Recursive SQL is one of the most fascinating and powerful (and dangerous!) features offered in DB2 for z/OS Version 8. In this session, we will introduce the feature and show numerous examples of how it can be used to achieve things you would not have imagined being possible with SQL – all in one SQL statement! Fasten your seat belts and come join us in this exciting journey!
Session Outline

1. Recursion basics
2. Case studies - mathematical
3. Case studies - business
4. Performance aspects
5. Pitfalls and recommendations

Recursion basics
  • Theory and introduction
Case studies - mathematical
  • String of numbers
  • Factorial
  • Primes
  • Fibonacci Series
Case studies - business
  • Org chart
  • Generating test data
  • Missing data
  • Rollup
  • Allocation
  • Weighted allocation
  • RI children
  • Cheapest fare
  • Account linking
Performance aspects
  • Comparison to procedural logic
    • Org chart
    • Rollup
    • Allocate
Pitfalls and recommendations
  • Best practices
About the Instructor

Suresh Sane

♦ Co-author-IBM Redbooks
  ➢ SG24-6418, May 2002
  ➢ SG24-7083, March 2004
  ➢ SG24-7111, July 2006

♦ Educational seminars and presentations at IDUG North America, Asia Pacific, Canada and Europe

♦ IDUG Solutions Journal article – Winter 2000

♦ Numerous DB2 courses at various locations

♦ IBM Certified Solutions Expert for both platforms for Application Development and Database Administration

Suresh Sane works as a Database Architect at DST Systems in Kansas City, MO and manages the Database Consulting Group, which provides strategic direction for deployment of database technology at DST and has overall responsibility for the DB2 curriculum for about 1,500 technical associates.

Contact Information:

sssane@dstsystems.com

Suresh Sane
DST Systems, Inc.
1055 Broadway
Kansas City, MO 64105
USA

(816) 435-3803
If you have ever invested in a mutual fund, have had a prescription filled, or are a cable or satellite television subscriber, you may have already had dealings with our company.

DST Systems, Inc. is a publicly traded company (NYSE: DST) with headquarters in Kansas City, MO. Founded in 1969, it employs about 12,000 associates domestically and internationally.

The three operating segments - Financial Services, Output Solutions and Customer Management - are further enhanced by DST’s advanced technology and e-commerce solutions.
It’s cool but I will never use it...

♦ Of pure academic interest …”cool stuff” but
♦ … does it have any business value?
♦ THINK AGAIN!

 “…400,000 customers in a hierarchy of about 4,000 nodes resulting in over 2.5 million queries, the application took days to calculate total sales.”
“ …a single recursive query processed …in about 5 minutes”


See ref #7 for details on this article. I have to admit my first reaction was that this feature was “cool” but of little business value. Dan’s article provoked my interest in this fascinating area of SQL.
We will begin with a simple explanation which introduces the concept of recursion in general (not just for SQL).
What is 4!?

It is 4 * 3!

But what is 3!?

It is 3 * 2!

But what is 2!?

It is 2 * 1!

But what is 1!?

A simple example to illustrate the concept.
The best way to introduce recursive SQL? Look at a simple example. Let’s dive right in.

The definition – WITH NUMBERS – is an example of a Common Table Expression (CTE) which is similar to the Nested Table Expression (NTE) most of us already familiar with. It is required for using recursive SQL.

Also note the coloring scheme used consistently throughout this presentation to denote the 4 parts of the SQL.
Recursion Basics

- A technique where a function calls itself to perform some part of the task

- Requires:
  - An initialization select (“priming the pump”)
  - An iterative select (“pump more”) – from the previous, how do I get the next one?
  - A main select (“use the result”)

- Must use a common table expression (CTE)

- Allows for looping “procedural” logic – all in one statement!
  - DO WHILE, REPEAT etc from SQL Procedures can sometimes be replaced by one SQL statement

Some of the basics. The “pump” terminology is borrowed from Tink Tysor (see ref #1). Tink provides very thorough introduction to this fascinating topic.
Rules for CTE

♦ First full select of first UNION must not reference the CTE
♦ All selects within CTE cannot use DISTINCT
  ➢ This is a major limitation when cycles are present – DISTINCT on the outer query is expensive
  ➢ Need an ability to specify DISTINCT without the level
♦ All selects within CTE cannot use GROUP BY or HAVING
♦ Include only 1 reference to the CTE
♦ Initialization select and Iterative select columns must match (data types, lengths, CCSIDs)
♦ UNION must be a UNION ALL
♦ Outer joins cannot be part of any recursion cycle
♦ Subquery cannot be part of any recursion cycle

Some of the restrictions on what can be coded within the common table expression (CTE).
Let us look at some mathematical applications starting with some simple examples. We will look at Factorial, Generating primes and Fibonacci series.

If mathematics scares you, do not get discouraged – these are actually easier to illustrate the concept. We will build on this foundation and cover several business cases in the next section.
**Case 2 – Factorial**

WITH NUMBERS (LEVEL, FACTO) AS
  (SELECT 1, 1
   FROM SYSIBM.SYSDUMMY1
   UNION ALL
   SELECT LEVEL + 1, FACTO * (LEVEL + 1)
   FROM NUMBERS
   WHERE LEVEL < 12)

SELECT LEVEL AS NUMBER, FACTO AS FACTORIAL
FROM NUMBERS
ORDER BY LEVEL

<table>
<thead>
<tr>
<th>LEVEL</th>
<th>FACTORIAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>24</td>
</tr>
<tr>
<td>5</td>
<td>120</td>
</tr>
</tbody>
</table>

Just a little bit more complex – we simply generate the numbers and report them in the “use the results” section of the SQL.
WITH NUMBERS (LEVEL, PRIME) AS
( SELECT 1, 1
    FROM SYSIBM.SYSDUMMY1
    UNION ALL
    SELECT LEVEL + 1, LEVEL + 1
    FROM NUMBERS
    WHERE LEVEL < 5000 )

SQL to generate prime numbers.
Actual SQL to generate prime numbers, continued.

Note that we can stop checking for factors once we cross SQRT of that number (a big gain in performance – for example, instead of checking 5000, we can stop at 70).

On a theoretical note – the answer to the question “Is 1 a prime?” depends on the definition. Let’s leave that discussion to Math forums. Here, I assume it is a prime.
Case 4 – Fibonacci Series

♦ The first two numbers in the series are one and one.
♦ To obtain each number of the series, you simply add the two numbers that came before it. In other words, each number of the series is the sum of the two numbers preceding it.

The Parthenon with the “Golden ratio”

The Fibonacci Spiral

1 1 2 3 5 8 13 21

Some historical background information about Fibonacci Series (made popular by the recent “Da Vinci Code”):

• A sequence of numbers first created by Leonardo Fibonacci in 1202.
• It is a deceptively simple series, but its ramifications and applications are nearly limitless.
• The number 1.618..., or Phi, is the ratio of the next number to the previous number. Called the “Golden ratio”, it has implications in art, architecture and numerous other disciplines.
WITH NUMBERS (LEVEL, NEXTNUM, TOTAL) AS
(SELECT 1, 1, 1
FROM SYSIBM.SYSDUMMY1
UNION ALL
SELECT LEVEL + 1
  , (NEXTNUM + TOTAL)
  , (NEXTNUM + TOTAL + TOTAL)
FROM NUMBERS
WHERE LEVEL = LEVEL
AND   LEVEL <= 22 )

<table>
<thead>
<tr>
<th>Level</th>
<th>Nextnum</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>13</td>
<td>21</td>
</tr>
<tr>
<td>5</td>
<td>34</td>
<td>55</td>
</tr>
</tbody>
</table>

SQL to generate Fibonacci series.
Case 4 – Fibonacci Series – SQL (cont)

```
SELECT A.NEXTNUM
FROM NUMBERS A
WHERE A.LEVEL <= 22
UNION ALL
SELECT B.TOTAL
FROM NUMBERS B
WHERE B.LEVEL <= 22
ORDER BY 1
```

<table>
<thead>
<tr>
<th>Level</th>
<th>Nextnum</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>13</td>
<td>21</td>
</tr>
<tr>
<td>5</td>
<td>34</td>
<td>55</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Series

1
1
2
3
5
8
13
21
34
55
...

Actual SQL to generate Fibonacci series, continued.
Now let us start to explore recursive SQL for business applications.
A list of case studies we explore in this section.

- Case 5 – Org chart
- Case 6 – Generating test data
- Case 7 – Missing data
- Case 8 – Rollup
- Case 9 – Even allocation
- Case 10 – Weighted allocation
- Case 11 – RI children
- Case 12 – Cheapest fare
- Case 13 – Account linking
Case of how recursive SQL can be used effectively for traversing hierarchical structures.
## Case 5 – RECASE05 Table

<table>
<thead>
<tr>
<th>EMPID</th>
<th>EMPNAME</th>
<th>MGRID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Wolf</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>Sontag</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Truscott</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>Rodwell</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>Hamman</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>Jacoby</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>Wei</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>Kelsey</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>Reese</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>Schapiro</td>
<td>9</td>
</tr>
</tbody>
</table>

Contents of the table RECASE05 that defines the hierarchy.
WITH OC (LEVEL, MGRID, MGRNAME, EMPID, EMPNAME) AS
( SELECT 0, 0, '', EMPID, EMPNAME
FROM RECASE05
WHERE MGRID IS NULL
UNION ALL
SELECT BOSS.LEVEL + 1, SUB.MGRID, BOSS.EMPNAME,
SUB.EMPID, SUB.EMPNAME
FROM OC BOSS, RECASE05 SUB
WHERE BOSS.EMPID = SUB.MGRID AND BOSS.LEVEL < 5 )
SELECT OC.LEVEL, OC.MGRNAME, OC.EMPNAME
FROM OC
WHERE LEVEL > 0
ORDER BY OC.LEVEL, OC.MGRID, OC.EMPID

Actual SQL to generate the org chart.

The initialization select (WHERE MGRID IS NULL) could have been written instead as: (WHERE EMPID = 1) .
Contents of the common table expression OC as the SQL is executed.
Case 5 - Result

<table>
<thead>
<tr>
<th>LEVEL</th>
<th>MGRNAME</th>
<th>EMPNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>WOLF</td>
<td>SONTAG</td>
</tr>
<tr>
<td>1</td>
<td>WOLF</td>
<td>TRUSCOTT</td>
</tr>
<tr>
<td>1</td>
<td>WOLF</td>
<td>RODWELL</td>
</tr>
<tr>
<td>2</td>
<td>SONTAG</td>
<td>HAMMAN</td>
</tr>
<tr>
<td>2</td>
<td>SONTAG</td>
<td>JACOBY</td>
</tr>
<tr>
<td>2</td>
<td>TRUSCOTT</td>
<td>WEI</td>
</tr>
<tr>
<td>2</td>
<td>RODWELL</td>
<td>KELSEY</td>
</tr>
<tr>
<td>2</td>
<td>RODWELL</td>
<td>REESE</td>
</tr>
<tr>
<td>3</td>
<td>REESE</td>
<td>SCHAPIRO</td>
</tr>
</tbody>
</table>

… and the result.
Table structure for a table containing various data types which needs to be populated with test data.

If you needed 10,000 rows of test data on this table you would use a table editor (or SPUFI) to insert some rows and repeat them. How “random” would this data really be? In real life, not really random at all. This technique allows you to do so quite easily.
Case 6 – Generating Test Data - SQL

INSERT INTO RECASE06 WITH NUMBERS (LEVEL, NEXTONE) AS
(SELECT 1, 1
 FROM SYBASE.SYSDUMMY1
 UNION ALL
 SELECT LEVEL + 1, LEVEL + 1
 FROM NUMBERS
 WHERE LEVEL < 10 )
SELECT INTEGER(ROUND(RAND()*9999,0)) + 1
 , LEFT(SUBSTR('BCDFGHJKLMNPRSTVWZ',
 INTEGER(ROUND(RAND()*17,0))+1, 1)
 CONCAT
 SUBSTR('AEIOUY', INTEGER(ROUND(RAND()*5,0))+1, 1)
 <<< REPEAT 5 TIMES>>>
 , INTEGER(ROUND(RAND()*4,0)) + 3)

1 thru 10,000
5 sets of letters + vowels
Min 3, max 7

SQL used for this purpose.
Case 6 – Generating Test Data – SQL (cont)

, LEFT(SUBSTR('BCDFGHJKLMNPRSTVWZ', INTEGER(ROUND(RAND()*17,0))+1, 1)
  CONCAT SUBSTR('AEIOUY', INTEGER(ROUND(RAND()*5,0))+1, 1)
    <<< REPEAT 5 TIMES>>>
      , INTEGER(ROUND(RAND()*7,0)) + 3)
      , DECIMAL((1000.00 + RAND()*4000),7,2)
      , CURRENT DATE - 1 YEAR - INTEGER(20*365*RAND()) DAYS
FROM NUMBERS

SQL continued.
…and the result from one of my test runs (we come up with some interesting names!). It could be adjusted to reflect the regional demographics.
A simple case involving a set of two tables with time reported by week. Some employees may fail to report their time.
**Case 7 – Missing Data Non-recursive**

**Required Report**

<table>
<thead>
<tr>
<th>EMPNAME</th>
<th>FRIDAY_DATE</th>
<th>TOTHOURS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alan Sontag</td>
<td>2005-03-04</td>
<td>41.0</td>
</tr>
<tr>
<td>Alan Sontag</td>
<td>2005-03-11</td>
<td>42.0</td>
</tr>
<tr>
<td>Alan Sontag</td>
<td>2005-03-18</td>
<td>43.0</td>
</tr>
<tr>
<td>Alan Sontag</td>
<td>2005-03-25</td>
<td>44.0</td>
</tr>
<tr>
<td>Bobby Wolf</td>
<td>2005-03-04</td>
<td>39.0</td>
</tr>
<tr>
<td>Bobby Wolf</td>
<td>2005-03-11</td>
<td>0.0</td>
</tr>
<tr>
<td>Bobby Wolf</td>
<td>2005-03-18</td>
<td>46.0</td>
</tr>
<tr>
<td>Bobby Wolf</td>
<td>2005-03-25</td>
<td>0.0</td>
</tr>
<tr>
<td>Dorothy Truscott</td>
<td>2005-03-04</td>
<td>0.0</td>
</tr>
<tr>
<td>Dorothy Truscott</td>
<td>2005-03-11</td>
<td>0.0</td>
</tr>
<tr>
<td>Dorothy Truscott</td>
<td>2005-03-18</td>
<td>0.0</td>
</tr>
<tr>
<td>Dorothy Truscott</td>
<td>2005-03-25</td>
<td>0.0</td>
</tr>
</tbody>
</table>

List all employees and the reported time for March 2005. If an employee failed to report time, show it as zeroes.

We need a report that includes any missing information. Since we are dealing with optional data, our first thought would be a Left Outer Join.

However, how do you create a “left” table that has all the data when rows themselves are missing? See next slide to see how we can generate such a “left” table.
SELECT CART.EMPNAME, CART.FRIDAY_DATE,
    SUM(COALESCE(TIME.HOURS,0.0)) AS TOTHOURS
FROM
    ( SELECT
        EMPL.EMPID, EMPL.EMPNAME,
        FRIDAYS.FRIDAY_DATE
    FROM EX7EMPL EMPL
    INNER JOIN
        ( SELECT DISTINCT FRIDAY_DATE
            FROM EX7TIME
            WHERE FRIDAY_DATE BETWEEN '2005-03-01' AND '2005-03-31') AS FRIDAYS
    ON 1=1
    ) AS CART

CART is a nested table expression (NTE) of all employee-Fridays combinations that is such a “left” table, mentioned in the previous slide.
We now perform a Left Outer Join of CART with the TIME table to show the time reported, if any.

Notice that if all of them cooperate and no one enters the time, this query will fail!
CASE 7 – Missing Data - Recursive

WITH
DATES (LEVEL, NEXTDAY) AS
(
SELECT 1, DATE('2005-03-01')
FROM SYSIBM.SYSDUMMY1
UNION ALL
SELECT LEVEL + 1, NEXTDAY + 1 DAY
FROM DATES
WHERE LEVEL < 32
)

An alternative solution to the same problem using recursive SQL.

Also note that unlike the previous solution that requires at least one employee to enter the time, this solution works irrespective of who has entered the time.
SQL to generate the missing data, continued.

Alternatively, we could generate just the Friday dates by pushing these constructs into the CTE itself. However, care must be taken to start with a Friday (2005-03-04), otherwise the initial select adds zero rows to the CTE. You must also increment by 7 since first non-Friday will terminate the loop (this is left as an exercise for the reader).
Case of a hierarchy where the salary amounts must be rolled up to the higher costs center (e.g. Database Services and Programming are to be rolled up to the IT salary budget).
Case 8 – Rollup SQL

WITH ROLLUP (ACCT_NUM, ACCT_NAME, PARENT_ACCT_NUM, TOTAL_BUDGET) AS
(SELECT ACCT_NUM, ACCT_NAME, PARENT_ACCT_NUM, BUDGET_AMT
FROM   RECASE08
UNION ALL
SELECT A.ACCT_NUM, A.ACCT_NAME, A.PARENT_ACCT_NUM, B.TOTAL_BUDGET
FROM   RECASE08  A
       , ROLLUP            B
WHERE  A.ACCT_NUM = B.PARENT_ACCT_NUM
      AND  B.PARENT_ACCT_NUM IS NOT NULL
)

SQL used for this purpose.

The table RECCASE09 contains:

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCT_NUM</td>
<td>INTEGER NOT NULL</td>
<td>(PK)</td>
</tr>
<tr>
<td>ACCT_NAME</td>
<td>CHAR(20) NOT NULL</td>
<td></td>
</tr>
<tr>
<td>PARENT_ACCT_NUM</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td>BUDGET_AMT</td>
<td>DEC(9,2) NOT NULL</td>
<td></td>
</tr>
</tbody>
</table>
SELECT ACCT_NUM, ACCT_NAME,
    SUM(TOTAL_BUDGET) AS TOTAL_BUDGET
FROM ROLLUP
GROUP BY ACCT_NUM, ACCT_NAME
ORDER BY ACCT_NUM

SQL continued.
### Case 8 – Rollup Result

<table>
<thead>
<tr>
<th>ACCT_NUM</th>
<th>ACCT_NAME</th>
<th>TOTAL_BUDGET</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CORP</td>
<td>500</td>
</tr>
<tr>
<td>2</td>
<td>I.T.</td>
<td>400</td>
</tr>
<tr>
<td>3</td>
<td>H.R.</td>
<td>100</td>
</tr>
<tr>
<td>4</td>
<td>DB SVC</td>
<td>1000</td>
</tr>
<tr>
<td>5</td>
<td>PROG</td>
<td>5000</td>
</tr>
<tr>
<td>6</td>
<td>EMPL REL</td>
<td>50</td>
</tr>
<tr>
<td>7</td>
<td>BENEFITS</td>
<td>30</td>
</tr>
</tbody>
</table>

… and the result.
Case 9 – Even Allocation

Case of a hierarchy where bonus amounts are to be allocated evenly across all subordinates. For example, the $6,000 for DST is to be split into 2 parts ($3,000 each) for IT and HR. We must allocate from top down.
Case 9 – Even Allocation SQL

WITH ALLOC (ACCT_NUM, ACCT_NAME, PARENT_ACCT_NUM, TOTAL_BONUS) AS
(SELECT ACCT_NUM, ACCT_NAME, PARENT_ACCT_NUM, BONUS_AMT FROM RECASE10
UNION ALL
SELECT A.ACCT_NUM, A.ACCT_NAME, A.PARENT_ACCT_NUM, B.TOTAL_BONUS /
(SELECT COUNT(*) FROM RECASE10 X
WHERE X.PARENT_ACCT_NUM = A.PARENT_ACCT_NUM)
FROM RECASE10 A,
ALLOC B
WHERE B.ACCT_NUM = A.PARENT_ACCT_NUM)

SQL used for this purpose.

The table RECCASE10 contains:

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCT_NUM</td>
<td>INTEGER</td>
<td>NOT NULL (PK)</td>
</tr>
<tr>
<td>ACCT_NAME</td>
<td>CHAR(20)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>PARENT_ACCT_NUM</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td>BONUS_AMT</td>
<td>DEC(9,2)</td>
<td>NOT NULL</td>
</tr>
</tbody>
</table>
SELECT ACCT_NUM, ACCT_NAME,
SUM(TOTAL_BONUS)
FROM ALLOC
GROUP BY ACCT_NUM, ACCT_NAME
ORDER BY ACCT_NUM

SQL continued.
...and the result.

<table>
<thead>
<tr>
<th>ACCT_NUM</th>
<th>ACCT_NAME</th>
<th>TOTAL_BONUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CORP</td>
<td>6000</td>
</tr>
<tr>
<td>2</td>
<td>I.T.</td>
<td>4000 7000</td>
</tr>
<tr>
<td>3</td>
<td>H.R.</td>
<td>1000 4000</td>
</tr>
<tr>
<td>4</td>
<td>DB SVC</td>
<td>5000 8500</td>
</tr>
<tr>
<td>5</td>
<td>PROG</td>
<td>1000 4500</td>
</tr>
<tr>
<td>6</td>
<td>EMPL REL</td>
<td>100 2100</td>
</tr>
<tr>
<td>7</td>
<td>BENEFITS</td>
<td>50 2050</td>
</tr>
</tbody>
</table>
Similar hierarchy as before but this time, we need to accomplish a weighted allocation based on rating. For example the $6,000 bonus for CORP is to be divided using the ratings of 5 and 1 for IT and HR - IT will obtain $5,000 \times \frac{4}{5} = \frac{4}{5}$ of the $5,000. As before, we must process top down.
WITH ALLOC (ACCT_NUM, ACCT_NAME, PARENT_ACCT_NUM, TOTAL_BONUS) AS
(SELECT ACCT_NUM, ACCT_NAME, PARENT_ACCT_NUM, BONUS_AMT
FROM   RECASE11
UNION ALL
SELECT A.ACCT_NUM, A.ACCT_NAME, A.PARENT_ACCT_NUM, B.TOTAL_BONUS * A.RATING
/ (SELECT SUM(RATING) FROM RECASE11 X
    WHERE X.PARENT_ACCT_NUM = A.PARENT_ACCT_NUM)
FROM   RECASE11  A
   , ALLOC             B
WHERE  B.ACCT_NUM = A.PARENT_ACCT_NUM

SQL for this purpose.

The table RECCASE11 contains:

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCT_NUM</td>
<td>INTEGER NOT NULL</td>
<td>(PK)</td>
</tr>
<tr>
<td>ACCT_NAME</td>
<td>CHAR(20) NOT NULL</td>
<td></td>
</tr>
<tr>
<td>RATING</td>
<td>SMALLINT</td>
<td></td>
</tr>
<tr>
<td>PARENT_ACCT_NUM</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td>BONUS_AMT</td>
<td>DEC(9,2) NOT NULL</td>
<td></td>
</tr>
</tbody>
</table>
SELECT ACCT_NUM, ACCT_NAME, SUM(TOTAL_BONUS) FROM ALLOC GROUP BY ACCT_NUM, ACCT_NAME ORDER BY ACCT_NUM

SQL continued.
### Case 10 – Weighted Allocation Result

<table>
<thead>
<tr>
<th>ACCT_NUM</th>
<th>ACCT_NAME</th>
<th>TOTAL_BONUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CORP</td>
<td>5000</td>
</tr>
<tr>
<td>2</td>
<td>I.T.</td>
<td>4000 8000</td>
</tr>
<tr>
<td>3</td>
<td>H.R.</td>
<td>1000 2000</td>
</tr>
<tr>
<td>4</td>
<td>DB SVC</td>
<td>5000 10000</td>
</tr>
<tr>
<td>5</td>
<td>PROG</td>
<td>1000 4000</td>
</tr>
<tr>
<td>6</td>
<td>EMPL REL</td>
<td>100 1300</td>
</tr>
<tr>
<td>7</td>
<td>BENEFITS</td>
<td>50 850</td>
</tr>
</tbody>
</table>

... and the result.
An example of a hierarchy that show the referential integrity (RI) implemented using foreign keys. These relationships are visible in the catalog table SYSIBM.SYSRELS.

Notice the recursive relationship between B and itself, as well as the circular definition of constraints from D to H to J and back to D.
WITH OC (LEVEL, PARENTTAB, CHILDTAB) AS
  (SELECT 0 , 'Z', 'your start table'
   FROM SYSIBM.SYSDUMMYU
  UNION ALL
  SELECT PARENT.LEVEL + 1
   , SUBSTR(CHILD.REFTBNAME,1,1)
   , SUBSTR(CHILD.TBNAME,1,1)
  FROM OC PARENT, SYSIBM.SYSRELS CHILD
  WHERE  PARENT.CHILDTAB = CHILD.REFTBNAME
  AND    CHILD.CREATOR = …
  AND    CHILD.REFTBCREATOR = …
  AND    CHILD.REFTBNAME  <> CHILD.TBNAME
  AND PARENT.LEVEL < 10 )

Unicode in catalog

my children

Eliminate Self-ref

SQL used for this purpose.
SQL continued.

Notice that the CTE generates much more data than necessary (e.g. node D is revisited multiple times) and the DISTINCT is needed to eliminate the duplicates. A very handy SQL enhancement would be for DB2 to allow us to specify DISTINCT (without level) within the CTE itself. V(future)) perhaps?
Case 11 – RI Children Result

<table>
<thead>
<tr>
<th>PATRENTTAB</th>
<th>CHILDTAB</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>A</td>
<td>D</td>
</tr>
<tr>
<td>B</td>
<td>E</td>
</tr>
<tr>
<td>B</td>
<td>F</td>
</tr>
<tr>
<td>C</td>
<td>G</td>
</tr>
<tr>
<td>C</td>
<td>J</td>
</tr>
<tr>
<td>D</td>
<td>H</td>
</tr>
<tr>
<td>D</td>
<td>I</td>
</tr>
<tr>
<td>H</td>
<td>J</td>
</tr>
<tr>
<td>J</td>
<td>D</td>
</tr>
</tbody>
</table>

... and the result.
A case study to list and evaluate multiple paths.
### RECASE12

<table>
<thead>
<tr>
<th>FROM_CITY</th>
<th>TO_CITY</th>
<th>FARE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCI</td>
<td>ORD</td>
<td>100</td>
</tr>
<tr>
<td>MCI</td>
<td>CDG</td>
<td>200</td>
</tr>
<tr>
<td>MCI</td>
<td>DFW</td>
<td>500</td>
</tr>
<tr>
<td>ORD</td>
<td>STL</td>
<td>50</td>
</tr>
<tr>
<td>ORD</td>
<td>IAH</td>
<td>100</td>
</tr>
<tr>
<td>STL</td>
<td>IAH</td>
<td>25</td>
</tr>
<tr>
<td>STL</td>
<td>DFW</td>
<td>150</td>
</tr>
<tr>
<td>IAH</td>
<td>DFW</td>
<td>50</td>
</tr>
</tbody>
</table>

Contents of the table RECASE12 that defines the available routes and the associated fares.
WITH
ROUTING (LEVEL, FROM_CITY, TO_CITY, CHAIN, FARE) AS
( SELECT 0,
  CAST('-->' AS CHAR(3))
  , CAST('MCI' AS CHAR(3))
  , CAST('-->MCI' AS VARCHAR(60))
  , 0
FROM SYSIBM.SYSDUMMY1
UNION ALL
SELECT PARENT.LEVEL + 1,
  CAST(SUBSTR(CHILD.FROM_CITY,1,3) AS CHAR(3))
  , CAST(SUBSTR(CHILD.TO_CITY,1,3) AS CHAR(3))
  , PARENT.CHAIN CONCAT '->' CONCAT
    CAST(SUBSTR(CHILD.TO_CITY,1,3) AS CHAR(3))
  , PARENT.FARE + CHILD.FARE
FROM ROUTING PARENT, RECASE12 CHILD
WHERE PARENT.TO_CITY = CHILD.FROM_CITY
AND PARENT.LEVEL < 10)

SQL used for this purpose.
SELECT DISTINCT
    SUBSTR(ROUTING.CHAIN,4,56) AS ROUTE,
    ROUTING.FARE
FROM ROUTING
WHERE ROUTING.LEVEL > 0
AND ROUTING.CHAIN LIKE '%MCI%'
AND ROUTING.CHAIN LIKE '%DFW'
ORDER BY ROUTE

SQL continued.

The length of the generated path (60 bytes, of which 56 are reported) is arbitrary and truncated here to make the output readable.
### Case 12 – Cheapest Fare Result

<table>
<thead>
<tr>
<th>ROUTE</th>
<th>FARE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCI-&gt;DFW</td>
<td>500</td>
</tr>
<tr>
<td>MCI-&gt;ORD-&gt;IAH-&gt;DFW</td>
<td>250</td>
</tr>
<tr>
<td>MCI-&gt;ORD-&gt;STL-&gt;DFW</td>
<td>300</td>
</tr>
<tr>
<td>MCI-&gt;ORD-&gt;STL-&gt;IAH-&gt;DFW</td>
<td>225</td>
</tr>
<tr>
<td>MCI-&gt;STL-&gt;DFW</td>
<td>350</td>
</tr>
<tr>
<td>MCI-&gt;STL-&gt;IAH-&gt;DFW</td>
<td>275</td>
</tr>
</tbody>
</table>

… and the result.

A further enhancement (including the flight start and end days/times) would allow us to select connecting flights (e.g. next one must leave 1 hour to 4 hours from the arrival of the previous flight etc) and prepare a true itinerary like Travelocity or other search engines (… all with no pop-up ads!!).
A case study to group accounts that share at least one customer. Such linked accounts may receive favorable pricing.
WITH LINKING (LEVEL, ACCT_NUM, CUST_NUM) AS
  ( SELECT 0
    , ACCT_NUM
    , CUST_NUM
    FROM RECASE13
    WHERE ACCT_NUM = 4
  UNION ALL
  SELECT PARENT.LEVEL + 1
    , CHILD2.ACCT_NUM
    , CHILD2.CUST_NUM
  FROM LINKING PARENT
    , RECASE13 CHILD1
    , RECASE13 CHILD2
  WHERE PARENT.CUST_NUM = CHILD1.CUST_NUM
  AND PARENT.ACCT_NUM <> CHILD1.ACCT_NUM
  AND CHILD1.ACCT_NUM = CHILD2.ACCT_NUM
  AND PARENT.LEVEL < 5)
SELECT DISTINCT ACCT_NUM
FROM LINKING

SQL used for this purpose.
OK, it is complex and compact but from a performance perspective, how does recursive SQL look? We will explore this issue by comparing it with procedural logic (COBOL program) that accomplishes the same tasks.
The hierarchical structure used for the case study – each node with 4 children, 8 level deep.
Table structure and the indexes available.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Indexes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCT_NUM</td>
<td>INTEGER</td>
<td>Index K0 (uniq)</td>
</tr>
<tr>
<td>ACCT_NAME</td>
<td>CHAR(20)</td>
<td>Index K1</td>
</tr>
<tr>
<td>PARENT_ACCT_NUM</td>
<td>INTEGER</td>
<td>Index K2</td>
</tr>
<tr>
<td>AMOUNT</td>
<td>DEC(15,2)</td>
<td></td>
</tr>
<tr>
<td>ACCT_LEVEL</td>
<td>INTEGER</td>
<td></td>
</tr>
</tbody>
</table>
SELECT C.PARENT_ACCT_NUM, C.ACCT_NUM
FROM RECASE14 P
    , RECASE14 C
WHERE P.ACCT_LEVEL = :WS-LOOP-LEVEL
    AND P.ACCT_NUM = C.PARENT_ACCT_NUM
ORDER BY PARENT_ACCT_NUM
    , ACCT_NUM

SET LEVEL = 0 FOR ROOT
FOR EACH LEVEL (0 TO 7 BY 1)
    FOR EACH ROW OF CHILD-CURSOR
        UPDATE CHILD WITH LEVEL = PARENT + 1
    NEXT CHILD
NEXT LEVEL

SQL for setting the level (used in all 3 cases) later.
Logic for Org chart.

<<< using the pre-set level-number >>>

SELECT BOSS.LEVEL_NUM
, BOSS.ACCT_NAME
, SUB.ACCT_NAME
FROM RECASE14 BOSS
, RECASE14 SUB
WHERE SUB.PARENT_ACCT_NUM = BOSS.ACCT_NUM
ORDER BY BOSS.LEVEL_NUM
, BOSS.ACCT_NUM, SUB.ACCT_NUM
Logic for rollup.
Procedural Logic for Allocate

<<< using the pre-set level-number >>>

FOR EACH LEVEL 0 TO 7 BY 1
    FOR EACH ROW OF PARENT-CURSOR
        DETERMINE # OF CHILDREN
    FOR EACH ROW OF CHILD-CURSOR
        ADD PRO-RATED AMOUNT
        AND UPDATE CHILD
    NEXT CHILD
    NEXT PARENT
NEXT LEVEL

SHOW ALL ROWS

Logic for allocation.
…and a comparison of they stack up.

Unlike the Dan Luksetich case mentioned earlier, my results are not dazzling in favor of recursive SQL – it is better, but not by an order of magnitude.

Admittedly, other variables will also impact the benchmarks. For example, if the procedural logic is needlessly complex or inefficient, recursive SQL can look much better. I have avoided a biased approach – the best SQL and the best procedural logic are being compared. Other variables that could affect are the presence of indexes, depth of the hierarchy and how balanced the hierarchy is (very evenly balanced in this “lab” exercise). Perhaps, the size and number of SORTWORK datasets for DSNDB07 may also affect the performance.
Let’s summarize what we have learned and point out some pitfalls.
Limiting the Depth

♦ Prone to infinite cycles unless controlled properly
♦ DB2 imposes no limit on the depth of recursion but issues a warning (SQLSTATE = 01605, SQLCODE=+347) for an SQL statement that does not use a control variable to limit the depth
  ➢ Warning is essentially useless in SPUFI (displayed too late)
♦ Warning is based on the absence of:
  ➢ An integer column that increments by a constant
  ➢ A predicate of the type LEVEL < constant or LEVEL < :hv
  ➢ LEVEL < (subselect query) is allowed but issues a warning
  ➢ The SQL parser logic is quite primitive – for example:
    • a loop from 10 to 1 is not infinite but will still generate a warning
    • Subtracting -1 (same as adding 1) will still generate a warning

The potential to create an infinite loop is ever present since DB2 will not limit it to a specified depth.
Danger lurks around the corner! Let’s revisit case 11.
Causing Infinite Loops

WITH OC (LEVEL, PARENTTAB, CHILDTAB) AS
(SELECT 0, 'Z', 'your start table'
FROM SYSIBM.SYSDUMMYU
UNION ALL
SELECT PARENT.LEVEL + 1,
    SUBSTR(CHILD.REFTBNAME,1,1),
    SUBSTR(CHILD.TBNAME,1,1)
FROM OC PARENT, SYSIBM.SYSDR
    CHILD
WHERE PARENT.CHILDTAB =
    CHILD.REFTBNAME
AND CHILD.CREATOR = ...
AND CHILD.REFTBCREATOR = ...
AND CHILD.REFTBNAME <>
    CHILD.TBNAME
AND PARENT.LEVEL < 10 )

Without the extra protection offered by the level clause, the warning and possible infinite loop.
Recommendations

- Use controls to limit the depth of recursion
- Consider possible future changes that could cause infinite loops (e.g. recursive or circular references)
- Very useful for reporting data that does not exist!
- Allows looping logic that would otherwise require procedural code or SQL Procedures
- In most cases, simplifies processing logic
- In specific instances, can lead to a huge performance gain

Powerful but dangerous in the wrong hands.
But it seems too hard…

♦ Seems too complex for “the average programmer”?
♦ Perhaps, but consider what Prof Dijkstra had to say…

“Don't blame me for the fact that competent programming, as I view it as an intellectual possibility, will be too difficult for 'the average programmer', you must not fall into the trap of rejecting a surgical technique because it is beyond the capabilities of the barber in his shop around the corner.”

E. F. Dijkstra

Just because it is not for the masses, let it not stop you.
Summary

1. Recursion basics
   - Introducing the new feature

2. Case studies - mathematical
   - String of numbers, factorial, primes and Fibonacci
   - It’s not that hard!

3. Case studies - business
   - Org chart, Generating test data, Missing data, Rollup, Even allocation, Weighted allocation, RI children, Cheapest fare, Account linking
   - A little complex but compact

4. Performance aspects
   - Org chart, rollup, allocate
   - Compact and a better performer

5. Pitfalls and recommendations
   - Powerful but dangerous!

I trust this session has empowered you with the knowledge to exploit Recursive SQL fully. Good Luck!
1. Recursive SQL for Dummies – B.L. “Tink” Tysor - IDUG NA 2005 - Session A1
2. Recursive SQL – Unleash the Power! – Suresh Sane - IDUG EU 2007 - Session E5
3. DB2 UDB for z/OS V8 – Everything You Ever Wanted to Know, ... and More – SG24-6079
4. DB2 UDB for z/OS Version 8 Performance Topics – SG24-6465
5. Having Fun with Complex SQL – Suresh Sane - IDUG AP 2005 - Session B5

Some of the many useful references.
Not really…in the unlikely event that we actually have time ....I doubt it!
Thank you and good luck with Recursive SQL!