

2-6 October, 2006
Hilton Vienna
Vienna, Austria

F07 & F08
Utilizing DB2 V8 Table Expressions

IDUG® 2006
Europe

Daniel L. Luksetich
YL&A

03, October, 2006 • 3:15 PM - 4:15 PM
03, October, 2006 • 4:45 PM - 5:45 PM

Platform: Cross Platform

GoFurther



Dan Luksetich is a senior DB2 for YL&A, a consulting firm based in the United States. He works as a DBA, application architect, presenter, author, and teacher. Dan has been in the information technology business for over 21 years, and has worked with DB2 for over 16 years. He has been a COBOL and BAL programmer, DB2 system programmer, DB2 DBA, and DB2 application architect. His experience includes major implementations on z/OS, AIX, and Linux environments.

Dan's experience includes:

- Application design
- Database administration
- Complex SQL
- SQL tuning
- DB2 performance audits
- Replication
- Disaster recovery
- Stored procedures, UDFs, and triggers

Dan works 8-16 hours a day, everyday, on some of the largest and most complex DB2 implementations in the world. He is a certified DB2 DBA and application developer, and the author of several DB2 related articles.

Abstract

This presentation focuses on DB2 table expressions across all platforms, their usefulness and performance implications. It also contains a very inspirational story and some real world examples of how table expressions were used to replace application programs and save customers considerable resources. Included are some examples of some amazingly complex SQL statements, as well as some that improved SQL application response times from days to minutes or seconds! Wild and true stories help inspire the participant to "Do it all" with DB2!

2

- Platform: Cross Platform
- Objective1: Table Expression Basics
- Objective2: The Latest Table Expressions with V8
- Objective3: Table Expression Performance Issues
- Objective4: Achieving High Performance with Table Expressions
- Objective5: Stories of Real Life Table Expression Usage

Disclaimer PLEASE READ THE FOLLOWING NOTICE

- The information contained in this presentation is based on techniques, algorithms, and documentation published by the several authors and companies, and in addition is the result of research. It is therefore subject to change at any time without notice or warning.
- The information contained in this presentation has not been submitted to any formal tests or review and is distributed on an “As is” basis without any warranty, either expressed or implied.
- The use of this information or the implementation of any of these techniques is a client responsibility and depends on the client’s ability to evaluate and integrate them into the client’s operational environment.
- While each item may have been reviewed for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere.
- Clients attempting to adapt these techniques to their own environments do so at their own risks.
- Foils, handouts, and additional materials distributed as part of this presentation or seminar should be reviewed in their entirety.

3

DB2 is a registered trademark of IBM Corporation.

Resources include DB2 manuals, Q/A with IBM as well as assorted user presentations, and rigorous testing.

It might be reassuring to know that in spite of this disclaimer, the information provided was obtained primarily from real world experience, and tests that I conducted myself. This is not a theory, idea, or concept. This is the result of work that I performed solving various user problems. Lots of planning and considerations were made, and several theories and pieces of the code were tested, and only the ideas that worked are presented here.

No animals were harmed during testing.

Thank You!

- Susan Lawson, YL&A
- Richard Yevich (RIP)
- Don Chamberlin, IBM
- Richard Fazio, TransUnion
- Nicole Bergman, Social Security Administration
- Steve Boeckman, Social Security Administration
- Dale Woolridge, Ethoca
- Terry Purcell, IBM
- Steve Clark, TransUnion

4

Friendly cooperation, mutual respect, lack of pride, fearlessness, and willingness to learn are attributes of a talented and successful person.

These people have all helped me do my job, and their efforts are represented in various places within this presentation.

Objective: Learn About Table Expressions

- Table Expression Basics
 - Scalar Subqueries
 - Common Table Expressions
 - Nested Table Expressions
 - Correlated Table Expressions
 - Recursion
 - Table User-Defined Functions
- Learn When it's Appropriate to Use Table Expressions
- Getting Program Solutions in a Single SQL Statement
- Demonstrate Performance Advantages When Using Table Expressions
- Demonstrate Extreme SQL Power
- Inspire!?.....Terrify?!

5

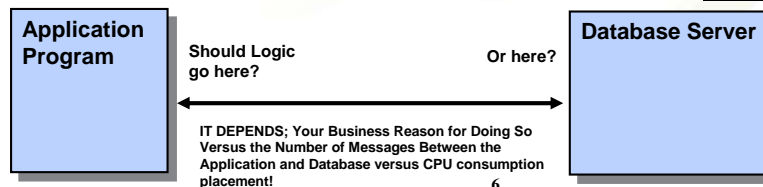
SQL has evolved into a very powerful programming language. The application development environment has evolved into a multi-platform flexible environment. Its expensive to move data around. So, if I can process that data, and only return a result from the server to the client, then I should be able to save time and money. I can also be more flexible.

Table expressions help us to do this. They enable us to process more of the data at the server. They allow us to add advanced programming features to our SQL statements:

- Program calls within programs
- Loop processing
- Cross references across tables
- Processing data multiple times

A Word on Complex Database Objects

- You Add Complexity to the Database for a Reason
 - Centralized Logic (Reusable Code)
 - Faster Time to Delivery (Easier to Program)
 - Performance
 - Sometimes!
- Complex Objects Include
 - Database Enforced RI and Check Constraints
 - Complex SQL
 - User-Defined Functions
 - Triggers
 - Stored Procedures



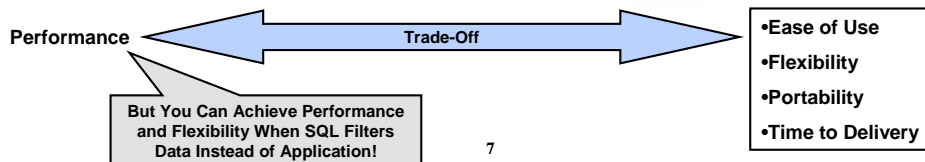
Why do we add complexity to our database, and what is our cost. There are many advanced features of DB2 that allow us to move complex application logic into the database engine. Why?

Pushing this logic into the database is important for object and code reuse, and centralization of the code. It removes dependency from client code, and pushes it to the server. This makes change management easier. Instead of changing thousands of clients you only need to change the server. Also, in many cases the implementation of a feature in the database can happen faster than in application code. This improves time to market, as well as the manageability of that feature.

There are always trade-offs with your choice of implementation, and a lot of times you are trading things like flexibility, manageability, security, and time to market for performance. Sometimes, though, you can get it all!

Advanced SQL Trade-Offs

- Complex SQL
 - Performance Improvement
 - Data is Filtered or Aggregated
 - Transactions Process Little Data
 - One Large Query Instead of Many Small Queries
 - Business Rules Pushed Towards Data
 - Set Processing or Filtering
 - Little or No Logic in SELECT List
 - Avoid UDFs, CASE Expressions, Data Conversions
- Flexibility
 - SQL is Highly Portable – IT RUNS ON THE SERVER!
 - Relatively Easy to Code – Fast Time to Delivery



Complex SQL has its place in application development, and there are trade-offs. Complex SQL allows us to push enormous data driven complexity into the database engine. If that logic is filtering data then that's exactly where that logic should be for performance reasons. If the complex SQL is processing data (e.g. functions in SELECT), and not filtering data then that could be a performance detriment.

Besides performance we are pushing logic into the SQL statements because it's easier and faster to code. It's extremely portable and flexible. I can write a SQL statement on the mainframe, say in SPUFI. Then I can put that statement anywhere; in a COBOL program, in Java, into a web server, and place on the network. That's because SQL always runs on the server. SO, the more my SQL statement does the less the application invoking it has to do, and the more flexible and portable my application becomes. This is extremely important in a dynamic world!

What is a Table Expression?

- Any SQL Statement Within Another SQL Statement
 - SQL That Lies Between Two Parenthesis
 - The Result of Any SQL Statement is a Table
- Why Use Table Expressions?
 - Push Program Logic Towards the Data
 - Use SQL as a Programming Language
- Why is This Important?
 - Flexibility
 - Reusability
 - Portability
 - Performance

8

The input to a SQL statement is tables. The output of a SQL statement is a table. Therefore, the output of any SQL statement can be the input to another SQL statement. Wrap a SQL statement in parenthesis, give it a name, and you've got another table you can access within an outer SQL statement. This will allow you to create really important and complex statements. There is really no limit. Programs that are thousands of lines in length have been rewritten as SQL statements that are hundreds of lines in length!

Simple Non-Correlated Subquery Example

All Platforms; DB2 for z/OS
and DB2 for LUW

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

Complete table expression inside of a
predicate, surrounded by parenthesis

9

A subquery is a type of table expression that most of us are already familiar with.

In this example a SQL statement is used as part of an “IN” expression. The results of the subquery are used in the predicate “WHERE TABNAME IN” when processing data from the SYSCAT.INDEXES table.

Processing of a Non-Correlated Subquery

- A Non-Correlated Subquery is Processed First as a Stage 1 Predicate in a “Bottom-Up” Fashion (Most of the Time, but not Always)
- A Non-Correlated Subquery can be Transformed into a Join by DB2 if the Number of Rows Returned is not Impacted

```
SELECT *  
FROM EMPLOYEE  
WHERE WORKDEPT IN  
  (SELECT DEPTNO  
   FROM PROJECT  
   WHERE PROJNO LIKE 'OP%')
```

**Non-Correlated:
bottom - up!**

10

Non-Correlated subqueries are processed typically in a bottom-up fashion. That is, the subquery is processed before the outer query, and the results of the subquery are available as input to the predicate in a stage 1 (sargable) process.

In some situations where the query is ambiguous, contains a row expression, or the subquery is sorted then the predicate can be transformed into a stage 2 (residual) process.

DB2 can also rewrite subqueries as joins if the number of rows returned is not impacted.

Row Expressions

Multiple columns referenced
on the left side of the IN

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

All Platforms; DB2 for
z/OS and DB2 for LUW

The multiple columns match the select
list in the subquery. Index access is
possible.

11

This example of using a subquery in a predicate is very much like the previous example, except for the fact that instead of searching for a single column, we use something called a “row expression”. Here a combination of the two columns in the expression, TABSCHEMA and TABNAME, are compared to the same combination in the subquery. In other words, we use the result of the table expression, row by row, in our comparison. For that reason, its called a “row expression”.

Using a Row Expression in a Range Query

- Here's a Nice Batch Cursor That Uses a Boolean Term Predicate to Get One Column Matching Index Access

```
SELECT [columns]
FROM V3FIXHDR
WHERE (SEG >= :LOW-SEG
      AND NOT
        (SEG = :LOW-SEG AND SUBSEG < :LOW-SUBSEG))
      AND (SEG <= :HIGH-SEG
      AND NOT
        (SEG = :HIGH-SEG AND SUBSEG > :HIGH-SUBSEG))
ORDER BY SEG, SUBSEG, CLMSSN, REC_ESTBT_DT1
FOR FETCH ONLY
```

Here's a great example of a query that has to scan a range of data based upon the value of two columns two columns. It uses a Boolean term predicate in order to get a match on one column of the supporting index. However, the query might have to unnecessarily scan extra index pages due to the fact that it only matched on one of the columns of the index, and has to screen on the other.

Using a Row Expression in a Range Query

- This is the Same Query Using a Row Expression to Get Two Column Matching Index Access

```
SELECT [columns]
FROM    V3FIXHDR
WHERE (SEG, SUBSEG) IN
      (SELECT SEG, SUBSEG
       FROM RANGE_TABLE
       WHERE (SEG >= :LOW-SEG
             AND NOT
             (SEG = :LOW-SEG AND SUBSEG < :LOW-SUBSEG))
             AND (SEG <= :HIGH-SEG
             AND NOT
             (SEG = :HIGH-SEG AND SUBSEG > :HIGH-SUBSEG)))
ORDER BY SEG, SUBSEG, CLMSSN, REC_ESTBT_DT1
FOR FETCH ONLY
```

This is the same query as the previous slide. However, the range predicate processing has been moved to code table that contains all possible values. Now the query benefits from greatly improved performance with two column matching index access. You can also use a statement that selects all the input values from SYSIBM.SYSDUMMY1.

Row Expression Processing

- Row Expression Processing by DB2 Varies
 - Sometimes It's a Stage 1 (Sargable) Process with Potential for Matching Index Access
 - Sometimes It's a Stage 2 (Residual) Process
- It Depends On the Subquery and Indexes
 - Any Non-Matching Row Expression is Stage 2

Scalar Subquery

All Platforms; DB2 for z/OS
and DB2 for LUW

```
SELECT *  
FROM SYSCAT.TABLES  
WHERE COLCOUNT =  
      (SELECT MAX(COLCOUNT)  
       FROM SYSCAT.TABLES);
```

A scalar subquery returns a table of one row and one column, or a single value.

A scalar subquery a single value only. That is, one row and one column are returned. Scalar subqueries can be extremely powerful tools when used in combination with correlated references, which we'll address next.

Scalar Subquery Versus Fetch First

- What's Better? A Scalar Subquery or "FETCH FIRST" With an ORDER BY?
 - Most Certainly It Depends on the Query, Tables, and Indexes!
- The Query Below is Functionally Equivalent to the Previous Query If I Just Wanted One Row

```
SELECT *  
FROM SYSCAT.TABLES  
ORDER BY COLCOUNT DESC  
FETCH FIRST 1 ROWS ONLY
```

16

For existence checking FETCH FIRST with ORDER BY might be an effective alternative to scalar non-correlated subqueries. The best advice is to try the two and compare access paths and operational performance.

Correlated Subquery

All Platforms; DB2 for z/OS and
DB2 for LUW

```

SELECT      *
FROM        SYSCAT.TABLES A
WHERE       EXISTS
            (SELECT      1
             FROM SYSCAT.INDEXES B
             WHERE       A.TABNAME = B.TABNAME
             AND         A.TABSCHEMA = B.TABSCHEMA);
  
```

Diagram annotations: A box labeled "Correlation names" has arrows pointing to the **A** and **B** in the query. Two arrows point from the text below to the **A** and **B** in the WHERE clause.

Columns from the outer table of the query (the table outside of the nested table expression or subquery) are referenced using the correlation name of the table

17

In a correlated subquery, references to an “outer” table, that is a table outside of the subquery can be made from inside the subquery.

This query is returning every table that has indexes. As the TABLES table is read, the subquery is checked for existing indexes, looking them up by the TABNAME and TABSCHEMA columns. Each table is given a correlation name (TABLES get “A”, and INDEXES gets “B”), which allows the optimizer to uniquely identify each column reference.

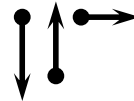
Processing of the Correlated Subquery

- A Correlated Subquery is Processed as a Stage 2 Predicate in a Top-Bottom-Top Manner
 - This is Because the Data Needed to Run the Subquery Has to Be Available
- The Outer Query is Executed, and For Each Row Processed the Subquery is Run
 - The Subquery Could Run Many Times!

```
SELECT *  
FROM EMPLOYEE E  
WHERE EMPNO =  
  (SELECT MGRNO  
   FROM DEPARTMENT D  
   WHERE D.DEPTNO = E.WORKDEPT)
```

18

**Correlated
Top-bottom-Top!**



A non-correlated subquery is a stage 2 process. This is because the data for the correlated reference has to be available when the subquery executes.

DB2 can run the correlated subquery for each qualifying row of the outer query. This can result in hundreds, or thousands, or millions of subquery executions per query.

Correlated Subquery, Non-Correlated Subquery, Or Join?

```
SELECT SNAME
FROM S
WHERE S# IN
(SELECT S# FROM SP
WHERE P# = 'P2')
```

```
SELECT DISTINCT SNAME
FROM S, SP
WHERE S.S# = SP.S#
AND SP.P# = 'P2'
```

```
SELECT SNAME
FROM S
WHERE EXISTS
(SELECT * FROM SP
WHERE SP.P# = 'P2'
AND SP.S# = S.S#)
```

It Depends on What!

Good: When there is no available index for inner select but there is on outer column (indexable). Or when there is no index on either inner or outer columns.

Good: When supporting indexes are available and most rows hook up in the join.

Good: When supporting index available on inner select and there is a cost benefit in reducing repeated executions to inner table and distinct sort for join.

Nested Table Expression

All Platforms; DB2 for z/OS
and DB2 for LUW

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

The “table” queried here is actually a table expression. The result of any SQL statement is a table!

The result table of the table expression is given a name via the AS clause. You can have many levels of nested table expressions.

20

Table expressions can be used to further process the result of a SQL statement.

UNION Everywhere

- From V7, UNION (ALL) May Appear Anywhere a Subselect or Table Expression was Allowed Previously:
 - CREATE VIEW
 - Nested Table Expressions (NTEs)
 - Predicates
 - INSERT INTO ... (subselect)
 - UPDATE ... SET COLA = (sql)
 - Declared Temporary Tables

All Platforms; DB2 for z/OS
and DB2 for LUW

```
SELECT *  
FROM TABLEA  
WHERE C1 IN  
  (SELECT C1  
   FROM TABLEB  
   UNION ALL  
   SELECT C1  
   FROM TABLE C)
```

21

UNION ALL is a powerful construct. We can now use UNION ALL in table expressions.

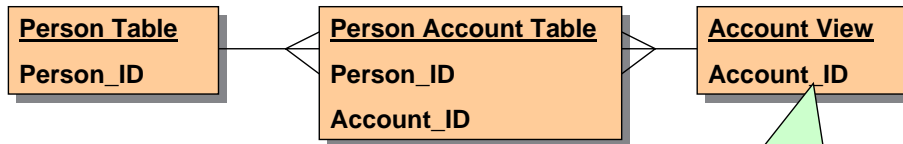
Limitations to Union in View

- Number of Underlying Tables Affects Query Complexity
 - More Tables in the View
 - More Query Blocks
 - Joins Add to Complexity
 - This Can Lead to SQLCODE –101
 - There is a Limit to the Predicate Distribution in a Join
 - If the Number of Join Tables Exceeds 225
 - Distribution Stops and Query Resorts to Materialized View
 - Query Block Pruning Only For
 - Literal Values
 - Host Variables (APAR PQ92434, More NOT to Come, call your IBM Rep)
 - NOT for Join Columns
 - But Predicate Transitive Closure Can Apply
 - If There is No Bind Time or Run Time Query Block Pruning
 - All Join Tables Will be Accessed for Each Query Block
 - EVEN if the Access is Redundant

22

We ended up limiting the number of tables UNIONED within a view or table expression. This limit was usually 8 tables.

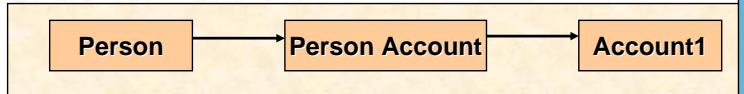
Redundant Table Access in Union in View



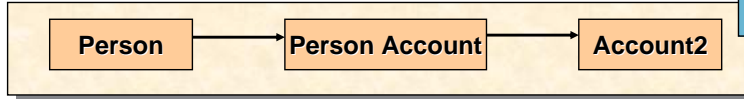
```
SELECT *
FROM PERSON A, PERSON_ACCOUNT B, ACCOUNT C
WHERE A.PERSON_ID = B.PERSON_ID
AND B.ACCOUNT_ID = C.ACCOUNT_ID
```

2 Table Union in View For Account_ID 1 Though 1000000, and 1000001 through 9999999

Query Block 1



Query Block 2

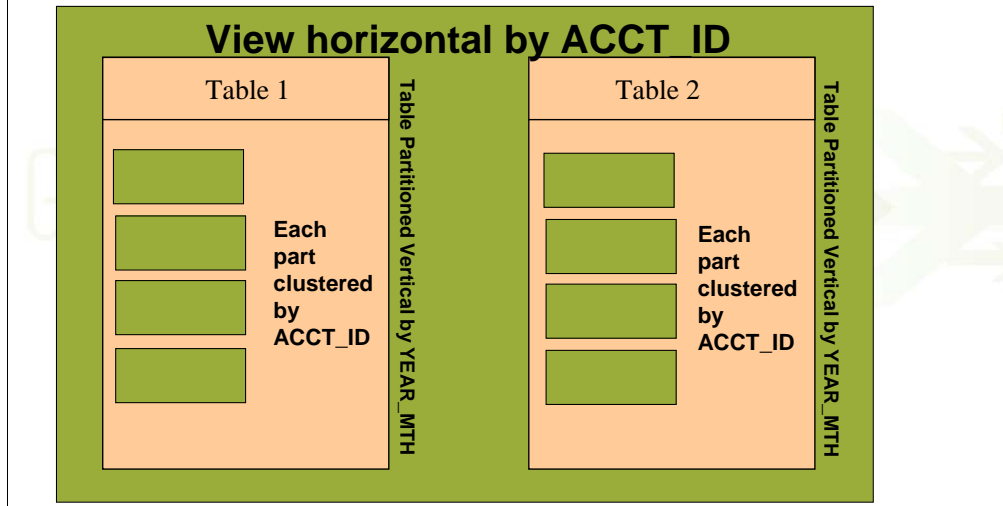


Same Rows in Person and Person Account Tables Accessed in Both Query Blocks!

Here I have a join. The join is distributed across multiple query blocks. Even though I have already read the person table in query block one, it will be read again in query block two.

Multi-Dimensional Clustering

- Multi-Dimensional Clustering is Built In to the DB2 for LUW Engine
 - But We Can Do It on z/OS with UNION Everywhere
 - You Need V8 and DPSI's To Really Make This Perform



This is a complex DB2 for z/OS design that takes UNION everywhere to the next extreme. This graphic is the image of a view. In side that view is two tables UNIONed. The data is distributed across the two tables by a certain column (ACCT_ID). Within each table the data is partitioned by another value (YEAR_MTH), and then within each part the data is indexed and clustered by the ACCT_ID.

While the application programs are responsible for making sure the data gets into the correct table, any queries can access the view as if it were a single table. DB2 will take care of pruning tables and partitions to make sure the only data accessed is the data your query needs (test, test, test, test ,test....did I mention you should test this?).

Table Specification Syntax

- SQL Standard Syntax for Naming Table Expressions
 - This is a REALLY Safe way to Explicitly Name your Nested Table Expressions!!!!
 - You Decide Where the Data is Coming From, and NOT DB2!
 - Avoid SQLCODE +012

All Platforms; DB2 for z/OS and DB2 for LUW

```
SELECT A.EMP_COUNT FROM
      (SELECT WORKDEPT, COUNT(*) AS EMP_COUNT
        , SUM(SALARY) AS SUM_SAL
       FROM EMPLOYEE
       GROUP BY WORKDEPT) AS A(WORKDEPT, EMP_COUNT, SUM_SAL)
WHERE A.EMP_COUNT > 5
```

```
SELECT * FROM DSN8710.EMP
WHERE WORKDEPT IN
      (SELECT WORKDEPT FROM
        (SELECT DEPTNO FROM DSN8710.PROJ
         UNION ALL
         SELECT WORKDEPT FROM DSN8710.EMP) AS XX) WITH UR;
```

Don't Give DB2 the Chance to Decide Where WORKDEPT Comes From. You May be Surprised!

25

Table specification is a SQL 99 standard. It allows you to properly name and table and all the columns of your table expression. I recommend this as a common practice for all of your table expressions. It is good documentation, clearly identifies your table expression, and helps avoid confusion by people and the DB2 optimizer!

Yes, the DB2 optimizer. When there is an ambiguous non-correlated reference in a SQL statement DB2 will try to resolve it. If it cannot resolved the ambiguous reference you will get a negative SQL code. However, if it does resolve the reference the statement will complete successfully with a SQLCODE +012 warning. This code basically said that DB2 found an ambiguous reference in the SQL statement, and made its best guess at resolving it. Yikes!

Don't leave the decision to DB2. Use table specification and correlation names in all references.

Scalar Fullselect Nested Table Expression

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

All Platforms; DB2 for z/OS
and DB2 for LUW

Scalar table expressions are embedded directly within the SELECT clause

26

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

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!

Scalar Fullselect to Avoid UDF Access?

- UDFs in a View are Invoked at Statement Execution Time
 - No Matter if the Columns is Referenced or Not
- A UDF in a Scalar Fullselect is Only executed When Referenced
 - Does This Work for Any Expression?

```
CREATE VIEW DRINKVIEW1 (COL1, COL2) AS
SELECT DRINK,
       INGREDIENT AS COL1,
       UDF1(DRINK) AS COL2
FROM   DRINKS;
```

```
SELECT COL1
FROM   DRINKVIEW1;
```

This SQL will not
invoke UDF1

```
CREATE VIEW DRINKVIEW2 (COL1, COL2) AS
SELECT DRINK,
       INGREDIENT AS COL1,
       (VALUES UDF1( DRINK)) AS COL2
FROM   DRINKS;
```

```
SELECT COL1
FROM   DRINKVIEW2;
```

27

This example, taken from a customer, shows two views. Both views are using a user-defined function to process a value from the table. However, in the second view the invocation of the UDF is embedded in a scalar fullselect table expression. This avoids UDF invocation if the column it provides is not reference in a SELECT statement against the view.

Correlated Table Expression

- Ability to Reference Across Tables
- Table Keyword in a Join Directs DB2 to Search Outside of the Table Expression for a Reference
- Useful Alternative to Outer Joins
- Encourages Index Access and Nested Loop Joins

28

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.

Correlated Nested Table Expression

```
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';
```

Correlation name

Correlated reference from inner to outer table

Correlation name

All Platforms; DB2 for z/OS and DB2 for LUW

29

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.

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.

Scalar Fullselect for Performance

- This is an Online Existence Check Query
- The Outer Table Returns Little Data, but the Inner Table has to be Materialized

```
select  distinct a.cust_id,  
        , b.Bundle_Flg  
from    cust_prdt_prc_summ a  
left outer join  
(select  distinct cust_id, 1 as Bundle_flg  
  from    cust_prc_summ_cmpt) b  
on a.cust_id = b.cust_id
```

30

This query performs poorly (30 seconds) for an online application. All it needs to do is return a flag if the customer has data present in the second table.

Scalar Fullselect for Performance

- This Improved Query will not Read the Entire Inner Table
- The Inner Table is Probed for Every Outer Row

```
select distinct a.cust_id,  
              (select max(cust_id)  
               from test.cust_prc_summ_cmpt b  
               where a.cust_id = b.cust_id) as  
Bundle_Flg  
from test.cust_prdt_prc_summ a ;
```

31

This query is subsecond, utilizing the index on cust_id, and avoiding materialization.

Table Expressions In Outer Joins

Determine the Employee Number and Salary of Sales Representatives, Along with the Average Salary of Their Departments

```
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;
```

All Platforms; DB2 for z/OS
and DB2 for LUW

32

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.

Same Solution as a Join to a Correlated Table Expression (Sideways Reference)

All Platforms; DB2 for z/OS
and DB2 for LUW

```
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';
```

DB2 for z/OS V7 APAR PQ66365

Z/OS ZPARM SARGSWRP (both V7 and V8)

33

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. Correlated table expressions such as these have been used quite extensively in the billing application, resulting in fantastic transaction performance involve complex processes.

Correlation for Performance!

- DB2 Engines Moving Toward Support of ERPs and Warehouses
- Query Rewrite; Blessing or Curse?
- Merge Scan Versus Nested Loop Join
- Correlation Encourages Index Access
- Correlation Encourages Nested Loop Join
- Want Subsecond Response? Nested Loop Join
- Processing Tons of Data? Merge Scan Join

34

In my opinion, the LUW 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.

My concept of merge scan versus nested loop is cross-platform. Also, a lot of the optimization and query rewrite features of the DB2 LUW engine are being incorporated into the mainframe DB2, so I've started to apply some of my LUW techniques up on the big iron!

Before Correlated Table Expressions

```
select      tab1.cust_id, tab1.prdt_cde,
            tab1.flat_fee_amt/tab2.totalproducts
from  (select a.thld_id, a.cust_id,
            a.prdt_cde,b.flat_fee_amt
        from  sub_rtl_prc_data a,
        custflt_prc_summ b
        where a.thld_id = b.thld_id
        ) as tab1 ,
      (select count(*) totalproducts, thld_id
        from  sub_rtl_prc_data C
        group by thld_id
        ) as tab2
where  tab1.thld_id = tab2.thld_id ;
```

Here is a more complicated query where two nested table expressions are being joined by the column THLD_ID.

This is a cross-platform query.

After Correlated Nested Table Expressions

```
select b.cust_id, b.prdt_cde,  
       a.flat_fee_amt / c.total_products  
from   custflt_prc_summ a,  
       sub_rtl_prc_data b,  
       table (select count(*) total_products  
              from   sub_rtl_prc_data  
              where  thld_id = a.thld_id  
              group by thld_id ) as c  
where  
       a.thld_id = b.thld_id;
```

36

This query produces the same results as the previous query, but is easier to read, and performs better than the first.

The two nested table expressions have been replaced by a more normal looking three table join, with two tables and a correlated nested table expression as the third table.

Using a correlated table reference, with the proper indexes, will probably result in a more efficient access path since DB2 is less likely to materialize result sets before joining.

This is a cross-platform query.

Before Performance Correlated Table Expression

```
select acct_num, tab2.bil_date, tab2.dollar_amt
from account a
left outer join
(select      bil_date, dollar_amt
 from      acct_hist b
 where     b.bil_date =
           (select      max(bil_date)
            from      acct_hist c
            where     c.acct_id = b.acct_id)) as tab2
on         a.acct_id = tab2.acct_id
```

37

In this situation, we desire the information for an account. We also want the most recent historical dollar amount, if that information exists.

This seems like a reasonable request, but there is a chance that the optimizer may not choose to use the provided index (first column is the acct_id of the acct_hist) for the “select max” subquery.

This is a cross-platform query.

After Performance Correlated Table Expression

```
select acct_num, tab2.bil_date, tab2.dollar_amt
from account a
left outer join
Table(select      bil_date, dollar_amt
from      acct_hist b
where     b.bil_date =
          (select  max(bil_date)
           from    acct_hist c
           where   c.acct_id = a.acct_id)) as tab2
on        a.acct_id = tab2.acct_id
```

38

By changing the correlated reference from the acct_hist table (the table with the correlation name “b”) to the account table (correlation name “a”) we may be able to influence the optimizer to an index that it was less likely to choose with the previous query.

This is a cross-platform query.

FETCH FIRST and ORDER BY

- FETCH FIRST and ORDER BY can be Placed in a Table Expression
- This Gives Us More Options to Tune Queries
- Could be an Alternative to Subqueries
 - Especially When Data Needs to be Returned

...

from Event E left join

**table (select ES.Event_ID as Event_ID, ES.Status as status,
ES.UPDATETIMESTAMP as UPDATETIMESTAMP
from EventStatus ES**

where ES.Event_ID = E.ID

order by ES.UPDATETIMESTAMP desc

fetch first 1 rows only) as ST

on ST.EVENT_ID = E.ID

where e.visibility != 3 and E.member_id in (?, ?, ?, ?, ?, ?);

DB2 for LUW

DB2 for z/OS version 9

Correlated Table Expressions Versus UNION Everywhere

- Quite Often Join Predicates Cannot be Distributed to a View
- This Can Result in the View Being Materialized
- This Example is a Simplified Version of an Audit Query
 - Table1 Holds Current Data and Table 2 Holds History

```
CREATE VIEW VIEW1 AS (  
SELECT COL1  
FROM TABLE1  
UNION ALL  
SELECT COL1  
FROM TABLE2)
```

```
SELECT <COLUMNS>  
FROM TABLE3 A  
LEFT JOIN  
VIEW1 B  
ON A.COL2 = B.COL2  
WHERE A.COL2 = 58673;
```

Correlated Table Expressions Versus UNION Everywhere

- Instead Use a Correlated Table Expression for Performance
 - In This Example You Get the Outer Join, and Index Access!

```
SELECT <COLUMNS>
FROM TABLE2 A
LEFT JOIN TABLE (SELECT COL1
                  FROM TABLE1 X
                  WHERE X.COL2=A.COL2
                  UNION ALL
                  SELECT COL1
                  FROM TABLE2 Y
                  WHERE X.COL2=A.COL2) AS B
ON A.COL2 = B.COL2
WHERE A.COL2 = 58673;
```

With DB2 for LUW you can put the correlated reference in the view using a table function!

Table Functions

- SQL Sourced and External Table Functions
 - A FUNCTION which returns a TABLE
 - Used like DB2 table in SQL FROM clause
- External Table Function
 - This is a User Written Program
 - Similar to a Stored Procedure
 - Allows data outside DB2 to be process by DB2 via SQL

42

A external table function is an external UDF that returns a table. This function call can be used wherever a table expression can be embedded in a SQL statement.

The table function accepts some special parameters to help control how the table is constructed. Otherwise, it is very similar to an external scalar UDF.

The TABLE keyword is used in a SQL statement that invokes a table UDF.

SQL Sourced Table UDFs

- **CREATE FUNCTION** <function name> (parameters)
RETURNS TABLE <column list> <options list>
 - Used to Define a User-Defined SQL Table Function
 - The CREATE FUNCTION Statement Contains the Source Code
 - Source Code is a SQL Statement
 - Function Returns a Table – It's a Parameterized View!!
 - Good for Transaction Processing Applications
- Specify the SQL statement in the RETURN

```
CREATE FUNCTION DANL.DEPT_INFO (IN_WORKDEPT CHAR(3))
RETURNS TABLE (AVGSAL DECIMAL(9,2) , HDCOUNT INTEGER)
LANGUAGE SQL
SPECIFIC DEPTINFO
NOT DETERMINISTIC
READS SQL DATA
RETURN SELECT AVG(SALARY), COUNT(*)
FROM DANL.EMPLOYEE
WHERE WORKDEPT = IN_WORKDEPT;
```

DB2 for LUW Only!

43

A SQL statement returns a table. You can place a sql statement in a SQL sourced table user-defined function, which I like to think of as parameterized views. This allows you to take common SQL statements, and make them available as table functions. The parameter input, and use of parameters in the SELECT statement coded in the RETURN clause allows control over the predicate applied in the statement. This allows for more rigid control over the statement than a view offers.

SQL sourced table UDFs are probably more appropriate for SQL statements that process little or no data. Because they are primarily used (or should be used) to limit the returned result via a predicate, they are best utilized for probing data, and thus best used in transaction based applications.

Invoking The SQL Source Table Function

DB2 for LUW Only!

- Table Keyword Used to Invoke the Function

```
SELECT *  
FROM TABLE(DANL.DEPT_INFO(CAST('E01' AS CHAR(3))))  
AS TAB1(AVGSAL, HDCOUNT)
```

Table Specification Names the Columns

```
SELECT TAB1.EMPNO, TAB1.SALARY,  
       TAB2.AVGSAL, TAB2.HDCOUNT  
FROM   DANL.EMPLOYEE TAB1  
       , TABLE(DANL.DEPT_INFO(TAB1.WORKDEPT)) AS TAB2  
WHERE  TAB1.JOB = 'SALESREP';
```

Column Names From Function Definition

Parameterized View! (same query as a few slides back)

44

You can reference your SQL sourced table function in any SQL statement by specifying the TABLE keyword. The function is otherwise used as any other table would be used. The SQL within the function is ultimately merged with the SQL in the outer statement.

External Table UDF

```
CREATE FUNCTION TBLFUNC()  
  RETURNS TABLE(COL1 INTEGER, COL2 INTEGER)  
  SPECIFIC TBLFUNC  
  EXTERNAL NAME 'TBLFUNC'  
  LANGUAGE C  
  PARAMETER STYLE DB2SQL  
  NOT DETERMINISTIC  
  FENCED  
  CALLED ON NULL INPUT  
  CONTAINS SQL  
  NO EXTERNAL ACTION  
  SCRATCHPAD 200  
  FINAL CALL  
  DBINFO  
  COLLID TEST  
  WLM ENVIRONMENT WLMENV2  
  STAY RESIDENT YES  
  PROGRAM TYPE SUB  
  SECURITY DB2  
  CARDINALITY 1 ;
```

All Platforms; DB2 for z/OS
and DB2 for LUW

DB2 for z/OS Example

The DDL for the definition of an external table UDF is very similar to that for an external scalar UDF, except for the RETURNS clause, which indicates that a table is to be returned.

Using an External Table Function UDF

```
SELECT COL1, TAB2.COLA
FROM FINANCETLB,
TABLE(TBLFUNC()) AS TAB2
WHERE ACCT_ID = 123
```

All Platforms; DB2 for z/OS
and DB2 for LUW

TABLE Keyword Identifies
a Table Expression, and
Table Function

Correlation Name Identifies
the Result "Table"

46

I can write an external program that returns a result set in the form of a DB2 table! Such programs can be written in a variety of languages, and need only to conform to a few simple rules, and handle parameters in a specific way to be DB2 user defined table functions.

In a SQL statement, the "TABLE" keyword is used to tell DB2 that the results of a function call will return a result set. The function is defined to DB2 via DDL, so the the query is away of the actual program to call, as well as the names of the columns it is returning, and the parameters that have to be passed to the function.

The results returned are an actual table, whose columns can be referenced within the SQL statement as if the data came from any other table.

Correlated Reference in a Table Function Call

```
SELECT  TAB2.COLA, TAB2.COLB, TAB1.COL1
FROM    (SELECT  COL1, COL2, COL3, COL4
        FROM    FINANCETLB
        WHERE   ACCT_ID = 123) AS TAB1
, TABLE (CALCUDF(TAB1.COL2,
                 TAB1.COL3, TAB1.COL4))
        AS TAB2
```

All Platforms; DB2 for z/OS
and DB2 for LUW

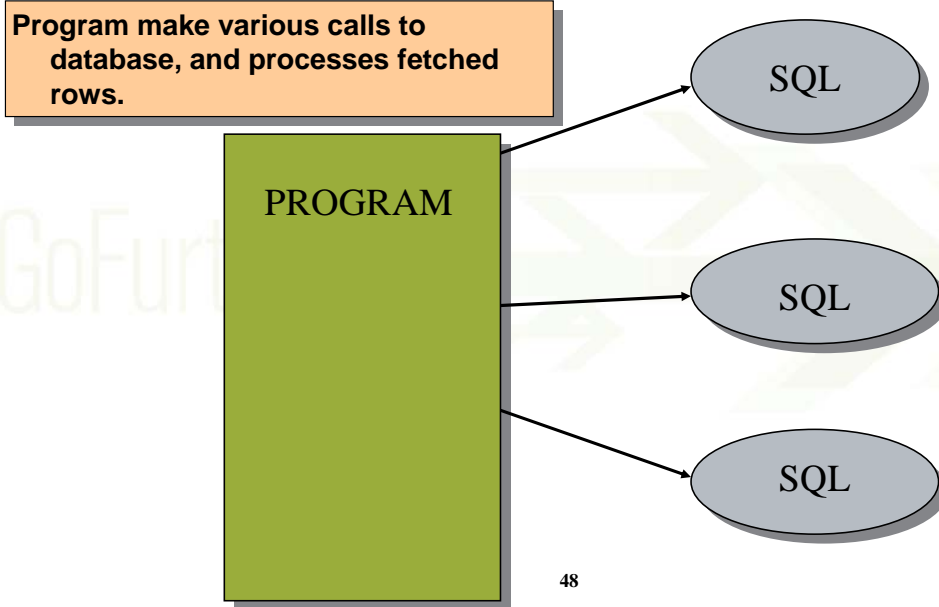
47

You can make a correlated reference from within the invocation of a table function. This allows you to send data into a table UDF (the same could be done for a scalar UDF in a correlated nested table expression) from another table.

This allows for some wonderful and amazingly sophisticated processing. You can basically take some of your current or legacy programs, and use this technique to embed them into SQL statements.

I know...I've done it!

Program Before Correlated Table UDF

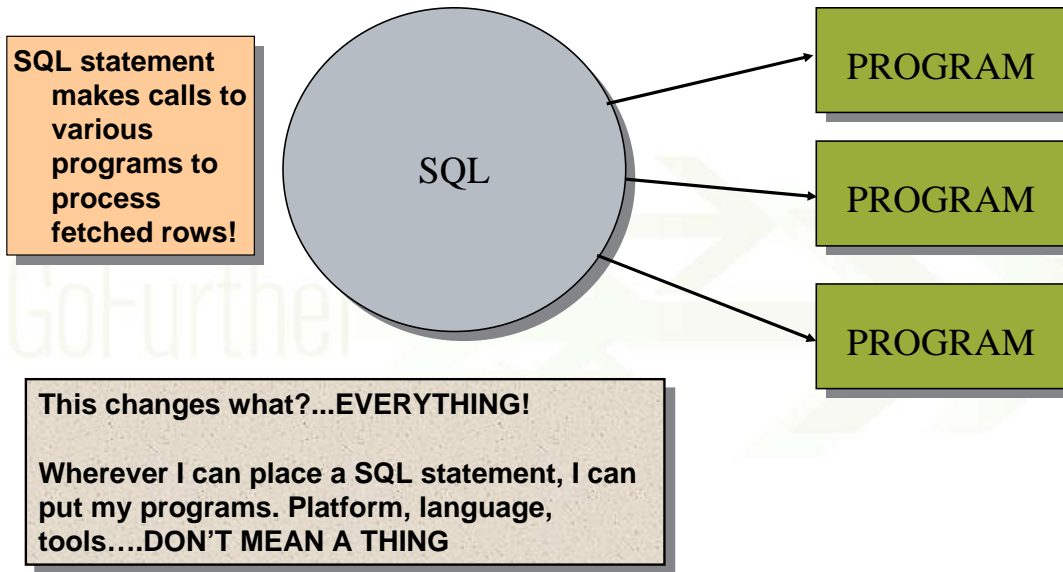


UDFs in this manner can change the world.

Take a look at our traditional database program paradigm. Here we have written a program, and that program may issue one or more SQL calls to our database.

While this program functions well, we may have to jump through some hoops in order to integrate it with our new WEB application.

Dare I Say “Paradigm Shift”?



49

Now I can take any new or legacy program, and turn it into a user-defined function. This allows me to do “anything” from within the context of a single SQL statement.

What does this change....EVERYTHING.

Because SQL is the most flexible language in the world, I can write SQL statements just about anywhere. From wherever I can access by database using SQL I can now run any program. I don't need a fancy web interface, CICS layers, or any other silly sort of gateway to my data. I simply run a SQL statement.

Imagine taking your boss' favorite legacy program, turning it into a UDF, setting it up to run under Excel, and putting an icon to invoke it on your boss' desktop. He's no longer bothering you to run that thing all the time, the data is piped right to a spreadsheet on his PC, and you're the hero that gets the big promotion!

Common Table Expression

WITH Clause Identifies the
Common Table Expression

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

All Platforms; DB2 for z/OS
and DB2 for LUW

Once the Table Expression
is Defined, it can be
Referenced in the Statement

50

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.

Here the results of the table expression "SELECT TABNAME FROM SYSCAT.TABLES" is placed in a common table called "TBLs". This common table expression can be used throughout the remainder of the statement. This example is quite simple, utilizing a simple query against the "TBLs" common table. However, it is quite easy to imagine just how powerful these expressions can be!

Common Table Expression

VALUES statement not supported on z/OS

All Platforms; DB2 for z/OS
and DB2 for LUW

```
WITH MYTABLE1 (COL1, COL2) AS
  (SELECT 'TESTCOL1', 'TESTCOL2' FROM
   SYSIBM.SYSDUMMY1)
SELECT COL1, COL2
FROM   MYTABLE1;
```

VALUES statement supported on LUW

DB2 for LUW Only!

```
WITH MYTABLE1 (COL1, COL2) AS
  (VALUES ('TESTCOL1', 'TESTCOL2'))
SELECT COL1, COL2
FROM   MYTABLE1;
```

51

This very simple example uses the SYSIBM.SYSDUMMY1 catalog table to generate some data within the MYTABLE1 common table expression, and then the subsequent fullselect within the SQL statement retrieves the data from that common table expression. Any valid fullselect can be placed within a common table expression, and any number of common table expressions can be specified within a SQL statement (perhaps limited by memory or the 225 maximum number of tables in a SELECT). This enables programmers to build impressively powerful SQL statements.

Use of Common Table Expression

- Useful for Building Intermediate Result Sets
 - Then Using Those Results Multiple Times
- Extreme Useful for Reprocessing Data in a Single Statement

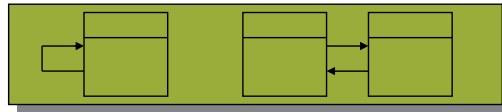
```
WITH ONHAND (INGREDIENT) AS
    (VALUES ('VODKA'), ('ORANGE JUICE'),
    ('PEACH SCHNAPPS'))
SELECT      DRINK
FROM        DRINKS A, ONHAND B
WHERE       A.INGREDIENT = B.INGREDIENT
GROUP      BY DRINK
HAVING COUNT(*) = (SELECT COUNT(*) FROM ONHAND);
```

52

In this example we are using relational division to determine which drinks contain vodka, orange juice, and peach schnapps. We build a table of these ingredients (ONHAND), and then reference that table twice in the query to get our results.

Introduction to Recursion

- Definition
 - When a function (or procedure) calls itself. Such a function is called "recursive".
 - factorial 0 = 1
 - factorial n = n * factorial (n-1)
 - If a fullselect of a common table expression contains a reference to itself in a FROM clause, the common table expression is a recursive common table expression.
 - More of a looping mechanism
- Queries using recursion are useful for
 - Traversing hierarchies or networks
 - Any design that contains a circular or self reference



53

In mathematics and programming, when a function or procedure calls itself it is referred to as being recursive. A common recursive function employed often today would be a function that calculates compound interest. In the programming world recursive functions perform functions faster and cheaper than their looping equivalents.

With DB2 SQL, when a fullselect of a common table expression contains a reference to itself in a FROM clause, that common table expression is considered recursive. Recursive queries are most useful for traversing hierarchies or networks. Recursive queries allow complex processes to be pushed from the application, and into the database. This allows the complex processes to be calculated before data is returned, and this can be enormously faster and cheaper than a programmatic equivalent.

Recursion

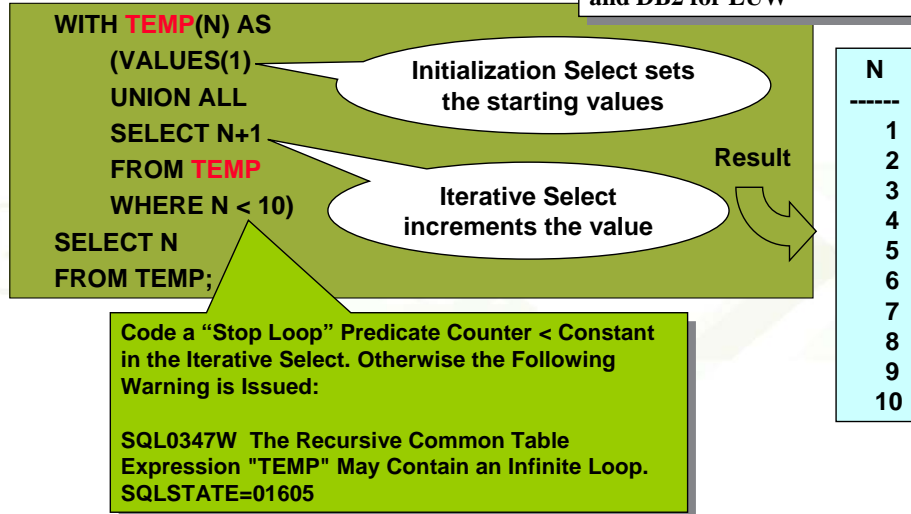
- Support of certain applications
 - Bill of materials
 - Reservation systems
 - Network planning
 - Grouping and Rolling Up Data Gathered in Transaction Systems
- Generating statements and data
 - Making Data for Testing
 - Making Statements for Testing

54

Recursion has a very specific place in SQL statements. I have used it for using parent-child relationship in a hierarchical fashion to traverse a tree structure. This is probably the best use of recursion within DB2.

Recursion – Very Simple Example

All Platforms; DB2 for z/OS
and DB2 for LUW



55

A common table expression is recursive when it contains a fullselect that references itself in a FROM clause. This self-reference creates an iterative process where the result of one iteration acts as input to the next. The SQL inside the common table expression takes the form of the union of two fullselects. Recursive SQL statements can result in infinite loops, and so care must be taken when constructing them so as to ensure that the recursion stops. Recursive SQL is best described with an example, and so the statement in here simply generates a result of a single integer column that increments in value from one to ten.

In this example, the WITH clause of the common table expression defines a table called "TEMP" with a single column called "N". The first fullselect of the union inside the common table expression simple generates a single row with the value 1 in it by using the VALUES statement or by selecting that value from the SYSIBM.SUSDUMMY1 catalog table. The fullselect in the second half of the union actually selects data from the common table expression "TEMP". This is know as a recursive reference, and selects data that was the result of the previous iteration in the cycle. This fullselect adds one to the value retrieved for the next iteration. The predicate "WHERE N < 10" is called a stop predicate because it causes the iterations to cease at a certain point. The final fullselect retrieves the data from the common table expression.

Because recursive SQL creates an iterative process that can result in an infinite loop, it is important that a stop predicate be included. In cases where DB2 cannot determine if a stop predicate exists it will issue a SQL warning code of +347.

Recursion Versus Looping

Classic Recursive Factorial Function

```
Factorial(n)
  if n = 0 return 1
  else return n * factorial(n-1)
end
```

- DB2 recursion is more of a looping method
- We've still got to play some games with explosion and selection to simulate true recursive functions

DB2 Recursive Factorial Function

```
CREATE FUNCTION FACTORIAL (N_IN INTEGER)
RETURNS INTEGER LANGUAGE SQL
DETERMINISTIC READS SQL DATA
RETURN
WITH FACTORIAL(N, CNT) AS
  (VALUES(N_IN, N_IN)
  UNION ALL
  SELECT A.N * (CASE WHEN A.CNT - 1 = 0
    THEN 1 ELSE A.CNT - 1 END), CNT - 1
  FROM FACTORIAL A
  WHERE A.CNT > 0)
SELECT N FROM FACTORIAL WHERE CNT = 0;
```

This is Also an Example of SQL Scalar UDF with an Embedded Table Expression (DB2 LUW Only). You Can Do This on z/OS, but Not in a UDF.

56

A recursive process is one that continually “folds in” on itself. That is, on each iteration a function call is stacked on top of the previous function call. The process iterates until the function finally returns a result at the inner most level. At that point function result being popping off the stack, and are returned to the previous caller in a “folding out” sort of way. A recursive SQL statement is actually a looping mechanism. Each iteration of the query has to produce results, that is it has to create rows of data in order for the recursion to continue. There is no “in and out” with a recursive query, only looping and producing results. Lots of times, in order to simulate true recursion, we have to track the looping process, and select only the results we want out from the initial, interim, and final data that the query produces.

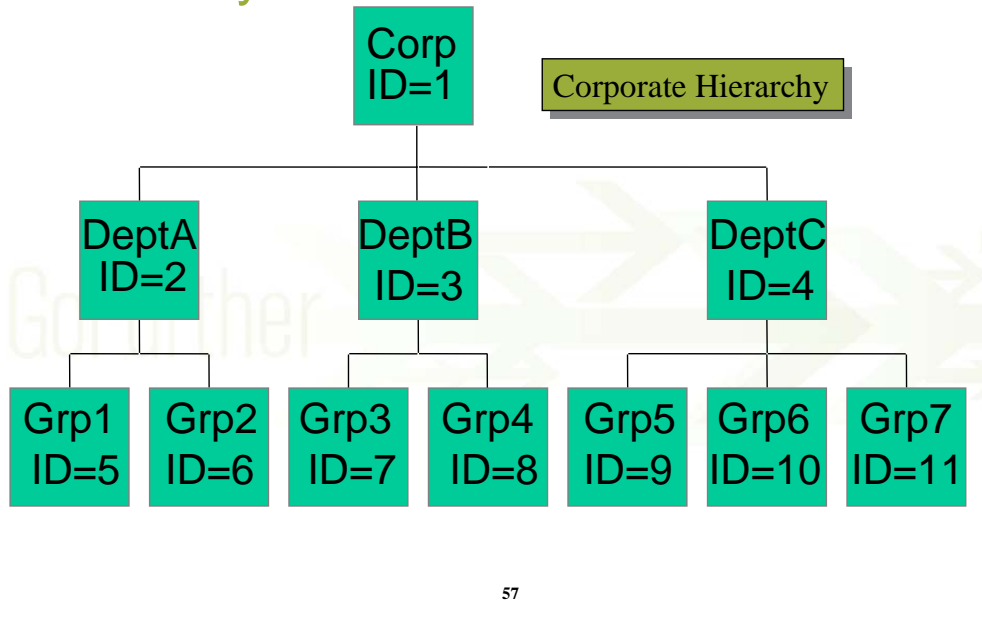
Take for example this *factorial* function. This is a great example of a recursive function in that it is a mathematically function that the human mind can typically calculate on its own. This function, when invoked, simply checks to see if the input data is zero, if it is then it returns the value of one. If it is not zero, then it runs a formula to multiply the number it received as input to the result of the factorial function with a parameter equal to the input value minus the number one. So $\text{factorial}(3) = 3 * \text{factorial}(2)$, $\text{factorial}(2) = 2 * \text{factorial}(1)$, $\text{factorial}(1) = 1 * \text{factorial}(0)$, and $\text{factorial}(0) = 1$. When “1” is returned, each result pops off the stack; $1 * 1 * 2 * 3 = 6$.

The recursive SQL statement has to loop, and produce results on each iteration. Therefore, the calculated value has to be passed to each level, and each level is returned as a result. The interim results for $\text{factorial}(3)$ are (as rows and columns of data):

(3, 3), (6, 2), (6, 1), (6, 0)

One final select then pulls the data we are interested in (the result) out of all the data generated.

Hierarchy



This particular example shows how you could use recursive SQL to traverse a corporate organization chart, which is a hierarchy.

Each position in the chart is labeled via a name and unique numeric identifier.

Hierarchy

Table definition of my organization hierarchy table

```
CREATE TABLE ORG_CHART  
(NAME          CHAR(30) NOT NULL,  
  ID           SMALLINT NOT NULL,  
  PARENT_ID    SMALLINT);
```

58

The ID column is the primary key, and the Parent_ID column is a foreign key to the ID column. Nodes within the hierarchy have a unique ID, and a Parent_ID that points to the next highest node in the hierarchy. The corporate node has no parent, and so its Parent_ID is set to NULL.

Hierarchy

```

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;

```

All Platforms; DB2 for z/OS
and DB2 for LUW

NAME	ID
-----	--
Group 1	5
Dept A	2
Corp	1

59

In this particular example, we traverse our org chart beginning with the organization component whose id is 5, and continuing up the hierarchy until there are no more parents (parent id is null). 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:

```

5 Group 1
2 Dept A
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").

Useful Recursion for the DBA

- Data Generation
 - Useful for test data creation
 - Can use the looping process to create lots of data
 - Even useful for DB2 V7 for z/OS and OS/390
 - Generate data on your PC to IXF
 - Import it to the mainframe
- Statement Generation
 - Can generate significant quantities of statements for test cases
 - Lots easier than keying it in!
 - Even useful for DB2 V7 for z/OS and OS/390
 - Generate statements on your PC
 - FTP them to the mainframe

Data Generation

- A look-up table was created to derive keys for a complex table design
- The look-up table was to contain 300,000 rows of data
- Not easy to type in!
- The following is an example of the look-up table for twenty values and 20 partitions

Combination of two key columns determines the partition

LOOK_UP_TBL	
<u>STALE_IND</u>	CHAR(1) NOT NULL
<u>KEYVAL</u>	SMALLINT NOT NULL
PART_NUM	SMALLINT NOT NULL

61

In one design it was determined that data should be spread across multiple partitions of a partitioned tablespace. Data was to be split according to its age (stale or not stale) and based upon the value of last position of its key. So, if the stale indicator was 'S' (for stale data) then the data will go into partitions 1 through 10 depending upon the value of the last position of the key (each partition corresponds to the last value of the key, e.g. '0' goes into partition 1, and '9' goes into partition 10). If the stale indicator is 'F' (for fresh data) then the data will go into partitions 11 through 20 depending upon the value of the last position of the key (e.g. '0' goes into partition 11, and '9' goes into partition 20). We can build a look-up table that holds all of the combinations of stale indicator and last position value.

- Recursive query to generate the data
- Place this in an EXPORT command or DSNTIAUL unload

```

WITH LASTPOS (KEYVAL) AS
  (VALUES (0)
   UNION ALL
   SELECT KEYVAL + 1
   FROM LASTPOS
   WHERE KEYVAL < 9)
,STALETBL (STALE_IND) AS
  (VALUES 'S', 'F')
SELECT STALE_IND, KEYVAL
, CASE STALE_IND WHEN 'S' THEN
  CASE KEYVAL WHEN 0 THEN 1
  WHEN 1 THEN 2 WHEN 2 THEN 3
  WHEN 3 THEN 4 WHEN 4 THEN 4
  WHEN 5 THEN 6 WHEN 6 THEN 7
  WHEN 7 THEN 8 WHEN 8 THEN 9
  WHEN 9 THEN 10 END
  WHEN 'F' THEN
  CASE KEYVAL WHEN 0 THEN 11
  WHEN 1 THEN 12 WHEN 2 THEN 13
  WHEN 3 THEN 14 WHEN 4 THEN 15
  WHEN 5 THEN 16 WHEN 6 THEN 17
  WHEN 7 THEN 18 WHEN 8 THEN 19
  WHEN 9 THEN 20 END
  END AS PART_NUM
FROM LASTPOS INNER JOIN
  STALETBL ON 1=1;

```

Recursive common table expression generates all key values

A second common table expression sets the values for stale or fresh data

A Cartesian join combines all the values, and the case expressions run the data transformation rules

We need to generate data to populate this table, and the this statement can be run on DB2 on our PC, its results exported into an IXF file, and subsequently imported into our mainframe table, or imported into a table on DB2 for LUW.

Although the example above generates 20 rows of data, the actual statement that was coded to generate data for a lookup table in a recent database I designed contained 300,000 rows. In situations like that generating the data with recursive SQL is a great time saver.

Statement Generation

- Some design situations call for small scale testing of SQL statements to prove a concept
- One situation: test an insert strategy for a high volume table
- Table was clustered by a date column and account identifier
- Need to test sequential versus random inserts on the table

ACCT_HIST_TBL	
<u>HIST_EFF_DTE</u>	DATE NOT NULL
<u>ACCT_ID</u>	DEC(11,0) NOT NULL

63

Some design situations call for small scale testing of SQL statements to prove a concept. In one recent situation I wanted to test an insert strategy for a high volume table. The table was clustered by a date column and account identifier, and I wanted to test sequential versus random inserts on the table.

Statement Generation

- Two queries to generate 50,000 statements
 - One sequential

```
WITH GENDATA (ACCT_ID, HIST_EFF_DTE) AS
(VALUES (CAST(1 AS DEC(11,0)), CAST('2004-02-01' AS DATE))
UNION ALL
SELECT ACCT_ID + 5, HIST_EFF_DTE
FROM GENDATA
WHERE ACCT_ID < 249996
)
SELECT 'INSERT INTO YLA.ACCT_HIST (ACCT_ID, HIST_EFF_DTE)' CONCAT
' VALUES(' CONCAT CHAR(ACCT_ID) CONCAT ',' CONCAT ''''
CONCAT CHAR(HIST_EFF_DTE,ISO) CONCAT '''' CONCAT ');'
FROM GENDATA ORDER BY ACCT_ID;
```

50,000 inserts in key sequence, incrementing the ACCT_ID from 1 by 5 with each statement

- One random

```
WITH GENDATA (ACCT_ID, HIST_EFF_DTE, ORDERVAL) AS
(VALUES (CAST(2 AS DEC(11,0)), CAST('2003-02-01' AS DATE), CAST(1 AS FLOAT))
UNION ALL
SELECT ACCT_ID + 5, HIST_EFF_DTE, RAND()
FROM GENDATA
WHERE ACCT_ID < 249997
)
SELECT 'INSERT INTO YLA.ACCT_HIST (ACCT_ID, HIST_EFF_DTE)' CONCAT
' VALUES(' CONCAT CHAR(ACCT_ID) CONCAT ',' CONCAT ''''
CONCAT CHAR(HIST_EFF_DTE,ISO) CONCAT '''' CONCAT ');'
FROM GENDATA ORDER BY ORDERVAL;
```

50,000 inserts in random order (by using a RAND function)

These statements allowed for some very important tests to be conducted with little programming effort!

More Table Expression Performance!

The Bad and the Good

Nested Table Expression Performance Gotcha's

- **DB2 will Merge Nested Table Expressions if it Can**
 - Inner (Nested) Expressions are Merged With Outer When Possible
- **This Can be a Performance Boost or a Bust**
 - Depends Also on if You Want Elapsed or CPU Performance

```
SELECT    SUM(CASE WHEN COL1=1 THEN 1 END) AS ACCT_CURR
          , SUM(CASE WHEN COL1>1 THEN 1 END) AS ACCT_LATE
FROM
(SELECT   CASE ACCT_RTE WHEN 'AA' THEN 1 WHEN 'BB'
          THEN 2 WHEN 'CC' THEN '2' WHEN 'DD' THEN 2
          WHEN 'EE' THEN 3 END AS COL1
FROM     YLA.ACCT_TABLE) AS TAB1
```

Applies to any
Expression,
Function, or UDF

- **If COL1 is Deterministic, Then it is Executed 2 Times**
- **If COL1 is Not Deterministic, it's Executed 1 Time**
 - BUT, TAB1 is Materialized First! I Add the RAND() Function Sometimes
- **This May Not be a Big Deal in this Example, but is HUGE for Complex Queries**
 - This Applies to all UDFs as Well, SQL Based or External!

SQL sourced UDFs can give you some performance surprises. In particular, defining a UDF as deterministic can affect performance.

A deterministic function is a function that returns the same value if the same values are passed to it, like a SUBSTR function. A non deterministic function is one that can return different results even if passed the same values, like a RAND function.

DB2 will merge the expression represented by a deterministic function if it is nested in a table expression, as long as the nested expression does not have to be materialized by DB2. A non deterministic function will force a nested table expression to be materialized. This can lead to a variety of SQL performance issues.

The perfect example of a non-deterministic function is the RAND() function. Place that into any table expression and it will materialize. We've used this to force materialization when it improved performance.

During Join Predicate and Search Performance

- Problem
 - Application needs to search on name reversals
 - E.G.
 - We need to search for “Christine Haas”
 - There’s a chance the operator accidentally typed “Haas Christine”
 - This happens less than 5% of the time
 - Index on LASTNAME and FIRSTNME
 - We need to find someone as cheaply as possible (reduced trips to database)

EMPLOYEE TABLE

EMPNO	FIRSTNME	LASTNAME	WORKDEPT	HIREDATE	JOB	BIRTHDATE	SALARY
10	CHRISTINE	HAAS	A00	1/1/1965	PRES	8/24/1933	52750
20	MICHAEL	THOMPSON	B01	10/10/1973	MANAGER	2/2/1948	41250
30	SALLY	KWAN	C01	4/5/1975	MANAGER	5/11/1941	38250
50	JOHN	GEYER	E01	8/17/1949	MANAGER	9/15/1925	40175
60	IRVING	STERN	D11	9/14/1973	MANAGER	7/7/1945	32250
70	EVA	PULASKI	D21	9/30/1980	MANAGER	5/26/1953	36170
90	EILEEN	HENDERSON	E11	8/15/1970	MANAGER	5/15/1941	29750

Search Performance with During Join Predicate

- A During Join Predicate is utilized
 - Perform the second join only when the first returns nothing
 - Only one trip to the database
 - Read index twice only when we have to
- This is useful when performance is critical!

```
SELECT COALESCE(A.EMPNO, B.EMPNO)
FROM   SYSIBM.SYSDUMMY1
LEFT OUTER JOIN
      (SELECT EMPNO
       FROM   EMPLOYEE
        WHERE LASTNAME = 'HAAS'
         AND   FIRSTNME = 'CHRISTINE') AS A
ON 1=1
LEFT OUTER JOIN
      (SELECT EMPNO
       FROM   EMPLOYEE
        WHERE LASTNAME = 'CHRISTINE'
         AND   FIRSTNME = 'HAAS') AS B
ON A.EMPNO IS NULL;
```

```
SELECT COALESCE(A.EMPNO, B.EMPNO)
FROM   SYSIBM.SYSDUMMY1
LEFT OUTER JOIN
      EMPLOYEE A
ON IBMREQD = 'Y'
LEFT OUTER JOIN
      (SELECT EMPNO
       FROM   EMPLOYEE
        WHERE LASTNAME = 'CHRISTINE'
         AND   FIRSTNME = 'HAAS') AS B
ON A.EMPNO IS NULL
WHERE (A.LASTNAME = 'HAAS'
      OR A.LASTNAME IS NULL)
AND   (A.FIRSTNME = 'CHRISTINE'
      OR A.FIRSTNME IS NULL);
```

```
SELECT COALESCE(A.EMPNO, B.EMPNO) FROM SYSIBM.SYSDUMMY1
LEFT OUTER JOIN EMP A ON IBMREQD = 'Y' AND (A.LASTNAME = 'HAAS' OR A.LASTNAME IS NULL)
AND (A.FIRSTNME = 'CHRISTINE' OR A.FIRSTNME IS NULL)
LEFT OUTER JOIN
      (SELECT EMPNO FROM EMP WHERE LASTNAME = 'CHRISTINE' AND FIRSTNME = 'HAAS') AS B
ON A.EMPNO IS NULL
```

Read all of the rules in the application programming guide about materialization!

Table Expression to Influence Table Access Sequence

- In Situations in Which the Table Access Sequence of a Join is Less Than Desired
 - Or There is No Good Supporting Index on the Inner Table
- A Table Expression Can Improve Performance

```
SELECT T1.C1, T2.C2
FROM T1 INNER JOIN T2
ON T1.C1 = T2.C1
AND T1.C3 = :ws-C3
AND T2.C4 = :ws-C4
```

Query Suffers from Death by Random I/O on table T2

DISTINCT forces a sort in joining column sequence

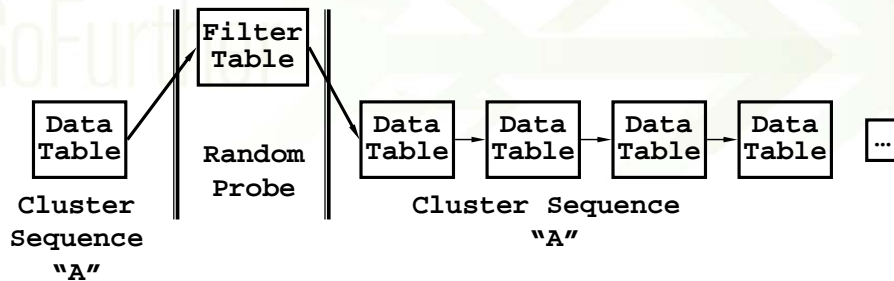
```
SELECT A.C1, B.C2
FROM T1 A INNER JOIN
(SELECT DISTINCT C1, C2
FROM T2
WHERE C4 = :ws-C4) AS B
ON A.C1 = B.C1
AND A.C3 = :ws-C3
```

Join with NOT IN list

Challenge: 7 tables all have the same primary key design.
 The obscure list is based upon a random probe into a
 30 Billion row table.

Scenario:

**Select data based upon NOT IN on an
 obscure list.**



70

As with the “IN” list example, there is also a problem with a “NOT IN” or “NOT EXISTS” probe into a secondary table. In this case the access to all tables in the join is always in sequential access, the problem is that the probes into the “exists” table is random.

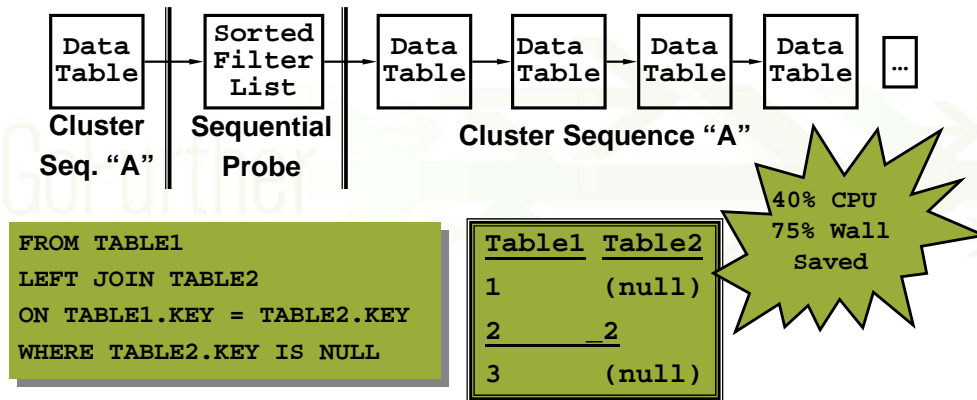
The random probe in my case is into a 1 billion row table (will be up to 37 billion when fully populated).

The solution will be to probe the data in a very effective manner, yet maintain my clustering sequence for further joined table access.

Join with NOT IN List

Solution:

- Use ANTI-Join to eliminate rows



71

IBM refers to this technique as an “ANTI-JOIN”. This process will effectively perform my “NOT EXISTS” criteria with an exceptional efficiency. However, to make the anti-join non-invasive, relative to random I/O, the data cluster coming back from the sequential probe will need to be in key sequence.

The filtering happens just after the join.;it screens the “NULL” data out.

NOT IN and NOT EXISTS are Stage 2 processes.

WHERE x IS NULL is stage 1.

Review the query on the next page.

ANTI - JOIN

```
SELECT result columns
FROM creator.table_1 T1
JOIN (SELECT KEY_COL
      FROM creator.table_2 AS KD
      LEFT JOIN exclude.tab ET
      ON KD.KEY = ET.KEY
      WHERE ET.COL IS NULL
      AND KD.local filter criteria
      GROUP BY KD.KEY_COL) AS SORTED_KEYS
ON T1.KEY_COL = SORTED_KEYS.KEY_COL
JOIN creator.table_2 T2 ON T1.KEY_COL = T2.KEY_COL
JOIN creator.table_3 T3 ON T1.KEY_COL = T3.KEY_COL
JOIN creator.table_4 T4 ON T1.KEY_COL = T4.KEY_COL
JOIN creator.table_5 T5 ON T1.KEY_COL = T5.KEY_COL
JOIN creator.table_6 T6 ON T1.KEY_COL = T6.KEY_COL
WHERE further filter criteria
```

The sorted area is done first...however, the entire ANTI-JOIN is done prior to the sort occurring. TABLE_2 and EXCLUDE.TABLE are joined together. The filtering is done up front, then the resultant keys are sorted back into clustering sequence. All subsequent access to the later tables is always in clustering sequence.

TA(B)LES from the Front

- SQL as a Program to Solve a Massive Billing Problem
- The Worlds Most Complex SQL Statement
- IBM Developers are Real People Too
- Of Coyotes, Alcohol, and Point of Sale Pricing
- The Complete Insanity of Sorting Data in a SQL Statement Without an ORDER BY!

Session F07 & F08

Utilizing DB2 V8
Table Expressions

Daniel L Luksetich

YL&A

Dan_Luksetich@ylassoc.com

GoFurther

