# Requêtes récursives avec la CTE
# Exemples avec SQL Server 2005

---

*Tout le monde à déjà eu affaire au moins une fois dans sa vie à la récursion. Lorsque j'étais enfant, mes parents et moi vivions dans un immeuble parisien ou figuraient dans le hall deux glaces se faisant face. Lorsque je passais entre ces deux miroirs, mon image se reflétait à l'infini et j'étais assez fier de palper le concept de récursion sur ma personne ! C'est cela la récursion : un processus capable de se reproduire aussi longtemps que nécessaire.*
*Mais en termes "mécaniques" nous ne pouvons accepter une récursion infinie. Dans le monde réel, nous avons besoin que le processus s'arrête parce que notre monde apparaît fermé. Woody Alen, parlant de l'infini du temps, disait "l'éternité c'est long, surtout vers la fin..." !*
*En informatique la récursion est une technique particulière, capable dans certains cas de traiter avec élégance des problèmes complexes : quelques lignes suffisent à effectuer un travail parfois considérable. Mais la récursion induit certains effets pervers : les ressources pour effectuer le traitement sont maximisées par le fait que chaque appel réentrant du processus nécessite l'ouverture d'un environnement de travail complet ce qui possède un coût généralement très élevé en mémoire. Heureusement, un mathématicien dont je ne me rappelle plus le nom, a découvert que tout processus récursif pouvait s'écrire de manière itérative, à condition de disposer d'une "pile"*

*Mais notre propos est de parler de la récursivité dans le langage de requête SQL et en particulier de ce que fait SQL Server 2005 au regard de la norme SQL:1999.*




Par Frédéric Brouard - MVP SQL Server
Expert SQL et SGBDR, Auteur de :

- SQL, Développement, Campus Press 2001
- SQL, collection Synthex, Pearson Education 2005, co écrit avec Christian Soutou
- http://sqlpro.developpez.com (site de ressources sur le langage SQL et les SGBDR)
- Enseigant aux Arts & Métiers et à l'ISEN Toulon

*Nota : le texte de cet article est paru dans SQL Server Magazine d'octobre 2005*


*Première parution originale :*
*http://www.sqlservercentral.com/columnists/fBROUARD/recursivequeriesinsql1999andsqlserver2005.asp*

# 1 Récursivité avec la norme SQL:1999

Voici une syntaxe normative édulcorée du concept de requête récursive :

```
WITH [ RECURSIVE ] <surnom_requête> [ ( <liste_colonne> ) ]
AS ( <requête_select> )
<requête_utilisant_surnom_requête>
```

Simple, n'est-ce pas ? En fait tout le mécanisme de récursivité est situé dans l'écriture de la <requête_select>. Nous allons d'abord montrer une version simplifiée, mais non récursive d'une telle requête et, lorsque nous aurons vu ce que nous pouvons faire avec le mot clef WITH, nous dévoilerons un aspect plus "sexy" de SQL, utilisant la récursivité..

# 2 Une simple expression de table (CTE : Common Table Expression)

L'utilisation du mot clef WITH, sans son complément RECURSIVE, permet de construire une expression de table dite "simple", soit en anglais "Common Table Expression" (CTE). En un sens, la CTE est une vue exprimée spécialement pour une requête et son usage exclusif et volatile. On peut donc parler de vue non persistante. L'utilisation classique du concept de CTE est de rendre plus clair l'écriture de requêtes complexes bâties à partir de résultats d'autres requêtes.
Voici un exemple basique :

```
-- creation de la table
CREATE TABLE T_NEWS
(NEW_ID        INTEGER NOT NULL PRIMARY KEY,
 NEW_FORUM     VARCHAR(16),
 NEW_QUESTION  VARCHAR(32))
GO
-- population de la table
INSERT INTO T_NEWS VALUES (1, 'SQL', 'What is SQL ?')
INSERT INTO T_NEWS VALUES (2, 'SQL', 'What do we do now ?')
INSERT INTO T_NEWS VALUES (3, 'Microsoft', 'Is SQL 2005 ready for use ?')
INSERT INTO T_NEWS VALUES (4, 'Microsoft', 'Did SQL2000 use RECURSION ?')
INSERT INTO T_NEWS VALUES (5, 'Microsoft', 'Where am I ?')

-- la requête exprimée de manière traditionnelle :
SELECT COUNT(NEW_ID) AS NEW_NBR, NEW_FORUM
FROM   T_NEWS
GROUP  BY NEW_FORUM
HAVING COUNT(NEW_ID) = ( SELECT MAX(NEW_NBR)
                         FROM ( SELECT COUNT(NEW_ID) AS NEW_NBR, NEW_FORUM
                                FROM   T_NEWS
                                GROUP  BY NEW_FORUM ) T )
-- le resultat :
NEW_NBR     NEW_FORUM
----------- ----------------
3           Microsoft
```

*Exemple 1*

Cette requête est assez classique dans le cadre d'un modèle de données de type "forum". Le but est de trouver la question qui a provoqué le plus de réponse. Pour exprimer une telle requête il faut faire un MAX(COUNT( ce qui n'est pas autorisé dans SQL du fait des groupages et doit donc être résolut par l'utilisation de sous requêtes. Mais notez que dans cette écriture, deux des SELECT présentent, à peu de choses près, la même structure :

```
SELECT
COUNT(NEW_ID) AS NEW_NBR, NEW_FORUM
FROM   T_NEWS
GROUP  BY NEW_FORUM
```

*Exemple 2*

L'utilisation du concept de CTE va rendre la requête plus lisible :

```
WITH
   Q_COUNT_NEWS (NBR, FORUM)
   AS
     (SELECT COUNT(NEW_ID), NEW_FORUM
      FROM   T_NEWS
```

```
      GROUP  BY NEW_FORUM)
SELECT NBR, FORUM
FROM   Q_COUNT_NEWS
WHERE  NBR = (SELECT MAX(NBR)
              FROM   Q_COUNT_NEWS)
```

*Exemple 3*

En fait nous utilisant la vue éphémère Q_COUNT_NEWS introduite par le mot clef WITH, pour écrire d'une manière plus élégante, la solution à notre problème.

Comme dans la cadre d'une vue SQL, vous devez nommer la CTE et vous pouvez donner des noms particuliers aux colonnes du SELECT qui construit l'expression de la CTE, mais cette dernière disposition n'est pas obligatoire.
Dans les faits on peut enchaîner deux, trois ou autant de CTE que vous voulez dans une même requête, chaque CTE pouvant être construite à partie des expression des CET précédentes. Voici un exemple de ce concept de CTE gigogne :

```
WITH
   Q_COUNT_NEWS (NBR, FORUM)
   AS
     (SELECT COUNT(NEW_ID), NEW_FORUM
      FROM   T_NEWS
      GROUP  BY NEW_FORUM),
   Q_MAX_COUNT_NEWS (NBR)
   AS (SELECT MAX(NBR)
       FROM   Q_COUNT_NEWS)
SELECT T1.*
FROM   Q_COUNT_NEWS T1
       INNER JOIN Q_MAX_COUNT_NEWS T2
             ON T1.NBR = T2.NBR
```

*Exemple 4*

Cette requête donne le même résultat que les précédentes : la première CTE, Q_COUNT_NEWS est utilisée comme s'il s'agissait d'une table dans la seconde CTE, Q_MAX_COUNT_NEWS. Ces deux CTE sont jointes dans l'expression de requête pour donner le résultat. Notez la virgule qui sépare les deux expression de table.

# 3 - Deux astuces pour la récursion

Pour rendre récursive une requête, deux astuces sont nécessaires :

Premièrement, vous devez donner un point de départ au processus de récursion. Cela doit se faire avec deux requêtes liées. La première requête indique ou l'on doit commencer et la seconde là où l'on doit se rendre ensuite. Ces deux requêtes doivent être jointes par l'opération ensembliste UNION ALL.

Deuxièmement, vous devez effectuer une corrélation entre l'expression de requête CTE et le code SQL qui la compose, afin de progresser par étapes successives. Cela se fait en référençant le <surnom_requête> à l'intérieur du code SQL exprimant la CTE

# 4 - Un premier exemple, une hiérarchie basique

Pour cet exemple, j'ai choisit une table contenant une typologie de véhicules :

```
-- creation de la table
CREATE TABLE T_VEHICULE
(VHC_ID         INTEGER NOT NULL PRIMARY KEY,
 VHC_ID_FATHER  INTEGER FOREIGN KEY REFERENCES T_VEHICULE (VHC_ID),
 VHC_NAME       VARCHAR(16))
-- population
INSERT INTO T_VEHICULE VALUES (1, NULL, 'ALL')
INSERT INTO T_VEHICULE VALUES (2, 1, 'SEA')
INSERT INTO T_VEHICULE VALUES (3, 1, 'EARTH')
INSERT INTO T_VEHICULE VALUES (4, 1, 'AIR')
INSERT INTO T_VEHICULE VALUES (5, 2, 'SUBMARINE')
INSERT INTO T_VEHICULE VALUES (6, 2, 'BOAT')
INSERT INTO T_VEHICULE VALUES (7, 3, 'CAR')
INSERT INTO T_VEHICULE VALUES (8, 3, 'TWO WHEELES')
INSERT INTO T_VEHICULE VALUES (9, 3, 'TRUCK')
INSERT INTO T_VEHICULE VALUES (10, 4, 'ROCKET')
INSERT INTO T_VEHICULE VALUES (11, 4, 'PLANE')
INSERT INTO T_VEHICULE VALUES (12, 8, 'MOTORCYCLE')
INSERT INTO T_VEHICULE VALUES (13, 8, 'BYCYCLE')
```

*Exemple 5*

Habituellement une hiérarchie se représente en utilisant une auto référence, c'est à dire à l'aide d'une clef étrangère provenant de la clef même de la table.
Les données de cette table peuvent se voir de la sorte :

```
ALL
|--SEA
|   |--SUBMARINE
|   |--BOAT
|--EARTH
|   |--CAR
|   |--TWO WHEELES
|   |   |--MOTORCYCLE
|   |   |--BYCYCLE
|   |--TRUCK
|--AIR
    |--ROCKET
    |--PLANE
```

Commençons maintenant à exprimer une première interrogation et demandons-nous d'où vient la moto ? En d'autres termes, nous recherchons les ancêtres de MOTORCYCLE. Nous devons donc partir de la ligne qui contient la moto :

```
SELECT VHC_NAME, VHC_ID_FATHER
FROM   T_VEHICULE
WHERE  VHC_NAME = 'MOTORCYCLE'
```

*Exemple 6*

Nous devons récupérer la valeur de l'identifiant père (VHC_ID_FATHER) pour aller à l'étape suivante.
La seconde requête, qui avance d'un pas dans l'arbre, doit être écrit comme suit :

```
SELECT VHC_NAME, VHC_ID_FATHER
FROM   T_VEHICULE
```

*Exemple 7*

Comme on le voit, il n'y a aucune différence entre les deux requêtes à l'exception du filtre WHERE qui sert de point de départ. Souvenez-vous simplement que vous devez introduire un UNION ALL entre les deux requêtes afin d'assurer la liaison entre le départ et le pas suivant :

```
SELECT VHC_NAME, VHC_ID_FATHER
FROM   T_VEHICULE
WHERE  VHC_NAME = 'MOTORCYCLE'
UNION ALL
SELECT VHC_NAME, VHC_ID_FATHER
FROM   T_VEHICULE
```

*Exemple 8*

Plaçons maintenant cette requête composite dans code SQL qui bâtie l'expression de la CTE :

```
WITH
   tree (data, id)
   AS (SELECT VHC_NAME, VHC_ID_FATHER
       FROM   T_VEHICULE
       WHERE  VHC_NAME = 'MOTORCYCLE'
       UNION ALL
       SELECT VHC_NAME, VHC_ID_FATHER
       FROM   T_VEHICULE)
```

*Exemple 9*

Nous sommes maintenant près de la récursion. La dernière étape de notre travail consiste à générer le cycle de récursion. Cela se fait en utilisant le nom de la CTE (ici "tree") à l'intérieur même de l'expression. Dans notre cas nous devons joindre à la seconde requête SELECT de l'expression CTE, la "table" tree à la table T_VEHICULE et assurer le chaînage par une jointure entre les identifiants : tree.id = (second query).VHC_ID.
Cela se réalise ainsi :

```
WITH
   tree (data, id)
   AS (SELECT VHC_NAME, VHC_ID_FATHER
       FROM   T_VEHICULE
       WHERE  VHC_NAME = 'MOTORCYCLE'
       UNION ALL
       SELECT VHC_NAME, VHC_ID_FATHER
       FROM   T_VEHICULE V
              INNER JOIN tree t
                    ON t.id = V.VHC_ID)
SELECT *
FROM   tree
```

*Exemple 10*

Il n'y a plus rien à faire que d'exprimer une requête finale, la plus simple possible, basée sur la CTE afin d'afficher les données :

```
data            id
--------------- -----------
MOTORCYCLE      8
TWO WHEELES     3
EARTH           1
ALL             NULL
```

Regardons maintenant la façon dont nous avons lié les tables et la CTE d'une façon pseudo graphique :

```
correlation
        _____________________________________
       |                                     |
       v                                     |
WITH tree (data, id)                         |
AS (SELECT VHC_NAME, VHC_ID_FATHER           |
    FROM   T_VEHICULE                        |
    WHERE  VHC_NAME = 'MOTORCYCLE'           |
    UNION ALL                                |
    SELECT VHC_NAME, VHC_ID_FATHER           |
    FROM   T_VEHICULE V                      |
           INNER JOIN tree t <-----------
                 ON t.id = V.VHC_ID)
SELECT *
FROM   tree
```

La question que l'on peut se poser est la suivante : qu'est ce qui a stoppé le processus de récursion ? C'est simplement le fait que plus rien ne peut être chaîné dès que l'identifiant atteint le marqueur NULL, ce qui est le cas dans notre exemple lorsque nous atteignons la colonne ou VHC_NAME = "ALL".

Maintenant vous avez la technique. Veuillez simplement noter que pour une raison que je ne m'explique pas encore, MS SQL Server n'accepte pas la présence du mot clef RECURSIVE suivant le mot clef WITH, alors que la norme le préconise...

# 5 - Indentation hiérarchique

Une chose importante et souvent réclamé avec les données structurées sous forme arborescentes, est de les voir à la manière d'un arbre... ce qui suppose une indentation des items combinée à un ordre particulier, lors de la restitution des données. Est-ce possible avec SQL ? Oui, bien sûr. Pour réaliser cet ordonnancement des données, nous devons connaître le cheminement dans l'arbre et le niveau du noeud, deux informations qui nous aiderons à rajouter des espaces d'indentation et à trier les lignes du résultat dans le bon ordre.
Il faut donc calculer à la fois le chemin et le niveau, et cela est possible avec la CTE :

```
WITH tree (data, id, level, pathstr)
AS (SELECT VHC_NAME, VHC_ID, 0,
           CAST('' AS VARCHAR(MAX))
    FROM   T_VEHICULE
    WHERE  VHC_ID_FATHER IS NULL
    UNION ALL
```

```
    SELECT VHC_NAME, VHC_ID, t.level + 1, t.pathstr + V.VHC_NAME
    FROM   T_VEHICULE V
           INNER JOIN tree t
                 ON t.id = V.VHC_ID_FATHER)
SELECT SPACE(level) + data as data, id, level, pathstr
FROM   tree
ORDER  BY pathstr, id

data                id          level       pathstr
------------------- ----------- ----------- --------------------------------
ALL                 1           0
 AIR                4           1           AIR
  PLANE             11          2           AIRPLANE
  ROCKET            10          2           AIRROCKET
 EARTH              3           1           EARTH
  CAR               7           2           EARTHCAR
  TRUCK             9           2           EARTHTRUCK
  TWO WHEELES       8           2           EARTHTWO WHEELES
   BYCYCLE          13          3           EARTHTWO WHEELESBYCYCLE
   MOTORCYCLE       12          3           EARTHTWO WHEELESMOTORCYCLE
 SEA                2           1           SEA
  BOAT              6           2           SEABOAT
  SUBMARINE         5           2           SEASUBMARINE
```

*Exemple 11*

Pour réaliser ce tour de force, nous avons utilisé un nouveau type de données introduit avec la version 2005 de SQL Server, le type VARCHAR(max) afin de ne pas limiter la concaténation des points de passage à quelques caractères. En effet, la profondeur d'un arbre peut être importante et dans ce cas la concaténation du nom des étapes peut conduire à saturer une colonne traditionnellement limitée à quelques caractères.
De plus et afin d'éliminer certains effets de bord, nous vous conseillons d'introduire un marqueur entre les noms des différents noeuds, par exemple le caractère "virgule" suivi d'un espace afin de lier les étapes. Cela empêchera de confondre une étape comme MARSALES (24) avec la concaténation de deux autres étapes comme ALES (30) et MARS (07).

# 6 - Arbres SQL sans récursion

Mais je dois dire que les représentations hiérarchiques par auto référence ne sont pas d'intéressants sujets pour la récursion... Pourquoi ? Parce qu'il existe une possibilité de structuration des données qui élimine tout traitement récursif !
Souvenez-vous de ce que j'ai dit dans le chapeau de cet article : *vous pouvez vous passer de la récursion à condition de disposer d'une pile*. Est-ce possible ? Oui : il suffit de rajouter la pile à la table... Comment ? En utilisant deux nouvelles colonnes RIGHT_BOUND et LEFT_BOUND...

```
ALTER TABLE T_VEHICULE ADD RIGHT_BOUND INTEGER
ALTER TABLE T_VEHICULE ADD LEFT_BOUND INTEGER
```

*Exemple 12*

Maintenant, à la manière d'un magicien, je vais ajouter des données a ces deux nouvelles colonnes. Des chiffres astucieusement calculés pour rendre inutile le recours à la récursivité pour toutes nos interrogations :

```
UPDATE T_VEHICULE SET LEFT_BOUND = 1 , RIGHT_BOUND = 26 WHERE VHC_ID = 1
UPDATE T_VEHICULE SET LEFT_BOUND = 2 , RIGHT_BOUND = 7  WHERE VHC_ID = 2
UPDATE T_VEHICULE SET LEFT_BOUND = 8 , RIGHT_BOUND = 19 WHERE VHC_ID = 3
UPDATE T_VEHICULE SET LEFT_BOUND = 20, RIGHT_BOUND = 25 WHERE VHC_ID = 4
UPDATE T_VEHICULE SET LEFT_BOUND = 3 , RIGHT_BOUND = 4  WHERE VHC_ID = 5
UPDATE T_VEHICULE SET LEFT_BOUND = 5 , RIGHT_BOUND = 6  WHERE VHC_ID = 6
UPDATE T_VEHICULE SET LEFT_BOUND = 9 , RIGHT_BOUND = 10 WHERE VHC_ID = 7
UPDATE T_VEHICULE SET LEFT_BOUND = 11, RIGHT_BOUND = 16 WHERE VHC_ID = 8
UPDATE T_VEHICULE SET LEFT_BOUND = 17, RIGHT_BOUND = 18 WHERE VHC_ID = 9
UPDATE T_VEHICULE SET LEFT_BOUND = 21, RIGHT_BOUND = 22 WHERE VHC_ID = 10
UPDATE T_VEHICULE SET LEFT_BOUND = 23, RIGHT_BOUND = 24 WHERE VHC_ID = 11
UPDATE T_VEHICULE SET LEFT_BOUND = 12, RIGHT_BOUND = 13 WHERE VHC_ID = 12
UPDATE T_VEHICULE SET LEFT_BOUND = 14, RIGHT_BOUND = 15 WHERE VHC_ID = 13
```

*Exemple 13*

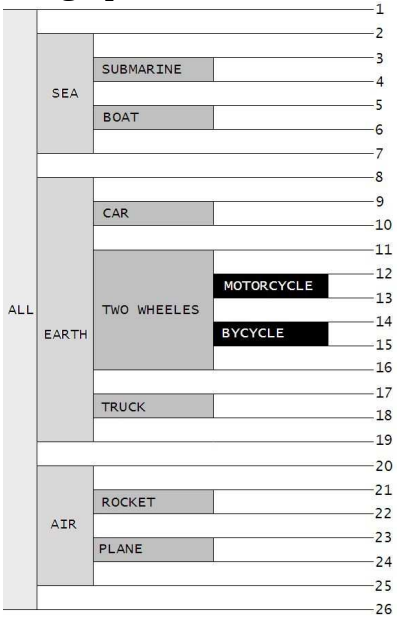
Et voici maintenant la requête "magique" qui donne le même résultat que notre précédente requête exprimée sous la forme complexe de la CTE récursive :

```
SELECT *
FROM   T_VEHICULE
WHERE  RIGHT_BOUND > 12
  AND  LEFT_BOUND < 13

VHC_ID      VHC_ID_FATHER VHC_NAME          RIGHT_BOUND LEFT_BOUND
----------- ------------- ----------------- ----------- -----------
1           NULL          ALL               26          1
3           1             EARTH             19          8
8           3             TWO WHEELES       16          11
12          8             MOTORCYCLE        13          12
```

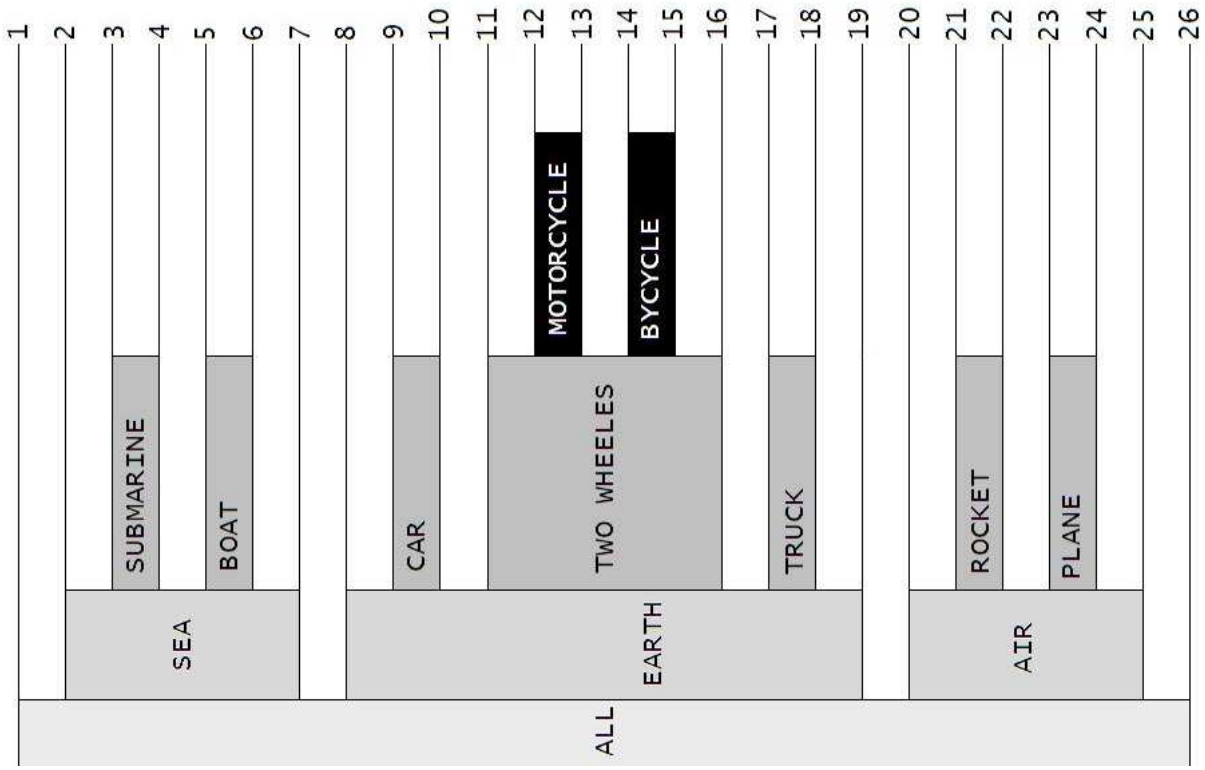
*Exemple 14*

La question est maintenant la suivante : quel est le truc ? En fait, nous avons réalisé la pile en numérotant les données par tranches. Comme rien ne vaut une image pour visualiser un tel concept, voici ce que j'ai dessiné :




La seule chose que j'ai fait, c'est de numéroter continuellement en commençant par 1 toutes les bornes droites et gauches des empilements de données constitués par chacune de nos données.
Afin d'obtenir la requête précédente, j'ai simplement pris les bornes de MOTORCYCLE, c'est à dire 12 gauche et 13 droite afin de les placer dans la clause WHERE et demandé d'extraire les lignes de la table pour lesquelles les valeurs des colonnes RIGHT BOUND étaient supérieures à 12 et LEFT BOUND inférieures à 13.

Notez d'ailleurs que mon joli dessin serait plus compréhensible si je l'avais fait pivoter de 90° :




J'espère ainsi que vous voyez mieux les piles ! Cette représentations est d'ailleurs connue dans la littérature spécialisée sous le nom de "représentation intervallaire des arborescences", en particulier dans le livre de Joe Celko "SQL Avancé" (Vuibert éditeur) ainsi que sur mon site web SQLpro sur lequel vous trouverez en outre toutes les requêtes et les procédures stockées MS SQL Server adéquates pour faire fonctionner un tel arbre (http://sqlpro.developpez.com/cours/arborescence/).

Dernière question sur ce modèle : pouvons nous reproduire l'indentation hiérarchique présentée dans la dernière requête construite avec la CTE ? Oui bien sûr, et cela d'autant plus facilement si l'on ajoute une nouvelle colonne indiquant le niveau du noeud. C'est simple à calculer puisque le niveau d'un noeud est celui du noeud de rattachement +1 si l'insertion se fait en fils, ou encore le même si l'insertion se fait en frère, et qu'a la première insertion, donc à la racine de l'arbre, le niveau est 0.
Voici maintenant les ordres SQL modifiant notre table pour assurer cette fonction :

```
ALTER TABLE T_VEHICULE
ADD LEVEL INTEGER

UPDATE T_VEHICULE SET LEVEL = 0 WHERE VHC_ID = 1
UPDATE T_VEHICULE SET LEVEL = 1 WHERE VHC_ID = 2
UPDATE T_VEHICULE SET LEVEL = 1 WHERE VHC_ID = 3
UPDATE T_VEHICULE SET LEVEL = 1 WHERE VHC_ID = 4
UPDATE T_VEHICULE SET LEVEL = 2 WHERE VHC_ID = 5
UPDATE T_VEHICULE SET LEVEL = 2 WHERE VHC_ID = 6
UPDATE T_VEHICULE SET LEVEL = 2 WHERE VHC_ID = 7
UPDATE T_VEHICULE SET LEVEL = 2 WHERE VHC_ID = 8
```

```
UPDATE T_VEHICULE SET LEVEL = 2 WHERE VHC_ID = 9
UPDATE T_VEHICULE SET LEVEL = 2 WHERE VHC_ID = 10
UPDATE T_VEHICULE SET LEVEL = 2 WHERE VHC_ID = 11
UPDATE T_VEHICULE SET LEVEL = 3 WHERE VHC_ID = 12
UPDATE T_VEHICULE SET LEVEL = 3 WHERE VHC_ID = 13
```

*Exemple 15*

Voici la requête présentant les données sous forme d'une arborescence avec indentation hiérarchique :

```
SELECT SPACE(LEVEL) + VHC_NAME as data
FROM   T_VEHICULE
ORDER  BY LEFT_BOUND

data
----------------------
ALL
 SEA
  SUBMARINE
  BOAT
 EARTH
  CAR
  TWO WHEELES
   MOTORCYCLE
   BYCYCLE
  TRUCK
 AIR
  ROCKET
  PLANE
```

*Exemple 16*

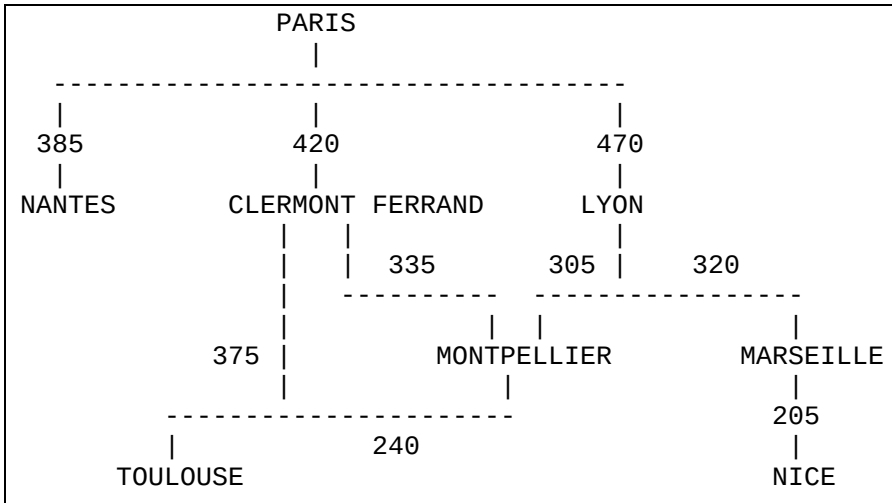
Beaucoup plus simples n'est-ce pas ?

## PREMIÉRES IMPRESSIONS...

La seule chose à dire au sujet de ces deux techniques de navigation dans les arbres est que le modèle par intervalle est beaucoup plus performant pour l'extraction des données que l'utilisation de l'expression de table récursive (CTE) introduite avec la norme SQL:1999.
En fait la récursivité dans SQL n'est pas si intéressante que cela dans ce cadre... Mais qu'en est-il dans un autre ? C'est ce que nous allons voir !

## 7 - Second exemple : un réseau complexe (et des requêtes plus *sexy* !)

Peut être ne voyagez-vous pas assez souvent en France. En tout cas ce que je peux vous dire, c'est qu'à Paris il y a des jolies filles et à Toulouse un délicieux plat appelé cassoulet et un petit constructeur d'avion de nom francisé "bus de l'air"...
Plus sérieusement, notre problème consiste à nous rendre de paris à Toulouse en utilisant le réseau autoroutier :




```
-- creation de la table :
CREATE TABLE T_JOURNEY
(JNY_FROM_TOWN  VARCHAR(32),
 JNY_TO_TOWN    VARCHAR(32),
 JNY_MILES      INTEGER)
-- population  :
INSERT INTO T_JOURNEY VALUES ('PARIS', 'NANTES', 385)
INSERT INTO T_JOURNEY VALUES ('PARIS', 'CLERMONT-FERRAND', 420)
INSERT INTO T_JOURNEY VALUES ('PARIS', 'LYON', 470)
INSERT INTO T_JOURNEY VALUES ('CLERMONT-FERRAND', 'MONTPELLIER', 335)
INSERT INTO T_JOURNEY VALUES ('CLERMONT-FERRAND', 'TOULOUSE', 375)
INSERT INTO T_JOURNEY VALUES ('LYON', 'MONTPELLIER', 305)
INSERT INTO T_JOURNEY VALUES ('LYON', 'MARSEILLE', 320)
INSERT INTO T_JOURNEY VALUES ('MONTPELLIER', 'TOULOUSE', 240)
INSERT INTO T_JOURNEY VALUES ('MARSEILLE', 'NICE', 205)
```

*Exemple 17*

Essayons maintenant une simple requête donnant tous les trajets entre deux villes :

```
WITH journey (TO_TOWN)
AS
   (SELECT DISTINCT JNY_FROM_TOWN
    FROM   T_JOURNEY
    UNION  ALL
    SELECT JNY_TO_TOWN
    FROM   T_JOURNEY AS arrival
           INNER JOIN journey AS departure
                 ON departure.TO_TOWN = arrival.JNY_FROM_TOWN)
SELECT *
FROM   journey

TO_TOWN
------------------------------
CLERMONT-FERRAND
LYON
MARSEILLE
MONTPELLIER
PARIS
NANTES
CLERMONT-FERRAND
LYON
MONTPELLIER
```

```
MARSEILLE
NICE
TOULOUSE
MONTPELLIER
TOULOUSE
TOULOUSE
TOULOUSE
NICE
MONTPELLIER
MARSEILLE
NICE
TOULOUSE
MONTPELLIER
TOULOUSE
TOULOUSE
```

*Exemple 18*

Constatez avec moi que ce n'est pas très intéressant car nous ne savons pas d'où nous venons. Seul la destination figure dans la réponse et il est probable que plusieurs trajets figurent pour un même voyage.. Pouvons nous avoir plus d'informations ?

Premièrement considérons que nous devons partir de Paris :

```
WITH journey (TO_TOWN)
AS
   (SELECT DISTINCT JNY_FROM_TOWN
    FROM   T_JOURNEY
    WHERE  JNY_FROM_TOWN = 'PARIS'
    UNION  ALL
    SELECT JNY_TO_TOWN
    FROM   T_JOURNEY AS arrival
           INNER JOIN journey AS departure
                 ON departure.TO_TOWN = arrival.JNY_FROM_TOWN)
SELECT *
FROM   journey

TO_TOWN
------------------------------
PARIS
NANTES
CLERMONT-FERRAND
LYON
MONTPELLIER
MARSEILLE
NICE
TOULOUSE
MONTPELLIER
TOULOUSE
TOULOUSE
```

*Exemple 19*

Nous avons probablement trois trajets différents pour aller à Toulouse. Pouvons nous filtrer la destination ? Bien sûr :

```
WITH journey (TO_TOWN)
AS
   (SELECT DISTINCT JNY_FROM_TOWN
    FROM   T_JOURNEY
```

```
    WHERE  JNY_FROM_TOWN = 'PARIS'
    UNION  ALL
    SELECT JNY_TO_TOWN
    FROM   T_JOURNEY AS arrival
           INNER JOIN journey AS departure
                 ON departure.TO_TOWN = arrival.JNY_FROM_TOWN)
SELECT *
FROM   journey
WHERE  TO_TOWN = 'TOULOUSE'

TO_TOWN
------------------------------
TOULOUSE
TOULOUSE
TOULOUSE
```

*Exemple 20*

Nous pouvons affiner cette requête afin qu'elle nous donne le nombre d'étapes des différents trajets, entre l'origine et la destination :

```
WITH journey (TO_TOWN, STEPS)
AS
   (SELECT DISTINCT JNY_FROM_TOWN, 0
    FROM   T_JOURNEY
    WHERE  JNY_FROM_TOWN = 'PARIS'
    UNION  ALL
    SELECT JNY_TO_TOWN, departure.STEPS + 1
    FROM   T_JOURNEY AS arrival
           INNER JOIN journey AS departure
                 ON departure.TO_TOWN = arrival.JNY_FROM_TOWN)
SELECT *
FROM   journey
WHERE  TO_TOWN = 'TOULOUSE'

TO_TOWN                        STEPS
------------------------------ -----------
TOULOUSE                       3
TOULOUSE                       2
TOULOUSE                       3
```

*Exemple 21*

La cerise sur le gâteau serait de connaître la distance des différents trajets. Cela s'exprime comme ceci :

```
WITH journey (TO_TOWN, STEPS, DISTANCE)
AS
   (SELECT DISTINCT JNY_FROM_TOWN, 0, 0
    FROM   T_JOURNEY
    WHERE  JNY_FROM_TOWN = 'PARIS'
    UNION  ALL
    SELECT JNY_TO_TOWN, departure.STEPS + 1
           , departure.DISTANCE + arrival.JNY_MILES
    FROM   T_JOURNEY AS arrival
           INNER JOIN journey AS departure
                 ON departure.TO_TOWN = arrival.JNY_FROM_TOWN)
SELECT *
FROM   journey
WHERE  TO_TOWN = 'TOULOUSE'

TO_TOWN                  STEPS       DISTANCE
```

```
---------------------- ----------- -----------
TOULOUSE               3           1015
TOULOUSE               2           795
TOULOUSE               3           995
```

*Exemple 22*

La fille qui surgit du gâteau consisterait à afficher les différentes étapes intermédiaires de chaque trajet :

```
WITH journey (TO_TOWN, STEPS, DISTANCE, WAY)
AS
   (SELECT DISTINCT JNY_FROM_TOWN, 0, 0, CAST('PARIS' AS VARCHAR(MAX))
    FROM   T_JOURNEY
    WHERE  JNY_FROM_TOWN = 'PARIS'
    UNION  ALL
    SELECT JNY_TO_TOWN, departure.STEPS + 1
           , departure.DISTANCE + arrival.JNY_MILES
           , departure.WAY + ', ' + arrival.JNY_TO_TOWN
    FROM   T_JOURNEY AS arrival
           INNER JOIN journey AS departure
                 ON departure.TO_TOWN = arrival.JNY_FROM_TOWN)
SELECT *
FROM   journey
WHERE  TO_TOWN = 'TOULOUSE'

TO_TOWN    STEPS   DISTANCE  WAY
---------- ------- --------- --------------------------------------------
TOULOUSE   3       1015      PARIS, LYON, MONTPELLIER, TOULOUSE
TOULOUSE   2       795       PARIS, CLERMONT-FERRAND, TOULOUSE
TOULOUSE   3       995       PARIS, CLERMONT-FERRAND, MONTPELLIER, TOULOUSE
```

*Exemple 23*

Et maintenant, mesdames et messieurs, la technique de la CTE alliée à la puissance de la récursivité SQL sont fiers de vous présenter la solution à un problème de grande complexité, le problème du voyageur de commerce, un des casses têtes de la recherche opérationnelle sur lequel le mathématicien Edsger Wybe Dijkstra trouva un algorithme et lui valu le prix Turing en 1972... :

```
WITH journey (TO_TOWN, STEPS, DISTANCE, WAY)
AS
   (SELECT DISTINCT JNY_FROM_TOWN, 0, 0, CAST('PARIS' AS VARCHAR(MAX))
    FROM   T_JOURNEY
    WHERE  JNY_FROM_TOWN = 'PARIS'
    UNION  ALL
    SELECT JNY_TO_TOWN, departure.STEPS + 1,
           departure.DISTANCE + arrival.JNY_MILES,
           departure.WAY + ', ' + arrival.JNY_TO_TOWN
    FROM   T_JOURNEY AS arrival
           INNER JOIN journey AS departure
                 ON departure.TO_TOWN = arrival.JNY_FROM_TOWN)
SELECT TOP 1 *
FROM   journey
WHERE  TO_TOWN = 'TOULOUSE'
ORDER  BY DISTANCE

TO_TOWN    STEPS   DISTANCE  WAY
---------- ------- --------- --------------------------------------------
TOULOUSE   2       795       PARIS, CLERMONT-FERRAND, TOULOUSE
```

*Exemple 24*

Au fait, TOP n ne fait pas partie de la norme SQL... haïssez-le et bénissez la CTE :

```
WITH
   journey (TO_TOWN, STEPS, DISTANCE, WAY)
   AS
      (SELECT DISTINCT JNY_FROM_TOWN, 0, 0, CAST('PARIS' AS VARCHAR(MAX))
       FROM   T_JOURNEY
       WHERE  JNY_FROM_TOWN = 'PARIS'
       UNION  ALL
       SELECT JNY_TO_TOWN, departure.STEPS + 1,
              departure.DISTANCE + arrival.JNY_MILES,
              departure.WAY + ', ' + arrival.JNY_TO_TOWN
       FROM   T_JOURNEY AS arrival
              INNER JOIN journey AS departure
                    ON departure.TO_TOWN = arrival.JNY_FROM_TOWN),
   short (DISTANCE)
   AS
      (SELECT MIN(DISTANCE)
       FROM   journey
       WHERE  TO_TOWN = 'TOULOUSE')
SELECT *
FROM   journey j
       INNER JOIN short s
             ON j.DISTANCE = s.DISTANCE
WHERE  TO_TOWN = 'TOULOUSE'
```

*Exemple 25*

# 8 - Que faire de plus ?

En fait, une chose qui a limité notre processus de recherche dans notre réseau autoroutier, c'est que nous avons inséré les routes en sens unique. Je veux dire par-là que nos données permettent d'aller de Paris à Lyon, mais pas de Lyon à paris. Pour cela nous devons ajouter les routes inverses :

```
JNY_FROM_TOWN  JNY_TO_TOWN  JNY_MILES
-------------- ------------ ---------
LYON           PARIS        470
```

Cela peu être fait par une requête on ne peut plus simple :

```
INSERT INTO T_JOURNEY
SELECT JNY_TO_TOWN, JNY_FROM_TOWN, JNY_MILES
FROM T_JOURNEY
```

*Exemple 26*

Mais dès lors nos requêtes précédentes sont prises en défaut :

```
WITH journey (TO_TOWN)
AS
   (SELECT DISTINCT JNY_FROM_TOWN
    FROM   T_JOURNEY
    WHERE  JNY_FROM_TOWN = 'PARIS'
    UNION  ALL
    SELECT JNY_TO_TOWN
    FROM   T_JOURNEY AS arrival
           INNER JOIN journey AS departure
                 ON departure.TO_TOWN = arrival.JNY_FROM_TOWN)
SELECT *
```

```
FROM   journey

TO_TOWN
-------------------------------
PARIS
NANTES
CLERMONT-FERRAND
LYON
....
LYON
MONTPELLIER
MARSEILLE
PARIS

Msg 530, Level 16, State 1, Line 1
The statement terminated. The maximum recursion 100 has been exhausted before
statement completion.
```

*Exemple 27*

Que s'est-il passé ? Très simplement, le système essaye toutes les routes, y compris les trajets "ping-pong" comme Paris, Lyon, Paris, Lyon, Paris... *ad infinitum*... Est-il possible de se débarrasser des trajets "cycliques" ? Sans doute. Dans l'une de nos précédentes requête nous avons utilisé une colonne donnant la liste des étapes du trajet. Pourquoi ne pas l'utiliser pour empêcher un cycle de se produire ? La condition serait : ne pas passer par une ville dont le nom se trouve déjà dans la liste de ville du chemin (WAY). Ce qui peut s'écrire comme ceci :

```
WITH journey (TO_TOWN, STEPS, DISTANCE, WAY)
AS
   (SELECT DISTINCT JNY_FROM_TOWN, 0, 0, CAST('PARIS' AS VARCHAR(MAX))
    FROM   T_JOURNEY
    WHERE  JNY_FROM_TOWN = 'PARIS'
    UNION  ALL
    SELECT JNY_TO_TOWN, departure.STEPS + 1,
           departure.DISTANCE + arrival.JNY_MILES,
           departure.WAY + ', ' + arrival.JNY_TO_TOWN
    FROM   T_JOURNEY AS arrival
           INNER JOIN journey AS departure
                 ON departure.TO_TOWN = arrival.JNY_FROM_TOWN
    WHERE  departure.WAY NOT LIKE '%' + arrival.JNY_TO_TOWN + '%')
SELECT *
FROM   journey
WHERE  TO_TOWN = 'TOULOUSE'

TO_TOWN  STEPS DISTANCE WAY
-------- ----- -------- ----------------------------------------------------
TOULOUSE 3     1015     PARIS, LYON, MONTPELLIER, TOULOUSE
TOULOUSE 4     1485     PARIS, LYON, MONTPELLIER, CLERMONT-FERRAND, TOULOUSE
TOULOUSE 2     795      PARIS, CLERMONT-FERRAND, TOULOUSE
TOULOUSE 3     995      PARIS, CLERMONT-FERRAND, MONTPELLIER, TOULOUSE
```

*Exemple 28*

Comme vous pouvez le constater, une nouvelle route apparaît. C'est la plus longue, mais peut être la plus belle !

# CONCLUSIONS

Avec les CTE et donc la possibilité d'écrire des requêtes récursives, le langage SQL devient un langage complet capable de traiter en une seule requête le problème le plus complexe qui soit.

L'expression de table CTE peut simplifier l'écriture de requêtes complexes. Les requêtes récursives ne doivent être employées que lorsque la récursivité apparaît comme la seule solution. Si vous faites une erreur dans l'écriture de votre requête récursive, n'ayez pas peur, par défaut le nombre de cycles de récursion est limité à 100. Vous pouvez outrepasser cette limite en fixant vous-même la valeur à l'aide de la clause OPTION (MAXRECURSION n), qui doit figurer en dernier dans la requête.

Enfin, sachez que la norme SQL:1999 propose des compléments syntaxiques pour piloter votre SQL récursif. Par exemple vous pouvez naviguer dans les données DEPTH FIRST ou BREADTH FIRST (en profondeur ou en largeur en premier lieu) et aussi constituer une colonne contenant toutes les données des étapes intermédiaires dans un tableau de ligne (ARRAY of ROW) dont la taille doit être "suffisante" pour couvrir tous les cas de figure.
Voici la syntaxe complète de la CTE avec récursivité :

```
WITH [ RECURSIVE ]  [ ( <liste_colonne> ) ]
  AS ( <requete_select> )
[ <clause_cycle_recherche> ]

avec :
<clause_cycle_recherche> ::=
   <clause_recherche>
   | <clause_cycle>
   | <clause_recherche> <clause_cycle>

et :
   <clause_recherche> ::=
      SEARCH { DEPTH FIRTS BY
             | BREADTH FIRST BY  } <liste_specification_ordre>
      SET <colonne_sequence>

   <clause_cycle> ::=
      CYCLE <colonne_cycle1> [ { , <colonne_cycle2> } ... ]
      SET <colonne_marquage_cycle>
          TO <valeur_marque_cycle>
          DEFAULT <valeur_marque_non_cycle>
      USING <colonne_chemin>
```

# En bonus (CTE, requête récursive appliquée)

Voici une procédure stockée qui donne l'ordre exact des DELETE à effectuer dans des tables afin de vider une table de ses données, lorsque cette dernière est liée par l'intégrité référentielle :

```
CREATE PROCEDURE dbo.P_WHAT_TO_DELETE_BEFORE
     @TABLE_TO_DELETE VARCHAR(128),  -- targettable to delete
     @DB              VARCHAR(128),  -- target database
     @USR             VARCHAR(128)   -- target schema (dbo in most cases)
AS

WITH

T_CONTRAINTES (table_name, father_table_name)
AS (SELECT DISTINCT CTU.TABLE_NAME, TCT.TABLE_NAME
    FROM   INFORMATION_SCHEMA.REFERENTIAL_CONSTRAINTS RFC
           INNER JOIN INFORMATION_SCHEMA.CONSTRAINT_TABLE_USAGE CTU
                 ON RFC.CONSTRAINT_CATALOG    = CTU.CONSTRAINT_CATALOG
                    AND RFC.CONSTRAINT_SCHEMA = CTU.CONSTRAINT_SCHEMA
                    AND RFC.CONSTRAINT_NAME   = CTU.CONSTRAINT_NAME
           INNER JOIN INFORMATION_SCHEMA.TABLE_CONSTRAINTS TCT
                 ON RFC.UNIQUE_CONSTRAINT_CATALOG    = TCT.CONSTRAINT_CATALOG
                    AND RFC.UNIQUE_CONSTRAINT_SCHEMA = TCT.CONSTRAINT_SCHEMA
                    AND RFC.UNIQUE_CONSTRAINT_NAME   = TCT.CONSTRAINT_NAME
    WHERE  CTU.TABLE_CATALOG = @DB
      AND  CTU.TABLE_SCHEMA  = @USR),

T_TREE_CONTRAINTES (table_to_delete, level)
AS (SELECT DISTINCT table_name, 0
    FROM   T_CONTRAINTES
    WHERE  father_table_name = @TABLE_TO_DELETE
    UNION  ALL
    SELECT priorT.table_name, level - 1
    FROM   T_CONTRAINTES priorT
           INNER JOIN T_TREE_CONTRAINTES beginT
                 ON beginT.table_to_delete = priorT.father_table_name
    WHERE  priorT.father_table_name <> priorT.table_name)

SELECT DISTINCT *
FROM   T_TREE_CONTRAINTES
ORDER  BY level

GO
```

*Exemple 29*

Le cas d'auto référence est en principe intégré. Les paramètres sont :
@DB : nom de la base,
@USR : nom du schema, par défaut dbo,
@TABLE_TO_DELETE : nom de la table que vous voulez vider

# UN PARI

A ce stade, les possibilités de la norme SQL:1999 comme celles offertes par MS SQL Server 2005 sont-elles capable de résoudre tout problème bien modélisé en une seule requête ? Je fais le pari que oui !

# ADDENDUM

*(3 novembre 2007)*

Voici quelques points supplémentaires très précieux pour illustrer la récursivité et la puissance tant de la CTE que du mode intervallaire pour la gestion des arborescences :

- La transformation en mode intervallaire d'un arbre en autoréférence
- Le calcul des niveaux dans une représentation intervallaire
- La réalisation d'une vue en autoréférence dans une représentation intervallaire
- La numérotation hiérarchique des items d'un arbre

### Transformer une table en auto référence, en mode intervallaire

La méthode est plus simple qu'il n'y parait. A l'aide de deux tables intermédiaires (l'une contenant une pile de travail et l'autre l'arbre en auto référence) et d'une procédure, le tour est joué. Cette procédure, adaptée à SQL Server, s'inspire de celle décrite par Joe Celko dans son livre « *Joe Celko's Trees and Hierarchies in SQL for Smarties* ».

```
/*********************************
* Les objets de la transformation *
*********************************/

----------------------------------------
-- stockage de l'arbre en auto référence
----------------------------------------
CREATE TABLE T_ARBRE
(NOD_ACTUEL INT NOT NULL,    -- noeud actuel
 NOD_PARENT INT);            -- noeud parent
GO

----------------------------------------
-- pile de travail
----------------------------------------
CREATE TABLE T_PILE
(PIL_TOP INTEGER NOT NULL,      -- haut de pile
 PIL_ID  INT NOT NULL,          -- valeur traitée
 PIL_BG  INT,                   -- borne gauche
 PIL_BD  INT);                  -- borne droite
GO

------------------------------------------------------
-- procédure de calcul de l'arbre en mode intervallaire
------------------------------------------------------
```

```
CREATE PROCEDURE P_AUTOREF_VERS_INTERVAL
AS
BEGIN
-- borne actuelle, borne max, borne courante
DECLARE @TID INT, @TID_MAX INT, @TID_TOP INT;

-- assignation des valeurs de départ
SELECT @TID = 2, @TID_MAX = 2 * (SELECT COUNT(*) FROM T_ARBRE), @TID_TOP = 1;

-- mise en place de la racine dans la pile
INSERT INTO T_PILE
SELECT 1, NOD_ACTUEL, 1, @TID_MAX
FROM   T_ARBRE
WHERE  NOD_PARENT IS NULL;

-- suppression des lignes traitées de l'arbre
DELETE FROM T_ARBRE WHERE NOD_PARENT IS NULL;

-- boucle de traitement tant qu'il y a au moins un ecart de 1
WHILE @TID <= @TID_MAX - 1
BEGIN
-- s'il existe des relations parent enfant à traiter
   IF EXISTS (SELECT *
              FROM   T_PILE AS S1
                     INNER JOIN T_ARBRE AS T1
                           ON S1.PIL_ID = T1.NOD_PARENT
              WHERE  S1.PIL_TOP = @TID_TOP)
   BEGIN
-- alors insérer dans la pile
      INSERT INTO T_PILE
      SELECT @TID_TOP + 1, MIN(T1.NOD_ACTUEL), @TID, NULL
      FROM   T_PILE AS S1
             INNER JOIN T_ARBRE AS T1
                   ON S1.PIL_ID = T1.NOD_PARENT
      WHERE  S1.PIL_TOP = @TID_TOP;
-- les supprimer de l'arbre
      DELETE FROM T_ARBRE
      WHERE  NOD_ACTUEL = (SELECT PIL_ID
                           FROM   T_PILE
                           WHERE  PIL_TOP = @TID_TOP + 1)
-- on avance dans la hiérarchie
      SELECT @TID = @TID + 1, @TID_TOP = @TID_TOP + 1;
   END
   ELSE
-- il n'y a plus de relation parent enfant à traiter
   BEGIN
-- mise à jour de la pile
      UPDATE T_PILE
      SET    PIL_BD = @TID,
             PIL_TOP = - PIL_TOP
      WHERE  PIL_TOP = @TID_TOP;
-- on se décale dans la hiérarchie
      SELECT @TID = @TID + 1, @TID_TOP = @TID_TOP - 1;
   END;
END;
END;
GO

/*************************************
* fonctionnement de la transformation *
*************************************/
```

```
-- insertion des identifiants d'autoréférence dans la table T_ARBRE
INSERT INTO T_ARBRE (NOD_ACTUEL, NOD_PARENT)
SELECT VHC_ID, VHC_ID_FATHER
FROM   T_VEHICULE;
GO

-- lancement de la procédure
EXECUTE P_AUTOREF_VERS_INTERVAL;
GO

-- voir le résultat :
SELECT V.*, PIL_BD, PIL_BG
FROM   T_VEHICULE AS V
       INNER JOIN T_PILE AS P
             ON V.VHC_ID = P.PIL_ID
```

A ce stade deux possibilités s'offre à vous :

- modifier la table en autoréférence en y ajoutant les colonnes adéquates (borne)
- ajouter à la base une table pour gérer l'arbre intervallaire, en relation avec la table en autoréférence

```
------------------------------
-- SOLUTION 1 : modif
------------------------------

-- modifie la table originale
ALTER TABLE T_VEHICULE ADD VHC_BORNE_GAUCHE INT;
ALTER TABLE T_VEHICULE ADD VHC_BORNE_DROITE INT;
GO

-- met à jour les bornes
UPDATE T_VEHICULE
SET    VHC_BORNE_GAUCHE = PIL_BG,
       VHC_BORNE_DROITE = PIL_BD
FROM  T_VEHICULE AS V
      INNER JOIN T_PILE AS P
            ON V.VHC_ID = P.PIL_ID;
GO

------------------------------
-- SOLUTION 2 : ajout
------------------------------

-- création de la table intervallaire en référence
-- à la table en autoréférence
CREATE TABLE T_VEHICULE_INTERVAL_VHI
(VHC_ID INT NOT NULL PRIMARY KEY,
 VHI_BG INT NOT NULL,
 VHI_BD INT NOT NULL,
 CONSTRAINT FK_VHC FOREIGN KEY (VHC_ID)
                   REFERENCES T_VEHICULE (VHC_ID));
GO

-- insertion des bornes
INSERT INTO T_VEHICULE_INTERVAL_VHI
SELECT PIL_ID, PIL_BD, PIL_BG
```

```
FROM   T_PILE;
GO
```

Enfin, cerise sur le gâteau, il est intéressant d'ajouter l'**information de niveau** dans la table :

```
-- ajoute la colonne niveau dans la table des véhicules
ALTER TABLE T_VEHICULE ADD VHC_NIVEAU INT;
GO
-- met à jour les niveaux
UPDATE T_VEHICULE
SET    VHC_NIVEAU =
          COALESCE(
             (SELECT COUNT(*)
              FROM   T_VEHICULE AS VC1
                     INNER JOIN T_VEHICULE AS VC2
                           ON     VC2.VHC_BORNE_GAUCHE < VC1.VHC_BORNE_GAUCHE
                              AND VC2.VHC_BORNE_DROITE > VC1.VHC_BORNE_DROITE
              WHERE  VC1.VHC_ID = VHC.VHC_ID
              GROUP  BY VC1.VHC_ID
             ),  0)
FROM   T_VEHICULE VHC
GO
```

### Implémenter une vue en auto référence à partir du mode intervallaire

Le problème : nous avons une table en mode intervallaire et pour des besoins divers (par exemple un composant visuel attaqué en mode d'autoréférence) nous aimerions obtenir une vue représentant la table en auto référence.

Voici la vue, elle ne présente aucune difficulté :

```
CREATE VIEW dbo.V_VEHICULE_AUTOREF
AS
SELECT VHC.*,
-- calcul du père
       (SELECT VHC_ID
        FROM   dbo.T_VEHICULE T
        WHERE  T.VHC_BORNE_GAUCHE < VHC.VHC_BORNE_GAUCHE
          AND  T.VHC_BORNE_DROITE > VHC.VHC_BORNE_DROITE
          AND  T.VHC_NIVEAU = VHC.VHC_NIVEAU -1) AS VHC_ID_PERE
FROM   dbo.T_VEHICULE VHC;
GO
```

### Numéroter de manière hiérarchique les items d'un arbre

Une colle m'a été un jour posé par M. Bruno Petit de la sté Gomez Technologies : dans un arbre, comment numéroter de manière hiérarchique les différents items ?

La numérotation hiérarchique d'un arbre est une chose très utile pour présenter certaines informations. Par exemple pour numéroter les sections, chapitres, items, paragraphes… d'un document scriptural.
Cela conduit à quelque chose comme :

```
N        VHC_NAME
-------- ---------------
1        ALL
1.1      SEA
1.1.1    SUBMARINE
1.1.2    BOAT
1.2      EARTH
1.2.1    CAR
1.2.2    TWO WHEELES
1.2.2.1  MOTORCYCLE
1.2.2.2  BYCYCLE
1.2.3    TRUCK
1.3      AIR
1.3.1    ROCKET
1.3.2    PLANE
```

Pour cela je vous propose de procéder en deux temps :

- créer une fonction qui calcule la position de l'item par rapport à ses collatéraux (frères) plus « jeunes »
- utiliser une requête récursive pour concaténer le calcul des collatéraux aux différents niveaux.

La fonction de comptage :

```
CREATE FUNCTION dbo.F_COUNTCOLLATERAUX (@VHC_BORNE_GAUCHE INT, @VHC_ID_PERE INT)
RETURNS INT
AS
BEGIN
   RETURN (SELECT COUNT(*)
           FROM   dbo.V_VEHICULE_AUTOREF AS T
           WHERE  T.VHC_ID_PERE = @VHC_ID_PERE
             AND  T.VHC_BORNE_GAUCHE <= @VHC_BORNE_GAUCHE);
END;
GO
```

La requête qui calcule les numéros hiérarchique :

```
WITH T AS
(
SELECT VHC_ID, VHC_ID_PERE,
       VHC_BORNE_GAUCHE, VHC_BORNE_DROITE,
       VHC_NIVEAU, VHC_NAME,
       CAST(dbo.F_COUNTCOLLATERAUX (VHC_BORNE_GAUCHE, VHC_ID_PERE) + 1
            AS VARCHAR(max)) AS N
FROM   dbo.V_VEHICULE_AUTOREF AS TOUT1
WHERE  VHC_ID_PERE IS NULL
UNION  ALL
SELECT TIN2.VHC_ID, TIN2.VHC_ID_PERE,
       TIN2.VHC_BORNE_GAUCHE, TIN2.VHC_BORNE_DROITE,
       TIN2.VHC_NIVEAU, TIN2.VHC_NAME,
       N + '.' +
```

```
        CAST(dbo.F_COUNTCOLLATERAUX (TIN2.VHC_BORNE_GAUCHE, TIN2.VHC_ID_PERE)
             AS VARCHAR(32)) AS N
FROM    dbo.V_VEHICULE_AUTOREF AS TIN2
        INNER JOIN T
              ON TIN2.VHC_ID_PERE = T.VHC_ID
)
SELECT N, SPACE(VHC_NIVEAU) + VHC_NAME AS VHC_NAME
FROM   T
ORDER  BY VHC_BORNE_GAUCHE;
```

Et le résultat :

```
N        VHC_NAME
-------- ------------
1        ALL
1.1       SEA
1.1.1      SUBMARINE
1.1.2      BOAT
1.2       EARTH
1.2.1      CAR
1.2.2      TWO WHEELES
1.2.2.1     MOTORCYCLE
1.2.2.2     BYCYCLE
1.2.3      TRUCK
1.3       AIR
1.3.1      ROCKET
1.3.2      PLANE
```



# Bibliographie :


Sur les requêtes récursives et la CTE :

- Le langage SQL : Frédéric Brouard, Christian Soutou - Pearson Education 2005

- Joe Celko's Trees & Hierarchies in SQL for Smarties : Joe Celko - Morgan Kaufmann 2004
- SQL:1999 : J. Melton, A. Simon - Morgan Kauffman, 2002
- SQL développement : Frédéric Brouard - Campus Press 2001
- SQL for Dummies : Allen G. Taylor - Hungry Minds Inc 2001
- SQL avancé : Joe Celko - Vuibert 2000
- SQL-99 complete really : P. Gulutzan, T. Pelzer - R&D Books, 1999
- SQL 3, Implementing the SQL Foundation Standard : Paul Fortier - Mc Graw Hill, 1999


Sur le problème du voyageur de commerce (PVC) et l'algorithme de Dijkstra, voir :


- http://fr.wikipedia.org/wiki/Algorithme_de_Dijkstra
- http://www.hsor.org/downloads/Speedy_Delivery_teacher.pdf
- Précis de recherche opérationnelle : Méthodes et exercices - Robert Faure, Bernard Lemaire, Christophe Picouleau - Dunod 2004
- Exercices et Problemes Resolus de Recherche Operationnelle T1 Graphes Leurs Usages Leurs Algorithmes - Roseaux (collectif) - dunod 2004
- A Discipline of Programming - Edsger Wybe Dijkstra - Prentice Hall 1997