May 6–10, 2007

San Jose Convention Center

San Jose, California, USA

Session: F05

(Not So) Stupid SQL Tricks

Or
"My SPUFI writes JCL— Yours Can Too!"

IDUG® 2007 North America

Mark Doyle
Winterthur U. S. Holdings

May 8, 2007 09:20 a.m. – 10:20 a.m.

Platform: DB2 for z O/S & DB2 for LUW

GoFurther

Objectives and Agenda

- **Objective 1:**
  We will discuss the theory behind SMART SQL; and why it works the way it does.
- **Objective 2:**
  We will look at the various SPUFI, DB2I, ISPF defaults that make creating & using SMART SQL easy.
- **Objective 3:**
  We will look at the various SQL features, functions & constructs that make SMART SQL possible, including special registers, string functions (CHAR, DIGITS, RTRIM, SUBSTR & CONCAT), CASE, UNION ALL, "semi-Cartesian" joins, and "driver" tables.
- **Objective 4:**
  We will discuss the anatomy of a SMART SQL statement, including its four main parts. We will walk through the process of turning a template into SMART SQL.
- **Objective 5:**
  We will discuss several examples of SMART SQL statements, paying particular attention to the ways that the SQL can be changed to fit different environments, situations and requirements. Attendees are invited to bring their hardest database administration challenges to see if these techniques can assist them in overcoming those challenges

2

1. Introduction -- The need for flexible tools.
2. SMART SQL -- the big idea -- Any data that can be obtained from a table can appear in the output.
3. Advantages and Disadvantages of SMART SQL
4. Setting up for success - TSO, ISPF, DB2I & SPUFI defaults
5. Anatomy of a SMART SQL statement
6. Implementing SMART SQL -- turning a template into SMART SQL
7. The features, functions and facilities of SQL that make "smart" SQL possible
8. “Semi-Cartesian” and Cartesian Joins, Driver tables
9. The power of the CASE Statement
10. ISPF editing tricks
11. Troubleshooting SMART SQL
12. The examples:

Counting rows in the catalog; Mass GRANTs; Building GDGs;
“Show me table spaces that aren’t there”; Stacking datasets on a tape;
Building an “audit” tables;

Suggestions from the attendees (or more examples from my files)

Audit table triggers; Mass REBINDs; Mass DCLGENs;
INSERT INTO . . . SELECT * FROM . . .; Altering the buffer pool placement;

For More Information:

All the examples presented here (and many more) can be found at:

- IDUG Code Place
- CBT Tape

3

Administrative notes:

1) I tend to be a fairly informal speaker, and this presentation is intended to be a participatory experience, so please ask questions as they occur. Each slide and example builds on previous slides and examples, so it's important that everyone understands the material as it is presented.

2) I have made every effort to ensure accuracy in the examples and code snippets, but there may be mistakes or omissions therein. It is your responsibility to ensure that the examples are applicable to your environment and situation, and work as you intend them to.

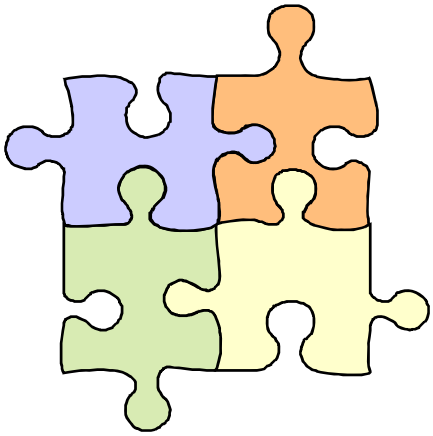
3) I have way more material than can even be printed in the foils and notes, so I have uploaded the SQL examples to these places above.

4) This presentation is heavily skewed toward DB2 on the z/OS, not because the techniques presented are any less useful or applicable to other platforms, but instead for these three reasons:
    1) Mainframe users have more environmental constraints than do users on other platforms (e.g. 80-character-wide datasets, SPUFI, etc.)
    2) Most of the examples presented will work as-is on the other platforms, once the z/OS elements are removed.
    3) I'm a dinosaur!

5) In the interest of saving space on the foils and notes, I have omitted features that I would consider best-practices. The web resources on the slide do not omit those features, and should be considered definitive.

Introduction – The Need for Flexible Tools

- Our jobs are getting more complex:
  - More “things” to manage
  - More availability
  - Less down time
  - More functionality
  - Need it faster, cheaper, better
- Information Technology in general and Database Administrators specifically face those same challenges
  - Third-party DB2 Tools have helped
  - We need a really flexible tool.



4

DBAs now are expected to manage more information, in more environments, with less down time and fewer resources.

Here is a partial list of requirements I have been asked to support in the past year:

- Grant security on the tables of a database, based on the presence of a particular character string in the table comments.
- Create audit tables for all base tables in a database, with before & after values of the data that changed, and with insert, delete & update triggers to populate the audit tables.
- “Give me a way to tell the data value differences between all tables in two databases, given that there is the same logical structure for all logical objects in the two databases.”
- Given a database with both application control data and application transaction data (these two types of tables distinguished by the presence or absence of a particular character string in the table comments), produce a process to translate the current primary key values of just the transaction tables (and the foreign keys that relate to those primary keys) from their current “random” generation method value to use a sequence object generation method. Use this process to change the current primary & foreign keys to their new values.

Third-party tools have helped, but they can't keep pace with the ever-growing list of new business requirements.

This presentation will discuss a set of techniques, constructs and "tricks" -- a.k.a. "Smart SQL" -- that will allow DBAs to build their own tools, and build tools that build tools, based on their own environment and the constraints they face. Smart SQL can make DBAs (and others) more productive and better able to handle the challenges put before them.

Although Smart SQL is presented in the context of DB2 on a z/OS mainframe computer with emphasis on DBA tasks, the techniques presented here are usable in ANY relational database management system, on ANY hardware platform, and for ANY set of tasks.

Attendees are invited to bring their hardest database administration challenges, to see if these techniques can assist them in overcoming those challenges.

SMART SQL – The Big Idea

Any data that can be obtained from a table can appear in the output.

- Any data:
  - Column values (obviously)
  - Aggregate Functions (MAX, MIN, AVG, COUNT, etc.)
  - Scalar Functions (CHAR, CONCAT, RTRIM, etc.)
  - User Defined Functions (UDFs)
  - Literals / Constants
  - Special Registers (CURRENT SQLID, CURRENT SERVER, etc.)
  - Combinations of the above

5

Any data includes a wide variety of constructs:

Column values (obviously)

Aggregate Functions (MAX, MIN, AVG, COUNT, etc.)

There are 9 aggregate functions supplied by IBM

Scalar Functions (CHAR, CONCAT, RTRIM, etc.)

There are 128 scalar functions supplied by IBM

User Defined Functions (UDFs)

Because they are user defined, these are hard to discuss.

Literals / Constants

Special Registers (CURRENT SQLID, CURRENT SERVER, etc.)

There are 24 special registers in DB2 for z/OS V8.

Combinations of the above

SMART SQL – The Big Idea

Any data that can be obtained from a table can appear in the output.

- From a table:
  - A ‘real’ table
  - A view
  - A Nested Table Expression (NTE)
  - Any full select
  - Combinations of the above

6

The last few releases of DB2 have really added to the power of SQL, by changing what is allowed in the “FROM” clause. Not all that long ago, a “FROM” clause could only contain:

- Table, view, alias or synonym names,
- Their correlation names, and
- Commas.

Now, with only a few exceptions*, anything we can imagine as group of rows and columns can be the object of a “FROM” clause.

* The most notable exception is that a single SQL statement cannot reference objects in more than one subsystem / instance (without additional software, special setup, etc).

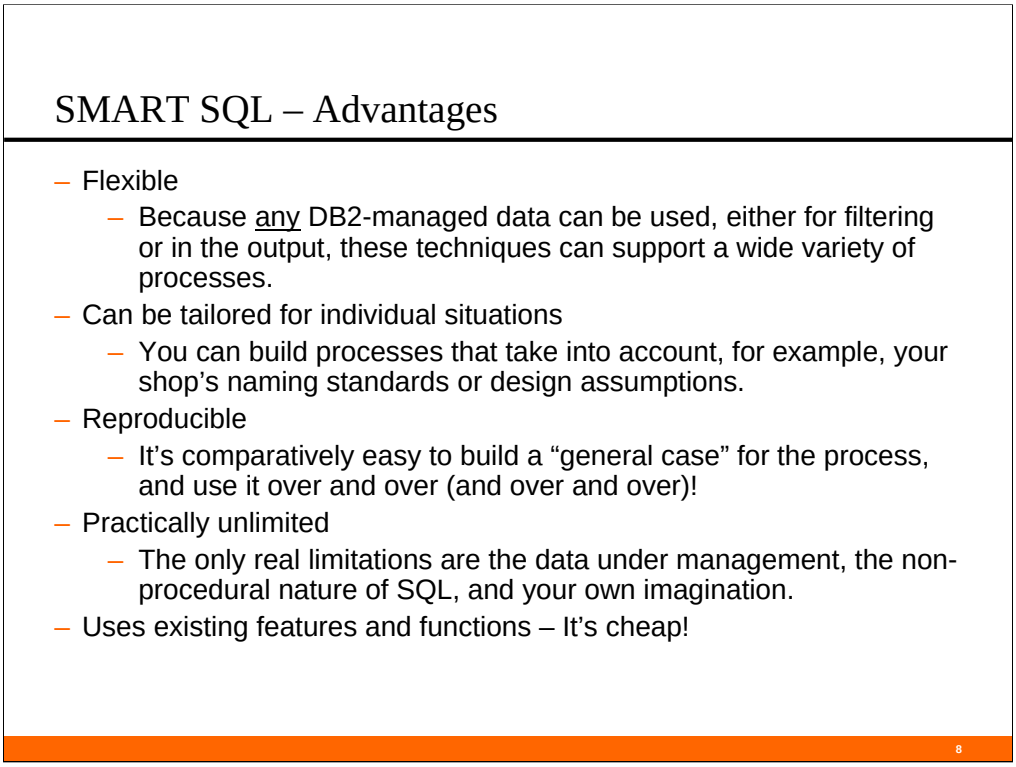
SMART SQL – The Big Idea

Any data that can be obtained from a table can appear in the output.

- The Output:
  - A one-column, eighty-character-wide output table equals
    - JCL
  - A one-column, seventy-two-character-wide output table equals
    - SPUFI Input

7

Here is one place where DB2 on different platforms behave much differently. On the LUW platforms, these constraints don't normally apply.



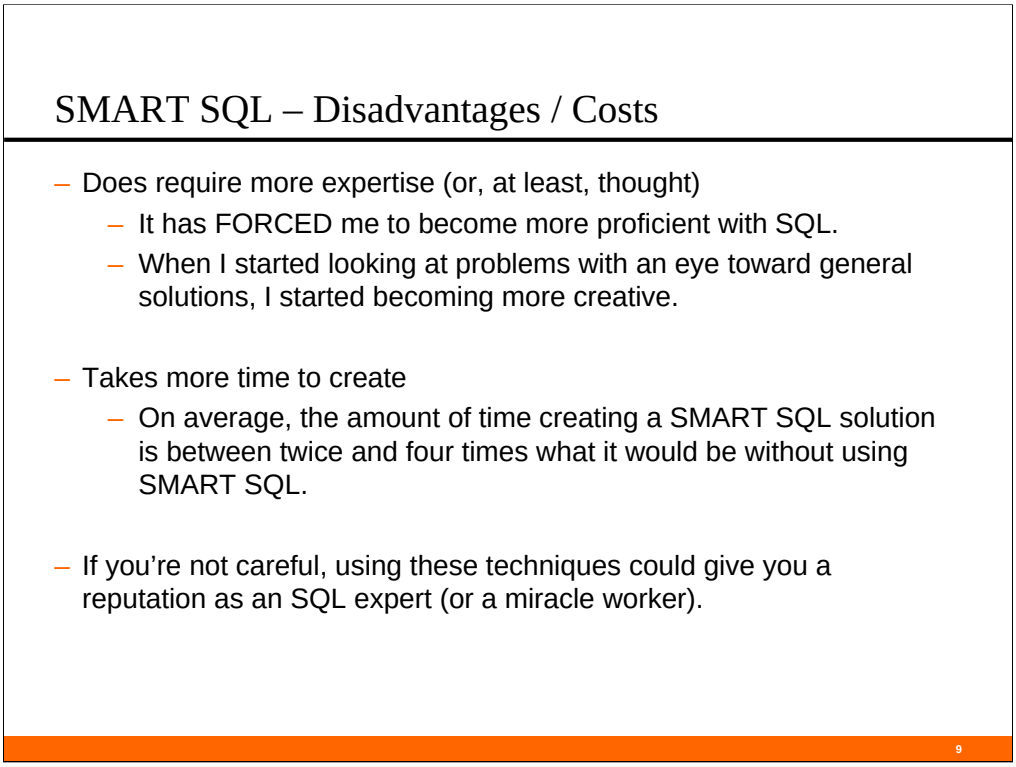
These techniques came from my years as a consultant on short and medium term contracts. Because I could never be sure which third-party software products any particular client might have (if any), I made it a point to be able to be as productive with the functionality of the base DB2 product as others were with a third-party software tool.

Those of us who manage data assets using DB2 work very hard to ensure that DB2's functionality is available to the developers and users that we support. But we seldom avail ourselves of that same functionality to make our own work easier or give ourselves a productivity boost. It is the classic case of the shoemaker's children going barefoot.

Over the years, third-party software products have become continually better, with increased features and more functionality. They usually perform only the functions they are designed for, but they typically do those functions very well. But they can never keep up with changing requirements and the increasing state of the art.

By making our tools simpler, we can make them more flexible and more able to keep up with new DB2 functionality.

I recognized that I didn't have to solve a business problem (or a database problem) all at once. I realized that I really could "build a tool to build a tool". With this realization, SMART SLQ came to life. This meta-tool approach turned insolvable problems into solvable ones, and hard problems into easy ones. (continued on next note page)



(continued from previous note page)

For over fifteen years, I have investigated how to use basic SQL to make me more productive, and my job easier. This presentation is the culmination of that investigation.

As I developed templates for different database activities, I saw that my turn-around time for those tasks went down considerably. There were fewer times I had to say: "Sure, I can do that; give me a week to finish it" and more times I could say: "Sure, I can do that; give me half an hour to finish it."

Another implication of using SMART SQL is that the time I spend accomplishing a task is no longer dependent on the number of objects I am dealing with. Because I have a SMART SQL template, it takes me about as long to, say, set up GDG image-copy datasets for an entire DB2 subsystem as it does for one database.

It's true, using these techniques does require more time initially to set up. And it has forced me to become more proficient with SQL (like that's a bad thing!). But as I look back on it, I have usually recouped all my initial development time by the second, third or fourth time I have re-done that same process.

All-in-all, I'd say the results are worth the investment.

SPUFI, DB2I, ISPF and TSO Defaults

```
DSNESP02                  CURRENT SPUFI DEFAULTS            SSID: DB2T
===>

Enter the following to control your SPUFI session:
 1  SQL TERMINATOR .. ===> ;          (SQL Statement Terminator)
 2  ISOLATION LEVEL   ===> CS         (CS=Cursor Stability)
 3  MAX SELECT LINES  ===> 0          (Maximum number of lines to be
                                         returned from a SELECT)
 4  ALLOW SQL WARNINGS===> NO         (Continue fetching after sqlwarning)
 5  CHANGE PLAN NAMES ===> NO         (Change the plan names used by SPUFI)

Output data set characteristics:
 6  RECORD LENGTH ... ===> 80         (LRECL=Logical record length)
 7  BLOCK SIZE ...... ===> 24000      (Size of one block)
 8  RECORD FORMAT ... ===> FB         (RECFM=F, FB, FBA, V, VB, or VBA)
 9  DEVICE TYPE ..... ===> SYSDA      (Must be DASD unit name)

Output format characteristics:
10  MAX NUMERIC FIELD ===> 80         (Maximum width for numeric fields)
11  MAX CHAR FIELD .. ===> 99999      (Maximum width for character fields)
12  COLUMN HEADING .. ===> NAMES      (NAMES, LABELS, ANY or BOTH)

PRESS:  ENTER to process    END to exit             HELP for more information
```

10

Before we get into the implementation details of SMART SQL, we need to take a short side trip into some environmental parameters that will make using these techniques easier.

The first is our SPUFI defaults.

To access these, make sure line 5 of your SPUFI panel is set to YES:

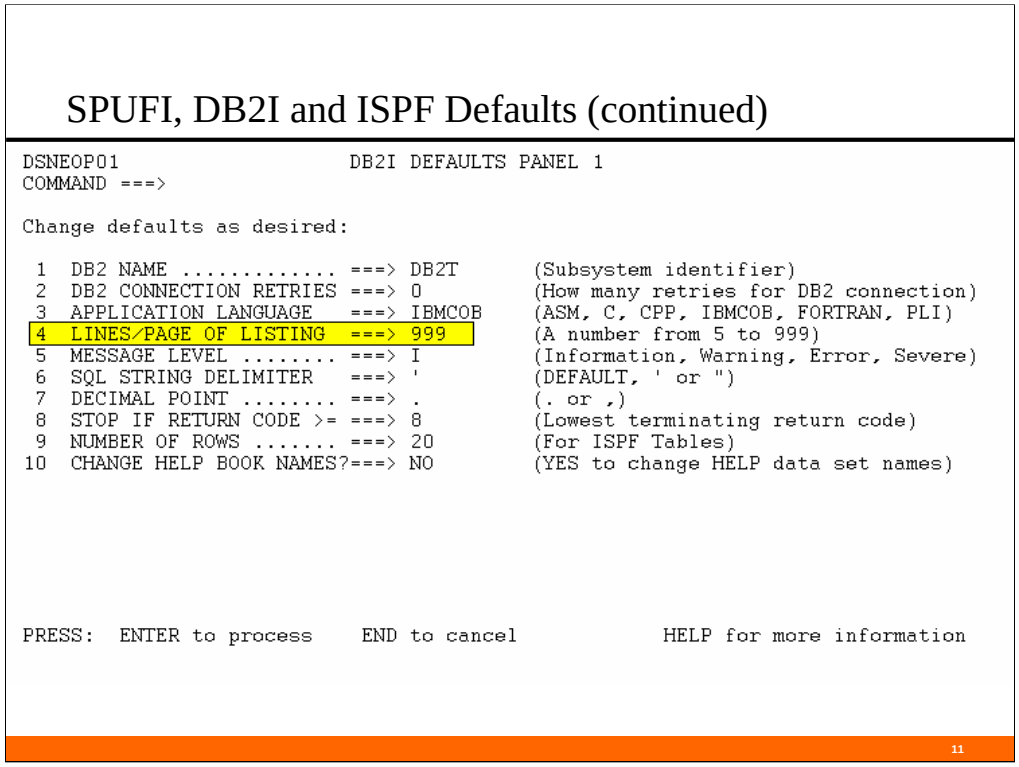
Specify processing options:

5 CHANGE DEFAULTS ===> YES (Y/N - Display SPUFI defaults panel?)

The values above will give us an 80-character output.

24,000 is a useful block size because it is divisible so many ways (e.g. 80, 100, 120, 150, 200, 240, 250, 300, 400, 500, 600, 800, etc.)

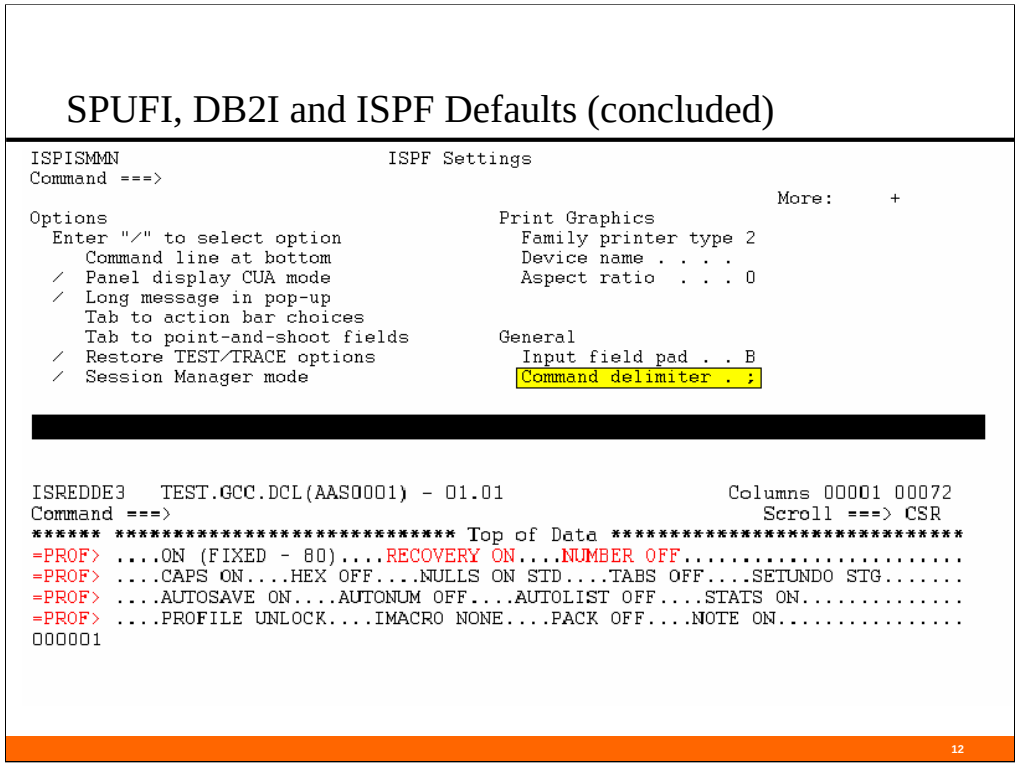
SPUFI, DB2I and ISPF Defaults (continued)



The next part of our side trip is to the DB2I defaults (option D from the DB2I Main panel).

Here, we are changing the number of lines that are written between “page headers”.

SPUFI, DB2I and ISPF Defaults (concluded)



Occasionally, it is useful to treat the semi-colon as just another character, not as the ISPF command delimiter. In these cases, we can change the command delimiter on the ISPF defaults panel -- option 0 (that's zero) from the main ISPF panel.

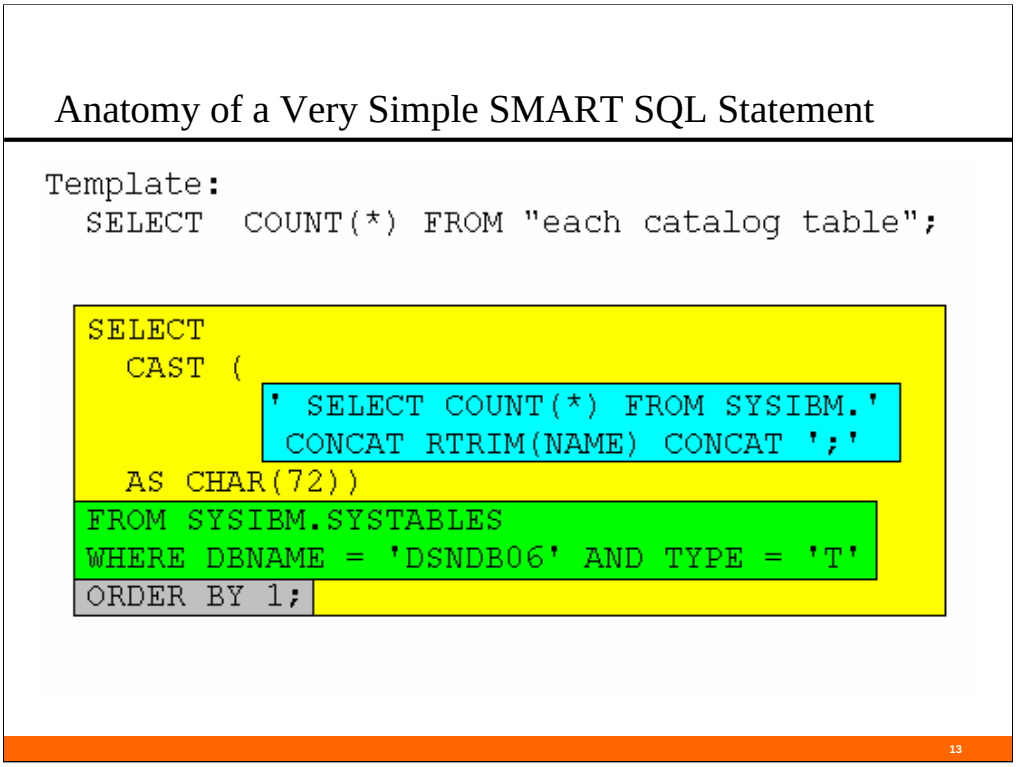
Finally, there are some editing profile defaults that are handy to use:

•You can type "PROFILE" on the command line to get the display above.

•Turning off the numbering option (type "UNNUM" on the command line) will leave card columns 73-80 blank, and also have the line numbers on the left side of the screen ascend by 1 (not 10).

•Turning on the recovery option in the dataset you are editing will give you access to the "UNDO" editing command. This can be useful for those "Oops!" moments. Whether or not you can actually use this function is dependant on the amount of storage assigned to each TSO session, and may not work fully in all environments.

•The retrieve key. You can set a PF key to the retrieve command, which will keep a history of the TSO/ISPF command line commands you issued. To create a retrieve key, type "KEYS" on the command line. Choose a PF key and overtype the command with "RETREIVE" (or "CRETRIEV" in some installations). This is useful in conjunction with the "UNDO" command.



This SQL (which is somewhat contrived) will answer the question "How many rows are there in each catalog table?"

It shows portions of the four main parts of a SMART SQL statement:

1. Required elements of ANY SQL statement. (in yellow) The CAST function is not a required, but I find it convenient to include it as a required element. Later, we will discuss the reasons why I usually have a cast function in my SQL.
2. What do we want to see in the output (in blue). This example has three text fields, concatenated together as one output "column". They are a literal character string, the table name (with trailing blanks removed) and another character string.
3. Where do we get the information (in green). In this example, the table is SYSIBM.SYSTABLES, furthermore, we only want tables in the DSNDB06 database that are of the "table" type.
4. How do we want the output arranged (in grey). In this example, the ordering is not particularly important, but it will be later, and so it is included here.

A Very Simple SMART SQL Statement (continued)

```
ISREDDE3   DDBA02.SPUFI.OUT                                  Columns 00001 00072
Command ===>                                                    Scroll ===> CSR
****** ***************************** Top of Data ******************************
000001 ---------+---------+---------+---------+---------+---------+---------+--
000002   SELECT
000003     CAST (
000004        ' SELECT COUNT(*) FROM SYSIBM.'
000005       CONCAT RTRIM(NAME)
000006       CONCAT ';'
000007     AS CHAR(72))
000008   FROM
000009   SYSIBM.SYSTABLES
000010   WHERE DBNAME = 'DSNDB06'
000011     AND TYPE = 'T'
000012   ORDER BY 1;
000013 ---------+---------+---------+---------+---------+---------+---------+--
000014
000015 ---------+---------+---------+---------+---------+---------+---------+--
000016  SELECT COUNT(*) FROM SYSIBM.IPLIST;
000017  SELECT COUNT(*) FROM SYSIBM.IPNAMES;
000018  SELECT COUNT(*) FROM SYSIBM.LOCATIONS;
- - -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  - 79 Line(s) not Displayed
000098  SELECT COUNT(*) FROM SYSIBM.USERNAMES;
000099 DSNE610I NUMBER OF ROWS DISPLAYED IS 83
000100 DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 100
000101 ---------+---------+---------+---------+---------+---------+---------+--
```


If we ran the SQL on the last slide in SPUFI, this is the result set we would get.

Notice that lines 1 – 13 are an echo of the input statement.

Lines 14-15 are the result set header lines.

Line 14 would contain column names or labels (depending on our SPUFI default column heading – line 12 of the SPUFI defaults panel), if the output were unformatted columns or if we gave them an explicit name (which we didn't) .

Because we changed our lines-per-page value to be 999, instead of the default 50, lines 14 and 15 were not repeated as lines 51 and 52 (and all subsequent output lines shifted down 2).

Note: in order to show the significant parts of the output, 79 lines of the output (line 19 through line 97) were excluded.

A Very Simple SMART SQL Statement (continued)

- Once we remove the 'echo' lines, the page header lines and the 'footer' lines, the results can be used as SPUFI input.

- Note: We can tell that there are no other extraneous lines in the output because the number of rows displayed (in line 84) is equal to the line number of the line above.

```
ISREDDE3   DDBA02.SPUFI.OUT                              Columns 00001 00072
Command ===>                                               Scroll ===> CSR
****** ***************************** Top of Data ******************************
000001  SELECT COUNT(*) FROM SYSIBM.IPLIST;
000002  SELECT COUNT(*) FROM SYSIBM.IPNAMES;
000003  SELECT COUNT(*) FROM SYSIBM.LOCATIONS;
- - -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  - 79 Line(s) not Displayed
000083  SELECT COUNT(*) FROM SYSIBM.USERNAMES;
000084 DSNE610I NUMBER OF ROWS DISPLAYED IS 83
```

15

We can remove lines 1-15, and lines 84 to the end, and this output is now ready to be used as SPUFI input

Note: in order to show the significant parts of the output, 79 lines of the output (line 19 through line 97) were excluded.

A Very Simple SMART SQL Statement (concluded)

– This is the result if the output of the first SQL is used as SPUFI Input.

```
ISREDDE2   DDBA02.SPUFI.OUT                                Columns 00001 00072
Command ===>                                                 Scroll ===> CSR
****** ***************************** Top of Data ******************************
000001 ---------+---------+---------+---------+---------+---------+---------+--
000002  SELECT COUNT(*) FROM SYSIBM.IPLIST;
000003 ---------+---------+---------+---------+---------+---------+---------+--
000004
000005 ---------+---------+---------+---------+---------+---------+---------+--
000006           0
000007 DSNE610I NUMBER OF ROWS DISPLAYED IS 1
000008 DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 100
- - -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -   648 Line(s) not Displayed
000657 ---------+---------+---------+---------+---------+---------+---------+--
000658  SELECT COUNT(*) FROM SYSIBM.USERNAMES;
000659 ---------+---------+---------+---------+---------+---------+---------+--
000660
000661 ---------+---------+---------+---------+---------+---------+---------+--
000662           1
000663 DSNE610I NUMBER OF ROWS DISPLAYED IS 1
000664 DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 100
000665 ---------+---------+---------+---------+---------+---------+---------+--
000666 ---------+---------+---------+---------+---------+---------+---------+--
000667 DSNE617I COMMIT PERFORMED, SQLCODE IS 0
000668 DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
000669 ---------+---------+---------+---------+---------+---------+---------+--
000670 DSNE601I SQL STATEMENTS ASSUMED TO BE BETWEEN COLUMNS 1 AND 72
000671 DSNE620I NUMBER OF SQL STATEMENTS PROCESSED IS 83
000672 DSNE621I NUMBER OF INPUT RECORDS READ IS 83
000673 DSNE622I NUMBER OF OUTPUT RECORDS WRITTEN IS 673
```

16

When we run the output back through SPUFI, here is the result.

Hmmm. . . The output is not as useful as I would have thought. It takes almost 700 lines to answer the original question. It would be better as an 83-line output, with each line containing the catalog table name and the row count for that table. Better still, since this (hypothetical) output is to be reported to upper management, and they really only care about the bottom-line number, I would prefer not to make them add up the 83 lines of numbers themselves (not that I don't think they could do it).

There ought to be a way to get exactly what I want.

There is . . . But we haven't yet gotten all the pieces to make it happen.

We'll look at another example or two before revisiting this question.

Note: in order to show the significant parts of the output, 648 lines of the output (line 9 through line 656) were excluded.

A Slightly More Complex Example

```
Template:
  GRANT SELECT, INSERT, UPDATE, DELETE
  ON Creator.Name TO PUBLIC;
```

```
SELECT
    CAST (
          '  GRANT SELECT, INSERT, UPDATE, DELETE '
     AS CHAR(72) )
, NAME, 1
FROM
SYSIBM.SYSTABLES
WHERE DBNAME = 'E%'
UNION ALL
SELECT
    CAST (
          '  ON RTRIM(CREATOR) CONCAT '.'
          CONCAT RTRIM(NAME) CONCAT ' TO PUBLIC;'
    AS CHAR(72) )
, NAME, 2
FROM
SYSIBM.SYSTABLES
WHERE DBNAME = 'E%'
ORDER BY 2,3;
```

17

Say we wanted to do select, insert, update & delete grants to public for each table in every database that begins with the letter “E”. We know that the SQL command will NOT fit on one line, so we need to split the command on two lines. How might we do that?

This command will give us that result set.

The parts in blue are our output text. We need one UNION ALL for each line of output that we want.

The parts in green tell us the source of the data.

The parts in grey order our output to give us well-formed SQL.

A Slightly More Complex Example

– Here is the output, ready to be run as input in SPUFI

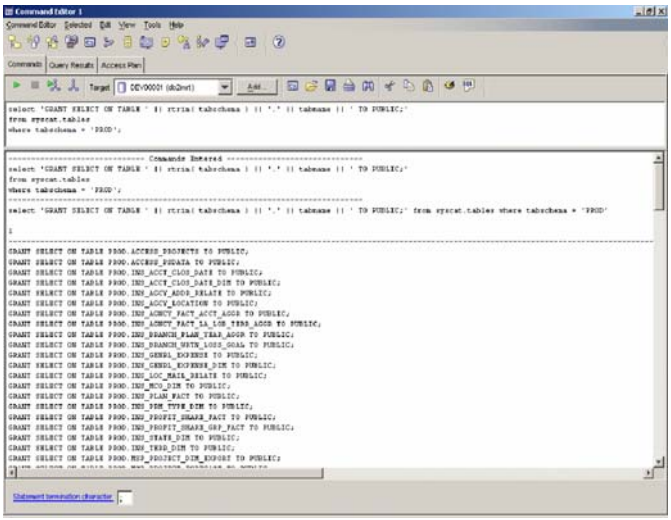
```
ISREDDE2   DDBA02.SPUFI.OUT                                  Columns 00001 00072
Command ===>                                                   Scroll ===> CSR
****** ***************************** Top of Data ******************************
000001   GRANT SELECT, INSERT, UPDATE, DELETE
000002     ON DVL3.PLQ_ADDL_INT TO PUBLIC;
000003   GRANT SELECT, INSERT, UPDATE, DELETE
000004     ON DVL3.PLQ_ADDL_INT_TYP TO PUBLIC;
- - -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -   314 Line(s) not Displayed
000319   GRANT SELECT, INSERT, UPDATE, DELETE
000320     ON DVL3.PLQ_YRS_CURR_RES TO PUBLIC;
000321   GRANT SELECT, INSERT, UPDATE, DELETE
000322     ON DVL3.PMS_UPDATE_RQST TO PUBLIC;
000323 DSNE610I NUMBER OF ROWS DISPLAYED IS 322
****** **************************** Bottom of Data ****************************
```

18

This is what we get when we run the SQL on the previous slide, and remove the header lines. I know that the result set is accurate because there are 161 tables that meet the criterion (times two lines per grant = 322 lines of grant statements).

Note: in order to show the significant parts of the output, 314 lines of the output (line 5 through line 318) were excluded.

A Slightly More Complex Example (AIX)



19

This is the obligatory LUW example. The screen shot above shows that it works in the DB2 Command Center for an AIX database. The DB2 line command below gives us essentially the same result (output not shown):

db2 "export to temp.out of del modified by nochardel select 'GRANT SELECT ON TABLE ' || table_name || ' TO PUBLIC;' from sysibm.tables"

Turning a Template into SMART SQL

```
//JOBNAME JOB ACCT#,'CRE8GDG',CLASS=C,MSGCLASS=X,NOTIFY=&SYSUID

//STEP010  EXEC PGM=IDCAMS
//SYSPRINT DD  SYSOUT=*
//SYSIN    DD  *
DEFINE GDG (NAME(HLQ.FCOPY.DPLQ0001.DPLQ001.DATA) LIMIT(15) SCRATCH)
   (repeat line for each tablespace)
/*
//
```

(Assume we need a GDG for each tablespace in every database beginning with the letter "E", to be used for image copies)

– Get an example of the output (a Template)

20

We've done two specific SQL examples of turning a business question or problem into SQL to answer that question or resolve that problem. Let's get a general solution to that process, and this time, get our output to be JCL.

Note that this example is only vaguely DB2-related, and only because the datasets will be used to hold image copies.

We will start with a template of our desired output.

Turning a Template into SMART SQL

```
//JOBNAME JOB ACCT#,'CRE8GDG',CLASS=C,MSGCLASS=X,NOTIFY=&SYSUID

//STEP010  EXEC PGM=IDCAMS
//SYSPRINT DD  SYSOUT=*
//SYSIN    DD  *
DEFINE GDG (NAME(HLQ.FCOPY. DBNAME . TSNAME.DATA) LIMIT(15) SCRATCH)
   (repeat line for each tablespace)
/*
//
```

- Identify each variable

- Determine source of each variable
  (SYSDUMMY1, Catalog table, Other tables)

21

Next, we identify literals and variables, and determine what is the source for each variable.

•In this example, the variables are in red, and the literal values are in black.

•(Notice that the period between the database name variable and the tablespace name variable is black).

•We can get the data for each variable from SYSIBM.SYSTABLESPACE.

•Because in every line except the one that starts “CREATE GDG. . .”, the literals happen only once in the output. We can use SYSIBM.SYSDUMMY1 to output the literals on those lines.

Turning a Template into SMART SQL

```
‘//JOBNAME JOB ACCT#,‘‘CRE8GDG‘‘,CLASS=C‘ CONCAT
   ‘,MSGCLASS=X,NOTIFY=&SYSUID‘
‘//STEP010  EXEC PGM=IDCAMS‘
‘//SYSPRINT DD  SYSOUT=*‘
‘//SYSIN    DD  *‘
‘DEFINE GDG (NAME(HLQ.FCOPY.‘ CONCAT RTRIM(DBNAME) CONCAT
   ‘.‘ CONCAT RTRIM(NAME)CONCAT ‘.DATA) LIMIT(15) SCRATCH)‘
‘/*‘
‘//‘
```

– Double each existing tick in the input

– Delimit each literal with ticks

22

1. Double each existing string delimiter* (for simplicity, I will use the word “tick” instead of the phrase “string delimiter” from now on) in the input
    - There is no real significance to the order in which we do this editing, except for this item. It’s much easier to identify existing ticks in the template before we add any new ticks for literal values.
    - A note about mainframe (DB2 and JCL) literals: as DB2 or JES parses a statement, it starts with the “Literal-string” switch OFF. As it encounters the first tick, it turns the “Literal-string” switch ON.
        - While the “Literal-string” switch is ON, any non-tick character is passed as-is to the literal string,
        - While the “Literal-string” switch is ON, it takes two characters to completely parse any tick the parser encounters.
            - If the string is “tick, any non-tick character”, the “Literal-string” switch turns OFF, and the second character is parsed as though it were not part of the literal string.
            - If the string is “tick, tick”, the string is parsed as a single tick in the literal, and the “Literal-string” switch remains ON.
    - Using this conceptual framework, and assuming we are not in the middle of a literal, it is easy to interpret the string “blank, 6 ticks, blank” as a two-character literal containing 2 ticks or to interpret the string “blank, 10 ticks, blank” as a four-character literal with 4 ticks, or to interpret the string “blank, 3 ticks, blank, 3 ticks, comma” as a three-character literal “tick, blank, tick” with a comma after it.

(continued on next note page)

* String delimiter: DB2 has several ZPARMs that affect the default string delimiter in various contexts (see install panel DSNTIPF). While an apostrophe is the supplied SQL string delimiter default, it may be different in your installation. This MAY also be the case for JCL (although I have not personally seen this).

Turning a Template into SMART SQL

```
‘//JOBNAME JOB ACCT#,‘‘CRE8GDG‘‘,CLASS=C‘ CONCAT
   ‘,MSGCLASS=X,NOTIFY=&SYSUID‘
‘//STEP010  EXEC PGM=IDCAMS‘
‘//SYSPRINT DD  SYSOUT=*‘
‘//SYSIN    DD  *‘
‘DEFINE GDG (NAME(HLQ.FCOPY.‘ CONCAT RTRIM(DBNAME) CONCAT
   ‘.‘ CONCAT RTRIM(NAME)CONCAT ‘.DATA) LIMIT(15) SCRATCH)‘
‘/*‘
‘//‘
```

– Format variables as needed (e.g. to make them character type, to remove embedded blanks, etc.)

– Put CONCAT between variables / literals

23

----------(continued from last note page)

2. Delimit the start and end of each literal with a tick

It is valid to split a literal by adding the following string between any two non-tick characters in the literal:

“tick (blank) CONCAT (blank) tick”

As long as we are not inside a literal, any number of blanks can be added after a blank character in the SQL statement without changing the parsing.

You can split a single line of SQL into two lines of SQL before or after any non-literal blank.

These three ‘rules’ allow us to change the first line of the template to it’s present form (see previous slide).

----------(notes for this slide)

3. Format variables as needed. In this example, the variables are already character-type, so we don’t need to do any data type conversion. But the variables are also part of a TSO dataset name; embedded blanks in the name would cause a syntax error, so we need to use the RTRIM function (or an equivalent function) to remove the blanks.

4. Put CONCAT between each pair of variables, each pair of literals, each literal / variable pair and each variable / literal pair

Turning a Template into SMART SQL

```
SELECT
  '//JOBNAME JOB ACCT#,''CRE8GDG'',CLASS=C' CONCAT
  ',MSGCLASS=X,NOTIFY=&SYSUID'
UNION ALL SELECT
  '//STEP010  EXEC PGM=IDCAMS'
UNION ALL SELECT
  '//SYSPRINT DD  SYSOUT=*'
UNION ALL SELECT
  '//SYSIN    DD  *'
UNION ALL SELECT
   'DEFINE GDG (NAME(HLQ.FCOPY.' CONCAT RTRIM(DBNAME) CONCAT
   '.' CONCAT RTRIM(NAME)CONCAT '.DATA) LIMIT(15) SCRATCH)'
UNION ALL SELECT '/*'
UNION ALL SELECT  '//'
```

– Add “SELECT” and “UNION (ALL)” clauses

24

Here is the well-formatted SQL:

```
SELECT
  '//JOBNAME JOB ACCT#,''CRE8GDG'',CLASS=C' CONCAT
   ',MSGCLASS=X,NOTIFY=&SYSUID'
UNION ALL
SELECT
  '//STEP010  EXEC PGM=IDCAMS'
UNION ALL
SELECT
  '//SYSPRINT DD  SYSOUT=*'
UNION ALL
SELECT
  '//SYSIN    DD  *'
UNION ALL
SELECT
   'DEFINE GDG (NAME(HLQ.FCOPY.' CONCAT RTRIM(DBNAME) CONCAT
   '.' CONCAT RTRIM(NAME)CONCAT '.DATA) LIMIT(15) SCRATCH)'
UNION ALL
SELECT  '/*'
UNION ALL
SELECT  '//'
```

Turning a Template into SMART SQL

```
SELECT
  '//JOBNAME JOB ACCT#,''CRE8GDG'',CLASS=C' CONCAT
  ',MSGCLASS=X,NOTIFY=&SYSUID'
FROM SYSIBM.SYSDUMMY1
UNION ALL SELECT  '//STEP010  EXEC PGM=IDCAMS'
FROM SYSIBM.SYSDUMMY1
UNION ALL SELECT   '//SYSPRINT DD  SYSOUT=*'
FROM SYSIBM.SYSDUMMY1
UNION ALL SELECT   '//SYSIN    DD  *'
FROM SYSIBM.SYSDUMMY1
UNION ALL SELECT
   'DEFINE GDG (NAME(HLQ.FCOPY.' CONCAT RTRIM(DBNAME) CONCAT
   '.' CONCAT RTRIM(NAME)CONCAT '.DATA) LIMIT(15) SCRATCH)'
FROM SYSIBM.SYSTABLESPACE WHERE DBNAME LIKE 'E%'
UNION ALL SELECT '/*'
FROM SYSIBM.SYSDUMMY1
UNION ALL SELECT  '//'
FROM SYSIBM.SYSDUMMY1
```

– Add "FROM" and "WHERE" clauses

25

Here is the well-formatted SQL:

```
 SELECT
    '//JOBNAME JOB ACCT#,''CRE8GDG'',CLASS=C' CONCAT ',MSGCLASS=X,NOTIFY=&SYSUID'
  FROM SYSIBM.SYSDUMMY1
UNION ALL
  SELECT
    '//STEP010  EXEC PGM=IDCAMS'
  FROM SYSIBM.SYSDUMMY1
UNION ALL
  SELECT
    '//SYSPRINT DD  SYSOUT=*'
  FROM SYSIBM.SYSDUMMY1
UNION ALL
  SELECT
    '//SYSIN    DD  *'
  FROM SYSIBM.SYSDUMMY1
UNION ALL
  SELECT
     'DEFINE GDG (NAME(HLQ.FCOPY.' CONCAT RTRIM(DBNAME) CONCAT
     '.' CONCAT RTRIM(NAME)CONCAT '.DATA) LIMIT(15) SCRATCH)'
  FROM SYSIBM.SYSTABLESPACE
  WHERE DBNAME LIKE '
UNION ALL
  SELECT       '/*'
  FROM SYSIBM.SYSDUMMY1
UNION ALL
  SELECT       '//'
  FROM SYSIBM.SYSDUMMY1
```

Turning a Template into SMART SQL

```
SELECT
  '//JOBNAME JOB ACCT#,''CRE8GDG'',CLASS=C' CONCAT
  ',MSGCLASS=X,NOTIFY=&SYSUID'
,1 , ' ', ' ' FROM SYSIBM.SYSDUMMY1
UNION ALL SELECT  '//STEP010  EXEC PGM=IDCAMS'
,2 , ' ', ' ' FROM SYSIBM.SYSDUMMY1
UNION ALL SELECT   '//SYSPRINT DD  SYSOUT=*'
,3 , ' ', ' ' FROM SYSIBM.SYSDUMMY1
UNION ALL SELECT   '//SYSIN    DD  *'
,4 , ' ', ' ' FROM SYSIBM.SYSDUMMY1
UNION ALL SELECT
   'DEFINE GDG (NAME(HLQ.FCOPY.' CONCAT RTRIM(DBNAME) CONCAT
   '.' CONCAT RTRIM(NAME)CONCAT '.DATA) LIMIT(15) SCRATCH)'
,5 , DBNAME, TSNAME FROM SYSIBM.SYSTABLESPACE
WHERE DBNAME LIKE 'E%'
UNION ALL SELECT '/*'
,6 , ' ', ' ' FROM SYSIBM.SYSDUMMY1
UNION ALL SELECT  '//'
,7 , ' ' FROM SYSIBM.SYSDUMMY1
ORDER BY 2,3,4;
```

– Add ordering columns and "ORDER BY" clause

26

Here is the well-formatted SQL:

```
  SELECT '//JOBNAME JOB ACCT#,''CRE8GDG'',CLASS=C' CONCAT
    ',MSGCLASS=X,NOTIFY=&SYSUID'    ,1 , ' ', ' '
  FROM SYSIBM.SYSDUMMY1
UNION ALL
  SELECT  '//STEP010  EXEC PGM=IDCAMS'   ,2 , ' ', ' '
  FROM SYSIBM.SYSDUMMY1
UNION ALL
  SELECT   '//SYSPRINT DD  SYSOUT=*'    ,3 , ' ', ' '
  FROM SYSIBM.SYSDUMMY1
UNION ALL
  SELECT   '//SYSIN    DD  *'     ,4 , ' ', ' '
  FROM SYSIBM.SYSDUMMY1
UNION ALL
  SELECT 'DEFINE GDG (NAME(HLQ.FCOPY.' CONCAT RTRIM(DBNAME) CONCAT
     '.' CONCAT RTRIM(NAME)CONCAT '.DATA) LIMIT(15) SCRATCH)'
  ,5 , DBNAME, TSNAME
  FROM SYSIBM.SYSTABLESPACE
  WHERE DBNAME LIKE 'E%'
UNION ALL
  SELECT '/*'    ,6 , ' ', ' '
  FROM SYSIBM.SYSDUMMY1
UNION ALL
  SELECT  '//'    ,7 , ' '
  FROM SYSIBM.SYSDUMMY1
  ORDER BY 2,3,4;
```

Turning a Template into SMART SQL (summary)

- Get an example of the output
- Identify each variable
- Determine source of each variable
- Double each existing tick in the input
- Delimit each literal with ticks
- Put CONCAT between variables / literals
- Format variables as needed
- Add “SELECT” and “UNION (ALL)” clauses
- Add “FROM” and “WHERE” clauses
- Add ordering columns and “ORDER BY” clause

27

Some additional note:

1) In general, a line of output is the SQL between two UNION ALLs.

2) Having consistent formatting and having one clause per line are both really helpful to making changes and troubleshooting your SQL.

3) Comment your SQL . . . The sanity you save may be your own!

4) Under most circumstances, UNION ALL is more appropriate than UNION.

5) Because of space issues in the presentation materials, I have left off the following items:

- CAST (. . . AS CHAR(80)) -- for the output text on all lines
- Keeping only one clause on a line
- Comments

Now that we have a checklist for building SMART SQL, let’s revisit our first example (counting the rows in the catalog).

A Very Simple SMART SQL statement Revisited

– Remember this . . .

```
Template:
  SELECT  COUNT(*) FROM "each catalog table";
```

```
SELECT
  CAST (
        ' SELECT COUNT(*) FROM SYSIBM.'
        CONCAT RTRIM(NAME) CONCAT ';'
  AS CHAR(72))
FROM SYSIBM.SYSTABLES
WHERE DBNAME = 'DSNDB06' AND TYPE = 'T'
ORDER BY 1;
```

28

How can we modify this SQL to give us a single number answer that is the total number of rows in the catalog?

A Very Simple SMART SQL Statement Revisited

```
SELECT ' SELECT SUM(A.CNT) FROM ('
,0, ' '
FROM SYSIBM.SYSDUMMY1
UNION ALL
SELECT
  CAST (' SELECT COUNT(*) "CNT" FROM SYSIBM.'
         CONCAT RTRIM(NAME) CONCAT ' UNION ALL'
  AS CHAR(72))
,1, NAME
FROM SYSIBM.SYSTABLES
WHERE DBNAME = 'DSNDB06' AND TYPE = 'T'
UNION ALL
SELECT ' SELECT 0 "CNT" '
       CONCAT 'FROM SYSIBM.SYSDUMMY1 ) AS A;'
,2, ' '
FROM SYSIBM.SYSDUMMY1
ORDER BY 2,3;
```

29

Here is one way to do it.

The SQL in yellow is what we started with -- except for the missing ';' inside the CAST clause (it was replaced with the ' UNION ALL ')

Some things to note:

1. We've turned our original 83 SQL select statements into a single nested table expression (NTE) which we are calling A.

2. When we created our NTE, we needed to label our output column– COUNT(*) – so that we could use the column in sum function of the outer select.

3. All of the text in the third UNION ALL section before the ")" has no effect on the final sum; it is only there to resolve the final "UNION ALL" in the second UNION ALL section. We could call this a No-operation clause that prevents a syntax error.

4. We use some non-outputted columns (in grey) to give us the correct order to our output.

A Very Simple SMART SQL Statement Revisited

```
ISREDDE2   DDBA02.SPUFI.OUT                                  Columns 00001 00072
 Command ===>                                                  Scroll ===> CSR
 ****** ***************************** Top of Data ******************************
 000001 ---------+---------+---------+---------+---------+---------+---------+--
 000002  SELECT SUM(A.CNT) FROM (
 000003  SELECT COUNT(*) "CNT" FROM SYSIBM.IPLIST UNION ALL
 000004  SELECT COUNT(*) "CNT" FROM SYSIBM.IPNAMES UNION ALL
 - - -  -  -  -  -  -  -  -  -  -  -  -  -  -  - 79 Line(s) not Displayed
 000084  SELECT COUNT(*) "CNT" FROM SYSIBM.SYSVTREE UNION ALL
 000085  SELECT COUNT(*) "CNT" FROM SYSIBM.USERNAMES UNION ALL
 000086  SELECT 0 "CNT" FROM SYSIBM.SYSDUMMY1 ) AS A;
 000087 ---------+---------+---------+---------+---------+---------+---------+--
 000088
 000089 ---------+---------+---------+---------+---------+---------+---------+--
 000090     8375508
 000091 DSNE610I NUMBER OF ROWS DISPLAYED IS 1
 000092 DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 100
 000093 ---------+---------+---------+---------+---------+---------+---------+--
 000094 ---------+---------+---------+---------+---------+---------+---------+--
 000095 DSNE617I COMMIT PERFORMED, SQLCODE IS 0
 000096 DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
 000097 ---------+---------+---------+---------+---------+---------+---------+--
 000098 DSNE601I SQL STATEMENTS ASSUMED TO BE BETWEEN COLUMNS 1 AND 72
 000099 DSNE620I NUMBER OF SQL STATEMENTS PROCESSED IS 1
 000100 DSNE621I NUMBER OF INPUT RECORDS READ IS 85
 000101 DSNE622I NUMBER OF OUTPUT RECORDS WRITTEN IS 101
 ****** *************************** Bottom of Data ***************************
```

30

You can see the results of the previous SQL here (lines 2 – 86), and the results of that SQL in line 90.

Gee . . . My catalog has 8,375,508 rows!

Note: in order to show the significant parts of the output, 79 lines of the output (line 5 through line 83) were excluded.

DB2 & SQL Features that enable SMART SQL

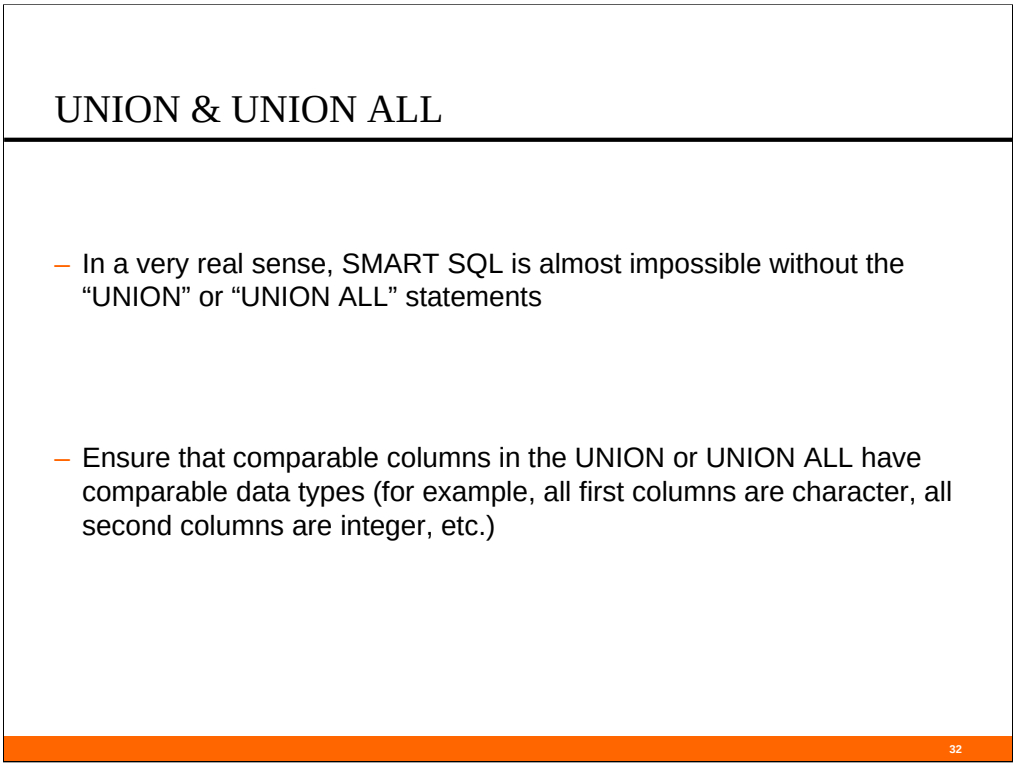
- UNION & UNION ALL
- Special Registers,
  - Current Timestamp & Current Server
- Scalar String Functions
  - CHAR, DIGITS, RTRIM, STRIP, SUBSTR & CONCAT
- Other Scalar Functions and UDFs
- Aggregate Functions
  - COUNT
  - Semi-Cartesian Joins & Driver Tables
- CASE

31

In our previous three examples, we have barely begun to highlight the DB2 and SQL features that are useful to SMART SQL.

Here are the various categories of DB2 and SQL features, and some of the most useful or interesting features or functions within that category.

We won't be going into any depth about any of the functions; they are well documented in the SQL Reference. We will, however, discuss some implications of those features and functions, and how to put them together for maximum effect. I encourage you to spend some time acquainting yourself with DB2 functions, especially the new ones.



Ideally, I like to have all my SMART SQL in one statement, even if I am doing six or eight different things within that statement. For example, I have a SMART SQL solution that produces "audit" tables. The SMART SQL statement is about 350 lines long, and the output builds SQL to do ALL of the following:

1. Drop existing triggers.
2. Drop the tablespaces containing the existing audit tables.
3. Create new tablespaces which will contain the new audit tables.
4. Create the new audit tables.
5. Create an INSET, UPDATE and DELETE trigger for each new audit table.

(We will discuss various bits and pieces of this solution later in the presentation)

The reason I like to have it all in one statement is that it makes the final editing easier. I end up with one header echo and one set of footer lines, rather than one set of each for each (separated) SQL statement. Furthermore, if I have set my page size large, I usually have fewer page headers when the SQL is all in one statement.

Special Registers

- Current Timestamp
  - Use this register to do processing based on elapsed time.

```
WHERE TS.STATSTIME <= CURRENT TIMESTAMP – 90 DAYS
```

(Use this filter to determine which tablespaces haven't been RUNSTATed in the last quarter)

- Current Server

```
SELECT  CASE WHEN CURRENT SERVER = 'DB2T'
             THEN '//STEPLIB  DD DISP=SHR,DSN=TEST.DB2.RUNLIB'
             WHEN CURRENT SERVER = 'DB2P'
             THEN '//STEPLIB  DD DISP=SHR,DSN=PROD.DB2.RUNLIB'
             ELSE '//** STEPLIB ERROR'
             END . . .
```

(Tailor the JCL based on the DB2 subsystem where the SQL is run)

33

There are 24 special registers in DB2; here is the complete list:

DB2 V 8 Special registers (from the SQL Reference, Chapter 2-15)

1. CURRENT APPLICATION ENCODING SCHEME
2. CURRENT CLIENT_ACCTNG
3. CURRENT CLIENT_APPLNAME
4. CURRENT CLIENT_USERID
5. CURRENT CLIENT_WRKSTNNAME
6. CURRENT DATE
7. CURRENT DEGREE
8. CURRENT LOCALE LC_CTYPE
9. CURRENT MEMBER
10. CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION
11. CURRENT OPTIMIZATION HINT
12. CURRENT PACKAGE PATH
13. CURRENT PACKAGESET
14. CURRENT PATH
15. CURRENT PRECISION
16. CURRENT REFRESH AGE
17. CURRENT RULES
18. CURRENT SCHEMA
19. CURRENT SERVER
20. CURRENT SQLID
21. CURRENT TIME
22. CURRENT TIMESTAMP
23. CURRENT TIMEZONE
24. USER

Some Useful Scalar String Functions

- DIGITS – turns a number into a character string. This is "old school" – (now) CHAR will do everything DIGITS does, and more besides.
- SUBSTR – Substring function (string, start, length)
- CONCAT – 'adds' two strings together. I prefer the spelled-out version to the double-pipe version, because it isn't ever adversely affected by CCSID code translations.
- RTRIM / LTRIM – removes trailing and leading blanks, respectively. Each can be replicated by the STRIP function.
- STRIP – this is like RTRIM & LTRIM combined and on steroids.
- CHAR – the 'general purpose' make-it-a-string function

34

This is NOT an exhaustive list of String functions, just the ones I find most useful (or my sentimental favorites).

I emphasize the string functions, because JCL is entirely text-based, so these functions are useful in creating JCL.

If you were only going to become proficient in only six string functions, the six I would choose are:

1. CHAR
2. CONCAT
3. SUBSTR
4. STRIP
5. LOCATE
6. REPLACE

But add another 10-12 (you should be able to figure out which ones) and you can do just about anything.

Other Useful Scalar Functions

- CAST – the 'all-data-types' conversion function
- COALESCE – gets rid of those pesky nulls.
- UPPER / LOWER – useful for mixed-case data
- TRANSLATE – this one is useful to deal with embedded blanks and apostrophes.
- The Functions that work on parts of date, time & timestamp columns.
  - (YEAR, MONTH, DAY,HOUR, MINUTE, SECOND and a host of others)

35

There has been an explosion of new IBM-supplied scalar functions in the last few releases of DB2:

- z/OS Version 5 – 24 Scalar functions

(fully 9 of them were date/time specific)

- z/OS Version 6 – 85 Scalar functions
- z/OS Version 7 – 111 Scalar functions
- z/OS Version 8 – 128 Scalar functions

Generally, the LUW folks got the new functionality one or two releases earlier than the z/OS folks.

A note about the TRANSLATE function: If the output is SQL and will be run in SPUFI, an unexpected apostrophe (and to a lesser extent, a quote) will wreck havoc in the output. The unexpected apostrophe instantly turns every literal after it into a non-literal, and vice-versa. The translate function can help overcome this potential problem.

I will talk about the issues with embedded blanks during the discussion of ISPF editing.

There are a lot more functions than we have time to discuss here, these are just some of the most useful to SMART SQL.

Spend a couple of hours getting to know the IBM-supplied scalar functions.

Aggregate Functions

- COUNT
  - Use this function (and a “semi-Cartesian join”) any time you need ordinal (counting) numbers.

```
SELECT
'//S' CONCAT DIGITS(CAST(COUNT(DISTINCT(B.NAME)) AS SMALLINT))
      CONCAT '    EXEC PGM=IKJEFT1B,DYNAMNBR=20'
 ,A.DBNAME, A.NAME, 05, ' '
FROM SYSIBM.SYSTABLESPACE A,
     SYSIBM.SYSTABLESPACE B
WHERE A.DBNAME= B.DBNAME AND A.NAME >= B.NAME . . .
GROUP BY A.DBNAME,A.NAME
```

(This SQL snippet gives you a TSO shell Step card with ascending step numbers– S00001 through S00???)

36

You may remember from school that a Cartesian join is one in which every row of the left table is paired with every row in the right table.

• In the old join syntax (Version 6 and before, but still supported), this was done by omitting the joining where clauses.

• In new syntax (only available in Version 7 and later), this is done by using an “always true” ON clause. I like to use “ON 1=1” .

Usually, any Cartesian join (which is also called a Cartesian product) is a BAD THING and is an indication that the SQL is incorrectly coded.

I want to introduce you to a variation on the Cartesian join I call a “semi-Cartesian join” . It can be used to make, for example, continuously ascending step numbers in JCL (see above). It works because my join criteria include a “>=“ condition (“<=“ also works).

Let’s take another little side trip – this time to a place where even Cartesian joins are a GOOD THING.

“Show Me the Tablespaces that Aren’t There”

```
1.   WITH TEMP1 (DIGIT) AS
2.    ( SELECT '0'  FROM SYSIBM.SYSDUMMY1  UNION ALL
3.      SELECT '1'  FROM SYSIBM.SYSDUMMY1  UNION ALL
4.      SELECT '2'  FROM SYSIBM.SYSDUMMY1  UNION ALL
5.      SELECT '3'  FROM SYSIBM.SYSDUMMY1  UNION ALL
6.      SELECT '4'  FROM SYSIBM.SYSDUMMY1  UNION ALL
7.      SELECT '5'  FROM SYSIBM.SYSDUMMY1  UNION ALL
8.      SELECT '6'  FROM SYSIBM.SYSDUMMY1  UNION ALL
9.      SELECT '7'  FROM SYSIBM.SYSDUMMY1  UNION ALL
10.     SELECT '8'  FROM SYSIBM.SYSDUMMY1  UNION ALL
11.     SELECT '9'  FROM SYSIBM.SYSDUMMY1 )
12.   SELECT ‘ZPLQ’ CONCAT T1.DIGIT CONCAT T2.DIGIT CONCAT T3.DIGIT
13.   FROM TEMP1 T1 JOIN TEMP1 T2 ON 1=1 JOIN TEMP1 T3 ON 1=1
14.   WHERE T1.DIGIT CONCAT T2.DIGIT CONCAT T3.DIGIT NOT IN
15.    (SELECT SUBSTR(NAME,5,3) FROM SYSIBM.SYSTABLESPACE WHERE DBNAME LIKE 'ZPLQ%')
16.      AND T1.DIGIT CONCAT T2.DIGIT CONCAT T3.DIGIT <=
17.    (SELECT MAX(SUBSTR(NAME,5,3)) FROM SYSIBM.SYSTABLESPACE WHERE DBNAME LIKE 'ZPLQ%')
18.   AND T1.DIGIT CONCAT T2.DIGIT CONCAT T3.DIGIT >=
19.    (SELECT MIN(SUBSTR(NAME,5,3)) FROM SYSIBM.SYSTABLESPACE WHERE DBNAME LIKE 'ZPLQ%')
20.  ORDER BY 1;
```

37

This is an example that shows even Cartesian products have their place.

The business problem: Show me the gaps in my tablespace usage for a particular database.

It works because I know a lot about my environment:

- The LIKE ‘ZPLQ%’ only returns one database
- Shop naming standards (which are followed in this database) have a sequence number in characters 5-7 of the tablespace name.

I am able to set up a one-column ‘virtual table’ containing the values “000” through “999” (in the SQL lines 1 – 13).

I compare that ‘virtual table’ with character 5-7 of my tablespace names, and put into the result set only the ‘virtual table’ values that don’t match the tablespace table values (in SQL lines 14-15)

SQL lines 16-17 filter out the values that are greater than the maximum tablespace name value. SQL lines 18-19 filter out the values that are less than the minimum tablespace name value.

My output adds the literal “ZPLQ” to the result set to give it some context.

"Show Me the Tablespaces that Aren't There"

```
ISREDDE3   DDBA02.SPUFI.OUT                              Columns 00001 00072
Command ===>                                               Scroll ===> CSR
****** ***************************** Top of Data ***************************
000001 ZPLQ009
000002 ZPLQ012
000003 ZPLQ013
000004 ZPLQ014
000005 ZPLQ024
- - -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  - 41 Line(s) not Displayed
000047 ZPLQ177
000048 ZPLQ178
000049 ZPLQ184
000050 DSNE610I NUMBER OF ROWS DISPLAYED IS 49
```

Min Tablespace Name: ZPLQ001
Max Tablespace Name: ZPLQ195

Count of Tablespaces: 146
Plus Gaps + 49
Total Tablespaces 195

38

Sure enough, it works! (and the counts tally, too!)

----------(text for next note page)

Assume a single database with three tablespaces (called X, Y & Z). Using the join criteria on the next slide – ((A semi-Cartesian B) Cartesian C) – and no group by clause, we get 18 rows of output:

A	B	C	A	B	C	A	B	C
X	X	X	Y	X	X	Z	X	X
X	X	Y	Y	X	Y	Z	X	Y
X	X	Z	Y	X	Z	Z	X	Z
			Y	Y	X	Z	Y	X
			Y	Y	Y	Z	Y	Y
			Y	Y	Z	Z	Y	Z
						Z	Z	X
						Z	Z	y
						Z	Z	Z

Adding the group by A gives us this result set:

A	Count (Distinct (B))	Count(*) / (Count (Distinct (B))
X	1	3
Y	2	3
Z	3	3

Notice that Count (Distinct (B)) gives the same result as the Count(*) of a two-table semi-Cartesian join, and that Count (*) / Count (Distinct (B)) is always equal to the number of tablespaces we started with.

Another Cartesian Product example

```
1.  SELECT
2.   CASE WHEN COUNT(DISTINCT(B.NAME)) = 1
3.         AND COUNT(DISTINCT(B.NAME)) < COUNT(*)/COUNT(DISTINCT(B.NAME))
4.        THEN '//         VOL=(,RETAIN,,99)'
5.        WHEN COUNT(DISTINCT(B.NAME)) = 1
6.         AND COUNT(DISTINCT(B.NAME)) = COUNT(*)/COUNT(DISTINCT(B.NAME))
7.        THEN '//         VOL=(,,,99)'
8.        WHEN COUNT(DISTINCT(B.NAME)) > 1
9.         AND COUNT(DISTINCT(B.NAME)) < COUNT(*)/COUNT(DISTINCT(B.NAME))
10.       THEN '//         VOL=(,RETAIN,,99,REF=*.S'
11.            CONCAT DIGITS(CAST(COUNT(DISTINCT(B.NAME)) -1 AS SMALLINT))
12.            CONCAT '.SYSREC00)'
13.       WHEN COUNT(DISTINCT(B.NAME)) = COUNT(*)/ COUNT(DISTINCT(B.NAME))
14.       THEN '//         VOL=(,,,99,REF=*.S'
15.            CONCAT DIGITS(CAST(COUNT(DISTINCT(B.NAME)) -1 AS SMALLINT))
16.            CONCAT '.SYSREC00)'
17.       ELSE '//***** VOLUME ERROR ************'
18.       END
19.  ,A.DBNAME, A.NAME, 13
20. FROM SYSTABLESPACE A, SYSTABLESPACE B, SYSTABLESPACE C
21. WHERE A.DBNAME= B.DBNAME AND A.NAME >= B.NAME AND A.DBNAME= C.DBNAME
22. GROUP BY A.DBNAME,A.NAME
```

39

Here is another example of a situation where the only solution to a business problem involves a Cartesian join. Assume you want to do tape-stacking in a job that creates multiple tapes. In order to efficiently use your tapes and tape drives, the all steps that use the tape (except the last) should retain the tape on the drive for the next step. Furthermore, all steps that use the tape (except the first) should refer back to the dataset of the previous step (NOT the first dataset on the tape, in case the stacked datasets use more than one tape). In English:

- Lines 2 - 4: The dataset we are outputting is the first dataset on the tape, but not the last – no refer back, yes retain.
- Lines 5 - 7: The dataset we are outputting is the first dataset on the tape, and also ot the last – no refer back, no retain.
- Lines 8 - 12: The dataset we are outputting is not the first dataset on the tape, and not the last – refer back to previous dataset, yes retain.
- Lines 13 - 16: The dataset we are outputting is the last dataset on the tape (it is implied that this dataset is not the first, otherwise and earlier WHEN would have been chosen instead of this one) – refer back to previous dataset, no retain.
- Line 17 – there's a problem, show it as an error.

But the really interesting part of this example is the join.

Copy A of SYSIBM.SYSTABLESPACES is semi-Cartesian-joined to copy B, and the result of that join participates in a full Cartesian join with copy C. It would be much simpler to use "AND A.NAME = (SELECT MAX(NAME) FROM . . ." , but because we can't use sub-selects in a WHEN clause (or so, by extension, in a CASE statement), we can't use the sub-select here. This three-table join allows us to get the same result. It uses the same logic that allows the optimizer to change a sub-select into a join under certain circumstances. Further explanation (but not a full mathematical proof) can be found on the previous note page.

"Driver Tables"

- Here is some potentially useful driver table columns:

1. Database name
2. Tablespace name
3. Creator / Owner
4. Table name
5. "Object" name
6. Sequence number of object in full list
7. Sequence number of object in group (Database, Tablespace, etc.)
8. Number of objects in group (Database, Tablespace, etc.)
   (max object or count of objects in the group)
9. Referential Integrity Level

40

Cartesian and semi-Cartesian joins are resource intensive, and so are not generally recommended for large tables. This can be an issue when using SMART SQL techniques, because sometimes the only way to order the output is to repeatedly do these kinds of joins. One way to avoid this situation is to create a special table to hold the results of that semi-Cartesian join and reference that special table, rather than doing the join over and over. We could call this a driver table, as it "drives" the order of the output.

Some potentially useful driver tables columns are shown in the slide. In general, the driver table's columns should support one of these three purposes: 1) identify or help identify the object, 2) provide sequencing data, or 3) provide other data that eliminates the need for additional joins.

Until now, the discussion has focused on using a driver table to avoid Cartesian joins, but that is not the only reason these types of tables might be used. Two other situations come to mind:

- Say we're doing data movement using between two databases that have DB2-managed Referential Integrity (RI). We would want delete the data in the target database from each child table before we deleted the data in their parent table ("from the bottom up") and we would want to insert the data for all the parent tables before we inserted data to the child table ("from the top down") . There is an SQL-only solution to this problem that could be incorporated into the SMART SQL, but the SQL to do this RI-structure ordering is long, convoluted and resource intensive. A better option is to make a driver table and use that table for the SMART SQL.

- Another situation where a driver table is useful is for any non-standard collation sequence. I like to process named objects in alphabetical order, and numbered objects in sequence-number order. But for any other ordering, it's usually easier to use a driver table & build the sequencing there.

It's important to have the driver table be at the correct level of granularity. For instance, if you are creating utility JCL that would have done a semi-Cartesian join at the tablespace level, then the driver table should be unique at the tablespace level as well.

CASE Statement

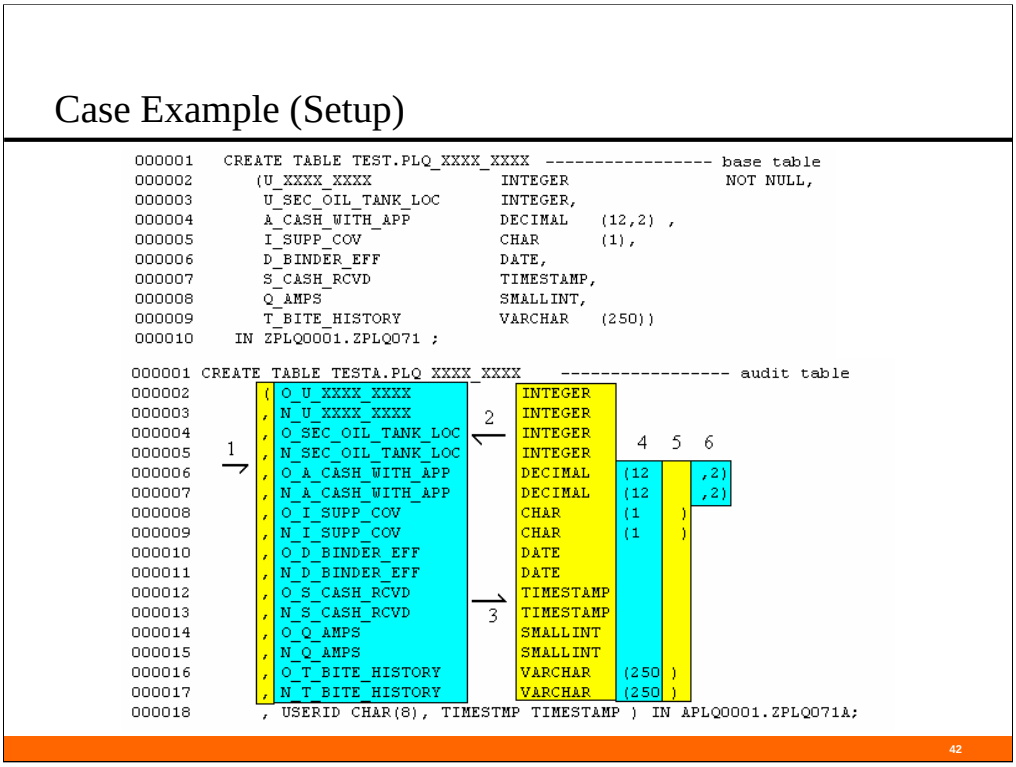
- Advantages of the CASE statement
  - A CASE statement can reduce the number of UNION ALL clauses needed to create an output.
  - CASE statements can be nested for even more flexibility.
- Limitations
  - The CASE statement uses a WHEN clause, not a WHERE clause.
  - The WHEN clause doesn't have as many options as the WHERE clause.

41

You can't use IN lists, or sub-selects in a WHEN clause.

You can get a run-time error under certain circumstances if none of your WHEN clauses evaluates to "TRUE" and you haven't coded an ELSE clause. Always code an ELSE clause!

Even with the limitations of the WHEN clause, the CASE statement is very useful for creating SMART SQL.



This example requires some setup.

(It comes from the “Create a set of audit tables” solution. The full solution requires about 350 lines of SQL. This example is 40 lines long, and highlights the use of the CASE statement)

Assume that we have an existing table with the structure like that on the top part of the slide. We want to create an audit table that has an old value column and a new value column for each existing column of the existing table.

We will distinguish the audit table columns by appending an “O_” (for old) or “N_” (for new) to the existing column name, unless that name is 17 or 18 characters long, in which case we will replace the first two characters of the name with “O_”’ or “N_”. Generate the “CREATE TABLE . . .” SQL for this audit table.

A full example of the desired output is at the bottom of the slide, but we will be focusing on the column definitions (all lines but the first and last lines of the lower “CREATE TABLE . . .” statement).

It helps to think of the output NOT as a single 72-character text column, but instead as 6 individual sub-columns within a larger column. In this example, the sub-columns are 1) the open-parenthesis or comma, 2) the column name, 3) the column type, 4) the (optional) column length, 5) the (optional) close-parenthesis and 6) the (optional) column scale and close-parenthesis.

Each sub-column has its own case statement, and is highlighted in either yellow or blue on the slide.

Note: On the slide, the spacing on the expected output has been adjusted to highlight the sub-columns. The SQL will not reproduce this spacing.

Case Example (continued)

```
000001 SELECT
000002  CAST(
000003   CASE WHEN CO.COLNO = 1 THEN '      (' ELSE '      ,' END CONCAT
000004   CASE WHEN LENGTH(RTRIM(CO.NAME)) <= 16
000005        THEN         ' O_' CONCAT SUBSTR(CO.NAME,1,18) CONCAT '    '
000006        ELSE         ' O_' CONCAT SUBSTR(CO.NAME,3,16) CONCAT '    ' END
000007   CONCAT
000008   CASE WHEN CO.COLTYPE='TIMESTMP' THEN 'TIMESTAMP' ELSE CO.COLTYPE END
000009   CONCAT
000010   CASE WHEN CO.COLTYPE='CHAR' OR CO.COLTYPE='DECIMAL' OR
000011             CO.COLTYPE='FLOAT' OR CO.COLTYPE='VARCHAR'
000012        THEN '(' CONCAT STRIP(DIGITS(CO.LENGTH),L,'0') ELSE ' ' END
000013   CONCAT
000014   CASE WHEN CO.COLTYPE='CHAR' OR CO.COLTYPE='FLOAT' OR
000015             CO.COLTYPE='VARCHAR'
000016        THEN ') '
000017        WHEN CO.COLTYPE='DECIMAL' AND SCALE <> 0
000018        THEN ',' CONCAT STRIP(DIGITS(CO.SCALE),L,'0') CONCAT ')'
000019        WHEN CO.COLTYPE='DECIMAL' AND SCALE = 0
000020        THEN ',' CONCAT '0)' ELSE ' ' END
000021  AS CHAR(80))
```

43

- Line 3 gives us either an open parenthesis or a comma, depending on whether the column (in the base table) is the first column or not. Note the leading blanks on this line. It's no harder to generate nicely-indented, readable output, than it is to generate syntactically accurate but unreadable output.
- Lines 4 – 6 will create the "old value" column name (prefixed with an "O_"). These lines will either append the "O_" to the front of the existing column name, or replace the first two characters with "O_", depending on the length of the (base table) column name.
- Line 8 adds the column type to the output text. Note that the only column type that is not passed directly to the output is "TIMESTMP". It is translated to "TIMESTAMP" in the output.
- Lines 10 – 12 puts an open-parenthesis and the column length to the output, if the (base table) column type is one that requires a length parameter. Note that we can use the STRIP function, as the length value is required NOT to be zero for column types that require a length.
- Lines 14 – 16 put a close-parenthesis on CHAR, FLOAT & VARCHAR columns.
- Lines 17 – 20 put the column scale and a close-parenthesis on DECIMAL columns. Notice how the STRIP here is used differently than in it is in line 12. (Why?)
- In the set-up for this example, I said that there were 6 sub-columns, and that each sub-column has its own CASE statement. But if you look, the word CASE only appears 5 times – what's up with that? Well, we can see that all the WHEN conditions in lines 14 – 20 are mutually exclusive, so we can combine them into one CASE statement. It would be equally valid to add "ELSE ' ' END CASE" between lines 16 and 17; the statement would still be syntactically correct. But why add more text than we need?

Note: All SQL clauses except the SELECT clause have been removed from the slide in order to save space. The other SQL clauses for this SELECT are:

```
,3 , CO.TBCREATOR, CO.TBNAME , CO.COLNO, 01 --- ordering columns
FROM SYSIBM.SYSTABLES TB, SYSIBM.SYSCOLUMNS CO
WHERE TB.DBNAME LIKE 'ZPLQ%' AND TB.TYPE ='T'
  AND CO.TBCREATOR = TB.CREATOR AND CO.TBNAME = TB.NAME
UNION ALL
```

Case Example (continued)

```
000028 SELECT
000029  CAST(

000030   CASE WHEN LENGTH(RTRIM(CO.NAME)) <= 16
000031        THEN '      , N_' CONCAT SUBSTR(CO.NAME,1,18) CONCAT '    '
000032        ELSE '      , N_' CONCAT SUBSTR(CO.NAME,3,16) CONCAT '    ' END
000033   CONCAT
000034   CASE WHEN CO.COLTYPE='TIMESTMP' THEN 'TIMESTAMP' ELSE CO.COLTYPE END
000035   CONCAT
000036   CASE WHEN CO.COLTYPE='CHAR' OR CO.COLTYPE='DECIMAL' OR
000037             CO.COLTYPE='FLOAT' OR CO.COLTYPE='VARCHAR'
000038        THEN '(' CONCAT STRIP(DIGITS(CO.LENGTH),L,'0') ELSE ' ' END
000039   CONCAT
000040   CASE WHEN CO.COLTYPE='CHAR' OR CO.COLTYPE='FLOAT' OR
000041             CO.COLTYPE='VARCHAR'
000042        THEN ') '
000043        WHEN CO.COLTYPE='DECIMAL' AND SCALE <> 0
000044        THEN ',' CONCAT STRIP(DIGITS(CO.SCALE),L,'0') CONCAT ')'
000045        WHEN CO.COLTYPE='DECIMAL' AND SCALE = 0
000046        THEN ',' CONCAT '0)' ELSE ' ' END
000047  AS CHAR(80))
```

44

This slide looks <u>almost exactly</u> like the last one!

The first CASE statement (on line 3 in the last slide) is gone – I have replaced it with the "blank line" between lines 29 and 30 on this slide. The comma from the else path of that CASE statement on line 3 has been appended to the beginning of the first literal on lines 31 and 32.

This makes sense, as the output "CREATE TABLE. . . " statement MUST already have a "first column" (that column starts with a "O_"), so the CASE statement that would create an "open-parenthesis" column is not needed here.

The ONLY other difference on the entire slide is that the "O_" has been changed to an "N_".

This slide illustrates that it's comparatively easy to reuse code in SMART SQL.

Notes: All SQL clauses except the SELECT clause have been removed from the slide in order to save space. The spacing of the SQL on this slide has also been adjusted to highlight how close this snippet of SQL matches that of the last slide.

The other SQL clauses for this SELECT are:

```
,3 , CO.TBCREATOR, CO.TBNAME , CO.COLNO, 02  --- ordering columns
FROM SYSIBM.SYSTABLES TB, SYSIBM.SYSCOLUMNS CO
WHERE TB.DBNAME LIKE 'ZPLQ%'
  AND TB.TYPE ='T'
  AND CO.TBCREATOR = TB.CREATOR AND CO.TBNAME = TB.NAME
UNION ALL
```

Case Example (continued)

```
ISREDDE3   DDBA02.SPUFI.OUT                    Columns 00001 00072
Command ===>                                      Scroll ===> CSR
001002       ( O_U_XXXX_XXXX          INTEGER
001003       , N_U_XXXX_XXXX          INTEGER
001004       , O_SEC_OIL_TANK_LOC     INTEGER
001005       , N_SEC_OIL_TANK_LOC     INTEGER
001006       , O_A_CASH_WITH_APP      DECIMAL (12,2)
001007       , N_A_CASH_WITH_APP      DECIMAL (12,2)
001008       , O_I_SUPP_COV           CHAR    (1)
001009       , N_I_SUPP_COV           CHAR    (1)
001010       , O_D_BINDER_EFF         DATE
001011       , N_D_BINDER_EFF         DATE
001012       , O_S_CASH_RCVD          TIMESTAMP
001013       , N_S_CASH_RCVD          TIMESTAMP
001014       , O_Q_AMPS               SMALLINT
001015       , N_Q_AMPS               SMALLINT
001016       , O_T_BITE_HISTORY       VARCHAR (250)
001017       , N_T_BITE_HISTORY       VARCHAR (250)
```


This is the (partial) output result of the SQL from the previous two slides.

ISPF Editing Tricks

- ISPF has one of the best editors ever created.
  - It allows for excluding and showing lines, and processing based on whether the line is excluded or not.
  - It allows for labeling lines, and using those labels to limit processing.
  - It allows for setting card column boundaries.
  - It recognizes card columns within the file, and allows processing based on the card column position of the text.
  - It allows for global changes (by excluded or shown lines, card columns, and labels).
  - Allows for generic searches and editing (using picture clauses)
  - Text split and especially Text Flow (by card column and indenting)

46

This is an area we could spend an entire session on, not just one slide.

It's definitely worth your while to learn about ISPF editing, and there are over 50 tutorial panels that are as close as your F1 key

Let me talk about a technique that can simplify the SQL at the cost of additional editing of the output. For smaller SQL outputs (about 1,000 characters or less), you can avoid all the ordering columns and let the SQL get as wide as it needs to be. Remove the header lines, page headers and footer lines. Then, globally change every blank into an unused character (I like to use the exclamation point as my blank surrogate). The editing command would look something like this: "C ' ' ! ALL". Then, every 50 – 70 characters, find a column that only contains exclamation points. Selectively change the "!"s in those columns back to blanks using an editing command like "C ' ' ! ALL 50" or "C ' ' ! ALL 110" or "C ' ' ! ALL 172". There are two restrictions on where the blank columns can be. First, be sure not to create a blank column inside a literal (between two apostrophes), and second, leave at least 75 blanks after the final semi-colon. These blank columns are where the text will wrap to the next line. Once the output is broken into 50-70 character wide pieces, type TF (text flow) in the number area of line 1, and watch the text break at the blank columns. You should now have all your text in cc 1 – 70. Finally, change all the remaining "!"s back to blanks. The output should be ready to be used as SPUFI input.

Troubleshooting SMART SQL

- SQL Errors
  - Any time you get a negative SQL code, temporarily put a semi-colon before each UNION ALL. This allows you to isolate the offending UNION ALL section.
- Disordered Output
  - The first thing to do with these errors is to widen your output dataset to see the ordering columns. Usually this will allow you to focus in on the problem UNION ALL section(s).
- Too Much Output
  - This usually indicates that one of your assumptions is invalid. Most of the time there is either an unintended Cartesian join, or something you assume to be unique is not. Additional WHERE clauses can sometimes resolve these situations.

47

This is another area we could spend a entire session on, so we'll just hit the highlights:

In general:

- Having well-formatted SQL, with consistent indenting and one clause per line will make troubleshooting much easier. This allows you to, for instance, exclude all lines and only show, say, the ordering column lines or the "UNION ALL" lines.
- Comment out parts of the SQL to simplify the SQL and identify the offending code. The SQL error(s) will be in the last line(s) that were commented out before the SQL stops having the error.

SQL Errors – three SQL codes come up the most often:

- 'illegal symbol' (-104) – look at the SQL immediately before the identified token (at most a line or two before it). The error will be there.
- 'invalid argument' (-171) – look at your data types for the one(s) that are of the wrong type for the function; comment out arguments until you identify the offending one.
- 'ungrouped aggregate function' (-122) – look at your GROUP BY and compare those columns to the ones in the SELECT. Either remove the ungrouped column, or use an meaningless aggregate function on the column (e.g. a MAX function where all the values are the same)

Disordered Output:

- The slide pretty much says it all.

Too Much Output:

- Use a separate SELECT to count the number of rows to expect for a particular piece of SQL. If the number of rows in your SMART SQL is different from rows expected from your count, the erroneous assumption lies in that area. Showing the ordering columns may also help.

The Audience Participation Portion of the Presentation

This is your opportunity to ask about business problems that you deal with in your shop.

(Don't worry . . . I have other examples if you don't)

48

I expect that there will be one of four outcomes from your question:

1) I have an example in the handout and we will discuss it.
2) I have an example, but it's not in the handout, so we will discuss it in general terms, but I will defer the details of your question to the speakers corner.
3) I don't have an example, but the techniques of SMART SQL are applicable to the question or problem, and we will discuss it in general terms.
4) It may be that the question or problem is not amenable to the techniques of SMART SQL.

Audit Table Trigger Example

```
CREATE TRIGGER ZPLQ020D AFTER DELETE  ON XXXX.PLQ_XXXX_XXX_XXX
  REFERENCING OLD AS OROW FOR EACH ROW
  MODE DB2SQL INSERT INTO XXXXA.XXX_XXXX_XXX_XXX VALUES (
     OROW.U_ADDL_INT_TYP, NULL
    ,CURRENT SQLID, CURRENT TIMESTAMP );

000001   SELECT '  CREATE TRIGGER ' CONCAT RTRIM(TSNAME) CONCAT 'D' CONCAT
000002   ' AFTER DELETE  ON ' CONCAT RTRIM(CREATOR) CONCAT '.' CONCAT
000003     RTRIM(NAME)                   ,6   , CREATOR, NAME , 00, 01
000004   FROM SYSIBM.SYSTABLES WHERE DBNAME LIKE 'ZPLQ%' AND TYPE ='T'
000005   UNION ALL
000006   SELECT '    REFERENCING OLD AS OROW FOR EACH ROW '
000007                                  ,6   , CREATOR, NAME , 00, 02
000008   FROM SYSIBM.SYSTABLES WHERE DBNAME LIKE 'ZPLQ%' AND TYPE ='T'
000009   UNION ALL
000010   SELECT '    MODE DB2SQL INSERT INTO ' CONCAT RTRIM(CREATOR) CONCAT
000011   'A.' CONCAT RTRIM(NAME) CONCAT ' VALUES ('
000012                                  ,6   , CREATOR, NAME , 00, 03
000013   FROM SYSIBM.SYSTABLES WHERE DBNAME LIKE 'ZPLQ%' AND TYPE ='T'
000014   UNION ALL SELECT CASE WHEN CO.COLNO = 1
000015               THEN '       OROW.' CONCAT RTRIM(CO.NAME) CONCAT ', NULL'
000016               ELSE '      ,OROW.' CONCAT RTRIM(CO.NAME) CONCAT ', NULL'
000017               END               ,6   , CO.TBCREATOR, CO.TBNAME , CO.COLNO, 01
000018   FROM SYSIBM.SYSTABLES TB, SYSIBM.SYSCOLUMNS CO
000019   WHERE TB.DBNAME LIKE 'ZPLQ%' AND TB.TYPE ='T'
000020     AND  CO.TBCREATOR = TB.CREATOR AND CO.TBNAME = TB.NAME
000021   UNION ALL SELECT '      ,CURRENT SQLID, CURRENT TIMESTAMP );'
000022                                  ,6   , CREATOR, NAME , 9999, 01
000023   FROM SYSIBM.SYSTABLES WHERE DBNAME LIKE 'ZPLQ%' AND TYPE ='T'
000024   UNION ALL SELECT '   COMMIT;'   ,6   , CREATOR, NAME , 9999, 02
000025   FROM SYSIBM.SYSTABLES WHERE DBNAME LIKE 'ZPLQ%' AND TYPE ='T'
000026   ORDER BY 2,3,4,5,6;
```

49

A template of what we want to create is above, the SQL to do it is below.

Things to note:

This is a delete trigger; only the old rows need to be populated with data values. By shop convention, all new column data values are NULL.

Lines 15 – 16 distinguish between the first column in the table and all other columns. Because of the organization of the table, an N_???-named column always follows each O_???-named column, so we can always follow the data value of the O_???-named column with a “NULL” for the N_???-named column.

It’s easy to turn this SQL into an INSERT trigger. Change:

1. the “OLD” to “NEW”
2. “OROW” to “NROW”
3. The order of the “NULL” and the column name in lines 15 & 16
4. the “D” suffix of the trigger name to an “I” -- meaningful names, very important!
5. The value of the first ordering column (here, it’s 6) to, say, 7.

And it’s just as easy to construct an update trigger (but the changes needed are left to you as an learning exercise).

REBIND Example

```
000001   SELECT '//JOBCARD  JOB ACCT#,''REBIND'',CLASS=C,MSGCLASS=X'
000002    ,' '     ,' ' , 00 FROM SYSIBM.SYSDUMMY1
000003   UNION ALL SELECT '//S010     EXEC PGM=IKJEFT1B,DYNAMNBR=20,REGION=0M'
000004    ,' '     ,' ' , 01 FROM SYSIBM.SYSDUMMY1
000005   UNION ALL SELECT '//STEPLIB  DD DISP=SHR,DSN=TEST.DB2.RUNLIB'
000006    ,' '     ,' ' , 02 FROM SYSIBM.SYSDUMMY1
000007   UNION ALL
000008   SELECT '//SYSPRINT DD  SYSOUT=*' ,' ' ,' ' , 03 FROM SYSIBM.SYSDUMMY1
000009   UNION ALL
000010   SELECT '//SYSTSPRT DD  SYSOUT=*' ,' ' ,' ' , 04 FROM SYSIBM.SYSDUMMY1
000011   UNION ALL
000012   SELECT '//SYSTSIN    DD  *'      ,' ' ,' ' , 05 FROM SYSIBM.SYSDUMMY1
000013   UNION ALL
000014   SELECT '  DSN SYSTEM(DB2T)' ,' '     ,' ' , 06 FROM SYSIBM.SYSDUMMY1
000015   UNION ALL   SELECT '  REBIND PLAN(' CONCAT RTRIM(NAME) CONCAT
000016                      ') DBPROTOCOL(DRDA) ' ,'01'    , NAME, 07
000017               FROM SYSIBM.SYSPLAN WHERE DBPROTOCOL = 'P'
000018   UNION ALL   SELECT '  REBIND PACKAGE(' CONCAT RTRIM(COLLID) CONCAT '.'
000019                       CONCAT RTRIM(NAME) CONCAT
000020                          CASE WHEN VERSION <> ' '
000021                               THEN '.('  CONCAT RTRIM(CHAR(VERSION)) CONCAT ')'
000022                               ELSE '' END
000023                       CONCAT ') DBPROTOCOL(DRDA) ' ,'02'    , NAME, 08
000024               FROM SYSIBM.SYSPACKAGE WHERE DBPROTOCOL ='P'
000025   UNION ALL SELECT '/* ' ,'ZZZZ','ZZZZ' , 99 FROM SYSIBM.SYSDUMMY1
000026   ORDER BY 2,3,4;
```

50

We had to do a mass REBIND of every plan and package to use DRDA protocol rather than PRIVATE protocol – 25,000 plans and packages in test and 8,000 plans and packages in production. Here is SQL to generate the JCL to do it.

Things to note:

1. Lines 1 – 12 & 25, standard batch TSO JCL datasets and JCL cards -- we've seen these before.
2. Lines 15-17, plan rebinds. Notice the WHERE clause – we are only doing what is necessary to accomplish the goal.
3. Lines 18-24, package rebinds. Use the version token if the existing package has one, if not, don't (lines 20 – 22). Same WHERE clause as line 17.

Mass DCLGEN Example

```
1.  DCLGEN TABLE(XXX.XXXXXXXXXXX)  -
2.         LIBRARY(XXXX.GCC.DCL(XXXXXXX))-
3.         ACTION(REPLACE) APOST

1.  UNION ALL
2.      SELECT '  DCLGEN TABLE (' CONCAT  RTRIM(CREATOR) CONCAT '.' CONCAT
3.          RTRIM(NAME) CONCAT ') -' , CREATOR , TSNAME , 07
4.      FROM SYSIBM.SYSTABLES WHERE NAME LIKE 'PLQ%' AND TYPE ='T'
5.    UNION ALL
6.      SELECT '      LIBRARY(''' CONCAT
7.          CASE WHEN CREATOR = 'TST3' THEN 'TEST'
8.               WHEN CREATOR = 'TST4' THEN 'TEST' ELSE RTRIM(CREATOR) END
9.          CONCAT
10.         CASE WHEN CREATOR = 'TST3' THEN '.' CONCAT 'XXX.DCL3('
11.              WHEN CREATOR = 'TST4' THEN '.' CONCAT 'XXX.DCL4('
12.                                   ELSE '.' CONCAT 'XXX.DCL(' END
13.         CONCAT RTRIM(SUBSTR(TSNAME,2,7)) CONCAT ')'') -'
14.       , CREATOR , TSNAME , 08
15.     FROM SYSIBM.SYSTABLES WHERE NAME LIKE 'PLQ%' AND TYPE ='T'
16.   UNION ALL
17.     SELECT '      ACTION (REPLACE) APOST' , CREATOR , TSNAME , 09
18.     FROM SYSIBM.SYSTABLES WHERE NAME LIKE 'PLQ%' AND TYPE ='T'
```

51

We will reuse the first fourteen lines of the previous example; assume they are present just above what is shown on this slide. This slide shows how we can reuse SMART SQL – steal something close, file off the serial numbers, and modify it for a new purpose.

I have one application that is doing multiple parallel development efforts, each with a copy of the database. In the past, we have had problems with DCLGEN members getting 'lost' or mysteriously 'changed'. This is also an application where the structures are not necessarily the same across all of the different development environments. One development team may have additional tables not found in the other development environments. Furthermore, our shop naming standards did not envision multiple simultaneous development efforts, so we have had to contort our library names to accommodate this situation.

This SMART SQL is my effort to ensure that all of the DCLGEN members for all environments are present, and match the structure of the tables. I run this periodically to refresh the various DCLGEN libraries. Using the CASE statements, I can even make sure the members go into the right library!

The template is above, the SQL to do it below. The library name contortions are in lines 7 – 12.

"INSERT INTO . . . SELECT FROM . . ." Example

```
1.  SELECT '  INSERT INTO ' CONCAT RTRIM(T1.CREATOR) CONCAT 'O.' CONCAT
2.      RTRIM(T1.NAME) ,4,T1.NAME, 10, 00, ' '
3.   FROM SYSIBM.SYSTABLES T1
4.   WHERE T1.DBNAME = 'ZPLQ0001' AND T1.TYPE = 'T'
5.     AND UPPER(SUBSTR(T1.REMARKS,1,3)) <> 'DOM'
6.   UNION ALL
7.   SELECT '      SELECT * FROM ' CONCAT RTRIM(T1.CREATOR) CONCAT '.'
8.      CONCAT RTRIM(T1.NAME) CONCAT ';' ,4,T1.NAME, 10, 01, ' '
9.   FROM SYSIBM.SYSTABLES T1
10.  WHERE T1.DBNAME = 'ZPLQ0001' AND T1.TYPE = 'T'
11.    AND UPPER(SUBSTR(T1.REMARKS,1,3)) <> 'DOM'
12.  UNION ALL
13.  SELECT '         ' ,4,T1.NAME, 10, 02, ' '
14.  FROM SYSIBM.SYSTABLES T1
15.  WHERE T1.DBNAME = 'ZPLQ0001' AND T1.TYPE = 'T'
16.   AND UPPER(SUBSTR(T1.REMARKS,1,3)) <> 'DOM'
17.  ORDER BY 2,3,4,5,6;
```

52

This is the simplest of the all-SQL data migrations. It assumes the structures of the source table and the target table are the same, and there is no DB2-managed RI. When these conditions are met, we can do

"INSERT INTO target_table_name SELECT * FROM source_table_name ;"

Things to note:

1. In this example, the source and target tables differ only in the creator name (the target table creator has an "O" suffix).
2. We are using an unusual filtering criterion (lines 5, 11 and 16). One of the powerful aspects of SMART SQL is that we can use this kind of "odd" criterion in our SQL.
3. To make the output more readable, there is a blank line between inserts (lines 13 – 16). We could also add a "COMMIT;" line to the output in the same way.

“ALTER (BUFFERPOOL)” Example

```
000014 ---- CATALOG 4K TABLESPACES (SHOULD BE IN BP0)
000015    SELECT '  ALTER TABLESPACE ' CONCAT RTRIM(DBNAME) CONCAT'.' CONCAT
000016           RTRIM(NAME) CONCAT ' BUFFERPOOL BP0; -- ' CONCAT BPOOL
000017    FROM DB2T.SYSIBM.SYSTABLESPACE
000018    WHERE (  (    DBNAME LIKE 'DSN%'
000019               AND DBNAME <> 'DSNDB07'
000020               AND DBNAME NOT LIKE 'DSN8%'
000021               AND DBNAME <> 'DSNDB04'
000022              ) OR DBNAME = 'CC390'
000023          ) AND BPOOL<>'BP0' AND PGSIZE = 4
000024    UNION ALL
000025 ---- DB2 SORTWORK 4K TABLESPACES (SHOULD BE IN BP1)
000026    SELECT '  ALTER TABLESPACE ' CONCAT RTRIM(DBNAME) CONCAT'.' CONCAT
000027           RTRIM(NAME) CONCAT ' BUFFERPOOL BP1; -- ' CONCAT BPOOL
000028    FROM DB2T.SYSIBM.SYSTABLESPACE
000029    WHERE (DBNAME = 'DSNDB07' AND BPOOL<>'BP1' AND PGSIZE = 4)
000030    UNION ALL
```

53

Every month or so, I run this SQL to make sure that all my tablespaces and indexes are in their proper buffer pool. The complete SQL is almost 500 lines long; this is only a small part of the whole SQL. The output is in the form of “ALTER TABLESPACE . . . “ DDL, but only for objects that are NOT in their proper buffer pool.

The “ALTER TABLESPACE . . .” command will fit into 72 characters, so no ordering columns are needed; there is even enough room to show the existing buffer pool after the “ALTER TABLESPACE” SQL.

Most of the ‘processing’ of this SQL happens in the WHERE clause.

Session: F05

(Not So) Stupid SQL Tricks Or
"My SPUFI writes JCL—Yours
Can Too!"

Mark Doyle
Winterthur U. S. Holdings
mark.doyle@wush.com