# Recursive Common Table Expressions in Oracle Database 11g Release 2

*Iggy Fernandez, Database Specialists*

## Introduction

Oracle was late to the table with recursive common table expressions which have been part of the SQL standard since 2003 but—to Oracle's credit—it has provided the CONNECT BY clause for hierarchical queries from the very beginning. However, recursive common table expressions can be used for much more than hierarchical queries. Also note that Oracle uses the non-standard terms "subquery factoring" and "recursive subquery factoring" for "common table expressions" and "recursive common table expressions" respectively.

## Recap of Non-Recursive Common Table Expressions

Here is a quick recap of common table expressions of the non-recursive kind. They are an alternative to inline views and make a complex SQL query much easier to read and maintain. Another advantage is that they only need to be evaluated once if they are used more than once within the query. Here is an example: suppliers who supply all parts. Here is the data for the example.

```
CREATE TABLE suppliers
(
  supplername VARCHAR2(30)
);

CREATE TABLE parts
(
  partname VARCHAR2(30)
);

CREATE TABLE quotes
(
  suppliername VARCHAR2(30),
  partname VARCHAR2(30),
  quote NUMBER
);
```

Here's how we might formulate the query using traditional inline views.

```
SELECT *
FROM
  -- Suppliers Who Supply All Parts
  (
    -- Suppliers
    (
      SELECT *
      FROM Suppliers
    )
    MINUS
    -- Suppliers Who Don't Supply All Parts
    (
      SELECT SupplierName
      FROM
      -- Invalid Supplier Part Pairs
```

```
        (
          -- All Supplier Part Pairs
          (
            SELECT *
            FROM Suppliers, Parts
          )
          MINUS
          -- Valid Supplier Part Pairs
          (
            SELECT SupplierName, PartName
            FROM Quotes
          )
        )
      )
    );
```

Even with the provided annotations, the above formulation is difficult to read and understand. Common table expressions can be used to formulate the following much more readable version.

```
WITH

AllSupplierPartPairs AS
(
  SELECT *
  FROM Suppliers, Parts
),

ValidSupplierPartPairs AS
(
  SELECT SupplierName, PartName
  FROM Quotes
),

InvalidSupplierPartPairs AS
(
  SELECT *
  FROM AllSupplierPartPairs
  MINUS
  SELECT *
  FROM ValidSupplierPartPairs
),

SuppliersWhoDontSupplyAllParts AS
(
  SELECT SupplierName
  FROM InvalidSupplierPartPairs
),

SuppliersWhoSupplyAllParts AS
(
  SELECT *
  FROM Suppliers
  MINUS
  SELECT *
  FROM SuppliersWhoDontSupplyAllParts
)
```

```
SELECT *
FROM SuppliersWhoSupplyAllParts;
```

## Structure of Recursive Common Table Expressions

On to recursive common table expressions. A recursive common table expression (recursive CTE) contains subqueries called "anchor members" and "recursive members." The rows produced by the anchor members are processed by the recursive members. The recursive members produce other rows that are fed right back to them for further processing. Recursion stops only when the recursive members fail to produce additional rows. The explanation in the Oracle documentation is fairly cryptic but a good explanation can be found on the Microsoft Developer Network.

1. Run the anchor member(s) creating the first invocation or base result set (T0).
2. Run the recursive member(s) with Ti as an input and Ti+1 as an output.
3. Repeat step 3 until an empty set is returned.
4. Return the result set. This is a UNION ALL of T0 to Tn.

## A Number Generator

Here's a simple example of a recursive common table expression: a number generator. The following example generates the consecutive numbers from 1 to 9. The anchor member generates the first row which is then processed by the recursive member. The recursive member uses the name of the recursive CTE as a placeholder for the output of the anchor member or a previous execution of the recursive CTE. In this example, each execution of the recursive member produces one more row. Recursion stops when nine records have been produced.

```
WITH

-- The following number generator is a simple example of a recursive CTE. It
-- produces the consecutive digits from 1 to 9.

Numbers(n) AS
(

  -- The "anchor member." It contains exactly one row (N = 1).

  SELECT 1 AS N
  FROM dual

  UNION ALL

  -- The "recursive member." Notice that it references the name of the recursive
  -- CTE as a placeholder for the results of the anchor member or the previous
  -- execution of the recursive CTE. Each iteration of the recursive member
  -- produces the next value of N. Recursive execution stops when N = 9.

  SELECT N + 1 AS N
  FROM Numbers
  WHERE N < 9

)

SELECT *
FROM Numbers;
```

The above example can be simply duplicated using the CONNECT BY clause as follows:

```
SELECT level AS N
FROM dual
```

```
  CONNECT BY level <= 9;
```

## A Standard Hierarchical Query

Next consider a standard hierarchical query; an org-chart of managers and employees. First, here's the old solution using the CONNECT BY clause; it is short and sweet.

```
  SELECT
    LPAD (' ', 4 * (LEVEL - 1)) || first_name || ' ' || last_name AS name
  FROM employees
  START WITH manager_id IS NULL
  CONNECT BY manager_id = PRIOR employee_id;
```

The solution using recursive common table expressions is much more verbose. Note especially the SEARCH DEPTH FIRST clause; refer to the Oracle documentation for an explanation.

```
  WITH

  RecursiveCTE (employee_id, first_name, last_name, lvl) AS
  (

    -- The "anchor member" of the recursive CTE. It locates employees who don't
    -- have any manager; presumably there is at least one such employee.

    SELECT
      employee_id,
      first_name,
      last_name,
      1 AS lvl
    FROM
      employees
    WHERE manager_id IS NULL

    UNION ALL

    -- The "recursive member" of the recursive CTE. Notice that it uses the name
    -- of the recursive CTE as a placeholder for the results of the anchor member
    -- or the previous execution of the recursive CTE. Each iteration of the
    -- recursive member locates the employees who report to the employees located
    -- in the previous iteration. Recursive execution stops when all employees
    -- have been located.

    SELECT
      e.employee_id,
      e.first_name,
      e.last_name,
      lvl + 1 AS lvl
    FROM
      RecursiveCTE r INNER JOIN employees e
      ON (r.employee_id = e.manager_id)

  )

  -- Go deep in order to produce records in exactly the same order as the CONNECT
  -- BY clause. The default order of processing is BREADTH FIRST which would
  -- produce all managers at the same level before any of their employees; this is
  -- not not the order in which the CONNECT BY produces rows. The pseudocolumn
  -- seq# has been designated here to capture the order in which records are
```

```
-- produced by the recursive CTE; it will be used in the main query.

SEARCH DEPTH FIRST BY employee_id ASC SET seq#

-- This is the main query. It processes the results produced by the recursive
-- CTE.

SELECT LPAD (' ', 4 * (lvl - 1)) || first_name || ' ' || last_name AS name
FROM RecursiveCTE
ORDER BY seq#;
```

## A More Complex Example

From the two examples above, it might appear that a recursive CTE is little more than a verbose way of specifying what could be more succinctly achieved with the CONNECT BY clause. However recursive common table expressions are significantly more powerful than the CONNECT BY clause. For example, consider the following example from the TSQL Challenges team:

1. Given a list of products and a list of discount coupons, we need to use the following rules to find the minimum price for each product:
2. Not more than two coupons can be applied to the same product.
3. The discounted price cannot be less than 70% of the original price.
4. The total amount of the discount cannot exceed $30.

Syed Mehroz Alam provided an elegant solution to the above problem using recursive common table expressions; a modified version is shown below.

```
CREATE TABLE products
(
  ID INTEGER PRIMARY KEY,
  Name VARCHAR2(20),
  Price NUMBER
);

INSERT INTO products VALUES (1,'PROD 1',100);
INSERT INTO products VALUES (2,'PROD 2',220);
INSERT INTO products VALUES (3,'PROD 3',15);
INSERT INTO products VALUES (4,'PROD 4',70);
INSERT INTO products VALUES (5,'PROD 5',150);

CREATE TABLE coupons
(
  ID INTEGER PRIMARY KEY,
  Name VARCHAR2(20),
  Value INTEGER,
  IsPercent CHAR(1)
);

INSERT INTO coupons VALUES (1,'CP 1 : -15$',15,'N');
INSERT INTO coupons VALUES (2,'CP 2 : -5$',5,'N');
INSERT INTO coupons VALUES (3,'CP 3 : -10%',10,'Y');
INSERT INTO coupons VALUES (4,'CP 4 : -12$',12,'N');

COLUMN Id FORMAT 99999 HEADING "Id"
COLUMN Name FORMAT a10 HEADING "Name"
COLUMN Price FORMAT 990.00 HEADING "Price"
COLUMN DiscountedPrice FORMAT 990.00 HEADING "Discounted|Price"
COLUMN DiscountAmount FORMAT 990.00 HEADING "Discount|Amount"
COLUMN DiscountRate FORMAT 990.00 HEADING "Discount|Rate"
COLUMN CouponNames FORMAT a30 HEADING "Coupon|Names"

SET linesize 250
SET pagesize 1000
SET sqlblanklines on

WITH

RCTE
(
  ID,
  Name,
  Price,
  DiscountedPrice,
  DiscountAmount,
  DiscountRate,
  CouponNames,
  CouponCount,
  CouponID
) AS

(

  -- The "anchor member" of the recursive CTE. It lists the undiscounted prices
  -- of each product.

  SELECT

    ID,
    Name,
    Price,
    Price AS DiscountedPrice,
    0 AS DiscountAmount,
    0 AS DiscountRate,
    CAST(' ' AS VARCHAR2(1024)) AS CouponNames,
    0 AS CouponCount,
    -1 AS CouponId

  FROM

    products

  UNION ALL

  -- The "recursive member" of the recursive CTE. It applies one additional
  -- coupon to data rows generated by a previous iteration while obeying the
  -- rules: not more than two coupons can be applied to the same product, the
  -- discounted price cannot be less than 70% of the original price, and the
  -- total amount of the discount cannot exceed $30.

  SELECT
  RCTE.ID,
  RCTE.NAME,
  RCTE.price,
  (
    DECODE
    (
      c.ispercent,
      'N', RCTE.discountedprice - c.Value,
      RCTE.discountedprice - (RCTE.discountedprice / 100 * c.Value)
    )
  ) AS discountedprice,
  (
    RCTE.price -
    DECODE
    (
      c.ispercent,
      'N', RCTE.discountedprice - c.Value,
      RCTE.discountedprice - (RCTE.discountedprice / 100 * c.Value)
    )
  ) AS discountamount,
  (
    RCTE.price -
    DECODE
    (
      c.ispercent,
      'N', RCTE.discountedprice - c.Value,
      RCTE.discountedprice - (RCTE.discountedprice / 100 * c.Value)
    )
  ) / RCTE.price * 100 AS discountrate,
  DECODE
  (
    RCTE.couponnames,
    ' ', c.Name,
    RCTE.couponnames || ' + ' || c.Name
  ) AS couponnames,
  RCTE.couponcount + 1 AS couponcount,
  c.ID AS couponid

FROM

  RCTE,
  coupons c

WHERE

  -- cannot reuse a coupon
  instr(RCTE.couponnames, c.Name) = 0

  -- not more than two coupons can be applied to the same product
  AND couponcount < 2

  -- the total amount of the discount can not exceed 30$
  AND
```

```
    (
      RCTE.price -
      DECODE
      (
        c.ispercent,
        'N', RCTE.discountedprice - c.Value,
        RCTE.discountedprice - (RCTE.discountedprice / 100 * c.Value)
      )
    ) <= 30

    -- the discounted price cannot be less than 70% of the original price
    AND
    (
      RCTE.price -
      DECODE
      (
        c.ispercent,
        'N', RCTE.discountedprice - c.Value,
        RCTE.discountedprice - (RCTE.discountedprice / 100 * c.Value)
      )
    ) / RCTE.price * 100 <= 30

),

-- sort the results in order of discounted price

SortedPrices As
(
  SELECT
    RCTE.*,
    ROW_NUMBER() OVER (PARTITION BY ID ORDER BY DiscountedPrice) AS RowNumber
  FROM RCTE
)

SELECT
  ID,
  Name,
  Price,
  DiscountedPrice,
  DiscountAmount,
  DiscountRate,
  CouponNames
FROM SortedPrices
WHERE RowNumber = 1
ORDER BY ID;

                          Discounted Discount Discount Coupon
    Id Name         Price      Price   Amount     Rate Names
------ ---------- ------- ---------- -------- -------- ------------------------------
     1 PROD 1      100.00      73.00    27.00    27.00 CP 1 : -15$ + CP 4 : -12$
     2 PROD 2      220.00     193.00    27.00    12.27 CP 1 : -15$ + CP 4 : -12$
     3 PROD 3       15.00      13.50     1.50    10.00 CP 3 : -10%
     4 PROD 4       70.00      49.50    20.50    29.29 CP 1 : -15$ + CP 3 : -10%
     5 PROD 5      150.00     120.00    30.00    20.00 CP 3 : -10% + CP 1 : -15$
```

## Sudoku

My final exhibit is a Sudoku puzzle. It turns out that a Sudoku puzzle can be elegantly solved with recursive common table expressions. The solution was discovered by Anton Scheffer. The listing below is a heavily annotated version of Anton Scheffer's solution with some cosmetic changes for better understandability.[1]

```
   -- The following SQL statement solves a Sudoku puzzle that is provided in the
   -- form of a one-dimensional array of 81 digits. Note that a Sudoku puzzle is
   -- really a 9x9 square grid. Here is how the positions in the one-dimensional
   -- array correspond to positions in the 9x9 grid.

   -- +---------+---------+---------+
   -- |  1  2  3|  4  5  6|  7  8  9|
   -- | 10 11 12| 13 14 15| 16 17 18|
   -- | 19 20 21| 22 23 24| 25 26 27|
   -- +---------+---------+---------+
   -- | 28 29 30| 31 32 33| 34 35 36|
   -- | 37 38 39| 40 41 42| 43 44 45|
   -- | 46 47 48| 49 50 51| 52 53 54|
   -- +---------+---------+---------+
   -- | 55 56 57| 58 59 60| 61 62 63|
   -- | 64 65 66| 67 68 69| 70 71 72|
   -- | 73 74 75| 76 77 78| 79 80 81|
   -- +---------+---------+---------+

   -- A recursive common table expression (CTE) is used to solve the puzzle. The
   -- "anchor member" of the recursive CTE contains the unsolved Sudoku puzzle. The
   -- "recursive member" generates partially solved Sudokus. Each iteration of the
   -- recursive member completes one blank cell. Recursive execution stops when no
   -- more blank cells are left or if no value can be found for a blank cell
   -- (meaning that the Sudoku has no solution). All solutions are produced if the
   -- puzzle has multiple solutions.

   -- Consider the following Sudoku puzzle: (All but the last 9 cells are filled.)

   -- 534678912672195348198342567859761423426853791713924856961537284287419635

   -- Here is the output produced for the above puzzle: (The output has been
   -- condensed in order to accomodate it within the available space.)

   -- Partially Solved Sudoku
   -- --------------------------------------------------------------------------
   -- 534678912672195348198342567859 ... 53791713924856961537284287419635
   -- 534678912672195348198342567859 ... 537917139248569615372842874196353
   -- 534678912672195348198342567859 ... 5379171392485696153728428741963534
   -- 534678912672195348198342567859 ... 53791713924856961537284287419635345
   -- 534678912672195348198342567859 ... 537917139248569615372842874196353452
   -- 534678912672195348198342567859 ... 5379171392485696153728428741963534528
   -- 534678912672195348198342567859 ... 53791713924856961537284287419635345286
   -- 534678912672195348198342567859 ... 537917139248569615372842874196353452861
   -- 534678912672195348198342567859 ... 5379171392485696153728428741963534528617
   -- 534678912672195348198342567859 ... 53791713924856961537284287419635345286179
```

[1] Anton Scheffer also has two solutions that work with Oracle Database 10*g* Release 2: a solution using the MODEL clause and a solution using collections.

```
SET sqlblanklines on
SET linesize 132
SET pagesize 66

WITH

-- The following number generator is itself an example of a recursive CTE. It
-- produces the consecutive digits from 1 to 9.

Numbers(n) AS
(

  -- The "anchor member." It contains exactly one row (N = 1).

  SELECT 1 AS N
  FROM dual

  UNION ALL

  -- The "recursive member." Each iteration of the recursive member produces the
  -- next value of N. Recursive execution stops when N = 9.

  SELECT N + 1 AS N
  FROM Numbers
  WHERE N < 9

),

RecursiveCTE(PartiallySolvedSudoku, BlankCell) AS
(
  -- The "anchor member" of the recursive CTE. It contains exactly one row: the
  -- unsolved Sudoku puzzle.

  SELECT
    cast(rpad('&&SudokuPuzzle', 81) AS VARCHAR2(81)) AS SudokuPuzzle,
    instr(rpad('&&SudokuPuzzle', 81), ' ', 1) AS FirstBlankCell
  FROM dual

  UNION ALL

  -- The "recursive member" of the recursive CTE. It generates intermediate
  -- solutions by providing values for the first blank cell in the intermediate
  -- solutions produced by the previous iteration. Recursive execution stops
  -- when no more blank cells are left or if none of the intermediate solutions
  -- generated by the previous iteration can be improved (meaning that the
  -- Sudoku puzzle has no solution). All solutions are generated if the puzzle
  -- has multiple solutions.

  SELECT

    -- Construct an intermediate solution by completing one blank cell.

    cast(
      substr(RecursiveCTE.PartiallySolvedSudoku, 1, BlankCell - 1)
      || to_char(Candidates.N)
      || substr(RecursiveCTE.PartiallySolvedSudoku, BlankCell + 1)
      AS VARCHAR2(81)
    ) AS PartiallySolvedSudoku,

    -- Locate the next blank cell, if any. Note that the result of instr is 0 if
    -- the string we are searching does not contain what we are looking for.

    instr(
      RecursiveCTE.PartiallySolvedSudoku,
      ' ',
      RecursiveCTE.BlankCell + 1
    ) AS NextBlankCell

  FROM

    -- The intermediate solutions from the previous iteration.
    RecursiveCTE,

    -- Consider all 9 candidate values from the Numbers table.
    Numbers Candidates

  WHERE NOT EXISTS

  -- Filter out candidate values that have already been used in the same row,
  -- the same column, or the same 3x3 grid as the blank cell. Note that a Sudoku
  -- puzzle is really a 9x9 grid but we are using a one-dimensional array of 81
  -- cells instead. Recursive execution will stop if none of the intermediate
  -- solutions generated by the previous iteration can be improved.

  -- The position within the one-dimensional array of the first cell in the same
  -- row of the 9x9 grid as the blank cell is trunc((BlankCell-1)/9)*9 + 1. The
  -- position of the Nth cell is obtained by adding N-1. For example, if
  -- BlankCell = 41, then the positions of the cells in the same row are 37,
  -- 37+1, 37+2, 37+3, 37+4, 37+5, 37+6, 37+7, and 37+8.

  -- The position of the first cell in the same column of the 9x9 grid as the
  -- blank cell is mod(BlankCell-1,9) + 1. The position of the Nth cell is
  -- obtained by adding 9*(N-1). For example, if BlankCell = 41, then the
  -- positions of the cells in the same column are 5, 5+9, 5+18, 5+27, 5+36,
  -- 5+45, 5+54, 5+63, and 5+72.

  -- The position of the first cell in the same 3x3 grid as the blank cell is
  -- trunc((BlankCell-1)/27)*27 + mod(trunc((BlankCell-1)/3),3)*3 + 1. The
  -- position of the Nth cell in the same 3x3 grid is obtained by adding (N-1) +
  -- trunc((N-1)/3)*6. For example, if BlankCell = 41, then the positions of the
  -- cells of in the same 3x3 grid are 31, 31+1, 31+2, 31+9, 31+10, 31+11,
  -- 31+18, 31+19, and 31+20.

  (
    SELECT N FROM Numbers

    WHERE to_char(Candidates.N) IN
    (
          -- Check the contents of the row containing the blank cell

          substr
          (
            RecursiveCTE.PartiallySolvedSudoku,
            trunc((BlankCell-1)/9)*9 + 1
              + (N-1),
            1
          ),

          -- Check the contents of the column containing the blank cell

          substr
          (
            RecursiveCTE.PartiallySolvedSudoku,
            mod(BlankCell-1,9) + 1
              + 9*(N-1),
            1
          ),

          -- Check the contents of the 3x3 grid containing the blank cell

          substr
          (
            RecursiveCTE.PartiallySolvedSudoku,
            trunc((BlankCell-1)/27)*27 + mod(trunc((BlankCell-1)/3),3)*3 + 1
              + (N-1) + trunc((N-1)/3)*6,
            1
          )
        )
      )

      -- Stop processing when no more blank cells are left.

      AND BlankCell > 0

    )

    SELECT PartiallySolvedSudoku "Partially Solved Sudoku"
    FROM RecursiveCTE;
```

## Concluding Remarks

At first glance, it might appear that recursive common table expressions in Oracle Database 11*g*R2 are little more than a verbose way of specifying what could be more succinctly achieved with the CONNECT BY clause. However—as shown by the examples in this paper—recursive common table expressions are significantly more powerful than the CONNECT BY clause.

## About The Author

Iggy Fernandez is an Oracle DBA with Database Specialists and has fifteen years of experience in Oracle database administration. He is the editor of the quarterly Journal of the Northern California Oracle Users Group (NoCOUG), the author of *Beginning Oracle Database 11*g *Administration* (Apress, 2009), and an Oracle ACE.