

Really Cool DB2 UDB SQL: Utilizing Table Expressions

By Daniel L. Luksetich, Software Alternatives Incorporated

The Introduction

Approximately two and a half years ago, I received my first DB2 UDB for AIX assignment. An atypical first project, this was to be an enterprise-wide billing application, processing millions of transactions per day, and storing gigabytes of data. The application was to incorporate some very complex business logic, and was expected to push the database to its limits. Knowing nothing about DB2 UDB for AIX would turn out to be an advantage. Not knowing the limits of the engine meant there weren't any until we found them.

The Story

The application categorized the sale of products by characteristics. A sale could be categorized by any number of these characteristics, and the price a customer could be charged is dependent upon the characteristics of the sale. The front office application would process a customer request for a product and then, after delivery of the product, would send the product sale information, including the characteristics, to the billing application for pricing. The characteristics were grouped together in a table [Figure 1].

| CHAR PGRP | |
|--------------|----------|
| GRP_ID | INT |
| CHAR_ID_NUM | SMALLINT |
| CHAR_VAL_NUM | SMALLINT |

Figure 1: Characteristic Group Table

Each set of characteristics of a sale would have to be matched against the grouping table to determine the group identifier for the price. This seemed like a perfect opportunity for my first recursive query. After several days and several fruitless attempts to solve the problem, I could not write a query that worked. One night, very late, I was awoken by the calls of coyotes outside my window (I live in the "sticks"). Not being able to go back to sleep, I finally solved the query. The "coyote query" [Figure 2] solved the problem by denormalizing the table, then matched the set of characteristics to the denormalized result. While the query worked, it did not perform with the desired sub-second response, and the problem was solved programmatically. Nonetheless, my eyes were opened. After using table expressions and powerful SQL constructs, I then realized the true power of DB2 UDB SQL.

```

with parps(string, grp_id, char_id_num, char_val_num, sort_num, n) as
  ((select cast(x.char_id_num as char(2)) concat
    cast(x.char_val_num as char(2)) as varchar(100)),
    grp_id, x.char_id_num, x.char_val_num, y.sort_num, 1
  from qual.char_parp_grp x, qual.prdt_char_prty_mtx y
  where x.char_id_num = 9
    and x.char_val_num = 10
    and y.prdt_cde = '07000'
    and y.sort_num = 1
    and y.char_id_num = x.char_id_num)
union all
  (select cast(p.string concat ',' concat
    cast(a.char_id_num as char(2)) concat
    cast(a.char_val_num as char(2)) as varchar(100)),
    a.grp_id, a.char_id_num, a.char_val_num, z.sort_num,
    p.n + 1
  from parps p, qual.char_parp_grp a, qual.prdt_char_prty_mtx z
  where p.grp_id = a.grp_id
    and z.char_id_num = a.char_id_num
    and z.prdt_cde = '07000'
    and z.sort_num = p.sort_num + 1)
)
select grp_id, min(n)
from parps
where string like
  '26 1, 28 1, 25 2, 19 2, 17 7, 27 1, 9 10, 21 1, 24 1%'
group by grp_id
having min(n) = 9 and max(n) = 9;

```

Figure 2: The “Coyote” Query

Table Expressions

A table expression is simply an SQL statement used within another SQL statement, in place of what would have been a table. Since the result of an SQL SELECT statement is a table, this result can be used as a table in another part of an SQL statement. This is true for mainframe and non-mainframe versions of DB2 UDB. However, the non-mainframe version can use table expressions just about anywhere you can have an expression in a SQL statement. By nesting these expressions in a variety of places within a statement, days and days of application programming can be replaced by a single SQL statement. Pushing more application logic into the database engine will conserve valuable network transfer time, improving transaction response times, and better utilizing the machines an application will run on. This article will attempt to demonstrate these powerful expressions.

People have been coding table expressions for years. A subquery utilizes a table expression [Figure 3].

```

SELECT *
FROM SYSCAT.INDEXES
WHERE TABNAME IN
    (SELECT TABNAME
     FROM SYSCAT.TABLES
     WHERE DEFINER = 'USER1');

```

Figure 3: A Simple Subquery

Table expressions can be used to further process the result of a SQL statement [Figure 4].

```

SELECT MAX(CNT_COL)
FROM (SELECT COUNT(*) AS CNT_COL
      FROM SYSCAT.INDEXES
      GROUP BY SCHEMA) AS TAB1;

```

Figure 4: Another Simple Subquery

DB2 UDB allows you to code an SQL statement into something called a common table expression. Utilizing a "WITH" clause, a common table expression can allow the programmer to create extremely powerful statements. Although the table created lasts for only the length of a statement, common table expressions can be nested and reused throughout the statement. **Figure 5** is a simple example of a common table expression.

```

WITH TLBS (TBNAME) AS
    (SELECT TBNAME
     FROM SYSCAT.TABLES)
SELECT TBNAME
FROM TLBS;

```

Figure 5: A Common Table Expression

The Examples

Two years after the failed "coyote query", I had the distinct pleasure of running the problem past Richard Yevich, who exclaimed "That's relation division!" Again my eyes were opened, and I once again needed to solve the problem of identifying a group of characteristics from their parts. Since I was now off the billing project, I needed a new and more interesting way of demonstrating the problem. I thought, instead of products and characteristics, I'll do birds and their features. A bird watcher would have a table of birds and their features. Then the bird watcher goes into the woods, sees a bird and writes down the features. The features are typed into the query, which then returns the "best fit" bird. Great example, but I'm now working with Richard and Susan...so here goes. Instead of products and characteristics, we're going to do cocktails and their ingredients. We'll use common table expressions to enable the relational

division and to eliminate the need for dynamic SQL. Any example from this point forward will be specific to the non-mainframe UDB with mainframe differences pointed out.

Query 1: OK, so the party starts, and like any good techno-geek, I've got a DB2 table filled with my favorite cocktails and their ingredients [Figure 6].

| DRINKS | |
|------------|----------|
| DRINK | CHAR(20) |
| INGREDIENT | CHAR(20) |

| Drink | Ingredient | Drink | Ingredient |
|-------------|------------|-------------|----------------|
| MARTINI | VERMOUTH | SCREWDRIVER | VODKA |
| MARTINI | GIN | SCREWDRIVER | ORANGE JUICE |
| MARTINI | OLIVE | FUZZY NAVEL | PEACH SCHNAPPS |
| GIN&TONIC | GIN | FUZZY NAVEL | ORANGE JUICE |
| GIN&TONIC | TONIC | FUZZY SCREW | PEACH SCHNAPPS |
| MANHATTAN | WISKEY | FUZZY SCREW | ORANGE JUICE |
| MANHATTAN | VERMOUTH | FUZZY SCREW | VODKA |
| MANHATTAN | CHERRY | | |
| MIND ERASER | VODKA | | |
| MIND ERASER | CLUB SODA | | |

Figure 6: The Drinks Table

I've got a fully stocked bar, so the first question I get from a curious party goes is:

What drinks contain vodka, orange juice, and peach schnapps?

```
WITH ONHAND (INGREDIENT) AS
  (VALUES ('VODKA'), ('ORANGE JUICE'), ('PEACH SCHNAPPS'))
SELECT DRINK
FROM DRINKS A, ONHAND B
WHERE A.INGREDIENT = B.INGREDIENT
GROUP BY DRINK
HAVING COUNT(*) = (SELECT COUNT(*) FROM ONHAND);
```

This query first begins with a common table expression called "ONHAND", and then it joins the ONHAND table to the DRINKS table (the relational division). A "GROUP BY" clause is used to count the number of ingredients for each drink that contained one or more of the desired ingredients. Finally, a scalar ("returning one value") table expression (aka scalar subquery) is used to count the number of ingredients we are interested in, assuring that all of the drinks selected contain all of those ingredients.

The common table expression is utilized because I wanted to be able to code these queries in such a way that they could be used in an application program as static SQL. In this particular case, my common table expression is simply a "VALUES" clause containing the list of

ingredients that I will use to query my table. I could very easily put as many host variables as I want into this part of the query, if the query was static and embedded in a program. Any variable which does not receive a valid input, would be set to a default value, say blanks. Then, by simply adding a predicate to deal with columns in the ONHAND table that are set to blank, the static query is possible:

```

WITH ONHAND (INGREDIENT) AS
  (VALUES (:H1), (:H2), (:H3), (:H100))
SELECT DRINK
FROM DRINKS A, ONHAND B
WHERE A.INGREDIENT = B.INGREDIENT
GROUP BY DRINK
HAVING COUNT(*) =
  (SELECT COUNT(*) FROM ONHAND WHERE INGREDIENT <> ' ');

```

The dynamic SQL version of the same query is:

```

SELECT DRINK
FROM DRINKS A
WHERE A.INGREDIENT = 'VODKA'
AND EXISTS
  (SELECT 1
   FROM DRINKS B
   WHERE A.DRINK=B.DRINK
        AND B.INGREDIENT = 'ORANGE JUICE'
        AND EXISTS
          (SELECT 1
           FROM DRINKS C
           WHERE C.DRINK=B.DRINK
                AND C.INGREDIENT = 'PEACH SCHNAPPS'))

```

Since this query grows with each ingredient added to the search, the advantage of the static solution is evident.

Query 1 Answer:

FUZZY SCREW

Query 2: Late in the evening when things are really rocking, and the supply of ingredients starts to run a little low, one question I might ask is:

I have only vodka, orange juice, and peach schnapps. What can I make?

```

WITH ONHAND (INGREDIENT) AS
  (VALUES ('VODKA'), ('ORANGE JUICE'), ('PEACH SCHNAPPS'))
SELECT DRINK
FROM DRINKS A, ONHAND B
WHERE A.INGREDIENT = B.INGREDIENT
GROUP BY DRINK
HAVING COUNT(*) =
  (SELECT COUNT(*) FROM DRINKS C
   WHERE C.DRINK=A.DRINK);

```

This query is almost exactly the same as the first query, except for the scalar table expression at the end. Here, we are matching the count of ingredients of all drinks that match one or more of the requested ingredients with the total count of ingredients in that drink. This is accomplished by correlating the scalar table expression subquery C with the joined DRINK table A. This gets a total count of all ingredients for each selected drink, and the query ultimately returns only the drinks that can be made with those ingredients. Yippee!

Query 2 Answer:

| |
|-------------|
| FUZZY NAVEL |
| FUZZY SCREW |
| SCREWDRIVER |

Query 3: With the stockpile of ingredients running low, but my intoxicated friends becoming increasingly more adventurous, another important question to answer would be:

I have only vodka, orange juice, and peach schnapps. What drinks can I make that utilize the most of these ingredients?

```

WITH ONHAND (INGREDIENT) AS
    (VALUES ('VODKA'), ('ORANGE JUICE'), ('PEACH
SCHNAPPS'))
,MATCHES (DRINK, NUM_ING) AS
    (SELECT DRINK, COUNT(*)
    FROM DRINKS A, ONHAND B
    WHERE A.INGREDIENT = B.INGREDIENT
    GROUP BY DRINK
    HAVING COUNT(*) = (SELECT COUNT(*) FROM DRINKS C
    WHERE C.DRINK=A.DRINK))

SELECT DRINK
FROM MATCHES
WHERE NUM_ING = (SELECT MAX(NUM_ING) FROM MATCHES);

```

The answer to query 3 is simply to take the answer from query 2, and keep only the drink with the most ingredients. The easiest way to accomplish this is to place query 2 into a common table expression. This query then contains 2 common table expressions, separated by a comma. The first common table expression (ONHAND) feeds the second common table expression (MATCHES), which feeds the final piece of the query. This final piece of the query actually looks quite simple, pulling of the drink from the MATCHES common table expression that contains the most ingredients. YES!

Query 3 Answer:

| |
|-------------|
| FUZZY SCREW |
|-------------|

Now, we replace drinks with products, and ingredients with characteristics, and we have solved the coyote query problem with a very fast and efficient single SQL statement! Since common table expressions are not available on the mainframe, the problem cannot be solved with a single statement. However, utilizing global temporary tables could solve the problem. In the drink example query 3, an INSERT loop would place the requested ingredients into a global temporary table (ONHAND), that table would be joined to the DRINKS table with the results of that query inserted into another global temporary table (MATCHES). The final query would process the second global temporary table (MATCHES) exactly as it does in the query 3 example.

Nested Table Expressions

DB2 UDB allows you to place a scalar nested table expression into a SELECT clause, which can result in some extremely powerful SQL statements. Here is a simple example:

```
SELECT DRINK, INGREDIENT,
       (SELECT COUNT(DISTINCT DRINK) FROM DRINKS) ,
       (SELECT COUNT(*) FROM DRINKS WHERE DRINK = 'MARTINI')
FROM DRINKS
WHERE DRINK = 'MARTINI' ;
```

This query delivers the ingredients for a martini, the count of the drinks in the table, and a count of the ingredients in a martini. Goodbye application program, hello SQL! This query is non-mainframe specific.

Hey Buddy...Correlate This!

A nested table expression can also be correlated, which allows for greater flexibility and power in the coding of SQL statements, as well as the opportunity for tremendous performance improvements. The preceding SQL statement can be improved by correlating the predicate from the "SELECT COUNT" nested expression with the outer part of the query:

```
SELECT DRINK, INGREDIENT,
       (SELECT COUNT(DISTINCT DRINK) FROM DRINKS) ,
       (SELECT COUNT(*) FROM DRINKS B WHERE A.DRINK = B.DRINK)
FROM DRINKS A
WHERE DRINK = 'MARTINI' ;
```

Not only have I coded 'MARTINI' only once in the query, but also I influenced the optimizer toward using an index on the DRINK column of the nested table expression.

In my opinion, the UDB engine is primarily a warehouse engine. Large, complex queries are processed with absolutely astonishing speed. On an SMP machine, DB2 can use intra-partition parallelism quite effectively to improve query elapsed time. In this regard, the engine has a propensity towards a merge scan join. While merge scan is a significant performance advantage when the query is processing very large amounts of data, it can be a disadvantage for

transaction processing, especially when fewer rows of data are expected to be processed. Correlating table expressions encourages the optimizer to use indexes to resolve the correlated references, which subsequently favors nested loop join over merge scan. This is demonstrated in the following real-world examples:

```
SELECT DISTINCT A.CUST_ID, A.CUST_ID AS SUM_FLG,  
B.BUNDLE_FLG  
  FROM CUST_SUMMARY A  
LEFT OUTER JOIN  
  (SELECT DISTINCT CUST_ID, 1 AS BUNDLE_FLAG  
  FROM CUST_SUMM_PART) B  
ON A.CUST_ID = B.CUST_ID;
```

The above query is basically an existence check, getting a customer id, and a flag, indicating that the customer has a summary, and certain parts (here called bundles) of the summary. Two tables are required to produce this set of flags, the summary table, and the part table. Since it's an existence check, it should be expected to process very few rows, and returns only one row per customer in the summary table. The expected elapsed time of this query is very fast, since the query is part of a transaction. The problem here is that the entire part table is searched, regardless of whether or not the customer is in the summary table. This is because the table expression called B has to be materialized prior to being joined to the summary table called A. DB2 utilizes a merge scan join, and reads the entire part table. This query can be modified, utilizing a correlated nested table expression:

```
SELECT DISTINCT A.CUST_ID,  
  (SELECT MAX(CUST_ID)  
   FROM CUST_SUMM_PART B  
   WHERE A.CUST_ID = B.CUST_ID) AS BUNDLE_FLG  
FROM CUST_SUMMARY A;
```

Not only does this query utilize an index on the CUST_ID column of the part table, it's also a lot easier to read. Performance is greatly improved, because the optimizer no longer has to process the entire part table to return a result for the nested table expression. The MAX function is used to make the table expression scalar, since you can only code scalar table expressions in a SELECT clause...COOL!

HOW COOL CAN THIS GET?

Not only can you correlate a table expression in a subquery or a SELECT clause, you can also correlate a table expression in a join! This, in my opinion, changes everything, and allows programmers to code some unbelievably powerful SQL statements. The following examples utilize the DB2 sample table (mainframe) called EMP [Figure 7].

| EMP | |
|----------|--------------|
| EMPNO | CHAR(6) |
| FIRSTNME | VARCHAR(12) |
| MIDINIT | CHAR(1) |
| LASTNAME | VARCHAR(15) |
| WORKDEPT | CHAR(3) |
| PHONENO | CHAR(4) |
| HIREDATE | DATE |
| JOB | CHAR(8) |
| EDLEVEL | SMALLINT |
| SEX | CHAR(1) |
| SALARY | DECIMAL(9,2) |
| BONUS | DECIMAL(9,2) |
| COMM | DECIMAL(9,2) |

Figure 7: The Mainframe V6 EMP Table

Problem: Determine the employee number and salary of sales representatives, along with the average salary and head count of their departments.

This problem can be solved using a left outer join, joining two table expressions.

```
SELECT TAB1.EMPNO, TAB1.SALARY, TAB2.AVGSAL, TAB2.HDCOUNT
FROM
  (SELECT EMPNO, SALARY, WORKDEPT
   FROM DSN8610.EMP
   WHERE JOB='SALESREP') AS TAB1
LEFT OUTER JOIN
  (SELECT AVG(SALARY) AS AVGSAL, COUNT(*) AS HDCOUNT,
   WORKDEPT
   FROM DSN8610.EMP
   GROUP BY WORKDEPT) AS TAB2
ON TAB1.WORKDEPT = TAB2.WORKDEPT;
```

The first table expression selects the salary and department information for all of the sales representatives. Assuming there is an index on the JOB column, there would probably be very good index access to the table, given the simple equality predicate. The second table expression gets the average salary and headcount for all of the departments. This would result in a complete table scan of the employee table, even though we only need these figures for the departments, which have sales representatives. We can solve this problem by correlating the second table expression to the first:

```
SELECT TAB1.EMPNO, TAB1.SALARY, TAB2.AVGSAL, TAB2.HDCOUNT
```

```

FROM DSN8610.EMP TAB1
    ,TABLE (SELECT AVG(SALARY) AS AVGSAL, COUNT(*) AS HDCOUNT
           FROM DSN8610.EMP
           WHERE WORKDEPT = TAB1.WORKDEPT) AS TAB2
WHERE TAB1.JOB = 'SALESREP';

```

This query produces the same result as the previous query, but if there is an index on the WORKDEPT column, the optimizer is likely to pick it based upon the predicate in the table expression. This can result in a dramatic improvement in query performance. In the previous query, you can expect a merge scan join, and a table scan on the EMP table to satisfy the second table expression. In contrast, this query should result in a nested loop join, using an index on the WORKDEPT column (assuming one exists). The TABLE keyword is required in order use the correlated reference. You can code this query on mainframe (V6+), and non-mainframe platforms. Correlated table expressions such as these have been used quite extensively in the billing application, resulting in fantastic transaction performance involve complex processes.

Recursive SQL

A common table expression can be self-reference, resulting in a recursive process. This type of query is highly effective when querying self-referencing tables **[Figure 8]**, or circular references **[Figure 9]**.

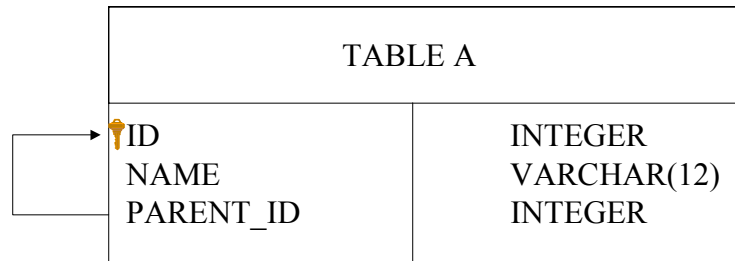


Figure 8: A Self Referencing Table

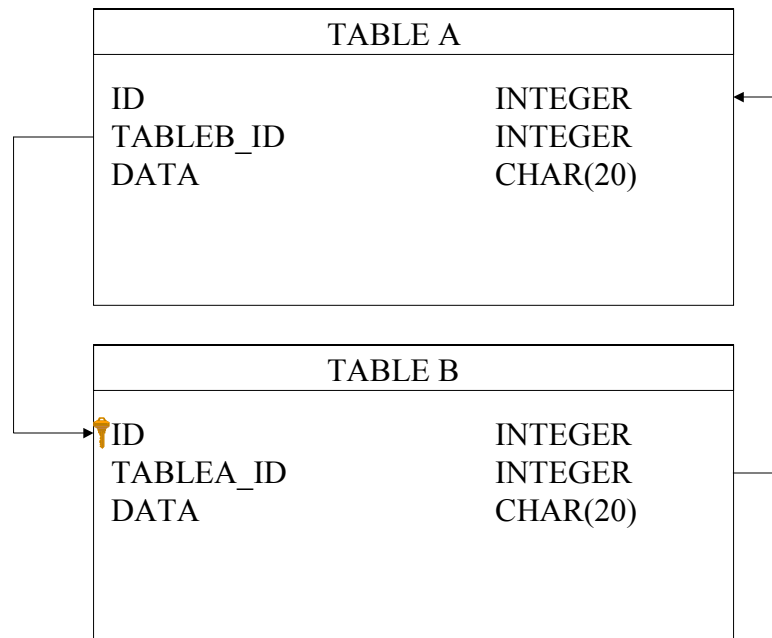


Figure 9: A Circular Relationship

A classic example is a company organizational chart. An org chart, which represents a hierarchical relationship, can be stored in a single table [fig].

The following example shows how a recursive query can be used to traverse a company org chart [Figure 10].

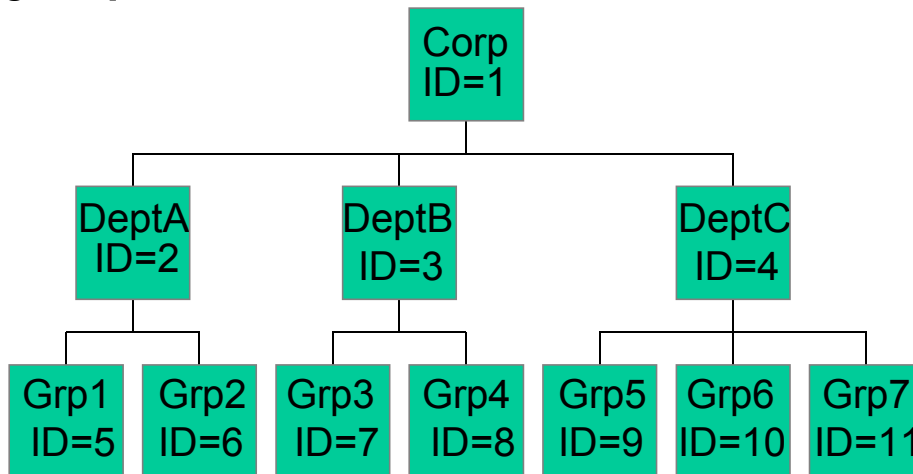


Fig 10: A Company Org. Chart

Each entry in the table [Figure 11] has a name, the identifier, and the identifier of the parent entry in the organization.

| ORG_CHART | |
|-----------|-------------|
| ID | INTEGER |
| NAME | VARCHAR(12) |
| PARENT_ID | INTEGER |

Figure 11: The Company Org. Chart Table

The root of the org chart has a null value for the parent identifier. By allowing a reference to the table defined by the WITH clause, within the statement itself, a recursive reference is defined. In this particular example, we traverse our org chart beginning with the department whose id is 5, and continuing up the hierarchy until there are no more parents (parent id is null).

```

WITH ROLLUP (NAME, ID, PARENT_ID) AS
  ((SELECT NAME, ID, PARENT_ID
    FROM ORG_CHART
    WHERE ID= 5)
  UNION ALL
  (SELECT X.NAME, X.ID, X.PARENT_ID
    FROM ROLLUP A, ORG_CHART X
    WHERE A.PARENT_ID= X.ID
    AND A.PARENT_ID IS NOT NULL))
SELECT NAME, ID
FROM ROLLUP;

```

The first half of the union is invoked once (initial query), and the lower part is run until the parent id is null (stop predicate). The final portion of the query selects the results from the temporary table called ROLLUP. In the case, the answer set is:

| NAME | ID |
|------|-------|
| 5 | Grp1 |
| 2 | DeptA |
| 1 | Corp |

In actuality, the stop predicate is not necessary, since the query will naturally stop when part identifiers are no longer found (null PARENT_ID for “Corp”). This query is not yet possible on the mainframe.

Superquery

Back to my billing application...Once we solved all of our complex transaction performance problem, and the day to day processes ran quickly, we moved on to the end of month processing. After all of the transactions for a given month were processed, the customers needed to be invoiced. The invoicing process read the customer table, and did look-ups in several

of the transaction summary tables to see which items to include on the invoice. The problem with this was that of the 400,000 customers in the system, only about 125,000 were active. Millions upon millions of queries were running, eating up CPU, and returning no data. The invoicing process would take several days to run, which was completely unacceptable. DB2 to the rescue!!!

Each vital query from the invoicing process was placed into a table expression, and joined. This single query, called the Activity Query, collected all the customer data for active customers in a single statement. This replaced millions of statements, and days of elapsed time, and returned a result in about 3 minutes. No tuning required; no problems with join methods; this is what UDB was designed for.

The application benefited greatly from the Activity Query, but there was another problem. Customers were organized into groups, and these groups required consolidated invoices. To complicate matters, groups were grouped into groups (like a hierarchical org chart). The invoicing application, which now processed the customers quickly, took days to process the same information for groups. Since the organization of the groups was defined in the database by a circular reference, with each group being “owned” by a customer [Figure 12], recursive SQL could be utilized.

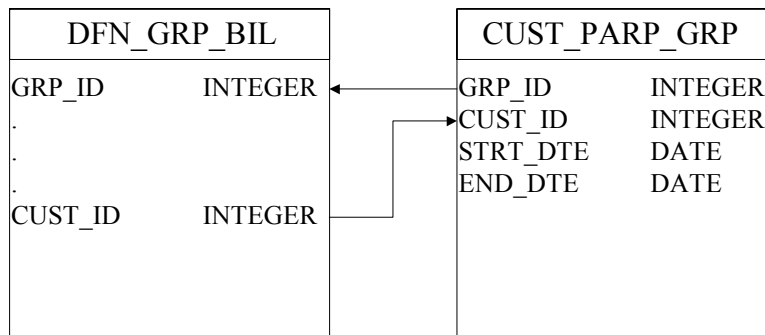


Fig 12: The Customer Group Tables

The Activity query was placed into a common table expression called ACTIVITY. This table expression was read as the first part of a second table expression call GROUPS. The GROUPS table expression was recursive:

```

with activity(cust_id, tot_recv_amt, tot_glflt_fee_amt,
             tot_flat_fee_amt, tot_tax_amt,
             prod_a_flg, prod_b_flg) as
(
select e.cust_id, gl_amt, glflt_rte_amt,
       custflt_rte_amt, cust_tax_amt, prod_a_flg, prod_b_flg
from
  (select c.cust_id, gl_amt, glflt_rte_amt,
          custflt_rte_amt
   from
     (select a.cust_id, gl_amt, glflt_rte_amt
      from
        (select cust_id, sum(post_amt) gl_amt
         from   qual.gl_item_dtl
         where  bus_proc_cde = '002'
        )
      )
  )
)

```

```

        group by cust_id) as a
    left outer join
        (select cust_id, sum(tot_amt)
         gl_flt_rte_amt
         from qual.cust_prdt_prc_summ
         where (prc_typ_cde = 'FR' or
              (prc_typ_cde = 'FT' and tier_num = 1))
         group by cust_id) as b
    on a.cust_id = b.cust_id) as c
    left outer join
        (select cust_id, sum(flat_fee_amt)
         cust_flt_rte_amt
         from qual.cust_flt_prc_summ
         group by cust_id) as d
    on c.cust_id = d.cust_id) as e
    left outer join
        (select cust_id, sum(tot_tax_amt) cust_tax_amt
         from qual.cust_prdt_prc_tax
         group by cust_id) as f
    on e.cust_id = f.cust_id
    left outer join
        (select s.cust_id,
         cast(max( case when coalesce(t.cmpnt_prdt_cde,
         s.prdt_cde) = '07113'
         then 'Y' else 'N' end) as char(1)) as
prod_a_flg,
         cast(max( case when coalesce(t.cmpnt_prdt_cde,
         s.prdt_cde)
         in ('06500', '06510', '06520', '06530')
         then 'Y' else 'N' end) as char(1)) as
prod_b_flg
         from qual.cust_prdt_prc_summ s
         left outer join
             qual.cust_prc_summ_cmpt t
         on s.cust_id = t.cust_id
         and s.file_own_bur_num = t.file_own_bur_num
         and s.prc_id = t.prc_id
         and s.char_prfl_grp_id = t.char_prfl_grp_id
         and s.tier_num = t.tier_num
         group by s.cust_id) as q
    on q.cust_id = e.cust_id
),
groups(cust_id, tot_recv_amt, tot_gl_flt_fee_amt,
       tot_flat_fee_amt, tot_tax_amt,
       prod_a_flg, prod_b_flg, level) as
((select cust_id, tot_recv_amt, tot_gl_flt_fee_amt,
       tot_flat_fee_amt, tot_tax_amt,
       prod_a_flg, prod_b_flg, 0
       from activity
       )
)

```

```

union all
(select c.cust_id, a.tot_recv_amt, a.tot_glflt_fee_amt,
      a.tot_flat_fee_amt, a.tot_tax_amt,
      a.prod_a_flg, a.prod_b_flg, a.level + 1
 from   groups a,
      qual.cust_parp_grp_bil b,
      qual.dfn_grp_bil c
 where  a.cust_id = b.cust_id
 and    b.grp_id  = c.grp_id
 and    b.strt_dte <= (select end_dte
                       from   qual.bil_cycle
                       where  bil_cycl_sts_cde = 'C')
 and    b.end_dte  >= (select end_dte
                       from   qual.bil_cycle
                       where  bil_cycl_sts_cde = 'C')
))
select cust_id, sum(tot_recv_amt) as tot_recv_amt,
      sum(tot_glflt_fee_amt) as tot_glflt_fee_amt,
      sum(tot_flat_fee_amt) as tot_flat_fee_amt,
      sum(tot_tax_amt) as tot_tax_amt,
      max(prod_a_flg) as prod_a_flg,
      max(prod_b_flg) as prod_b_flg
 from   groups
 group by cust_id
 order by cust_id;

```

This query rolls all of the customer data, from the ACTIVITY common table expression, up the group hierarchy. Customer data is rolled into groups, and group data is rolled up to further groups. The final piece of the query adds all of the data together for each customer, and group, and the result is all the information required for page 1 of an invoice. Days of processing now ran in a single DB2 statement, processing 10 of millions of rows of data, and it runs in about 5 minutes. AMAZING!

In Short Summary

UDB nested table expressions, and common table expressions, are the key to code the most powerful and efficient SQL statements possible. They can eliminate days upon days of application coding, dramatically improve performance, and fully utilize the database engine. REALLY COOL!

The Challenge

While assembling this article, I tried to think of usefulness for the drink portion of the article beyond simple party tricks. While I was studying recursive SQL, I ran across a lot of query examples that processed a bill of materials table [Figures 13, 14].

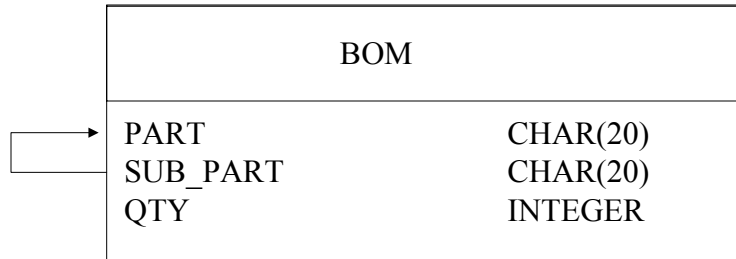


Fig 13: The BOM Table

| Part | Subpart | Qty |
|-------------|----------------|------------|
| EYEGGLASS | LENS | 2 |
| EYEGGLASS | FRAME | 1 |
| FRAME | NOSEREST | 2 |
| FRAME | HINGE | 2 |
| FRAME | CENTERPIECE | 1 |
| FRAME | EARREST | 2 |
| FRAME | SCREW | 2 |
| NOSEREST | SCREW | 1 |
| HINGE | SCREW | 1 |
| HINGE | HINGEPLATE | 1 |

Figure 14: The BOM Table Contents

Every example I saw would take a part, and find the parts that were required to make a part:

```

WITH GLASSPARTS (SUBPART, QTY) AS
  ((SELECT SUBPART, 0 + QTY
    FROM BOM
    WHERE PART='EYEGGLASS')
UNION ALL
  (SELECT A.SUBPART, A.QTY * B.QTY
    FROM BOM A, GLASSPARTS B
    WHERE A.SUBPART = B.PART))
SELECT SUBPART, SUM(QTY)
FROM GLASSPARTS
GROUP BY SUBPART;

```

This was pretty exciting, but I couldn't find an example that processed the table in the opposite direction. In other words, like my drink queries, lets find the thing (in this case, a part) by its parts. In other words, lets suppose that I have a collection of parts. Given the parts that I have, what can I make?

Question: I have 8 lens, 20 noserests, 5 centerpieces, 80 screws, 27 hingeplates, and 15 earrests; What can I make?

You should have all the information you need in this article to find the answer, if its possible (I don't know, as I haven't solved it yet myself). The query should be a single statement that one can embed as a static statement in a program. Please email all attempts to info@YLASSOC.COM. I'll post my attempt by the end of August. Cheers!