DB2 V8 promoted Multi-Row processing as one of the major performance enhancements, you get this for free in SPUFI and DSNTEP4 but how can you change your current COBOL programs to take advantage of this - it is not as simple as it sounds. This presentation shows how to change COBOL programs to use Multi-Row processing and covers problems were encountered and overcome while writing the programs - problems that are very difficult to find the solutions to in the manuals. Sample programs will also be provided so that anyone can try them out to see how the code works.

Important Disclaimer

THE INFORMATION CONTAINED IN THIS PRESENTATION IS PROVIDED FOR INFORMATIONAL PURPOSES ONLY.

WHILE EFFORTS WERE MADE TO VERIFY THE COMPLETENESS AND ACCURACY OF THE INFORMATION CONTAINED IN THIS PRESENTATION, IT IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED.

IN ADDITION, THIS INFORMATION IS BASED ON IBM’S CURRENT PRODUCT PLANS AND STRATEGY, WHICH ARE SUBJECT TO CHANGE BY IBM WITHOUT NOTICE.

IBM SHALL NOT BE RESPONSIBLE FOR ANY DAMAGES ARISING OUT OF THE USE OF, OR OTHERWISE RELATED TO, THIS PRESENTATION OR ANY OTHER DOCUMENTATION.

NOTHING CONTAINED IN THIS PRESENTATION IS INTENDED TO, OR SHALL HAVE THE EFFECT OF:

- CREATING ANY WARRANTY OR REPRESENTATION FROM IBM (OR ITS AFFILIATES OR ITS OR THEIR SUPPLIERS AND/OR LICENSORS); OR
- ALTERING THE TERMS AND CONDITIONS OF THE APPLICABLE LICENSE AGREEMENT GOVERNING THE USE OF IBM SOFTWARE.

Multi-Row Processing

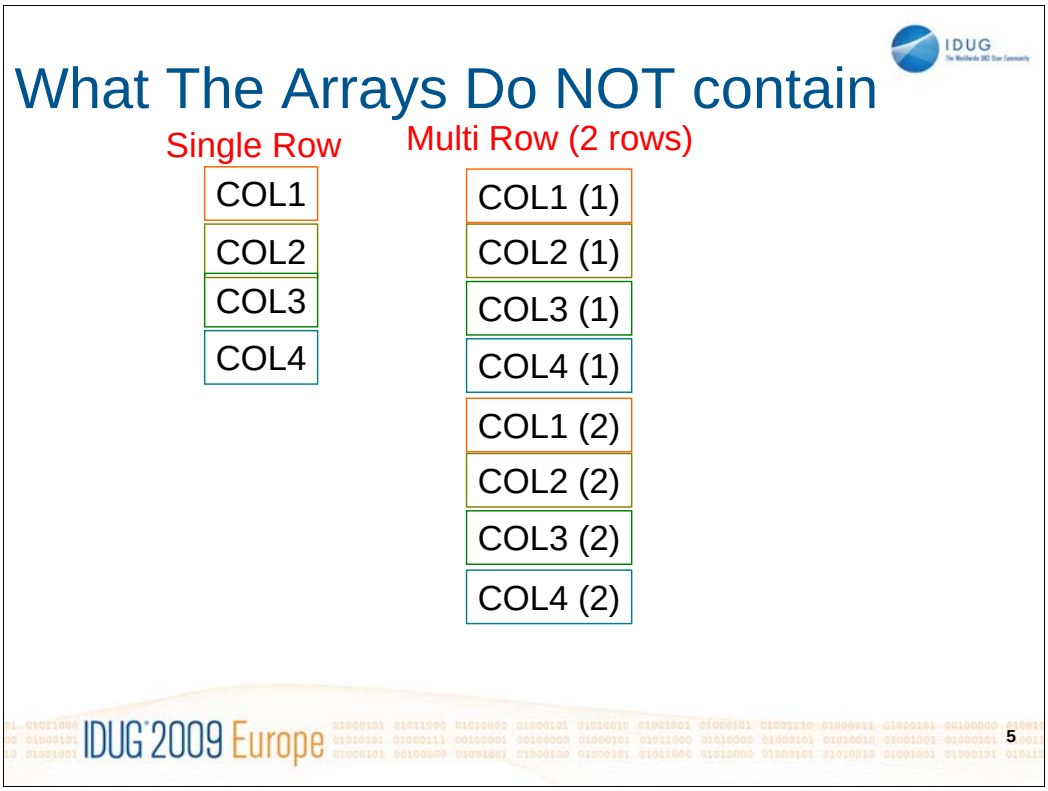
- What Multi-Row processing is
- How to code Multi-Row processing in COBOL
- Discovering and overcoming pitfalls with this process
- Practical examples of the coding techniques
- Sample programs to take away

Multi-row INSERT What it is

- Inserts multiple rows on one API call
- Can be ATOMIC or NOT ATOMIC
- Can be static or dynamic SQL
- Significant performance boost

```
INSERT INTO T1 FOR :hv ROWS
    VALUES( :ARRAY1, :ARRAY2) ATOMIC;
```


This is the information that can be found in most overview presentations for DB2 V8 but it doesn't give much of a clue how to put this into real code.



The previous foil does not describe what is contained in the arrays, some people's first impression was that the arrays are multiple occurrences of a row (which after all is what the name Multi-Row implies), this would make 2 occurrences of a row appear as if it were 2 records from a file.

This shows an example of a 4 column table with what the Working Storage may be for a single row represented on the left. On the right is what I thought that the Working Storage may look like for Multi-Row processing using 2 rows – but I was wrong.

What The Arrays Do contain

Single Row

COL1
COL2
COL3
COL4

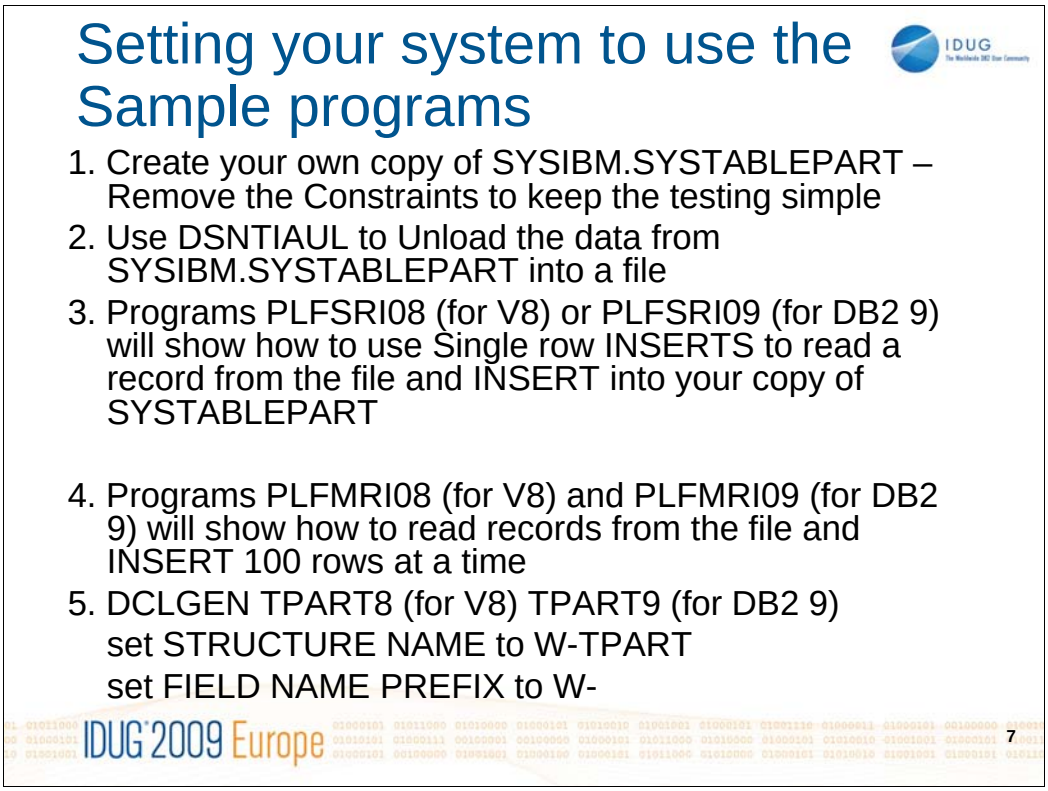
Multi Row (2 rows)

COL1 (1)	COL2 (1)	COL3 (1)	COL4 (1)
COL1 (2)	COL2 (2)	COL3 (2)	COL4 (2)

There is an array for each column NOT for each Row

An array for each column has to be coded so it is not a simple case of changing the 01 level in the DCLGEN to have an OCCURS clause, the OCCURS must be on every field normally each 10 level.

So rather than call it Multi-Row it could be regarded as Multi-Column



Rather than just give the theory of how to put multi-row processing into COBOL code I decided to write some code that anyone could follow. Perhaps you may want to try the sample programs out after seeing this presentation – if you do then here are the steps to follow.

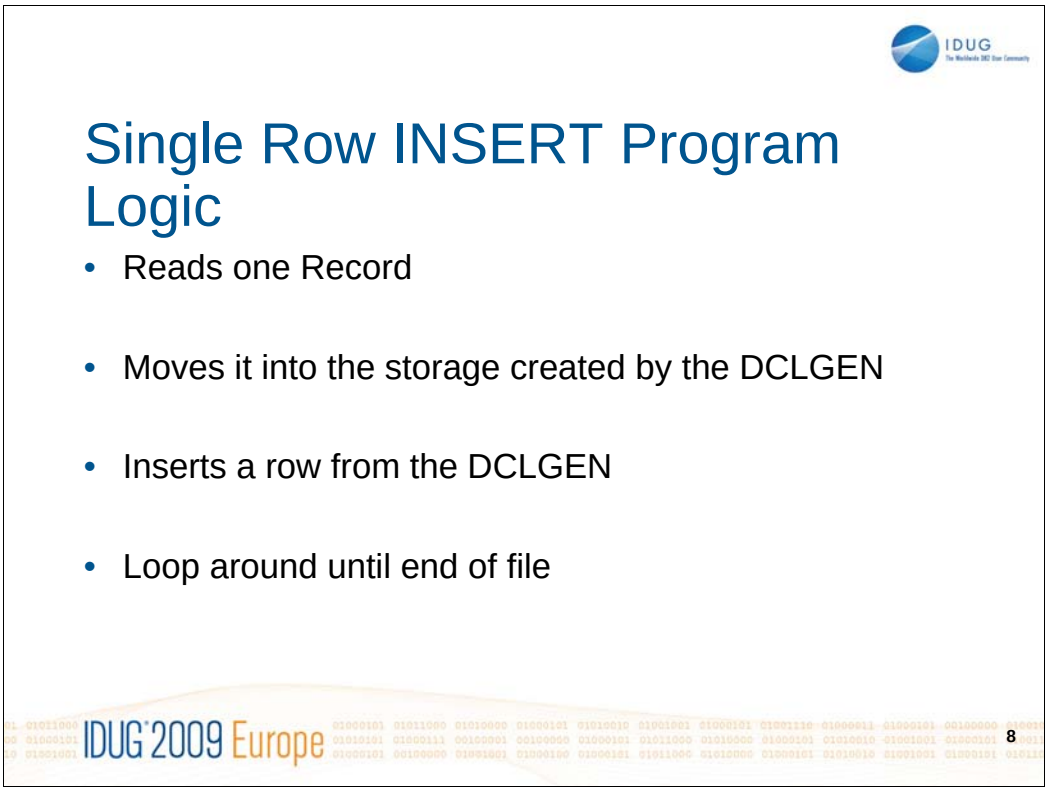
1. Create a table with the same column definitions as SYSIBM.SYSTABLEPART – I chose this table as it contains most of the Column types to try to see if there was any special processing for any of the common column types.

2. Use DSNTIAUL to unload the data from SYSIBM.SYSTABLEPART into a file, you can use any other program you may have to do this as long as it produces the file in the same format as from DSNTIAUL.

3. I wrote 2 sets of programs one set for DB2 V8 and the other set for DB2 9. If you are using a DB2 V8 system then use the PLFSRI08 or use PLFSRI09 for DB2 9. Both of these programs will read one record from the file produced by DSNTIAUL and do single row inserts into your copy of SYSTABLEPART.

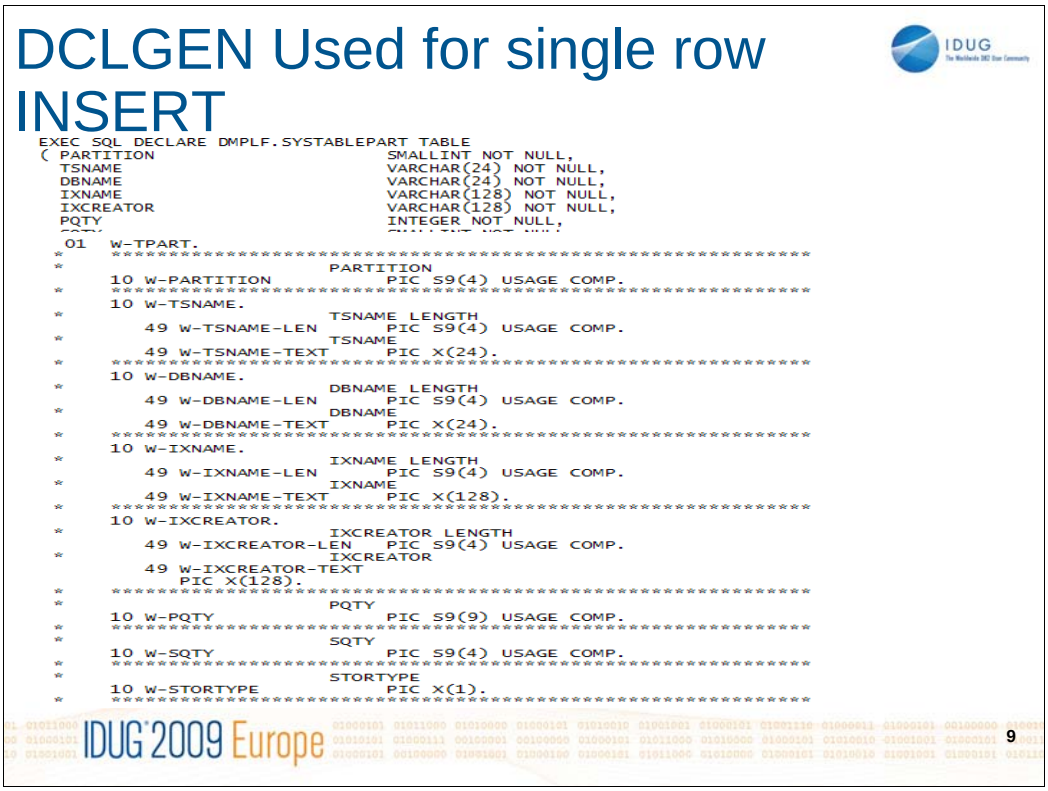
4. Programs PLFMRI08 and PLFMRI09 will show what changes need to be coded in order to use Multi-Row processing – these are covered later in the presentation.

5. Before you can compile any of these programs you will need to produce the DCLGENS from the table definition – create TPART8 if you are using DB2 V8 and TPART9 for DB2 9.



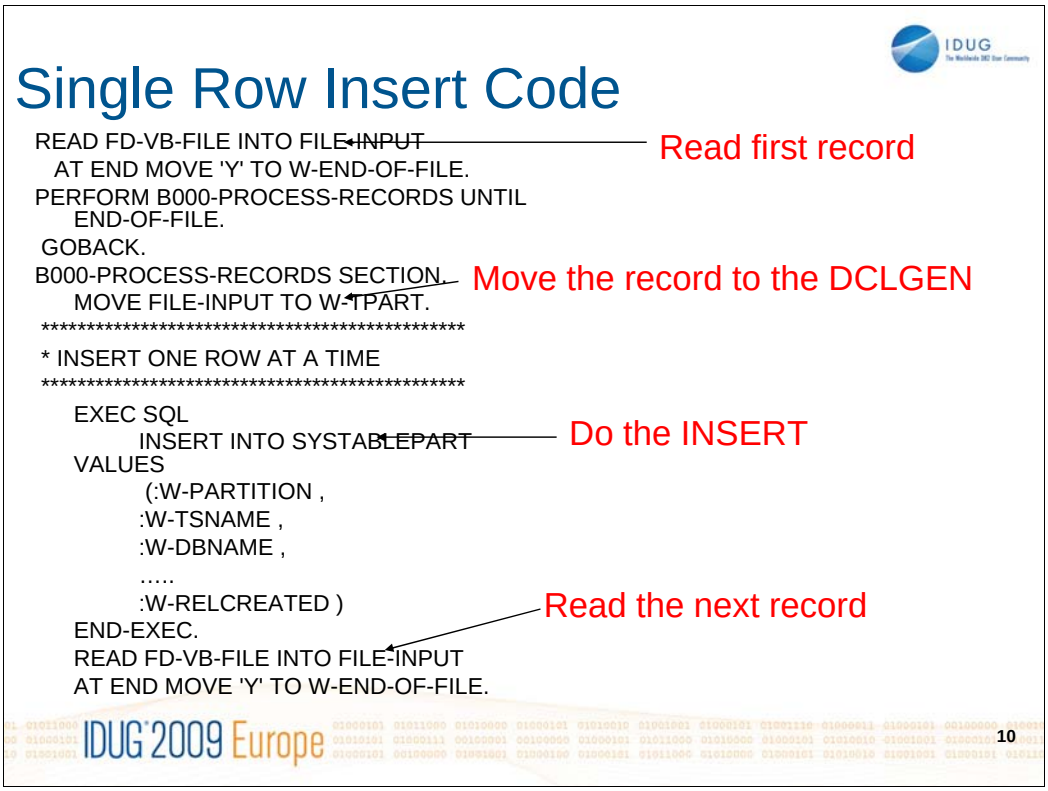
I have kept the logic very simple so we can concentrate on what changes will need to be made in order to convert to use multi-row processing.

DCLGEN Used for single row INSERT



```
EXEC SQL DECLARE DMPLF.SYSTABLEPART TABLE
( PARTITION                       SMALLINT NOT NULL,
  TSNAME                          VARCHAR(24) NOT NULL,
  DBNAME                          VARCHAR(24) NOT NULL,
  IXNAME                          VARCHAR(128) NOT NULL,
  IXCREATOR                       VARCHAR(128) NOT NULL,
  PQTY                            INTEGER NOT NULL,
  SQTY                            SMALLINT NOT NULL,
  STORTYPE                        CHAR(1) NOT NULL
) END-EXEC.
 01  W-TPART.
*    *************************************************************
*                         PARTITION
     10 W-PARTITION             PIC S9(4) USAGE COMP.
*    *************************************************************
     10 W-TSNAME.
*                         TSNAME LENGTH
        49 W-TSNAME-LEN         PIC S9(4) USAGE COMP.
*                         TSNAME
        49 W-TSNAME-TEXT        PIC X(24).
*    *************************************************************
     10 W-DBNAME.
*                         DBNAME LENGTH
        49 W-DBNAME-LEN         PIC S9(4) USAGE COMP.
*                         DBNAME
        49 W-DBNAME-TEXT        PIC X(24).
*    *************************************************************
     10 W-IXNAME.
*                         IXNAME LENGTH
        49 W-IXNAME-LEN         PIC S9(4) USAGE COMP.
*                         IXNAME
        49 W-IXNAME-TEXT        PIC X(128).
*    *************************************************************
     10 W-IXCREATOR.
*                         IXCREATOR LENGTH
        49 W-IXCREATOR-LEN      PIC S9(4) USAGE COMP.
*                         IXCREATOR
        49 W-IXCREATOR-TEXT
           PIC X(128).
*    *************************************************************
*                         PQTY
     10 W-PQTY                  PIC S9(9) USAGE COMP.
*    *************************************************************
*                         SQTY
     10 W-SQTY                  PIC S9(4) USAGE COMP.
*    *************************************************************
*                         STORTYPE
     10 W-STORTYPE              PIC X(1).
*    *************************************************************
```


This is an example of the DCLGEN used for Single row processing, it is just the standard output from a DCLGEN



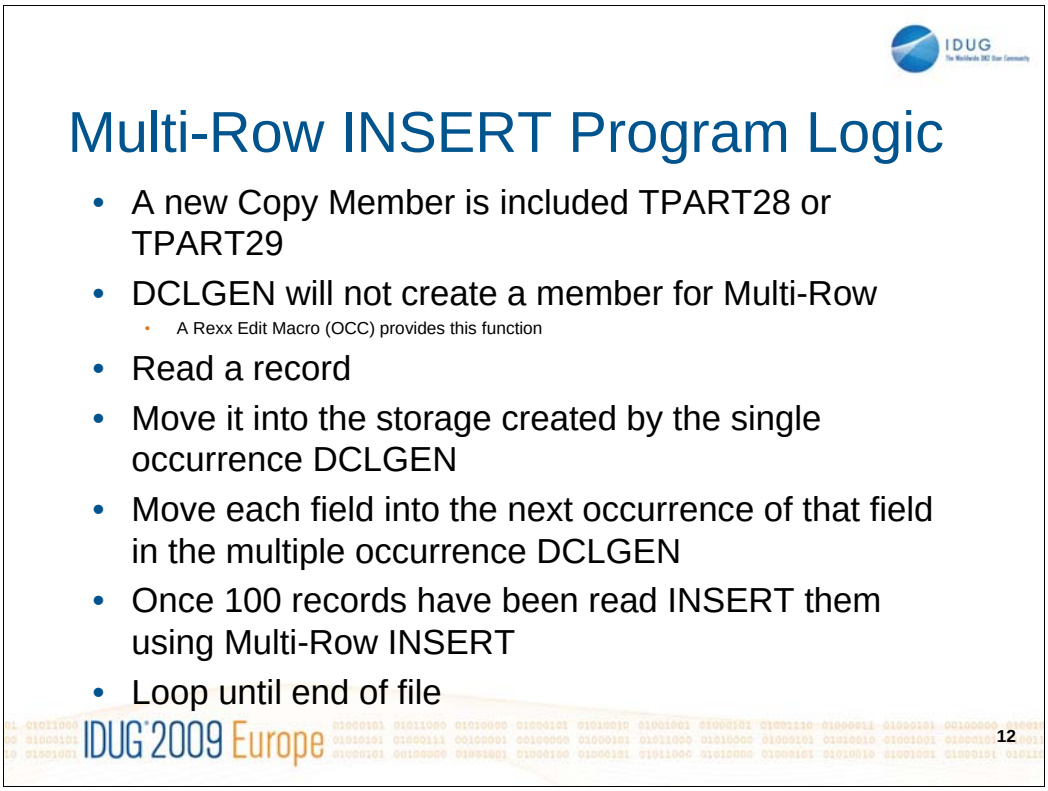
A record is read from the file and moved into the copy record from the DCLGEN, as the file is an extract of SYSIBM.SYSTABLEPART using DSNTIAUL the record correctly maps over the copy record so the Insert just references each field in the DCLGEN.

Single Row INSERT Run JCL

```
//PLFSRI09 EXEC PGM=IKJEFT01,DYNAMNBR=25,ACCT=SHORT,
//           REGION=4096K
//STEPLIB  DD DSN=SYS2.DB2.V910.SDSNLOAD,DISP=SHR
//         DD DSN=SYS2.DB2.V910.SDSNEXIT.PIC,DISP=SHR
//         DD  DSN=FLETCHP.MASTER.LOAD,DISP=SHR
//DDSEQ01R DD  DSN=FLETCHP.TPART.INPUT,DISP=SHR
//SYSUDUMP DD  SYSOUT=*
//SYSTSPRT DD  SYSOUT=*
//SYSOUT   DD  SYSOUT=*
//SYSABOUT DD  SYSOUT=*
//SYSTSIN  DD *
DSN SYSTEM(PB1I)
RUN PROGRAM(PLFSRI09) PLAN(PLFSRI09)
END
/*
```

File created by DSNTIAUL

This is an example of the JCL used to run the Single row Insert program – note that the DD DDSEQ01R is the file that was produced by DSNTIAUL.



I created a copy of the original COBOL Copy record and added an occurs clause for each columns working storage field. I created a Rexx to convert the DCLGEN – the details of this will be shown in another slide.

As we saw at the start of this presentation the Working Storage is not multi-row but multi-column, this makes the processing in the program more complicated. I decided that the simple approach would be to leave the code to read a record from the file and to move it into the original COBOL copy record produced by the DCLGEN, then to move the Working storage field for each column to the next occurrence in the new copy record.

Once 100 records have been moved a Multi-Row INSERT is executed to put the rows into the table.

2 Errors Encountered

On one DB2 9 system Precompiler failed with

HOST VARIABLE ARRAY "W-PARTITION" IS EITHER NOT DEFINED OR IS NOT USABLE

On another DB2 9 and V8 system it precompiled, compiled etc. but the run failed with -311

THE LENGTH OF INPUT HOST VARIABLE NUMBER 17 IS NEGATIVE OR GREATER THAN THE MAXIMUM

But Why?

I had coded the program but when I tried to run it on one DB2 system it wouldn't precompile, I kept getting a message saying that HOST VARIABLE ARRAY "W-PARTITION" IS EITHER NOT DEFINED OR IS NOT USABLE – I checked and double checked this field in the copy produced by DB2 DCLGEN – I had altered the output from the DCLGEN to add the OCCURS clause for each column but had not altered anything else – I was baffled.

I then tried to precompile in a different DB2 system – this time the precompile worked so I ran the program. It started to work but after inserting a number of rows it failed with -311. Once again I was baffled so I decided to search to see if anyone had raised a PMR for the same problem.

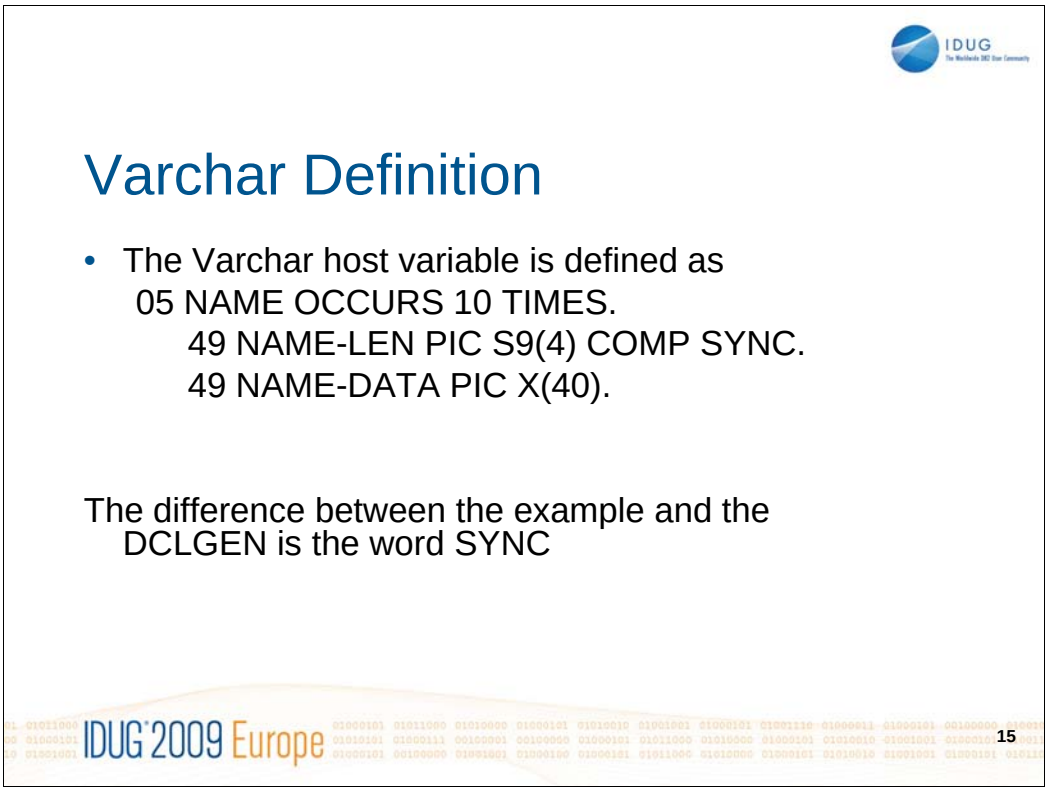
Why where there problems?

- PMR 24142,180,000 Has The Answer
  - It refers to Application Programming and SQL Guide 2.4.3.6 Declaring host variable arrays
  - 2.4.3.6 could not be found but a search on Declaring host variable arrays in the PDF found:-

Example: The following example shows declarations of a fixed-length character array and a varying-length character array:

```
01 OUTPUT-VARS.
    05 NAME OCCURS 10 TIMES.
            49 NAME-LEN PIC S9(4) COMP SYNC.
            49 NAME-DATA PIC X(40).
    05 SERIAL-NUMBER PIC S9(9) COMP-4 OCCURS 10 TIMES.
```


I eventually found a PMR which told me what the problem was – it said that the example in the Programming Manual showed how to code for Varchar columns – above is that example from the manual. See if you can spot the significant keyword that solves the problems.



The example in the manual shows the word SYNC after COMP but the DCLGEN generated the field without the word SYNC.

What does SYNC do

- The COBOL manual says

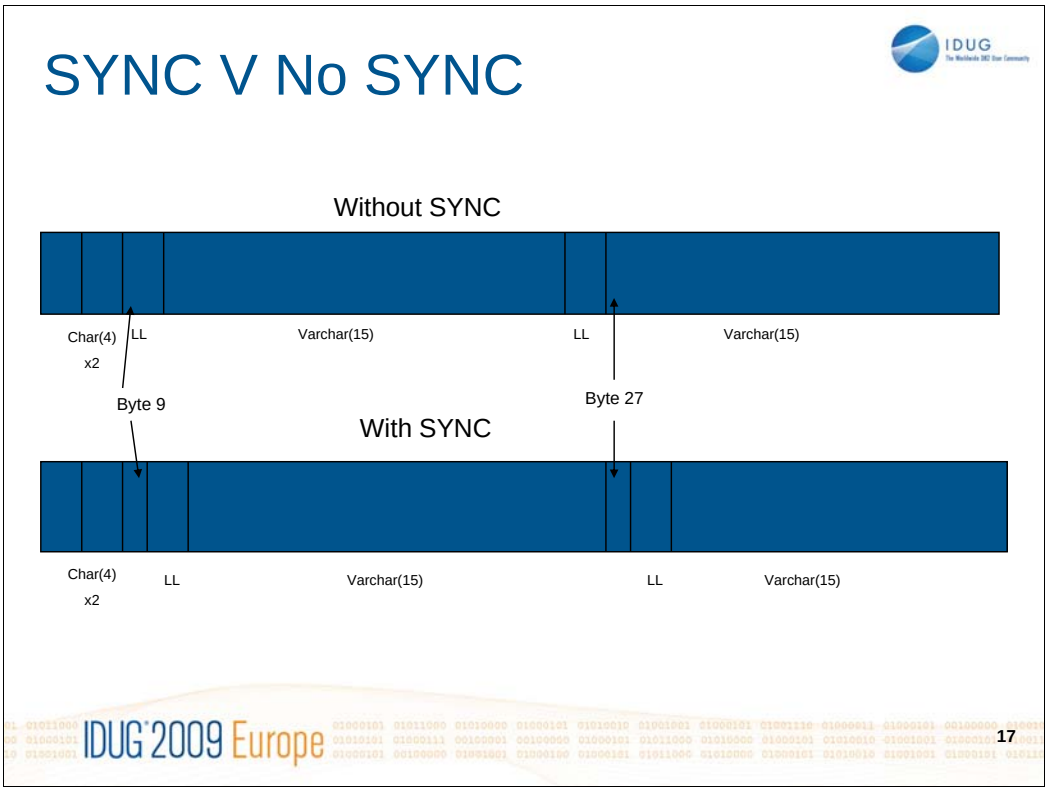
The SYNCHRONIZED clause is never required, but can improve performance on some systems for binary items used in arithmetic.

For S9(4) COMP it aligns on a half word boundary i.e. 2bytes

For S9(9) COMP it aligns on a full word boundary i.e.4 bytes

The COBOL manual says that Sync is never required – in fact it is required for using Multi-Row processing in DB2 V8 and above.

There was a time many years ago where it was standard practice for programs to require parameters to be on half word or full word boundaries but over the years less programs insisted upon this.



This diagram illustrates the difference between using Sync an not using Sync.

In this example we have a table consisting of the following columns.

COL1 CHAR(4)

COL2 VARCHAR(15)

The working storage shows 2 occurrences of both columns to be used with multi-row fetch.

In the example without Sync we can see that the length field of the first occurrence of the varchar column starts at byte 9. Because DB2 is expecting the length field to start on a half word boundary it will look at byte 10 for the start of the length field – this is the second byte of the length field so it is hit and miss if DB2 will receive the correct length.

E.G.

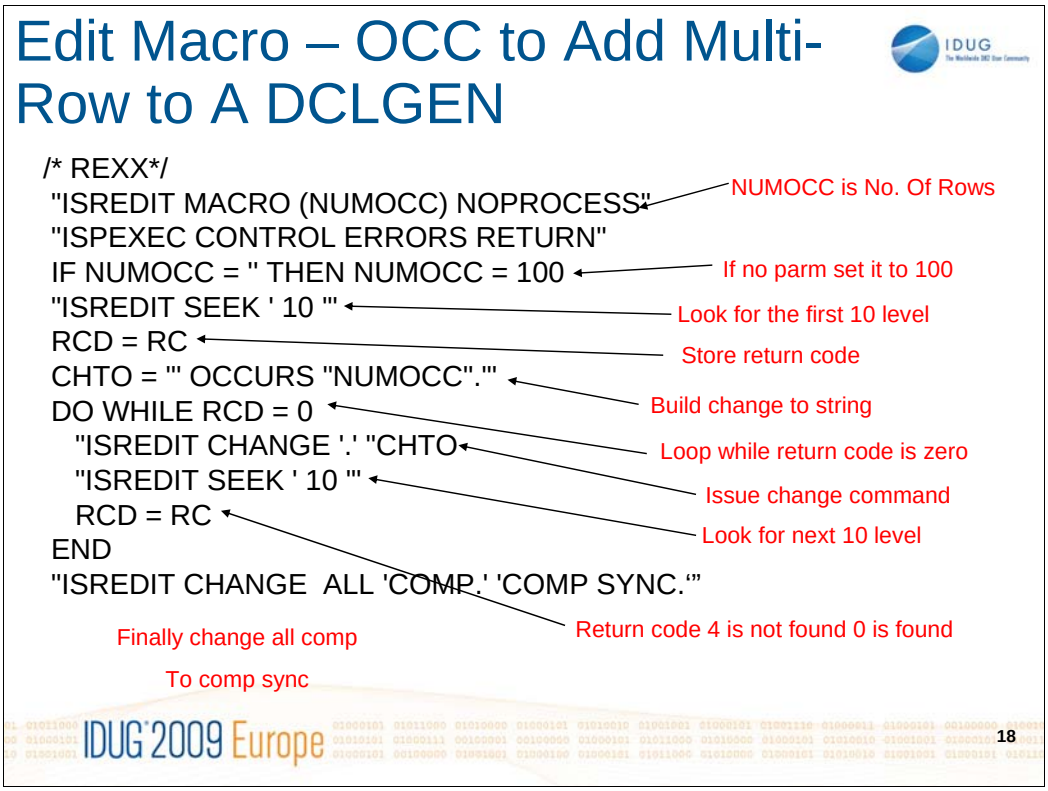
A smallint field can contain positive values from 0 to 32767 – the hex equivalent is ‘0000’ to ‘7FFF’

And negative values from -32768 to -1 – the hex equivalent is ‘8000’ to ‘FFFF’

As DB2 is looking in the second byte of the length field for what it regards as the first byte it is likely that it will get a length field a lot longer than it should be – If the length was meant to be 15 (hex 000F) DB2 will see hex 0F as the first byte and whatever is in the first byte of text in the varchar field – in this case the length will be at least 3840 (hex 0F00) this will be treated as invalid by DB2.

For the second occurrence it will probably see what it regards as a negative number – anything from hex ’80’ to hex ‘FF’ – decimal 128 to 255 as it thinks that the first 2 characters in the text field are in fact the length.

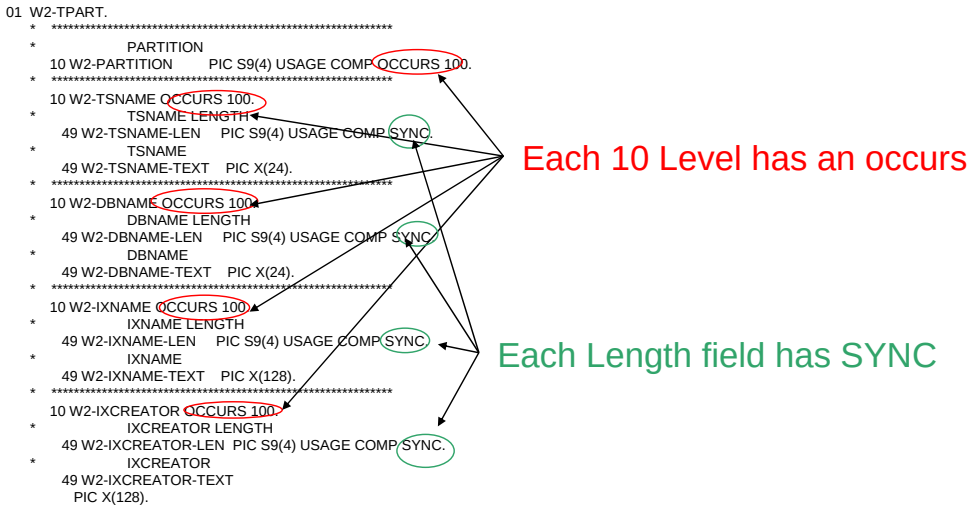
Edit Macro – OCC to Add Multi-Row to A DCLGEN



This is the complete Rexx which will add occurrences to each 10 level then add SYNC to any length fields.

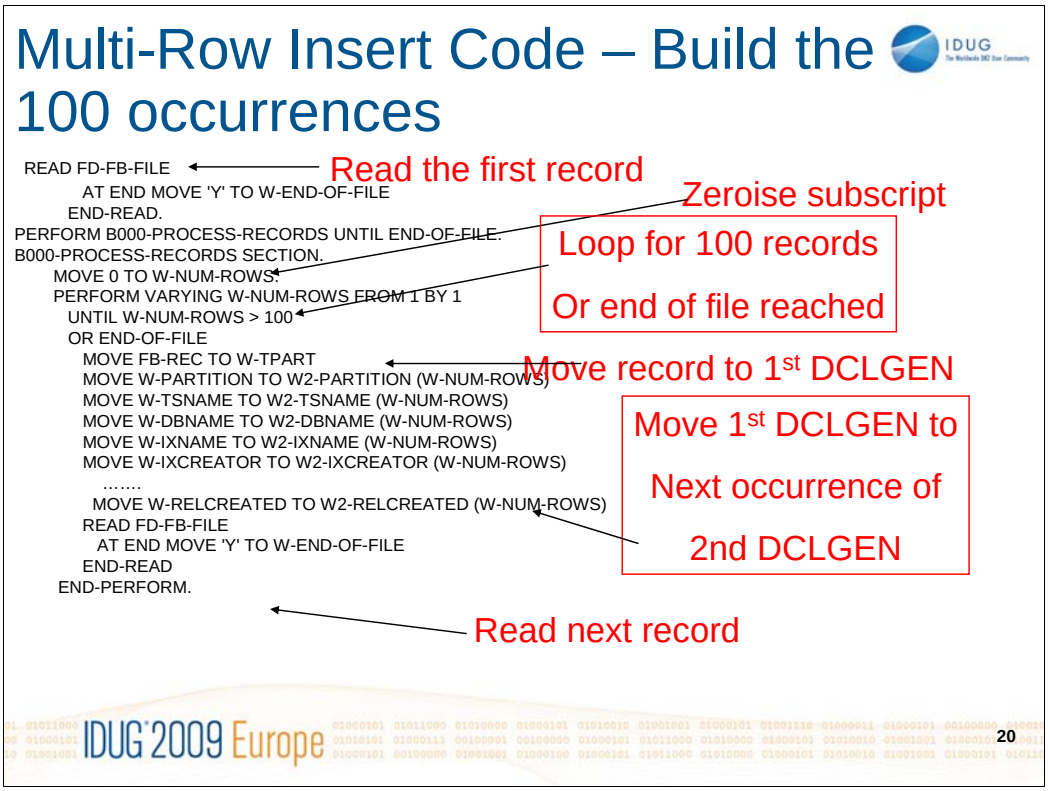
At first sight you may think that the final line will change all COMP fields to COMP SYNC but it won't as all of the other COMP fields will have been level 10 fields so they will have been changed from COMP to COMP OCCURS 100 so the only fields that still contain 'COMP.' are the ones which are not level 10.

New DCLGEN for Multi-Row after running OCC



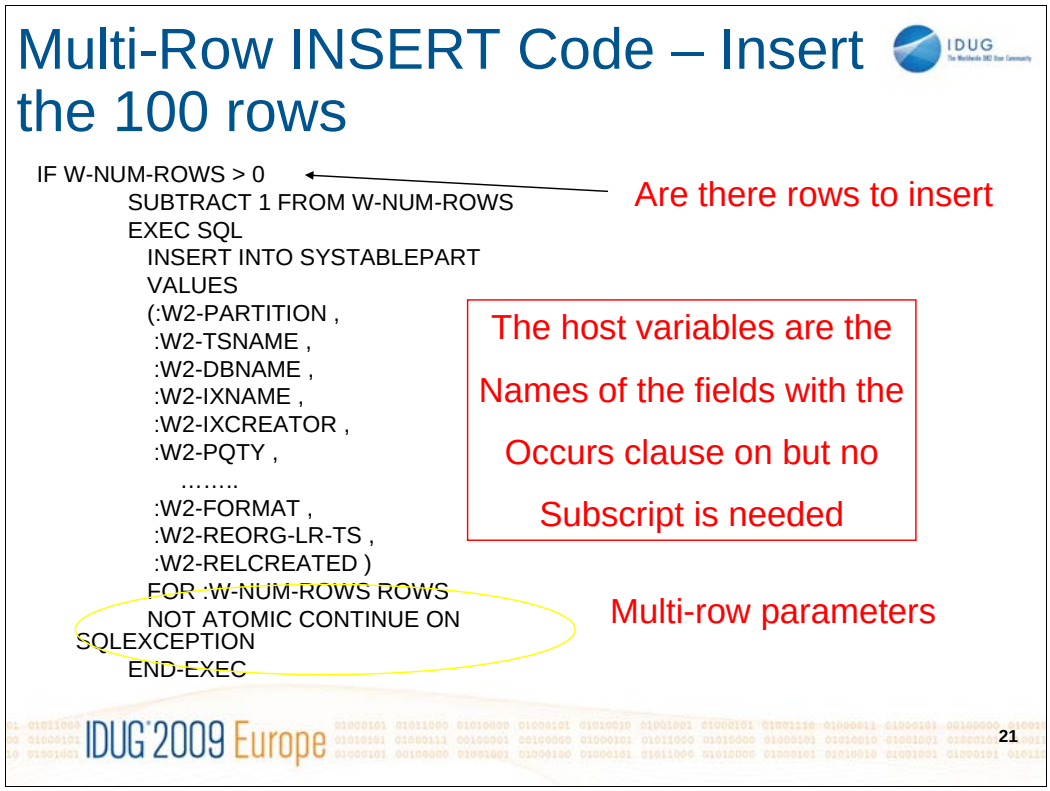
Each field has now become an array with 100 occurrences and each length field in each varchar has had the SYNC clause added. Note there is no DECLARE statement for the table, there can only be 1 per table in the program and that was defined in the first DCLGEN so you must remove it from the second DCLGEN or the precompiler will fail.

Multi-Row Insert Code – Build the 100 occurrences



If the array had been at Row level then the code would simply have been to move the record from the file into the next occurrence of the DCLGEN but because the array is column based every field has to be moved individually. Quite a time consuming change.

Multi-Row INSERT Code – Insert the 100 rows



Note the new keywords, FOR xx ROWS and NOT ATOMIC CONTINUE ON SQLEXCEPTION, the first is required for multi-row fetch, the second is optional.

These new keywords are covered in more detail on the next few foils.

Alternative to Using FOR :W-NUM-ROWS

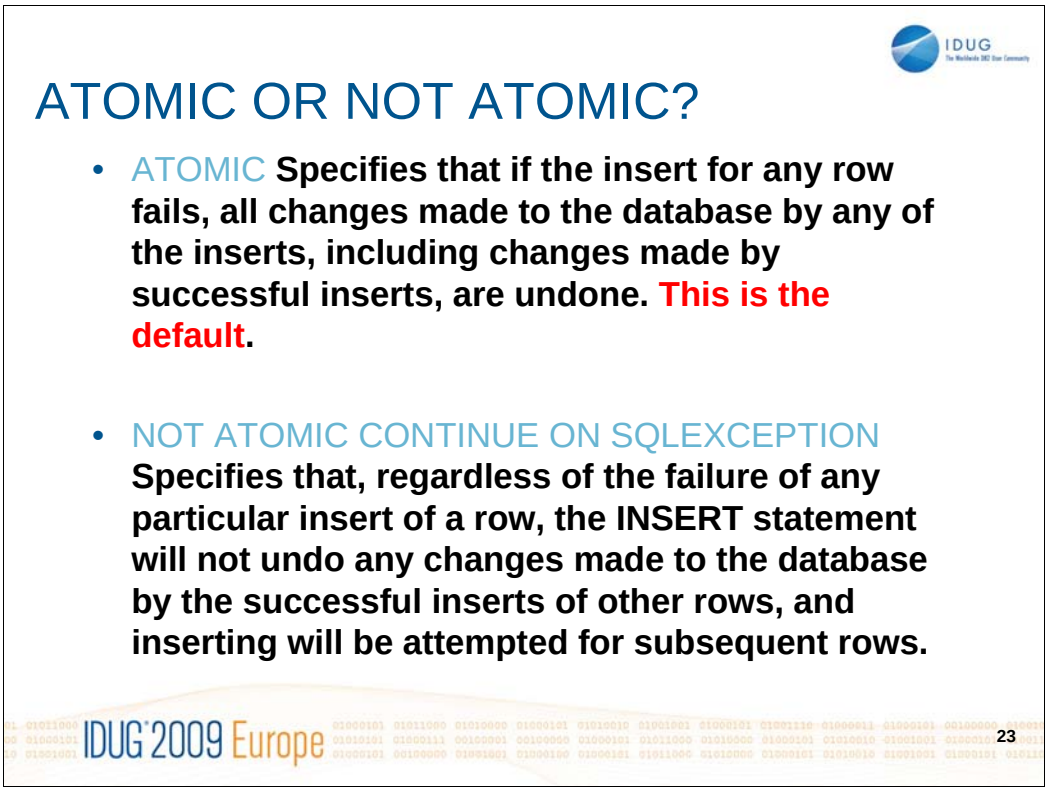
- FOR 100 ROWS could have been coded
  - What if the file contained 226 records
    - First INSERT would insert 100 rows
    - Second INSERT would insert 100 rows
    - Third INSERT would insert 100 rows
    - So we now have 300 rows but only 226 records

  The last 74 would be the same as the last 74 of the second insert but using FOR :W-NUM-ROWS ROWS tells DB2 that the last Insert has 26 so no unexpected duplicates

ON the previous foil it shows that the SQL contains a host variable called W-NUM-ROWS which contains the number of rows that you want to be inserted using multi-row INSERT – this is by far the best way of coding but I decided to see what would happen if I hard coded the number of rows.

I change the code to say FOR 100 ROWS.

There were 226 records in the file – the first insert put 100 rows into the table as did the second one but the third one also inserted 100 rows when the intention was only to insert 26 – this shows that DB2 will do what you tell it to do not what you think it should do.



A decision must be made to decide which of these will best suit the program you are writing.

Handling Errors in Single Row Insert Program

- Add the following in B000 to cause an error

```
IF W-ROW-COUNT = 10
   MOVE -10 TO W-DBNAME-LEN
END-IF.
```

- This will stop the 11th record from being inserted

Handling errors is a very important part of any program, the next few foils will show the traditional way of handling errors within DB2 and look at the new method which is particularly relevant for multi-row processing.

I put the above code in the program to force an error.

It is invalid to have a negative length field for a varchar so this will cause an error

Using DSNTIAR to Obtain the Message

Working Storage

```
•
01  W150-ERROR-MESSAGE.
      03  W150-ERR-LEN         PIC S9(4) COMP VALUE +288.
      03  W150-ERR-MSG-1       PIC X(72).
      03  W150-ERR-MSG-2       PIC X(72).
      03  W150-ERR-MSG-3       PIC X(72).
      03  W150-ERR-MSG-4       PIC X(72).
01  W150-ERROR-MESSAGE-LEN    PIC S9(9) COMP VALUE +72.
```

Total Length of following 4 fields

Length of each message line

Procedure Division

```
CALL 'DSNTIAR' USING SQLCA
                W150-ERROR-MESSAGE
                W150-ERROR-MESSAGE-LEN.

DISPLAY W150-ERR-MSG-1.
DISPLAY W150-ERR-MSG-2.
DISPLAY W150-ERR-MSG-3.
DISPLAY W150-ERR-MSG-4.
MOVE 12 TO RETURN-CODE.
GOBACK.
```


This code shows that the SQLCA is passed to DSNTIAR to obtain the full error message, the code then ends the program with a code of 12, many installations will call some routine to cause an abend rather than setting a return code.

Output From Job after Error

DSNT408I SQLCODE = -311, ERROR: THE LENGTH OF INPUT HOST VARIABLE
NUMBER 3 IS NEGATIVE OR GREATER THAN THE MAXIMUM
DSNT418I SQLSTATE = 22501 SQLSTATE RETURN CODE
DSNT415I SQLERRP = DSNXRIHB SQL PROCEDURE DETECTING ERROR

This has been the recommended method

Of handling errors in programs using DB2

This is the traditional method for error reporting in DB2 programs – it tells us exactly what the problem is

Put Same Error Into Multi-Row Insert

- Add the following in B000 to cause an error

  MOVE -10 TO W2-DBNAME-LEN (11)

- This will stop the 11th record from being inserted

Now the code to cause an error is put into the multi-row insert program to stop the 11th row from being inserted.

Output After Error in Multi-Row Insert Program

DSNT408I SQLCODE = -253, ERROR: A NON-ATOMIC INSERT STATEMENT

SUCCESSFULLY COMPLETED FOR SOME OF THE REQUESTED ROWS,

POSSIBLY WITH WARNINGS, AND ONE OR MORE ERRORS

DSNT418I SQLSTATE = 22529 SQLSTATE RETURN CODE

This is not very helpful as it only tells us that some worked

SQLERRD (3) can be used to find the number inserted

Using the traditional method of error reporting in a program using multi-row processing can be problematic – as we can see here the real error has not been reported – instead all we have been told is that some of the rows have been inserted and one or more have failed.

This was using NON ATOMIC which tells DB2 to try to continue after hitting an error so the next test was to see what would happen if ATOMIC was used.

What Happens if the INSERT is Atomic

DSNT408I SQLCODE = -311, ERROR: THE LENGTH OF INPUT HOST VARIABLE
NUMBER 3 IS NEGATIVE OR GREATER THAN THE MAXIMUM
DSNT418I SQLSTATE = 22501 SQLSTATE RETURN CODE
DSNT415I SQLERRP = DSNXRIHB SQL PROCEDURE DETECTING ERROR

This message now tells what the real problem is but the 10 Inserted rows have been rolled back.

How can correct messages be issued and allow inserts to continue.

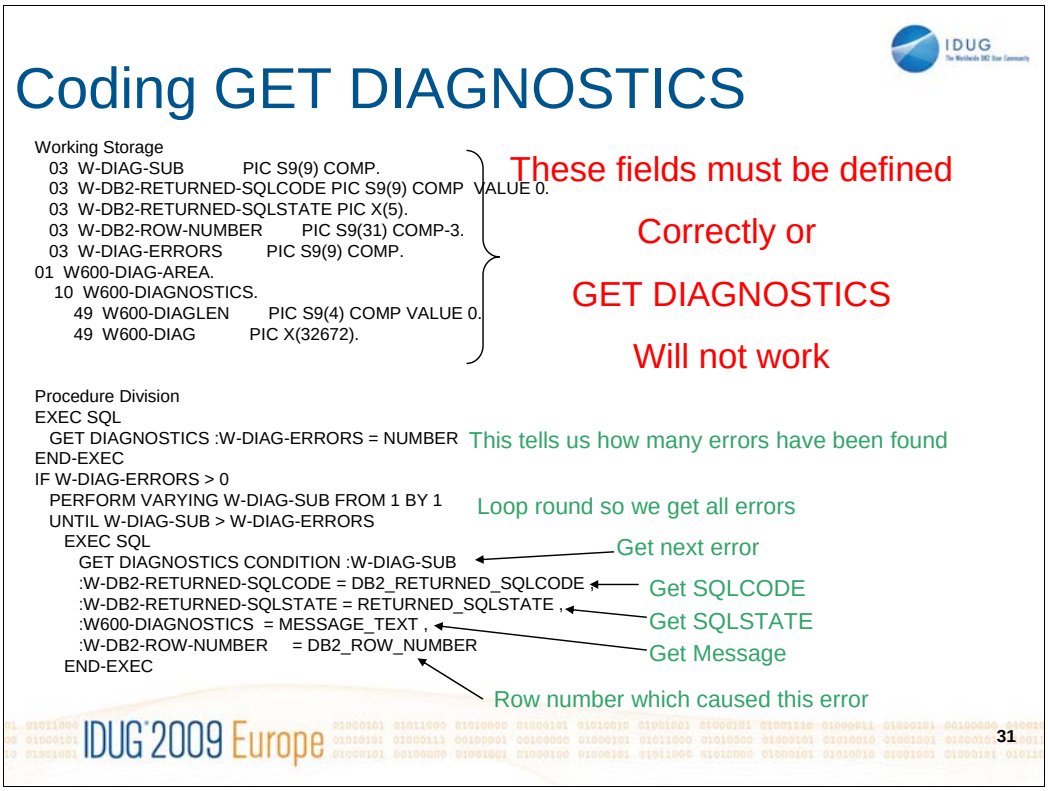
Now the code has been changed to ATOMIC so as soon as the first error is hit the successful inserts will be rolled back.

Now we can see the real error just as we did in the single row program but how can we use NON ATOMIC so any successful inserts are not rolled back but we can still report the real errors?

Issuing correct messages and allowing inserts

- NOT ATOMIC CONTINUE ON SQLEXCEPTION will allow processing to continue
- But the correct error can not be seen by calling DSNTIAR
- The solution is to use GET DIAGNOSTICS

To be able to allow the program to continue and to be able to report correctly on errors a new method of error reporting has to be used – this new method is GET DIAGNOSTICS



This shows how to code GET DIAGNOSTICS, if the Working Storage fields are not coded correctly then GET DIAGNOSTICS will not work.

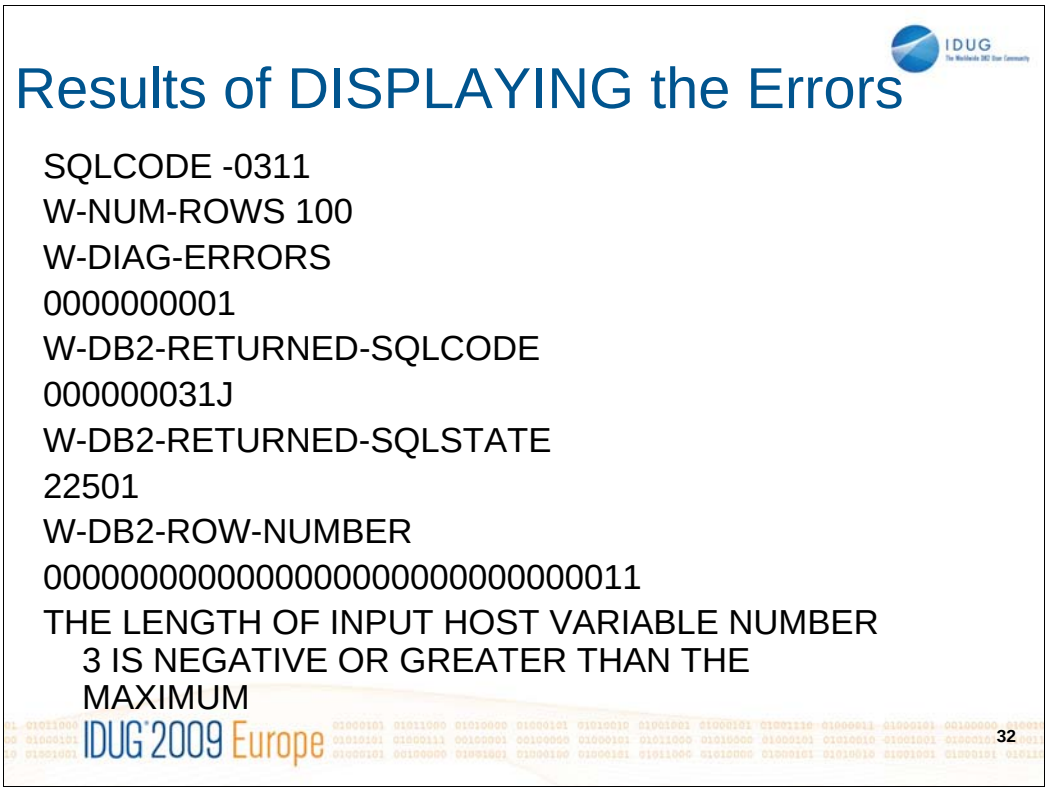
The first thing we need to do is to ask GET DIAGNOSTICS how many errors have been found, this is done using GET DIAGNOSTICS :W-DIAG-ERRORS = NUMBER – the number of errors found is placed in the host variable called W-DIAG-ERRORS.

Once we know how many errors there are we can loop round asking for each one in turn.

GET DIAGNOSTICS CONDITION :W-DIAG-SUB will ask for the next set of diagnostics – please note that to obtain the SQLCODE you must use DB2_RETURNED_SQLCODE but to get the SQLSTATE you must use RETURNED_SQLSTATE not DB2_RETURNED_SQLSTATE

A full list of these is contained in the DB2 Manuals.

The code would get each error message and then display each error message. The DISPLAYS are not shown here but can be found in the code.



This is the output from the displays issued after the GET DIAGNOSTICS, we see that there is only one error.

The SQLCODE is showing as 31J – this shows a quirk of COBOL that most programmers hit at some time.

The SQLCODE was stored in a field defined as PIC S9(9) COMP – when COBOL displays this field it tries to convert the contents to displayable format – the last character is a letter to show if the field is positive or negative if the first hex character of this starts with a C it is positive or if it starts with a D it is negative – in this case J has a hex value of D1 (in EBCDIC) so the real value for SQLCODE is -311.

You can get round this problem by simply moving the SQLCODE field to one defined as something like –ZZZ999 so that it will start with – if it is negative, the leading zeroes will be removed if the field contains a number greater than 999.

Multi-row FETCH

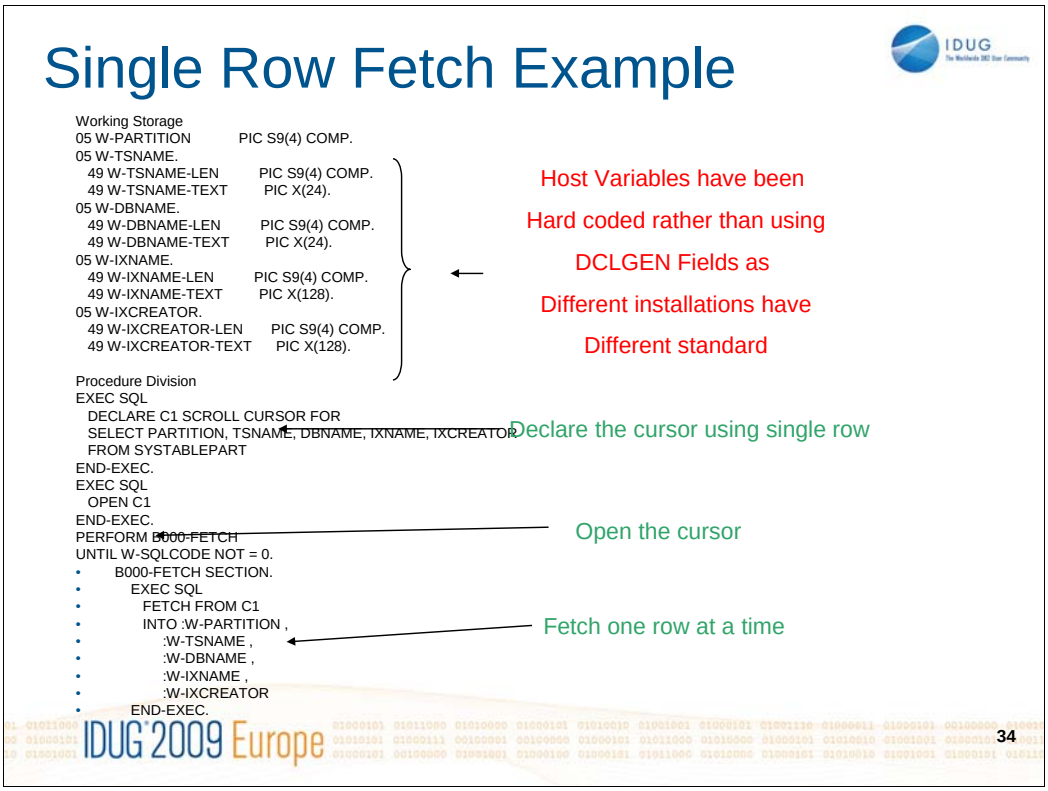
- Returns multiple rows on one API crossing
- "wide" cursor with locks on multiple rows
- Supports scrollable and non-scrollable, static and dynamic SQL
- Significant performance boost
- DSNTEP4 = DSNTEP2 + MRF

```
DECLARE C1 CURSOR
    WITH ROWSET POSITIONING
    FOR SELECT COL1, COL2 FROM T1;
OPEN C1;
FETCH FROM C1
    FOR :hv ROWS  INTO  :ARRAY1, :ARRAY2;
```


So we have seen how to change a COBOL program to provide multi-row processing for INSERTS now we will turn our attention to multi-row fetches and what you need to do to a program reading a single row at a time. As for multi-row inserts most people will have only seen this foil in various DB2 V8 presentations and of course it does not really tell you what you need to code in a COBOL program.

Multiple row FETCH also helps with application portability and performance. It provides a new concept called a "wide" cursor, which contains multiple rows rather than just one. A rowset is a set of rows that is retrieved through a multiple-row fetch.

As for multi-row inserts



This is part of the program to read each row in the SYSTABLEPART table that was populated using the previous program.

In this case I decided that I would only SELECT 5 columns from the table rather than every column – the aim of that is to show a more true to life example as in my experience most programs do not need to access every column in a table.

There is no rule saying that columns must be read into a COBOL copy statement so there are probably plenty of programs that use their own code rather than output from a DCLGEN – this shows that you can code your own fields.

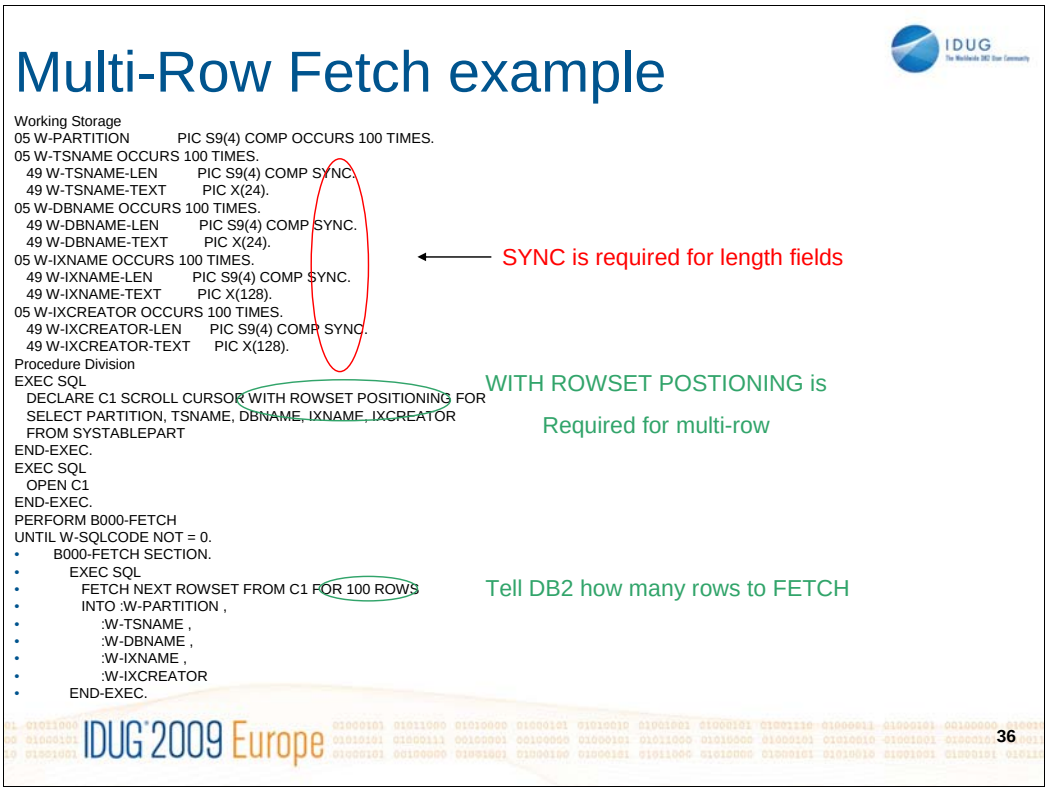
Although you can code your own fields I have seen occasions when a programmer has got one of the fields wrong – this caused DB2 to have to translate the field and in doing so it had to switch off index access and do a tablespace scan instead.

JCL to run PLFSRI01

```
//PLFSRI01 EXEC PGM=IKJEFT01,DYNAMNBR=25,ACCT=SHORT,
//           REGION=4096K
//STEPLIB  DD DSN=SYS2.DB2.V910.SDSNLOAD,DISP=SHR
//         DD DSN=SYS2.DB2.V910.SDSNEXIT.PIC,DISP=SHR
//         DD  DSN=FLETCHP.MASTER.LOAD,DISP=SHR
//SYSUDUMP DD  SYSOUT=*
//SYSTSPRT DD  SYSOUT=*
//SYSOUT   DD  SYSOUT=*
//SYSABOUT DD  SYSOUT=*
//SYSTSIN  DD *
DSN SYSTEM(PB1I)
RUN PROGRAM(PLFSRF01) PLAN(PLFSRF01)
END
/*
```


This is an example of the JCL to run the single row fetch program.

Multi-Row Fetch example



To fetch multiple rows at a time there are only a few things that need to be changed:-

1. Add Sync to the length fields for the Varchars.
2. Add WITH ROWSET POSITIONING to the DECLARE CURSOR
3. Change the FETCH to say how many rows to fetch at a time.

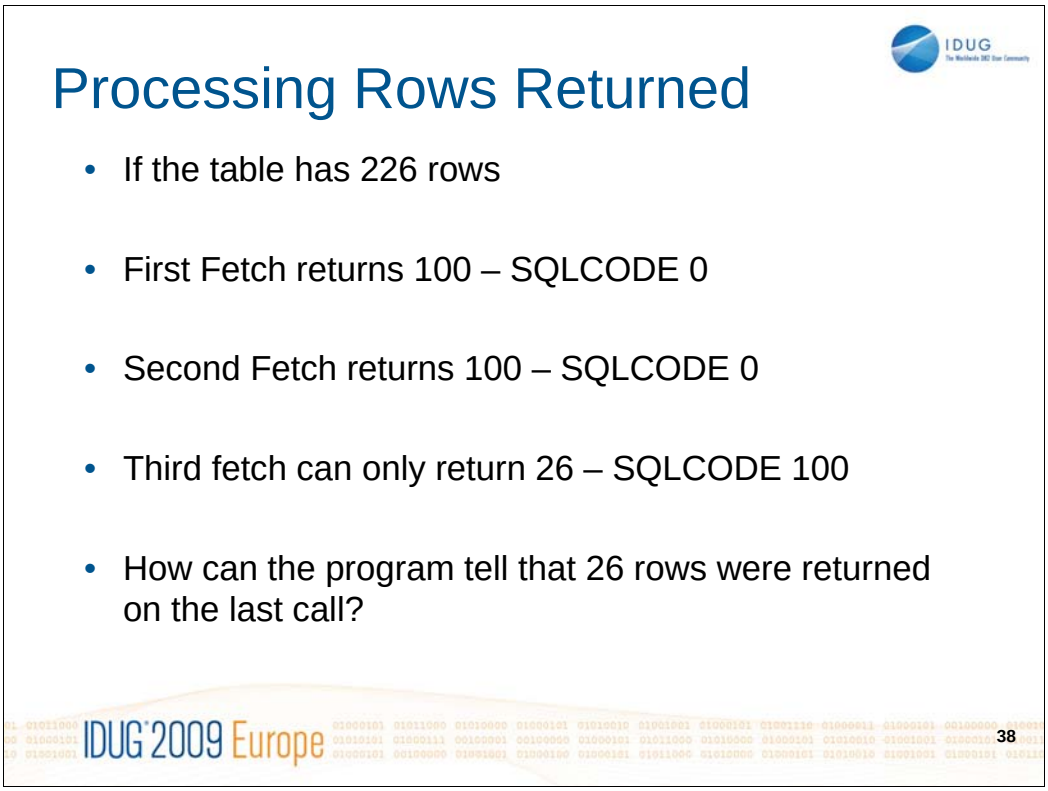
As we saw for Inserts SYNC is required for all varchar length fields but it is not required for smallint fields, the program still works if SYNC is coded for the field W-PARTITION.

Processing the rows returned by Multi-Row fetch

```
 PERFORM VARYING W-SUB FROM 1 BY 1
UNTIL W-SUB > 100
  DISPLAY W-PARTITION (W-SUB) ' '
  W-TSNAME-TEXT (W-SUB) (1:W-TSNAME-LEN (W-SUB)) ' '
  W-DBNAME-TEXT (W-SUB) (1:W-DBNAME-LEN (W-SUB))
   '',
  W-IXNAME-TEXT (W-SUB) (1:W-IXNAME-LEN (W-SUB)) ' '
  W-IXCREATOR-TEXT (W-SUB) (1:W-IXCREATOR-LEN (W-
   SUB))
END-PERFORM
```

Loop round in this case Displaying 100 rows returned

The only other things we need to change in the code is to add a subscript to each field that the SQL put the columns into and to enclose this in a loop to process each of the rows returned by the multi-row fetch.



We saw in the multi-row insert processing that we had a problem when we hard coded the number of rows in the INSERT – we said we wanted 100 rows INSERTED. I decided to see what would happen if I did the same with a multi-row fetch and found that the third fetch gave me an SQLCODE of +100 – which of course means that all rows have been fetched from the table.

In a single row fetch program a loop is normally coded to fetch each row until the SQLCODE is +100, if we use the same approach in our multi-row fetch program then some rows may not get processed so we need to change the code to cater for that. In order to change the code we need to identify how many (if any) rows have been returned on our last request to fetch 100 rows.

How to tell How many rows have been returned

- SQLERRD (3) in SQLCA

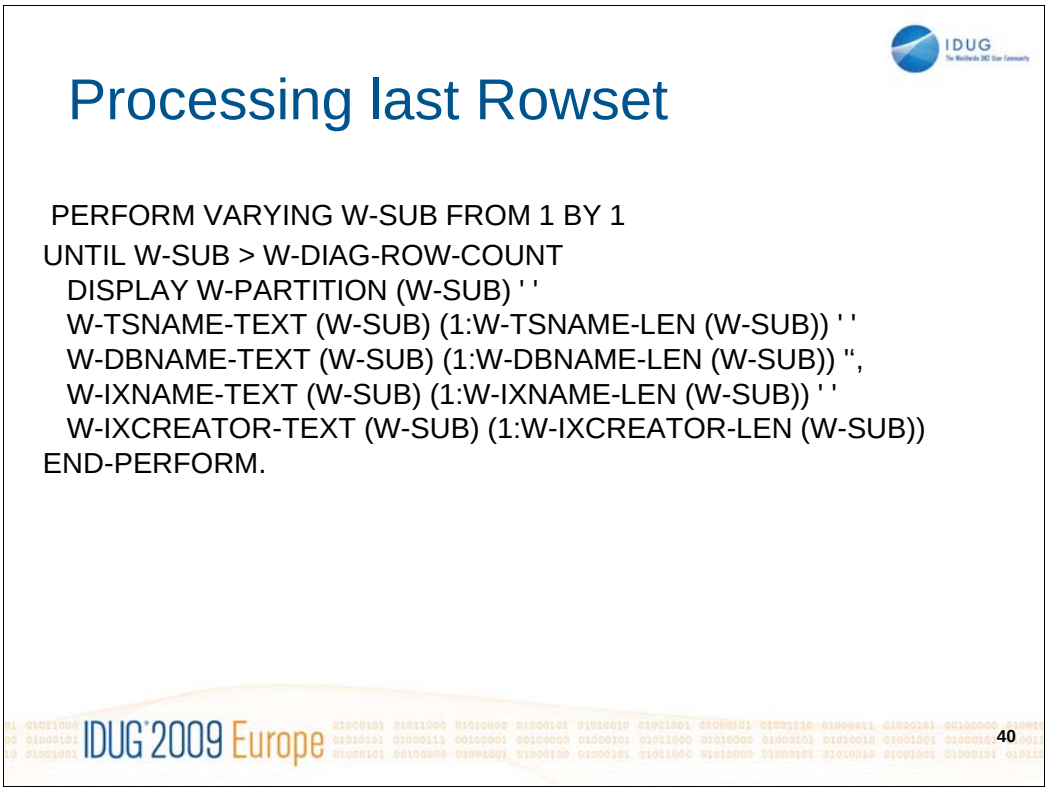
- Or Use Get DIAGNOSTICS

```
EXEC SQL
  GET DIAGNOSTICS
  :W-DIAG-ROW-COUNT = ROW_COUNT

END-EXEC
```


In order to find out how many rows have been returned we can either look at the third occurrence of SQLERRD in the SQLCA or we can get the ROW_COUNT from GET DIAGNOSTICS.

I would recommend making GET DIAGNOSTICS your standard approach as it can be used for FETCHes but it is almost essential for multi-row INSERTS .

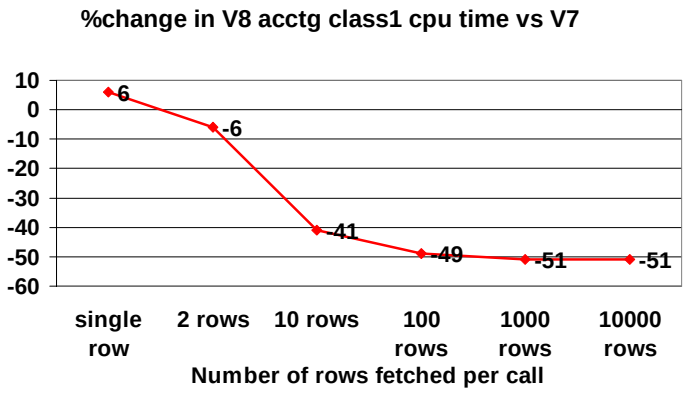


I came up with 2 approaches for handling the rows from the last FETCH they both have their merits but which one you choose will depend upon how complicated the current code is.

The first method (which is the one on this foil) is to leave the original loop in the program so it process rows and does the next fetch until It gets an SQLCODE of +100 then process the last rowset in separate code.

The second approach is to change the loop that was added to process the rows returned so it loops for the number of rows returned rather than the hard coded number (in this case 100).

20column 100000row Fetch CPU Time



We have seen that it is more complicated to change a single INSERT to a multi-row INSERT than it is to change a single FETCH to a multi-row FETCH so there has to be good reasons to spend the time and effort to change the code.

I would say that the figures in this foil can be used to justify the change but it is also probable that most of the SQL that is run in the world is a FETCH rather than an INSERT so your changes need not be too complicated.

Session 17701

Multi-Row Processing Coding it in COBOL

Paul Fletcher
IBM
fletchpl@uk.ibm.com