
Enter value for destination: H
old   1: with req(src, dst) as (select '&source', '&destination' from dual) -- User Input
new   1: with req(src, dst) as (select 'E', 'H' from dual) -- User Input

FLIGHTS                                    CHANGES
---------------------------------------- ----------
E->H                                              0
E->D, D->H                                        1
E->D, D->F, F->G, G->H                            3
E->D, D->C, C->B, B->G, G->H                      4
```

Excellent, there's a direct flight.

For picking out just the flights with the least changes, that's just a simple case of using standard SQL. The results already tell us the number of changes, so using "… KEEP (DENSE_RANK FIRST …" or other ranking methods in our main query we could easily exclude all but the routes with the least changes. I'll leave that as something for you to play with.

SUMMARY

In summary we've seen how the WITH clause can be used to factor out subqueries, just for providing example data, or for allowing re-use of subqueries in our main query; how we can use them to traverse a hierarchy of data, and then we've gone on to look at the power of using recursive subquery factoring to offer greater processing power, including traversing a network of data. The examples given are by no means complete, or necessarily the most efficient means of achieving what you want; but hopefully they give you an idea of how flexible factored subqueries and recursive subquery factoring can be, for achieving processing of data that you may otherwise have considered using PL/SQL code for.

If you feel you're still not familiar with the WITH clause (though hopefully you've been playing along and trying things yourself), take some time to read through the documentation:

https://docs.oracle.com/cd/E11882_01/server.112/e41084/statements_10002.htm#i2077142