



# 144

# DB2

*October 2004*

---

## **In this issue**

- [3 Materialized query tables and how to use them](#)
  - [6 Modify column attributes – part 2](#)
  - [18 Business rules as code objects](#)
  - [30 Using dynamic SQL for maximum flexibility](#)
  - [45 November 2001 – October 2004 index](#)
  - [47 DB2 news](#)
- 

© Xephon Inc 2004

# update

# ***DB2 Update***

---

## **Published by**

Xephon Inc  
PO Box 550547  
Dallas, Texas 75355  
USA

Phone: 214-340-5690

Fax: 214-341-7081

## **Editor**

Trevor Eddolls  
E-mail: [trevore@xephon.com](mailto:trevore@xephon.com)

## **Publisher**

Nicole Thomas  
E-mail: [nicole@xephon.com](mailto:nicole@xephon.com)

## **Subscriptions and back-issues**

A year's subscription to *DB2 Update*, comprising twelve monthly issues, costs \$380.00 in the USA and Canada; £255.00 in the UK; £261.00 in Europe; £267.00 in Australasia and Japan; and £265.50 elsewhere. In all cases the price includes postage. Individual issues, starting with the January 2000 issue, are available separately to subscribers for \$33.75 (£22.50) each including postage.

## ***DB2 Update on-line***

Code from *DB2 Update*, and complete issues in Acrobat PDF format, can be downloaded from our Web site at <http://www.xephon.com/db2>; you will need to supply a word from the printed issue.

## **Disclaimer**

Readers are cautioned that, although the information in this journal is presented in good faith, neither Xephon nor the organizations or individuals that supplied information in this journal give any warranty or make any representations as to the accuracy of the material it contains. Neither Xephon nor the contributing organizations or individuals accept any liability of any kind howsoever arising out of the use of such material. Readers should satisfy themselves as to the correctness and relevance to their circumstances of all advice, information, code, JCL, and other contents of this journal before making any use of it.

## **Contributions**

When Xephon is given copyright, articles published in *DB2 Update* are paid for at the rate of \$160 (£100 outside North America) per 1000 words and \$80 (£50) per 100 lines of code for the first 200 lines of original material. The remaining code is paid for at the rate of \$32 (£20) per 100 lines. To find out more about contributing an article, without any obligation, please download a copy of our *Notes for Contributors* from [www.xephon.com/nfc](http://www.xephon.com/nfc).

---

© Xephon Inc 2004. All rights reserved. None of the text in this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior permission of the copyright owner. Subscribers are free to copy any code reproduced in this publication for use in their own installations, but may not sell such code or incorporate it in any commercial product. No part of this publication may be used for any form of advertising, sales promotion, or publicity without the written permission of the publisher. Copying permits are available from Xephon in the form of pressure-sensitive labels, for application to individual copies. A pack of 240 labels costs \$36 (£24), giving a cost per copy of 15 cents (10 pence). To order, contact Xephon at any of the addresses above.

*Printed in England.*

## Materialized query tables and how to use them

This article discusses what Materialized Query Tables (MQTs) are, and how you can use them. They were introduced in DB2 UDB V8.1. In V7 we had the concept of summary tables, which were created using a GROUP BY statement (they were actually introduced in V6).

I won't discuss summary tables in great detail, but I will show how to create a summary table so that we can compare that process with how to create MQTs. Say we wanted to create a REFRESH DEFERRED summary table (ie user maintained) called sum14, based on the EMPLOYEE table in the SAMPLE database, and grouped on the workdept column. We would issue:

```
create summary table sum14
as
(select distinct(workdept) as dept, sum(bonus) as bon
from employee
group by workdept)
data initially deferred
refresh deferred
```

Note: I ran all the SQL in this article on a Windows 2000 machine running DB2 UDB 8.1 FP2 using the DB2ADMIN userid and the SAMPLE database.

Because we created the summary table with refresh deferred specified, we need to populate it using the command below:

```
>db2 refresh table sum14
```

We can now select from the table:

```
>db2 select * from sum14
```

```
DEPT BON
---- -
A00          2500.00
B01           800.00
C01          1900.00
D11          4400.00
D21          2900.00
```

E01	800.00
E11	2100.00
E21	1500.00

8 record(s) selected.

You can see the use of the GROUP BY clause. Obviously you wouldn't create a summary table on a 32-row table – I only used that table to show the principle.

So let's move on to MQTs. What is the difference between a summary table and an MQT? Well, with an MQT you do not need to specify a GROUP BY clause (although you could do if you wanted to!). Just like summary tables, MQTs can be defined as system maintained or user maintained.

The SQL reference manual states "A materialized query table whose fullselect contains a GROUP BY clause is summarizing data from the tables referenced in the fullselect. Such a materialized query table is also known as a summary table. A summary table is a specialized type of materialized query table".

Let's use the above summary table and create the 'corresponding' MQT. The SQL would look like:

```
create table mqt14a
as
(select distinct(workdept) as dept, sum(bonus) as bon
from employee
group by workdept)
data initially deferred
refresh deferred
```

To populate and select from the mqt14a table you would use:

```
>db2 refresh table mqt14a
>db2 select * from mqt14a
```

As you can see, we do not have to supply the SUMMARY keyword. I think this just allows backward compatibility with V7 summary tables.

One of the real benefits of MQTs is that you don't need a GROUP BY clause, as shown below:

```
create table mqt14b
as
(select * from employee)
data initially deferred
refresh deferred
```

To populate and select from the table you would use:

```
>db2 refresh table mqt14b
>db2 select * from mqt14b
```

So why would I want to do this – aren't I just copying one table into another? Yes I am! There are two advantages to this. Firstly you can make the second table a read-only table, which you might only want to refresh overnight. Secondly, if you are using the Information Integrator functionality (introduced with UDB DB2 V8.1) you could set up an MQT based on a non-DB2 remote source.

Let's look at this in more detail. Say we have defined an Excel spreadsheet as a data source using Information Integrator V8.1 and created a nickname for this data source called restnum\$. We could create an MQT based on this nickname as follows:

```
create table restmqt
as
(select * from restnum$)
data initially deferred
refresh deferred
```

The table restmqt will be in 'check pending' state. To take it out of this state we would issue:

```
>db2 SET INTEGRITY FOR DB2ADMIN.restmqt IMMEDIATE CHECKED
```

Now we can select from the MQT:

```
>db2 select * from restmqt
```

What I have shown here is that you can create an MQT over a nickname, which is itself a reference to an Excel spreadsheet.

You can create an index on the MQTs and sample their contents from the Control Center etc.

As with the V7 summary tables, you would generally not access

MQTs directly from a query – you would still include the base table name in your query and the DB2 optimizer would decide if it was appropriate to use the MQT.

The introduction of materialized query tables in V8.1 has expanded the functionality of the V7 summary tables by allowing you to create them without using a GROUP BY parameter. You can also create an MQT over a federated nickname, which enhances your ability to access non-DB2 source data from a DB2 query.

---

*C Leonard*  
*Freelance Consultant (UK)*

© Xephon 2004

---

## Modify column attributes – part 2

*This month we conclude the code to modify column attributes.*

```
/* SYSINDEXES/SYSINDEXPART/SYSKEYS */
process_idx: PROCEDURE EXPOSE RepeatI LineForRepeatI. DB2V
  parse arg Tbc Tbn
  i = 1
  SQLQUERY = "SELECT A.PARTITION, A.IXNAME, A.IXCREATOR, ",
             "A.PQTY, A.SQTY, A.STORNAME, ",
             "STRIP(A.LIMITKEY) AS LK, A.FREEPAGE, ",
             "A.PCTFREE, A.SPACE, ",
             "B.UNIQUERULE, B.CLUSTERING, B.BPOOL, B.PGSIZE, ",
             "B.ERASERULE, B.CLOSERULE, B.INDEXTYPE, ",
             "C.COLNAME, C.COLNO, C.ORDERING, B.PIECESIZE, ",
             "D.COLTYPE ",
             "FROM SYSIBM.SYSINDEXPART A, ",
             "SYSIBM.SYSINDEXES B, ",
             "SYSIBM.SYSKEYS C, ",
             "SYSIBM.SYSCOLUMNS D ",
             "WHERE B.TBCREATOR = ' " || Tbc || "' AND ",
             "B.TBNAME = ' " || Tbn || "' AND ",
             "B.CREATOR = A.IXCREATOR AND ",
             "B.NAME = A.IXNAME AND ",
             "A.IXCREATOR = C.IXCREATOR AND ",
             "A.IXNAME = C.IXNAME AND ",
             "B.TBCREATOR = D.TBCREATOR AND ",
             "B.TBNAME = D.TBNAME AND "
```

```

        "C.COLNAME = D.NAME ",
        "ORDER BY 3, 2, 1, 19"
ADDRESS ISPEXEC "SELECT PGM(SQLISPF)";
Do While (i <= _nrows)
  If UNIQUERULE.i = "D" Then uniq = " "
  Else uniq = " UNIQUE "
  RepeatI = RepeatI + 1
  LineForRepeatI.Ø = RepeatI
  LineForRepeatI.RepeatI = "CREATE TYPE "||INDEXTYPE.i||uniq||,
                          "INDEX "||STRIP(IXCREATOR.i)||"."||,
                          IXNAME.i||" ON "||Tbc||"."||Tbn

pomcre = IXCREATOR.i
pomname = IXNAME.i
collist = ""
partN = Ø
j = 1
pompart = Ø
If PARTITION.i = Ø Then Do
  sg = "USING STOGROUP " || STRIP(STORNAME.i)
  pq = "  PRIQTY "||PQTY.i * PGSIZE.i / 1Ø24
  sq = "  SECQTY "||SQTY.i * PGSIZE.i / 1Ø24
  fr = "PCTFREE "||PCTFREE.i
  bp = "BUFFERPOOL "||BPOOL.i
  If CLOSERULE.i = "N" Then clo = "CLOSE NO"
  Else clo = "CLOSE YES"
  If ERASERULE.i = "N" Then def = "DEFER NO"
  Else def = "DEFER YES"
  piec = "PIECESIZE " || STRIP(PIECESIZE.i) || "K;"
End
Do While (i <= _nrows & IXCREATOR.i = pomcre & IXNAME.i = pomname)
  If PARTITION.i < 2 Then Do
    If j = 1 Then Do
      collist = "(" || COLNAME.i || " "
      If ORDERING.i = "A"
        Then collist = collist || "ASC"
      Else collist = collist || "DESC"
    End
  Else Do
    collist = collist || ", " || COLNAME.i || " "
    If ORDERING.i = "A"
      Then collist = collist || "ASC"
    Else collist = collist || "DESC"
  End
End
If PARTITION.i > Ø & pompart ≠ PARTITION.i Then Do
  pompart = PARTITION.i
  If C2X(Left(LK.i, 1)) = "8Ø"
  Then Do
    a = C2X(LK.i)
    If Index(a, "FF") > Ø

```

```

Then a = Left(a, Index(a, "FF") - 1)
a = x2b(a)
b = \ substr(a, 1, 1) || substr(a, 2, length(a) - 1)
c = b2x(b)
c = x2d(c,8)
End
Else c = LK.i
If COLTYPE.i = "CHAR" |,
COLTYPE.i = "VARCHAR" |,
COLTYPE.i = "LONGVAR" |,
COLTYPE.i = "DATE" |,
COLTYPE.i = "TIME" |,
COLTYPE.i = "TIMESTMP" Then c = "'" || c || "'"
If CLOSERULE.i = "N" Then clo = "NO"
Else clo = "YES"
If ERASERULE.i = "N" Then def = "NO"
Else def = "YES"
partN = partN + 1
LineForpartN.Ø = partN
LineForpartN.partN = "PART "||PARTITION.i||" VALUES("||,
c||") USING STOGROUP "||,
STRIP(STORNAME.i)||,
" PRIQTY "||PQTY.i * PGSIZE.i / 1024||,
" SECQTY "||SQTY.i * PGSIZE.i / 1024||,
" PCTFREE "||PCTFREE.i||,
" BUFFERPOOL "||BPOOL.i||,
" CLOSE "||clo||,
" DEFER "||def

End
i = i + 1
j = j + 1
End
collist = collist || ")"
RepeatI = RepeatI + 1
LineForRepeatI.Ø = RepeatI
LineForRepeatI.RepeatI = collist
If partN > Ø Then Do
Do j = 1 To partN
If j = 1 Then Do
RepeatI = RepeatI + 1
LineForRepeatI.Ø = RepeatI
LineForRepeatI.RepeatI = "CLUSTER"
End
RepeatI = RepeatI + 1
LineForRepeatI.Ø = RepeatI
If j = 1 Then LineForRepeatI.RepeatI = "("||LineForpartN.j
Else LineForRepeatI.RepeatI = LineForpartN.j
If j = partN
Then LineForRepeatI.RepeatI = LineForRepeatI.RepeatI || ");"
Else LineForRepeatI.RepeatI = LineForRepeatI.RepeatI || ","

```



```

    End
End
Else Do
    k = RepeatI
    RepeatI = RepeatI + 8
    LineForRepeatI.Ø = RepeatI
    Do jid = k + 1 To k + 8
        Select
            When jid = k + 1 Then LineForRepeatI.jid = sg
            When jid = k + 2 Then LineForRepeatI.jid = pq
            When jid = k + 3 Then LineForRepeatI.jid = sq
            When jid = k + 4 Then LineForRepeatI.jid = fr
            When jid = k + 5 Then LineForRepeatI.jid = bp
            When jid = k + 6 Then LineForRepeatI.jid = clo
            When jid = k + 7 Then LineForRepeatI.jid = def
            When jid = k + 8 Then LineForRepeatI.jid = piec
            Otherwise
                End
        End
    End
End
End
Return

/* SYSTRIGGERS */
process_trig: PROCEDURE EXPOSE RepeatTg LineForRepeatTg. DB2V
    parse arg Tbc Tbn
    SQLQUERY = "SELECT SCHEMA, NAME, SEQNO, TEXT ",
        "FROM SYSIBM.SYSTRIGGERS ",
        "WHERE TOWNER = '" || Tbc || "' AND ",
        "TBNAME = '" || Tbn || "' ",
        "ORDER BY 1, 2, 3"
    ADDRESS ISPEXEC "SELECT PGM(SQLISPF)";
    Do i = 1 To _nrows
        RepeatTg = RepeatTg + 1
        LineForRepeatTg.Ø = RepeatTg
        If i = _nrows
            Then LineForRepeatTg.RepeatTg = STRIP(TEXT.i) || "#"
            Else LineForRepeatTg.RepeatTg = STRIP(TEXT.i)
        End
    End
Return

/* SYSVIEWDEP */
process_view: PROCEDURE EXPOSE RepeatV LineForRepeatV. DB2V
    parse arg Tbc Tbn Lv1
    SQLQUERY = "SELECT COUNT(*) ",
        "FROM SYSIBM.SYSVIEWDEP ",
        "WHERE BCREATOR = '" || Tbc || "' AND ",
        "BNAME = '" || Tbn || "'"
    ADDRESS ISPEXEC "SELECT PGM(SQLISPF)";
    NUMROW = _vn1.1

```

```

If NUMROW = Ø Then Return;
SQLQUERY = "SELECT DCREATOR, DNAME ",
           "FROM SYSIBM.SYSVIEWDEP ",
           "WHERE BCREATOR = '" || Tbc || "' AND ",
           "BNAME = '" || Tbn || "'"
ADDRESS ISPEXEC "SELECT PGM(SQLISPF)";
Do i = 1 To NUMROW
  found = Ø
  Do j = 1 To RepeatV
    If LineForRepeatV.j = DCREATOR.i || " " || DNAME.i
      Then found = 1
  End
  If found = Ø Then Do
    RepeatV = RepeatV + 1
    LineForRepeatV.Ø = RepeatV
    LineForRepeatV.RepeatV = DCREATOR.i || " " || DNAME.i
    Call process_view DCREATOR.i DNAME.i Lvl + 1
  End
End
Return

/* SYSVIEWS */
process_view1: PROCEDURE EXPOSE RepeatV1 LineForRepeatV1. DB2V
parse arg Tbc Tbn
SQLQUERY = "SELECT CREATOR, NAME, SEQNO, TEXT ",
           "FROM SYSIBM.SYSVIEWS ",
           "WHERE CREATOR = '" || Tbc || "' AND ",
           "NAME = '" || Tbn || "' ",
           "ORDER BY 1, 2, 3"
ADDRESS ISPEXEC "SELECT PGM(SQLISPF)";
Do i = 1 To _nrows
  RepeatV1 = RepeatV1 + 1
  LineForRepeatV1.Ø = RepeatV1
  If i = _nrows
    Then LineForRepeatV1.RepeatV1 = STRIP(TEXT.i) || "; "
    Else LineForRepeatV1.RepeatV1 = STRIP(TEXT.i)
  End
Return

/* SYSTABAUTH */
process_auth: PROCEDURE EXPOSE RepeatAu LineForRepeatAu. DB2V
parse arg Tbc Tbn
SQLQUERY = "SELECT GRANTEE, DELETEAUTH, ",
           "INSERTAUTH, SELECTAUTH, UPDATEAUTH ",
           "FROM SYSIBM.SYSTABAUTH ",
           "WHERE TCREATOR = '" || Tbc || "' AND ",
           "TTNAME = '" || Tbn || "' AND ",
           "GRANTEETYPE = ' ' AND ",
           "GRANTEE <> USER AND ",
           "GRANTOR <> GRANTEE"

```

```

ADDRESS ISPEXEC "SELECT PGM(SQLISPF)";
Do i = 1 To _nrows
  If DELETEAUTH.i = "G" | INSERTAUTH.i = "G" &,
    SELECTAUTH.i = "G" | UPDATEAUTH.i = "G"
  Then pom2 = "WITH GRANT OPTION"
  Else pom2 = ""
  If DELETEAUTH.i <> " " & INSERTAUTH.i <> " " &,
    SELECTAUTH.i <> " " & UPDATEAUTH.i <> " "
  Then pom1 = "ALL"
  Else Do
    k = 0
    pom1 = ""
    If DELETEAUTH.i <> " " Then Do
      pom1 = pom1 || "DELETE"
      k = k + 1
    End
    If INSERTAUTH.i <> " " Then Do
      If k > 0 Then pom1 = pom1 || ", INSERT"
      Else pom1 = pom1 || "INSERT"
      k = k + 1
    End
    If SELECTAUTH.i <> " " Then Do
      If k > 0 Then pom1 = pom1 || ", SELECT"
      Else pom1 = pom1 || "SELECT"
      k = k + 1
    End
    If UPDATEAUTH.i <> " " Then Do
      If k > 0 Then pom1 = pom1 || ", UPDATE"
      Else pom1 = pom1 || "UPDATE"
      k = k + 1
    End
  End
  pom3 = ""
  gr = STRIP(GRANTEE.i)
  If STRIP(GRANTEE.i) = "PUBLIC*" Then Do
    pom3 = "AT ALL LOCATIONS"
    gr = "PUBLIC"
  End
  RepeatAu = RepeatAu + 1
  LineForRepeatAu.0 = RepeatAu
  LineForRepeatAu.RepeatAu = "GRANT " || pom1 ||,
    " ON TABLE " || Tbc || "." || Tbn ||,
    " TO " || gr ||,
    " " || pom3 || " " || pom2
  LineForRepeatAu.RepeatAu = STRIP(LineForRepeatAu.RepeatAu) || ";"
End
Return

/* SYSPLAN/SYSPACKAGE */
process_plpkg: PROCEDURE EXPOSE RepeatPP LineForRepeatPP. DB2V

```

```

parse arg Tbc Tbn
SQLQUERY = "SELECT 'REBIND PACKAGE (' || STRIP(COLLID) || '.' || ",
           "STRIP(NAME) || ')' ",
           "FROM SYSIBM.SYSPACKAGE A ",
           "WHERE EXISTS(SELECT * ",
                   "FROM SYSIBM.SYSPACKDEP ",
                   "WHERE DNAME = A.NAME AND ",
                   "BQUALIFIER = ' " || Tbc || "' AND ",
                   "BNAME = ' " || Tbn || "') ",
           "UNION ",
           "SELECT 'REBIND PACKAGE (' || STRIP(COLLID) || '.' || ",
           "STRIP(NAME) || ')' ",
           "FROM SYSIBM.SYSPACKSTMT A ",
           "WHERE STMT LIKE '%" || Tbc || "%' || Tbn || '%" AND ",
           "NOT EXISTS(SELECT * ",
                   "FROM SYSIBM.SYSPACKDEP ",
                   "WHERE DCOLLID = A.COLLID AND ",
                   "DNAME = A.NAME) ",
           "ORDER BY 1 ",
           "OPTIMIZE FOR 1 ROW";
ADDRESS ISPEXEC "SELECT PGM(SQLISPF)";
If _nrows > 0 Then Do
  Do i = 1 To _nrows
    found = 0
    Do j = 1 To RepeatPP
      If LineForRepeatPP.j = value(_vn1."strip(i,1,'0')')
        Then found = 1;
    End
    If found = 0 Then Do
      RepeatPP = RepeatPP + 1
      LineForRepeatPP.0 = RepeatPP
      LineForRepeatPP.RepeatPP = value(_vn1."strip(i,1,'0')')
    End
  End
End
SQLQUERY = ,
"SELECT STRIP(A.NAME) AS NAME, ",
           "VALUE(STRIP(B.COLLID), ' ') AS COLLID, ",
           "VALUE(STRIP(B.NAME), ' ') AS NAME1 ",
"FROM SYSIBM.SYSPLAN A LEFT OUTER JOIN SYSIBM.SYSPACKLIST B ",
           "ON A.NAME = B.PLANNAME ",
"WHERE EXISTS(SELECT * ",
           "FROM SYSIBM.SYSPLANDEP ",
           "WHERE DNAME = A.NAME AND ",
           "BCREATOR = ' " || Tbc || "' AND ",
           "BNAME = ' " || Tbn || "') ",
"UNION ",
"SELECT STRIP(A.PLNAME), ' ', ' ' ",
"FROM SYSIBM.SYSSTMT A ",
"WHERE TEXT LIKE '%" || Tbc || "%' || Tbn || '%" AND ",

```

```

        "NOT EXISTS(SELECT * ",
                "FROM SYSIBM.SYSPLANDEP ",
                "WHERE DNAME = A.PLNAME) ",
"ORDER BY 1 ",
"OPTIMIZE FOR 1 ROW";
ADDRESS ISPEXEC "SELECT PGM(SQLISPF)";
If _nrows > 0 Then Do
    MNAME = ""
    Do i = 1 To _nrows
        If MNAME <> NAME.i Then Do
            If i > 1 Then Do
                j = i - 1
                If COLLID.j <> ' ' Then stmt = stmt || ')'
                found = 0
                Do jj = 1 To RepeatPP
                    If LineForRepeatPP.jj = stmt
                        Then found = 1;
                End
                If found = 0 Then Do
                    RepeatPP = RepeatPP + 1
                    LineForRepeatPP.0 = RepeatPP
                    LineForRepeatPP.RepeatPP = stmt
                End
            End
            stmt = 'REBIND PLAN (' || NAME.i || ')'
            If COLLID.i <> ' '
                Then stmt = stmt || ' PKLIST(' || COLLID.i || '.' || NAME1.i
                MNAME = NAME.i
            End
        Else Do
            If LENGTH(stmt) + LENGTH(COLLID.i) + LENGTH(NAME1.i) > 72
                Then Do
                    RepeatPP = RepeatPP + 1
                    LineForRepeatPP.0 = RepeatPP
                    LineForRepeatPP.RepeatPP = stmt
                    stmt = '
                End
            Else stmt = stmt || ',' || COLLID.i || '.' || NAME1.i
        End
    End
    j = i - 1
    If COLLID.j <> ' ' Then stmt = stmt || ')'
    found = 0
    Do jj = 1 To RepeatPP
        If LineForRepeatPP.jj = stmt
            Then found = 1;
    End
    If found = 0 Then Do
        RepeatPP = RepeatPP + 1
        LineForRepeatPP.0 = RepeatPP
    End

```

```

        LineForRepeatPP.RepeatPP = stmt
    End
End
Return

/* TRANSLATE COLUMN NAME */
process_trans: PROCEDURE EXPOSE RepeatI LineForRepeatI. DB2V
    parse arg Tbc Tbn ColNew ColOld
    SQLQUERY = "SELECT NAME ",
               "FROM SYSIBM.SYSCOLUMNS ",
               "WHERE TBCREATOR = '" || Tbc || "' AND ",
               "TBNAME = '" || Tbn || "' AND ",
               "COLNO = " || ColOld
    ADDRESS ISPEXEC "SELECT PGM(SQLISPF)";
    If _nrows = 1 Then Do
        Do i = 1 To RepeatI
            s1 = TRANSLATE(LineForRepeatI.i)
            s2 = TRANSLATE(STRIP(NAME.1))
            Ind = INDEX(s1, s2)
            Do While (Ind > 0)
                l = LENGTH(STRIP(NAME.1))
                s1 = DELSTR(s1, Ind, l)
                s1 = INSERT(ColNew, s1, Ind - 1, LENGTH(ColNew))
                LineForRepeatI.i = DELSTR(LineForRepeatI.i, Ind, l)
                LineForRepeatI.i = ,
                    INSERT(ColNew, LineForRepeatI.i, Ind - 1, LENGTH(ColNew))
                Ind = INDEX(s1, s2)
            End
        End
    End
Return

process_transt: PROCEDURE EXPOSE RepeatTg LineForRepeatTg. DB2V
    parse arg Tbc Tbn ColNew ColOld
    SQLQUERY = "SELECT NAME ",
               "FROM SYSIBM.SYSCOLUMNS ",
               "WHERE TBCREATOR = '" || Tbc || "' AND ",
               "TBNAME = '" || Tbn || "' AND ",
               "COLNO = " || ColOld
    ADDRESS ISPEXEC "SELECT PGM(SQLISPF)";
    If _nrows = 1 Then Do
        Do i = 1 To RepeatTg
            s1 = TRANSLATE(LineForRepeatTg.i)
            s2 = TRANSLATE(STRIP(NAME.1))
            Ind = INDEX(s1, s2)
            Do While (Ind > 0)
                l = LENGTH(STRIP(NAME.1))
                s1 = DELSTR(s1, Ind, l)
                s1 = INSERT(ColNew, s1, Ind - 1, LENGTH(ColNew))
                LineForRepeatTg.i = DELSTR(LineForRepeatTg.i, Ind, l)
            End
        End
    End
Return

```

```

        LineForRepeatTg.i = ,
            INSERT(ColNew, LineForRepeatTg.i, Ind - 1, LENGTH(ColNew))
        Ind = INDEX(s1, s2)
    End
End
End
Return

```

```

process_transv: PROCEDURE EXPOSE RepeatV1 LineForRepeatV1. DB2V
parse arg Tbc Tbn ColNew ColOld
SQLQUERY = "SELECT NAME ",
            "FROM SYSIBM.SYSCOLUMNS ",
            "WHERE TBCREATOR = '" || Tbc || "' AND ",
            "TBNAME = '" || Tbn || "' AND ",
            "COLNO = " || ColOld
ADDRESS ISPEXEC "SELECT PGM(SQLISPF)";
If _nrows = 1 Then Do
    Do i = 1 To RepeatV1
        s1 = TRANSLATE(LineForRepeatV1.i)
        s2 = TRANSLATE(STRIP(NAME.1))
        Ind = INDEX(s1, s2)
        Do While (Ind > 0)
            l = LENGTH(STRIP(NAME.1))
            s1 = DELSTR(s1, Ind, l)
            s1 = INSERT(ColNew, s1, Ind - 1, LENGTH(ColNew))
            LineForRepeatV1.i = DELSTR(LineForRepeatV1.i, Ind, l)
            LineForRepeatV1.i = ,
                INSERT(ColNew, LineForRepeatV1.i, Ind - 1, LENGTH(ColNew))
            Ind = INDEX(s1, s2)
        End
    End
End
Return

```

```

process_tranrel: PROCEDURE EXPOSE RepeatAR LineForRepeatAR. DB2V
parse arg Tbc Tbn ColNew ColOld
SQLQUERY = "SELECT NAME ",
            "FROM SYSIBM.SYSCOLUMNS ",
            "WHERE TBCREATOR = '" || Tbc || "' AND ",
            "TBNAME = '" || Tbn || "' AND ",
            "COLNO = " || ColOld
ADDRESS ISPEXEC "SELECT PGM(SQLISPF)";
If _nrows = 1 Then Do
    Do i = 1 To RepeatAR
        s1 = TRANSLATE(LineForRepeatAR.i)
        s2 = TRANSLATE(STRIP(NAME.1))
        Ind = INDEX(s1, s2)
        Do While (Ind > 0)
            l = LENGTH(STRIP(NAME.1))
            s1 = DELSTR(s1, Ind, l)

```

```

        s1 = INSERT(ColNew, s1, Ind - 1, LENGTH(ColNew))
        LineForRepeatAR.i = DELSTR(LineForRepeatAR.i, Ind, 1)
        LineForRepeatAR.i = ,
            INSERT(ColNew, LineForRepeatAR.i, Ind - 1, LENGTH(ColNew))
        Ind = INDEX(s1, s2)
    End
End
End
Return

```

```

/* CREATE JOB FOR DDL/DML */

```

```

process_crejob: PROCEDURE EXPOSE RepeatDR LineForRepeatDR. ,
                                RepeatT LineForRepeatT. ,
                                RepeatR LineForRepeatR. ,
                                RepeatA LineForRepeatA. ,
                                RepeatI LineForRepeatI. ,
                                RepeatTg LineForRepeatTg. ,
                                RepeatV1 LineForRepeatV1. ,
                                RepeatAu LineForRepeatAu. ,
                                RepeatPP LineForRepeatPP. ,
                                RepeatAR LineForRepeatAR. DB2V

userid    = userid()
tick      = ''
outdsn    = tick||userid||".MODCOL.CNTL"||tick
ADDRESS TSO
If sysdsn(outdsn) = "OK" Then
    "alloc fi(dfile) da("outdsn") shr "
Else Do
    "alloc fi(dfile) da("outdsn") new ",
    " dsorg(ps) space(1,1) tracks",
    " recfm(F B) lrecl(132) blksize(27984)"
End
queue "///"||userid||"X JOB MSGCLASS=X,CLASS=A,NOTIFY="||
      userid||",REGION=4M"
queue "/* *****"
queue "/* MODIFY COLUMN(s) "
queue "/* *****"
queue "//STEP0001 EXEC PGM=IKJEFT01,DYNAMNBR=20"
queue "//SYSTSPRT DD SYSOUT=*"
queue "//SYSTSIN DD *"
queue " DSN SYSTEM("||DB2V||")"
"execio 8 diskw dfile"
Do j = 1 To RepeatPP
    Call crejob1 LineForRepeatPP.j
End
queue " RUN PROGRAM(DSNTEP2) PLAN(DSNTEP71) -"
queue " LIB('||DB2V||'710.RUNLIB.LOAD)"
queue "//SYSPRINT DD SYSOUT=*"
queue "//SYSUDUMP DD SYSOUT=*"
queue "//SYSIN DD *"

```



```

"execio 5 diskw dfile"
Do j = 1 To RepeatDR
  Call crejob1 LineForRepeatDR.j
End
Do j = 1 To RepeatT
  Call crejob1 LineForRepeatT.j
End
Do j = 1 To RepeatR
  Call crejob1 LineForRepeatR.j
End
Do j = 1 To RepeatA
  Call crejob1 LineForRepeatA.j
End
Do j = 1 To RepeatI
  Call crejob1 LineForRepeatI.j
End
Do j = 1 To RepeatV1
  Call crejob1 LineForRepeatV1.j
End
Do j = 1 To RepeatAu
  Call crejob1 LineForRepeatAu.j
End
If RepeatTg > 0 Then Do
  queue "--#SET TERMINATOR #"
  "execio 1 diskw dfile"
End
Do j = 1 To RepeatTg
  Call crejob1 LineForRepeatTg.j
End
If RepeatTg > 0 Then Do
  queue "--#SET TERMINATOR ;"
  "execio 1 diskw dfile"
End
Do j = 1 To RepeatAR
  Call crejob1 LineForRepeatAR.j
End
queue "/*"
"execio 1 diskw dfile"
queue "//"
"execio 1 diskw dfile"
"execio 0 diskw dfile(finis"
"free fi(dfile)"
"ispexec edit dataset("outdsn")"
"ispexec lmerase dataset("outdsn")"
Return

crejob1:
parse arg Text
If LENGTH(Text) > 66 Then Do
  s = ""

```

```

wN = WORDS(Text)
Do jw = 1 To wN
  If LENGTH(s) + LENGTH(WORD(Text, jw)) > 66 Then Do
    queue s
    "execio 1 diskw dfile"
    If SUBSTR(Text, 1, 2) = "--"
      Then s = "-- " || WORD(Text, jw) || " "
      Else s = WORD(Text, jw) || " "
    End
  Else Do
    s = s || WORD(Text, jw) || " "
  End
End
queue s
"execio 1 diskw dfile"
End
Else Do
  queue Text
  "execio 1 diskw dfile"
End
Return

```

---

*Nikola Lazovic*  
*DB2 System Administrator*  
*Postal Savings Bank (Serbia and Montenegro)*

© Xephon 2004

---

## Business rules as code objects

We are all aware of this trend: traditionally, DBAs were administering data objects, and now increasingly they are being asked to administer and manage code objects (COs) such as stored procedures, triggers, and User-Defined Functions (UDFs). This very important code implements the business rules of an organization. These COs are stored in an RDBMS.

As we all know, the main purpose of a DBMS is to store, manage, and access data. Now we are seeing the inclusion of programming code in the form of triggers, stored procedures, and UDFs.

We can think of COs in the same way as other database objects such as tables, views, and indexes. Why? They are controlled and managed by a DBMS such as DB2. These objects are referred to as code objects, or COs, because they are truly program code, which is managed by a database as a database object. They are administered by a new generation of DBAs called procedural DBAs who usually come from the application programming ranks.

## THE BENEFITS OF CODE OBJECTS

The best reason for using COs is to spread and promote code reusability within an organization. No organization wants to replicate code (cut and paste) on various servers or within various application programs. By using COs, they could let the code reside on a DBMS server. In this way, depending on context or activity, COs can be automatically executed or can be called from many different client programs. This reduces or eliminates the necessity to cut and paste chunks of business logic program code (reusable code) for each new application project.

Also, the use of COs has increased consistency. For obvious reasons, an organization would like to be sure that everyone is executing the same consistent code instead of various replicated code segments. There could be no assurance that the same business logic was being used if users within the organization execute their own and separate code. The organization would be assured of this consistency if all the users and database activity (with the same requirements) are using the same COs.

The more code an organization generates, the more it must maintain. But the use of COs reduces the total code maintenance effort and increases code reuse. This is achieved by moving the business logic to the database tier as COs. This results in a smaller code base to maintain and avoids various client-side copies. Since COs exist in an RDBMS, changes are made quickly in one place without replicating the changes to various workstations. This facilitates code reuse between the client

applications and the database tier. In order to be successful and agile, every IT organization must adopt this methodology for development.

## RDBMSs SUCH AS DB2 ON WINDOWS CAN HANDLE BUSINESS LOGIC

Traditional DBAs are quite concerned about the way their data objects are manipulated by applications. They are frightened by the fact that the application programmers may be unaware of core business rules within the organization that should never be broken. Why? The programmers are usually absent-minded. Now DBAs can be reassured by moving the enforcement of business rules as COs into the database tier. This move puts DBAs in charge of administering and controlling the code implementing core business rules. And the developers are required to reuse them in order to ensure that the business logic is enforced throughout the organization.

For quite a while now, many organizations have been using triggers to enforce business logic and maintain data integrity at the database tier. Let us go further by combining triggers with another powerful DB2 facility, Java UDFs. This combination enables the database to perform functions that it couldn't otherwise perform.

I will create a Java UDF that will be fired by a database trigger. One should be able to use similar techniques to utilize these DB2 features in your own environment and business settings. By now, we know very well why asking the database to enforce business rule/logic makes sense.

## ABC AUTOPARTS CO

My company wanted to make sure that it had enough auto parts available at all times. This is a business rule. The company must be able to take action as soon as the quantity of an auto part falls below a certain inventory level.

The detailed business rule states that the company must

maintain a minimum of 15 of any auto part at all times. If the inventory level goes below the minimum quantity, an e-mail is to be sent to the auto part supplier to supply an additional 15 parts.

## LET'S GET DOWN TO IT

I will be using a DB2 command line processor to create the required data objects and environment. The following steps are performed:

First, create a new database called partbdb. Also connect to the database:

```
db2 => CREATE DATABASE partbdb alias partbdb using codeset utf-8 territory us collate using system
```

```
db2 => connect to partbdb user di_owner using password
```

Next, create a table called carparts. Also populate the table with some data:

```
db2 => create table carparts (part integer not null primary key, stock smallint not null, description varchar(75) not null, supplier varchar(75) not null)
```

Please note that column 'part' is the primary key.

```
db2 => insert into carparts values (100000000, 65, 'Radiator
```

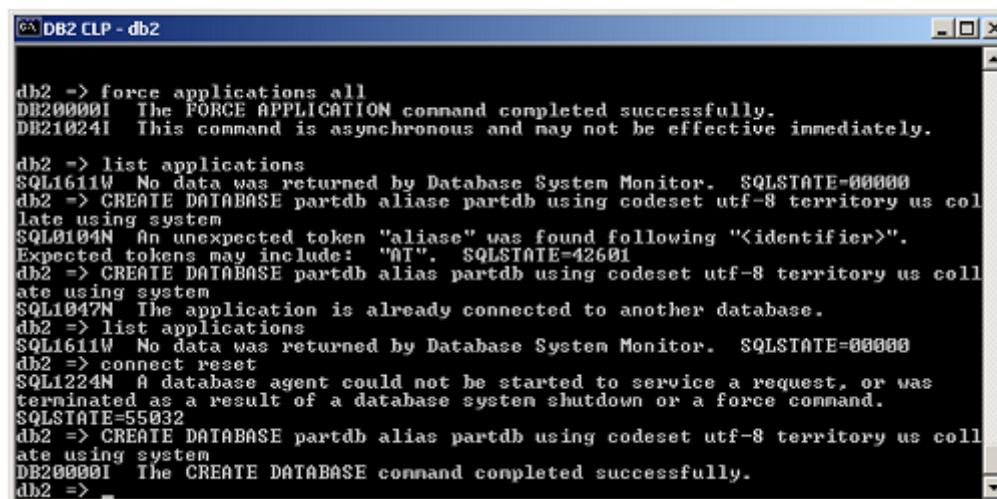


Figure 1: First screen capture

```

DB2 CLP - db2
db2 => connect to partdb user di_owner using password

Database Connection Information

Database server      = DB2/NT 8.1.0
SQL authorization ID = DI_OWNER
Local database alias = PARTDB

db2 => create table carparts (part integer not null primary key, stock smallint
not null, description varchar(75) not null, supplier varchar(75) not null)
DB20000I The SQL command completed successfully.
db2 => insert into carparts values(100000000,65,'Radiator Belt','ishaikh@bravespa
ce.com')
DB21034E The command was processed as an SQL statement because it was not a
valid Command Line Processor command. During SQL processing it returned:
SQL0104N An unexpected token "value" was found following "insert into
carparts". Expected tokens may include: "VALUES".  SQLSTATE=42601
db2 => insert into carparts values(100000000,65,'Radiator Belt','ishaikh@bravesp
ace.com')
DB20000I The SQL command completed successfully.
db2 => insert into carparts values(100000001,35,'Wiper Blades','silyas54@hotmail
.com')
DB20000I The SQL command completed successfully.
db2 =>

```

Figure 2: Second screen capture

```

Belt','supplier_one@one.com')
db2 => insert into carparts values (100000001, 35,'Wiper
Blades','supplier_two@two.com')

```

Do not forget to update the supplier e-mails with valid e-mail addresses.

Figures 1 to 3 show the screen captures of above-mentioned steps.

PART	STOCK	DESCRIPTION	SUPPLIER
100000000	65	Radiator Belt	ishaikh@bravespace.com
100000001	35	Wiper Blades	silyas54@hotmail.com

Next Rows in memory 2 [1 - 2] Filter Close Help

Figure 3: Third screen capture

Thirdly, write the trigger-fired Java UDF.

Let me re-state the business rule of my company, which states that the company must maintain a minimum of 15 of any auto part at all times. If the inventory level goes below the minimum quantity, an e-mail is to be sent to the auto part supplier to supply an additional 15 parts. The key idea in this business rule is the sending of an e-mail to the right supplier of the auto parts when the inventory level of a given auto part goes below 15. We can use the Java Mail API to send an e-mail.

The J2EE platform has a Java Mail API as a standard feature. In order to utilize this API from J2SE, I have downloaded activation.jar and mail.jar from <http://java.sun.com/products/javamail/>. The Java UDF, which we are about to write, uses these JAR files at run-time. I also had to register them as shown below:

```
db2 => call sqlj.install_jar ('file:\jars\mail.jar','MAILJAR')
db2 => call sqlj.install_jar
('file:\jars\activation.jar','ACTIVATIONJAR')
```

Note: the jar files are found in a directory called c:\jars\. A screen capture of the above events is shown in Figure 4.

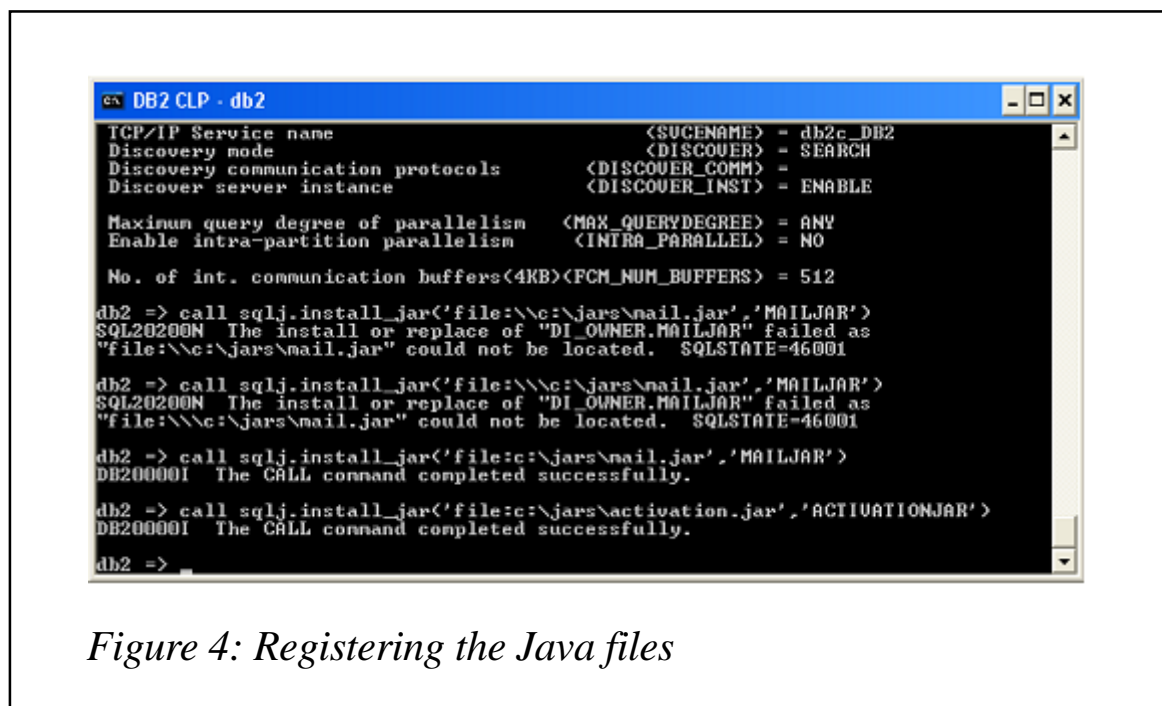


Figure 4: Registering the Java files

Our Java UDF code using the Java MAIL API would send an e-mail to a supplier. You must make a note of the following to be used in the Java UDF code:

- Your SMTP host
- Your sender e-mail address
- The recipient or *send to* e-mail address.

The UDF keeps configuration information in a *Properties* object in order to establish a mail session. Please note that the PARTSupply method is using parameters such as part, description, and supplier (e-mail). The trigger provides these parameters at the firing time when it invokes our Java UDF. The code follows:

```
import java.util.*;
import java.math.BigDecimal;
import javax.mail.*;
import javax.mail.internet.*;

public class PARTSupplyUDF
{
    public static String PARTSupply(int part, String description, String
supplier)
    {
        try
        {
            // The host url (smtp.abcautoparts.com), sender/from email,
            //                                     and send to (supplier email)
            String host = "smtp.bravespace.com";
            InetAddress from = new InetAddress("ishaikh@bravespace.com");
            InetAddress sendto = new InetAddress(supplier);

            // Retrieve the system properties
            Properties props = new Properties ();

            // Set the values for the protocol, the host, and port
            props.put("mail.smtp.protocol","smtp");
            props.put("mail.smtp.host",host);  props.put("mail.smtp.port","25");

            // Create session (triggered fired (tf))
            Session tfSession = Session.getInstance(props);

            // Create the message
            MimeMessage tfMessage = new MimeMessage(tfSession);
            tfMessage.setFrom(from);
```



```

        tfMessage.setSubject("This is a system generated order from ABC
Auto Parts Company");
        String msgText = "Our Stock is low. Please send us 15 more parts of ";
        msgText += " the part: " + description + "\n";
        msgText += " the part no. is " + part + "\n";
        tfMessage.setText(msgText);
        tfMessage.addRecipient(Message.RecipientType.TO,sendto);

        // Send the message
        javax.mail.Transport.send(tfMessage);
    }
    catch (Exception e)
    {
        System.out.println("UDF have caught an error: " + e);
        return "UDF have caught an error: " + e;
    }
    return "An Email has been sent to:" + supplier;
} // End PARTSupply
} // End PARTSupplyUDF

```

Let us discuss the code. This Java UDF implements an e-mail system by using the Java MAIL API. The code consists of the following sections:

- 1 Please modify the code in this section to update the host URL, sender/from e-mail, and send to (supplier e-mail) according to the set-up of your mail server.
- 2 Get the system properties. The configuration information is kept in a *Properties* object. This object is used to create a mail session.
- 3 Set the values for the protocol, the host, and port. These are default parameters.
- 4 Create session. This session is trigger fired (tf).
- 5 Create the message.
- 6 Send the message.

Let us compile our Java UDF code. Before compiling Java UDF, set the class path to c:\jars\mail.jar;c:\jars\activation.jar in order to use the Java MAIL API.

Place the class file on the server in the DB2 SQLLIB\FUNCTION folder. As shown below, I have placed the class file on DB2/

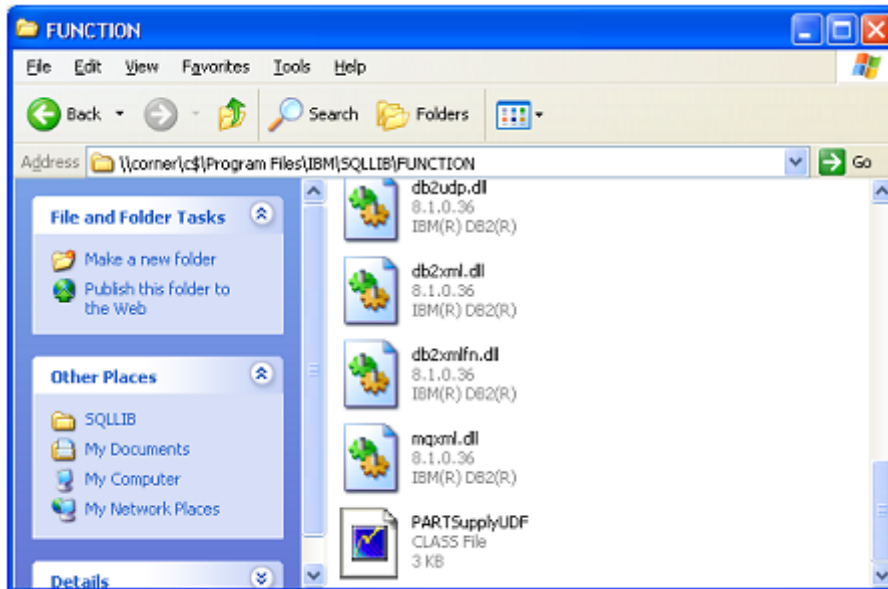


Figure 5: Class file on the server

Windows server \corner in default location c:\Program Files\IBM\SQLLIB\FUNCTION – see Figure 5.

Let us now register the UDF with DB2 (see Figure 6):

```
db2 => create function PARTSupply (part integer, description
varchar(75),
supplier varchar(75)) returns varchar(70) fenced variant no sql external
```

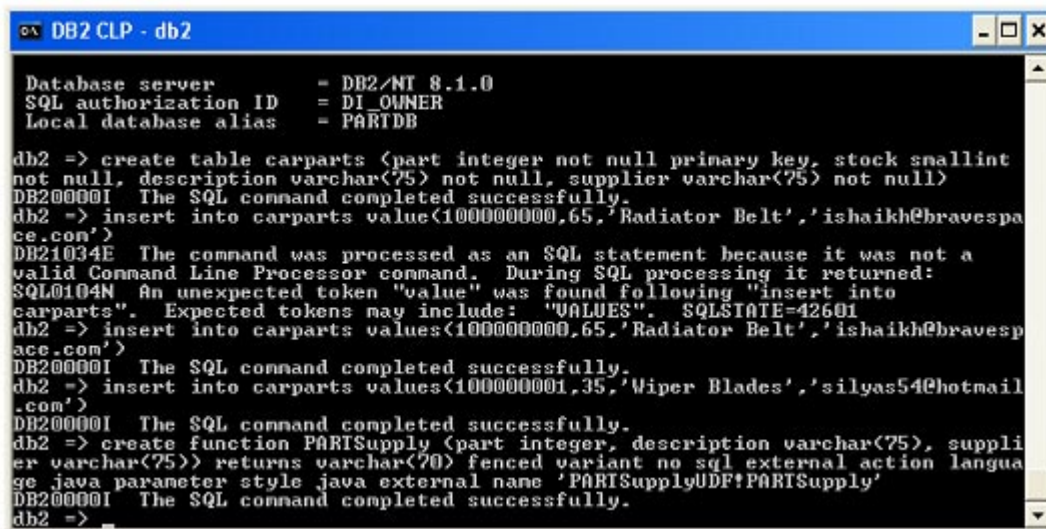
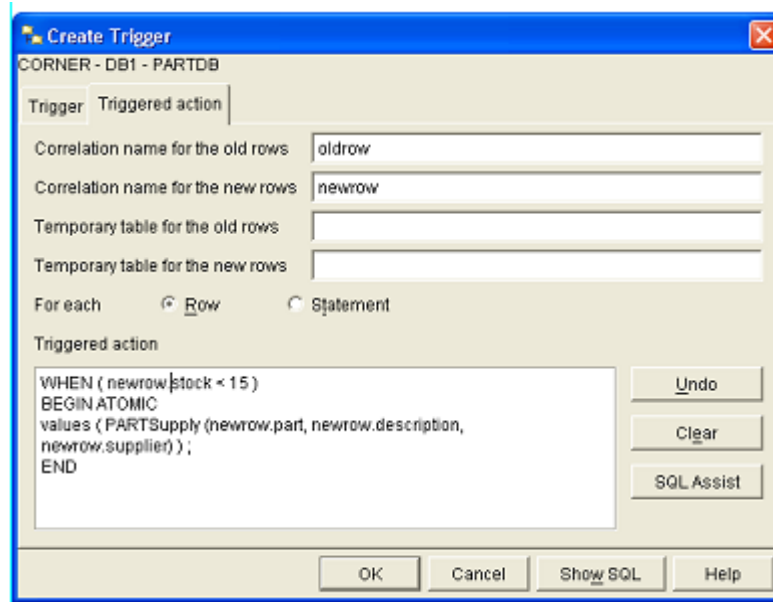
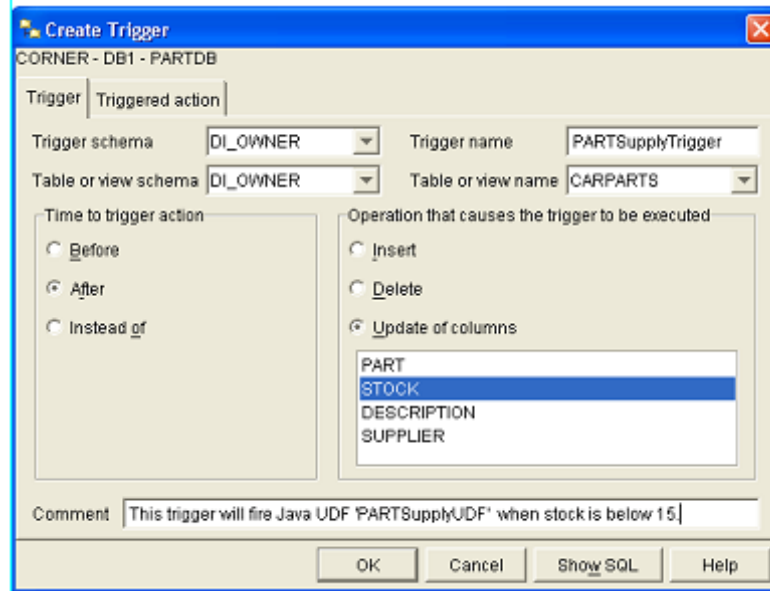


Figure 6: Register the UDF with DB2

action  
language java parameter style java external name  
'PARTSupplyUDF!PARTSupply'



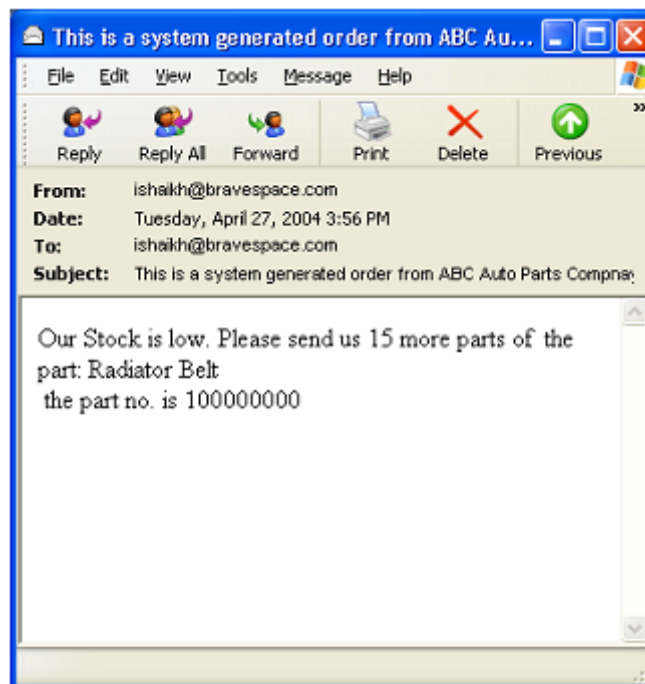
*Figure 7: Creating the trigger*

## LET US NOW CREATE THE TRIGGER

DB2 Control Center is the right tool for this step. After connecting to PARTDB database, expand the database folder and you will see a folder called 'Triggers'. Right-click and select **Create** from the context menu to start the wizard as shown in Figure 7.

Here we have two tabs as follows:

- Trigger tab – as you can see, I have provided a schema, the trigger name, and a table name. For *time to trigger*, I have selected 'After'. An update of the column STOCK will result in the firing of the trigger.
- Triggered action tab – here I have provided correlation names and the triggered action that calls our Java UDF to send an e-mail to the part supplier when the stock level goes below 15. When the trigger is fired, it passes the required parameter values for part, description, and supplier e-mail to our Java UDF.



*Figure 8: The trigger works*

## THE MOMENT OF TRUTH

Now we are ready to test our Java UDF and the associated trigger – see Figure 8.

As you can see, the test is successful and it simply simulated the condition of stock going below 15 (the trigger firing condition) by issuing an SQL update to change the STOCK value to below 15.

## WHAT HAVE WE LEARNED?

We have looked for a powerful and robust solution to implement business rules as COs. And we were successful in finding it. Let me explain.

Triggers allowed our parts database to react to the condition of stock going below an agreed level (defined condition or business rule) resulting from an update to the STOCK field (a registered event). Clearly another powerful DB2 feature, our UDF written using the rich features of Java has provided us with a mechanism to extend part database capabilities. When we combine triggers with this extensibility, we have in our hands a powerful and robust methodology to implement business rules as COs.

## RESOURCES

- 1 Introduction to the Java Mail API – <http://www.javaworld.com/javaworld/jw-06-1999/jw-06-javamail.html>.
- 2 Application Development Guide – <http://www-306.ibm.com/software/data/db2/udb/ad/v7/adg/db2a0/frame3.htm>.
- 3 Java Mail API – <http://java.sun.com/products/javamail>.

---

*Ilyas Shaikh*  
*Senior DB2 DBA*  
*BraveSpace (Canada)*

© Xephon 2004

---

## Using dynamic SQL for maximum flexibility

Most application programmers are comfortable coding embedded SQL in their programs to access DB2 data. But usually this SQL is written as static SQL. Static SQL is hard-coded, and only the values of the host variables in predicates can change.

But there is another type of SQL programming that is much more flexible than static SQL – it is known as dynamic SQL. Dynamic SQL is characterized by its capability to change the columns, tables, and predicates it references during the program's execution. This flexibility requires different techniques for embedding dynamic SQL in application programs.

You should understand what dynamic SQL is and what it can do for you for many reasons. Dynamic SQL makes optimal use of the distribution statistics accumulated by RUNSTATS. Because the values are available when the optimizer determines the access path, it can arrive at a better solution for accessing the data. Static SQL, on the other hand, cannot use these statistics unless all predicate values are hard-coded or REOPT(VARS) is specified.

Additionally, dynamic SQL is becoming more popular as distributed queries are being executed from non-mainframe platforms or at remote sites using distributed DB2 capabilities. Indeed, the JDBC and ODBC call-level interfaces deploy dynamic SQL, not static.

Using dynamic SQL is the only way to change SQL criteria such as complete predicates, columns in the SELECT list, and table names during the execution of a program. As long as application systems require these capabilities, dynamic SQL will be needed.

There are four classes of dynamic SQL – EXECUTE IMMEDIATE, non-SELECT dynamic SQL, fixed-list SELECT, and varying-list SELECT.

## EXECUTE IMMEDIATE

EXECUTE IMMEDIATE implicitly prepares and executes complete SQL statements coded in host variables. Only a subset of SQL statements are available when you use the EXECUTE IMMEDIATE class of dynamic SQL. The most important SQL statement that is missing is the SELECT statement. Therefore, EXECUTE IMMEDIATE dynamic SQL cannot retrieve data from tables.

If you do not need to issue queries, you can write the SQL portion of your program in two steps. First, move the complete text for the statement to be executed into a host variable. Second, issue the EXECUTE IMMEDIATE statement specifying the host variable as an argument. The statement is prepared and executed automatically.

The following pseudo-code shows a simple use of EXECUTE IMMEDIATE that DELETES rows from a table; the SQL statement is moved to a string variable and then executed:

WORKING-STORAGE SECTION.

```
.  
. .  
EXEC SQL  
    INCLUDE SQLCA  
END-EXEC.  
. .  
Ø1 STRING-VARIABLE.  
    49 STRING-VAR-LEN      PIC S9(4)  USAGE COMP.  
    49 STRING-VAR-TXT     PIC X(100).
```

PROCEDURE DIVISION.

```
.  
. .  
MOVE +45 TO STRING-VAR-LEN.  
MOVE "DELETE FROM DSN8810.PROJ WHERE DEPTNO = 'A00'"  
    TO STRING-VARIABLE.  
EXEC SQL  
    EXECUTE IMMEDIATE :STRING-VARIABLE
```

END-EXEC.

.  
. .  
.

You can replace the DELETE statement in this listing with any of the following supported statements:

- ALTER
- COMMENT ON
- COMMIT
- CREATE
- DELETE
- DROP
- EXPLAIN
- GRANT
- INSERT
- LABEL ON
- LOCK TABLE
- REVOKE
- ROLLBACK
- SET
- UPDATE.

Despite the simplicity of the EXECUTE IMMEDIATE statement, it usually is not the best choice for application programs that issue dynamic SQL for two reasons:

- 1 EXECUTE IMMEDIATE does not support the SELECT statement.
- 2 Performance can suffer when you use EXECUTE IMMEDIATE in a program that executes the same SQL statement many times.



After an EXECUTE IMMEDIATE is performed, the executable form of the SQL statement is destroyed. Thus, each time an EXECUTE IMMEDIATE statement is issued, it must be prepared again. This preparation is automatic and can involve a significant amount of overhead. A better choice is to code non-SELECT dynamic SQL using PREPARE and EXECUTE statements.

In general, you should consider using EXECUTE IMMEDIATE for quick, one-time tasks. For example, the following types of program are potential candidates:

- A DBA utility program that issues changeable GRANT and REVOKE statements.
- A program that periodically generates DDL based on input parameters.
- A parameter-driven modification program that corrects common data errors.

## NON-SELECT DYNAMIC SQL

The second type of dynamic SQL is known as non-SELECT dynamic SQL. This class of dynamic SQL uses PREPARE and EXECUTE to issue SQL statements. As its name implies, non-SELECT dynamic SQL cannot issue the SELECT statement. The following pseudo-code listing shows a simple use of non-SELECT dynamic SQL that DELETES rows from a table:

```
WORKING-STORAGE SECTION.
```

```
    .  
    .  
    .  
EXEC SQL  
    INCLUDE SQLCA  
END-EXEC.  
    .  
    .  
    .  
Ø1 STRING-VARIABLE.  
    49 STRING-VAR-LEN      PIC S9(4)  USAGE COMP.  
    49 STRING-VAR-TXT     PIC X(100).  
    .  
    .
```

```

      .
PROCEDURE DIVISION.
      .
      .
      .
      MOVE +45 TO STRING-VAR-LEN.
      MOVE "DELETE FROM DSN8810.PROJ WHERE DEPTNO = 'A00'"
        TO STRING-VARIABLE.
      EXEC SQL
        PREPARE STMT1 FROM :STRING-VARIABLE;
      END-EXEC.

      EXEC SQL
        EXECUTE STMT1;
      END-EXEC.
      .
      .
      .

```

You can replace the DELETE statement in this listing with any of the following supported statements:

- ALTER
- COMMENT ON
- COMMIT
- CREATE
- DELETE
- DROP
- EXPLAIN
- GRANT
- INSERT
- LABEL ON
- LOCK TABLE
- REVOKE
- ROLLBACK
- SET

- UPDATE

Non-SELECT dynamic SQL can use a powerful feature of dynamic SQL called a parameter marker, which is a placeholder for host variables in a dynamic SQL statement. This feature is demonstrated in the following pseudo-code:

```

WORKING-STORAGE SECTION.
.
.
.
EXEC SQL INCLUDE SQLCA END-EXEC.
.
.
.
Ø1 STRING-VARIABLE.
49 STRING-VAR-LEN      PIC S9(4)  USAGE COMP.
49 STRING-VAR-TXT     PIC X(100).
.
.
.
PROCEDURE DIVISION.
.
.
.
MOVE +40 TO STRING-VAR-LEN.
MOVE "DELETE FROM DSN881Ø.PROJ WHERE DEPTNO = ?"
    TO STRING-VARIABLE.
EXEC SQL
    PREPARE STMT1 FROM :STRING-VARIABLE;
END-EXEC.
MOVE 'AØØ' TO TVAL.
EXEC SQL
    EXECUTE STMT1 USING :TVAL;
END-EXEC.

```

The question mark is used as a parameter marker, replacing the 'A00' in the predicate. When the statement is executed, a value is moved to the host variable (:TVAL) and is coded as a parameter to the CURSOR with the USING clause. When this example is executed, the host variable value replaces the parameter marker.

Non-SELECT dynamic SQL can provide huge performance benefits over EXECUTE IMMEDIATE. Consider a program that executes SQL statements based on an input file. A loop in the

program reads a key value from the input file and issues a DELETE, INSERT, or UPDATE for the specified key. The EXECUTE IMMEDIATE class would incur the overhead of a PREPARE for each execution of each SQL statement inside the loop.

Using non-SELECT dynamic SQL, however, you can separate PREPARE and EXECUTE, isolating PREPARE outside the loop. The key value that provides the condition for the execution of the SQL statements can be substituted using a host variable and a parameter marker. If thousands of SQL statements must be executed, you can avoid having thousands of PREPAREs by using this technique. This method greatly reduces overhead at run-time and increases the efficient use of system resources.

A prepared statement can contain more than one parameter marker. Use as many as necessary to ease development.

## FIXED-LIST SELECT

Until now, we have been unable to retrieve rows from DB2 tables using dynamic SQL. The next two classes of dynamic SQL provide this capability. The first and simplest is fixed-list SELECT.

You can use a fixed-list SELECT statement to explicitly prepare and execute SQL SELECT statements when the columns to be retrieved by the application program are known and unchanging. You need to do so to create the proper working-storage declaration for host variables in your program. If you do not know in advance the columns that will be accessed, you must use a varying-list SELECT statement.

The following pseudo-code listing shows a fixed-list SELECT statement:

```
SQL to execute:
  SELECT  PROJNO, PROJNAME, RESPEMP
  FROM    DSN8810.PROJ
  WHERE   PROJNO   = ?
  AND     PRSDATE  = ?
  Move the "SQL to execute" to STRING-VARIABLE
```

```

EXEC SQL DECLARE CSR2 CURSOR FOR FLSQL;
EXEC SQL PREPARE FLSQL FROM :STRING-VARIABLE;
EXEC SQL OPEN CSR2 USING :TVAL1, :TVAL2;
Loop until no more rows to FETCH
EXEC SQL
    FETCH CSR2 INTO :PROJNO, :PROJNAME, :RESPEMP;
EXEC SQL CLOSE CSR2;

```

This example formulates a **SELECT** statement in the application program and moves it to a host variable. Next, a cursor is declared and the **SELECT** statement is prepared. The cursor is then opened and a loop to **FETCH** rows is invoked. When the program has finished, the cursor is closed. This example is simple because the SQL statement does not change. The benefit of dynamic SQL is its ability to modify the SQL statement. For example, you could move the SQL statement:

```

SELECT PROJNO, PROJNAME, RESPEMP
FROM DSN8810.PROJ
WHERE RESPEMP = ?
AND PRENDATE = ?

```

to the **STRING-VARIABLE** without modifying the **OPEN** or **FETCH** logic. Note that the second column of the predicate is different from the SQL statement as presented in the listing (**PRENDATE** instead of **PRSTDATE**). Because both are the same data type (**DATE**), however, you can use **TVAL2** for both if necessary. The host variables passed as parameters in the **OPEN** statement must have the same data type and length as the columns in the **WHERE** clause. If the data type and length of the columns in the **WHERE** clause change, the **OPEN** statement must be recoded with new **USING** parameters.

If parameter markers are not used in the **SELECT** statements, the markers could be eliminated and values could be substituted in the SQL statement to be executed. No parameters would be passed in the **OPEN** statement.

You can recode the **OPEN** statement also to pass parameters using an **SQLDA** (SQL Descriptor Area). The **SQLDA** would contain value descriptors and pointers to these values. You can recode the **OPEN** statement as follows:

```
EXEC-SQL
  OPEN CSR2 USING DESCRIPTOR :TVAL3;
END_EXEC.
```

DB2 uses the SQLDA to communicate information about dynamic SQL to an application program. The SQLDA sends information such as the type of the SQL statement being executed and the number and data type of columns being returned by a SELECT statement. It can be used by fixed-list SELECT and varying-list SELECT dynamic SQL. The following code illustrates the fields of the SQLDA:

```
*****
***      SQLDA: SQL DESCRIPTOR AREA FOR LE COBOL      ***
*****
Ø1  SQLDA.
    Ø5 SQLDAID          PIC X(8)   VALUE 'SQLDA'.
    Ø5 SQLDABC          COMP PIC S9(8) VALUE 13216.
    Ø5 SQLN            COMP PIC S9(4) VALUE 75Ø.
    Ø5 SQLD            COMP PIC S9(4) VALUE Ø.
    Ø5 SQLVAR OCCURS 1 TO 75Ø TIMES DEPENDING ON SQLN.
      1Ø SQLTYPE        COMP PIC S9(4).
        88 SQLTYPE-BLOB          VALUE 4Ø4 4Ø5.
        88 SQLTYPE-CLOB          VALUE 4Ø8 4Ø9.
        88 SQLTYPE-DBCLOB        VALUE 412 413.
        88 SQLTYPE-FLOAT         VALUE 48Ø 481.
        88 SQLTYPE-DECIMAL       VALUE 484 485.
        88 SQLTYPE-SMALLINT      VALUE 5ØØ 5Ø1.
        88 SQLTYPE-INTEGERS      VALUE 496 497.
        88 SQLTYPE-DATE          VALUE 384 385.
        88 SQLTYPE-TIME          VALUE 388 389.
        88 SQLTYPE-TIMESTAMP     VALUE 392 393.
        88 SQLTYPE-CHAR          VALUE 452 453.
        88 SQLTYPE-VARCHAR       VALUE 448 449.
        88 SQLTYPE-LONG-VARCHAR  VALUE 456 457.
        88 SQLTYPE-VAR-ONUL-CHAR VALUE 46Ø 461.
        88 SQLTYPE-GRAPHIC       VALUE 468 469.
        88 SQLTYPE-VARGRAPH      VALUE 464 465.
        88 SQLTYPE-LONG-VARGRAPH VALUE 472 473.
        88 SQLTYPE-ROWID         VALUE 9Ø4 9Ø5.
        88 SQLTYPE-BLOB-LOC      VALUE 961 962.
        88 SQLTYPE-CLOB-LOC      VALUE 964 965.
        88 SQLTYPE-DBCLOB-LOC    VALUE 968 969.
      1Ø SQLLEN          COMP PIC S9(4).
      1Ø SQLDATA         POINTER.
      1Ø SQLIND          POINTER.
      1Ø SQLNAME.
        15 SQLNAME1 COMP PIC S9(4).
        15 SQLNAMEC COMP PIC X(3Ø).
```

A description of the contents of the SQLDA fields is in the discussion of the next class of dynamic SQL, which relies heavily on the SQLDA.

Quite a bit of flexibility is offered by fixed-list SELECT dynamic SQL. Fixed-list dynamic SQL provides many of the same benefits for the SELECT statement as non-SELECT dynamic SQL provides for other SQL verbs. An SQL SELECT statement can be prepared once and then fetched from a loop. The columns to be retrieved must be static, however. If you need the additional flexibility of changing the columns to be accessed while executing, use a varying-list SELECT.

For fixed-list SELECT dynamic SQL, you cannot code the SQLDA in a VS/COBOL program. You will need to use LE COBOL.

## VARYING-LIST SELECT

The fourth and final class of dynamic SQL is varying-list SELECT. This class of dynamic SQL can be used to explicitly prepare and execute SQL SELECT statements when you do not know in advance which columns will be retrieved by an application program.

Varying-list SELECT provides the most flexibility for dynamic SELECT statements. You can change tables, columns, and predicates on-the-fly. Keep in mind, though, that because everything about the query can change during one invocation of the program, the number and types of host variable needed to store the retrieved rows cannot be known beforehand. The lack of knowledge regarding what is being retrieved adds considerable complexity to your application programs.

The SQLDA is the vehicle for communicating information about dynamic SQL between DB2 and the application program. It contains information about the type of SQL statement to be executed, the data type of each column accessed, and the address of each host variable needed to retrieve the columns. The SQLDA must be hard-coded into the LE COBOL program's WORKING-STORAGE area, as shown here:

```
EXEC-SQL
  INCLUDE SQLDA
END_EXEC.
```

The list below shows each item in the SQLDA when it is used with varying-list SELECT. The SQLDA data element definitions have their SQLDA field name followed by its use in DESCRIBE or PREPARE statements:

<i>NULL allowed</i>	<i>NULL not allowed</i>	<i>Data type</i>
384	385	DATE
388	389	TIME
392	393	TIMESTAMP
400	401	null-terminated graphic string
404	405	BLOB
408	409	CLOB
412	413	DBCLOB
448	449	Small VARCHAR
452	453	Fixed CHAR
456	457	Long VARCHAR
460	461	VARCHAR optionally null-terminated
464	465	Small VARGRAPHIC
468	469	Fixed GRAPHIC
472	473	Long VARGRAPHIC
480	481	FLOAT
484	485	DECIMAL
496	497	INTEGER
500	501	SMALLINT
904	905	ROWID
961	962	BLOB locator
964	965	CLOB locator
968	969	DBCLOB locator
972	973	result set locator
976	977	table locator

*Figure 1: Valid values for SQLTYPE*



- SQLDAID – descriptive only; usually set to the literal 'SQLDA' to aid in program debugging.
- SQLDABC – length of the SQLDA.
- SQLN – number of occurrences of SQLVAR available.
- SQLD – number of occurrences of SQLVAR used.
- SQLTYPE – data type and indicator of whether NULLs are allowed for the column; for UDTs, SQLTYPE is set based on the base data type.
- SQLLEN – external length of the column value; 0 for LOBs.
- SQLDATA – address of a host variable for a specific column.
- SQLIND – address of NULL indicator variable for the preceding host variable.
- SQLNAME – name or label of the column.

The steps needed to code varying-list SELECT dynamic SQL for your application program vary according to the amount of information known about the SQL beforehand. Let's walk through another pseudo-code listing showing the steps necessary when you know that the statement to be executed is a SELECT statement:

```
SQL to execute: SELECT PROJNO, PROJNAME, RESPEMP
                FROM DSN8810.PROJ
                WHERE PROJNO = 'A00'
                AND PRSTDATE = '1988-10-10';
```

```
Move the "SQL to execute" to STRING-VARIABLE
EXEC SQL DECLARE CSR3 CURSOR FOR VLSQL;
EXEC SQL
    PREPARE VLSQL INTO SQLDA FROM :STRING-VARIABLE;
EXEC SQL OPEN CSR3;
Load storage addresses into the SQLDA
Loop until no more rows to FETCH
    EXEC SQL FETCH CSR3 USING DESCRIPTOR SQLDA;
EXEC SQL CLOSE CSR3;
```

The code differs from fixed-list SELECT in three ways:

- The PREPARE statement uses the SQLDA.
- The FETCH statement uses the SQLDA.
- A step is added to store host variable addresses in the SQLDA.

When PREPARE is executed, DB2 returns information about the columns being returned by the SELECT statement. This information is in the SQLVAR group item of the SQLDA. Of particular interest is the SQLTYPE field. For each column to be returned, this field indicates the data type and whether NULLs are permitted. Note that in the SQLDA layout presented previously, all possible values for SQLTYPE are coded as 88-level COBOL structures. They can be used in the logic of your application program to test for specific data types. The valid values for SQLTYPE are shown in Figure 1.

The first value listed is returned when NULLs are not permitted; the second is returned when NULLs are permitted. These two codes aid in the detection of the data type for each column. The application program issuing the dynamic SQL must interrogate the SQLDA, analysing each occurrence of SQLVAR. This information is used to determine the address of a storage area of the proper size to accommodate each column returned. The address is stored in the SQLDATA field of the SQLDA. If the column can be NULL, the address of the NULL indicator is stored in the SQLIND field of the SQLDA. When this analysis is complete, data can be fetched using varying-list SELECT and the SQLDA information.

Note that the group item, SQLVAR, occurs 750 times. This number is the limit for the number of columns that can be returned by one SQL SELECT. You can modify the column limit number by changing the value of the SQLN field to a smaller number but not to a larger one. Coding a smaller number reduces the amount of storage required. If a greater number of columns is returned by the dynamic SELECT, the SQLVAR fields are not populated.

You can also code dynamic SQL without knowing anything

about the statement to be executed. An example is a program that must read SQL statements from a terminal and execute them regardless of statement type. You can create this type of program by coding two SQLDAs: one full SQLDA and one minimal SQLDA (containing only the first 16 bytes of the full SQLDA) that PREPAREs the statement and determines whether it is a SELECT. If the statement is not a SELECT, you can simply EXECUTE the non-SELECT statement. If it is a SELECT, PREPARE it a second time with a full SQLDA and follow the steps in the following pseudo-code listing:

```
EXEC SQL INCLUDE SQLDA
EXEC SQL INCLUDE MINSQLDA
Read "SQL to execute" from external source
Move the "SQL to execute" to STRING-VARIABLE
EXEC SQL DECLARE CSR3 CURSOR FOR VLSQL;
EXEC SQL
    PREPARE VLSQL INTO MINSQLDA FROM :STRING-VARIABLE;
IF SQLD IN MINSQLDA = Ø
    EXECUTE IMMEDIATE (SQL statement was not a SELECT)
    FINISHED.
EXEC SQL
    PREPARE VLSQL INTO SQLDA FROM :STRING-VARIABLE;
EXEC SQL OPEN CSR3;
Load storage addresses into the SQLDA
Loop until no more rows to FETCH
    EXEC SQL FETCH CSR3 USING DESCRIPTOR SQLDA;
EXEC SQL CLOSE CSR3;
```

In this section, I've provided a quick introduction to varying-list SELECT dynamic SQL. If you want to code parameter markers or need further information on acquiring storage or pointer variables, consult the appropriate compiler manuals and the following DB2 manuals:

- *DB2 Application Programming and SQL Guide.*
- *DB2 SQL Reference.*

## SUMMARY

Without proper knowledge of dynamic SQL you are going into battle without a full set of ammunition. Seriously consider using dynamic SQL under the following conditions:

- When the nature of the application program is truly changeable, not just a series of static SQL statements.
- When the columns to be retrieved can vary from execution to execution.
- When the predicates can vary from execution to execution.
- When benefit can be accrued from interacting with other dynamic SQL applications – for example, using the QMF callable interface.

---

*Craig S Mullins*  
*Director, Technology Planning*  
*BMC Software (USA)*

© Craig S Mullins 2004

---

## November 2001 – October 2004 index

Items below are references to articles that have appeared in *DB2 Update* since Issue 109, November 2001. References show the issue number followed by the page number(s). Subscribers can download copies of all issues in Acrobat PDF format from Xephon's Web site.

710utilities	112.3-19	Deadlock	117.12-32
Access path	122.7-21	Delete	135.45-47
Access programs	129.8-13	Dictionary pages	108.32-34
Authorizations	119.39-47, 120.21-35	Directory	130.23-47
Automation	134.7-23	DISPLAYDATABASE	133.3-13
Back-up	116.3-7, 128.3-6, 130.18-20	Distributed processing	124.3-9
Business rules	144.18-28	DISTSERV	136.15-21
CAF interface	129.20-47, 130.20-23	Drop table	135.23-44
Catalog	130.23-47	DSN1COPY	109.8-12, 111.26-50
CBPDO	131.3-7, 134.40-45	DSNACCOR	131.7-24
Changing attributes	139.29-47	DSNTIAUL	110.41-47, 111.20-26
Check constraints	110.18-25	DSNUTILS	138.19-28
Checking data	111.51	DSNWZP	129.14-19
CICS transactions	141.7-15	DSNZPARM	129.14-19
Code objects	144.18-28	Dynamic cursors	109.39-45, 110.13-18
Column attributes	143.22-38, 144.6-17	Dynamic SQL	136.15-21, 137.42-47, 144.28-41
COMMIT	126.25-30	Entity-relationship diagrams	126.44-51, 127.34-47
Consistency tokens	128.6-26	Erwin	135.6-14
Control statements	118.37-47	EXEC SQL	113.15-19
Copy data	131.25-38	EXPLAIN	113.4-14, 137.42-47
COPYTOCOPY	109.3-8	Federated database	121.3-7, 138.11-19
Data warehouse	112.40-51, 115.9-15	Federated systems	118.15-36
Database management	112.19-30, 113.19-47, 116.15-40	Force	143.3-8
DataPropagator	114.8-16, 120.35-50, 121.32-47, 126.8-25, 127.21-27	FREEPGE	125.26-42
Dataset placement	122.30-47, 123.39-51	Governor	112.31-40
Datasharing	140.30-47, 141.15-28	Health Center	130.3-9
DB2Everyplace	110.4-12, 122.22-29	Health Monitor	130.3-9
DB2 level display	115.5-9, 118.36	High Performance Unload	143.38-43
DB2 OLAP Server	119.6-10	Identity column	124.30-47, 126.30-44, 136.7-15
DB2 OLAP Server Analyzer	119.6-10	Image copy	143.12-22
DB2 OLAP Server Miner	119.6-10	IMWEBSRV	134.23-40
DB2 UDB V8.1	123.7-24	Indexes	121.8-18
DB2AUDIT	133.13-20	Indexspace	127.14-21
DB2BATCH	134.46-47	Infinite logging	135.3-5
db2relocatedb command	131.38-47	Insert	122.3-7, 128.46-47
DbVisualizer	114.35-41	ISPF-SQL interface	140.5-16
DDL	121.8-18	Joins	111.8-20
		Language interface	125.43-51

LISTTABLESPACE	125.3-8	Sign-on exit	113.3-4
Load times	132.29-31	Soundex	118.3-8
Log inventory	125.8-25	Space calculation	119.3-6
Log messages	130.9-17	SPUFI	132.18-29, 138.29-47, 139.17-29
Materialized Query Tables	144.3-5	SQL	138.3-11, 139.6-7, 142.6-19, 143.8-11
Memo Extension	131.3-7	Statistics	122.7-21
MERGE	140.3-5	Stored procedures	109.39-45, 110.13-18, 110.26-40, 138.19-28, 141.7-15
Messages	110.3	Subsystems	124.22-29
Monitor	115.42-47	Summary tables	114.42-47, 127.27-34, 144.3-5
Monitoring	121.32-47, 127.14-21, 139.7-17	SYSIBM.SYSLGRNX	124.10-21
Multi-dimensional clustering	129.3-7	Table access	142.3-6
NODYNAM	116.8-14	Table functions	127.3-13
Non-index data retrieval	139.3-5	Tables	123.24-28
Object manager	141.29-47, 142.20-47	Tablespaces	119.3-6, 127.14-21
ODS	120.13-21	Temporary tables	118.8-14
Package	128.27-45	Test	136.21-33
PC Utilities	115.42-47	Time dimension table	111.3-7
PCTFREE	125.26-42	Timeout	117.12-32
Performance	109.8-12, 122.30-47, 123.29-38, 123.39-51, 127.27-34	Top-ten problem	115.37-42
Perl	143.3-8	Triggers	135.6-14
PLAN	128.27-45	Tuning	123.29-38
Primary key constraints	109.31-38	UDB Extender	117.3-6
QUIESCE	134.3-6	UDB Version 7.2	114.3-7
Real-Time Statistics (RTS)	131.7-24, 140.17-29, 143.12-22-38	UDB Version 8	120.3-13
Recover	109.17-30	UDB Version 8.1.2	132.9-17
Recovery Log	119.10-39	UDB	115.16-36
Referential Integrity	123.3-7	UDF	121.18-31
Refresh	136.21-33	User-defined functions	109.12-16
Renaming	123.24-28, 140.30-47, 141.15-28	Utilities	116.40-51, 117.32-47
Replication	136.3-6	Utility lists	114.17-35
Restore	116.3-7	Verify	130.18-20
RESTRICT	136.34-47, 137.25-42	Version 8	132.3-9
RIMLIB	131.3-7	View	128.46-47
Sampling	141.3-7	Virtual storage	139.7-17
Scalar functions	142.6-19, 143.8-11	Web services	133.21-24
SELECT	125.26	Web	134.23-40, 135.14-23
Sequences	126.30-44	XCOM	137.3-22
Sequential numbers	136.7-15	XML Extender	115.3-5
SET operators	133.46-47	XPERANTO	117.6-11
SET	137.22-24	ZPARM	126.3-7, 133.24-46

Send your article for inclusion in *DB2 Update* to the editor, Trevor Eddolls, at [trevore@xephon.com](mailto:trevore@xephon.com).

Compuware has announced Version 3.1 of STROBE and Version 2.0 of iSTROBE, which are enterprise application performance management products. STROBE products are aimed at locating and eliminating sources of application inefficiencies in complex application environments.

STROBE 3.1 includes support for DB2 V8 in both new function and compatibility modes. STROBE helps users find and fix performance problems in applications using DB2. It also provides support for WebSphere Application Server, which, along with its existing support for Java, enables users to manage and improve the performance of Java and WebSphere applications. STROBE 3.1 also provides information on how Java and WebSphere applications interact with DB2, CICS, and other z/OS facilities.

iSTROBE 2.0 provides Web server architecture so users can run iSTROBE on both mainframe and distributed Web servers to improve centralized access and reduce requirements for user workstations. It also allows users to document and share their own performance hints, provides enhanced Java support, and improves report sorting.

For further information contact:  
Compuware, One Campus Martius, Detroit,  
MI 48226, USA.  
Tel: (313) 227 7300.  
URL: <http://www.compuware.com/products/strobe/default.htm>.

\* \* \*

Embarcadero has announced that it is strengthening its complete DB2 product line to support the latest versions of DB2 Universal Database (UDB) for z/OS and multi-platforms. It now provides support for DB2 UDB for z/OS Version 8.0 and DB2 Stinger.

Job Scheduler 3.1 and Change Manager 2.5 can be leveraged for the scheduling of DB2 database management tasks, or with change control in a DB2 database environment, respectively. Other products, including DBArtisan have also been enhanced, the company says.

For further information contact:  
Embarcadero Technologies, 100 California  
Street, 12th Floor, San Francisco, CA 94111,  
USA.  
Tel: (415) 834 3131.  
URL: [http://www.embarcadero.com/news/press\\_releases/db2\\_embarcadero.html](http://www.embarcadero.com/news/press_releases/db2_embarcadero.html).

\* \* \*

StorageTek has announced the introduction of its Lifecycle Director software for DB2. With the product, DB2 managers in z/OS environments can automate the transparent movement of data through its life-cycle to lower-cost storage media while delivering rapid access to archived data.

Additionally, Lifecycle Director reduces the management costs associated with archiving large DB2 databases by eliminating the need to manually update the underlying application code and minimizing data retrieval times.

By facilitating the transparent archiving and recall of data at the individual row level, Lifecycle Director can reduce active table sizes by an average of 85 percent, they claim, eliminating the over allocation of disk space.

For further information contact:  
StorageTek, One StorageTek Drive, Louisville,  
CO 80028-0001, USA.  
Tel: (303) 673 5151.  
URL: <http://storagetek.shareholder.com/releaseDetail.cfm?ReleaseID=141638>.

