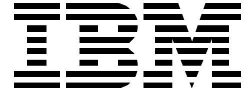
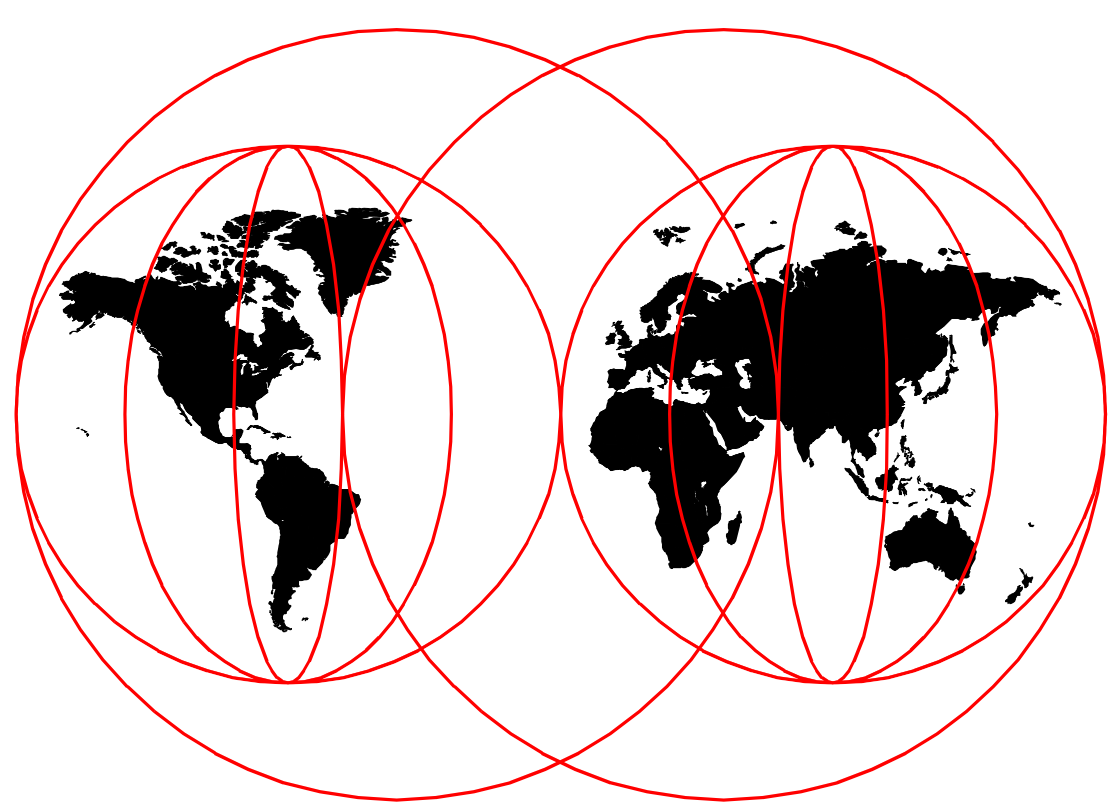


DB2 UDB for AS/400 Object Relational Support

Jarek Miszczyk, Bronach Bromley, Mark Endrei
Skip Marchesani, Deepak Pai, Barry Thorn



International Technical Support Organization

www.redbooks.ibm.com

SG24-5409-00

International Technical Support Organization

SG24-5409-00

DB2 UDB for AS/400 Object Relational Support

February 2000

Take Note!

Before using this information and the product it supports, be sure to read the general information in Appendix B, “Special notices” on page 229.

First Edition (February 2000)

This edition applies to Version 4 Release 4 of the Operating System/400 (5769-SS1).

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. JLU Building 107-2
3605 Highway 52N
Rochester, Minnesota 55901-7829

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 2000. All rights reserved.
Note to U.S Government Users - Documentation related to restricted rights - Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Contents

Figures vii

Preface xi
The team that wrote this redbook xi
Comments welcome xii

Chapter 1. Introduction 1
1.1 Why we need complex objects on an AS/400 system 1
1.2 Using complex objects 2

Chapter 2. Large object support in DB2 UDB for AS/400 3
2.1 A need for large objects 3
2.2 What is an LOB? 4
2.3 Using LOBs with SQL 5
2.3.1 Creating a table with LOB data types 5
2.3.2 Adding data to the CUSTOMERHUS table 7
2.4 LOB locators 8
2.4.1 LOB locator characteristics 8
2.4.2 LOB locator processing 9
2.4.3 Commitment control and LOB locators 12
2.5 LOB file reference variable 12
2.5.1 LOB file reference characteristics 13
2.5.2 LOB file reference processing 14
2.6 Commitment control and journaling for LOBs 18
2.7 SQL functions supporting LOBs 19
2.7.1 Basic predicate support for LOBs 19
2.7.2 Column functions 21
2.7.3 Scalar functions 21
2.8 LOBs and the native interface 22
2.9 LOB column considerations 24
2.9.1 Triggers. 25
2.9.2 Using in Net.Data 25

Chapter 3. User-defined Distinct Types (UDTs) 27
3.1 A need for user-defined types 27
3.2 Creating distinct types 28
3.2.1 Creating UDT sourced from DECIMAL 28
3.2.2 Creating a table using UDTs 29
3.2.3 Creating distinct types with the SQL interface 32
3.2.4 Altering and deleting distinct types 34
3.3 Casting for distinct types 34
3.3.1 Explicit casting 35
3.3.2 Implicit casting 38
3.3.3 Implicit casting and promotion 40
3.3.4 Implicit casting and host variables 42
3.4 SQL support for distinct types 43
3.4.1 Using predicates with UDT 44
3.4.2 Joining on UDT 45
3.4.3 Using a default value with UDT 47
3.5 DB2 UDB for AS/400 implementation 49
3.5.1 Native system interfaces 49

3.5.2 Keeping track of distinct types 57
3.5.3 Database recovery 62

Chapter 4. User Defined Functions (UDFs) 69
4.1 A need for User Defined Functions 69
4.2 UDF types 70
4.2.1 Sourced 70
4.2.2 SQL 70
4.2.3 External 71
4.3 Resolving UDF 71
4.3.1 UDF function overloading and function signature 72
4.3.2 Function path and the function selection algorithm 72
4.3.3 Parameter matching and promotion 74
4.3.4 The function selection algorithm 76
4.4 Coding UDFs 77
4.4.1 Coding sourced UDFs 78
4.4.2 Coding SQL UDFs 85
4.4.3 Coding external UDFs 95
4.5 Function resolution and parameter promotion in UDFs 108
4.5.1 An example of function resolution in UDFs 108
4.5.2 An example of parameter promotion in UDF 112
4.6 The system catalog for UDFs 116
4.6.1 SYSROUTINES catalog 116
4.6.2 SYSPARMS catalog 117
4.7 Dropping UDFs 118
4.8 Saving and restoring UDFs 119
4.9 Debugging UDFs 119
4.10 Coding considerations 127

Chapter 5. Programming alternatives for complex objects 129
5.1 Using complex objects in Java client applications 129
5.1.1 Getting ready to use JDBC 2.0 driver 129
5.1.2 Using a Blob object 130
5.1.3 Using a Clob object 134
5.1.4 Using metadata 137
5.2 Using complex objects in CLI or ODBC 139
5.2.1 DB2 CLI application flow 139
5.2.2 Passing LOB to a stored procedure written in CLI 139
5.2.3 Calling the CLI stored procedure 143
5.2.4 Retrieving LOBs in CLI 143

Chapter 6. DataLinks 147
6.1 A need for DataLinks 147
6.2 DataLinks components 150
6.2.1 DataLink data type 150
6.2.2 DataLink file manager 151
6.2.3 DataLink filter 153
6.2.4 APIs 153
6.3 DataLinks system configuration 154
6.3.1 Initializing the DLFM server 156
6.3.2 DLFM configuration 157
6.3.3 Starting the DLFM server 163
6.4 Using DataLinks with SQL 164
6.4.1 DataLink options: General 165

6.4.2 DataLink options: DB2 Universal Database for AS/400 167
6.4.3 Data manipulation examples 178
6.4.4 DataLink SQL scalar functions 182
6.4.5 Using the DataLink in dynamic Web pages 183
6.4.6 Using the DataLink access control token 186
6.5 Native interface considerations 193
6.6 DataLinks management considerations 202
6.6.1 Backup and recovery procedures 202
6.7 Using DataLinks in a heterogeneous environment 212
6.7.1 DataLinks Manager for Windows NT and for AIX 212

Appendix A. Source code listings 215
A.1 UDTLABA: Using UDTs 215
A.2 UDTLABB: Casting UDTs 216
A.3 PictCheck: External UDF 218
A.4 ChkHdr 220
A.5 RunGetPicture: Testing GetPicture UDF 220
A.6 Rating: External UDF using SCRATCHPAD 221
A.7 RtvPrdNbr3: External stored procedure written in CLI 222

Appendix B. Special notices 229

Appendix C. Related publications 231
C.1 IBM Redbooks publications 231
C.2 IBM Redbooks collections 231
C.3 Other resources 231
C.4 Referenced Web sites 232

How to get IBM Redbooks 233
IBM Redbooks fax order form 234

List of abbreviations 235

Index 237

IBM Redbooks evaluation 241

Figures

1. Pictorial demonstration of a database with large objects 3
2. LOB types 4
3. CUSTOMERHUS table 7
4. LOB file reference variables 13
5. LOB file reference variable expanded structure 13
6. Comparing lengths of CLOB values 20
7. Using "=" predicate with CLOB values 20
8. Using TRIM with CLOB values 20
9. Using "<>" predicate with CLOB values 21
10. Result of the Count function 21
11. Result of concat Customer_Number and House_Decsription 22
12. Displaying LOB data with ISQL 23
13. Displaying LOB data with the DSPPFM command 23
14. Displaying LOB column information with the DSPFFD command 24
15. New type dialog for distinct type MONEY 28
16. Casting functions registered in QSYS2/SYSROUTINES 29
17. Casting function parameters registered in QSYS2/SYSPARMS 29
18. New table dialog 30
19. Column type list in a new table dialog 30
20. Products master table 01 properties 32
21. Results window for explicit cast from MONEY to DECIMAL 36
22. UDT not equal query results 45
23. UDT IN query results 45
24. UDT JOIN query results 47
25. PRODMAST01 table properties with the UDT column default value 48
26. UDT column set using default value 49
27. UDTLABC test program results 51
28. UDTLABC job log error message 51
29. UDTLABC job log additional message information 52
30. UDTLFA display file screen 52
31. UDTLABD native I/O read results 54
32. UDTLABE native I/O results 56
33. UDTLABE job log entry with no error messages 56
34. Column read-only error 57
35. SYSTYPES catalog 58
36. SYSCOLUMNS catalog 58
37. SYSCOLUMNS catalog with SYSTYPES.SOURCE_TYPE 59
38. Operations Navigator view of user type objects 59
39. UDT properties dialog 60
40. Work with *SQLUDT objects 60
41. File field description for the PRODMAST01 table 61
42. UDT cannot be dropped error window 63
43. UDT cannot be dropped message details 64
44. UDT not found error window 66
45. Job log for UDT not found 66
46. UDT error message details 67
47. SYSCOLUMNS details for PRODMAST01 table 67
48. Function resolution algorithm 77
49. Opening up a Run SQL Scripts session 79
50. The CREATE FUNCTION statement for sourced UDF 79

© Copyright IBM Corp. 2000

51. The SUBSTR(PRDDESC, INTEGER, INTEGER) sourced function 81
52. Using the SUBSTR(PRDDESC, INTEGER, INTEGER) function in a query . . 81
53. Creating the MAX(MONEY) sourced UDF as a column function 82
54. Running the MAX(MONEY) column UDF 83
55. Creating the "+"(MONEY, MONEY) sourced UDF over arithmetic operators . 84
56. Using the "+"(MONEY, MONEY) sourced UDF 84
57. Creating an SQL UDF using the new SQL function dialog 86
58. New SQL function dialog. 86
59. Defining the input parameters for the SQL UDF 87
60. Typing in the body of the SQL UDF. 87
61. Creating an SQL UDF with UDT parameter 90
62. Using SQL UDF GetDescription(PRDDESC) in a query. 91
63. Creating the GetPicture SQL UDF which returns a BLOB as a return value . . 92
64. Calling the RunGetPicture. 94
65. The result of the call to the GetPicture SQL UDF 94
66. Running the IsGif external UDF with the SQL parameter style. 102
67. Running the IsBmp external UDF with the SQL parameter style 103
68. Creating the rating UDF with the DB2SQL parameter style 104
69. Using the rating external function with DB2SQL parameter style 108
70. Finding the number and name of the customer using the rating function 108
71. Executing the GetDescription (CHAR(5)) function 109
72. The query fails when it is run over the Prodmast01 table 110
73. Creating the GetDescrption(SRLNUMBER) sourced UDF 111
74. Running the GetDescription(SRLNUMBER) UDF. 111
75. The GetSize(CLOB(50K))SQL UDF 112
76. Running the GetSize(CLOB(50K) function 113
77. Creating the GetSize(VARCHAR(5))SQL UDF. 114
78. Running the GetSize(VARCHAR(5)) SQL UDF 115
79. Creating the GetSize(CHAR(5)) SQL UDF 115
80. Running the GetSize(CHAR(5)) function. 116
81. Content of SYSROUTINES catalog 117
82. UDF parameter details in SYSPARMS catalog. 118
83. The Work with Active Jobs screen listing all currently active jobs. 120
84. Working with the job in Session B. 121
85. Adding a breakpoint to the debug session 122
86. Invoking the IsGif(PICTURE) external UDF 123
87. Debugging the PICTCHECK service program 124
88. Checking the value of the program variables using the F11 key 125
89. Displaying the information in pointer variables using the EVAL command. . . 126
90. Displaying the contents of a variable in hexadecimal format 127
91. Using Java to display DB2 UDB for AS/400 BLOBs 132
92. Large objects in tables: The LOB approach 148
93. Large objects in tables: The DataLink approach. 149
94. DataLinks components summary 150
95. Inserting a column of data type DataLink 151
96. DLFM objects in library QDLFM 152
97. Distributed heterogeneous DLFM environment 153
98. Adding the TCP/IP server name 154
99. Adding the IP server name: IP address already configured 155
100.Adding the relational database directory entry (WRKRDBDIRE) 155
101.Initializing the DLFM tables (INZDLFM) 156
102.ADDPFXDLFM command prompt 158
103.ADDHDBDLFM command prompt 160

104.Table DFM_DBID in QDLFM library: Viewed with Operations Navigator . . . 162
105.Table DFM_FILE in QDLFM library: Viewed with Operations Navigator 162
106.Table DFM_PRFX in QDLFM library: Viewed with Operations Navigator . . . 162
107.Starting the DLFM server jobs 163
108.DLFM server jobs in Operations Navigator 164
109.DLFM server jobs in subsystem QSYSWRK 164
110.New table dialog 167
111.Inserting a DataLink column 168
112.Create table: DataLink column display 169
113.Create table: DataLink column link control Read FS/Write FS 170
114.Create table: DataLink column link control Read DB/Write Blocked 171
115.File ownership: Before linking 172
116.File ownership: After linking 173
117.Summary of DB2 Universal Database for AS/400 link control options 174
118.DataLink column with read permission DB/Write permission blocked 175
119.DataLink column with read permission FS/Write permission FS 175
120.Create table with DataLinks: SQL (Mode DB2Options) 176
121.Detailed journal entry: DataLink row insert 177
122.DSPFFD output for a table with a DataLink column 178
123.Insert with DLVALUE DataLink scalar function 178
124.DLVALUE function overloading 179
125.Table with empty DataLink column 179
126.Update with DLVALUE DataLink scalar function 180
127.Order by on DataLink column 181
128.DataLink SQL scalar functions script 182
129.Result set from the DLURLCOMPLETE scalar function 182
130.Result set from the DLURLPATH scalar function 183
131.Result set from the DLURLPATHONLY scalar function 183
132.Result sets from DLURLSCHEME and DLURLSERVER scalar functions . . 183
133.Using linked image files in HTML pages 186
134.Executing program READPM02: Direct file operations on boot1.jpg. 187
135.Executing program READPM02: Read of boot1.jpg with control token. 188
136.Executing program READPM03: Direct file operations on boot4.jpg. 192
137.Access control token: Dynamic generation 193
138.Table SPORTS for native tests 194
139.Table with DataLink input to RPG program: Error 198
140.Table with DataLink input to RPG program: Recovery 198
141.Script for save/restore exercise: Restore table before file 203
142.DSPFD of table: Link pending status after file restore. 204
143.WRKPFDL TEAMXX/SAVETABLE: Link pending 205
144.DataLink file attributes for TEAMXX/SAVETABLE 205
145.Delete from table in link pending status: Error message 206
146.WRKPFDL TEAMXX/SAVETABLE: Link pending after file restore 206
147.EDTDLFA display: Status LNKPND 207
148.EDTDLFA display: Status READY. 208
149.EDTDLFA display: Links reconciled 208
150.DSPFD of TEAMXX/SAVETABLE: Link pending status after reconciliation . 209
151.WRKPFDL TEAMXX/SAVETABLE: After link reconciliation 210
152.DSPFD of TEAMXX/SAVETABLE: Link pending status after table restore. . 211
153.WRKPFDL TEAMXX/SAVETABLE: No link pending. 211
154.The IBM heterogeneous DataLink server environment 212
155.DataLink environment for Windows NT 214
156.DataLink environment for AIX 214

Preface

Learn to efficiently use Universal Database (UDB) functions provided by the AS/400 database using the suggestions, guidelines, and examples in this redbook.This redbook is intended for programmers, analysts, and database administrators. It specifically focuses on the need to take the database applications beyond traditional numeric and character data to images, video, voice, and complex documents. By reading this redbook, you gain a broad understanding of DB2 UDB for AS/400 implementation that you can use for building a new generation of multimedia and Web-enabled database applications.

This redbook contains information that you may not find anywhere else and includes detailed coverage of the following topics:

- Large objects support
- LOB locators and LOB file reference variable processing
- User Defined Types (UDTs)
- User Defined Functions (UDFs)
- DataLinks

It also documents how the Universal Database enhancements support the object-oriented paradigms of data encapsulation and function overloading.

This redbook reports a wide range of code examples developed in several programming languages (SQL, C, Java) using different interfaces (JDBC, CLI). Prior to reading this book, you should be familiar with SQL and object-oriented programming concepts.

The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization, Rochester Center.

Jarek Miszczyk is an International Technical Support Organization Specialist at the ITSO Rochester Center. He writes extensively and teaches IBM classes worldwide in all areas of DB2 UDB for AS/400. Before joining the ITSO more than two years ago, he worked in IBM Poland as a Systems Engineer. He has over 12 years experience in the computer field. His areas of expertise include cross-platform database programming, SQL, and Object Oriented programming.

Bronach Bromley is the Technical Advisor for the Database and Languages team within the UK AS/400 Support Center. She has 16 years of experience in the IT field. She holds a degree in Business Information Technology from Bournemouth University. Her areas of expertise include databases, DB2/400, UDB2, SQL, and C++. She has written extensively on Large Complex Objects.

Mark Endrei is a Senior IT Architect in Australia. He has 10 years of experience in the application development and maintenance field. He holds a bachelor's degree in Computer Systems Engineering from Royal Melbourne Institute of Technology, and an MBA (Technology Management) from Deakin University/APESMA. His areas of expertise include C++, midrange systems, and DBMS.

© Copyright IBM Corp. 2000

Skip Marchesani retired from IBM in June of 1993 after a successful 25-year career. He is recognized by many as the leading industry expert on DB2 for AS/400. Skip is now a consultant with Custom Systems Corp, an independent consulting firm in Newton, NJ. In the past two years, he has spent much of his time teaching a variety of AS/400 topics, including Year 2000, Notes/Domino, and DB2 for AS/400. Skip spent much of his IBM career working with the Rochester Lab on projects for S/38 and AS/400 and was involved with the development of the AS/400 system. Skip is a frequent speaker for various AS/400 Technical Conferences, COMMON, and local user groups in the United States and worldwide.

Deepak Pai is a software engineer in India. He holds a degree in computer science from B.M.S College of Engineering, Bangalore India. His areas of expertise include database programming using ILE C/400 and SQL/400, client/server architecture, and native programming using COBOL/400 and RPG/400. He has written extensively on database programming in DB2/400.

Barry Thorn is a Consultant IT Specialist in IBM United Kingdom providing technical support to EMEA. He has 29 years of IT experience in IBM, including 11 years with AS/400. His areas of expertise include Business Intelligence and database. He has written papers and presentations and runs classes on AS/400 Business Intelligence and data warehouse implementation.

Thanks to the following people for their invaluable contributions to this project:

Mark Anderson
Rob Bestgen
Russ Bruhnke
John Edwards
Jim Flanagan
Kent Milligan
Cliff Nock
Tony Poirier
IBM Rochester

Comments welcome

Your comments are important to us!

We want our Redbooks to be as helpful as possible. Please send us your comments about this or other Redbooks in one of the following ways:

- Fax the evaluation form found in "IBM Redbooks evaluation" on page 241 to the fax number shown on the form.
- Use the online evaluation form found at http://www.redbooks.ibm.com/
- Send your comments in an Internet note to redbook@us.ibm.com

Chapter 1. Introduction

Object-oriented programming is rapidly gaining acceptance because it can reduce the cost and time required to build complex applications. During the last several years, a new generation of database systems, called object-relational systems, appeared in the marketplace. The object-relational database systems combine a high-level query language (SQL) and multiple views of data, which provides the ability to define new data types and functions for storage and manipulation of complex objects.

With V4R4 enhancements, IBM Rochester has set a clear direction for the product to evolve toward support for the object-oriented paradigm. The new database functions that were made available on the AS/400 include Large Binary Objects (LOBs), User Defined Types (UDTs), and User Defined Functions (UDFs). We sometimes refer to these functions as a complex object support.

UDTs are data types that you define. UDTs, such as built-in types, can be used to describe the data that is stored in columns of tables. UDFs are functions that you define. UDFs, such as built-in functions or operators, support the manipulation of UDT instances. Therefore, UDT instances are stored in columns of tables and manipulated by UDFs in SQL queries. UDTs can be internally represented in different ways. LOBs are just one example of this.

1.1 Why we need complex objects on an AS/400 system

The IT industry is undergoing very rapid changes, stimulated by the dramatic growth of Internet-based businesses. The applications used on the Internet face fundamentally different challenges than traditional host-centric applications. The new paradigm of programming has been devised and implemented for the Web to cope with problems, such as demand for very high availability, scalability, and seamless integration of heterogeneous environments.

This new programming model is based on the three-tier application architecture, which consists of the thin client, the dedicated application server, and the database server. To implement the three-tier architecture, the software vendors often use a new set of tools based on the Java technology (Applets, Servlets, Java Script, Enterprise Java Beans). Furthermore, the Web applications need to be easy to use and visually attractive. A typical Internet page contains a lot of multimedia content, such as graphics, audio, and video.

So, how does DB2 UDB for AS/400 fit into this new programming paradigm? We believe that the complex object support available in V4R4 makes the AS/400 system an excellent choice for a robust and highly scalable database server. With the LOB and Datalinks support, you can use DB2 UDB for AS/400 as a central repository of multimedia objects. UDT and UDF support allow you to reflect the object-oriented features of a Java application directly in your database design. UDTs provide for data encapsulation, and UDFs provide for function overloading and polymorphism.

© Copyright IBM Corp. 2000

1.2 Using complex objects

Generally, the DB2 UDB for AS/400 supports complex objects only through an SQL interface. On the AS/400 system, there are many different ways to work with the SQL. You can use the following methods:

- Interactive SQL in a traditional 5250 emulation
- Operations Navigator GUI
- Operations Navigator SQL script utility
- High-level language with embedded SQL
- DB2 Call Level Interface (CLI)
- Native JDBC
- SQLJ
- Client/server through ODBC, JDBC, OLE DB

The Operations Navigator provides an attractive graphical interface that allows you to perform typical database administration tasks. It allows easy access to all server administration tools, gives a clear overview of the entire database system, enables remote database management, and provides assistance for complex object manipulation. The Run SQL Scripts window lets you create, edit, run, and troubleshoot scripts of SQL statements. You can save the scripts with which you work on your PC.

In this redbook, we decided to use the Operations Navigator as a primary user interface. Most of our coding examples are written in ILE C with embedded SQL and Java language. We assume that the Client Access Express with the Operation Navigator Interface is installed on your workstation.

Refer to *AS/400 Client Access Express for Windows: Implementing V4R4M0*, SG24-5191, for more details on how to install this product.

Chapter 2. Large object support in DB2 UDB for AS/400

This chapter describes:

- Large Object Types, concepts, and benefits
- SQL functions supporting LOBs
- LOB column considerations
- LOB locators, concepts, and benefits
- LOB file references, concepts, and benefits

2.1 A need for large objects

Today's multimedia applications depend on the storage of many types of large data objects, such as X-ray images, large text documents, and audio messages. The data types provided by DB2 for AS/400 were not large enough to hold this amount of data, the limit being 32 KB. With Large Object support, the AS/400 database can store and manipulate data objects that are much larger than the current limits. In the V4R4 release of OS/400, this limit is extended to 15 MB, with future releases of the AS/400 providing an increase to 2 GB.

Figure 1 demonstrates how large objects can be used within a database. For each video title in a repository, there is a record in the database that contains the traditional information, such as how many copies there are, the rating, the artist, and so on. With Large Object support, we can also hold the actual video recording, a picture of the video cover, and the sound track for the video.

SOLD	ONHAND	RATING	ARTIST	TITLE	COVER	VIDEO	MUSIC	SCRIPT
234	59	PG-13	Arnold	The Exter minator				
13	45	R	Kevin	Dancing with Bulls				
1295	209	G	Glenn	101 Doll Imitations				
379	112	G	Buzz	Toy Glory				

Figure 1. Pictorial demonstration of a database with large objects

© Copyright IBM Corp. 2000

Table 1 shows examples of the types of data that might be required to be held in a database. It also demonstrates how large some of these typical objects can become.

Table 1. Average size for LOB objects

Object	From	To
Bank checks	45 K	-
Text	30 KB per page	40 KB per page
Small image	30 KB	40 KB
Large image	200 KB	3 MB
Color image	20 MB	40 MB
Radiology image	40 MB	60 MB
Video	1 GB per hour	-
Feature-length movie	2 GB	-
High-resolution video	3 GB per hour	-
High-resolution movie	5 GB	6 GB
High-definition TV	200 MB per second	-

The AS/400 system provides support for three Large Object data types: Binary Large Objects (BLOB), Character Large Object (CLOB), and Double Byte Large Objects (DBCLOB).

This chapter discusses how we store, access, and control these new LOB data types within DB2 UDB for AS/400 databases.

2.2 What is an LOB?

An LOB, put simply, is a Large Object. Currently, an LOB field holds a string of ordered bytes from zero to 15 MB in length. There is the potential in future releases of the AS/400 to increase this value up to 2 GB in length.

There are three different type of LOBs, each with its own definition, behavior, functionality, and so on. Figure 2 illustrates these three LOB types.

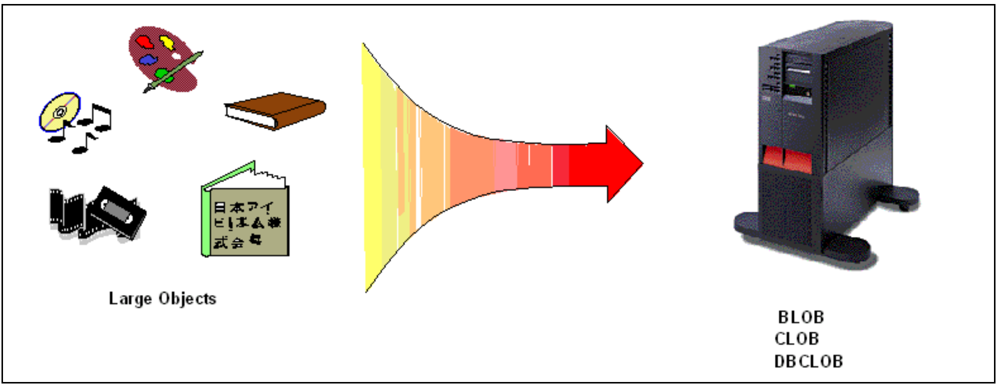


Figure 2. LOB types

The following list contains a short description of the LOB data types supported on the AS/400 system:

- **Binary Large Object (BLOB) strings**

  A Binary Large Object (BLOB) is a varying-length string with a maximum length of 15 MB. A BLOB is designed to store non-traditional data, such as pictures, voice, and mixed media. BLOBs can also store structured data for use by distinct types and user defined functions. A BLOB is considered to be a binary string. A BLOB cannot be assigned or compared with values of other data types.

- **Character Large Object (CLOB) strings**

  A Character Large Object (CLOB) is a varying-length character string with a maximum length of 15 MB and an associated code page. A CLOB is designed to store large Single Byte Character Set (SBCS) data or mixed data, such as lengthy documents, where the data could grow beyond the limits of a regular VARCHAR data type. For example, you can store information, such as an employee resume, the script of a play, or the text of novel in a CLOB. A CLOB can be assigned to, and compared with, values of other character-string data types: CHAR and VARCHAR.

- **Double-byte Character Large Object (DBCLOB) strings**

  A Double-Byte Character Large Object (DBCLOB) is a varying-length graphic string with a maximum length of 15 MB double-byte characters and an associated code page. A DBCLOB is designed to store large DBCS data, such as lengthy documents using, for example, UCS-2. A DBCLOB can be assigned to or compared with values of other double byte string data types, Graphic, and VARGRAPHIC.

In this chapter, we mainly discuss and give examples of BLOBs and CLOBs. Whenever a CLOB is discussed, a DBCLOB may be substituted.

Note

Any operation that combines an LOB type along with any of the other character types always returns a result that is an LOB. Refer to *DB2 UDB for AS/400 SQL Reference*, SC41-5612, for detailed information on casting allowed for the new data types.

2.3 Using LOBs with SQL

In this section, we document the steps required to create and insert data into a table with LOB fields. We then carry out some of the SQL functions supporting LOB data types.

2.3.1 Creating a table with LOB data types

As mentioned in 1.2, “Using complex objects” on page 2, we decided to use the Operations Navigator as a primary interface to manipulate the complex objects on the AS/400 system. By doing so, we want to encourage you to have a closer look at this powerful DB2 UDB for AS/400 interface. Where applicable, we also provide the corresponding SQL statements that can be run either in the Run SQL Scripts utility or in the traditional 5250 Interactive SQL session.

The example table we are going to create is used to store information about house details. It consists of three fields:

- **Customer Number**: A unique character data type of length 5 MB, short name CUSNUM, used to hold a reference number for a customer.
- **House Reference**: A BLOB data type of length 1 MB, short name HUSREF, used to hold a reference number for a particular house.
- **House_Description**: A CLOB data type of length 1 MB, short name HUSDES, used to hold a large string of text describing the house and its location in detail.

LOB data types are varying length types. When declaring a column of an LOB data type, you must declare its maximum length which, at the V4R4 release, can be anywhere in the range from one byte to 15 MB. The maximum length can be declared as a single integer representing a number of bytes or as an integer followed by one of the following suffixes:

- K = size value *1024, the number of kilobytes.
- M = size value * 1048576, the number of megabytes

The CCSID for a BLOB field is set to 65535 and, consequently, requires no CCSID conversion at I/O time. The default CCSID, for example, for a U.S. English-based system and for a CLOB field, is 37 and, therefore, may require conversion at I/O time. The default CCSID for a DBCLOB is 835 and, therefore, may require conversion at I/O time.

Important

To define and then manipulate a column, which is based on one of the LOB data types, you must use one of the SQL interfaces. In other words, the new data types, are not supported by either the Data Definition Specification (DDS) or native I/O functions.

The major steps required to create a table with LOB columns using the Create Table dialog of the Operations Navigator are outlined here:

1. Start Operations Navigator and expand the Database object by right-clicking the (**+**) icon next to it.
2. Select a library that should contain the new table, and right-click it. From the context menu, select **New->Table**.

   The Create New Table dialog window appears as shown in Figure 3.
3. Enter the details as shown in Figure 3.

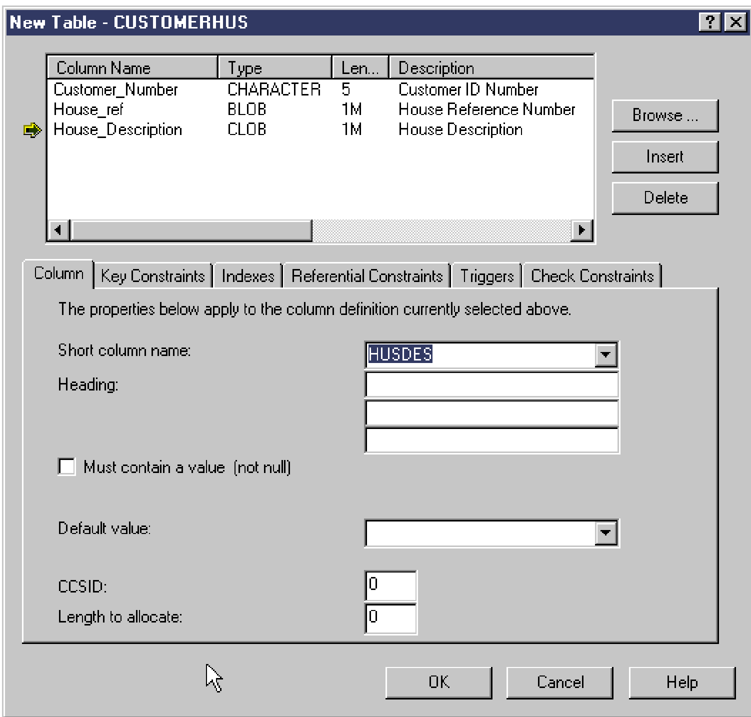


Figure 3. CUSTOMERHUS table

The corresponding SQL statement to create the CUSTOMERHUS table is shown here:

```
create table TEAMXX/CUSTOMERRHUS
    (Customer_Number     Char(5)    not null with default,
     House_Ref           Blob(1M)   not null with default,
     House_Description   Clob(1M)   not null with default)
```

2.3.2 Adding data to the CUSTOMERHUS table

We insert data into a table with LOB objects through the Operations Navigator Script utility. You can access this utility from the main Operations Navigator window by right-clicking the **Database** object and selecting **Run SQL Scripts** from its context menu.

> **Note**
>
> You can run the same SQL statements in the Interactive SQL session.

An example SQL syntax for inserting data into the CUSTOMERHUS table is as follows:

```
INSERT INTO CUSTOMERHUS VALUES ('12345, BLOB(X'1234'), 'A very long text
string');
```

Note, that in a real life application, you would probably never insert BLOB data in the way shown in this SQL statement. In 2.5, “LOB file reference variable” on page 12, we show code examples on how to use file reference variables to insert BLOB data into a table. The purpose of the above SQL statement is to illustrate important differences between LOB and CLOB data types. We use system

supplied function BLOB to insert data of type BLOB, but we don’t use the CLOB function to insert CLOB data. The reason for this is the compatibility of data types. CLOB and VARCHAR data types are both character data. Because of this, there is no problem in mapping from VARCHAR to CLOB on an insert request. The character constant, such as "A very long text string" in the INSERT statement shown previously, is treated by DB2 UDB for AS/400 as VARCHAR, and the system knows how to implicitly convert it to CLOB.

However, a BLOB value is binary data, and binary data is incompatible with character data. As a result, we are unable to automatically map from a character data type to a BLOB. The same is true for character and integer. We do not allow the insert of a character value into an integer field without first casting the character to an integer.

For more information on the BLOB function, see *DB2 UDB for AS/400 SQL Reference*, SC41-5612.

2.4 LOB locators

LOB fields can be up to 15 MB in length, which is more than the current maximum record length within the AS/400 system. This would be costly in both performance and space if we had to keep moving large objects back and forth between the database and an application.

It would be more desirable to defer the actual movement of the data from the database into the application for as long as possible and, if possible, move only those portions of data that are really needed.

For example, say a user wants to read an LOB value from one file and update a second file with that value. A poorer performing implementation would copy the LOB value into a separate buffer space at read time and then update the second file using this copy as the update image. A better performing implementation would be to defer any data movement until the update operation itself. Sometimes, however, it is reasonable to access LOB data without a locator. For example, if you know that you always need the data in a variable, you may materialize the LOB immediately instead of using a locator.

2.4.1 LOB locator characteristics

An LOB locator is intended to refer to the data we are manipulating. Operations against the LOB Locator avoid the need for copies of the data to be held in a host variable.

Conceptually, LOB locators represent a simple idea and use a small, easily managed value to refer to a much larger value. Specifically, an LOB locator is a 4-byte value stored in a host variable that a program uses to refer to an LOB value (or LOB expression) held in the database. Using an LOB locator, a program can manipulate the LOB value as if the LOB value was stored in a regular host variable

An LOB locator gives read-only access to the data it addresses. It is important to understand that the LOB locator is associated with an LOB value or LOB expression, not a row or physical storage location in the database. Once a value is selected into a locator, no operation performed on the original rows or tables

would have any effect on the value referenced by the locator. The value associated with the locator is constant until the unit of work ends, or the locator is explicitly freed, whichever comes first.

In our example, if the user were to select the LOB value into an LOB locator rather than buffer area, we would set the locator to reference the actual data in the file rather than copying the data from the file into a buffer area.

Using the LOB locator, the application program can issue subsequent database operations on the LOB value (such as applying the scalar functions SUBSTR, CONCAT, VALUE, LENGTH, doing an assignment, searching the LOB with LIKE or POSSTR, or applying UDFs against the LOB) by supplying the locator value as input. The resulting output of the locator operation, for example, the amount of data assigned to a client host variable, would then typically be a small subset of the input LOB value.

An LOB locator is only a mechanism used to refer to an LOB value during a transaction. It does not persist beyond the transaction in which it was created. Also, it is not a database data type. It is never stored in the database and, as a result, cannot participate in views or check constraints. However, since a locator is a representation of an LOB type, there are SQLTYPEs for LOB locators. They can be described within an SQLDA structure that is used by FETCH, OPEN, CALL, and EXECUTE statements.

The FREE LOCATOR statement releases a locator from its associated value. In a similar way, a commit or rollback operation frees all LOB locators associated with the transaction.

2.4.2 LOB locator processing

An LOB locator variable is a host variable that contains the locator representing an LOB value on the application server, which can be defined in the following host languages:

- C
- C++
- ILE RPG
- ILE COBOL
- PL/I

Note

The AS/400 JDBC driver uses locators under the covers so it is transparent to the client code.

A locator variable in an SQL statement must identify an LOB locator variable described in the program according to the rules for declaring locator variables. This is always indirectly through an SQL statement, for example, in C:

SQL TYPE IS BLOB_LOCATOR blobhand;
SQL TYPE IS CLOB_LOCATOR clobhand;
SQL TYPE IS DBCLOB_LOCATOR dbclobhand;

2.4.2.1 C example 1 using an LOB locator

The example program shown in this section demonstrates how to declare an LOB locator and select a CLOB value into it. It also shows how to use the locator to substring a portion of the CLOB and display it to the screen. The numbered sections of the source code are explained in the notes following this listing.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
EXEC SQL INCLUDE SQLCA;
 char dummy[[ 5 ];
void main(int argc, char **argv)
{
/* Host variable declarations */
1 EXEC SQL BEGIN DECLARE SECTION;
       SQL TYPE IS CLOB(1M) desc01;
       SQL TYPE IS CLOB_LOCATOR clobhand;
       long pos;
       long len;
EXEC SQL END DECLARE SECTION;

EXEC SQL
2      SELECT House_description INTO :clobhand
      FROM TEAMXX/CUSTOMERHUS
      WHERE CUSNUM = ('00001');

3 EXEC SQL VALUES posstr(:clobhand, 'Description') INTO :pos;
 EXEC SQL VALUES char_length(:clobhand) INTO :len;
 EXEC SQL
       VALUES substr(:clobhand, :pos, :len - 105)
       INTO :desc01;

printf(
    "The Description of the House Details for Customer 00001 is: \n" \
    "%s\n",
    desc01.data);
printf(
    "\n" \
    " Hit enter key TWICE to continue and end the program \n");
    getchar ();

gets(dummy);
exit(0);

  badnews:
     printf( "Error occured in stored procedure. SQLCODE = %5d\n",
     SQLCODE);
     gets(dummy);
     exit(1); }
```

Example 1 LOB locator program notes

1. Declare host variables. The BEGIN DECLARE SECTION and END DECLARE SECTION statements delimit the host variable declarations. Host variables are prefixed with a colon (:) when referenced in an SQL statement. A CLOB locator host variable *clobhand* is declared.
2. Select the LOB value into the Locator *clobhand* host variable. A SELECT routine is used to obtain the location of the LOB field *House_description,* in the database to a locator host variable *clobhand*.
3. Use the locator host variable *clobhand* to substring a portion of the CLOB into another CLOB *desc1*.

Our example 1 program was coded in ILE C with embedded SQL. The following CL commands show how to compile and bind the sample on the AS/400 system:

```
CRTSQLCI OBJ(TEAMXX/EXAMPLE1) COMMIT(*ALL)            1
CRTPGM PGM(TEAMXX/EXAMPLE1) MODULE(TEAMXX/EXAMPLE1)
```

CL commands note **1**

For performance reasons, we use the COMMIT(*ALL) isolation level. Refer to 2.4.3, “Commitment control and LOB locators” on page 12, for more details.

2.4.2.2 C example 2 using LOB locators

The following example program demonstrates how to declare two LOB locators. Select a CLOB value into one, use it to access specific data in the CLOB, and then use the other LOB to create a new record in the database. The numbered sections of the source code are explained in the notes following the listing.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
 EXEC SQL INCLUDE SQLCA;
 char dummy[ 5 ];
void main(int argc, char **argv)
{

 /* Host variable declarations */
1 EXEC SQL BEGIN DECLARE SECTION;
       SQL TYPE IS CLOB_LOCATOR clobhand;
       SQL TYPE IS CLOB_LOCATOR newhand;
       long pos;
       long len;
EXEC SQL END DECLARE SECTION;

/* Using a CLOB Locator to access specific data in a CLOB field  */
/* and using a new CLOB Locator to inset the retrieved data back */
/* into the table as a new record.                               */

EXEC SQL WHENEVER SQLERROR GO TO badnews;


/* Select the LOB Locator for the CLOB field House_Description   */
/* for Customer '00001' into 'clobhand'                          */
2 EXEC SQL
      SELECT House_description INTO :clobhand
      FROM TEAMXX/CUSTOMERHUS
      WHERE CUSNUM = ('00001');

/* Find the word 'Description' in the CLOB and copy the word,    */
/* 'Description' plus the CLOB's remaining trailing text         */
/* into the value for the CLOB Locator, newhand.                 */

3 EXEC SQL VALUES posstr(:clobhand, 'Description') INTO :pos;
 EXEC SQL VALUES char_length(:clobhand) INTO :len;
 EXEC SQL
        VALUES substr(:clobhand, :pos, :len - 105)
        INTO :newhand;
 /* Insert a new record into table CUSTOMERHUS for             */
 /* CUSNUM '12345', HUSREF X'4444', with a HUSDES from the     */
 /* value referenced by the CLOB Locator 'newhand'.            */
 /* INSERT into TEAMxx/CUSTOMERHUS VALUES ('12345',            */
 /* Blob(X'4444'), :newhand);                                  */

4 EXEC SQL
      INSERT into TEAMxx/CUSTOMERHUS VALUES ('12345', Blob(X'4444'),
        :newhand);

 /* To check that the record has been inserted we must first   */
 /* COMMIT the database changes i.e the INSERT.                */
 /* EXEC SQL COMMIT WORK; */

 5 EXEC SQL COMMIT WORK;
 ...
 badnews:
    ...
    error handling code
    ...
 }
```

Example 2 LOB locator program notes

1. Declare host variables. The BEGIN DECLARE SECTION and END DECLARE SECTION statements delimit the host variable declarations. Host variables are prefixed with a colon (:) when referenced in an SQL statement. Two CLOB locator host variables *clobhand* and *newhand* are declared.
2. Select the LOB value into the Locator *clobhand* host variable. A SELECT routine is used to obtain the location of the LOB field *House_description* in the database to a locator host variable *clobhand*.
3. Use the locator host variable *clobhand* to substring a portion of the CLOB into the data space for the CLOB Locator *newhand*.
4. Insert a new record into the database table with a *House_description* from the databases referred to by the CLOB Locator *newhand*,
5. Commit the SQL so that the insertion of this row could be seen through the Operations Navigator's Quick View context menu option. Refer to 2.6, "Commitment control and journaling for LOBs" on page 18, for more details on commitment control considerations while using locators to access data.

2.4.3 Commitment control and LOB locators

The commit level of *NONE is not allowed for programs using LOB locators, because DB2 UDB for AS/400 implementation requires the commitment control to cleanup the internal structures used to keep track of the locators. We recommend that you use the commit level of *ALL for programs using LOB locators if you want to achieve best performance. DB2 UDB for AS/400 doesn't have to create a copy of the LOB data when running under this isolation level. However, the down side of using this setting is a more restricted concurrent access to the underlying tables.

The following example shows how to use the SQL precompiler options to set the commitment control level of *ALL for a C program:

```
CRTSQLCI OBJ(TEAMXX/LOBLOCLB4) COMMIT(*ALL) OUTPUT(*PRINT) DBGVIEW(*SOURCE)
CRTPGM PGM(TEAMXX/LOBLOCLB4) MODULE(TEAMXX/LOBLOCLB4)
```

2.5 LOB file reference variable

File reference variables are similar to host variables, except they are used to transfer data to and from IFS files, rather than to and from memory buffers. A file reference variable represents (rather than contains) the file, just as an LOB locator represents (rather than contains) the LOB value. Database queries, updates, and inserts may use file reference variables to store or retrieve single LOB values.

For very large objects, files are natural containers. It is likely that most LOBs begin as data stored in files on the client before they are moved to the database on the server. The use of file reference variables assists in moving LOB data. Programs use file reference variables to transfer LOB data from the IFS file directly to the database engine. To carry out the movement of LOB data, the application does not have to write utility routines to read and write files using host variables.

Note

The file referenced by the file reference variable must be accessible from (but not necessarily reside on) the system on which the program runs. For a stored procedure, this would be the server.

2.5.1 LOB file reference characteristics

A file reference variable has a data type of BLOB, CLOB, or DBCLOB as shown in Figure 4. It is used either as the source of data (input) or as the target of data (output). The file reference variable may have a relative file name or a complete path name of the file. The file name length is specified within the application program. The data length portion of the file reference variable is unused during input. During output, the data length is set by the application requestor code to the length of the new data that is written to the file.

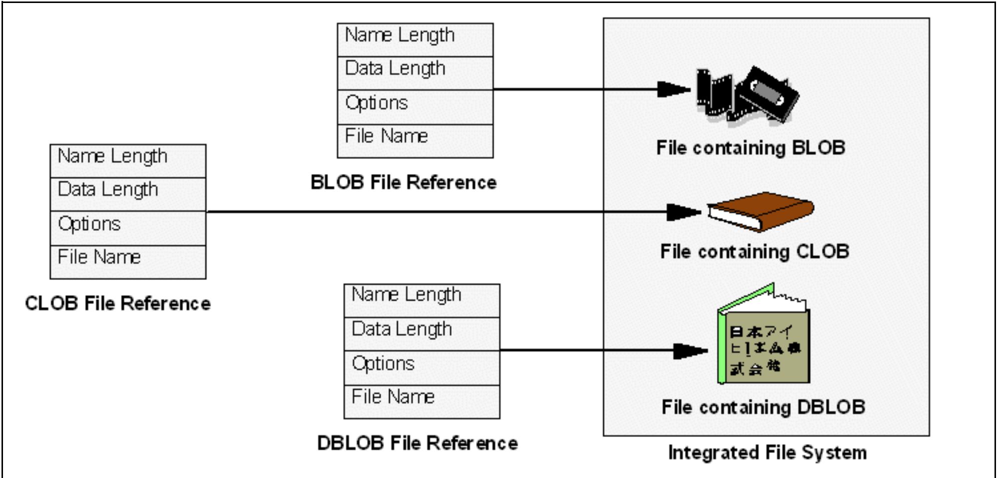


Figure 4. LOB file reference variables

Figure 5 shows an example on how the SQL precompiler expands the file reference variable into a structure containing four fields. In this example, we use a CLOB File Reference Variable *myfile_txt*.

```
SQL TYPE IS CLOB_FILE  myfile_txt;
struct {
         unsigned long  name_length;       /* Length of file name          */
         unsigned long  data_length;       /* Length of data in file       */
         unsigned long  file_options;      /* Denote usage of file         */
         char           name[255];         /* Filename                     */
        } myfile_txt;
```

Figure 5. LOB file reference variable expanded structure

When using file reference variables, there are different options on both input and output. You must choose an action for the file by setting the file_options field in

the file reference variable structure. Choices for assignment to the field covering both input and output values are shown in Table 2.

Table 2. File reference variable file options

Option	Option value	Meaning
SQL_FILE_READ	2	This is a file that can be opened, read, and closed. DB2 determines the length of the data in the file (in bytes) when opening the file. DB2 then returns the length through the data_length field of the file reference variable structure.
SQL_FILE_CREATE	8	This option creates a new file. Should the file already exist, an error message is returned.
SQL_FILE_OVERWRITE	16	This option creates a new file if none already exist. If the file already exists, the new data overwrites the data in the file.
SQL_FILE_APPEND	32	This option has the output appended to the file if it exists. Otherwise, it creates a new file.

2.5.1.1 CCSID and file reference variables

As stated earlier, a large object of data type BLOB has a CCSID of 65535 associated with it, and no conversion is carried out on this data type.

An LOB of data type CLOB can have a CCSID associated with it. If the file option CREATE is used, where the column in the table to be auctioned is a CLOB, the created file has the same CCSID as the column in the database table. For example, if a CLOB column in a table is created with CCSID of 37, which represents US English, and a file reference variable with a file option CREATE is used on that column, the resulting created file will also have a CCSID of 37. This means that the character data is stored in EBCDIC format. If you want to store a CLOB value in a file that is used by a PC application, you need to force an EBCDIC to ASCII conversion while writing the data into the file. You can achieve this by creating a dummy file in the IFS with an ASCII code page of 437 and use the file option OVERWRITE to write the new data to the file. For example, you can map an IFS directory to a network drive on your PC and use WordPad to create an empty file myfile.txt in this directory. Then, you can use the file reference variable with the file_option set to SQL_FILE_OVERWRITE to copy the CLOB column data into myfile.txt. The data is then converted on the fly from EBCDIC to ASCII by the database manager.

2.5.2 LOB file reference processing

An LOB file reference variable is used for direct file input and output for an LOB, which can be defined in the following host languages:

- C
- C++
- ILE RPG
- ILE COBOL
- PL/1

Since these are not native data types, SQL extensions are used, and the precompilers generate the host language constructs necessary to represent each variable. A file reference variable represents, rather than contains, the file.

Database queries, updates, and inserts may use file references variables to store or retrieve single column values.

As with all host variables, a file reference variable may have an associated indicator variable.

2.5.2.1 C example 3 using LOB file reference variables

The following example demonstrates how to declare an LOB file reference variable and move data from the Integrated File System to a CLOB column in a database table using the file option READ. The numbered sections are explained in the notes that follow.

```
 /* DB2 UDB for AS/400 File Reference Test Program               */
 /* Use a File Reference to move data from the IFS to a column in */
 /* the table USERPROGRAMS. This is using the File option Code    */
 /* SQL_FILE_READ.                                                */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

EXEC SQL INCLUDE SQLCA;
char dummy[ 5 ];
void main(int argc, char **argv)
{
 /* Host variable Declaration                                     */
 /* CLOB FILE REFERENCE host variable set up.                     */
 /* for the File Reference txt_file.                              */
1 EXEC SQL BEGIN DECLARE SECTION;
    SQL TYPE IS CLOB_FILE txt_file;
 EXEC SQL END DECLARE SECTION;

 EXEC SQL WHENEVER SQLERROR GO TO badnews;

 /*set up the txt_file variable                                   */
 2  strcpy(txt_file.name, "/TEAMXX/Text_Files/qcsrc.txt");
  txt_file.name_length = strlen(txt_file.name);
  txt_file.file_options = SQL_FILE_READ;
 /* Insert the File Reference txt_file for option Read,           */
 /* into the CLOB column of the USERPROGRAMS table, for           */
 /* a Identiy Number of 100.                                      */

 3 EXEC SQL
       INSERT INTO TEAMXX/USERPROGRAMS
       VALUES ('100','C', :txt_file);

  EXEC SQL COMMIT WORK;
   ...
 badnews:
    ...
    error handling code
    ...
  }
```

Example 3 LOB file reference variable program notes

1. Declare Host Variable. The BEGIN DECLARE SECTION and END DECLARE SECTION statements delimit the host variable declarations. Host variables are prefixed with a colon (:) when referenced in an SQL statement. A CLOB File Reference host variable *txt_file is* declared.
2. CLOB file reference host variable is set up. The attributes of the file reference are set up.

   Note: A file name without a fully declared path is, by default, placed in the user's current directory. If the pathname does not begin with the forward slash(/) character, it is not qualified.

   String copy the full pathname into *name*, the string length of name into *name_length*, and the usage of file, in this case READ, into *file_options*.

3. Insert a new record in the database table USERPROGRAMS. Insert the file reference variable *txt_file* for option read into the CLOB column of the database table.

2.5.2.2 C example 4 using LOB file reference variables

The following example program demonstrates how to declare an LOB file reference variable and move data from the Integrated File System to a CLOB column in a database table using the file option READ. Then, manipulate the data in the LOB column via the LOB Locator, and send the manipulated data to a file in the IFS via a file reference variable. The numbered sections are explained in the notes that follow.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

EXEC SQL INCLUDE SQLCA;
  char dummy[ 5 ];
  void main(int argc, char **argv)
{
1 EXEC SQL BEGIN DECLARE SECTION;
     SQL TYPE IS CLOB_FILE txt_file;
     SQL TYPE is CLOB_FILE outtxt_file;
     SQL TYPE is CLOB_LOCATOR clobhand;
     SQL TYPE is CLOB_LOCATOR newhand;
     long pos;
     long len;
EXEC SQL END DECLARE SECTION;

EXEC SQL WHENEVER SQLERROR GO TO badnews;

/*set up the txt_file variables     */
2 strcpy(txt_file.name, "/TEAMXX/Text_Files/lobloclab2.txt");
txt_file.name_length = strlen(txt_file.name);
txt_file.file_options = SQL_FILE_READ;

/* set up the outtext_file variable */
strcpy(outtxt_file.name, "/TEAMXX/Text_Files/loblab2bak.txt");
outtxt_file.name_length = strlen(outtxt_file.name);
outtxt_file.file_options = SQL_FILE_OVERWRITE;

/* Insert the File Reference txt_file for option Read, */
/* into the CLOB column of the USERPROGRAMS table, for */
/* a Identity Number of 999.                           */

3 EXEC SQL
     INSERT INTO TEAMXX/USERPROGRAMS
     VALUES ('999','C', :txt_file);

/* Select the column PRMSRC (which was created from a    */
/* file reference) form USERPROGRAMS where IDNUM is      */
/* equal to '999'.                                       */
4 EXEC SQL
     SELECT PRMSRC INTO :clobhand
     FROM TEAMXX/USERPROGRAMS
     WHERE IDNUM = ('999');

/* Manipulate the data using the CLOB handler, so that   */
/* we find the string IBM Corp, and insert in front of   */
/* it '& 2000'. When the manipulation is complete the    */
/* text line should look like this :                     */
/* Copyright (c) 1999 & 2000 IBM Corp                    */

5 EXEC SQL VALUES posstr(:clobhand, 'IBM Corp') INTO :pos;
EXEC SQL VALUES char_length(:clobhand) INTO :len;

EXEC SQL
     SET :newhand =
6         concat(substr(cast(:clobhand as clob(200k)),
                 1, :pos -1 ), ' & 2000 ');

EXEC SQL
6    VALUES concat(cast(:newhand as clob(200k)),
```

```
     substr(cast(:clobhand as clob(200k)), :pos, :len - :pos))
     INTO :clobhand;

/* Insert the now maipulated data, via an LOB Locator     */
/* into a new record in table USERPROGRAMS.               */

 EXEC SQL
     INSERT INTO TEAMXX/USERPROGRAMS VALUES ('919','C', :clobhand);

  printf(
 /* Select column PRMSRC (CLOB) from table USERPROGRAMS  */
 /* where IDNUM = '919', into File Reference outtxt_file */
 /* That is, move the manipulated data in the CLOB to    */
 /* the file in the IFS system referenced by outtxt_file */

 7 EXEC SQL
   SELECT PRMSRC INTO :outtxt_file FROM USERPROGRAMS
   WHERE IDNUM = '919';

 EXEC SQL COMMIT WORK;
 ...
 badnews:
    ...
    error handling code
    ...
 }
```

Example 4 LOB file reference variable program notes

1. Declare Host Variable. The BEGIN DECLARE SECTION and END DECLARE SECTION statements delimit the host variable declarations. Host variables are prefixed with a colon (:) when referenced in an SQL statement. Two CLOB File Reference host variables *txt_file* and *outtext_file* are declared. Two CLOB LOB locators *clobhand* and *newhand* are also declared.

2. CLOB file reference host variables are set up. The attributes of the file reference are set up.

   String copy the full pathname of *lobloclab2.txt* into *txt_file.name*, string length of name into *name_length*, and the usage of file, in this case READ, into *file_options*.

   String copy the full pathname of *loblab2bak.txt* into *outtext_file.name*, string length of name into *name_length*, and the usage of file, in this case OVERWRITE, into *file_options*.

   > **Note**
   >
   > The file option OVERWRITE is used because we have a file in the IFS with a CCSID that allows us to view the contents of the file through Operations Navigator.

3. Create a new record in the database table USERPROGRAMS where column PRMSRC is referenced by the file reference *txt_file*.

4. The LOB locator *clobhand* is set to reference the column PRMSRC (which was created from the file reference).

5. Manipulate the data in the column via the LOB locators *clobhand* and *newhand* to add some characters before a certain point in the LOB. Put the manipulated data into the data space for LOB locator *newhand.*

6. Cast the locator to its underlying CLOB type.

Note

This cast is required since the database manager does not know the size of the locator result when it is validating the *concat(..., substr(..))* operation. First, the database manager tries to determine the result size of the substring. Because there is no size associated with any LOB Locator, and because the substring in this program uses host variables rather than constants, the database cannot assess the size of the locator operand at validation time. Consequently, it chooses the maximum LOB size. After choosing the max LOB size, the concatenation of even a single byte literal along with the substring result will result in exceeding the maximum result size limit. One way of getting around this is to cast the CLOB Locator to a CLOB of a defined length:

```
EXEC SQL
  VALUES concat( substr(cast(:clobhand as clob(200k)), 1, :pos -1 ),
  ' & 2000 ')
```

By doing this, the validation code is able to use 200 KB as the maximum size of the result rather than 15 MB.

7. Move the contents of the manipulated column into the file reference *outtxt_file* that has a file option of overwrite. We selected this file option so that we could view the resulting manipulated column in the IFS via Operations Navigator. See 2.5.1.1, “CCSID and file reference variables” on page 14, for an explanation of CCSID and file reference variable’s interfaces.

2.6 Commitment control and journaling for LOBs

As you probably noticed in the coding examples discussed in the previous sections, you need to commit your changes to an LOB object before other users can get access to this object. The reason why "dirty data" is not viewable for records that hold LOB data is pretty straightforward. It is a result of the way the database returns LOB data at read time. For every other data type on the AS/400 system, a read of a record makes a copy of the data in the user's I/O buffer. However, for LOBs, the database does not return a copy of the data.

DB2 UDB for AS/400 treats LOB data differently primarily because of a restriction on the amount of data that can be returned for each record. For now, the database is only able to return 32 KB of data for each record. The performance cost of changing that limit to be able to accommodate 15 MB (in the future 2 GB) of copied data was too high.

Because the database returns a pointer rather than a copy of the data, it must ensure that the LOB data addressed does not change while the pointer is in use. The LOB data that is addressed in one cursor cannot be updated or deleted by another cursor. We cannot allow the holder of this address to look at LOB data that has been updated. Even if an LOB has been deleted, the disk location may be reused and end up holding other data. For security reasons, the database must ensure that the data it is addressing stays fixed while a user is looking at it.

Even though we may not be running under commitment control, we need to acquire a read lock on the record (or, in a Get Multiple, the records) that we read.

We do not escalate the lock level from *None to *Cursor Stability but, instead, simply get read locks on each record. We hold these locks until the next read request or until the cursor is closed.

The bottom line is that, since we work with addresses into the data space, rather than copies of the data, we can never allow another cursor to read a record that holds LOB data without acquiring at least a read lock on that record.

The following example illustrates this database behavior. We insert a new record into the CUSTOMERHUS table with the following SQL statement:

```
INSERT INTO customerhus
VALUES ('11111', BLOB(X'1111'), CLOB('This is a house description'))
```

This statement is run under the commitment control level of *CHG. Now, we switch to another session, which runs under a commitment level of *NONE, and issue the following SQL statement:

```
SELECT customer_number, HEX(CAST(house_ref AS BLOB(1k)))
FROM customerhus
```

Since we didn't commit our changes in the first session, the database manager is not able to obtain the required read lock on the newly inserted record, and the SELECT statement times out with the inquiry message QRY5050 `Record in use`. Note, that if the CUSTOMER table had not contained LOB columns, the SELECT statement would have run to completion with uncommitted changes visible in the session run under the commitment control level of *NONE.

2.7 SQL functions supporting LOBs

This section describes the basic predicates, and the most important column and scalar function that support LOB data types. For a complete list of functions that support the new data types, refer to *DB2 UDB for AS/400 SQL Reference*, SC41-5612.

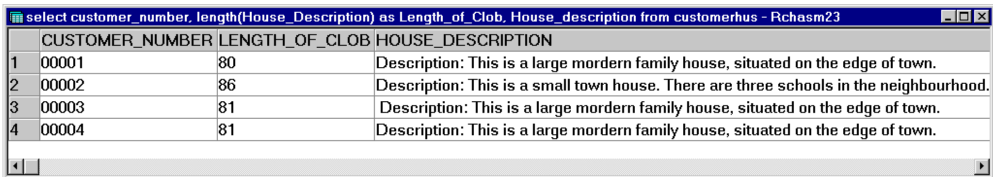
2.7.1 Basic predicate support for LOBs

The basic predicates supported are:"=", ">", "<", "¬=", "<>", "<=", "¬>", ">=", "¬<"

2.7.1.1 Comparing CLOB data

When comparisons are made between CLOB data, the database runtime removes the *trailing* spaces first and then compares the character values. Note that the leading spaces are *not* trimmed.

To clarify this, let's take a closer look at the following example. From the data that we inserted into the CUSTOMERHUS table, we can see that Customer_Number 00001, 00003, and 00004 have a House_Description CLOB value, which looks the same as the example shown in Figure 6 on page 20. Notice the actual length of the House_Description column values.



select customer_number, length(House_Description) as Length_of_Clob, House_description from customerhus - Rchasm23

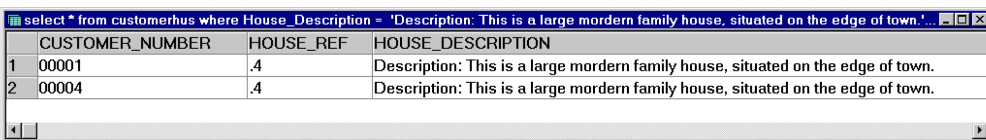
	CUSTOMER_NUMBER	LENGTH_OF_CLOB	HOUSE_DESCRIPTION
1	00001	80	Description: This is a large mordern family house, situated on the edge of town.
2	00002	86	Description: This is a small town house. There are three schools in the neighbourhood.
3	00003	81	Description: This is a large mordern family house, situated on the edge of town.
4	00004	81	Description: This is a large mordern family house, situated on the edge of town.

Figure 6. Comparing lengths of CLOB values

Now, let’s check how the"=" basic predicate works with CLOB values. In the Run SQL Scripts utility, run the following SQL statement:

```
select * from customerhus where House_Description =  'Description: This is a
large modern family house, situated on the edge of town.';
```

Note that the result set from this query, as shown in Figure 7, only contains the records for Customer_Number "00001" and "00004."



select * from customerhus where House_Description = 'Description: This is a large mordern family house, situated on the edge of town.'...

	CUSTOMER_NUMBER	HOUSE_REF	HOUSE_DESCRIPTION
1	00001	.4	Description: This is a large mordern family house, situated on the edge of town.
2	00004	.4	Description: This is a large mordern family house, situated on the edge of town.

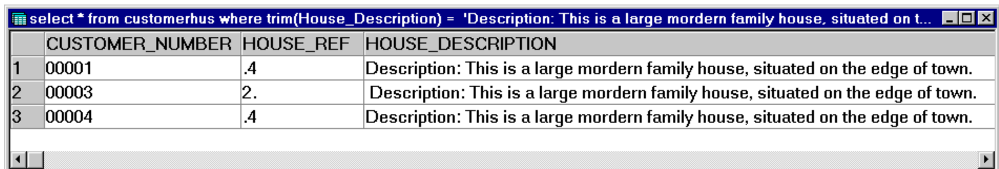
Figure 7. Using "=" predicate with CLOB values

You may now wonder why the row for Customer_Number "00003" is *not* shown in the result set. The reason for this is that the database manager first removes the trailing space from the CLOB value in row 00004 so that the values in rows 0001 and 00004 are exactly the same as the character constant used in the equal expression in the WHERE clause of our SELECT statement. The House_Description column for row 00003 still has the leading space, so it doesn’t satisfy the search condition.

Now, we modify the SELECT statement to remove both trailing and leading spaces. To achieve this, we use the TRIM function as shown here:

```
select * from customerhus where trim(House_Description) =  'Description: This
is a large modern family house, situated on the edge of town.';
```

Figure 8 shows the results for this statement.



select * from customerhus where trim(House_Description) = 'Description: This is a large mordern family house, situated on t...

	CUSTOMER_NUMBER	HOUSE_REF	HOUSE_DESCRIPTION
1	00001	.4	Description: This is a large mordern family house, situated on the edge of town.
2	00003	2.	Description: This is a large mordern family house, situated on the edge of town.
3	00004	.4	Description: This is a large mordern family house, situated on the edge of town.

Figure 8. Using TRIM with CLOB values

As expected, this time the result set contains three rows.

The next example uses the *not equal* predicate. In the Run SQL Script window, run the following SQL statement:

```
select * from customerhus where House_Description <> 'Description: This is a
large modern family house, situated on the edge of town.';
```

The results are shown in Figure 9.

select * from customerhus where House_Description <> 'Description: This is a large mordern family house, situated on the edge...

	CUSTOMER_NUMBER	HOUSE_REF	HOUSE_DESCRIPTION
1	00002	#A	Description: This is a small town house. There are three schools in the neighbourhood.
2	00003	2.	Description: This is a large mordern family house, situated on the edge of town.

Figure 9. Using "<>" predicate with CLOB values

Notice that *Customer_Number 00003* is displayed as part of this result set.

2.7.1.2 Comparing BLOB data

When comparisons are made between BLOB data types, the first comparison is of their length. Once the lengths are found to be equal, a comparison of the binary contents is made. If it is not equal, no further comparison is made. It applies the predicate to test their binary content only if the database finds out that the length of the objects is equal. For example, the following expression is evaluated as FALSE:

```
blob(X'123456') > blob(X'1234')
```

Because the lengths of the two BLOB objects are different, no further comparison is made. In other words, blob(`X'123456'`) is neither greater, nor equal, nor smaller than blob ('1234'). Because it's different, only the '<>' predicate is evaluated as TRUE.

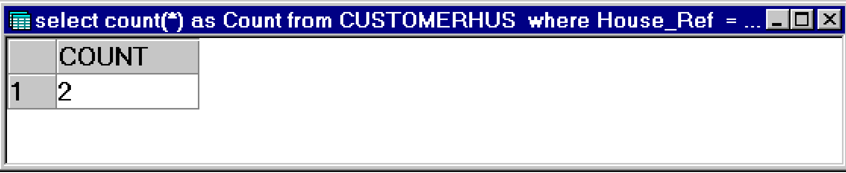
2.7.2 Column functions

The most important column functions supporting LOBs are: COUNT() and COUNT ALL. In our example, we count the number of times a particular House Reference Number is present in the CUSTOMERHUS table using the COUNT function. Remember, we have declared the House Reference Number as data type BLOB.

In the Run SQL Script window, run the following SQL statement:

```
select count(*) as Count from CUSTOMERHUS where House_Ref = blob(X'1234');
```

The result set for this query is shown in Figure 10.



select count(*) as Count from CUSTOMERHUS where House_Ref = ...

	COUNT
1	2

Figure 10. Result of the Count function

Again, you can see the method of comparing the LOBs at work. The database first compares the length. Then, if it is equal, it compares the binary content.

2.7.3 Scalar functions

The most important scalar functions supporting LOBs are: CHAR(), CONCAT(), LENGTH(), and SUBSTR().

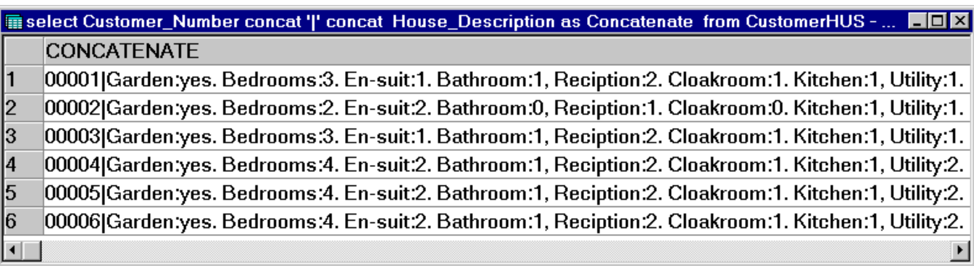
In Figure 6 on page 20, we demonstrated how the LENGTH scalar function works. Now, we will concatenate the Customer_Number (Character 5) with the House_Description (of type CLOB) using the CONCAT function.

In an operation where one of the operands is an LOB, the result from the operation is always an LOB. For example, if you added a Hex value to a BLOB, the result is always of data type BLOB.

In the Run SQL Script window, run the following SQL statement, where '|' represents the pipe character:

```
select Customer_Number concat '|' concat  House_Description as Concatenate
from CustomerHUS
```

The results of the query are shown in Figure 11.



select Customer_Number concat '|' concat House_Description as Concatenate from CustomerHUS - ...

<table>
<tr><th></th><th>CONCATENATE</th></tr>
<tr><td>1</td><td>00001|Garden:yes. Bedrooms:3. En-suit:1. Bathroom:1, Reciption:2. Cloakroom:1. Kitchen:1, Utility:1.</td></tr>
<tr><td>2</td><td>00002|Garden:yes. Bedrooms:2. En-suit:2. Bathroom:0, Reciption:1. Cloakroom:0. Kitchen:1, Utility:1.</td></tr>
<tr><td>3</td><td>00003|Garden:yes. Bedrooms:3. En-suit:1. Bathroom:1, Reciption:2. Cloakroom:1. Kitchen:1, Utility:1.</td></tr>
<tr><td>4</td><td>00004|Garden:yes. Bedrooms:4. En-suit:2. Bathroom:1, Reciption:2. Cloakroom:1. Kitchen:1, Utility:2.</td></tr>
<tr><td>5</td><td>00005|Garden:yes. Bedrooms:4. En-suit:2. Bathroom:1, Reciption:2. Cloakroom:1. Kitchen:1, Utility:2.</td></tr>
<tr><td>6</td><td>00006|Garden:yes. Bedrooms:4. En-suit:2. Bathroom:1, Reciption:2. Cloakroom:1. Kitchen:1, Utility:2.</td></tr>
</table>

Figure 11. Result of concat Customer_Number and House_Decsription

2.8 LOBs and the native interface

As previously stated, LOBs are *not* supported by either DDS or native I/O. The only interface that allows access to the new data types is SQL. However, we conducted a number of tests to identify the behavior of various traditional 5250 interfaces with tables containing LOB columns.

Our tests were based on the CUSTOMERHUS table. Its definition is shown in Figure 3 on page 7. The table was created with both BLOB and CLOB columns.

First, we check how the Interactive SQL utility handles LOB data:

1. Open a 5250 session, and at the command prompt, start the Interactive SQL session with the following CL command:

   ```
   STRSQL
   ```

   > **Note**
   >
   > Make sure that you have the DB2 Query Manager and SQL Development Kit for AS/400 (5769-ST1) license program installed on your AS/400 system.

2. To display the rows contained in the CUSTOMERHUS table, type the following SQL statement at the ISQL prompt:

   ```
   select * from customerhus
   ```

   The results are shown in Figure 12.

```
                              Display Data
                                            Data width . . . . . . :      94
 Position to line  . . . . .             Shift to column  . . . . . .
 ....+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....
 Customer Number  House Ref Nbr                                    House Description
      00001       *POINTER                                          *POINTER
      00002       *POINTER                                          *POINTER
      00003       *POINTER                                          *POINTER
      00004       *POINTER                                          *POINTER
 ********  End of data  ********

                                                                          Bottom
F3=Exit   F12=Cancel   F19=Left   F20=Right   F21=Split
```

Figure 12. Displaying LOB data with ISQL

Note, that no content for LOB objects is displayed. Instead, the ISQL shows *POINTER as the value for both BLOB and CLOB columns.

We can also display the content of the CUSTOMERHUS table with the Display Physical File Member (DSPPFM) command. Again, the values for the LOB columns are displayed as *POINTER. The results are shown in Figure 13.

```
                          Display Physical File Member
 File . . . . . . :   CUSTO00001          Library  . . . . :   TEAMXX
 Member . . . . . :   CUSTO00001          Record . . . . . :   1
 Control  . . . . .                       Column . . . . . :   1
 Find . . . . . . .
 *...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+...
 00001                          *POINTER                        *POINTER
 00002                          *POINTER                        *POINTER
 00003                          *POINTER                        *POINTER
 00004                          *POINTER                        *POINTER
```

Figure 13. Displaying LOB data with the DSPPFM command

The Display File Field Description (DSPFFD) CL command has been updated in V4R4 so that it can now be used to display column-level information for a file containing LOBs. The results of running the DSPFFD command for the CUSTOMERHUS table are shown in Figure 14 on page 24.

```
                              Display Spooled File
File  . . . . . :   QPDSPFFD                           Page/Line   1/26
Control . . . . .                                      Columns     1 - 78
Find  . . . . . .
*...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+...
 Field Level Information
             Data       Field  Buffer    Buffer       Field    Column
   Field     Type       Length  Length  Position      Usage    Heading
   CUSNUM    CHAR            5       5         1      Both     Customer Num
     Field text  . . . . . . . . . . . . . . . :  Customer ID Number
     Alternative name  . . . . . . . . . . . . :  CUSTOMER_NUMBER
     Allows the null value
     Coded Character Set Identifier  . . . . . :     37
   HUSREF    BLOB      1048576      43         6      Both     House Ref Nb
     Field text  . . . . . . . . . . . . . . . :  House Reference Number
     Alternative name  . . . . . . . . . . . . :  HOUSE_REF
     Allocated Length  . . . . . . . . . . . . :      0
     Allows the null value
     Coded Character Set Identifier  . . . . . :  65535
   HUSDES    CLOB      1048576      32        49      Both     House Descri
     Field text  . . . . . . . . . . . . . . . :  House Description
     Alternative name  . . . . . . . . . . . . :  HOUSE_DESCRIPTION
     Allocated Length  . . . . . . . . . . . . :      0
     Allows the null value
     Coded Character Set Identifier  . . . . . :     37
```

Figure 14. Displaying LOB column information with the DSPFFD command

Note, that the Buffer Length is the space needed to store the LOB column pointer value in the buffer. DB2 UDB for AS/400 uses 32 bytes to store this pointer. However, since any AS/400 pointer must be aligned on a 16-byte boundary, the buffer for the HUSREF column is 43 bytes long. 11 bytes are needed to offset from the end of the CUSNUM column buffer to the next 16-byte boundary and 32 bytes for the pointer itself. The buffer for the HUSDES column is just 32 bytes long because it's already aligned.

2.9 LOB column considerations

This section describes the restrictions that are in place on LOB columns at this time:

- A single column can only be up to 15 MB (architecture to 2 GB).
- LOB column is not allowed in distributed tables.
- LOB column is not allowed as a key field in an index. LOBs are not allowed as key fields because of the size restriction.
- Distinct, Union, and Group By on LOB fields are not supported because each of these functions require the building of an index over the fields.
- LOB column is not allowed in IN predicate.
- LOB column is not allowed in COUNT(DISTINCT c1).
- LOB parameters are allowed in external stored procedures but not in SQL stored procedures.
- There is no support for REXX with LOBs.
- The following scalar functions are not supported at this time:

– LAND
– LNOT
– LOR
– XOR

- CHECK constraints are limited on tables with LOBs.
- REUSEDLT(*YES) must be used with tables with LOB columns.

2.9.1 Triggers

A file containing LOB fields cannot support triggers. The buffer images passed to trigger programs are currently unable to handle LOB data. Creating a new trigger parameter to make the LOB data available to the trigger program would present a non-SQL interface with pointers that directly address LOB data in the database file. By doing this, the database would lose control over the LOB data. Access to the LOB data should only be through the use of LOB locators.

The Add Physical File Trigger (`ADDPFTRG`) command has been changed so that it will not allow adding a trigger to a file that contains LOB fields.

2.9.2 Using in Net.Data

You can store large object files (LOBs) in DB2 databases and access them using the SQL language environment for your Web applications. The SQL language environment does not store large objects in Net.Data table processing variables (such as V1 or V2), or Net.Data table fields when a SQL query returns LOBs in a result set. Instead, when Net.Data encounters an LOB, it stores the LOB in a file that Net.Data creates. This file is in a directory specified by the HTML_PATH path configuration variable. The values of Net.Data table fields and table processing variables are set to the path of the file. Note that, in a busy Web environment, the number of files created on the fly by Net.Data may grow very rapidly and your application is responsible for cleaning up the directory on a regular basis. Therefore, we recommend that you use DataLinks, which eliminate the need to store files in directories by the SQL language environment, resulting in better performance and the use of much less system resources. Refer to *Net.Data Administration and Programming Guide for OS/400*, available for download from the Web at: `http://www.as400.ibm.com/products/netdata/docs/doc.htm`

Chapter 3. User-defined Distinct Types (UDTs)

This chapter describes:

- User-defined Distinct Types concepts and benefits
- Creating and using distinct types
- Casting of distinct types
- SQL support for distinct types
- Specifics on the AS/400 implementation of distinct types

3.1 A need for user-defined types

DB2 UDB for AS/400 provides a range of built-in data types, which include the basic data types, such as INTEGER, DECIMAL, CHAR, plus the large object data types discussed in the previous chapters of this book (BLOB, CLOB, DBCLOB). Your database design may, however, require that you use one of the built-in data types in a specialized way. You may use, for example, DECIMAL(11,2) data type to represent amounts of money. From the database semantic point of view, it makes sense to add and subtract two amounts of money, but it probably makes no sense to multiply two amounts of money.

DB2 UDB for AS/400 provides a way to declare such specialized usages of data types and the rules that go with them in a form of User-defined Distinct Type or UDT. Distinct data types are user-defined types that are derived from existing built-in types (predefined types, such as INTEGER, DECIMAL, CLOB). They share the same representation with the types they are derived from but, because they are incompatible types, they can have quite different semantics.

The most important characteristics of user-defined types include:

- **Strong typing**

  Strong typing insures that your UDTs behave appropriately. It guarantees that only user-defined functions defined on your UDT can be applied to instances of the UDT.

- **Casting**

  Casting from a distinct type to its source types and vice versa is allowed.

- **Performance**

  Distinct types are highly integrated into the database manager. Because distinct types are internally represented the same way as built-in data types, they share the same efficient code used to implement built-in functions, comparison operators, indexes, joins, and so forth, for built-in data types. Once the type check or conversion is completed, a join, for example, can be performed with no overhead.

- **Foundation for object-oriented extensions**

  UDTs are the foundation for most object-oriented features. They represent the most important step towards object-oriented extensions and future support for abstract or complex data types.

- **Compatible with the ANSI X3H2-99 standard (better known as SQL3)**

© Copyright IBM Corp. 2000

3.2 Creating distinct types

DB2 UDB for AS/400 supports User-defined Distinct Types only through the SQL interface. This section describes how to use Operations Navigator to:

- Create User-defined Distinct Types
- Create a table containing distinct type columns

We also repeat the procedure using the CREATE DISTINCT TYPE SQL statement.

3.2.1 Creating UDT sourced from DECIMAL

The Operations Navigator Library context menu can be used to create a new distinct type. Select the **New->Type** option from this menu to display the New Type dialog. This dialog allows you to set:

- Type name
- Description
- Source data type details

The source data type is the existing system data type that the distinct type is derived from.

To create a new distinct type using Operations Navigator, open the Libraries folder under the Database object. Select a library from the list, and right-click it to open its context menu. Select **New->Type** to display the New Type dialog.

Figure 15 shows the New Type dialog settings used to create a distinct type called MONEY with source data type of DECIMAL(11,2). It may be used to store and manipulate money values.

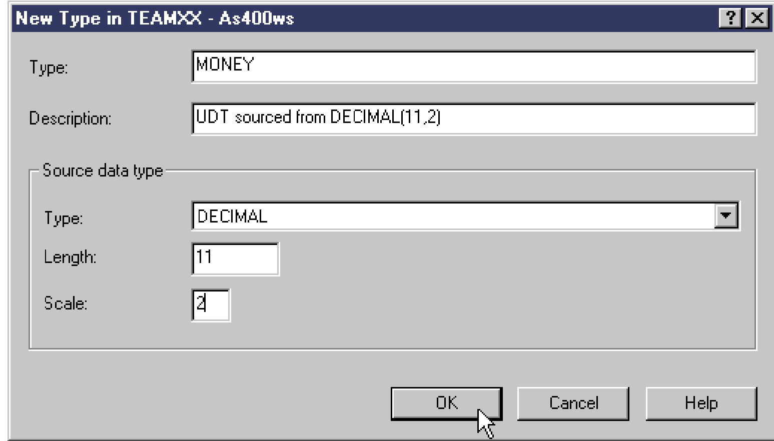


Figure 15. New type dialog for distinct type MONEY

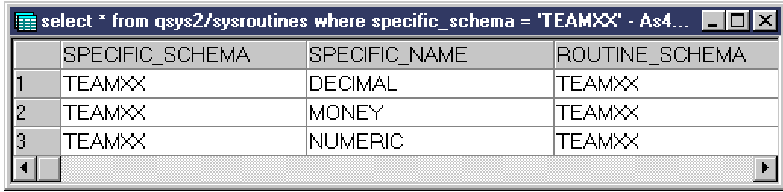
Once you have entered the required details, click the **OK** button to create the new distinct type in the selected library.

Along with the new type definition, the database manager also registers casting functions in the selected library. You can see these casting functions in the QSYS2/SYSROUTINES catalog and the casting function parameters in the QSYS2/SYSPARMS catalog.

Running the following SQL statement in the Operations Navigator Run SQL Script window displays the casting functions registered for the TEAMXX library:

```
select * from qsys2/sysroutines where specific_schema = 'TEAMXX';
```

Having previously created the MONEY distinct type source from DECIMAL(11,2), the results window for this SQL statement query is shown in Figure 16. It shows that three casting functions were created for the MONEY distinct type.



select * from qsys2/sysroutines where specific_schema = 'TEAMXX' - As4...

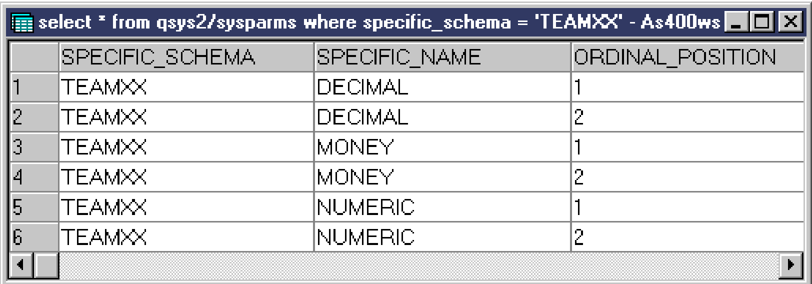
	SPECIFIC_SCHEMA	SPECIFIC_NAME	ROUTINE_SCHEMA
1	TEAMXX	DECIMAL	TEAMXX
2	TEAMXX	MONEY	TEAMXX
3	TEAMXX	NUMERIC	TEAMXX

Figure 16. Casting functions registered in QSYS2/SYSROUTINES

Running the following SQL statement in the Operations Navigator Run SQL Script window will display the casting parameters registered for the TEAMXX library:

```
select * from qsys2/sysparms where specific_schema = 'TEAMXX';
```

The results window, shown in Figure 17, shows that two parameters were registered for each casting function. For example, the DECIMAL casting function has a MONEY IN parameter and a DECIMAL OUT (return) parameter.



select * from qsys2/sysparms where specific_schema = 'TEAMXX' - As400ws

	SPECIFIC_SCHEMA	SPECIFIC_NAME	ORDINAL_POSITION
1	TEAMXX	DECIMAL	1
2	TEAMXX	DECIMAL	2
3	TEAMXX	MONEY	1
4	TEAMXX	MONEY	2
5	TEAMXX	NUMERIC	1
6	TEAMXX	NUMERIC	2

Figure 17. Casting function parameters registered in QSYS2/SYSPARMS

The database manager has automatically registered the following casting functions:

- DECIMAL(MONEY) returns DECIMAL
- MONEY(DECIMAL) returns MONEY
- NUMERIC(MONEY) returns NUMERIC

These functions allow you to convert from the underlying built-in source types into the new distinct type and vice versa.

3.2.2 Creating a table using UDTs

Having created a distinct type, it is possible to use the new type in table column definitions when creating or altering a table. To create a table using Operations Navigator, open the Libraries folder under the Database object. Select a library from the list, and right-click to open its context menu. Select **New->Table** to

display the New Table dialog. Figure 18 shows the New Table dialog settings for our example table using distinct types.

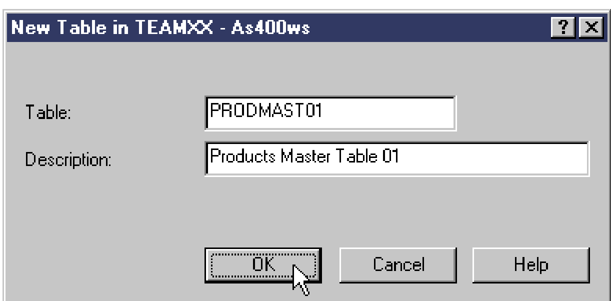


Figure 18. New table dialog

The purpose of our example table is to store information about the ski equipment offered in a Web warehouse. We store a GIF file with the equipment picture and a detailed, structured description of the product in the table rows. The aim is to make a Web application visually attractive and, at the same time, easy to use.

Once you enter the Table name and optional Description, click the **OK** button to begin column definition using the New Table dialog.

As shown in Figure 19, the column type drop-down list in the New Table dialog contains the available data types. The available data types include the new built-in types implemented in V4R4 (DATALINK, BLOB, CLOB, DBCLOB), other built-in types, and User-defined Distinct Types.

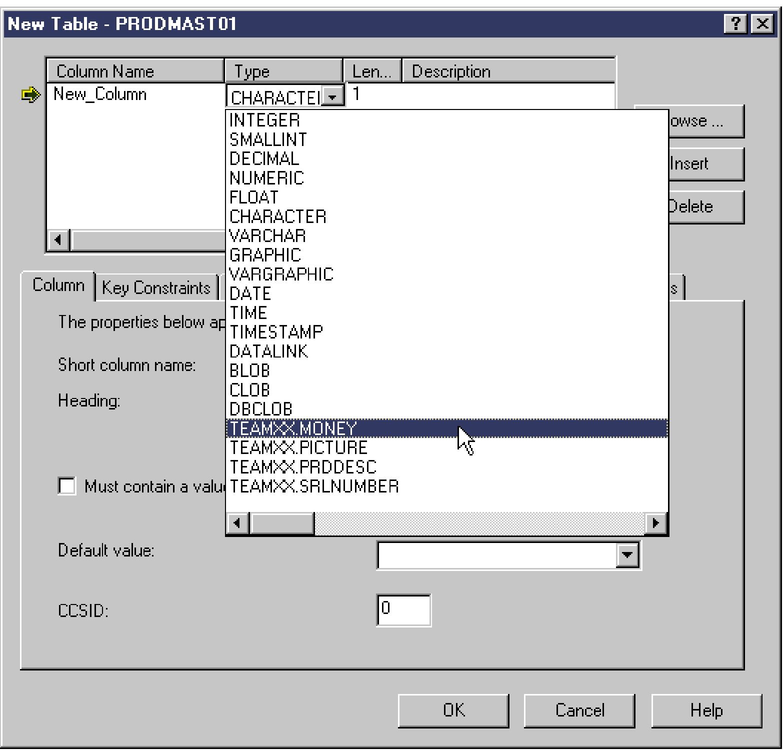


Figure 19. Column type list in a new table dialog

To add a new column, click the **Insert** button. Type in a column name and select the required distinct type from the drop-down list of available types as shown in Figure 19. You can also set other column options, such as Short column name or not null.

Table 3 shows the required column definitions of our example table.

Table 3. Products Master Table 01 properties

Name	Type	Size	Default value	Short column name	Must contain a value
Product_Number	TEAMXX. SRLNUMBER			PMNBR	Yes
Product_Name	CHARACTER	25		PMNAM	Yes
Product_Description	TEAMXX.PRDDESC			PMDESC	No
Product_Price	TEAMXX.MONEY			PMPRIC	Yes
Product_Picture	TEAMXX.PICTURE			PMPICT	No

Table 4 lists the distinct types needed in our table.

Table 4. Distinct type properties

Type name	Description	Source data type
PRDDESC	UDT sourced from CLOB(50 K)	CLOB(50 K)
MONEY	UDT sourced from DECIMAL(11,2)	DECIMAL(11,2)
PICTURE	UDT sourced from BLOB(1 M)	BLOB(1 M)
SRLNUMBER	UDT sourced from CHAR(5)	CHAR(5)

Refer to 3.2.3, “Creating distinct types with the SQL interface” on page 32, for the SQL DDL statements to create these distinct types.

Figure 20 on page 32 shows the completed table definition for our example table.

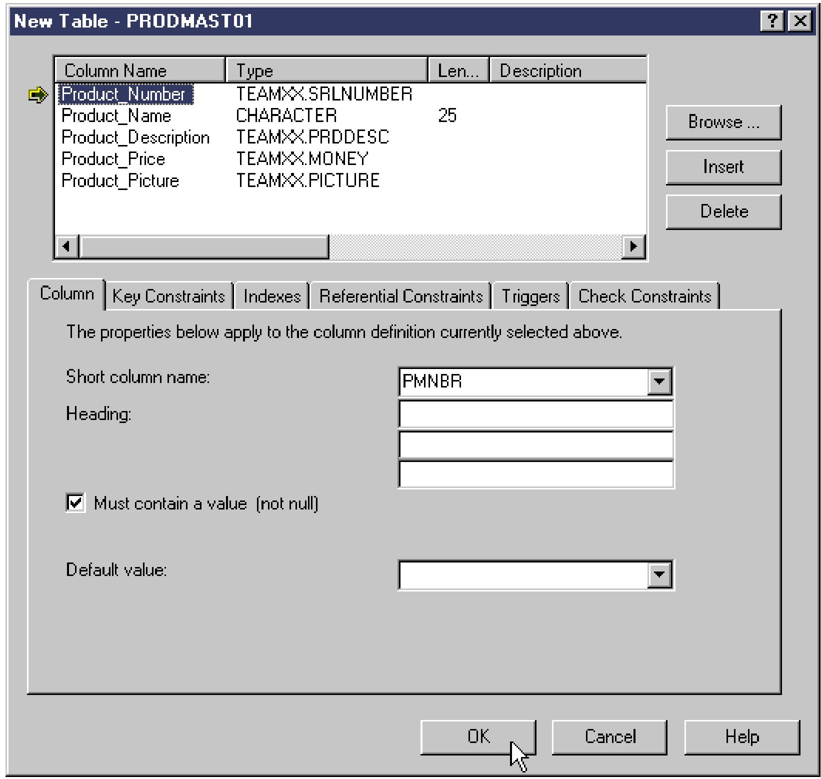


Figure 20. Products master table 01 properties

If required, constraints and triggers can also be defined for a table containing distinct types.

To create the table, click the **OK** button. The new table object appears in the selected library in the right panel of the main Operations Navigator window.

3.2.3 Creating distinct types with the SQL interface

The CREATE DISTINCT TYPE statement creates a distinct type sourced from one of the built-in data types:

```
CREATE [DISTINCT] TYPE distinct-type-name AS source-data-type
  [WITH COMPARISONS]
```

This statement allows you to:

- Set the distinct type name
- Select the source data type

Note that the `WITH COMPARISONS` option is the default, so that the system generates comparison operators whether it is specified. However, we recommend that you specify it for compatibility with other DB2 products.

You can add a description to the distinct type using the COMMENT ON DISTINCT TYPE statement.

To run an SQL statement using Operations Navigator, right-click on the **Database** object under your AS/400 Connection, and select **Run SQL Script**. After typing in the SQL statement, either place the cursor over the statement you want to run

and click the **Run Selected** icon, or click the **Run All** icon to run all SQL statements.

The following SQL statement creates a distinct type called money sourced from built-in type DECIMAL(11,2):

```
create distinct type teamxx/money as decimal(11,2)
  with comparisons;
```

Optionally, use the `COMMENT ON DISTINCT TYPE` SQL statement to add a description to the distinct type:

```
comment on distinct type teamxx/money is
  'UDT sourced from DECIMAL(11,2)';
```

The following SQL statements create a distinct type called prddesc sourced from built-in type CLOB(50K) and adds a description to the new type:

```
create distinct type teamxx/prddesc as clob(50k)
  with comparisons;
comment on distinct type teamxx/prddesc is
  'UDT sourced from CLOB(50K)';
```

The following SQL statements create other distinct types used in our test table:

```
CREATE DISTINCT  TYPE teamxx/srlnumber AS CHAR( 5 ) WITH COMPARISONS;
COMMENT ON DISTINCT TYPE teamxx/srlnumber IS 'UDT sourced from CHAR(5)';

CREATE DISTINCT TYPE teamxx/picture AS BLOB(1M) WITH COMPARISONS;
COMMENT ON DISTINCT TYPE teamxx/picture IS 'UDT sourced from BLOB(1M)';
```

The following SQL statement creates a table called prodmast01 with five columns:

```
create table teamxx/prodmast01(
  product_number for column pmnbr teamxx/srlnumber not null,
  product_name for column pmnam char(25) not null,
  product_description for column pmdesc teamxx/prddesc,
  product_price for column pmpric teamxx/money not null,
  product_picture for column pmpict teamxx/picture);
```

The table column definitions are in the following form:

```
column-name FOR COLUMN system-column-name data-type ...
```

The `data-type` can be a built-in data type specification or a distinct type name. If the distinct type is specified without a collection name, the distinct type name is resolved by searching the collections in the SQL path. We explicitly qualified the distinct type names with the TEAMXX library name in this CREATE TABLE example.

Optionally, use the `COMMENT ON TABLE` SQL statement to add a description to the table:

```
comment on table teamxx/prodmast01 is 'Products Master Table 01';
```

The above steps use the Operations Navigator Run SQL Scripts window. The SQL statements could equally be used in a 5250 Interactive SQL session. The only change needed is to leave out the terminating semicolon.

3.2.4 Altering and deleting distinct types

The database manager does not allow a distinct type to be deleted if it is being used by an existing table. The dependent table must be deleted first. This may be a problem if you want to alter a distinct type without having to delete the dependent tables.

If you need to alter a User-defined Distinct Type without having to delete all the dependent tables, follow these steps:

1. Use alter table to change the data type of the distinct type column to its source type.
2. Delete the distinct type.
3. Re-create the distinct type with its new attributes.
4. Use alter table to change the data type of the column back to the distinct type.

A practical example may involve the product_discription column in the PRODMAST01 table, which is a distinct type prddesc sourced from clob(50k). You may want to alter the prddesc distinct type to increase its CLOB size from 50 KB to 100 KB:

1. To alter the data type of the product_description column to the source data type of the prddesc distinct clob(50k), use the following SQL statement:

   ```
   alter table teamxx/prodmast01
     alter column product_description set data type clob(50k);
   ```

2. To drop the prddesc distinct type, use the following SQL statement:

   ```
   drop distinct type teamxx/prddesc;
   ```

3. To re-create the prddesc distinct type as clob(100k), use the following SQL statements:

   ```
   create distinct type teamxx/prddesc as clob(100k) with comparisons;
   comment on distinct type teamxx/prddesc is 'UDT sourced from CLOB(100K)';
   ```

4. To alter the product_description column data type back to distinct type prddesc, enter the following SQL statement:

   ```
   alter table teamxx/prodmast01
     alter column product_description set data type prddesc;
   ```

3.3 Casting for distinct types

The strong typing characteristic of distinct types means that distinct types are not compatible with other types. Compatibility with other types can be achieved by *casting*. Casting allows a value of one data type to be changed to another data type. When a data type can be changed to another data type, it is *castable* from the source data type to the target data type.

Casting functions are used to convert one data type to another. We saw in 3.2.1, “Creating UDT sourced from DECIMAL” on page 28, that the database manager automatically registers distinct type casting functions to allow casting between the distinct type and its source types.

Casting can occur implicitly or explicitly. It occurs *explicitly* when you use casting functions to cast a data type. Casting occurs *implicitly* when the database manager recognizes that an automatic cast is allowed in certain situations.

Strong typing requires that distinct types be explicitly cast when using:

- Built-in functions and operators
- Basic predicate comparisons, such as "=", "<", ">", involving different types
- Other predicate comparisons, such as BETWEEN, IN, LIKE

Implicit casting allows some distinct type assignments without exact type matching. Implicit casting allows:

- Castable constant and other type values to be assigned to distinct types
- Castable distinct type values to be assigned to other types
- Host variable assignments for non-SQL language access to distinct types

Promotion of data types allows the database manager to consider additional data types when performing implicit casts based on the precedence of data types.

3.3.1 Explicit casting

The strong typing characteristic of distinct types prevents comparisons between distinct types and other types. It is meaningless, for example, to compare APPLES with ORANGES.

The strong typing characteristic also prevents distinct types from being accepted as parameters to built-in functions or operators. The built-in functions and operators are not defined for distinct types. It may be pointless to take the square root of a MONEY value, for example.

Explicit casting can be used on the many occasions when you need to use built-in functions or operators with distinct types or compare distinct types with other types.

Table 5 provides a summary of the various SQL elements and their compatibility with distinct types. Incompatible SQL elements require that a distinct type be explicitly cast to its source type.

Table 5. UDT compatibility with various SQL elements

SQL element	Examples	Distinct type compatible	Comment
Basic predicates	"=", "<>", "<", ">"	Yes	Defined by the *with comparisons* default. Explicit cast only necessary if comparing a UDT with another type.
Other predicates	BETWEEN, IN, LIKE	Yes	Same as above.
Expressions	CASE	Yes	Same as above.
Special case scalar functions	NULLIF, COALESCE, VALUE	Yes	NULLIF, COALESCE, and VALUE were special cased to be allowed for UDTs.
Other scalar functions	LENGTH, MAX, MIN	No	Not defined for new type. Explicitly cast UDT or create a UDF sourced from built-in function.
Column functions	SUM, AVG, MIN, MAX, COUNT	No	Same as above.

SQL element	Examples	Distinct type compatible	Comment
Arithmetic operators	"*", "+", "-"	No	Same as above.

The following examples demonstrate how explicit casting can be used to:

- Use a built-in operator with a distinct type
- Compare a distinct type with another type
- Use a built-in function with a distinct type

3.3.1.1 Explicit casting UDT sourced from DECIMAL

This example demonstrates why explicit casting of User-defined Distinct Types is needed and how it can be used for UDT sourced from DECIMAL.

Say you want to add a 10 percent sales tax to the price of each product in a table named prodmast01. The product_price column in this table is distinct type money. Money is sourced from built-in type DECIMAL(11,2).

You could try multiplying the product_price column, which is distinct type money, by the decimal constant 1.1, as follows:

```
select product_price * 1.1 as "AFTER_TAX_PRICE"
  from teamxx/prodmast01;
```

Running this SQL statement in the Operations Navigator Run SQL Script window will fail with the following message in the run history:

```
SQL0402 - * use not valid.
```

It fails because the multiplication operator * does not accept arguments that are User-defined Distinct Types, such as the product_price in this case.

Explicit casting can be used to cast a User-defined Distinct Type to a data type that is accepted by the multiplication operator.

Use the decimal casting function to cast the product_price to decimal, a type that is accepted by the multiplication operator, as follows:

```
select decimal(product_price) * 1.1 as "AFTER_TAX_PRICE"
  from teamxx/prodmast01;
```

Running this SQL statement in the Operations Navigator Run SQL Script. window will successfully display a results window as shown in Figure 21.

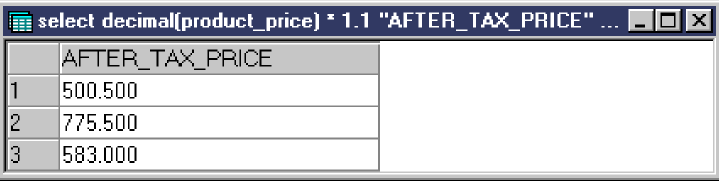


Figure 21. Results window for explicit cast from MONEY to DECIMAL

The multiplication operator can multiply two decimal values. We have explicitly cast the product_price money value to a decimal value using the decimal casting function, allowing multiplication with the decimal constant: `1.1`.

Notice how the value displayed in the AFTER_TAX_PRICE column in Figure 21 is not in the decimal(11,2) format of money. You could use explicit casting to cast the multiplication result back to money:

```
select money(decimal(product_price) * 1.1) as "AFTER_TAX_PRICE"
  from teamxx/prodmast01;
```

You could also use the built-in decimal function to set the required precision and scale of the decimal result:

```
select decimal(decimal(product_price) * 1.1,11,2) as "AFTER_TAX_PRICE"
  from teamxx/prodmast01;
```

You could also create a User Defined Function (UDF) to register a new '*' function that accepts money data type as input parameters. See Chapter 4, “User Defined Functions (UDFs)” on page 69, for details.

Explicit casting can also be used for comparisons between a User-defined Distinct Type and another type. For example, you may want to find the number of products with a price less than $500.00.

The SQL statement:

```
select product_price from teamxx/prodmast01
  where product_price < 500.00;
```

will fail with the following message in the run history:

```
SQL0401 - Comparison operator < operands not compatible.
```

The less than comparison operator cannot compare distinct type value with a DECIMAL constant: `500.00`. The database manager treats the constant 500.00 as DECIMAL data type.

Use the money casting function to cast the decimal constant to money as follows:

```
select product_price from teamxx/prodmast01
  where product_price < money(500.00);
```

Running this SQL statement in the Operations Navigator Run SQL Script window successfully displays a results window showing products with a price of < $500.00.

The less than comparison operator can compare two money values. We have explicitly cast the decimal constant to a money value using the money casting function, therefore allowing comparison with the product_price money value.

3.3.1.2 Explicit casting UDT sourced from CLOB

This example demonstrates how to use explicit casting to cast a distinct type sourced from CLOB to a data type that is accepted by the POSSTR built-in function.

For example, you may want to find all the ski products in a table named PRODMAST01 that have a description containing the word *moguls*. The PRODUCT_DESCRIPTION column in this table is distinct type PRDDESC. PRDDESC is sourced from built-in type CLOB(50K).

You could try passing the PRODUCT_DESCRIPTION column, which is distinct type PRDDESC, to the POSSTR function as follows:

```
select product_number, product_description from teamxx/prodmast01
  where posstr(product_description, 'moguls') <> 0;
```

This statement will fail with the following run history message:

```
SQL0171 - Argument 1 of function POSSTR not valid.
```

The PRODUCT_DESCRIPTION column is distinct type PRDDESC. The POSSTR built-in function does not accept arguments that are User-defined Distinct Types, such as the PRODUCT_DESCRIPTION in this case.

Explicit casting can be used to cast a User-defined Distinct Type to a data type that is accepted by the function.

Use the CLOB casting function to cast the PRODUCT_DESCRIPTION to CLOB, a type that is accepted by the POSSTR function, as follows:

```
select product_number, product_description from teamxx/prodmast01
  where posstr(clob(product_description), 'moguls') <> 0;
```

Running this SQL statement in the Operations Navigator Run SQL Script window will successfully display a results window showing the list of products with PRODUCT_DESCRIPTION containing the word *moguls*.

The POSSTR built-in function can search for a substring in a CLOB. By explicitly casting the PRODUCT_DESCRIPTION to CLOB using the CLOB casting function, POSSTR can be used on the PRODUCT_DESCRIPTION.

Alternatively, you can create a User Defined Function that accepts the PRDDESC distinct type as an argument. The UDF could be sourced from the built-in POSSTR function. See Chapter 4, “User Defined Functions (UDFs)” on page 69, for details.

3.3.2 Implicit casting

The database manager can perform implicit casting on assignments involving User-defined Distinct Types. Table 6 shows that a distinct data type is castable to its source data type. It also shows that the source data type of a distinct data type is castable to the distinct data type.

Table 6. Supported casts when a distinct type is involved

Data type ...	Is castable to data type ...
Distinct type DT	Source data type of distinct type DT.
Source data type of distinct type DT	Distinct type DT.
Distinct type DT	Distinct type DT.
Data type A	Distinct type DT, where A is promotable to the source data type of distinct type DT (see “Implicit casting and promotion” on page 40).
INTEGER	Distinct type DT if the DTs source type is SMALLINT.
DOUBLE	Distinct type DT if the DTs source type is REAL.
VARCHAR or VARGRAPHIC	Distinct type DT if the DTs source type is CHAR or GRAPHIC.

Implicit casting occurs *on assignments* where a distinct type is the source or the target of an assignment with its source data type. This allows you to make assignments between a distinct type and its source type without the need for explicit casting.

The following statement is an example on how implicit casting works when a distinct type is the target in an SQL assignment:

```
update teamxx/prodmast01 set product_price = 99.999
  where product_number = srlnumber('00001');
```

This statement assigns the DECIMAL constant 99.999 to PRODUCT_PRICE of the MONEY distinct type column. The source data type of distinct type MONEY is DECIMAL; so, DECIMAL is castable to MONEY. The database manager can, therefore, implicitly cast the DECIMAL constant on assignment to the MONEY PRODUCT_PRICE column.

Note that the price assigned to product 00001 will be truncated to 99.99 using the standard numeric assignments rules, that is, extra trailing digits in the fractional part of the number are eliminated.

The next statement is an example of how implicit casting works when a distinct type is the source in an SQL assignment:

```
update teamxx/prodmast01 set product_name = srlnumber('12345')
  where product_number = srlnumber('00001');
```

Implicit casting can also occur when assigning a distinct type to a compatible built-in type. The SRLNUMBER functions casts the CHAR constant '12345' to distinct type SRLNUMBER, which is sourced from built-in type CHAR(5).

The database manager performs an implicit cast when assigning a SRLNUMBER distinct type value to the PRODUCT_NAME column, which is built-in type CHAR.

Table 6 shows that Distinct type DT is castable to data type Source data type of distinct type DT. The source data type of distinct type SRLNUMBER is CHAR, so, SRLNUMBER is castable to CHAR.

We have seen that implicit casting can occur when a distinct type is either the source or the target in an assignment with a compatible built-in type using SQL.

Implicit casting can also occur for assignments involving host variables in embedded SQL.

Using the C programming language, we can declare a DECIMAL host variable named dec_price_in with an initial value of 88.88 as follows:

```
/* host variable declaration */
decimal(11,2) dec_price_in  = 88.88d;
```

Note that a C program must include the decimal.h header file to use the decimal type:

```
#include <decimal.h>
```

The following embedded SQL UPDATE statement assigns the dec_price_in host variable to the product_price column using an implicit cast from DECIMAL to distinct type MONEY:

```
/* implicit cast on assignment from decimal into money */
exec sql
  update prodmast01 set product_price = :dec_price_in
  where product_number = srlnumber('00001');
```

The dec_price_in host variable is implicitly cast to the target MONEY distinct type, to allow assignment to the PRODUCT_PRICE column.

A DECIMAL host variable named dec_price_out is declared as follows:

```
/* host variable declaration */
decimal(11,2) dec_price_out =  0.00d;
```

The next embedded SQL statement uses the MONEY distinct type PRODUCT_PRICE column as the source of an assignment to the dec_price_out host variable:

```
/* implicit cast on assignment from money into decimal */
exec sql
  select product_price into :dec_price_out from prodmast01
  where product_number = srlnumber('00001');
```

The SQL SELECT statement assigns a MONEY value from the PRODUCT_PRICE column to the dec_price_out host variable using an implicit cast.

See A.1, “UDTLABA: Using UDTs” on page 215, for the full source listing of the source fragments used in this section.

3.3.3 Implicit casting and promotion

Promotion of source types allows additional source types to be assigned to the target distinct type. Table 7 shows the precedence that the database manager uses to promote one data type to another.

As an example, Table 7 shows that the INTEGER data type can be promoted from INTEGER to DECIMAL or NUMERIC, to REAL, and to DOUBLE. This means that we can assign an INTEGER field to a MONEY distinct type field. The MONEY distinct type is sourced from DECIMAL, and INTEGER can be promoted to DECIMAL so that implicit casting can occur.

Table 7. Precedence of Data Types

Data type	Data type precedence list (in best-to-worst order)
CHAR or GRAPHIC	CHAR or GRAPHIC, VARCHAR or VARGRAPHIC, CLOB or DBCLOB
VARCHAR or VARGRAPHIC	VARCHAR or VARGRAPHIC, CLOB or DBCLOB
CLOB or DBCLOB	CLOB or DBCLOB
BLOB	BLOB
SMALLINT	SMALLINT, INTEGER, DECIMAL or NUMERIC, REAL, DOUBLE
INTEGER	INTEGER, DECIMAL or NUMERIC, REAL, DOUBLE
DECIMAL or NUMERIC	DECIMAL or NUMERIC, REAL, DOUBLE
REAL	REAL, DOUBLE

Data type	Data type precedence list (in best-to-worst order)
DOUBLE	DOUBLE
DATE	DATE
TIME	TIME
TIMESTAMP	TIMESTAMP
DATALINK	DATALINK
A distinct type	The same distinct type

Implicit casting with promotion will allow an INTEGER column to be assigned to a distinct type column sourced from DECIMAL.

Consider a simple table named TABLEA with an INTEGER column named INT_COL and one row of data created using the following SQL statements:

```
create table teamxx/tablea (int_col integer);
insert into teamxx/tablea values (12);
```

Implicit casting with promotion will allow a value from INT_COL to be assigned to a column of distinct type MONEY, where MONEY is sourced from DECIMAL(11,2).

The following SQL statement selects INT_COL from the TABLEA table and assigns it to PRODUCT_PRICE to update a row in the PRODMAST01 table:

```
update teamxx/prodmast01
  set product_price = (select int_col from teamxx/tablea)
  where product_number = srlnumber('00001');
```

The PRODUCT_PRICE column is distinct type MONEY sourced from DECIMAL(11,2). The INT_COL is built-in type INTEGER.

Table 7 shows that the INTEGER Data Type has DECIMAL second in its Data Type Precedence List. The INTEGER Data Type is, therefore, promotable to DECIMAL. The database manager performs this assignment by promoting INTEGER to DECIMAL and implicitly casting DECIMAL to MONEY.

Now, let’s look at the Data Type Precedence List for the DECIMAL Data Type in Table 7. Notice that DECIMAL can only be promoted to REAL or DOUBLE. Promotion will not occur if attempting to assign a DECIMAL value to an INTEGER value.

The promotion precedence order does not allow the reverse assignment used in our previous example. We cannot assign a MONEY distinct type column to an INTEGER column.

We may try selecting PRODUCT_PRICE from the PRODMAST01 table and assigning it to INT_COL to update a row in the TABLEA table as follows:

```
update teamxx/tablea
  set int_col = (
    select product_price from teamxx/prodmast01
    where product_number = srlnumber('00001'));
```

Then, the following error will be returned:

```
SQL0408 - Value for column or variable INT_COL not compatible.
```

INTEGER cannot be promoted to DECIMAL, so, the assignment fails. The precedence order of data types does not allow reverse promotion.

3.3.4 Implicit casting and host variables

Programming languages do not allow host variables to be declared with distinct types. Special rules extend implicit casting on assignment of distinct types to host variables. The source type of the distinct type must be assignable to the host variable for implicit casting to occur.

The following C-program fragment declares a long integer host variable named int_price_in with an initial value of 111:

```
/* host variable declaration */
long int_price_in = 111;

/* implicit cast on assignment from long integer into money */
exec sql
  insert into prodmast01 (product_number, product_name, product_price)
  values( '00004', 'New product', :int_price_in);
```

The embedded SQL INSERT statement inserts a row and assigns the int_price_in host variable to the PRODUCT_PRICE column. Again, the PRODUCT_PRICE column is distinct type MONEY, and MONEY is sourced from DECIMAL(11,2). The database manager performs this assignment using an implicit cast from an INTEGER type to DECIMAL.

The database manager will also allow the reverse assignment of a MONEY value to a INTEGER host variable.

In the next C-program fragment, a host variable array named product_rec is declared with an INTEGER field named int_price_out:

```
/* host variable declaration */
_Packed struct {
  char number[5];
  char name[25];
  long int_price_out;
} product_rec[10];
struct { short ind[3]; } product_ind[10];

/* declare and open fetch cursor */
exec sql
  declare c1 cursor for
  select product_number, product_name, product_price from prodmast01;
exec sql open c1;
/* implicit cast on assignment from money into long integer */
exec sql
  fetch c1 for 10 rows into :product_rec indicator :product_ind;
```

We then use embedded SQL to declare and open a cursor to be used in a multiple-row fetch statement.

The SQL FETCH statement will assign the PRODUCT_PRICE MONEY column to the int_price_out field in the product_rec host variable array using an implicit cast.

The FETCH statement sets the SQLERRD(3) field in the SQL Communication Area to the number of rows fetched. Include the SQLCA structure declaration in a C-program using the following statement:

```
exec sql include SQLCA;
```

The SQLERRD(3) field can then be accessed using the SQLCA structure member sqlca.sqlerrd[2]. The following `for` loop displays the records fetched:

```
for (i=0; i<sqlca.sqlerrd[2]; i++)
{
  printf("product_rec[%d].number        = %5.5s\n",
    i, product_rec[i].number);
  printf("product_rec[%d].name          = %25.25s\n",
    i, product_rec[i].name);
  printf("product_rec[%d].int_price_out = %d\n",
    i, product_rec[i].int_price_out);
}
```

The `for` loop code snippet shown above produces the following output listing for our test table:

```
values assigned to host variable array:

product_rec[0].number        = 00001
product_rec[0].name          = Atomic Betaflex 9.08
product_rec[0].int_price_out = 455
product_rec[1].number        = 00002
product_rec[1].name          = Atomic BetaCarvX 9.26
product_rec[1].int_price_out = 705
product_rec[2].number        = 00003
product_rec[2].name          = Tecnica Explosion SR
product_rec[2].int_price_out = 530
product_rec[3].number        = 00004
product_rec[3].name          = New product
product_rec[3].int_price_out = 111
```

The last record, product_rec[3], shows that PRODUCT_PRICE for the previously inserted row has been successfully assigned to product_rec[3].int_price_out.

We have seen that different casting rules apply when assigning distinct type values to host variables.

Assignment of an INTEGER host variable to a MONEY distinct type column behaves as explained previously for the implicit cast with promotion assignment. Implicit casting with promotion also allows an INTEGER value to be assigned to a MONEY value, but it does not allow the reverse cast.

However, implicit casting can be used on the reverse assignment of a MONEY distinct type column to an INTEGER host variable. Implicit casting on assignment to the host variable can occur because the source type of the distinct type is assignable to the host variable type.

See A.2, “UDTLABB: Casting UDTs” on page 216, for the full source listing of the source fragments used in this section.

3.4 SQL support for distinct types

In this section, we examine the SQL support provided for dealing with User-defined Distinct Types. The basic operations of SQL are *comparison* and *assignment*.

Comparison operations are performed when using predicates and other language elements, such as ORDER BY. As explained in 3.2.3, “Creating distinct types with the SQL interface” on page 32, the `CREATE DISTINCT TYPE` SQL statement `WITH COMPARISONS` default option means that the system will always generate comparison operators for distinct types. The SQL basic predicates, "=", "<>", "<", ">", "<=", ">=", can therefore be used to compare values of the same distinct type.

Basic predicates also allow distinct type columns to be used in SQL table joins and in SQL subqueries. Other predicates, such as BETWEEN and IN, cannot be used with distinct types directly, but casting functions allow use with some restrictions. The COALESCE (or VALUE), NULLIF scalar functions, and the CASE expression can be used to compare distinct type values.

Assignment operations are performed during the execution of statements, such as INSERT and UPDATE. In our look at casting on assignments in 3.3, “Casting for distinct types” on page 34, we found that the source and target data types in an assignment must be compatible. Specifying a DEFAULT value when creating a table is another example where SQL can assign a value to a distinct type column.

Strong typing prevents the use of functions defined for other types on distinct types. SQL distinct type operations can be extended by using User Defined Functions. See Chapter 4, “User Defined Functions (UDFs)” on page 69, for details.

3.4.1 Using predicates with UDT

The basic predicates: "=", "<>", "<", ">", "<=", ">=", allow comparison of two values. The BETWEEN predicate compares a value with a range of values. The IN predicate compares a value with a set of values. Other predicates include the LIKE predicate and the NULL predicate.

The values being compared by a basic predicate must be compatible. Strong typing means that distinct type values are not compatible with other types.

We may attempt to compare the PRODUCT_PRICE column, which is distinct type MONEY, with a NUMERIC constant using the following SQL statement:

```
select product_name, product_price from teamxx/prodmast01
  where product_price <> 530;
```

Then, the following error will be returned:

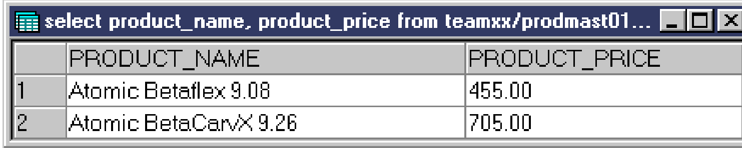
```
SQL0401 - Comparison operator ¬= operands not compatible.
```

The not equals operator **<>** cannot compare a MONEY value with a NUMERIC constant.

As shown in 3.3.1, “Explicit casting” on page 35, explicit casting can be used to compare distinct types with other data types. We can use the MONEY casting function to cast the NUMERIC constant to distinct type MONEY as follows:

```
select product_name, product_price from teamxx/prodmast01
  where product_price <> money(530);
```

If we run this statement using the Operations Navigator Run SQL Scripts window, the query results viewer successfully displays the rows with a PRODUCT_PRICE that does not equal $530.00 as shown in Figure 22.



select product_name, product_price from teamxx/prodmast01...

	PRODUCT_NAME	PRODUCT_PRICE
1	Atomic Betaflex 9.08	455.00
2	Atomic BetaCarvX 9.26	705.00

Figure 22. UDT not equal query results

As with basic predicates, the values being compared by other predicates must be compatible.

The following SQL statement attempts to use the IN predicate to compare a distinct type MONEY value with a set of NUMERIC constants:

```
select product_name, product_price from teamxx/prodmast01
  where product_price in (530, 705);
```

This statement will fail with the following message:

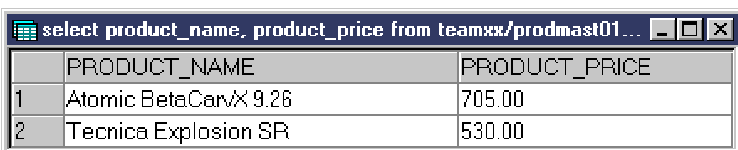
```
SQL0401 - Comparison operator IN operands not compatible.
```

The IN predicate cannot compare MONEY with NUMERIC constants.

The DECIMAL casting function can be used to cast the PRODUCT_PRICE column to DECIMAL to allow comparison with a set of NUMERIC constants as follows:

```
select product_name, product_price from teamxx/prodmast01
  where decimal(product_price) in (530, 705);
```

If we run this statement using the Operations Navigator Run SQL Scripts window, the query results viewer successfully displays the rows with PRODUCT_PRICE in 530.00 and 705.00 as shown in Figure 23.



select product_name, product_price from teamxx/prodmast01...

	PRODUCT_NAME	PRODUCT_PRICE
1	Atomic BetaCarvX 9.26	705.00
2	Tecnica Explosion SR	530.00

Figure 23. UDT IN query results

Note

The system-generated distinct type comparison operator behavior is the same as for the source data type. Comparison operator behavior for distinct types cannot be customized.

3.4.2 Joining on UDT

User-defined Distinct Type columns can be used to join tables in an SQL statement.

For example, we may have a table called PRODMAST01 containing an inventory list of products with the columns listed in Table 3 on page 31. Another table, called ORDERDTL, contains a row for each product listed on a customer order with the columns listed in Table 8.

Table 8. Order Detail table properties

Name	Type	Size	Default value	Short column name	Must contain a value
Order_Number	TEAMXX. SRLNUMBER			ODONBR	Yes
Product_Number	TEAMXX. SRLNUMBER			ODPNBR	Yes
Orderdtl_Quantity	DECIMAL()	5,0		ODOQTY	No
Orderdtl_Item_Cost	TEAMXX.MONEY			ODDCST	Yes
Order_Abstract	CLOB()	50 K		ODABS	Yes

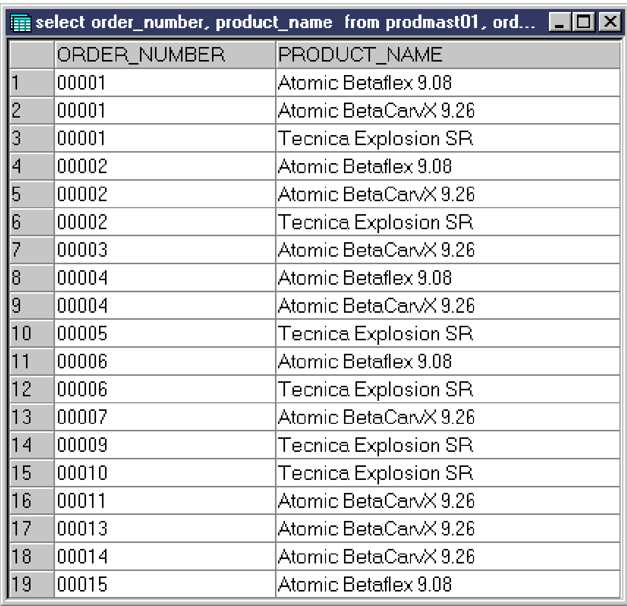
Both the PRODMAST01 and ORDERDTL tables have a PRODUCT_NUMBER column with data type of distinct type SRLNUMBER.

We can list the ORDER_NUMBER from the ORDERDTL table with the PRODUCT_NAME from the PRODMAST01 table using the following SQL statement:

```
select order_number, product_name
  from prodmast01, orderdtl
  where prodmast01.product_number = orderdtl.product_number;
```

This statement performs a default inner join on the PRODMAST01 table and the ORDERDTL table with join condition, `prodmast01.product_number = orderdtl.product_number`. The join columns are both distinct type SRLNUMBER sourced from CHAR(5) so that the database manager can perform the join comparison.

If we run this statement using the Operations Navigator Run SQL Scripts window, the query results viewer successfully displays the required list of ORDER_NUMBER versus PRODUCT_NAME as shown in Figure 24.



select order_number, product_name from prodmast01, ord...

	ORDER_NUMBER	PRODUCT_NAME
1	00001	Atomic Betaflex 9.08
2	00001	Atomic BetaCarvX 9.26
3	00001	Tecnica Explosion SR
4	00002	Atomic Betaflex 9.08
5	00002	Atomic BetaCarvX 9.26
6	00002	Tecnica Explosion SR
7	00003	Atomic BetaCarvX 9.26
8	00004	Atomic Betaflex 9.08
9	00004	Atomic BetaCarvX 9.26
10	00005	Tecnica Explosion SR
11	00006	Atomic Betaflex 9.08
12	00006	Tecnica Explosion SR
13	00007	Atomic BetaCarvX 9.26
14	00009	Tecnica Explosion SR
15	00010	Tecnica Explosion SR
16	00011	Atomic BetaCarvX 9.26
17	00013	Atomic BetaCarvX 9.26
18	00014	Atomic BetaCarvX 9.26
19	00015	Atomic Betaflex 9.08

Figure 24. UDT JOIN query results

Note

Use casting functions when joining tables on columns that are not of the same distinct type.

3.4.3 Using a default value with UDT

As with built-in data type columns, a default value can be specified for User-defined Distinct Type columns.

A default value can be specified using the Table Properties window in Operations Navigator. To display the Table Properties window, open the Libraries folder under the Database object. Click on the required library to display its contents. Right-click the required table, and select **Properties**.

Figure 25 on page 48 shows the Table Properties window for the table named PRODMAST01. The default value field shows that we have selected the PRODUCT_PRICE column and set its default value to:

```
MONEY(99.99)
```

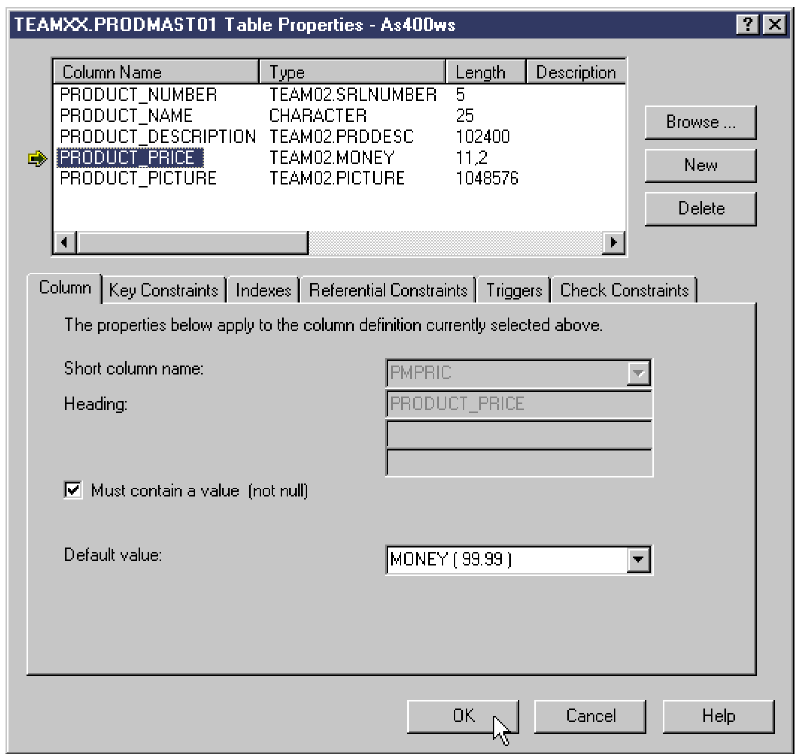


Figure 25. PRODMAST01 table properties with the UDT column default value

Click the **OK** button to update the table properties. The column default value is used when adding a new row without specifying a value for the column.

You can also set the same default value using the following SQL ALTER TABLE statement:

```
alter table teamxx/prodmast01 alter column product_price
  set default money(99.99);
```

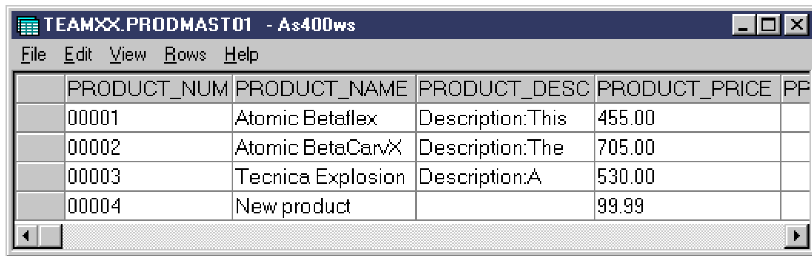
If you need to create a new table using a distinct type column default value, the following example applies:

```
create table teamxx/prodmast01(
  product_number for column pmnbr teamxx/srlnumber not null,
  product_name for column pmnam char(25) not null,
  product_description for column pmdesc teamxx/prddesc,
  product_price for column pmpric teamxx/money not null
    with default money(99.99),
  product_picture for column pmpict teamxx/picture
);
comment on table teamxx/prodmast01 is 'Products Master Table 01';
```

The column default value will be used when inserting a new row without specifying a value for the column:

```
insert into teamxx/prodmast01 (product_number,product_name)
  values('00004','New product');
```

The Operations Navigator Quick View window should show that the PRODUCT_PRICE column in the new row has been assigned the default value of 99.99 as shown in Figure 26.



PRODUCT_NUM	PRODUCT_NAME	PRODUCT_DESC	PRODUCT_PRICE	PF
00001	Atomic Betaflex	Description:This	455.00	
00002	Atomic BetaCarvX	Description:The	705.00	
00003	Tecnica Explosion	Description:A	530.00	
00004	New product		99.99	

Figure 26. UDT column set using default value

3.5 DB2 UDB for AS/400 implementation

In this section, we examine specifics related to the DB2 UDB for AS/400 implementation of User-defined Distinct Types. These specifics include distinct type access limitations using native I/O, keeping track of distinct types, and database recovery.

Data Description Specification (DDS) does *not* support distinct types, so native I/O access to distinct type fields is not possible. The preferred alternative is to use embedded SQL to handle distinct type columns. Otherwise, restricted native I/O access to tables containing distinct type columns is possible using logical files or SQL views.

A logical file or SQL view that excludes distinct type columns can provide read and write access to the non-distinct type columns in a table. An SQL view can also cast distinct type fields to their source type to provide *read only* access to a distinct type column.

The AS/400 implementation of distinct types provides a number of data dictionary facilities that can be used to keep track of your distinct types. The SYSTYPES catalog contains information on all distinct types in the database. The SYSCOLUMNS catalog also contains information on all columns in the database.

The AS/400 implementation creates distinct types as *SQLUDT objects. The basic Work with Objects options are available using a 5250 session. The DSPFFD CL command can also be used to display File Field Descriptions providing details, such as column definitions and distinct type source types.

Dependencies exist between distinct type objects and other objects that use the distinct type, such as tables or user-defined functions. These dependencies have implications for the sequence in which you save and restore objects that use distinct types. For example, the implications of restoring a table before restoring a distinct type that it uses need to be considered.

The AS/400 implementation does not log distinct types when journaling. Only the source type is logged. This is consistent with the database manager using the same internal representation for a distinct type and its source type.

3.5.1 Native system interfaces

As stated in the introduction of this section, Data Description Specification (DDS) does not support distinct types, so native I/O access to distinct type fields is not possible. While the preferred alternative is to use embedded SQL to handle

distinct type columns, restricted native I/O access is possible using logical files or SQL views.

3.5.1.1 Creating a logical file from a table containing UDT

A C program attempting to access a table (physical file) containing distinct type columns using native I/O is shown in the following source listing. The numbered lines are explained in the notes that follow the listing.

```
#include <stdio.h>
#include <recio.h>
#include <decimal.h>

#pragma mapinc("prodmast01","PRODMAST01(*ALL)","both","d z _P","")

1 #include "prodmast01"

static char* FILE_NAME = "PRODMAST01";


int main(int argc, char** argv)
{

2 TEAMXX_PRODMAST01_PRODMAST01_both_t buf;

3 _RFILE   *fp;
  _RIOFB_T *fb;

  printf("\n");
  printf("AS/400 DB2 UDB UDT Lab Test Program: %s\n", argv[0]);
  printf("\n");

4 if ((fp = _Ropen(FILE_NAME, "rr")) == NULL)
  {
    perror("File open error");
    return 0;
  }

  printf("Read all records: %s\n", FILE_NAME);
  printf("\n");
  printf("%-25.25s\n", "PRODUCT_NAME");

5 for (
    fb = _Readf(fp, (void *)&buf, sizeof(buf), __NO_LOCK);
    fb->num_bytes == sizeof(buf);
    fb = _Readn(fp, (void *)&buf, sizeof(buf), __NO_LOCK))
  {
    printf("%-25.25s\n", buf.PRODUCT_NAME);
  }

6 _Rclose(fp);

  return -1;

}
```

C program notes

1. Include the typedefs generated from the external AS/400 file descriptions (DDS) based on the preceding `mapinc` pragma.
2. Declare record file pointers needed to work with native files. These are defined in recio.h.
3. Declare a record buffer using the typedef generated from the DDS. You can compile your ILE C program with OUTPUT(*PRINT) OPTION(*SHOWUSR) to see the typedefs in your compiler listing.
4. Open existing native file for reading records, setting `fp` to the return record file pointer.

5. The `for` loop initializes by reading the first record from the file. It checks that the record buffer has been successfully read before each iteration, and it reads the next record at the end of each iteration,
6. Close the native file on exit.

Create the bound C program using the following CL command:

```
CRTBNDC PGM(TEAMXX/UDTLABC) SRCMBR(UDTLABC)
```

The UDTLABC test program attempts to open the PRODMAST01 table for read using native I/O. This table contains a number of distinct type fields. If we call the program using: `CALL TEAMXX/UDTLABC`, the program terminates with a `File open error` as shown in Figure 27.

```
AS/400 DB2 UDB UDT Lab Test Program: TEAMXX/UDTLABC

File open error: A non-recoverable I/O error occurred.
Press ENTER to end terminal session.
```

Figure 27. UDTLABC test program results

Displaying the job log using the `DSPJOBLOG` command, we find the following error message under the CALL TEAMXX/UDTLABC job log entry as shown in Figure 28.

```
4 > CALL TEAMXX/UDTLABC
    Open of member PRODMAST01 file PRODMAST01 in TEAMXX failed.
```

Figure 28. UDTLABC job log error message

The Additional Message Information screen for this message is shown in Figure 29 on page 52.

Cause 3 applies in this case. The native I/O interface is not able to process user defined data type fields. The recovery information suggests that embedded SQL be used.

```
                          Additional Message Information

 Message ID . . . . . . :   CPF428A       Severity . . . . . . . :   40
 Message type . . . . . :   Escape

 Message . . . . :   Open of member PRODMAST01 file PRODMAST01 in TEAMXX
   failed.
 Cause . . . . . :   Member PRODMAST01 file PRODMAST01 in library TEAMXX was
   not opened because of error code 3. The error codes and their meanings are:
      1 -- The format for file PRODMAST01 contains one or more large object
   fields and the open request did not indicate that large object fields could
   be processed by the user of the open.
      2 -- The format for file PRODMAST01 contains one or more data link fields
   and the open request did not indicate that data link fields could be
   processed by the user of the open.
      3 -- The format for file PRODMAST01 contains one or more user defined
   data type fields and the open request did not indicate that user defined
   data type fields could be processed by the user of the open.
      4 -- A user-defined type for a field for the file does not exist.
 Recovery  . . . :   Either specify a different file, use the DSPFFD command to
   determine what user-defined type is missing, change the open request to
   indicate that the specified field type can be processed, or change the
   program to use embedded SQL to process the file. Then try your request
   again. These field types are fully supported only through SQL. Therefore, if
   you do not have the DB2 Query Manager and SQL Development Tool Kit for
   AS/400 product, your program may not be able to access file PRODMAST01.
```

Figure 29. UDTLABC job log additional message information

We now examine another alternative using native I/O. It is possible to use logical files or SQL views to gain limited native I/O access to tables with distinct type columns. We can create a logical file from a table that contains distinct types. We can create it without including the source table distinct types fields.

Figure 30 shows the Data Description Specification source to create a logical file named UDTLFA that only defines the PMNAM field from our PRODMAST01 table. Table 3 on page 31 shows that this is the PRODUCT_NAME field, which is built-in type CHARACTER.

```
 Browse : TEAMXX/QDDSSRC(UDTLFA)
 Record . :       1 of      2 by  15           Column:   13 of   92 by  79
 Control  :

..+....2....+....3....+....4....+....5....+....6....+....7....+....8....+....9.
 ************Beginning of data**************
               R PRODMAST01                PFILE(TEAMXX/PRODMAST01)
                 PMNAM
  ************End of Data********************
```

Figure 30. UDTLFA display file screen

Create the logical file from the DDS using the following command:

```
CRTLF FILE(TEAMXX/UDTLFA) SRCFILE(TEAMXX/QDDSSRC)
```

We need to change the UDTLABC program to access the UDTLFA view rather than the PRODMAST01 table. See the following program listing of the updated program UDTLABD. The numbered lines are explained in the note that follows.

```
#include <stdio.h>
#include <recio.h>
#include <decimal.h>

#pragma mapinc("udtlfa", "UDTLFA(*ALL)", "both", "d z _P","")

1 #include "udtlfa"

static char* FILE_NAME = "UDTLFA";


int main(int argc, char** argv)
{

2 TEAMXX_UDTLFA_PRODMAST01_both_t buf;

  _RFILE   *fp;
  _RIOFB_T *fb;

  printf("\n");
  printf("AS/400 DB2 UDB UDT Lab Test Program: %s\n", argv[0]);
  printf("\n");

  if ((fp = _Ropen(FILE_NAME, "rr")) == NULL)
  {
    perror("File open error");
    return 0;
  }

  printf("Read all records: %s\n", FILE_NAME);
  printf("\n");
  printf("%-25.25s\n", "PRODUCT_NAME");

  for (
    fb = _Readf(fp, (void *)&buf, sizeof(buf), __NO_LOCK);
    fb->num_bytes == sizeof(buf);
    fb = _Rreadn(fp, (void *)&buf, sizeof(buf), __NO_LOCK))
  {
    printf("%-25.25s\n", buf.PRODUCT_NAME);
  }

  _Rclose(fp);

  return -1;

}
```

UDTLABD program notes

1. Changed `mapinc` pragma and include to use UDTLFA logical file.
2. Changed record buffer declaration to use the new typedef generated from the UDTLFA DDS.

Create the bound C program using the following CL command:

```
CRTBNDC PGM(TEAMXX/UDTLABD) SRCMBR(UDTLABD)
```

The UDTLABD test program attempts to open the UDTLFA logical file for read using native I/O. This logical file excludes the distinct type fields in the PRODMAST01 table. If we call the program using: `CALL TEAMXX/UDTLABD`, the program terminates having read all records successfully as shown in Figure 31 on page 54.

```
AS/400 DB2 UDB UDT Lab Test Program: TEAMXX/UDTLABD

Read all records: TEAMXX/UDTLFA

PRODUCT_NAME
Atomic Betaflex 9.08
Atomic BetaCarvX 9.26
Tecnica Explosion SR
Press ENTER to end terminal session.
```

Figure 31. UDTLABD native I/O read results

We have demonstrated that native read can be performed on a table containing distinct types if we create a logical file excluding the distinct types.

3.5.1.2 Creating an SQL view from a table containing UDT

We have seen that logical files can be used to provide native I/O access to non-distinct type fields in tables using distinct types. SQL views can be used to extend native I/O access to enable read-only access to distinct type columns.

The SQL interface provides access to distinct type casting functions. These casting functions can be used to cast a distinct type column in a table to a source type in a view. View columns are read-only if a column function is used. The cast distinct type field will, therefore, be read-only.

The following SQL statement creates a view named UDTLFB:

```
create view teamxx/udtlfb (product_name, product_dec_price)
  as select product_name, decimal(product_price)
  from teamxx/prodmast01;
```

The statement selects the PRODUCT_NAME column and DECIMAL of the PRODUCT_PRICE column from our PRODMAST01 table.

The following source listing of a C program attempts to access this SQL view containing distinct type column cast to its source type using native I/O. The numbered lines are explained in the note that follows.

```
#include <stdio.h>
#include <decimal.h>
#include <recio.h>

#pragma mapinc("udtlfb", "UDTLFB(*ALL)", "both", "d z _P","")

1 #include "udtlfb"

static char* FILE_NAME = "UDTLFB";


int main(int argc, char** argv)
{

  TEAMXX_UDTLFB_UDTLFB_both_t buf;

  _RFILE   *fp;
  _RIOFB_T *fb;

  printf("\n");
  printf("AS/400 DB2 UDB UDT Lab Test Program: %s\n", argv[0]);
  printf("\n");

2 if ((fp = _Ropen(FILE_NAME, "rr+")) == NULL)
```

```
  {
    perror("File open error");
    return 0;
  }

3 sprintf(buf.PRODUCT_NAME, "%-25.25s", "New name");
  buf.PRODUCT_DEC_PRICE = 0;

  printf("Update first record: %s\n", FILE_NAME);
  printf("\n");
  printf("%-25.25s %-17.17s\n", "PRODUCT_NAME", "PRODUCT_DEC_PRICE");
  printf("%-25.25s ",    buf.PRODUCT_NAME);
  printf("%17D(11,2)\n", buf.PRODUCT_DEC_PRICE);
  printf("\n");

4 _Rlocate(fp, NULL, 0, __FIRST);
5 fb = _Rupdate(fp, (void *)&buf, sizeof(buf));
6 if (fb->num_bytes != sizeof(buf))
  {
    perror("File update error");
  }

  printf("Read all records: %s\n", FILE_NAME);
  printf("\n");
  printf("%-25.25s %-17.17s\n", "PRODUCT_NAME", "PRODUCT_DEC_PRICE");

7 for (
    fb = _Rreadf(fp, (void *)&buf, sizeof(buf), __NO_LOCK);
    fb->num_bytes == sizeof(buf);
    fb = _Rreadn(fp, (void *)&buf, sizeof(buf), __NO_LOCK))
  {
    printf("%-25.25s ",    buf.PRODUCT_NAME);
    printf("%17D(11,2)\n", buf.PRODUCT_DEC_PRICE);
  }

  _Rclose(fp);

  return -1;

}
```

C program notes

1. `mapinc` pragma and include set to use UDTLFB logical file.
2. Open existing native file for reading, writing, or updating records.
3. Initialize the record buffer with the record update data.
4. Locate the first record.
5. Update the first record with the data in the record buffer.
6. Check that the record was successfully updated.
7. Read all records.

Create the bound C program using the following CL command:

```
CRTBNDC PGM(TEAMXX/UDTLABE) SRCMBR(UDTLABE)
```

The UDTLABE test program attempts to open the UDTLFB SQL view for read and update using native I/O. This logical file excludes the distinct type fields in the PRODMAST01 table. If we call the program using: `CALL TEAMXX/UDTLABE`, the program terminates having updated the first record and then reads all records successfully as shown in Figure 32 on page 56.

```
AS/400 DB2 UDB UDT Lab Test Program: TEAMXX/UDTLABE

Update first record: TEAMXX/UDTLFB

PRODUCT_NAME              PRODUCT_DEC_PRICE
New name                               0.00

Read all records: TEAMXX/UDTLFB

PRODUCT_NAME              PRODUCT_DEC_PRICE
New name                             455.00
Atomic BetaCarvX 9.26                705.00
Tecnica Explosion SR                 530.00
Press ENTER to end terminal session.
```

Figure 32. UDTLABE native I/O results

Looking closely at these results, we can see that the first record was updated with PRODUCT_NAME set to `New name`, and PRODUCT_DEC_PRICE set to `0.00`.

When reading all records after the write, we can see that the PRODUCT_NAME field was successfully updated, but the PRODUCT_DEC_PRICE was not.

As shown in Figure 33, the `DSPJOBLOG` command shows no error or warning messages under the `CALL TEAMXX/UDTLABE` job log entry.

```
4 > CALL TEAMXX/UDTLABE
```

Figure 33. UDTLABE job log entry with no error messages

Important

As we have seen, the native I/O interface does not flag an error when attempting to update a distinct type field using a view that casts the distinct type column to its source type.

The following SQL statement updates the same column using the SQL interface:

```
update teamxx/udtlfb set product_dec_price = 0
  where product_name = 'New name';
```

This statement will fail with the following message:

```
SQL0151 - Column PRODUCT_DEC_PRICE in table UDTLFB in TEAMXX read-only.
```

The job log provides further details on SQL execution errors.

In the Operations Navigator Run SQL Scripts window, select **View->Job Log...**. Then double-click the Message ID of interest, **SQL0151**, to display the Detailed Message Information window as shown in Figure 34.

In this case, the detailed message information indicates that the PRODUCT_DEC_PRICE column is read only because it is derived from an

expression. The recovery advice suggests removing the PRODUCT_DEC_PRICE column from the column list.

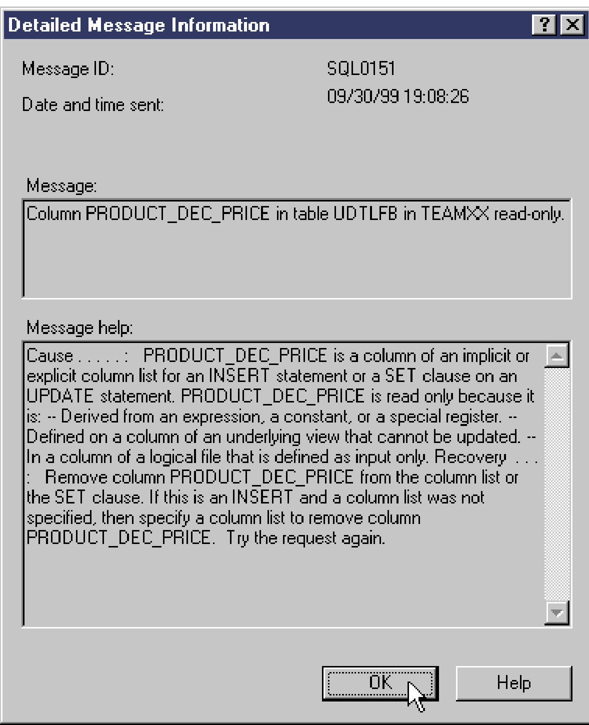


Figure 34. Column read-only error

3.5.2 Keeping track of distinct types

The database manager provides a number of data dictionary facilities that can be used to keep track of User-defined Distinct Types. In this section, we see how to view UDT information using the SYSTYPES catalog, the SYSCOLUMNS catalog, the *SQLUDT object, and the DSPFFD CL command.

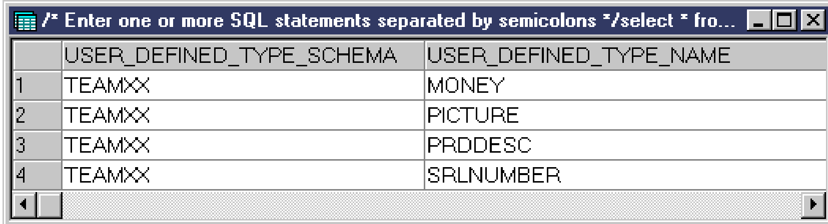
3.5.2.1 SYSTYPES catalog

Distinct types (and built-in types) are stored in the SYSTYPES catalog. Refer to *DB2 UDB for AS/400 SQL Reference*, SC41-5612, for the detailed description of the catalog views.

The following SQL statement displays SYSTYPES information on User-defined Distinct Types in the TEAMXX library:

```
select * from systypes where user_defined_type_schema = 'TEAMXX';
```

If we run this statement using the Operations Navigator Run SQL Scripts window, with the distinct types listed in Table 4 on page 31 in the TEAMXX library, the query results viewer displays distinct type details as shown in Figure 35 on page 58.



/* Enter one or more SQL statements separated by semicolons */select * fro...

	USER_DEFINED_TYPE_SCHEMA	USER_DEFINED_TYPE_NAME
1	TEAMXX	MONEY
2	TEAMXX	PICTURE
3	TEAMXX	PRDDESC
4	TEAMXX	SRLNUMBER

Figure 35. SYSTYPES catalog

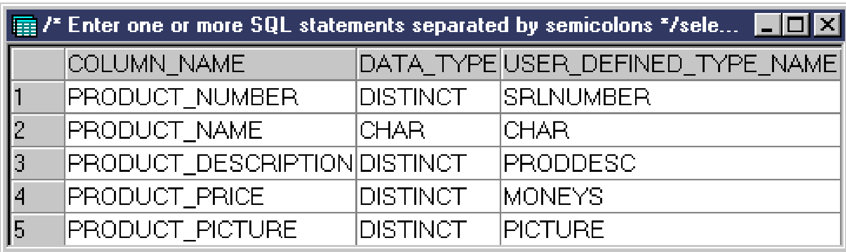
3.5.2.2 SYSCOLUMNS catalog

Column details are stored in the SYSCOLUMNS catalog. Refer to *DB2 UDB for AS/400 SQL Reference*, SC41-5612, for the detailed description of the catalog views.

The following SQL statement displays SYSCOLUMNS information on the PRODMAST01 table in the TEAMXX library:

```
select column_name, data_type, user_defined_type_name from syscolumns
  where table_name = 'PRODMAST01' and table_schema = 'TEAMXX';
```

If we run this statement using the Operations Navigator Run SQL Scripts window, with the PRODMAST01 table defined, as shown in Table 3 on page 31, the query results viewer displays PRODMAST01 column details as shown in Figure 36.



/* Enter one or more SQL statements separated by semicolons */sele...

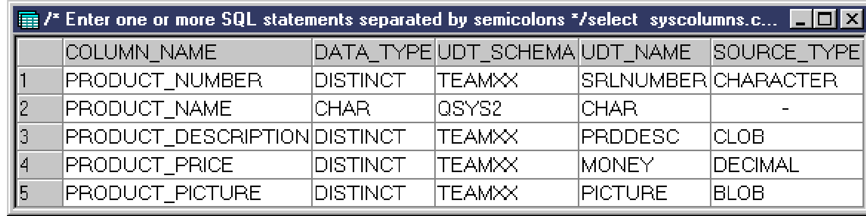
	COLUMN_NAME	DATA_TYPE	USER_DEFINED_TYPE_NAME
1	PRODUCT_NUMBER	DISTINCT	SRLNUMBER
2	PRODUCT_NAME	CHAR	CHAR
3	PRODUCT_DESCRIPTION	DISTINCT	PRODDESC
4	PRODUCT_PRICE	DISTINCT	MONEYS
5	PRODUCT_PICTURE	DISTINCT	PICTURE

Figure 36. SYSCOLUMNS catalog

We can join the SYSCOLUMNS table to the SYSTYPES table to find the source data type as follows:

```
select
  syscolumns.column_name,
  syscolumns.data_type,
  syscolumns.user_defined_type_schema as "UDT_SCHEMA",
  syscolumns.user_defined_type_name as "UDT_NAME",
  systypes.source_type
from
  syscolumns left join systypes on
    syscolumns.user_defined_type_schema =
      systypes.user_defined_type_schema and
    syscolumns.user_defined_type_name =
      systypes.user_defined_type_name
where
  syscolumns.table_name = 'PRODMAST01' and
  syscolumns.table_schema = 'TEAMXX';
```

If we run this statement using the Operations Navigator Run SQL Scripts window, with the PRODMAST01 table defined, as shown in Table 3 on page 31, the query results viewer displays PRODMAST01 column details and source type as shown in Figure 37.



/* Enter one or more SQL statements separated by semicolons */select syscolumns.c...

	COLUMN_NAME	DATA_TYPE	UDT_SCHEMA	UDT_NAME	SOURCE_TYPE
1	PRODUCT_NUMBER	DISTINCT	TEAMXX	SRLNUMBER	CHARACTER
2	PRODUCT_NAME	CHAR	QSYS2	CHAR	-
3	PRODUCT_DESCRIPTION	DISTINCT	TEAMXX	PRDDESC	CLOB
4	PRODUCT_PRICE	DISTINCT	TEAMXX	MONEY	DECIMAL
5	PRODUCT_PICTURE	DISTINCT	TEAMXX	PICTURE	BLOB

Figure 37. SYSCOLUMNS catalog with SYSTYPES.SOURCE_TYPE

3.5.2.3 The *SQLUDT object

The *SQLUDT object type contains all of the information for a distinct type. There is one *SQLUDT object for each distinct type in the system.

To view *SQLUDT objects using Operations Navigator, click the required library object in the Libraries folders. Objects in the library are displayed in the right panel as shown in Figure 38. This view contains a list of the distinct types in the TEAMXX library.

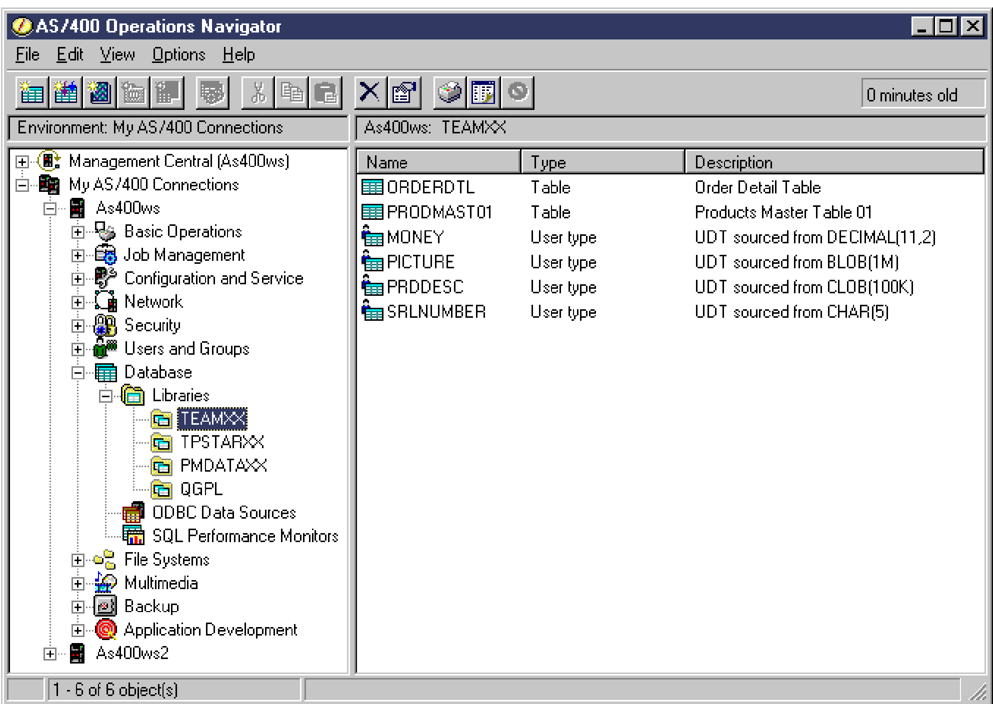


Figure 38. Operations Navigator view of user type objects

To view the properties of an *SQLUDT object, right-click the required object, and select **Properties**. The Properties window appears showing details on the Source data type. Figure 39 on page 60 shows properties for the MONEY distinct type.

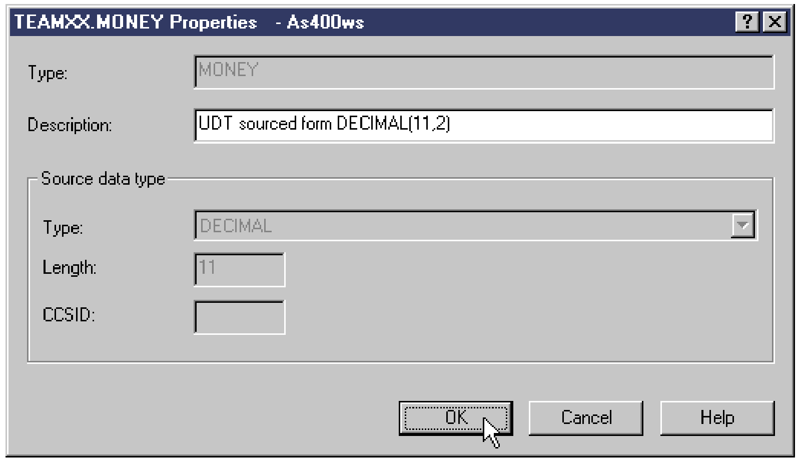
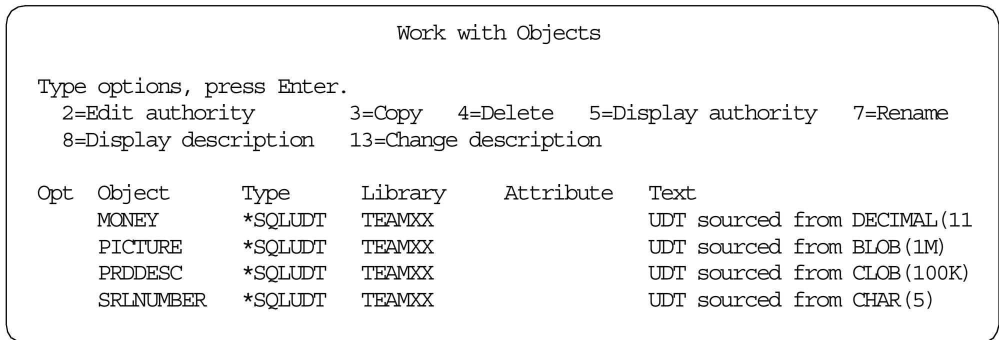


Figure 39. UDT properties dialog

You can also view distinct type information using the *SQLUDT object from a 5250 session. To work with *SQLUDT objects in the TEAMXX library, use the following command:

```
WRKOBJ OBJ(TEAMXX/*ALL) OBJTYPE(*SQLUDT)
```

The Work with Objects screen is displayed as shown in Figure 40. Information on the source types of distinct types is not available here, but you can find out what distinct types are in a library.



```
                                 Work with Objects

 Type options, press Enter.
   2=Edit authority        3=Copy   4=Delete   5=Display authority   7=Rename
   8=Display description   13=Change description

 Opt  Object       Type      Library     Attribute   Text
      MONEY        *SQLUDT   TEAMXX                  UDT sourced from DECIMAL(11
      PICTURE      *SQLUDT   TEAMXX                  UDT sourced from BLOB(1M)
      PRDDESC      *SQLUDT   TEAMXX                  UDT sourced from CLOB(100K)
      SRLNUMBER    *SQLUDT   TEAMXX                  UDT sourced from CHAR(5)
```

Figure 40. Work with *SQLUDT objects

3.5.2.4 The DSPFFD CL command

The Display File Field Description (DSPFFD) CL command can be used to view table column descriptions from a 5250 session.

The following command displays the File Field Description for the PRODMAST01 table in the TEAMXX library:

```
DSPFFD FILE(TEAMXX/PRODMAST01)
```

If we run this statement with the PRODMAST01 table defined, as shown in Table 3 on page 31, the DSPFFD Display Spooled File screen is displayed as shown in Figure 41.

```
                                 Display Spooled File
File  . . . . . :   QPDSPFFD                              Page/Line   1/1
Control . . . . .                                         Columns     1 - 130
Find  . . . . . .
*...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8....
                             Display File Field Description
 Input parameters
   File  . . . . . . . . . . . . . . . . . . . :  PRODMAST01
     Library . . . . . . . . . . . . . . . . . :  TEAMXX
 File Information
   File  . . . . . . . . . . . . . . . . . . . :  PRODMAST01
     Library . . . . . . . . . . . . . . . . . :  TEAMXX
   File location . . . . . . . . . . . . . . . :  *LCL
   Externally described  . . . . . . . . . . . :  Yes
   Number of record formats  . . . . . . . . . :      1
   Type of file  . . . . . . . . . . . . . . . :  Physical
   SQL file type . . . . . . . . . . . . . . . :  TABLE
   File creation date  . . . . . . . . . . . . :  09/28/99
 Record Format Information
   Record format . . . . . . . . . . . . . . . :  PRODMAST01
   Format level identifier . . . . . . . . . . :  5D683E61EECB3
   Number of fields  . . . . . . . . . . . . . :      5
   Record length . . . . . . . . . . . . . . . :    112
 Field Level Information
              Data        Field  Buffer    Buffer        Field    Column
   Field      Type       Length  Length  Position        Usage    Heading
   PMNBR      CHAR            5       5         1        Both     PRODUCT_NUMBER
     Alternative name  . . . . . . . . . . . . :  PRODUCT_NUMBER
     User defined-type name  . . . . . . . . . :  SRLNUMBER
     User defined-type library name  . . . . . :  TEAMXX
     Default value . . . . . . . . . . . . . . :  None
     Coded Character Set Identifier  . . . . . :     37
   PMNAM      CHAR           25      25         6        Both     PRODUCT_NAME
     Alternative name  . . . . . . . . . . . . :  PRODUCT_NAME
     Default value . . . . . . . . . . . . . . :  None
     Coded Character Set Identifier  . . . . . :     37
   PMDESC     CLOB       102400      34        31        Both     PRODUCT_DESCRIPTION
     Alternative name  . . . . . . . . . . . . :  PRODUCT_DESCRIPTION
     Allocated Length  . . . . . . . . . . . . :      0
     User defined-type name  . . . . . . . . . :  PRDDESC
     User defined-type library name  . . . . . :  TEAMXX
     Allows the null value
     Coded Character Set Identifier  . . . . . :     37
   PMPRIC     PACKED      11  2       6        65        Both     PRODUCT_PRICE
     Alternative name  . . . . . . . . . . . . :  PRODUCT_PRICE
     User defined-type name  . . . . . . . . . :  MONEY
     User defined-type library name  . . . . . :  TEAMXX
     Default value . . . . . . . . . . . . . . :  None
   PMPICT     BLOB      1048576      42        71        Both     PRODUCT_PICTURE
     Alternative name  . . . . . . . . . . . . :  PRODUCT_PICTURE
     Allocated Length  . . . . . . . . . . . . :      0
     User defined-type name  . . . . . . . . . :  PICTURE
     User defined-type library name  . . . . . :  TEAMXX
     Allows the null value
     Coded Character Set Identifier  . . . . . :  65535
```

Figure 41. File field description for the PRODMAST01 table

The File Field Description contains a Field Level Information section that lists the Data Type, Field Length, Buffer Length, Buffer Position, Field Usage, and Column Heading for each field. The User defined-type name and User defined-type library name are provided for fields using a distinct type.

Note how the Buffer Length was adjusted for the PRODUCT_DESCRIPTION and PRODUCT_PICTURE fields. Both fields are sourced from an LOB type. An LOB type value is represented in the record structure by a pointer to a data space location. This pointer must be aligned on a 16-byte boundary. Therefore, the database manager assigns a buffer, which is large enough to accommodate a required shift to the next 16-byte boundary and 32 bytes for the pointer.

Note also how the Data Type (`PACKED`) and Field Length (`11 2`) of the PRODUCT_PRICE column relate to the source type of MONEY, that is, DECIMAL(11,2).

3.5.3 Database recovery

This section describes how to save and restore distinct types and dependent tables and some of the considerations that apply.

3.5.3.1 Saving a table using UDT

To save a table using a distinct type to a save file, complete these steps:

1. Use the `CRTSAVF` CL command to create a new save file.
2. Use the `SAVOBJ` CL command to save the table object.

To create a new save file in the TEAMXX library, use the following command:

```
CRTSAVF FILE(TEAMXX/UDTASAVF) TEXT('UDT Lab A Save File')
```

To save the PRODMAST01 table in the TEAMXX library, defined as shown in Table 3 on page 31, use the following command:

```
SAVOBJ OBJ(PRODMAST01) LIB(TEAMXX) DEV(*SAVF) OBJTYPE(*FILE)
  SAVF(TEAMXX/UDTASAVF)
```

Note

The database manager allows you to save a table that is using distinct types without saving the required distinct types. A database administrator may want to distribute a shared UDT library to a number of servers without having to save the UDT library on each server.

3.5.3.2 Saving a UDT

To save a distinct type to a save file, perform these steps:

1. Use the `CRTSAVF` CL command to create a new save file.
2. Use the `SAVOBJ` CL command to save the distinct type object.

Note that, as with other objects, distinct types can also be saved to other offline media.

To create a new save file in the TEAMXX library, use the following command:

```
CRTSAVF FILE(TEAMXX/UDTBSAVF) TEXT('UDT Lab B Save File')
```

To save the MONEY distinct type in the TEAMXX library, defined as shown in Table 4 on page 31, use the following command:

```
SAVOBJ OBJ(MONEY) LIB(TEAMXX) DEV(*SAVF) OBJTYPE(*SQLUDT)
  SAVF(TEAMXX/UDTBSAVF)
```

3.5.3.3 Dropping a UDT in use

There are dependencies between User-defined Distinct Types and other objects, such as tables with distinct type columns. The database manager requires that dependent objects be dropped first. This section shows what to expect if attempting to drop a distinct type that is being used by a table.

To drop a distinct type using Operations Navigator, you open the required library, right-click on the distinct type object you wish to delete, and select **Delete** from the context menu.

If you attempt to delete the MONEY distinct type from the TEAMXX library when it is in use by a table, a Database Error window appears indicating that the distinct type cannot be dropped as shown in Figure 42.

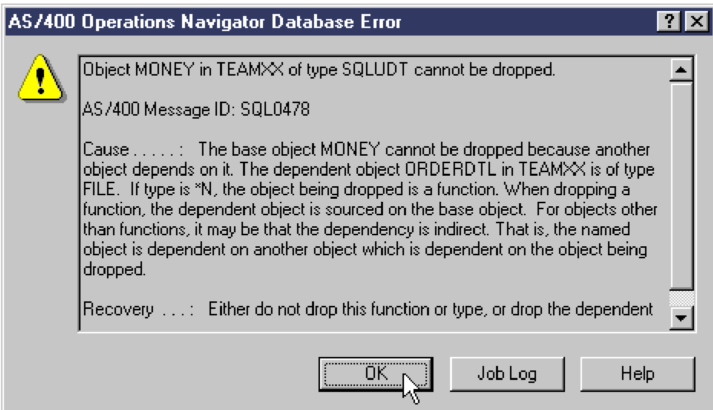


Figure 42. UDT cannot be dropped error window

The error occurs if the database manager finds another object that uses the distinct type to be dropped. In this case, the error message indicates that the dependent object is the ORDERDTL table, which has a column of distinct type MONEY. As advised in the error message Recovery note, dependent objects must be dropped first.

To drop a distinct type using the SQL interface, use the DROP DISTINCT TYPE statement.

We may attempt to delete the MONEY distinct type from the TEAMXX library when it is in use by a table as follows:

```
drop distinct type teamxx/money;
```

Then, the run history in the Operations Navigator Run SQL Scripts window shows that this statement failed by providing the following message:

```
SQL0478 - Object MONEY in TEAMXX of type SQLUDT cannot be dropped.
```

The job log provides further details on SQL execution errors. In the Operations Navigator Run SQL Scripts window, select **View->Job Log...** to display the Job Log window.

Double-clicking the Message ID of interest (in this case SQL0478) displays the Detailed Message Information window as shown in Figure 43 on page 64.

The detailed message information indicates that the dependent object is the PRODMAST01 table, which has a column of distinct type MONEY. The Recovery note again advises that dependent objects must be dropped first.

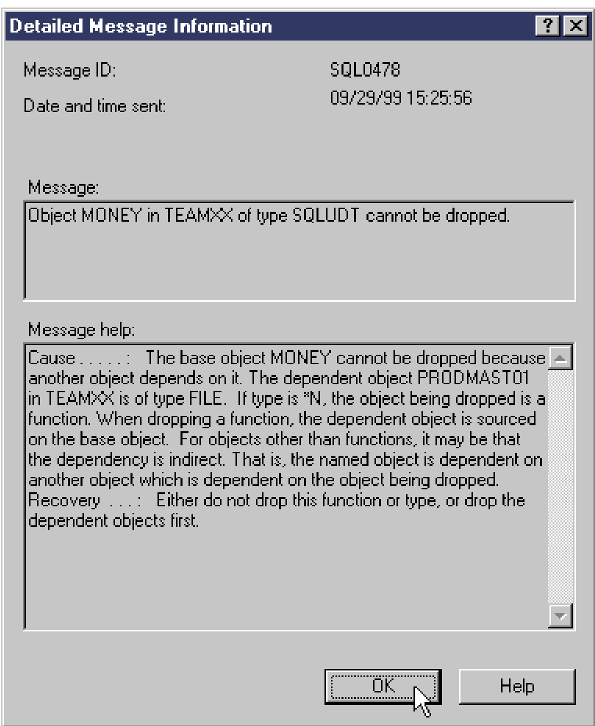


Figure 43. UDT cannot be dropped message details

3.5.3.4 Dropping a table using UDT

As seen in 3.5.3.3, “Dropping a UDT in use” on page 63, objects that depend on a User-defined Distinct Type must be dropped before the distinct type can be dropped.

To drop a table using Operations Navigator, you open the required library, right-click on the table object you want to delete, and select **Delete** from the context menu.

To drop a table using the SQL interface, use the DROP TABLE statement. For example, use the following statement to drop the PRODMAST01 table in the TEAMXX library:

```
drop table teamxx/prodmast01;
```

3.5.3.5 Dropping a UDT

A distinct type can be dropped by issuing the DROP DISTINCT TYPE statement. This statement accepts two additional options that determine what actions are performed by the database manager:

- No option specified: If a user-defined type can be dropped, every User-defined Function that has the following elements is also dropped:
  - Parameters of the type being dropped
  - A return value of the type being dropped
  - A reference to the type being dropped

Consider this example:

```
DROP DISTINCT TYPE money
```

If there is no table using the `money` data type, the type definition, along with all dependent functions, are dropped.

- CASCADE: All dependent objects, along with the UDT definition, are dropped, for example:

```
DROP DISTINCT TYPE money CASCADE
```

This statement drops all tables and UDFs that reference the `money` distinct type.

- RESTRICT: The UDT is dropped only if there are no dependent objects, for example:

```
DROP DISTINCT TYPE money RESTRICT
```

This statement drops the UDT only if there are no UDFs and tables that refer to it.

Note

You can omit the keyword DISTINCT in the DROP DISTINCT TYPE statement.

To drop a distinct type using Operations Navigator, you open the required library, right-click on the distinct type object you wish to delete, and select **Delete** from the context menu. If there are no dependent objects, the right panel refreshes and you should see that the distinct type object has been removed from the library. Note that the Operations Navigator Delete Object dialog uses the DROP TYPE statement with no option specified.

3.5.3.6 Restoring a table using a UDT with UDT not restored

As shown in 3.5.3.3, "Dropping a UDT in use" on page 63, there are dependencies between User-defined Distinct Types and other objects, such as tables with distinct type columns. These dependencies need to be considered when restoring objects that use distinct types. We now examine what to expect if a table is restored from a save file when a required distinct type is not accessible.

We may restore the PRODMAST01 table, used in our example 3.5.3.1, "Saving a table using UDT" on page 62, with the following CL command:

```
RSTOBJ OBJ(PRODMAST01) SAVLIB(TEAMXX) DEV(*SAVF) OBJTYPE(*FILE)
  SAVF(TEAMXX/UDTASAVF)
```

Then, the following message appears in the 5250 message line, indicating that the table was successfully restored:

```
1 objects restored from TEAMXX to TEAMXX.
```

If we then try accessing the restored table using Operations Navigator by double-clicking the table object, a Database Error window appears. The error indicates that a distinct type cannot be found as shown in Figure 44 on page 66.

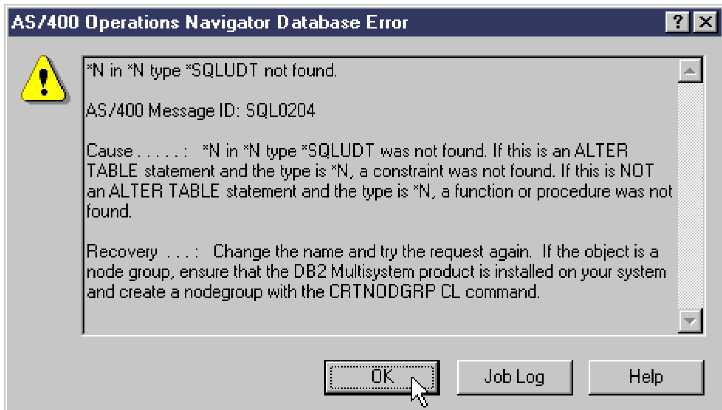


Figure 44. UDT not found error window

The error occurs if the database manager cannot find a required distinct type. In this case, the MONEY distinct type is missing. If the missing distinct type is not known, the data dictionary facilities discussed in 3.5.2, “Keeping track of distinct types” on page 57, can be used to identify it.

If we try accessing the restored table using the SQL interface as follows:

```
select * from teamxx/prodmast01;
```

The run history shows that this statement failed and the following message appears:

```
SQL0204 - *N in *N type *SQLUDT not found.
```

We can select **View->Job Log...** in the Run SQL Scripts window to display the Job Log window.

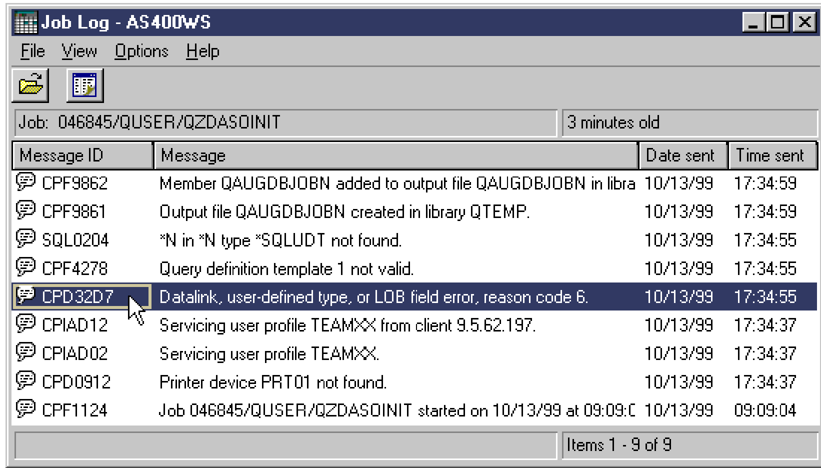


Figure 45. Job log for UDT not found

Double-clicking on the Message ID of interest, in this case SQL0204, does not identify the missing UDT. Returning to the job log, we see two other messages logged with the SQL0204 message as shown in Figure 45. Double-clicking on the `Datalink, user-defined type, or LOB field error, reason code 6`**.** message, as highlighted in Figure 45, displays the Detailed Message Information window shown in Figure 46.

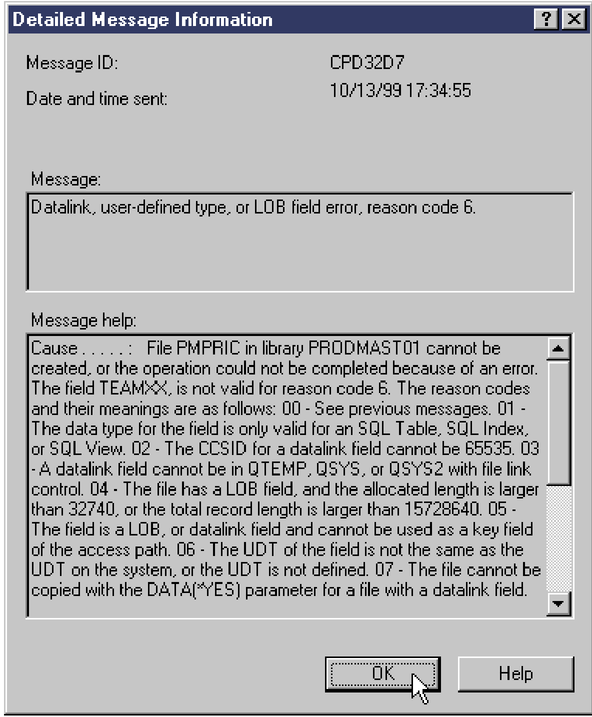


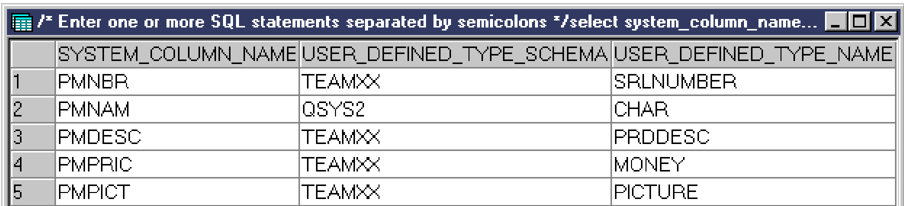
Figure 46. UDT error message details

The detailed message information shown in Figure 46 indicates that the UDT not found problem is with the PMPRIC column in the TEAMXX/PRODMAST01 table.

We can check the data type of the PMPRIC column in the SYSCOLUMNS catalog with the following SQL statement:

```
select
  system_column_name,
  user_defined_type_schema,
  user_defined_type_name
from
  syscolumns
where
  table_name = 'PRODMAST01' and
  table_schema = 'TEAMXX';
```

If we run this statement using the Operations Navigator Run SQL Scripts window, the query results viewer displays PRODMAST01 column details as shown in Figure 47.


/* Enter one or more SQL statements separated by semicolons */select system_column_name...

	SYSTEM_COLUMN_NAME	USER_DEFINED_TYPE_SCHEMA	USER_DEFINED_TYPE_NAME
1	PMNBR	TEAMXX	SRLNUMBER
2	PMNAM	QSYS2	CHAR
3	PMDESC	TEAMXX	PRDDESC
4	PMPRIC	TEAMXX	MONEY
5	PMPICT	TEAMXX	PICTURE

Figure 47. SYSCOLUMNS details for PRODMAST01 table

Looking at Figure 47 on page 67, we can see that USER_DEFINED_TYPE_NAME for the PMPRIC column is TEAMXX/MONEY. We need to redefine the MONEY distinct type or restore it from the save file to reestablish access to the PMPRIC column.

You could also use the DSPFFD CL command to find the data type of the PMPRIC column instead of using the SYSCOLUMNS catalog. Refer to 3.5.2.4, “The DSPFFD CL command” on page 60, for further details. Another alternative is to use Operations Navigator interface. We show how to display the UDT’s Property dialog in 3.5.2.3, “The *SQLUDT object” on page 59.

Note: Only columns that are defined using missing distinct types are inaccessible. You can still select other columns.

The following SQL example will work because the PRODUCT_NAME column is a built-in data type:

```
select product_name from teamxx/prodmast01;
```

3.5.3.7 Restoring a UDT to allow access to a table using UDT

We may restore the MONEY distinct type, used in our example 3.5.3.2, “Saving a UDT” on page 62, with the following CL command:

```
RSTOBJ OBJ(MONEY) SAVLIB(TEAMXX) DEV(*SAVF) OBJTYPE(*SQLUDT)
  SAVF(TEAMXX/UDTBSAVF)
```

Then, the following message appears in the 5250 message line, indicating that the distinct type was successfully restored:

```
1 objects restored from TEAMXX to TEAMXX.
```

If we then try accessing the restored table using Operations Navigator by double-clicking on the table object, the Edit Table window appears, confirming that the distinct type has been correctly restored.

If we try accessing the restored table using the SQL interface as follows:

```
select * from teamxx/prodmast01;
```

the query results viewer is displayed, again confirming that the distinct type has been correctly restored.

Restoring the distinct type also reestablishes access to objects dependent on the distinct type.

Chapter 4. User Defined Functions (UDFs)

This chapter describes:

- User-defined Function (UDF) types
- Resolving UDFs
- Coding UDFs in SQL and High-level Languages
- Parameters styles for external UDFs
- Using LOBs and UDTs with UDFs
- Debugging UDFs
- Backup/Recovery considerations for UDFs

4.1 A need for User Defined Functions

A function is a relationship between a set of input values and a set of result values. When invoked, a function performs some operation (for example, concatenate) based on the input and returns a single result to the invoker. Functions can be specified anywhere where an expression is allowed in SQL.

On a DB2 UDB system, the functions that are available for use fall into three categories:

- **Built-in Functions**: These functions come pre-installed with the system. They are built into the code of the DB2 UDB system. Examples of such functions are the SUBSTR and the CONCAT function.
- **System Generated Functions**: These functions are automatically generated when a distinct type is created on the system. When a distinct type is created, you are automatically provided with the cast functions between the distinct type and its source type. You are also provided with comparison operators, such as =, <, and >.
- **User Defined Functions (UDFs):** These functions are explicitly created by the users of the system using the `CREATE FUNCTION` SQL statement. This statement names the function and specifies its characteristics.

The User Defined Function (UDF) support is a facility given to the database programmers to create a function that can, subsequently, be used in SQL. It can be thought of as an interface that lets you extend and customize SQL to meet your needs. DB2 UDB for AS/400 comes with a set of built in functions, such as SUBSTRING and CONCAT, but these may not satisfy all of your requirements. With UDFs, you can write your own scalar functions and then, subsequently, use them in SQL statements just like any other system supplied function.

UDFs are useful for the following reasons:

- **Supplement built-in functions:** A User Defined Function is a mechanism with which you can write your own extensions to SQL. The built-in functions supplied with DB2 are a useful set of functions, but they may not satisfy all of your requirements. So, you may need to extend SQL. For example, porting applications from other database platforms may require coding of some platform specific functions.
- **Handle user-defined data types**: You can implement the behavior of a User-defined Distinct Type (UDT) using UDFs. When you create a distinct type, the database provides only cast functions and comparison operators for

© Copyright IBM Corp. 2000

the new type. You are responsible for providing any additional behavior. It is best to keep the behavior of a distinct type in the database where all of the users of the distinct type can easily access it. Therefore, UDFs are the best implementation mechanism for UDTs.

- **Provide function overloading**: Function overloading means that you can have two or more functions with the same name in the same library. For example, you can have several instances of the SUBSTR function that accept different data types as input parameters. Function overloading is one the key features required by the object-oriented paradigm.
- **Allow code re-use and sharing**: A business logic implemented as a UDF becomes part of the database, and it can be accessed by any interface or application using SQL.

UDFs can be written in any of the languages available on the AS/400 system, with the exception of REXX and Java (with Java support coming very soon). You can also use the SQL scripting language to write UDFs.

4.2 UDF types

There are three categories into which User Defined Functions can be divided. These categories and their characteristics are discussed in this section. Refer to 4.4, “Coding UDFs” on page 77, for code examples and implementation details.

4.2.1 Sourced

A sourced UDF enhances the functionality of a function that already exists on the system at the time of creation of the sourced function. In other words, these are functions registered to the database that themselves reference another function. There is no coding involved. You simply register a new function to the database using the CREATE FUNCTION statement. Sourced UDFs are often used to implement the required behavior of UDTs. The following example illustrates how to implement the "-" operator for the money data type without the need for reinventing arithmetic operations:

```
create function   TEAMxx/"-"( MONEY, MONEY )
returns           MONEY
specific          MINUS00001
source            QSYS2/"-"( decimal, decimal );
```

4.2.2 SQL

These are functions that are written entirely using SQL. The body of the function is embedded within the CREATE FUNCTION statement. The SQL UDFs have the structure as shown here:

```
create function myUDF (Parameters )
returns ReturnvValue
language SQL
BEGIN
      sql statements
END;
```

Since these functions are written using pure SQL, it is easy to port them to other database platforms. In the following, SQL UDF is used to retrieve the first two and last two characters of a CLOB value:

```
CREATE FUNCTION slice( p1 clob )
RETURNS CHAR(4)
LANGUAGE SQL
  -- returns the first two and the last two characters of the clob
  s1: BEGIN
    DECLARE temp CHAR(4);
    SET temp = CONCAT(SUBSTR(p1,1,2), SUBSTR(p1,LENGTH(p1)-1,2));
    RETURN temp;
  END s1;
```

Note

To create an SQL UDF, you must have the SQL Development Kit and the ILE C/400 products installed on your development system. Once created, the SQL UDF may be run on an AS/400 system without needing these license programs. The run time support for the SQL UDFs is part of the OS/400.

4.2.3 External

An external function is one that has been written by the user in one of the programming languages on the AS/400 system. External functions can be written in ILE C/400, ILE RPG/400, ILE COBOL/400, ILE CL/400, RPG/400, COBOL/400, and CL/400. You can compile the host language programs to create either programs or service programs. To create an external UDF, the source code for the host language program must first be compiled so that a program or a service program object is created. Then, the `CREATE FUNCTION` statement is used to tell the system where to find the program object that implements the function. The function registered in the following example checks whether the passed BLOB object contains a picture in GIF format. The function was implemented in the C language:

```
create function      TEAMxx/ISGIF( BLOB )
returns              INTEGER
language             C
specific             ISGIF00001
no sql
no external action
external name        'TEAMXX/PICTCHECK(fun_CheckPictureType)'
parameter style       SQL;
```

The following SQL statement uses the newly created function to retrieve product numbers of those products that have an accompanying GIF picture:

```
select product_number from prodmast01 where isgif(product_picture) = 1;
```

4.3 Resolving UDF

Resolving to the correct function to use for an operation is more complicated than other resolution operations since DB2 UDB supports function overloading. This means that a user may define a function with the same name as a built-in function or another UDF on the system. For example, SUBSTR is a built-in function, but the user may define their own SUBSTR function that takes slightly different parameters. Therefore, even resolving to a supposedly built-in function still requires that function resolution be performed. The following sections explain how DB2 UDB for AS/400 resolves references to functions.

4.3.1 UDF function overloading and function signature

As mentioned earlier, DB2 UDB supports the concept of function overloading. This means that you can have two or more functions with the same name in the same library, provided they have a different *signature*. The signature of a function can be defined as the combination of the qualified function name and the data types of the input parameters of the function.

No two functions on the system can have the same signature. The lengths and precision of the input parameters is not considered to be part of the signature. Only the data type of the input parameters is considered to be part of the signature. Therefore, if you have a function called myUDF in library LIB1 that accepts an input parameter of type CHAR(5), you cannot have another function called myUDF in the same LIB1 that accepts CHAR(10). The length of the variable is not considered part of the signature. However, it is possible to have another function myUDF in library LIB1 that accepts a DECIMAL value as an input parameter. The following examples illustrate the concept of the function signature. These two functions *can* exist in the same collection:

```
lib1.myUDF( char(5))
lib1.myUDF(decimal)
```

These two functions *cannot* exist in the same collection:

```
myUDF(char(10))
myUDF(char(5))
```

Notice that certain data types are considered equivalent when it comes to function signatures. For example, CHAR and GRAPHIC are treated as the same type from the signature point of view.

The data type of the value returned by the function is *not* considered to be part of the function signature. This means that you cannot have two functions called myUDF in library LIB1 that accept input parameters of the same data type, even if they return values of different data types.

4.3.2 Function path and the function selection algorithm

On the AS/400 system, there are two types of naming conventions when using SQL. One of them is called the *system naming convention*, and the other one is called the *SQL naming convention*. The system naming convention is native to the AS/400 system, and the SQL naming convention is specified by the ANSI SQL standard.

The function resolution process depends on which naming convention you are using at the time you execute the SQL statement, which refers to a UDF.

4.3.2.1 Function path

When *unqualified* references are made to a UDF inside an SQL statement, DB2 UDB for AS/400 uses the concept of *PATH* to resolve references to the UDF. The path is an ordered list of library names. It provides a set of libraries for resolving unqualified references to UDFs as well as UDTs. In cases where a reference to a UDF matches more than one UDF in different libraries, the order of libraries in the path is used to resolve to the correct UDF.

The path can be set to any desired set of libraries using the SQL `SET PATH` statement. The current setting of the path is stored in the CURRENT PATH special register.

For the SQL naming convention, the path is set initially to the following default value:

```
"QSYS", "QSYS2", "<USER ID>"
```

For the system naming convention, the path is set initially to the following default value:

```
*LIBL
```

When you are using the system naming convention, the system uses the library list of the current job as the path and uses this list to resolve the reference to the unqualified references to the UDFs.

The current path can be changed with the `SET PATH` statement. Note that this statement overrides the initial setting for both naming conventions. For example, you can use the following statement:

```
SET PATH = MYUDFS, COMMONUDFS
```

to set the path to the following list of libraries:

```
QSYS, QSYS2, MYUDFS, COMMONUDFS
```

Notice that the libraries QSYS and QSYS2 are automatically added to the front of the list. This is the case unless you explicitly change the position of these libraries in the `SET PATH` statement. For example, the following statement sets the CURRENT PATH registry to myfunc, QSYS, QSYS2:

```
SET PATH myfunc, SYSTEM PATH
```

For portability reasons, we recommend that you use SYSTEM PATH registry rather then QSYS and QSYS2 library names on the SET PATH statement.

4.3.2.2 Function resolution in the CREATE FUNCTION statements

The function resolution for the supported naming conventions works as described here:

- **SQL naming convention:** If the function name is qualified, the function is created in the library specified. If a user profile with the same name as the qualifying library exists, that user profile is the owner of the created function; otherwise, the user profile that is creating the function is the owner of the created function. If the function name is not qualified, the function is created in a library with the same name as the user profile executing the SQL statement. If such a library does not exist, you will receive an error message when executing the statement.
- **System naming convention:** If the function name is qualified, the function is created in the specified library. The owner of the function is the name of the user profile that executes the SQL statement. If the function name is not qualified, the function is created in the current library (*CURLIB). If there is no current library, the function is created in QGPL.

If you are using system naming convention, you code the qualified function name in the `CREATE FUNCTION` SQL statement in the following way:

```
CREATE FUNCTION LIB1/myUDF(CHAR(5) )
...
```

If you are using SQL naming convention, you code the qualified function name in the `CREATE FUNCTION` SQL statement in the following way:

```
CREATE FUNCTION LIB1.myUDF( CHAR(5) )
...
```

4.3.2.3 Function resolution in data manipulation statements

The function resolution for the supported naming conventions works as described here:

- **SQL naming convention:** If the name of the UDF is qualified, the system searches for the function in the specified library. The function matching the function signature specified in the SQL statement is chosen. The following statements show how to invoke a UDF with its qualified name:

  ```
  SELECT LIB1.myUDF( FIELD1 ) FROM LIB1.TABLE1
  ```

- **System Naming Convention:** You cannot have qualified references to UDFs using the system naming convention. Qualified references to functions are allowed only in the SQL naming convention. Therefore, a statement, such as `SELECT LIB1/myUDF( FIELD1 ) FROM LIB1/TABLE1`, is *not* allowed.

If there is more than one function having a signature that matches those specified in the SQL statement, the list of libraries in the current path is used to resolve the reference. The system picks the first function matching the signature from the libraries specified in the path. In case there are no functions exactly matching the signature, the system uses *parameter promotion* (this concept is discussed in the following section) to find the "best fit" for the function specified in the SQL statement. If the system cannot find the function matching the required signature, you receive an SQL error message similar to the one shown here:

```
SQL0204 - GETDESCRIPTION in *LIBL type *N not found.
```

All functions on the system, *including* built-in functions, have to pass through the function selection algorithm before being selected for execution.

4.3.3 Parameter matching and promotion

When an SQL DML statement references a UDF, the system, at first, tries to find an exact match for the function by searching for functions that have the same signature. If the system finds a function having input parameters that exactly match those specified in the DML statement, that function is chosen for execution. In case the system cannot find any function in the path that exactly matches those specified on the DML statement, the parameters on the function call in the DML statement are *promoted* to their next higher type. Then, another search is made for a function that accepts the *promoted* parameters as input. During parameter promotion, a parameter is cast to its next higher data type. For example, a parameter of type CHAR is promoted to VARCHAR, and then to CLOB. There are restrictions on the data type to which a particular parameter can be promoted. We explain this concept with an example.

Let us assume that you have created a table CUSTOMER in library LIB1. This table has, among its other fields, a field named CUSTOMER_NUMBER, which is a CHAR(5). Let us also assume that you have written a function GetRegion that will perform some processing and return the region to which your customer belongs. The data type of the parameter that this function accepts as input is defined to be of type CLOB(50K). Let us assume that there are no other functions called GetRegion in the path. Now, if you execute the following query, you will see that the function GetRegion(CLOB(50K)) is actually executed:

```
select GetRegion( customer_number ) from customer
```

How is this possible? The field CUSTOMER_NUMBER from the CUSTOMER table has the data type CHAR(5). The function GetRegion actually accepts a CLOB as a parameter, and there are no other functions called GetRegion in the path. In its attempt to resolve the function call, the system first searched the library path for a UDF called GetRegion, which accepts an input parameter of type CHAR. However, no such UDF was found. The system then *promoted* the input parameter, in our case the customer number, up in the hierarchy list of promotable types to a VARCHAR. Then, a search was made for an UDF called GetRegion, which accepted an input parameter of type VARCHAR. Again, no such UDF was found. Then, the system *promoted* the input parameter up the hierarchy list to a CLOB. A search was made for an UDF called GetRegion, which accepted an input parameter of type CLOB. This time the search was successful. The system invoked the UDF GetRegion(CLOB(50K)) to satisfy the user request.

The concept of parameter promotion is clearly demonstrated in the previous example. Table 9 on page 76 gives a list of data types and the data types to which they can be promoted.

Table 9. Precedence of data types

Data type	Data type precedence list (in best to worst order)
CHAR or GRAPHIC	CHAR or GRAPHIC, VARCHAR or VARGRAPHIC, CLOB, or DBCLOB
VARCHAR or VARGRAPHIC	VARCHAR or VARGRAPHIC, CLOB, or DBCLOB
CLOB or DBCLOB	CLOB or DBCLOB
BLOB	BLOB
SMALLINT	SMALLINT, INTEGER, DECIMAL or NUMERIC, REAL, DOUBLE
INTEGER	INTEGER, DECIMAL or NUMERIC, REAL, DOUBLE
DECIMAL or NUMERIC	DECIMAL or NUMERIC, REAL, DOUBLE
REAL	REAL, DOUBLE
DOUBLE	DOUBLE
DATE	DATE
TIME	TIME
TIMESTAMP	TIMESTAMP
DATALINK	DATALINK
A distinct type	The same distinct type

As you see from Table 9, data types can be promoted up the hierarchy only to particular data types. Distinct types cannot be promoted. Even though distinct types are based on one of the built-in data types, it is not possible to promote distinct types to anything other than the same type.

Parameters cannot be demoted down the hierarchy list as shown in Table 9. This means that, if the CUSTOMER_NUMBER column of the CUSTOMER table is a CLOB, and the GetRegion UDF was defined to accept a CHAR as an input parameter, a call, such as the one shown here, will fail because function resolution will not find the UDF:

```
SELECT GetRegion( CUSTOMER_NUMBER ) from customer
```

4.3.4 The function selection algorithm

The function selection algorithm searches the library path for a UDF using the steps outlined here:

1. Finds all functions from the catalog (SYSFUNCS) and built-in functions that match the name of the function. If a library was specified, it only gets those functions from that library. Otherwise, it gets all functions whose library is in the function path.
2. Eliminates those functions whose number of defined parameters does not match the invocation.
3. Eliminates functions whose parameters are not compatible or "promotable" to the invocation.

For the remaining functions, the algorithm follows these steps:

1. Considers each argument of the function invocation, from left to right. For each argument, it eliminates all functions that are not the best match for that argument. The best match for a given argument is the first data type appearing in the precedence list. Lengths, precessions, scales, and the "FOR BIT DATA" attribute are not considered in this comparison. For example, a DECIMAL(9,1) argument is considered an exact match for a DECIMAL(6,5) parameter, and a VARCHAR(19) argument is an exact match for a VARCHAR(6) parameter.
2. If more than one candidate function remains after the above steps, it has to be the case (the way the algorithm works) that all the remaining candidate functions have identical signatures but are in different schemas. It chooses the function whose schema is earliest in the user's function path.
3. If there are no candidate functions, it signals the error SQLSTATE 42884.

Figure 48 summarizes the steps performed by DB2 UDB for AS/400 to resolve a call to a UDF.

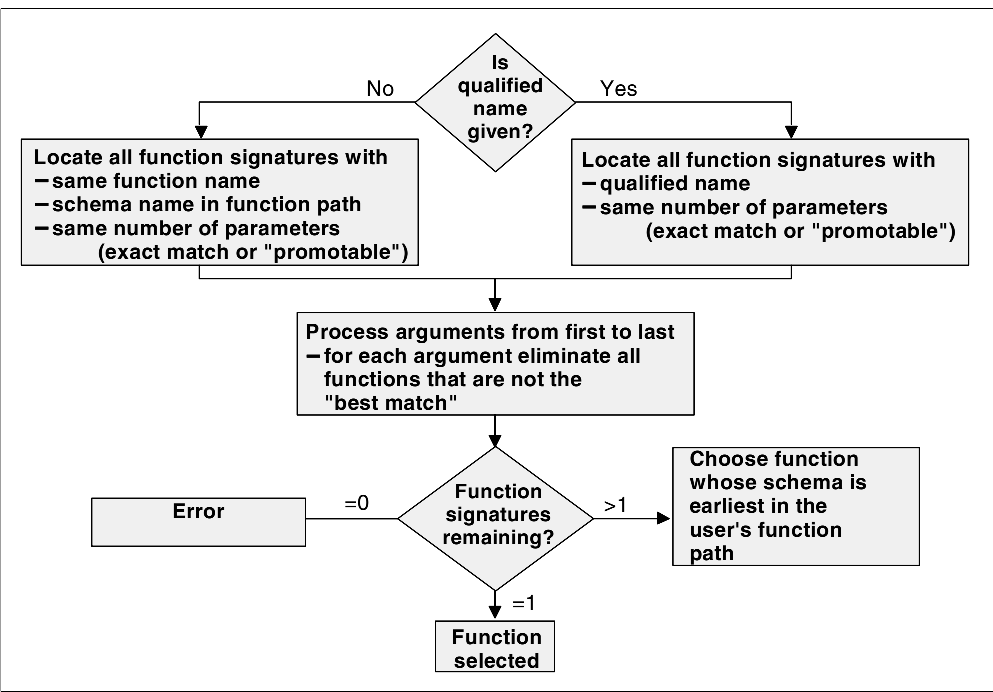


Figure 48. Function resolution algorithm

4.4 Coding UDFs

Before a UDF can be used in a Data Manipulation Language (DML) statement, it must be registered with the database. This can be done by using the `CREATE FUNCTION` DDL statement. The `CREATE FUNCTION` statement is used to define the name of the function, the type of the function, the number and data type of the input parameters, and the data type of the value returned by the UDF to the invoking process. The `CREATE FUNCTION` statement can be embedded in an application program, or it can be executed interactively. All three types of UDFs can be created by this statement. The syntax of the statement is different for

sourced UDFs, SQL, and external UDFs. After a UDF is registered, it can be used in any SELECT, UPDATE, DELETE DML statement from any interface from where an SQL statement can be executed.

When a UDF is registered with the database, entries are made into the SYSFUNCS and SYSPARMS system tables. These tables store information on every function that is registered with the database. The information that is recorded in these tables is discussed in 4.6, "The system catalog for UDFs" on page 116.

UDFs can be defined to accept and return parameters of any datatype including distinct types.

Apart from being classified as sourced, SQL, and external, UDFs can also be classified as *scalar* or *column*. The *scalar* functions return a single value each time they are invoked. These functions are executed once for every row of the table. The SUBSTR() built-in function is an example of a *scalar* function. *Column* functions receive a set of values as input. They return one value. The AVG() built-in function is an example of a *column* function. Scalar functions can be created as External, SQL, and Sourced functions. Column functions can only be created as sourced functions.

4.4.1 Coding sourced UDFs

A sourced function is a function that references another function which, in turn, is already registered with the database. The UDF can be sourced from any function that is registered to the database, including built in functions. These operators are: +, -, *, /, ||, CONCAT. The name of the sourced function cannot be any of the comparison operators on the system. Functions for these operators are part of the database system. There is also a number of other system functions that cannot be used as the name of the sourced UDF. For more information on these restrictions, refer to *DB2 UDB for AS/400 SQL Reference*, SC41-5612.

If the Sourced UDF being created references a scalar function, it inherits all the attributes of the referenced function. When a sourced UDF is created, a small service program is automatically created by the system in the background. This service program is there to assist the system in the save/restore and grant/revoke operations. You can think of it as a place holder for a function body, which is implemented elsewhere (typically by a built-in function).

4.4.1.1 Creating sourced UDFs as scalar functions

We illustrate the use of the CREATE FUNCTION statement with an example. We create an overloaded version of the SUBSTR function. The function accepts three input parameters: a distinct type PRDDESC and two integers. It returns a parameter of type PRDDESC. The function is sourced from the built-in function SUBSTR(CLOB, INTEGER, INTEGER). In this example, we show you how to create the script through the Operations Navigator Run SQL Scripts utility.

To open the Run SQL Scripts window, follow the steps outlined here:

1. Open an Operations Navigator session.
2. Right-click the **Database** object.
3. From the Database context menu, select the **Run SQL Scripts** option. This opens the Run SQL Scripts window (Figure 49).

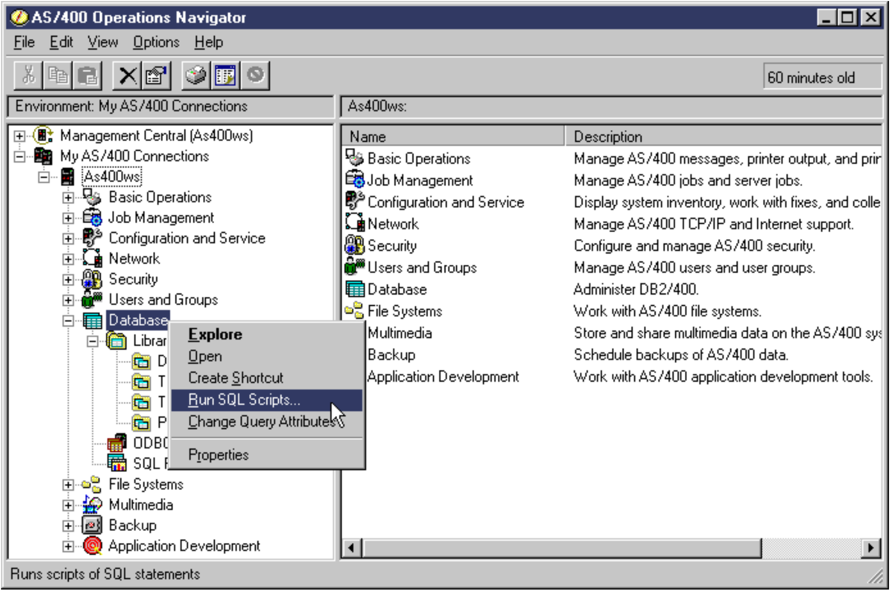


Figure 49. Opening up a Run SQL Scripts session

Figure 50 shows the CREATE FUNCTION statement for the SUBSTR(PRDDESC, INTEGER, INTEGER) function. The result of running the statement is shown in the Run History panel of the Run SQL Scripts utility.

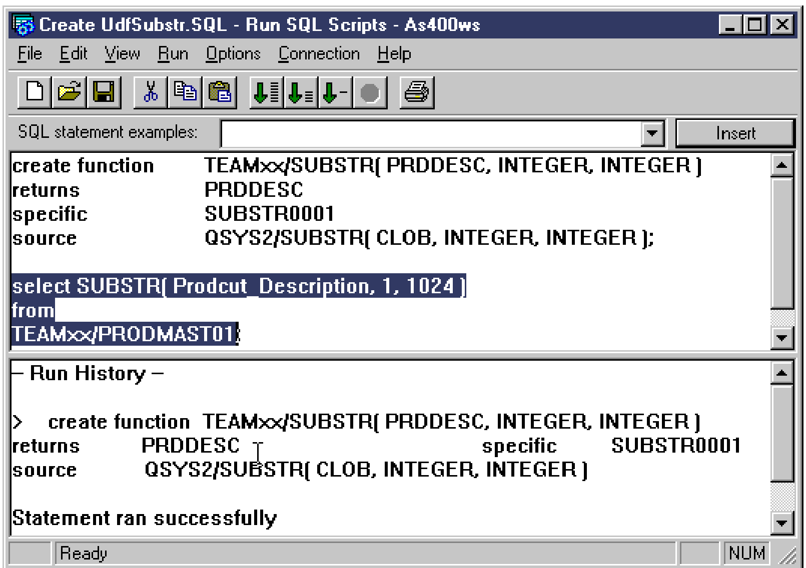


Figure 50. The CREATE FUNCTION statement for sourced UDF

Let us examine the CREATE FUNCTION statement shown in Figure 50 in detail. The numbered sections are explained in the list that follows:

```
create function   TEAMxx/SUBSTR( PRDDESC, INTEGER, INTEGER )   1
returns           PRDDESC                                       2
specific          SUBSTR0001                                    3
source            QSYS2/SUBSTR( CLOB, INTEGER, INTEGER );       4
```

CREATE FUNCTION statement notes

1. We qualify the function name with the library name, TEAMxx in this case. We use the system naming convention. If you do not qualify the function's name in the `CREATE FUNCTION` statement, the function is created in the current library. The function takes three input parameters: the distinct type PRDDESC and two parameters of type INTEGER. The definition for the distinct type PRDDESC is taken from a library in the library list. If no definition of the UDT is found, the `CREATE FUNCTION` statement returns an error. If multiple definitions of the distinct type are found, the first definition found in the library list is used.
2. The RETURNS clause specifies the data type of the value returned by the function. Note that the data type of the value returned by the function can be different from the type of the value returned from the referenced program object. However, the type returned from the program object must be castable to the data type of the value returned by the function you are creating. For example, you cannot define a SUBSTR function that returns a DECIMAL data type as the return value of the function.
3. This is the SPECIFIC NAME clause of the `CREATE FUNCTION` statement. Every function created on the AS/400 system must have a specific name. This name must be unique for a given library. The service program that is created by the DB2 UDB for AS/400 to implement the function has the same name as the specific name provided in this clause. This is an optional clause. If you do not specify a specific name for the function, the system will generate a specific name. Normally, the specific name is the same as the function's name, provided it is a valid system name (for instance, it's not longer than 10 characters). However, if a function with that specific name already exists, the system generates a unique name. When the service program created for the sourced function is saved and restored to another system, the attributes of the `CREATE FUNCTION` statement are automatically added to the system catalogs.
4. This is the SOURCE clause of the `CREATE FUNCTION` statement, which points to the existing function that is the source for the function being created. In our example, the source function is SUBSTR(CLOB, INTEGER, INTEGER) and it exists in the QSYS2 library.

You can use the Operations Navigator to check that your function was created correctly. To see the definition of your function, follow the steps outlined here:

1. In the main Operations Navigator window, click the (**+**) icon next the Database object to expand its content.
2. Expand the **Libraries** object. You see all the libraries in your library list.
3. Click the name of the library where you created the function. You should see all the database objects in that library displayed in the right panel of the display. Please note that *only* database objects are shown in this panel. You should now see your function listed as shown in Figure 51.

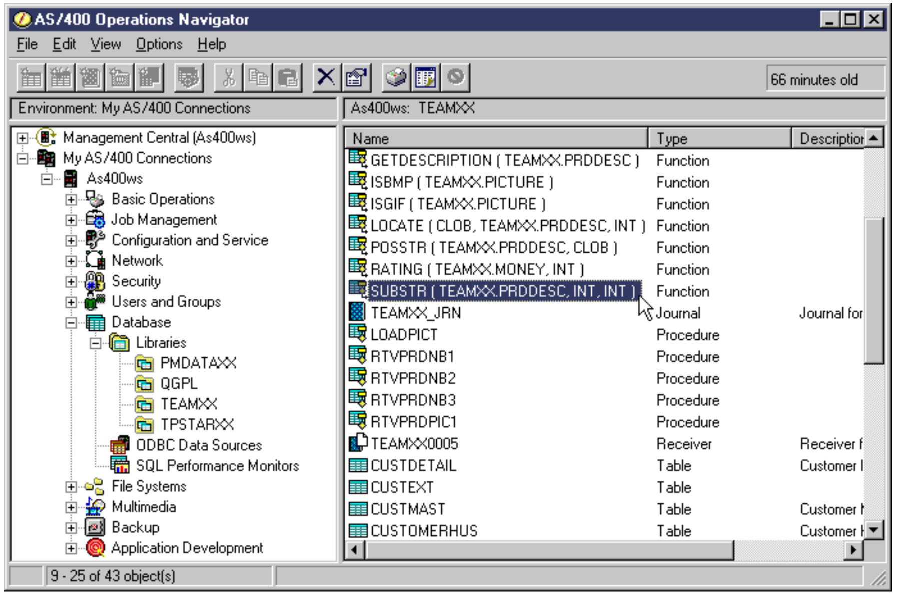


Figure 51. The SUBSTR(PRDDESC, INTEGER, INTEGER) sourced function

Once you register the function with the database, you can use it in your SQL DML statements. The following example shows you how to use the newly created SUBSTR function in a SELECT statement. Our PRODMAST01 test table has a column named PRODUCT_DESCRIPTION that is based on the PRDDESC distinct type. The PRODUCT_DESCRIPTION column is a structured text of type CLOB, which contains the description of the product, the range of sizes for the product, the color of the product, and the best use of the product. Let us assume that we want to get the range of sizes for all products in the PRODMAST01 table. We execute a SELECT statement, such as the one shown in Figure 52.

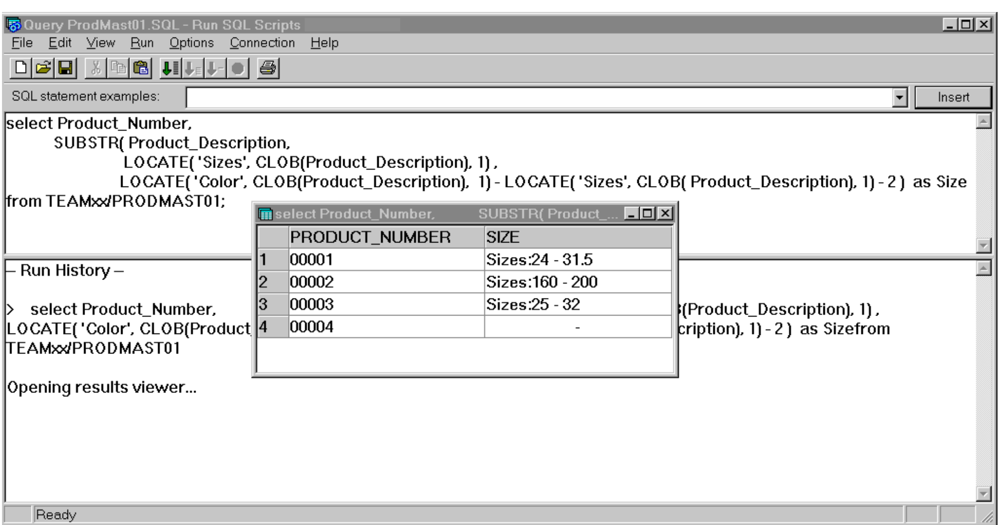


Figure 52. Using the SUBSTR(PRDDESC, INTEGER, INTEGER) function in a query

Notice that in the SELECT statement shown in Figure 52, we have to cast the PRODUCT_DESCRIPTION to CLOB when it is used in the LOCATE built-in function. This is because there is no function called LOCATE that accepts a column of type PRDDESC as an input parameter. However, there is a function

called LOCATE that accepts a CLOB as an input parameter. Therefore, we cast the PRODUCT_DESCRIPTION column of the PRODMAST01 table to a CLOB when we pass it as a parameter to the LOCATE function. Notice also that we do not cast the PRODUCT_DESCRIPTION column when we pass it as a parameter to the SUBSTR function. This is because we just created a SUBSTR function that accepts an input parameter of type PRDDESC. If we were to create a function called LOCATE(CLOB, PRDDESC, INTEGER), we would not need to cast the PRODUCT_DESCRIPTION column in the call to the function in the above statement.

4.4.1.2 Creating sourced UDFs as column functions

We have just seen how to create a *scalar* sourced UDF. We can also create sourced UDFs as column functions. Recall that the argument of a column function is a set of values derived from one or more columns and that it returns one value as the result. Only sourced UDFs can be created as column functions. External and SQL UDFs cannot be created as column functions.

As an example, we create a new MAX function as sourced UDF. The function takes one input parameter of distinct type MONEY. The function returns a value of type MONEY. It is based on the built-in function MAX(DECIMAL), which exists in the QSYS2 library. The `CREATE FUNCTION` statement for this function is shown in Figure 53.

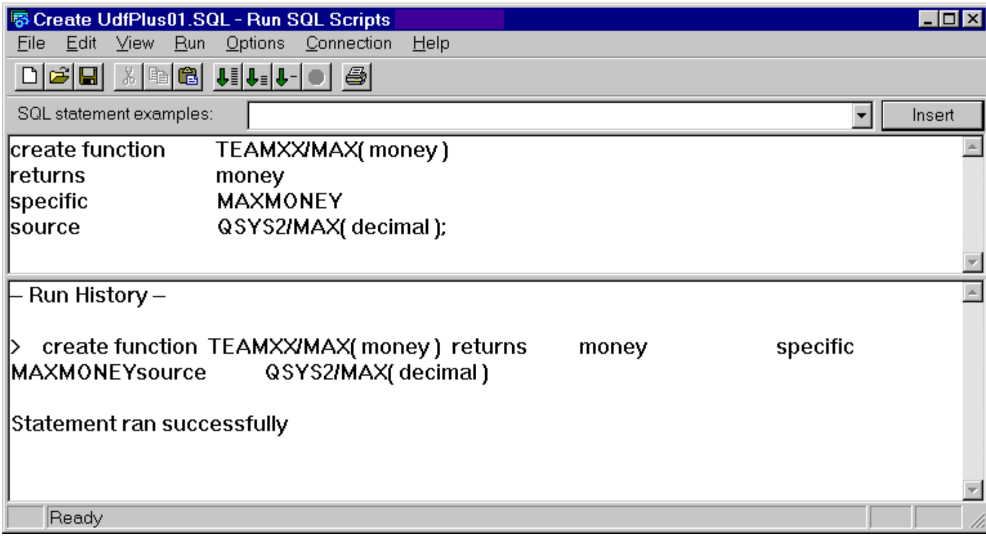


Figure 53. Creating the MAX(MONEY) sourced UDF as a column function

We can now use the newly created MAX(MONEY) function with the ORDER_TOTAL column of the ORDERHDR table as the input parameter. The ORDER_TOTAL column of the ORDERHDR table is of type MONEY. The query and its results are shown in Figure 54.

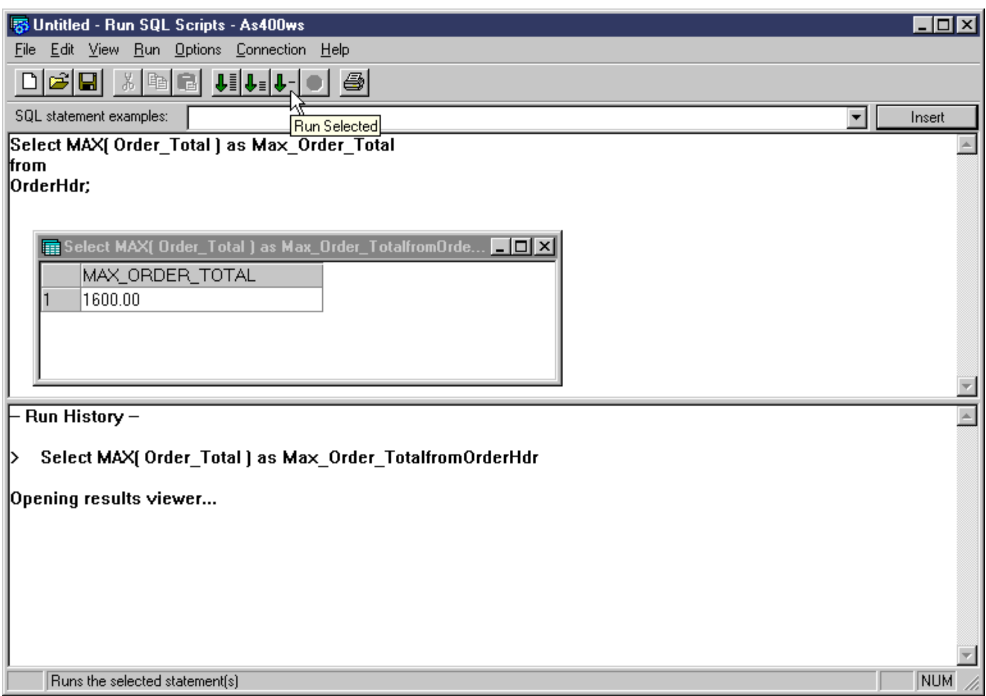


Figure 54. Running the MAX(MONEY) column UDF

Notice that, in the SOURCE clause of the `CREATE FUNCTION` statement in Figure 53, the precision of the DECIMAL input parameter for the referenced function is not specified. If you do not specify the precision of a parameter, the system ignores the precision of the value supplied as the input parameter to the function. In the example in Figure 53, this approach is used in the SOURCE CLAUSE. Similarly, you can specify input parameters without specifying their precision. If you do so, the system ignores the precision of the values that you are supplying as input to the UDF at run time. If the precision was specified for the function's parameters, the system looks for a function that has input parameters which exactly match the precision of those specified on the `CREATE FUNCTION` statement.

4.4.1.3 Creating sourced UDFs over arithmetic operators

You *can* define a sourced UDF over the arithmetic operators available in the system, provided *one* of the new function's parameters is a distinct type. These operators are +, -, *, /, ||. You *cannot* define sourced UDFs over comparison operators, such as =, <, >, and so forth.

As an example, we create a sourced UDF over the "+" operator. This function accepts two input parameters of type MONEY. The function returns a value of type MONEY. The function is based on the built-in function "+"(DECIMAL, DECIMAL). Figure 55 on page 84 shows the `CREATE FUNCTION` statement used to create this function.

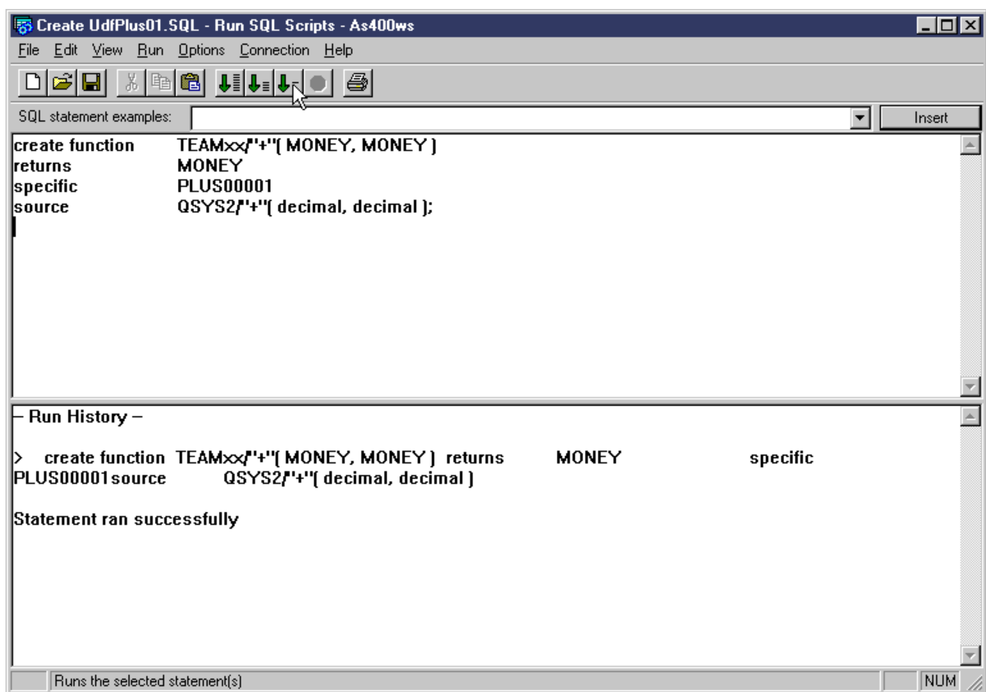
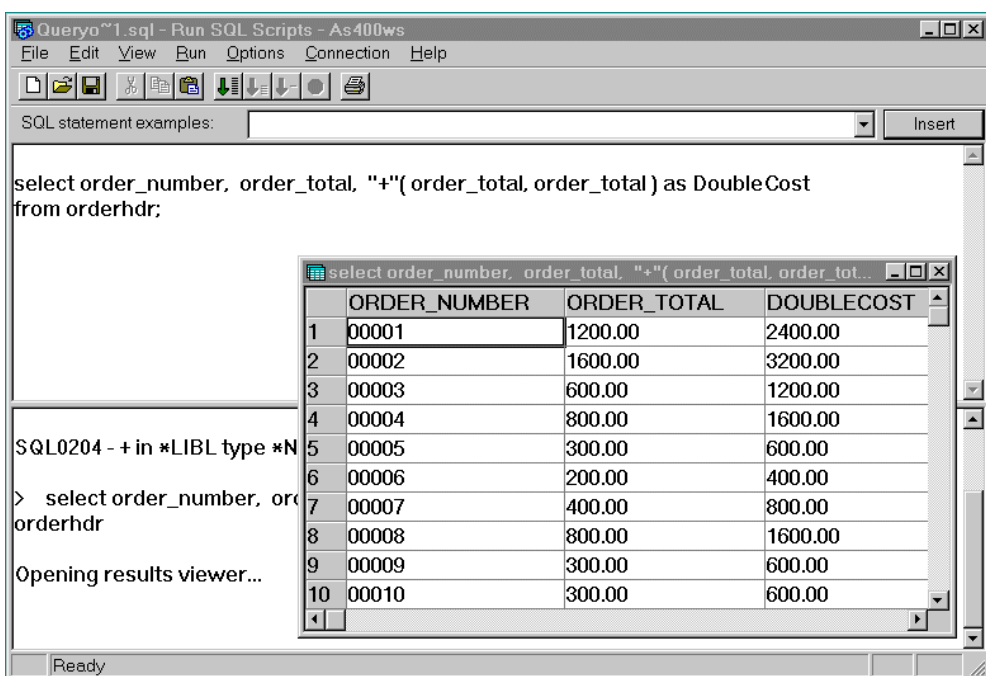


Figure 55. Creating the "+"(MONEY, MONEY) sourced UDF over arithmetic operators

An example query using the newly created function is shown in Figure 56.



	ORDER_NUMBER	ORDER_TOTAL	DOUBLECOST
1	00001	1200.00	2400.00
2	00002	1600.00	3200.00
3	00003	600.00	1200.00
4	00004	800.00	1600.00
5	00005	300.00	600.00
6	00006	200.00	400.00
7	00007	400.00	800.00
8	00008	800.00	1600.00
9	00009	300.00	600.00
10	00010	300.00	600.00

Figure 56. Using the "+"(MONEY, MONEY) sourced UDF

Note that we use the *prefix* notation for calling the "+"(MONEY, MONEY) UDF. Currently, DB2 UDB for AS/400 *does not* support the *infix* notation for calls to UDFs, even if the UDFs are created over arithmetic operators. Calls, such as the one shown here, will fail:

```
select order_number, order_total, order_total + order_total as DoubleCost from
OrderHdr;
```

In addition, when an UDF is defined over an arithmetic operator, you have to enclose the name of the called UDF in double quotes.

4.4.2 Coding SQL UDFs

Until now, you've seen how to create sourced UDFs. In this section, we discuss SQL UDFs. SQL UDFs are functions that use the SQL language to implement their business logic. In SQL UDFs, the entire procedure body is embedded within the `CREATE FUNCTION` statement.

When you execute the `CREATE FUNCTION` statement for the SQL UDF, DB2 UDB for AS/400 walks through a multiphase process to create an ILE C service program object (*SRVPGM). During this process, DB2 UDB for AS/400 generates an intermediary ILE C code with embedded SQL statements. This ILE C code is then precompiled, compiled, and linked automatically. This means that the SQL Development Kit for AS/400, and the ILE C compiler, need to be installed on the system where you plan to develop SQL stored procedures. Once the ILE C object is created, it can be restored onto any V4R4 or higher system and run without the SQL Development Kit and ILE C compiler. Note that the ILE C program object is created with the Activation Group parameter set to *CALLER.

As an example, we create the GetDescription function, which accepts one parameter: product number of type CHAR(5). The function returns the description of the product as a VARCHAR(1024) by substringing the structured text stored in the PRODUCT_DESCRIPTION field of the PRODMAST04 table.

In this example, we also show you how to create a UDF using the Operations Navigator Create SQL Function dialog. The required steps are listed here:

1. In the main Operations Navigator window, click the (**+**) icon next to the Database object to expand its content.
2. Expand the **Libraries** object. You see all the libraries in your library list.
3. Right-click the library in which you want to create the SQL UDF. A context menu appears (Figure 57 on page 86).

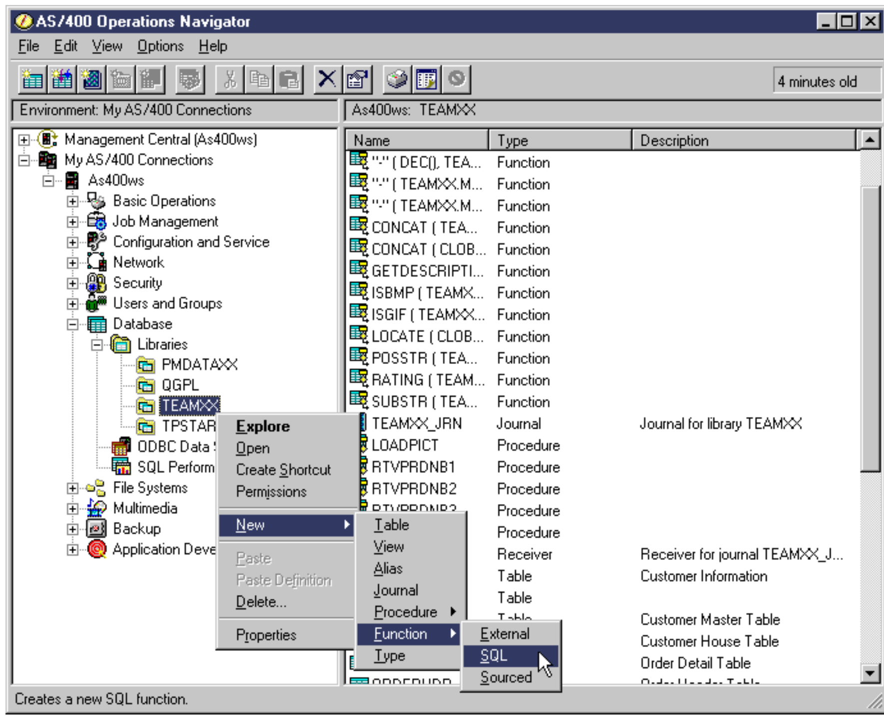


Figure 57. Creating an SQL UDF using the new SQL function dialog

4. Choose the **New->Function->SQL** option. The New SQL Function dialog box appears on the screen.
5. Type the name, description, and specific name of the function. Select the datatype of the value returned by the function (Figure 58).

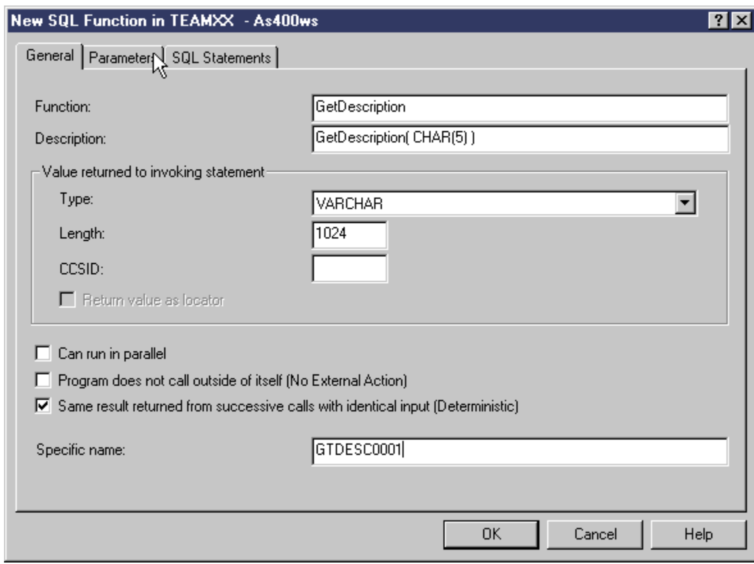


Figure 58. New SQL function dialog

6. Click the **Parameters** tab of the dialog. Click the **Insert** button. Type in the name of the parameter and the length of the parameter. Select the datatype of the parameter (Figure 59).

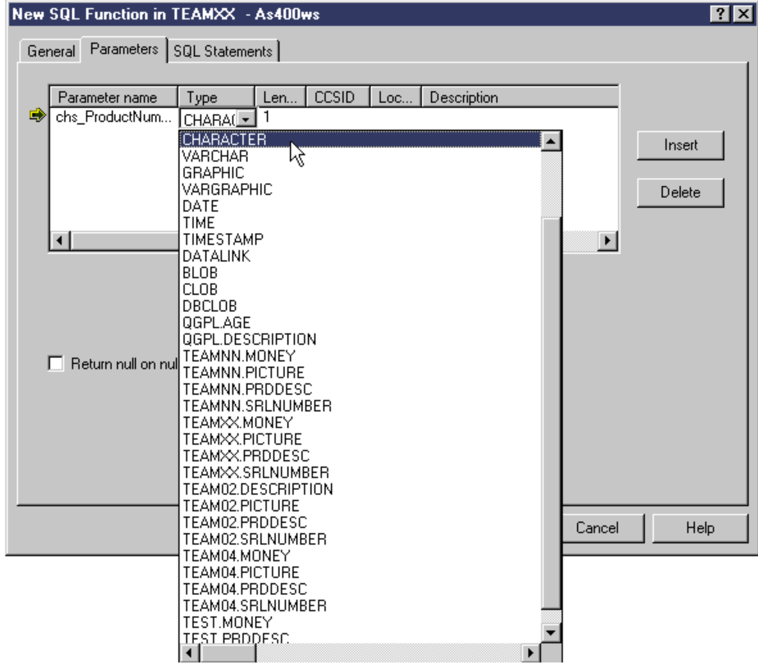


Figure 59. Defining the input parameters for the SQL UDF

7. Click the **SQL Statements** tab. Click in the **Statements** area, and type the SQL statements that will make up the body of the procedure. Click the **OK** button (Figure 60).

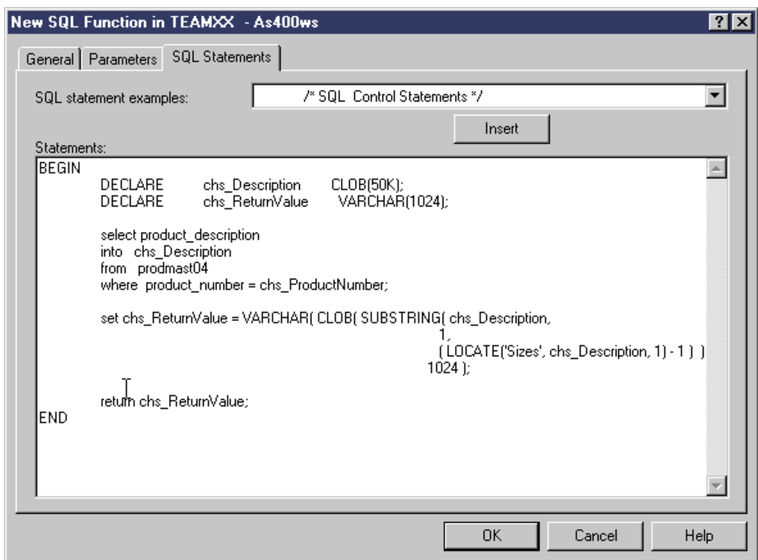


Figure 60. Typing in the body of the SQL UDF.

As you can see, this is a much easier method to create a UDF than with the Run SQL Scripts utility. However, you may find it advantageous to preserve the sources for all of your SQL functions in the form of an SQL script. In this form, your functions can be maintained and re-created more easily.

Important

Several options on the CREATE FUNCTION statement are not allowed for the SQL UDFs:

- FINAL CALL/NO FINAL CALL
- SCRATCHPAD/NO SCRATCHPAD
- DBINFO/NO DBINFO

However, these options may be used in external UDFs. Refer to 4.4.3.3, "Coding external UDFs using the DB2SQL parameter style" on page 103, for more details.

The SQL source for the GetDescription function is shown here. The numbered sections are explained in the list that follows.

```
create function    TEAMxx/GETDESCRIPTION( chs_ProductNumber CHAR(5) )  1
returns            VARCHAR(1024)                                       2
language           SQL                                                 3
specific           GTDESC0001                                          4
is deterministic                                                       5
reads              SQL DATA                                            5
no external action 7
BEGIN 8
DECLARE          chs_Description        CLOB(50K);
DECLARE          chs_ReturnValue        VARCHAR(1024);

select product_description                                            9
into   chs_Description
from   prodmast04
where  product_number = chs_ProductNumber;

set chs_ReturnValue =
VARCHAR( CLOB( SUBSTRING( chs_Description, 1, ( LOCATE('Sizes', 10
 chs_Description,
1) - 1 )  )  ), 1024 );

return chs_ReturnValue;  11
END 12
```

GetDescription function notes

1. The qualified name of the function and the input parameters to the function and their data types are specified here. Unlike the sourced UDFs, here you have to specify names for the input parameters to the function. The GetDescription function shown above has only one input parameter: the product number (chs_ProductNumber) which is of type CHAR(5).
2. This is the RETURNS clause of the `CREATE FUNCTION` statement. Here, you specify the data type of the value returned by the function. This can be any data type available on the system, including complex data types and distinct types. The only restriction is that you cannot return a distinct type if it is based on a datalink. It is a mandatory clause.
3. This is the LANGUAGE clause of the `CREATE FUNCTION` statement. It must be specified in SQL functions. If you specify the language of a function to be SQL, then the body of the function must be specified within the body of the `CREATE FUNCTION` statement. Also, when you specify the language of a function to be SQL, you cannot specify the EXTERNAL NAME clause. The EXTERNAL NAME clause identifies the name of the program to be executed when an external function is being created. The LANGUAGE SQL and the EXTERNAL NAME clauses are mutually exclusive.

4. This is the SPECIFIC NAME clause of the `CREATE FUNCTION` statement. Every function that is created must have a specific name, and this name must be unique for a given library. This clause specifies the specific name of the function. It is not mandatory, but if you do not specify this clause, the system generates a specific name for the function. The system generated name is normally the same as that of the function, provided it is a valid system name. However, if another function exists with the same specific name, the name is generated using rules that are similar to those used for generating unique table and column names.
5. This is the DETERMINISTIC or NOT DETERMINISTIC clause of the `CREATE FUNCTION` statement. Here, you specify whether the function returns the same value if it is called repeatedly with the same value of the input parameter. If you specify IS DETERMINISTIC, the function always returns the same value from successive invocations of the function with the same values of input parameters. If you specify IS NOT DETERMINISTIC, the function does not return the same value from successive invocations of the function. In the previous example, the function has been declared as DETERMINISTIC. For an example of a NON DETERMINISTIC function, look at the following scenario. Let us say you have written a function GetPrice that picks up the price of a specified product from the product master file, converts it to pounds, and returns the result. Let us also assume that it picks up the current rate of conversion from another file that contains the conversion rates from US dollars to any other currency for all major currencies. This would be an example of a non-deterministic function. This is because the value returned by the GetPrice depends on two variables: the conversion rate from the U.S. dollar to the pound, and the current U.S. price per unit of the product. Both of these values may change dynamically. Therefore, successive calls to the GetPrice function with the same input parameters might produce different results. The default setting for this clause is DETERMINISTIC.
6. This is the NO/READS/MODIFIES/CONTAINS SQL DATA clause of the `CREATE FUNCTION` statement. Here, you specify what kind of SQL statements the function will execute. Refer to *DB2 UDB for AS/400 SQL Reference*, SC41-5612, for detailed description of valid SQL statements for a given clause.
7. This is the EXTERNAL / NO EXTERNAL ACTION clause of the `CREATE FUNCTION` statement. This clause defines whether the function performs any action on external objects. This would be in addition to any processing the function performs on the input parameters that are sent to it. If the function writes/deletes/updates records in files, calls another sub program, or initiates any kind of processing, the EXTERNAL ACTION clause should be specified. In our GetDescription function, the function does not do any processing other than executing a `SELECT` statement. Therefore, NO EXTERNAL ACTION is specified.
8. The body of the SQL function begins here. This is signified by the `BEGIN SQL` statement in the `CREATE FUNCTION` statement above.
9. This is the `SET SQL` statement in the function where the function selects the structured CLOB value to a host variable. The description part of this variable is then extracted by the substring function.
10. Substrings and extracts the description part of the host variable and stores this value in the variable to be returned to the invoking process.

11. Returns the value stored in the return variable to the invoking process

12. The end of the function's body.

4.4.2.1 Passing LOBs and UDTs as parameters

In this section, we discuss a function that accepts a parameter being a distinct type based on a LOB. The function's name is GetDescription. It manipulates data from the PRODUCT_DESCRIPTION column in our test table PRODMAST01. The column stores the description of the product in a structured format. The contents of this column include a short description of the product, the range of sizes for the product, the color of the product, and the best use for the product. The data type of this column is the distinct type PRDDESC. This distinct type is sourced from the base type CLOB(50K). The GetDescription function takes the value of this column as an input, extracts the description part of the column, and returns this value to the calling program. In our example, the data type of the value returned by the function is VARCHAR(1024). Figure 61 shows the `CREATE FUNCTION` statement for this function.

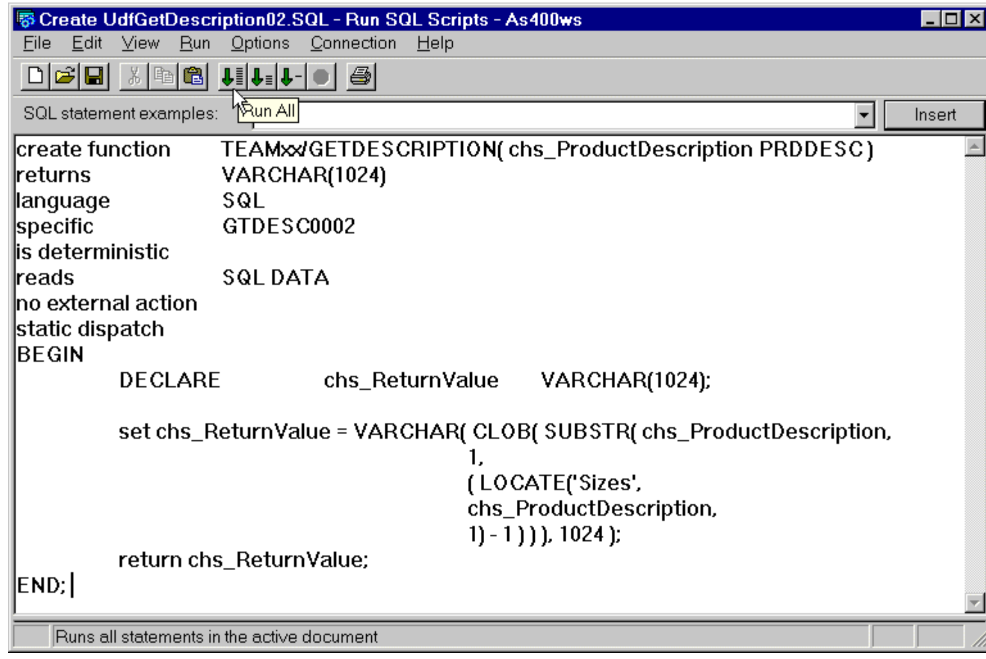


Figure 61. Creating an SQL UDF with UDT parameter

Notice in the `CREATE FUNCTION` statement in Figure 61 that the following clause is included in addition to the ones already described:

```
static dispatch
```

This is the STATIC DISPATCH clause of the `CREATE FUNCTION` statement. When you define an input parameter to a function as UDT, you have to specify this clause. If this clause is not specified, you are not allowed to create the function. The following error message is returned by the database:

```
SQL0104 - Token <END-OF-STATEMENT> was not valid. Valid tokens: STATIC.
```

Figure 62 shows how to use the GetDescription SQL UDF in an SQL statement.

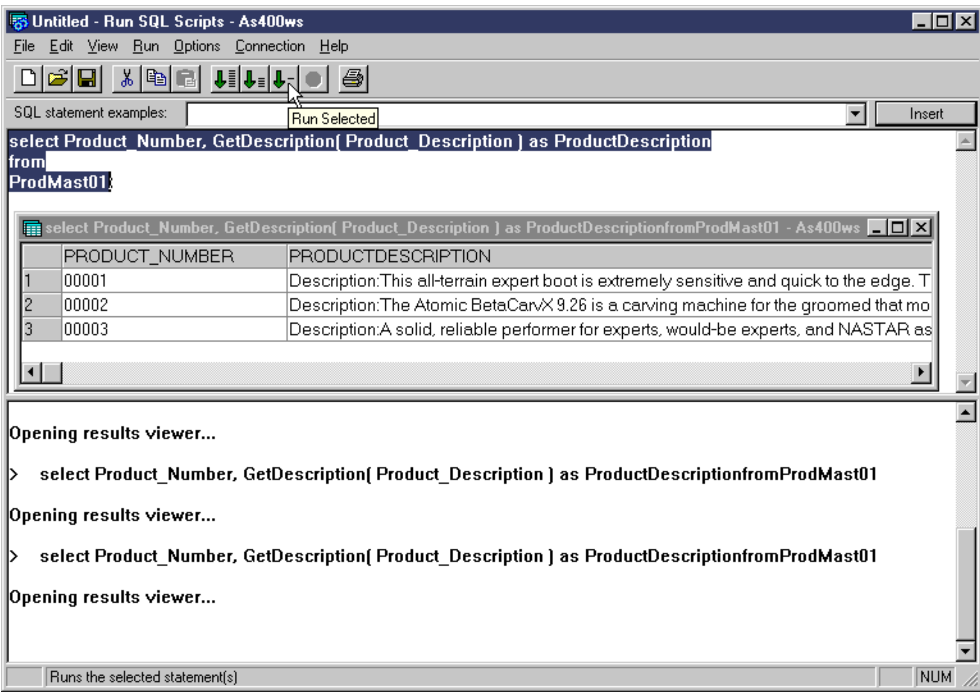


Figure 62. Using SQL UDF GetDescription(PRDDESC) in a query

As seen from the example shown in Figure 61, there is no extra handling involved in using LOBs in SQL UDFs. SQL Functions provide a simple interface for handling LOB parameters, since the system takes care of most of the complexities on how to pass and receive the LOB parameter into the function.

If you want to write an external function that uses LOBs, this is also possible. However, the handling of how to receive the LOB value into the function would have to be taken care of by you in the function program. This would be in addition to any other processing you do as part of the function.

4.4.2.2 Returning LOBs and UDTs as a result

In this section, we describe how an SQL function returns a LOB type value. Our PRODMAST01 test table contains the PRODUCT_DESCRIPTION column that stores a description of the product and the PRODUCT_PICTURE column that stores the picture of the product. It would be useful to have a function that accepts this description as an input and then returns a picture of the product. This way, you can have a list of product descriptions displayed on the screen, and upon selection you could display the picture for that product. The GetPicture function, which implements the outlined logic, accepts three parameters: avalue of type CLOB(50K), a value of type PRDDESC, and a value of type PICTURE. The first parameter is the description for which you require the picture of the product. The second parameter is the value of the PRODUCT_PICTURE column of the PRODMAST01 table. The third parameter is the PRODUCT_PICTURE column of the PRODMAST01 table. The `CREATE FUNCTION` statement is shown in Figure 63 on page 92.

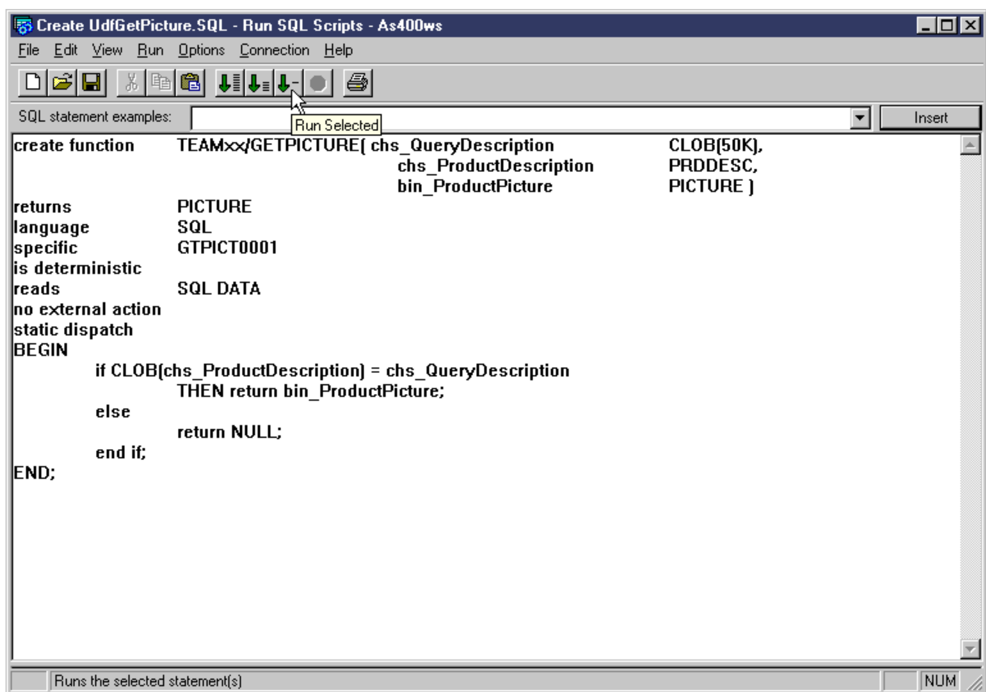


Figure 63. Creating the GetPicture SQL UDF which returns a BLOB as a return value

Notice that the CREATE FUNCTION statement defines the data type of the parameter returned by the function to be the distinct type PICTURE. This data type is based on the base type BLOB(1M). You also see that no other extra clauses are needed in the CREATE FUNCTION statement when you define the return type of the function to be a distinct type.

```
create functionTEAMxx/GETPICTURE( chs_QueryDescription CLOB(50K),
chs_ProductDescription PRDDESC,
bin_ProductPicturePICTURE )
returns PICTURE
languageSQL
specific GTPICT0001
is deterministic
reads SQL DATA
no external action
static dispatch
BEGIN
if CLOB(chs_ProductDescription) = chs_QueryDescription
then return bin_ProductPicture;
else
return NULL;
end if;
END;
```

In the code snippet of the GetPicture function shown here, you see that a comparison is made of the two descriptions sent as input to the program. One of the descriptions is the one for which you want the picture, and the other one is the description value for the product for the row of the table. If they match, the function returns the picture back to the calling program. Otherwise, it returns a NULL value.

Now, let us run this function and check the result. To demonstrate how this function works, we created a stored procedure called RunGetPicture. The stored procedure has a SELECT statement that calls the GetPicture function. The stored procedure accepts one parameter: the product number. Then, the stored procedure retrieves the description of the product corresponding to this product

number. Next, the SELECT statement, which invokes the GetPicture UDF, is executed. The following code sample shows the most important parts of the stored procedure:

```
...
EXEC SQL BEGIN DECLARE SECTION;
     SQL TYPE IS BLOB(1M) bin_ProductPicture;
     SQL TYPE IS CLOB(50K) chs_ProductDescription;
     char chs_ProductNumber[ 5 ];
     char chs_Description[ 1024 ];
EXEC SQL END DECLARE SECTION;
void main( int argc, char **argv )
{
strcpy( chs_ProductNumber, argv[1 ] );
     printf( "The product number - %s\n", chs_ProductNumber );

     EXEC SQL
          select Product_Description into :chs_ProductDescription
          from prodmast01
          where product_number = SRLNUMBER( :chs_ProductNumber );

     EXEC SQL
          DECLARE cur_Picture CURSOR FOR
          Select GetPicture( :chs_ProductDescription, Product_Description,
                             Product_Picture )
          from prodmast01;
.....
while ( sqlca.sqlcode != 100 )
     {
           printf( "\n" );
           if ( bin_ProductPicture.length != 0 )
           {
              printf("Values returned by GetPicture(CLOB,PRDDESC,PICTURE):\n" );
             printf( "The picture length - %d\n", bin_ProductPicture.length );
             printf( "The picture data - %s\n", bin_ProductPicture.data );
           }
           else
           {
             printf("GetPicture (CLOB,PRDDESC,PICTURE) returned NULL\n");
           }
bin_ProductPicture.length = 0;
           strcpy( bin_ProductPicture.data, " " );
EXEC SQL fetch cur_Picture into :bin_ProductPicture;
     }

....
}
```

The stored procedure prints out the length and the contents of the picture returned to it by the BLOB. The full code of the stored procedure is given in Appendix A, “Source code listings” on page 215. The stored procedure was called from the interactive SQL prompt. Figure 64 on page 94 shows the call to the stored procedure from the interactive SQL prompt.

```
Enter SQL Statements

Type SQL statement, press Enter.
     Current connection is to relational database RCHASM23.
===> CALL RUNGETPICTURE( '00001' )




















                                                                       Bottom
F3=Exit   F4=Prompt   F6=Insert line   F9=Retrieve   F10=Copy line
F12=Cancel            F13=Services     F24=More keys
                                        (C) COPYRIGHT IBM CORP. 1982, 1999.
```

Figure 64. Calling the RunGetPicture

The result of the call to the GetPicture is shown in Figure 65.

```
The product number - 00001

  Values returned by GetPicture( CLOB, PRDDESC, PICTURE ):
  The picture length - 26810
  The picture data - åñã  /  {

  The GetPicture function( CLOB, PRDDESC, PICTURE ) returned NULL

  The GetPicture function( CLOB, PRDDESC, PICTURE ) returned NULL
  Press ENTER to end terminal session.











===>

F3=Exit F4=End of File F6=Print F9=Retrieve F17=Top
F18=Bottom  F19=Left   F20=Right F21=User Window
```

Figure 65. The result of the call to the GetPicture SQL UDF

As shown in Figure 65, the length and data is returned only by the first call to the function. The other two calls to the function result in a NULL value. This is because one description matches only one product. If there were multiple products matching the same description, multiple non-null results would have

been returned. Since the data for the picture is in binary format, you cannot display this data on the 5250 terminal. You see some non-printable characters displayed on the screen. However, if you called the function from an interface that is capable of displaying graphics data, you could see the picture displayed on the screen.

4.4.3 Coding external UDFs

External functions are functions coded in one of the High Level Languages (HLL) available on the AS/400 system. Implementing an external function is more difficult than writing an SQL function. However, if you want to do some complex or sophisticated processing, or plan to re-use the code that already exists, the external functions are the best choice for you.

4.4.3.1 Parameter styles in external UDFs

You can specify several different *parameter styles* for an external function. On the external function invocation, DB2 UDB passes a number of parameters to the function in *addition* to those that you provide as input parameters. The number and type of extra parameters passed by DB2 UDB depends on the *parameter style*. You can specify the required parameter style at the time the function is created. DB2 UDB for AS/400 supports four parameter styles:

- SQL
- DB2SQL
- GENERAL
- GENERAL WITH NULLS

The various parameters passed in each of the parameter styles are discussed in this section. Later, we provide examples for each of these parameter styles.

> **Note**
>
> The current sizes for all the arguments supported by the different parameter styles are defined in the *sqludf.h* include file found in QSYSINC library. There are also equivalent include files for RPG and Cobol.

SQL parameter style

The required set of parameters for this parameter style are:

```
ExternalUDF(IN arguments (repeated),
OUT result,
IN argument indicator variables (repeated),
OUT result indicator,
OUT sqlstate,
IN function name,
IN specific name,
OUT diagnostic message)
```

The elements of the parameters are explained in the following list:

- **Arguments:** Input parameters. Passed in from database to UDF.
- **Result:** Result value. Returned from the UDF to database.
- **Argument indicators:** NULL indicator for each argument. If NULL was passed for the corresponding argument, the indicator variable contains -1. If a valid value was passed, the indicator variable contains 0. The function can

test the value of an argument indicator. If the corresponding argument contains NULL or was truncated, it can take corrective action. These are input parameters.

- **Result indicator:** NULL or mapping error indicator for each argument. This variable is examined by the invoking database process to check if the function returned a correct, NULL, or a truncated value. Set this parameter to `-1` to indicate NULL, or `0` to indicate correct return value. This is an output variable.
- **SQL state**: Corresponds to SQLSTATE in SQL. It is defined as CHAR(5). This value is set by the function to signal an error or warning to the database. It has one of the following values:
  - **00000**: No errors
  - **01Hxx**: Warning. It results in SQLCODE +462 from SQL. The last two characters, *xx*, are set by the function and can be anything you like.
  - **38xxx**: Error occurred in UDF. It results in SQL -443. The last three characters, *xx*x, are set by the function and can be anything you like. When you set this error state the database interrupts the execution of the invoking SQL statement. In the Interactive SQL environment, the following message is displayed in this situation:

    ```
    Query cannot be run.  See lower level messages.
    ```

    This is an output parameter.
- **Function name:** A fully qualified function name. The fully qualified function name follows the SQL naming standard. This is an input parameter.
- **Specific name**: The specific name of the function. This is an input parameter.
- **Diagnostic message**: The message text to put into an SQL message. It corresponds to the `sqlstate` setting. When the function signals an error message to the database, it can set this parameter to a customized error message. This text is then embedded inside the second level message for the CPF503E, which is placed in the job log of the job running the SQL statement. Refer to 4.4.3.2, “Coding UDFs using the SQL parameter style” on page 97, for more details. This is an output parameter.

DB2SQL parameter style

All the parameters passed to a function for the SQL paramours style are also passed to a function with the DB2SQL heptameter style. However, DB2SQL parameter style allows *additional* parameters to be passed. The supported set of parameters for this parameter style are:

```
externalUDF(IN arguments (repeated),
OUT result,
IN argument indicator variables (repeated),
OUT result indicator,
OUT sqlstate,
IN function name,
IN specific name,
OUT diagnostic message,
scratchpad,
call type,
dbinfo)
```

The additional parameters, not covered in the previous section, are explained in the following list:

- **Scratchpad:** The scratchpad if the SCRATCHPAD clause was specified in the CREATE FUNCTION statement. This can be used by the function as an area where it can save the results of the last call in between calls to the function. If the length of the scratchpad area required is not specified in the CREATE FUNCTION statement, the system reserves 100 bytes for the function by default. The maximum length that can be reserved for the scratchpad is 16,000,000 bytes. Each invocation of the function will be able to see the results stored by the last function invocation in the scratchpad. On the first call to the function, the contents of the scratchpad are initialized to all zeros. The data can be stored into the scratchpad area by a function only during the processing of a given SQL statement. No function can store data in the scratchpad area between SQL statements. This is an optional input and output parameter.
- **Call type**: A parameter for the type of call if the FINAL CALL was specified on the CREATE FUNCTION statement. This can be one of three values:

  -1 First call to UDF
  0 Normal call to UDF
  1 Final call to UDF

  This parameter is normally used with the SCRATCHPAD parameter. On the first call, the scratchpad area is set up by the function and then used in subsequent normal calls. On the last call to the function, the scratchpad area is cleaned up. This is an optional input parameter.
- **dbinfo**: A parameter for the dbinfo structure if the DBINFO clause is specified on the CREATE FUNCTION statement. Refer to the sqludf.h include found in the QSYSINC library for a detailed definition of this structure.

General parameter style

The supported set of parameters for this parameter style is:

```
externalUDF(IN arguments (repeated))
```

For this parameter style, the result is the return value of the value returning function itself.

Note

The maximum number of parameters allowed in CREATE FUNCTION is 90. For external functions created with PARAMETER STYLE SQL, and for SQL functions, the input and result parameters specified and the implicit parameters for indicators, such as SQLSTATE, function name, specific name, and message text, as well as any optional parameters, are included. The maximum number of parameters is also limited by the maximum number of parameters allowed by the licensed program that is used to compile the external program or service program.

4.4.3.2 Coding UDFs using the SQL parameter style

In this section, we look at examples on how to code external UDFs with the SQL parameter style. We also demonstrate how the parameters that DB2 passes to the function can be used within the function.

Our test PRODMAST01 table contains the PRODUCT_PICTURE column. This column stores a picture of the product. The picture can be stored in this column in one of the widely accepted formats, such as GIF, BMP, JPG, and so forth. The

data type of the column is the PICTURE distinct type which, itself, is based on the base type BLOB(1M). Let us suppose you need to find out how many pictures are stored in this column with a specified file format. To accomplish this task, we implement two functions IsGif and IsBmp, which take the value of the PRODUCT_PICTURE column and determine whether its contents are in GIF or BMP format, respectively. If the contents are in the GIF format, the IsGif function returns 1. Otherwise, it returns 0. If the input to the IsGif function is NULL, it returns NULL. Similarly, the IsBmp function returns 1 if the input is in BMP format. Otherwise, the function returns 0. It returns a NULL if the input is NULL.

Let us examine the CREATE FUNCTION statement for the IsGif function. The numbered sections are further explained in the list that follows:

```
create function      TEAMxx/ISGIF( PICTURE )                          1
returns              INTEGER                                          2
language             C                                                3
specific             ISGIF00001                                       4
no sql                                                                5
no external action                                                    6
static dispatch                                                       7
external name        'TEAMXX/PICTCHECK(fun_CheckPictureType)'         8
parameter style      SQL;                                             9
```

CREATE FUNCTION statement notes

1. Here, you define the name of the function, the input parameters to the function, and their data types. Refer to 4.4.1.1, “Creating sourced UDFs as scalar functions” on page 78, for more information on this.
2. This is the RETURNS clause of the CREATE FUNCTION statement. Refer to 4.4.1.1, “Creating sourced UDFs as scalar functions” on page 78, for more information on this.
3. This is the LANGUAGE clause of the CREATE FUNCTION statement. The LANGUAGE clause specifies what language was used to implement the external UDF. In our case, it is written in ILE C/400. This information helps the database to pass parameters to the UDF in the format required by the programming language. You can write the UDFs in any of the following languages:
   - CL
   - COBOL
   - COBOLLE
   - PLI
   - RPG
   - RPGLE
   - SQL
   - C/C++

   The LANGUAGE clause is optional. If it is not specified, the system tries to retrieve the attribute of the program object specified in the EXTERNAL NAME clause and set the language clause accordingly. If the program object does not exist, or if the attribute is not present, the language is defaulted to ILE C/400.
4. This is the SPECIFIC NAME clause of the CREATE FUNCTION statement. The specific name is checked for uniqueness and entered into the system catalogue.

5. This is the NO / READS / MODIFIES / CONTAINS SQL DATA clause of the CREATE FUNCTION statement. Refer to 4.4.2, “Coding SQL UDFs” on page 85, for more information on this.
6. This is the EXTERNAL / NO EXTERNAL ACTION clause of the CREATE FUNCTION statement. Refer to 4.4.2, “Coding SQL UDFs” on page 85, for more information on this.
7. This is the STATIC DISPATCH clause of the CREATE FUNCTION statement. Refer to 4.4.2.1, “Passing LOBs and UDTs as parameters” on page 90, for more information on this.
8. This is the EXTERNAL NAME clause of the CREATE FUNCTION statement. This is the name of the external program that this function calls when it is invoked by the database. In this example, TEAMXX is the name of the library in which the program resides. PICTCHECK is the name of the service program that is to be executed, and fun_CheckPicture is the name of the ILE C/400 function inside the program that will be called when the function is invoked. The program does not need to exist at the time of the creation of the function, but it must be created before the function is invoked for the first time. This is an optional clause. If it is not specified, the system assumes that the name of the program to be executed is the same as the name of the function.
9. This is the PARAMETER STYLE clause of the CREATE FUNCTION statement. This can be one of four values: SQL, DB2SQL, GENERAL WITH NULLS, or GENERAL. DB2 UDB passes additional parameters, apart from the input arguments defined in the CREATE FUNCTION statement, based on the parameter style specified.

Now let’s examine the external program PICTCHECK referred to in the CREATE FUNCTION statement above. We discuss what parameters DB2 sends to the program and how the program makes use of the parameters. The complete listing of the program is given in A.3, “PictCheck: External UDF” on page 218. This program also calls the fun_CheckHeader function. The source for this function is listed in A.4, “ChkHdr” on page 220. The PICTCHECK is used by both the IsGif and the IsBmp functions discussed earlier in this section. Depending on what function calls the program, its logic checks for the appropriate type of image. This leads to the reuse of the common code in the two functions. Both functions are defined with the parameter style SQL. The following code sample illustrates how a function with parameter style SQL is coded. The numbered areas are further explained in the list that follows:

```
void SQL_API_FN fun_CheckPictureType( BLOB1M   *str_ProductPicture,          1
             SQLUDF_INTEGER *nmi_IsCorrect,                                  2
             SQLUDF_NULLIND *nms_InputNullIndicator01,                       3
             SQLUDF_NULLIND *nms_OutputNullIndicator01,
             SQLUDF_CHAR    sqludf_sqlstate[ SQLUDF_SQLSTATE_LEN + 1 ],   4
             SQLUDF_CHAR    sqludf_fname[ SQLUDF_FQNAME_LEN + 1 ],        5
             SQLUDF_CHAR    sqludf_fspecname[ SQLUDF_SPECNAME_LEN + 1 ], 6
             SQLUDF_CHAR    sqludf_msgtext[ SQLUDF_MSGTEXT_LEN + 1 ] )  4
```

Code sample notes

1. The function named fun_CheckPictureType is the entry point in the ILE C service program. This entry point is referred to in the CREATE FUNCTION statement for the IsGif and the IsBmp functions as follows:

```
external name      'TEAMXX/PICTCHECK(fun_CheckPictureType)'
```

If the reference is to an entry point in a service program, the external name is specified as *lib.pgmname(entrypoint)* or just *pgmname(entrypoint).*

Note the use of the SQL_API_FN constant in the function declaration. This constant makes the function portable to multiple platforms. The value of this constant is set in an include file specific for a given platform, such as Windows NT, OS/2, or AIX. The value of this constant for the AS/400 system is defined in the header file sqlsystm.h found in the source file named H in the QSYSINC library.

The IsGif or the IsBmp functions accept an input parameter of type PICTURE, which is a distinct type based on the base type BLOB(1M). When passed to an external program, a UDT is implicitly cast to its source type. In our case, we defined a structure called BLOB1M inside our program, which serves as a buffer for the picture object. The structure definition is shown here:

```
typedef struct
{
        unsigned long length;
        char          data[ 1 ];
} BLOB1M;
```

The function accepts one input parameter, which is the picture whose format we wish to determine. This is the first parameter to the ILE C function fun_CheckPictureType, which implements the UDF.

2. This is the value returned by the function. In the CREATE FUNCTION statement shown on page 98, it was defined as INTEGER. However, in the ILE C implementation, we defined it to be of type SQLUDF_INTEGER. We used this convention so that the function is portable across DB2 UDB platforms. All the basic data types on the AS/400 system have their counterparts under the DB2 UDB convention. These counterparts are defined in the header file sqludf.h. This file has to be included when you write an external program for a UDF.
3. The next two parameters to the fun_CheckPictureType function are the null input indicators for the input parameter and the return value. Whenever a null value is passed into the function on input, the input null indicator contains -1. If the value is correct, it contains 0. In our program, we check for null input, and if we get a null input, we return a null output. This is shown in the code sample here:

```
if ( *nms_InputNullIndicator01 == -1 )
{
   *nms_OutputNullIndicator01 = -1;
   return;
}
```

If we want to pass a return value back to the database, we set the return variable, which in our case is nmi_IsCorrect, and set the return indicator to 0 as shown in the following code snippet:

```
if ( ( nmi_CompareResult01 == 1 ) || ( nmi_CompareResult02 == 1 ) )
{
   *nmi_IsCorrect = 1;
   *nms_OutputNullIndicator01 = 0;
}
else
{
```

```
   *nmi_IsCorrect = 0;
   *nms_OutputNullIndicator01 = 0;
}
```

4. The next two parameters, sqludf_sqlstate and sqludf_msgtext are used together. The sqludf_sqlstate contains the SQL state. This parameter can be used to signal an error or a warning condition on return to the database. The function can also set the message text parameter to a customized error message. However, the message text can only be set when sqludf_sqlstate is also set. Our program checks whether it was called by either the IsGif or the IsBmp function. If it is neither of these two, the program simply signals an error condition and returns it as in the following code example:

   ```
   *nms_OutputNullIndicator01 = -1;
   strcpy( sqludf_sqlstate, "38501" );
   strcpy( sqludf_msgtext, "Unregistered function" );
   return;
   ```

5. The parameter sqludf_fname contains the fully qualified name of the function that called this program. In our case, either the IsGif or the IsBmp functions can call the program. The program checks which function called it. If it was the IsGif function that made the call, the program checks the picture for GIF picture format; otherwise, it checks for the BMP picture format. This is implemented in the following code snippet:

   ```
   .
   #define GIF_FUNCTION "ISGIF"
   #define BMP_FUNCTION "ISBMP"
   ...
   void SQL_API_FN fun_CheckPictureType( ...... )
   {
   ...
   char       *chr_FunctionResolution;
   ...
   chr_FunctionResolution = strstr( sqludf_fname, GIF_FUNCTION );

   if ( chr_FunctionResolution != NULL )
   {
      ...
   }

   ...
   chr_FunctionResolution = strstr( sqludf_fname, BMP_FUNCTION );

   if ( chr_FunctionResolution != NULL )
   {
       ...
   }
   }
   ```

6. The parameter, sqludf_specname, is the specific name of the function that is passed by the database. Instead of using the function name, you can also use the specific name for comparisons. This is useful since UDFs can be overloaded. You can have more than one UDF with the same name calling the same program. Even if the function names were the same, the specific names would be unique.

As mentioned earlier, the PICTCHECK program was created as a service program. The advantage of this approach is that the service program becomes active (if run in activation group *CALLER) when the function is resolved, therefore minimizing call overhead at IO. We used the following CL commands to compile and bind the PICTCHECK service program:

```
CRTCMOD MODULE(TEAMXX/PICTCHECK) SRCFILE(TEAMXX/QCSRC) DBGVIEW(*SOURCE)
CRTCMOD MODULE(TEAMXX/CHKHDR) SRCFILE(TEAMXX/QCSRC) DBGVIEW(*SOURCE)
```

```
CRTSRVPGM SRVPGM(TEAMXX/PICTCHECK) MODULE(TEAMXX/PICTCHECK TEAMXX/CHKHDR)
EXPORT(*ALL)
```

To invoke the IsGif function, use a SELECT statement, such as the following example:

```
SELECT Product_Number, IsGif( Product_Picture ) from PRODMAST01;
```

The results of the above query are shown in Figure 66.

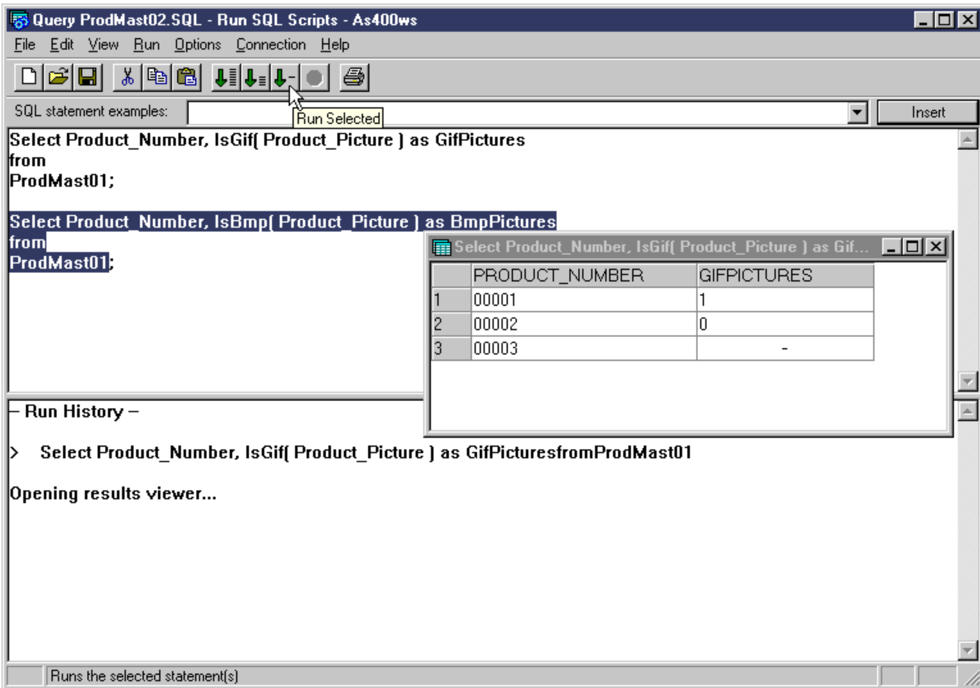


Figure 66. Running the IsGif external UDF with the SQL parameter style

You would, similarly, run the IsBmp function. The output of the IsBmp function is shown in Figure 67.

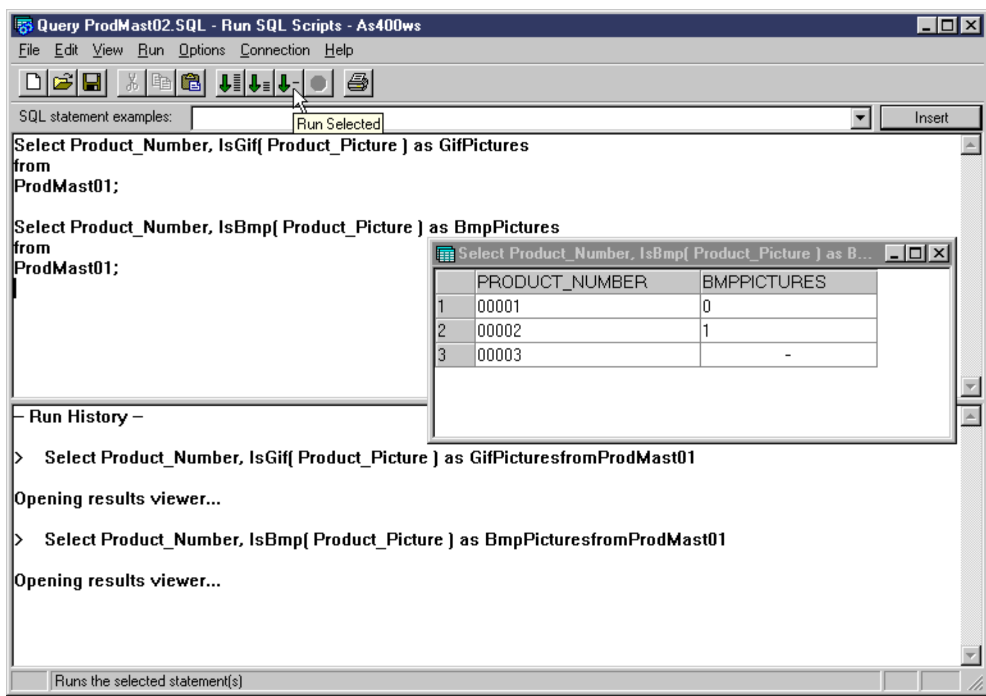


Figure 67. Running the IsBmp external UDF with the SQL parameter style

4.4.3.3 Coding external UDFs using the DB2SQL parameter style

This section shows you how to code external UDFs using the DB2SQL parameter style. You learn how to use the SCRATCHPAD and the FINAL CALL parameters inside the UDF.

The ORDER_TOTAL column in the ORDERHDR table contains the total of the customer's order. The data type of this column is the distinct type MONEY, which is based on the built-in type DECIMAL(11,2). Suppose you wanted to find out the second, third, or the fifth best order by order total from the ORDERHDR table. One of the approaches might involve writing a program that calculates, for instance, the third best order total and writes it to a file. Then, your application would need to access this file and read the data from the file. We believe it would be much better to have a UDF that does this processing. The UDF could then be used inside a `SELECT` statement in any AS/400 interface that supports SQL. To accomplish this task, we coded an external UDF called Rating. The function takes in two parameters: a value of type MONEY from the ORDER_TOTAL column and an INTEGER, which specifies which rating you want to retrieve. The function scans the ORDER_TOTAL column and returns for each row the Nth best order total where N is the INTEGER that you specified. The `CREATE FUNCTION` statement for the function is shown in Figure 68 on page 104.

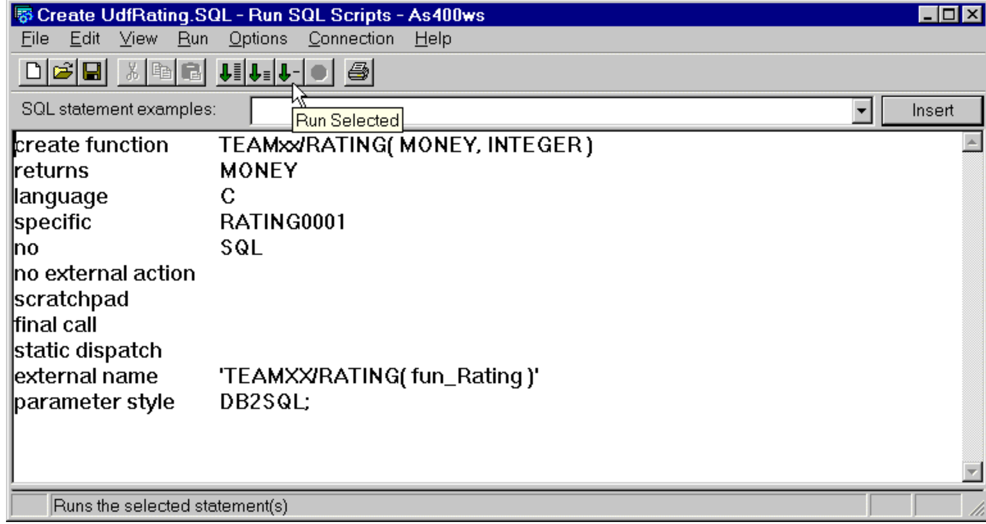


Figure 68. Creating the rating UDF with the DB2SQL parameter style

Let us examine the CREATE FUNCTION statement. The CREATE FUNCTION statement is presented here. The numbered areas are explained in the list that follows:

```
create function      TEAMxx/RATING( MONEY, INTEGER )
returns              MONEY
language             C
specific             RATING0001
no                   SQL
no external action
scratchpad                                                    1
final call                                                    2
static dispatch
external name     'TEAMXX/RATING( fun_Rating )'
parameter style   DB2SQL;
```

CREATE FUNCTION statement notes

1. This is the SCRATCHPAD clause of the CREATE FUNCTION statement. When you specify this clause, DB2 passes a scratchpad area to the function where the function can store results of the last call, and it will be available to the function the next time it is called. You can specify the required length for the scratchpad area if you want. This can be a maximum of 16,000,000 bytes. If not specified, system defaults the length of the scratch pad area to 100 bytes. In our example above, we did not specify the length of the scratchpad area. Therefore, the system will reserve 100 bytes of memory for scratchpad area and send the address of this area to the function program.

2. This is the FINAL CALL clause of the CREATE FUNCTION statement. When this clause is specified, DB2 UDB sends the type of call to the function every time the function is invoked. The value passed can be one of three values: 1 if this is the first call to the function, 0 if this is neither the first nor the last call to the function, and 1 if this is the last call to the function. This parameter is normally used along with the SCRATCHPAD clause. On the first call, the function sets up the scratchpad area. On a regular call, it accesses the scratchpad area. On the last call, the function cleans up the scratchpad area.

Now let’s examine the contents of the Rating program. The complete listing of the source code is given in Appendix A.6, “Rating: External UDF using SCRATCHPAD” on page 221. We start with the function declaration for a function with the DB2SQL parameter style, concentrating on the most important parameters. The numbered areas are explained in the list that follows:

```
void SQL_API_FN fun_Rating( decimal( 11, 2 )  *nmpd_InputMoneyValue,
SQLUDF_INTEGER*nml_InputRequiredRank,
decimal( 11, 2 )*nmpd_OutputMoneyValue,
SQLUDF_NULLIND*nms_InputNullIndicator01,
SQLUDF_NULLIND*nms_InputNullIndicator02,
SQLUDF_NULLIND*nms_OutputNullIndicator01,
SQLUDF_CHARsqludf_sqlstate[ SQLUDF_SQLSTATE_LEN + 1 ], 1
SQLUDF_CHARsqludf_fname[ SQLUDF_FQNAME_LEN + 1 ],
SQLUDF_CHARsqludf_fspecname[ SQLUDF_SPECNAME_LEN + 1 ],
SQLUDF_CHARsqludf_msgtext[ SQLUDF_MSGTEXT_LEN + 1 ],   1
SQLUDF_SCRATCHPAD*sqludf_scratchpad,                          2
SQLUDF_CALL_TYPE*sqludf_call_type )                           3
```

Code sample notes

1. The sqludf_sqlstate is set by the function to indicate an error condition to the database on return from the function. In our function, we set this parameter if the required rank parameter is either null or less than zero. This is shown in the following code snippet:

   ```
   if ( ( *nms_InputNullIndicator02 != 0 ) || ( *nml_InputRequiredRank < 0 ) )
   {
      strcpy( sqludf_sqlstate, "38601" );
      strcpy( sqludf_msgtext, "Incorrect rank value specified" );
      *nms_OutputNullIndicator01 = -1;
      return;
   }
   ```

   The function can also pass a custom message back to the database by setting the message text parameter sqludf_msgtext.

2. When a function is created as a scratchpad function, the database provides the function with a 100 byte scratchpad area. The function can store data in this area that it needs to preserve between function calls. Each invocation of the function can see the data stored by the last invocation of the function. The data in the scratchpad is stored only during the processing of a given SQL statement and not between SQL statements. The function is passed a pointer to the scratchpad area called sqludf_scratchpad. The scratchpad is initialized to zeros before the first call to the function. The following code snippets show how the scratchpad is implemented. First, we define our internal structure, called str_ScratchPad, that helps us keep track of different values stored in the ORDER_TOTAL column:

   ```
   typedef struct
   {
      decimal( 11, 2 ) *nmpd_LargeValue ;
      long             nml_RequiredRating;
      long             nml_ValuesStored;
   } str_ScratchPad;
   .
   .
   ..
   str_ScratchPad   *str_SPad;
   str_ScratchPad   **ptr_AlignmentPointer;
   .
   .
   ..
   /* Get the address of the scratchpad buffer passed by the DB2 UDB and align the pointer for
   the internal scratchpad structure at the 16 byte boundary   */
   ptr_AlignmentPointer = ( ( str_ScratchPad ** )( sqludf_scratchpad ) ) + 1;
   str_SPad = ( str_ScratchPad * ) ptr_AlignmentPointer;
   ```

In the previous code snippet, you see that a structure, called str_ScratchPad, has been declared. The variable, nmpd_LargeValue, is an array of packed decimals that is used to keep the list of values encountered so far. The variable, nml_RequiredRating, stores the rank that you wish to retrieve. The variable, nml_ValuesStored, stores the number of values stored so far in the packed decimal array. We declare a pointer to this structure called str_SPad. The scratchpad that is passed to the program itself is a structure of two elements. The following snippet gives the definition of the scratchpad structure as it is defined in the include file sqludf.h:

```
SQL_STRUCTURE  sqludf_scratchpad
{
   unsigned long  length;        /* length of scratchpad data */
   char           data[SQLUDF_SCRATCHPAD_LEN]; /* scratchpad data, init.
                                                   to all \0 */
};
```

In this program, you see that the data element of the scratchpad structure is cast to the str_scratchPad structure. In other words, we use the data element of the sqludf_scratchpad structure as a memory buffer for our internal str_ScratchPad structure. The method of casting, such as the one shown above, is used to align the str_SPad pointer on a 16-byte boundary. The AS/400 system requires that the memory addresses be placed on the 16-byte boundaries. If your code fails to align addresses properly, an exception is thrown at the run time, and the application is terminated.

In the following code snippet, the scratchpad area that was sent to the function by the database is being put to work. The largest numbers are moved to the top of the array, the smaller ones follow them, and the required rating is then returned to the database from the array. This processing is performed on every invocation of the function:

```
/* Check for regular function call */
if ( *nms_InputNullIndicator01 == 0 )
{
   /* Set the lowest value variable */
   nmpd_LowestValue = *nmpd_InputMoneyValue;

   for ( nmi_Counter = 0; nmi_Counter < str_SPad->nml_ValuesStored; nmi_Counter++ )
   {
       /* Exchange  if the current lowest value is higher than the stored lowest */
       /* value */
       if ( str_SPad->nmpd_LargeValue[ nmi_Counter ] < nmpd_LowestValue )
       {
          nmpd_Temp = nmpd_LowestValue;
          nmpd_LowestValue = str_SPad->nmpd_LargeValue[ nmi_Counter ];
          str_SPad->nmpd_LargeValue[ nmi_Counter ] = nmpd_Temp;
       }

       /* Array not full then add the next element */
      if ( str_SPad->nml_ValuesStored < str_SPad->nml_RequiredRating )
       {
         str_SPad->nml_ValuesStored++;
          str_SPad->nmpd_LargeValue[ str_SPad->nml_ValuesStored - 1 ] =
                                                                nmpd_LowestValue;
       }

      /* return NULL if required ranking not in the array*/
     if ( str_SPad->nml_ValuesStored < str_SPad->nml_RequiredRating )
      {
        *nms_OutputNullIndicator01 = -1;
        return;
      }
      /* Otherwise return the required ranking */
      else
      {
          *nmpd_OutputMoneyValue = str_SPad->nmpd_LargeValue[
```

```
                                    str_SPad->nml_RequiredRating - 1 ];
        *nms_OutputNullIndicator01 = 0;
        return;
      }
   }
}
```

3. The scratchpad parameter is normally used in conjunction with the FINAL CALL parameter. This is the last parameter in the `fun_Rating` function and is named `sqludf_call_type`. This parameter tells the function whether this is the first call, the last call, or a regular call. The following code snippets show how to use this parameter:

   ```
   if ( *sqludf_call_type == -1 )
   {
   if ( ( *nms_InputNullIndicator02 != 0 ) || ( *nml_InputRequiredRank < 0 ) )
   {
   strcpy( sqludf_sqlstate, "38601" );
   strcpy( sqludf_msgtext, "Incorrect rank value specified" );
   *nms_OutputNullIndicator01 = -1;
   return;
   }

   str_SPad->nml_RequiredRating = *nml_InputRequiredRank;
   str_SPad->nml_ValuesStored  = 0;
   nml_Temp = *nml_InputRequiredRank * sizeof( decimal( 11, 2 ) );
   str_SPad->nmpd_LargeValue = ( decimal( 11, 2 ) * )malloc( *nml_InputRequiredRank *
                                                      sizeof( decimal( 11, 2 ) ) );
   }
   ```

   First, a check is made to see if this is the first call. If so, at this point in time, the function must perform required initialization tasks. In our case, the program dynamically allocates the memory for storing the required number of values in the scratchpad using the ILE C/400 malloc function. This allocation is not done on the subsequent calls to the function that are considered to be regular calls.

   In the following snippet, you see that we also check if this is the final call:

   ```
   if ( *sqludf_call_type == 1 )
   {
      free( str_SPad->nmpd_LargeValue );
   }
   ```

   At this point in time, the function must perform any cleanup tasks that need to be performed. In our case, we allocated a piece of memory for our scratchpad using the malloc ILE C/400 function. This piece of memory needs to be freed. This is done by the `free` statement.

To invoke this function, you could use a `SELECT` statement, such as the one shown here:

```
SELECT Max( Decimal( Rating( Order_Total, 2) ) )from OrderHdr
```

In the `SELECT` statement above, you ask the function to calculate the second best order total in the ORDERHDR table. Note the usage of the MAX built-in function. Our rating function is invoked for each row of the ORDERHDR table, and it produces a result for each row. In our example, the function shows the second best result for all the rows retrieved *so far*. We want to calculate the second best rating for *all rows* in the table, which explains the need for a MAX function. The result of the above query is shown in Figure 69 on page 108.

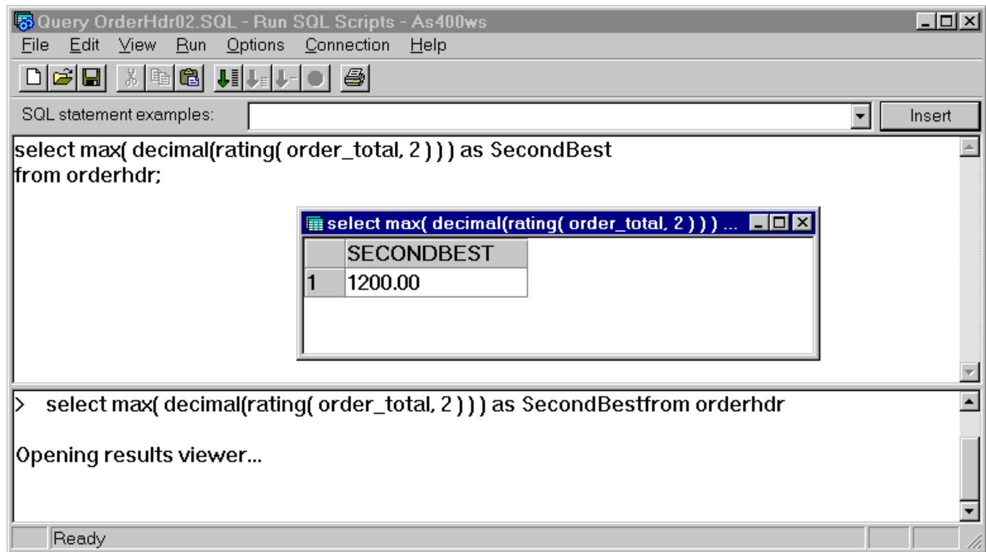


Figure 69. Using the rating external function with DB2SQL parameter style

A more complex query can be given to find the number or the name of the customer with the second best order total. Figure 70 shows a sample query that does this.

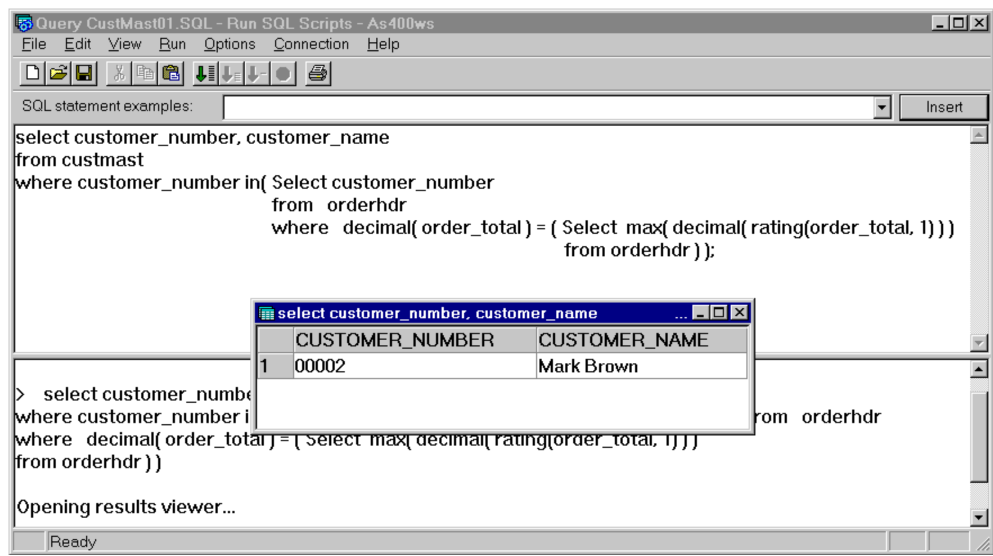


Figure 70. Finding the number and name of the customer using the rating function

4.5 Function resolution and parameter promotion in UDFs

In this section, we demonstrate function resolution and parameter promotion. For concepts of function resolution, parameter promotion, and overloading, refer to 4.3, “Resolving UDF” on page 71.

4.5.1 An example of function resolution in UDFs

Consider the following situation. Say there is a function, GetDescription, which accepts one parameter: A product number that is CHAR(5) and returns the product description as a VARCHAR(1024). This function operates on the

PRODUCT_NUMBER column of the PRODMAST04 table. It returns the contents of the PRODUCT_DESCRIPTION column, which is of data type CLOB(50K). Now, another table, PRODMAST01, is created with the same columns and data as PRODMAST04. Here, the columns PRODUCT_NUMBER and PRODUCT_DESCRIPTION are based on distinct types SRLNUMBER and PRDDESC, respectively. The SRLNUMBER distinct type is based on the built-in type CHAR(5), and the PRDDESC column is based on the built-in type CLOB(50K). Now, you execute the following query:

```
select Product_Number, GetDescription( Product_Number ) as Product_Description
from ProdMast04;
```

The GetDescription(CHAR5) function executes correctly. This is shown in Figure 71.

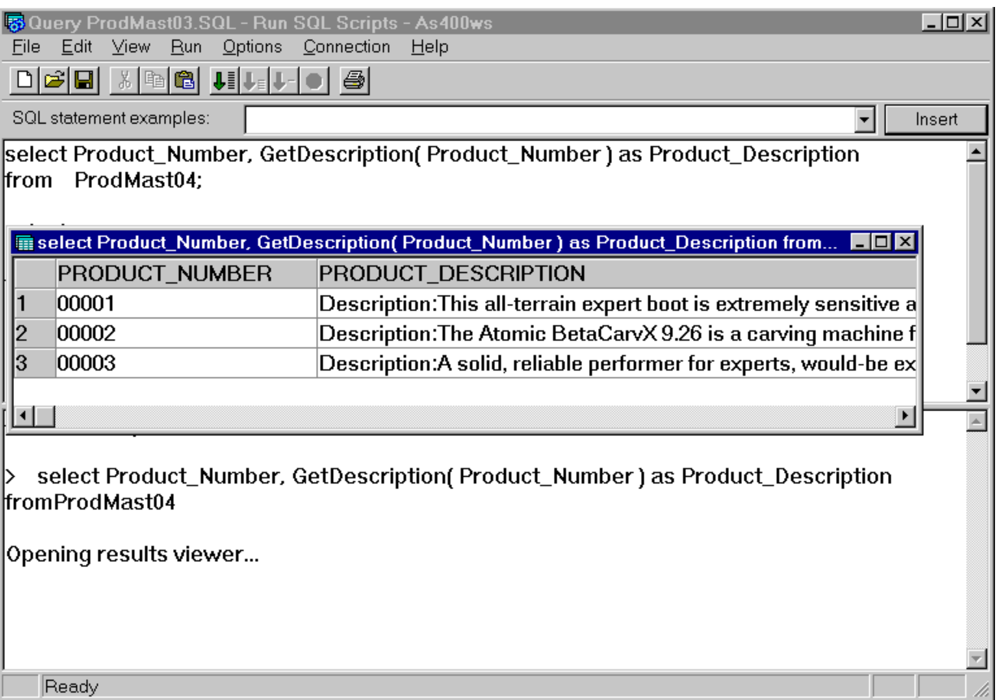


Figure 71. Executing the GetDescription (CHAR(5)) function

The system searched for a function called GetDescription that would accept an input parameter of data type CHAR(5) using the function selection algorithm described in 4.3.4, “The function selection algorithm” on page 76. It found one function that exactly matched the criteria. If there had been no function called GetDescription, accepting a CHAR(5) as input parameter, the system would have searched for the next best alternative: afunction called GetDescription, which accepts a VARCHAR as an input parameter. See 4.3.3, “Parameter matching and promotion” on page 74, for details.

Now, we try to execute the same query on the PRODMAST01 table. This time the query fails. Figure 72 on page 110 shows the result of the query.

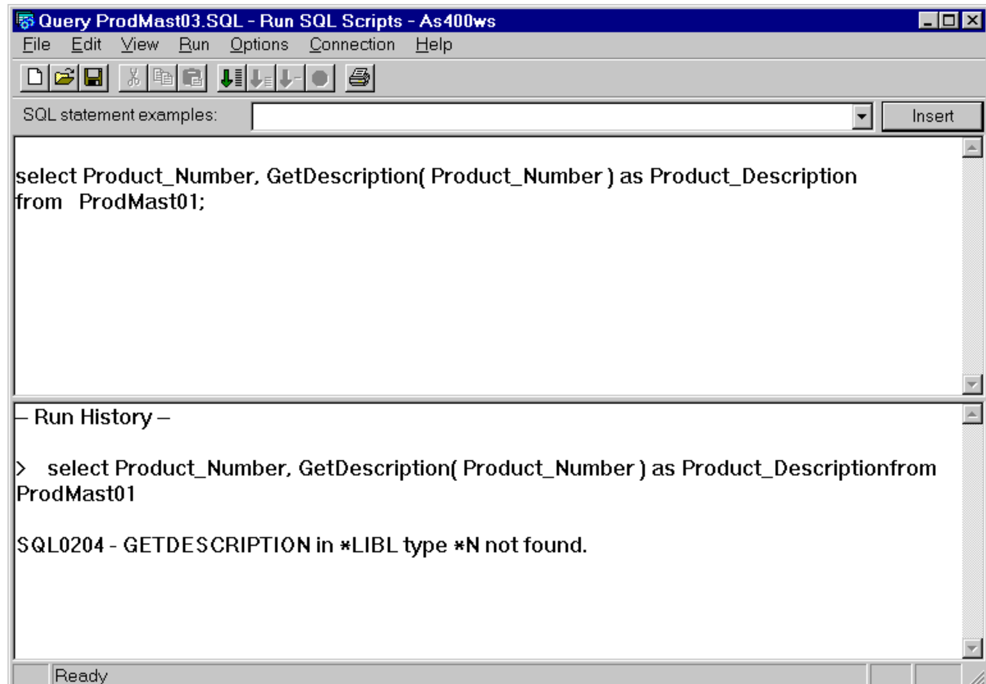


Figure 72. The query fails when it is run over the Prodmast01 table

The query fails because the system could not find a function called GetDescription that accepts an input parameter of data type SRLNUMBER. Then, the data type precedence is checked to see if the parameter can be promoted to other data types. In this case, since the data type of the parameter is a distinct type, it cannot be promoted to anything other than itself. Since the parameter is not promotable, the system returns the following message:

```
SQL0204 - GETDESCRIPTION in *LIBL type *N not found.
```

To solve this problem, we need to overload the GetDescription function. Figure 73 shows the `CREATE FUNCTION` statement that we used to create a GetDescription function that accepts the SRLNUMBER distinct type as an input parameter.

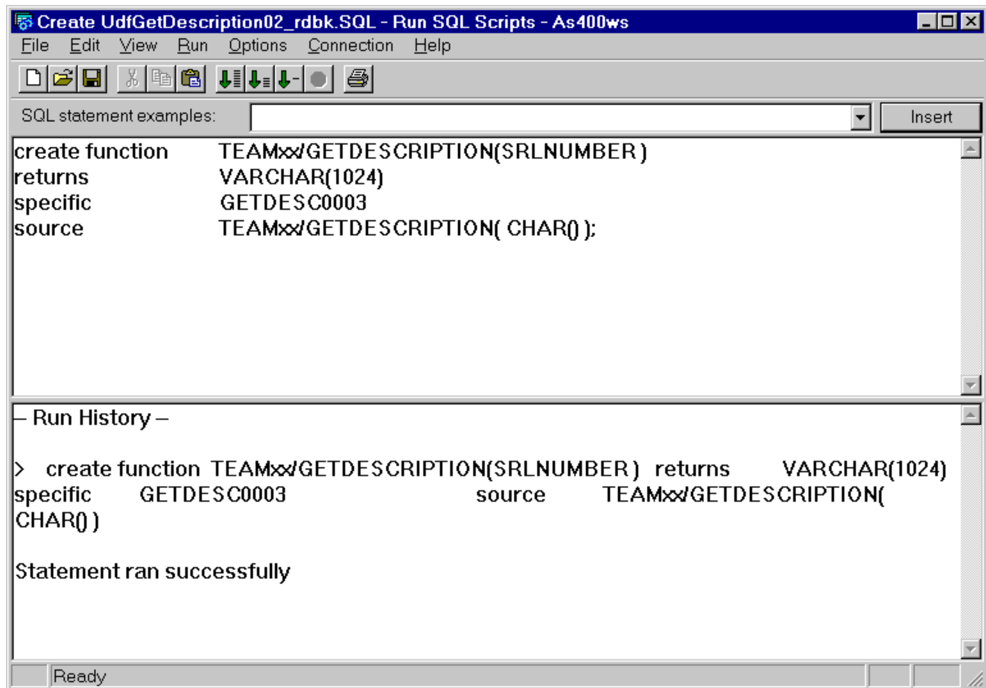


Figure 73. Creating the GetDescrption(SRLNUMBER) sourced UDF

Note that, instead of implementing the function from scratch, we reuse the existing implementation of the GetDesciption(char()) function.

After creating the function, we run the query again. This time it works. The results of the query are shown in Figure 74.

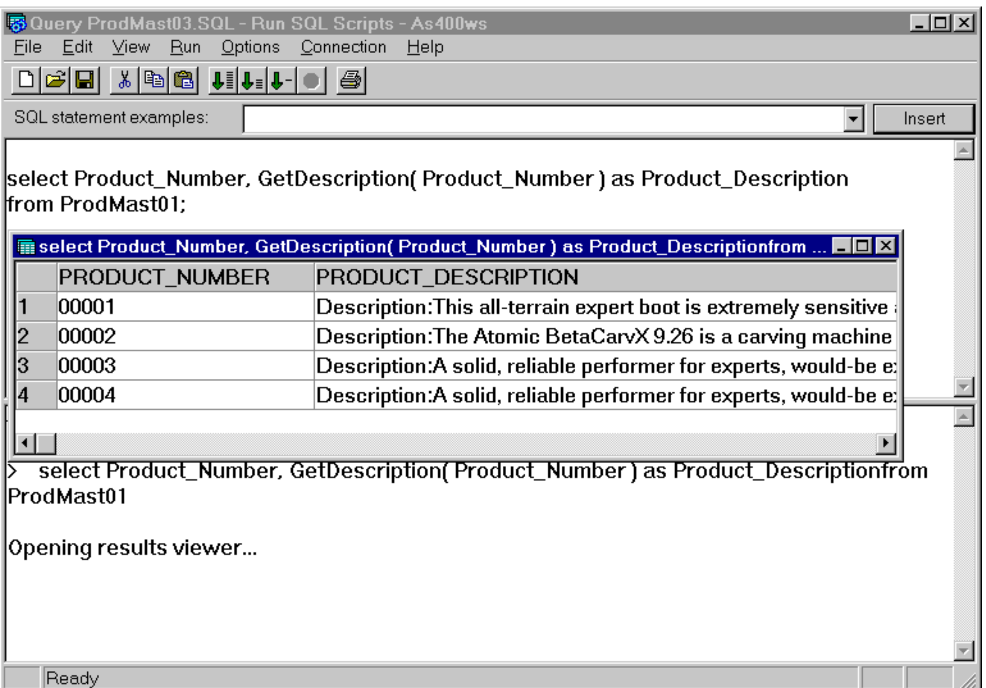


Figure 74. Running the GetDescription(SRLNUMBER) UDF

This time, the system was able to find a function called GetDescription, which accepts SRLNUMBER distinct type as a parameter. The function is then executed.

4.5.2 An example of parameter promotion in UDF

In this section, we will show you an example of parameter promotion. Consider the following scenario: The PRODMAST04 table has a PRODUCT_DESCRIPTION and a PRODUCT_NUMBER column. The data type of the PRODUCT_NUMBER column is CHAR(5), and the data type of the PRODUCT_DESCRIPTION column is CLOB(50K). The PRODUCT_DESCRIPTION column is a CLOB containing the description of the product, the size range of the product, and the color of the product. Suppose we wish to extract the size range from this column. We code a UDF, called GetSize, which accepts the product number and returns the size of the product. Let's assume, for illustration purposes, that the type of the product number parameter is CLOB(50K). Please note that the actual data type of the PRODUCT_NUMBER column in the table is CHAR(5). The `CREATE FUNCTION` statement for the function is shown in Figure 75.

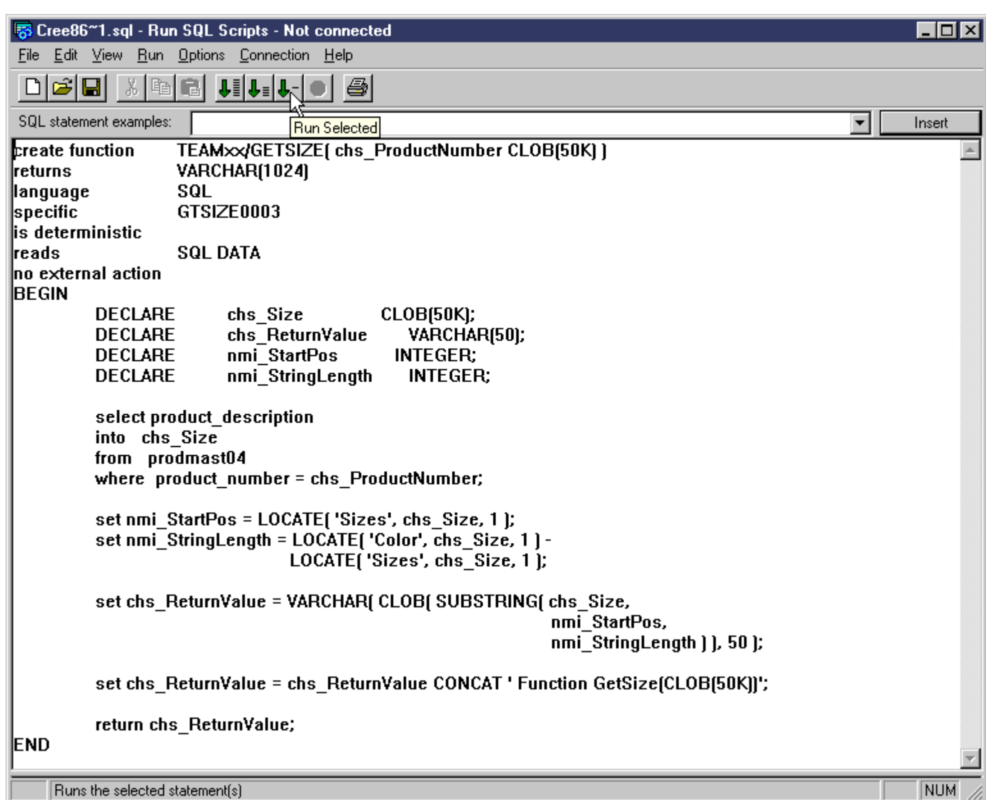


Figure 75. The GetSize(CLOB(50K))SQL UDF

The data type of the value returned by the function is VARCHAR(1024). Note that we concatenate the character constant 'Function GetSize(CLOB(50K))' with the return variable chs_ReturnValue. After the function was successfully created, we used it in a `SELECT` statement, such as the one shown here:

```
select product_number, GetSize( product_number ) as SizeRange from prodmast04
```

The function GetSize(CLOB(50K)) is executed by the system. This is shown in Figure 76.

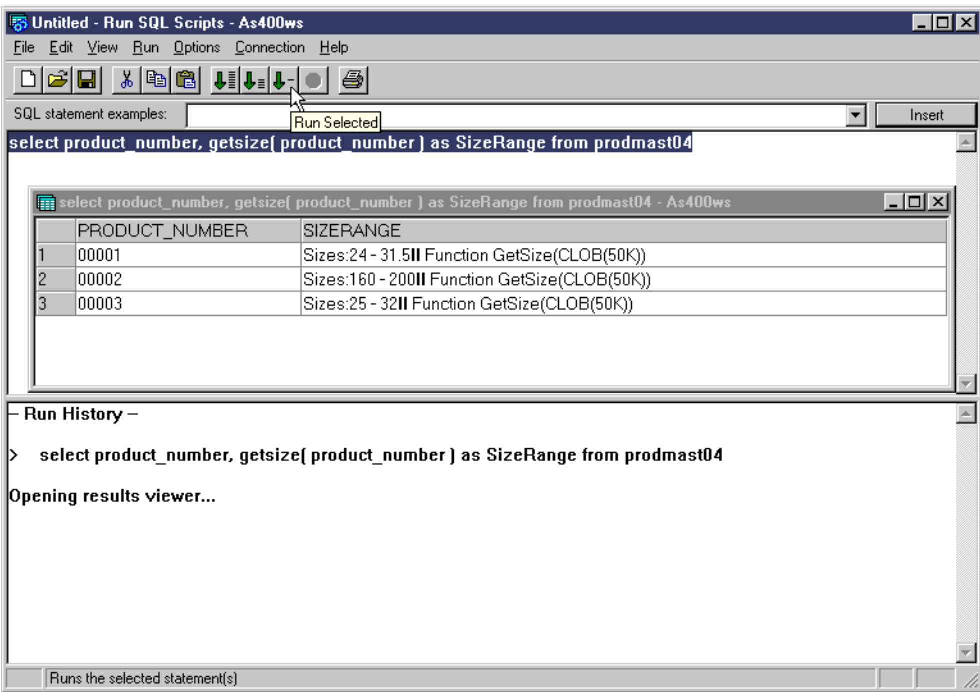


Figure 76. Running the GetSize(CLOB(50K) function

Notice that the text 'Function GetSize(CLOB(50K))' appears as part of the size range column. The input to the GetSize function was a value of type CHAR(5). The GetSize(CLOB(50K)) function executed because of parameter promotion. The system searches the system catalog for all functions named GetSize that are located in the library list of the job executing the SELECT statement. A list of all such functions is compiled. Then, all selected functions that have more than one input parameter are eliminated. Now, for the remaining functions, a search is made for a function GetSize that accepts CHAR(5). The system finds no such function. The function's parameter is now promoted to the next level. This is done by looking up the precedence list, as shown in Table 9 on page 76, and finding out which is the next datatype in the hierarchy for the CHAR data type. In our case, it is the VARCHAR data type. The product_number value, which we supplied to the GetSize function, is now cast to a VARCHAR data type, and the list is scanned to check for a function GetSize that accepts a VARCHAR as an input variable. Again, the system finds no such function and the precedence list is checked again to find the next data type higher in the hierarchy. In our example, it is CLOB. The product_number value is now cast to a CLOB, and the list of functions is again scanned to check for a function GetSize, which accepts a CLOB as an input parameter. This time the system finds the GetSize(CLOB(50K)) function. Therefore, this function is currently the best fit for the function referenced in the SELECT statement. Therefore, this function is executed.

Now, let's create another GetSize function. This time the input parameter is VARCHAR(5). The CREATE FUNCTION statement is shown in Figure 77 on page 114. Notice that here the character constant 'Function GetSize(VARCHAR(5))' is concatenated to the end of the return variable chs_ReturnValue.

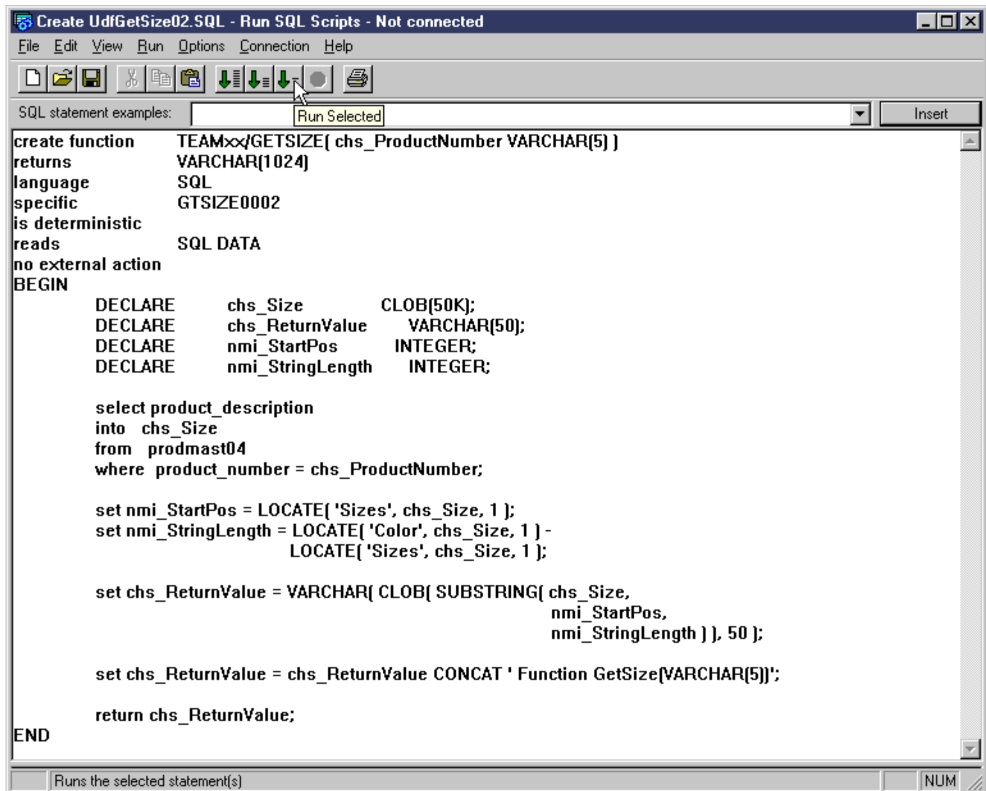


Figure 77. Creating the GetSize(VARCHAR(5))SQL UDF

Now, we run our query again. This time, function GetSize(VARCHAR(5)) is executed. The product_number value that we supplied to the GetSize function is now cast to a VARCHAR data type, and the list of selected GetSize functions is scanned to check for a function GetSize, which accepts a VARCHAR as an input variable. This time, the system finds the function with the signature GetSize (VARCHAR(5)). The function GetSize(VARCHAR(5)) is the best match for the function called in the SELECT statement. Figure 78 shows the result of the query.

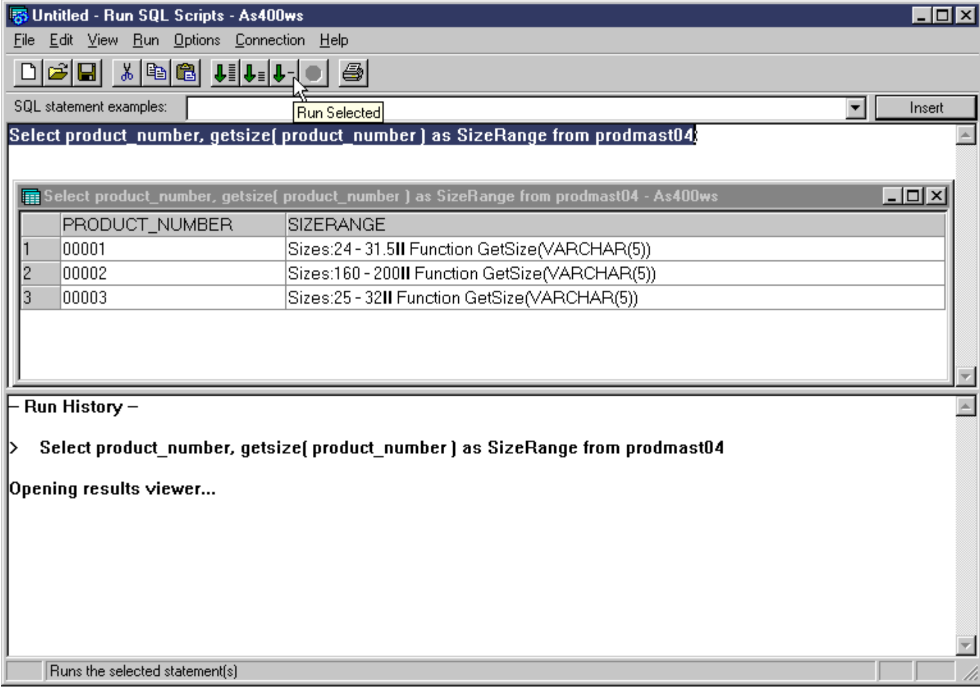


Figure 78. Running the GetSize(VARCHAR(5)) SQL UDF

Let's now create a third GetSize function. This time, the data type of the input parameter is CHAR(5). Figure 79 shows the `CREATE FUNCTION` statement.

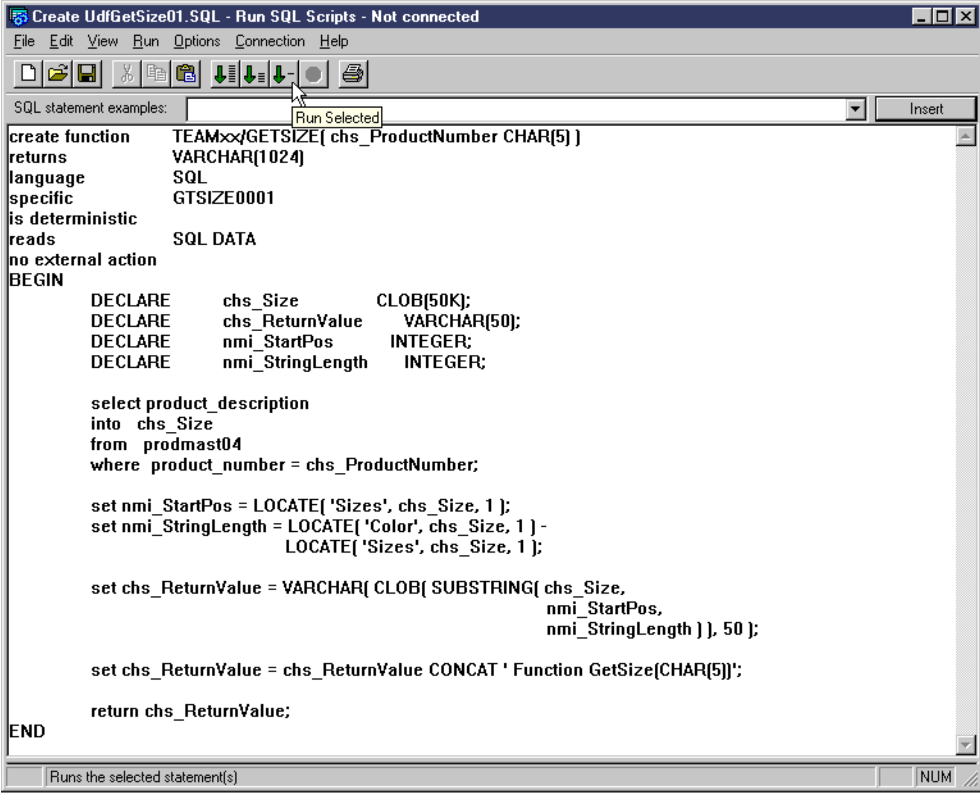


Figure 79. Creating the GetSize(CHAR(5)) SQL UDF

Again, the same query is run. This time, the system selects the function GetSize(CHAR(5)) to be executed because it constitutes an exact match for the function called in the SELECT statement. Figure 80 shows the results of the query.

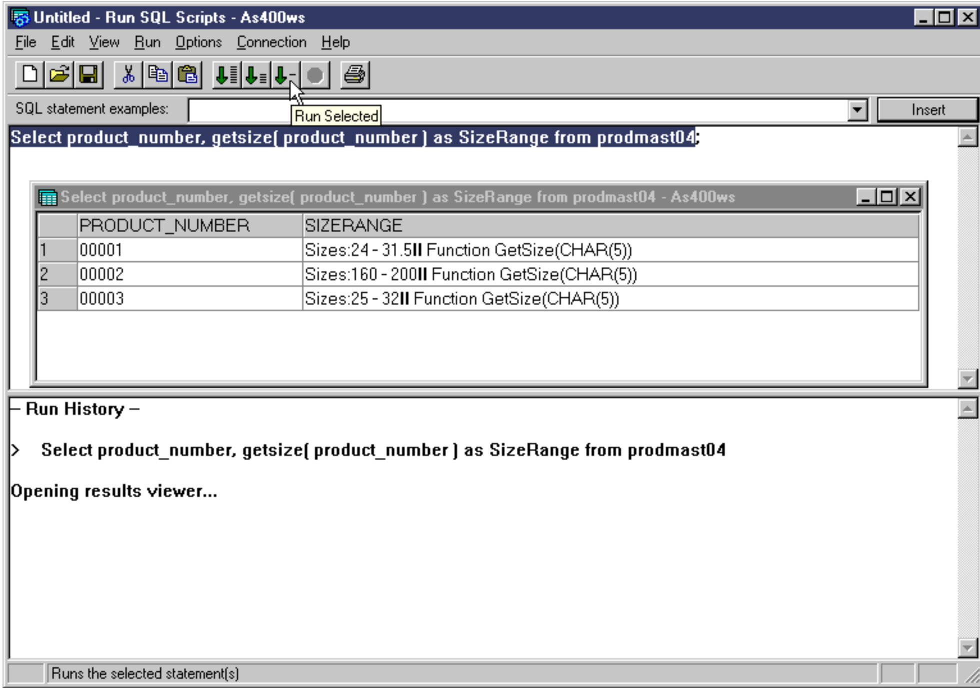


Figure 80. Running the GetSize(CHAR(5)) function.

4.6 The system catalog for UDFs

The database manager provides a number of data dictionary facilities that can be used to keep track of User Defined Functions. In this section we see how to view UDF information using the SYSROUTINES catalog, the SYSPARAMS catalog, and the SYSFUNCS view.

4.6.1 SYSROUTINES catalog

User Defined Functions are stored in the SYSROUTINES catalog. Refer to *DB2 UDB for AS/400 SQL Reference*, SC41-5612, for the detailed description of the catalog views.

Note

The SYSROUTINES catalog contains details for both User Defined Functions and stored procedures. When you want to work only with UDFs you can use a view called SYSFUNCS. This view was created over the SYSROUTINES catalog with the following SQL statement:

```
create view qsys2/sysfuncs as select * from qsys2/sysroutine
where routine_type='FUNCTION'
```

The following SQL statement displays SYSROUTINES information on User Defined Functions in our test TEAMXX library:

```
select * from sysroutines where routine_schema = 'TEAMXX' and routine_type =
'FUNCTION';
```

If we run this statement using the Operations Navigator Run SQL Scripts window, the query results viewer displays UDFs details as shown in Figure 81.

select * from sysroutines where routine_schema = 'TEAMXX' and routine_type = 'FUNCTION' - As400ws

	SPECIFIC_SCHEMA	SPECIFIC_NAME	ROUTINE_SCHEMA	ROUTINE_NAME
1	TEAMXX	CLOB	TEAMXX	CLOB
2	TEAMXX	DECIMAL	TEAMXX	DECIMAL
3	TEAMXX	GTSIZE0003	TEAMXX	GETSIZE
4	TEAMXX	GTSIZE0001	TEAMXX	GETSIZE
5	TEAMXX	BLOB	TEAMXX	BLOB
6	TEAMXX	CHARACTER	TEAMXX	CHARACTER
7	TEAMXX	PLUS00001	TEAMXX	+
8	TEAMXX	LOCATE0001	TEAMXX	LOCATE
9	TEAMXX	RATING0001	TEAMXX	RATING
10	TEAMXX	PRDDESC	TEAMXX	PRDDESC
11	TEAMXX	NUMERIC	TEAMXX	NUMERIC
12	TEAMXX	PICTURE	TEAMXX	PICTURE
13	TEAMXX	SRLNU00001	TEAMXX	SRLNUMBER
14	TEAMXX	GTDSC00001	TEAMXX	GETDESCRIPTION
15	TEAMXX	MONEY	TEAMXX	MONEY
16	TEAMXX	SRLNUMBER	TEAMXX	SRLNUMBER

Figure 81. Content of SYSROUTINES catalog

Note that our catalog query shows both user created UDFs, as well as system generated cast functions needed for the UDT implementation. If you want to select only non-cast UDFs, try the following query:

```
select * from sysfuncs where routine_schema = 'TEAMXX'
and is_user_defined_cast = 'NO'
```

4.6.2 SYSPARMS catalog

The SYSPARMS catalog contains one row for each parameter of an UDF created by the `CREATE FUNCTION` statement. Refer to *DB2 UDB for AS/400 SQL Reference*, SC41-5612, for the detailed description of the catalog views.

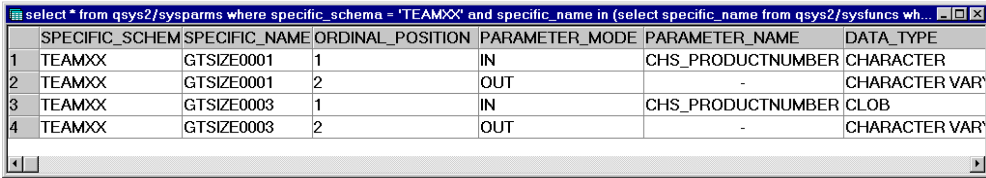
Note

The SYSPARMS catalog contains parameter detail for both User Defined Functions and stored procedures.

Let’s suppose you want to retrieve the parameter details for all instances of the GETSIZE function located in the TEAMXX library. The following SQL statement can be run to display this information:

```
select * from qsys2/sysparms where specific_schema = 'TEAMXX' and specific_name
in (select specific_name from qsys2/sysfuncs where specific_schema = 'TEAMXX'
and routine_name = 'GETSIZE');
```

Note that, due to function overloading, the TEAMXX library can contain several functions with the same routine name. Running this query produced the results shown in Figure 82 on page 118.


select * from qsys2/sysparms where specific_schema = 'TEAMXX' and specific_name in (select specific_name from qsys2/sysfuncs wh...

	SPECIFIC_SCHEM	SPECIFIC_NAME	ORDINAL_POSITION	PARAMETER_MODE	PARAMETER_NAME	DATA_TYPE
1	TEAMXX	GTSIZE0001	1	IN	CHS_PRODUCTNUMBER	CHARACTER
2	TEAMXX	GTSIZE0001	2	OUT	-	CHARACTER VAR'
3	TEAMXX	GTSIZE0003	1	IN	CHS_PRODUCTNUMBER	CLOB
4	TEAMXX	GTSIZE0003	2	OUT	-	CHARACTER VAR'

Figure 82. UDF parameter details in SYSPARMS catalog

There are two instances of the GETSIZE function in the TEAMXX library. Their signatures differ since they accept an input parameter of type CHARACTER or CLOB, respectively. Note, also, that the result of a function is stored in the SYSPARMS catalog as an OUTPUT parameter.

4.7 Dropping UDFs

To drop an UDF using the SQL interface, use the `DROP FUNCTION` statement. The `DROP FUNCTION` statement references the function by:

- **Name**: For example, DROP FUNCTION myUDF. This is only valid if exactly one function of that name exists in that library. Otherwise, SQLSTATE 42854 ('More than one found') or SQLSTATE42704 ('Function not found') is signalled.
- **Signature** (name and parameters): For example, DROP FUNCTION myUDF(int). The data type of the parameter(s) must match exactly those of the function found. Also, if length, precision, or scale are specified, they must match exactly the function to be dropped. SQLSTATE 42883 is signalled if a match to an existing function is not found.
- **Specific name**: For example, DROP SPECIFIC FUNCTION myFun0001. Since the SPECIFIC name must be unique per library, this will find, at most, one function. If the function is not found, SQLSTATE 42704 ('Function not found') is signalled.

Note

If a schema is not specified, the authorization ID (user lib) is used if SQL naming is specified. Otherwise, the library list is used.

To drop a UDF using Operations Navigator, you open the required library, right-click on the user defined function you wish to delete, and select **Delete** from the context menu.

If there are no dependent functions, the right panel refreshes, and you should see that the UDF object has been removed from the library.

Functions created implicitly by a `CREATE DISTINCT TYPE` statement cannot be explicitly dropped. They can only be deleted by dropping the type. Built-in functions, and those functions shipped with the database, cannot be deleted.

When a `DISTINCT TYPE` is dropped, all functions that have one or more parameters of that type are implicitly dropped as well. This is accomplished by use of the SYSPARMS catalog.

4.8 Saving and restoring UDFs

This section describes how to save and restore UDFs and some of the considerations that apply. The save and restore of functions currently can only be performed by saving (and restoring) the QSYS2 library. Note that the you *can* save and restore the catalogs themselves, but this is not recommended.

For external functions, enough information is saved with the external program such that, when it is saved and restored, the function is 're-created' on the restore. However, the external program should be implemented in one of the ILE languages, and it has to contain at least one embedded SQL statement

Note that storing information in an external function is a bit tricky. The program may not exist at function creation and may be deleted/re-created/moved at any time. For this reason, saving the SQL information in the program occurs if:

- The external program exists at the time the function is created.
- The function is invoked at least once so that the SQL information can be added to program during the reference/use of it.

As mentioned earlier, sourced and SQL UDFs are implemented as embedded SQL ILE C service programs. This implies that they have enough information stored in the program object such that the function can be re-created.

On a restore of an external program of a function, the following is performed:

- Function (signature) does not exist. In this case, add the function to the catalogs.
- Function (signature) exists (may or may not have exactly the same attributes, but same signature). Do nothing.
- If the function will be 'created' in QSYS2, do not create the function definition (to prevent user functions from being in QSYS2).

4.9 Debugging UDFs

In this section, we show you how to debug UDFs. SQL UDFs are always created as service programs. We recommend that you create external functions as service programs. Therefore, we show you how to debug a service program here. The same technique needs to be used if you wish to debug a program object that is being referenced by an external UDF.

In this example, we debug our IsGif External UDF. Debugging UDFs may be a bit tricky since they are run on the AS/400 system in secondary threads. The following steps outline the debug procedure:

1. Open two native AS/400 5250 sessions and sign on to both sessions. From here onwards, we refer to the first session as *Session A* and to the second session as *Session B*.
2. Switch to Session B, and type in the following command on the command line:

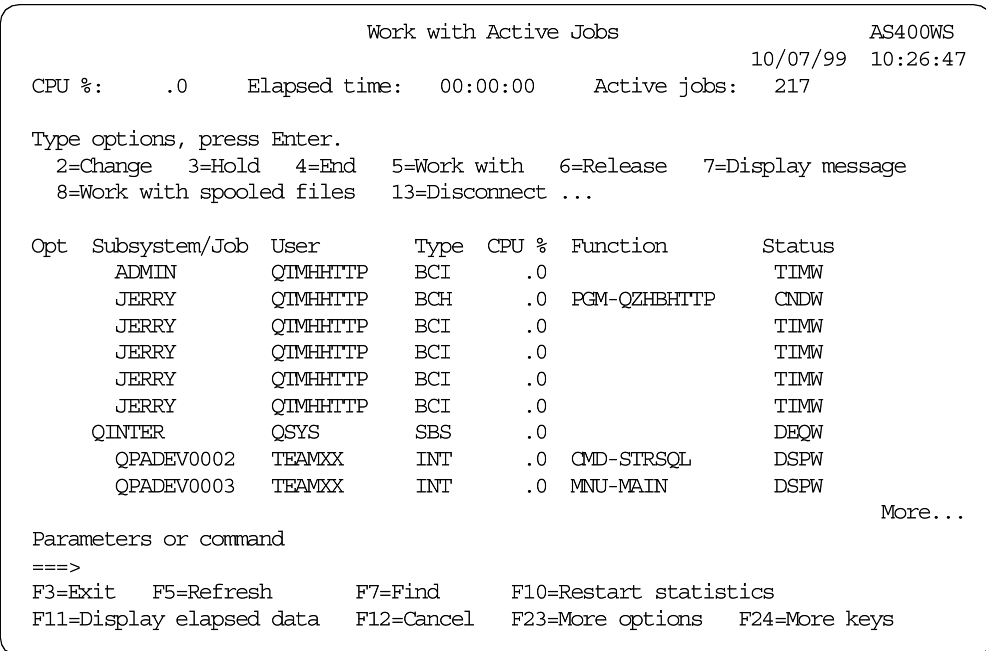
   ```
   STRSQL
   ```

   The interactive SQL session is started, and the SQL command line is displayed.

3. Switch to Session A and type in the following command line:

   `WRKACTJOB`

   The Work with Active Jobs screen is displayed as shown in Figure 83. This screen displays a list of all jobs that are currently active on the system. The job in Session B will be listed as one among these.



```
                              Work with Active Jobs                    AS400WS
                                                             10/07/99  10:26:47
CPU %:      .0     Elapsed time:    00:00:00     Active jobs:   217

Type options, press Enter.
  2=Change   3=Hold   4=End   5=Work with   6=Release   7=Display message
  8=Work with spooled files   13=Disconnect ...

Opt  Subsystem/Job  User        Type  CPU %  Function        Status
       ADMIN        QTMHHTTP    BCI      .0                   TIMW
       JERRY        QTMHHTTP    BCH      .0  PGM-QZHBHTTP     CNDW
       JERRY        QTMHHTTP    BCI      .0                   TIMW
       JERRY        QTMHHTTP    BCI      .0                   TIMW
       JERRY        QTMHHTTP    BCI      .0                   TIMW
       JERRY        QTMHHTTP    BCI      .0                   TIMW
     QINTER         QSYS        SBS      .0                   DEQW
       QPADEV0002   TEAMXX      INT      .0  CMD-STRSQL       DSPW
       QPADEV0003   TEAMXX      INT      .0  MNU-MAIN         DSPW
                                                                       More...
Parameters or command
===>
F3=Exit   F5=Refresh       F7=Find      F10=Restart statistics
F11=Display elapsed data   F12=Cancel   F23=More options   F24=More keys
```

Figure 83. The Work with Active Jobs screen listing all currently active jobs.

4. Find the job started in Session B under the QINTER subsystem. This is done by looking for jobs under the QINTER subsystem that are started with the user ID you used to log on. In our case, it is TEAMxx. Then, locate the job that has the action named STRSQL under the column named Function. When this job is located, use option `5` to work with that job. This is shown in Figure 84.

```
                           Work with Active Jobs                    AS400WS
                                                          10/07/99  10:26:47
CPU %:      .0     Elapsed time:   00:00:00     Active jobs:   217

Type options, press Enter.
  2=Change   3=Hold   4=End   5=Work with   6=Release   7=Display message
  8=Work with spooled files   13=Disconnect ...

Opt  Subsystem/Job  User        Type  CPU %  Function        Status
       ADMIN        QTMHHTTP    BCI      .0                   TIMW
       JERRY        QTMHHTTP    BCH      .0  PGM-QZHBHTTP     CNDW
       JERRY        QTMHHTTP    BCI      .0                   TIMW
       JERRY        QTMHHTTP    BCI      .0                   TIMW
       JERRY        QTMHHTTP    BCI      .0                   TIMW
       JERRY        QTMHHTTP    BCI      .0                   TIMW
     QINTER         QSYS        SBS      .0                   DEQW
5      QPADEV0002   TEAMXX      INT      .0  CMD-STRSQL       DSPW
       QPADEV0003   TEAMXX      INT      .0  MNU-MAIN         DSPW
                                                                   More...
Parameters or command
===>
F3=Exit   F5=Refresh      F7=Find      F10=Restart statistics
F11=Display elapsed data   F12=Cancel   F23=More options   F24=More keys
```

Figure 84. Working with the job in Session B

5. The Work with Job screen is displayed. This screen displays the various actions that can be taken for this job. On the top of the screen, you see the following information:

   - **Job**: This is the name of the job with which you are working.
   - **User**: This is the name of the user profile that is using the job.
   - **Number**: This is the number assigned to the job you are working with. Every job on the AS/400 system is assigned a six digit unique job number.

   Write down your fully qualified name for the Session B job. In our case, it is:

   ```
   044733/TEAMXX/QPADEV0002
   ```

   Now, start a service job for the Session B job. Enter the following command on the command line:

   ```
   STRSRVJOB 044733/TEAMXX/QPADEV0002
   ```

   Note

   The job name will be different for you.

6. Start a debug session for the service program used in the IsGif function. Type the following command on the command line:

   ```
   STRDBG UPDPROD(*YES) SRVPGM(TEAMXX/PICTCHECK)
   ```

7. The debug session appears on your screen with the source code loaded into the debugger. Enter a breakpoint for the first executable statement in the program. In our case, this is the following statement in the PICTCHECK program:

   ```
   if ( *nms_InputNullIndicator01 == -1 ).
   ```

This can be done by placing your cursor on the line of code at which you wish to place the breakpoint and pressing the F6 key. The following message appears at the bottom of the screen:

```
Breakpoint added to line 47
```

This is shown in Figure 85.

```
                              Display Module Source

Program:   PICTCHECK      Library:   TEAMXX        Module:   PICTTYPE
    46
    47       if ( *nms_InputNullIndicator01 == -1 )
    48       {
    49          *nms_OutputNullIndicator01 = -1;
    50          return;
    51       }
    52
    53      chr_FunctionResolution = strstr( sqludf_fname, GIF_FUNCTION );
    54
    55      if ( chr_FunctionResolution != NULL )
    56      {
    57         nmi_CompareResult01 = fun_CheckHeader( str_ProductPicture->data
    58                                                GIF_HEADER_LENGTH,
    59                                                chr_GifHeader87 );
    60         nmi_CompareResult02 = fun_CheckHeader( str_ProductPicture->data
                                                                         More...
Debug . . .

F3=End program   F6=Add/Clear breakpoint   F10=Step   F11=Display variable
F12=Resume       F17=Watch variable   F18=Work with watch   F24=More keys
Breakpoint added to line 47.
```

Figure 85. Adding a breakpoint to the debug session

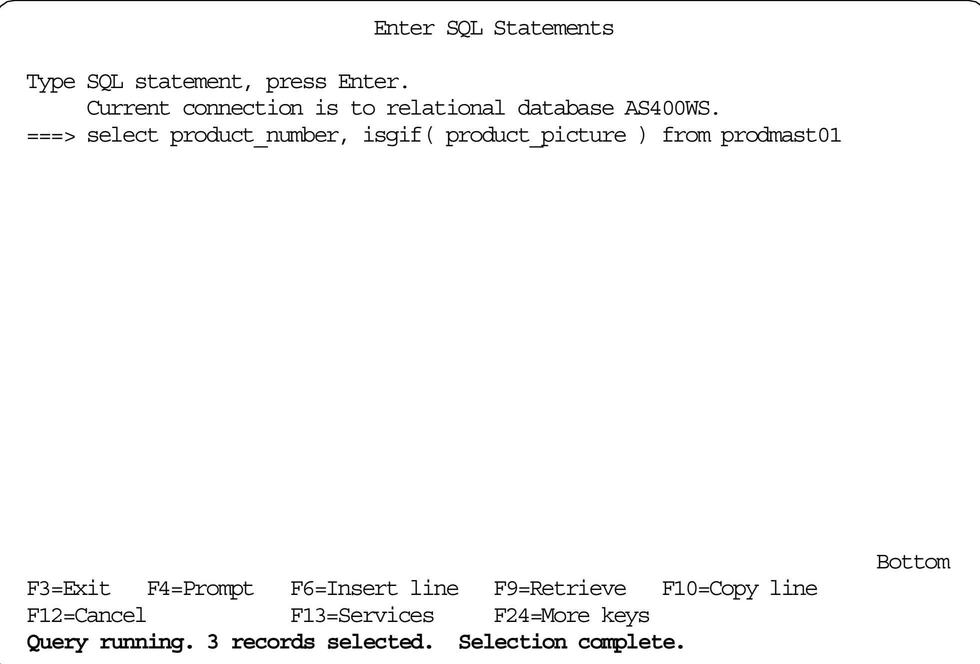
8. Press F12. This takes you back to the command line. Now, you need to invoke the UDF from the Interactive SQL run in Session B.
9. Switch to Session B and type in the following SQL statement on the SQL command line:

   ```
   select product_number, isgif( product_picture ) from prodmast01
   ```

   The SELECT statement begins to execute. The IsGif(PICTURE) UDF is invoked. This also means that the PICTCHECK program is invoked. The following message is displayed at the bottom of the screen:

   ```
   Query running. 3 records selected. Selection complete.
   ```

   This is shown in Figure 86. However, the results of the query do not show up. Instead, the session busy cross sign stays at the bottom of the screen.



```
                              Enter SQL Statements

Type SQL statement, press Enter.
     Current connection is to relational database AS400WS.
===> select product_number, isgif( product_picture ) from prodmast01

                                                                        Bottom
F3=Exit   F4=Prompt   F6=Insert line   F9=Retrieve   F10=Copy line
F12=Cancel            F13=Services     F24=More keys
Query running. 3 records selected.  Selection complete.
```

Figure 86. Invoking the IsGif(PICTURE) external UDF

10.Now, switch back to Session A. You see the source code of the PICTCHECK service program displayed on the screen. The line of source code that is to be currently executed is highlighted in white on the screen. In our case, this is the line at which you set the breakpoint in step 8. This is shown in Figure 87 on page 124.

Note

In print, the line of source code to be executed is shown in bold.

```
                              Display Module Source
Current thread:   00000020     Stopped thread:   00000020
Program:   PICTCHECK      Library:   TEAMXX         Module:   PICTTYPE
    43       int        nmi_CompareResult01 = 0;
    44       int        nmi_CompareResult02 = 0;
    45
    46
    47       if ( *nms_InputNullIndicator01 == -1 )
    48       {
    49          *nms_OutputNullIndicator01 = -1;
    50          return;
    51       }
    52
    53      chr_FunctionResolution = strstr( sqludf_fname, GIF_FUNCTION );
    54
    55      if ( chr_FunctionResolution != NULL )
    56      {
    57         nmi_CompareResult01 = fun_CheckHeader( str_ProductPicture->data
                                                                    More...
Debug . . .

F3=End program   F6=Add/Clear breakpoint   F10=Step   F11=Display variable
F12=Resume       F17=Watch variable   F18=Work with watch   F24=More keys
Breakpoint at line 47 in thread 00000020
```

Figure 87. Debugging the PICTCHECK service program

11. Press the F10 function key to execute the highlighted line of code. The line is executed and gets de-highlighted. The next line of code to be executed is highlighted. Each time you press the F10 key, the next line of code in sequence is executed.

12. You can check the value contained in any of the program variables. This can be done in two ways:

    - Pressing the F11 key after placing the cursor over the variable for which you wish to check the value.
    - Typing in the `EVAL` command on the debug command line.

    We now check the value of the program variable nmi_CompareResult01 . Place your cursor over the variable and press F11. The value of the variable is displayed on the bottom of the screen. This is shown in Figure 88.

```
                                 Display Module Source
Current thread:   00000020     Stopped thread:   00000020
Program:   PICTCHECK      Library:   TEAMXX         Module:   PICTTYPE
    43       int        nmi_CompareResult01 = 0;
    44       int        nmi_CompareResult02 = 0;
    45
    46
    47       if ( *nms_InputNullIndicator01 == -1 )
    48       {
    49          *nms_OutputNullIndicator01 = -1;
    50          return;
    51       }
    52
    53      chr_FunctionResolution = strstr( sqludf_fname, GIF_FUNCTION );
    54
    55      if ( chr_FunctionResolution != NULL )
    56      {
    57         nmi_CompareResult01 = fun_CheckHeader( str_ProductPicture->data
                                                                  More...
Debug . . .

F3=End program   F6=Add/Clear breakpoint   F10=Step   F11=Display variable
F12=Resume       F17=Watch variable   F18=Work with watch   F24=More keys
nmi_CompareResult01 = 0
```

Figure 88. Checking the value of the program variables using the F11 key

13.Place the cursor on the debug command line and type the following command:

```
EVAL *nmi_InputNullIndicator01
```

This time, the value of nmi_InputNullIndicator `01` is displayed on the bottom of the screen.

Note

To display the value of pointer variables, you have *to* use the `EVAL` command on the debug command line. You can use the ILE C/400 pointer notation to display the information in pointer variables. This is shown in Figure 89 on page 126.

```
                              Display Module Source
Current thread:   00000020     Stopped thread:   00000020
Program:   PICTCHECK      Library:   TEAMXX        Module:   PICTTYPE
    43       int        nmi_CompareResult01 = 0;
    44       int        nmi_CompareResult02 = 0;
    45
    46
    47       if ( *nms_InputNullIndicator01 == -1 )
    48       {
    49          *nms_OutputNullIndicator01 = -1;
    50          return;
    51       }
    52
    53      chr_FunctionResolution = strstr( sqludf_fname, GIF_FUNCTION );
    54
    55      if ( chr_FunctionResolution != NULL )
    56      {
    57         nmi_CompareResult01 = fun_CheckHeader( str_ProductPicture->data
                                                                      More...
Debug . . .   EVAL *nms_InputNullIndicator01

F3=End program   F6=Add/Clear breakpoint   F10=Step   F11=Display variable
F12=Resume       F17=Watch variable   F18=Work with watch   F24=More keys
*nms_InputNullIndicator01 = 0
```

Figure 89. Displaying the information in pointer variables using the EVAL command.

14. Sometimes, you may want to display the content of a variable in hexadecimal format. This is especially useful when you work with BLOB variables. You will now display the contents of the str_ProductPicture variable. This variable contains the data from the PRODUCT_PICTURE column of the PRODMAST01 table. The PRODUCT_PICTURE column is based on the distinct type PICTURE which, in turn, is based on BLOB(1M). In our program, we declared the BLOB1M structure to accommodate the BLOB value. This structure is composed of two parts: the length and the data part. The data part of the variable actually contains the binary information passed to our function by the database.

15. Type in the following command on the debug command line:

    ```
    EVAL *str_ProductPicture->data:x 64
    ```

    Since the data part of the str_ProductPicture variable is a string, we must use the pointer notation to display the contents of it. The `:x` after the variable name is used to display the contents of the variable in hexadecimal format. The value 64 instructs the system to display the first 64 bytes of the variable. The result is shown in Figure 90.

```
                        Evaluate Expression

Previous debug expressions

  nmi_CompareResult01 = 0
> EVAL nmi_CompareResult01
  nmi_CompareResult01 = 0
> EVAL nms_InputNullIndicator01
  nms_InputNullIndicator01 = SPP:E2F09C16E30011C0
> EVAL *nms_InputNullIndicator01
  *nms_InputNullIndicator01 = 0
> EVAL *nms_InputNullIndicator01
  *nms_InputNullIndicator01 = 0
> EVAL *str_ProductPicture->data:x 64
     00000     47494638 39613601 C000F700 000B0B0B   - åñã../..{.7.....
     00010     427B9430 117F3AAB EC454480 43484234   - â#m..".¿ÖáàØäçâ.
     00020     7BB56D4C C2D0CFB1 7C83BC22 18482F64   - #§_<B}õ£@c¯..ç.À
     00030     81B0C3D8 4866689F A4B8AFAE 99A6DBF7   - a^¯CQçÃÇ¤u½®Þrwû7
                                                                       Bottom
Debug . . .

F3=Exit   F9=Retrieve   F12=Cancel   F16=Repeat find   F19=Left   F20=Right
F21=Command entry       F23=Display output
```

Figure 90. Displaying the contents of a variable in hexadecimal format

16. Continue to press the F10 key until you step through the entire program. At any time, you can run the program to completion by pressing the F12 key.
17. Once debugging your code is finished, you return to the Work with Job screen. On the command line, type the following CL commands:

```
ENDDBG
ENDSRVJOB
```

This ends the debug mode and the service job being run to debug the service program.

4.10 Coding considerations

When coding UDFs, you should keep in mind some of the limitations and restrictions that apply to them. The following list contains important recommendations and hints for UDFs developers:

- UDFs should not perform operations that take a long time (minutes or hours).
- UDFs are invoked from a low-level in DB2 that holds resources (locks and seizes) for duration of the UDF execution.
- If UDF doesn't finish in an allocated time, the SQL statement fails. You can override the system time out value with UDF_TIME_OUT parameter in the query option file QAQQINI. Refer to *DB2 UDB for AS/400 SQL Programming*, SC41-5611, for details.
- Avoid inserts, updates, and delete operations on the same tables as the one referred to in the invoking statement.
- A UDF runs in the same job as the invoking SQL statement, but runs in a separate system thread, so secondary thread considerations apply:

– UDFs will conflict with thread-level resources held by the SQL statement. UDFs cannot perform any operation that is blocked from secondary threads.

– Activation Group (*NEW) is not allowed for UDFs.

– UDFs do not inherit program adopted authority that may have been active. Authority comes from the UDF program itself or the user running the SQL.

Chapter 5. Programming alternatives for complex objects

Throughout this redbook, we present a wide range of code examples that illustrate how to use complex objects in SQL and ILE C with embedded SQL programming interfaces. However, depending on your skills, you may use other programming tools available both on the AS/400 system and the clients. In this chapter, we provide you with a set of basic code examples to assist you in writing UDB-aware applications in other programming environments. We discuss building a client/server application with Java running on the client. We also show how to code using Call Level Interface (CLI).

The code examples illustrated in this chapter refer to an SQL table called PRODMASTOL. The column definition for this table is shown in 3.2.3, “Creating distinct types with the SQL interface” on page 32.

5.1 Using complex objects in Java client applications

In this section, we outline the steps required to use complex objects managed by the DB2 UDB for AS/400 in a Java client application. The combination of Java running on the client, and the SQL running on the powerful database server, such as the AS/400 system, can result in a highly scalable and robust software solution.

We assume that you are already familiar with JDBC, so we provide a detailed discussion only for the JDBC APIs pertaining to the complex object support. Refer to *Building AS/400 Client/Server Applications with Java*, SG24-2152, to learn how to use JDBC with the AS/400 database.

In our test scenario, we coded a Java client, which uses the AS/400 Toolbox for Java JDBC driver, to send the SQL request to DB2 UDB for AS/400. In the AS/400 client/server architecture, a JDBC client communicates with a corresponding AS/400 server job, which runs the SQL requests on behalf of this client. In other words, when we submit an SQL request from the Java client, there is an AS/400 server job that actually performs the requested operation on the server and then passes back the results to the client. The AS/400 server jobs associated with the database access are named QZDASOINIT and run in the QSERVER subsystem.

5.1.1 Getting ready to use JDBC 2.0 driver

JDBC is a Java API for executing SQL statements. The initial release of the JDBC API, JDBC 1.0, provided the basic functionality for database access. The second edition of this API specification, JDBC 2.0, supplements the basic set of functions with advanced features. In particular, JDBC 2.0 adds support for storing persistent Java objects and mapping for SQL3 data types, such as BLOBs and User Defined Types. Note that the JDBC 2.0 requires Java 2 platform.

The AS/400 Toolbox for Java supports JDBC 2.0. The AS/400 Toolbox for Java is currently available from IBM with OS/400 as a no charge licensed program product (LPP) 5769-JC1. To take advantage of the JDBC 2.0 features, you need to install modification 2 of the Toolbox, which is available at the following Web site: `http://www.ibm.com/as400/toolbox`

© Copyright IBM Corp. 2000

Refer to *Building AS/400 Client/Server Applications with Java*, SG24-2152, for details on how to install and utilize the Toolbox classes.

We tested our Java samples using JDK 1.2.2 for Windows NT. We set the full path of the \jdk1.2.2\bin directory with the following command:

```
SET PATH=D:\JDK1.2.2\BIN;
```

We also had to set the CLASSPATH variable to point to the Toolbox classes. We used the following command:

```
SET CLASSPATH=D:\Program Files\IBM\Client Access\jt400\lib\jt400.zip;
```

Make sure that you specify the full path of the directory where you installed the Toolbox classes. We installed the Toolbox as part of Client Access Express installation. In your case, the CLASSPATH may point to a different directory. Notice also that you may use jt400.jar rather than jt400.zip. This setup is valid for the duration of your DOS session. To set PATH and CLASSPATH permanently, start the Control Panel. Select **System->Environment**. Look for `PATH` or `CLASSPATH` in the User Variables and System Variables.

5.1.2 Using a Blob object

An SQL BLOB is mapped by the JDBC driver into a Java Blob object. You can access values of type Blob in the same way that you access traditional SQL92 built-in types. The interfaces `ResultSet`, `CallableStatement`, and `PreparedStatement` support methods `getBlob` and `setBlob` for a BLOB value. You can use these methods in the same way that you use `getString` and `setString` to manipulate a CHAR or VARCHAR value. The JDBC 2.0 specification defines Blob as an interface. The JDBC 2.0 driver provides a database specific class, which implements this interface. In case of the AS/400 Toolbox for Java driver, this class is called com.ibm.as400.access.AS400JDBCBlob.

5.1.2.1 Creating and materializing a Blob object

The following short Java program illustrates how to use the AS/400 Toolbox for Java JDBC 2.0 driver to retrieve a BLOB from an AS/400 table. The ImageDisplayer class accepts one input parameter: Product_Number. It connects to the AS/400 system and retrieves the Product_Picture for the given Product_Number. After materializing the Product_Picture data on the workstation, the program uses Swing GUI to display the picture. The numbered sections of the source code are explained in the notes following the listD.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.sql.*;

/* This class displays an image retrieved from DB2 UDB for AS/400. */
public class ImageDisplayer extends JFrame {

  public static void main(String[] args) {
    Image image = db2getImage(args[0]);
    ImagePanel imagePanel = new ImagePanel(image);

    JFrame f = new JFrame("ImageDisplayer");
    f.addWindowListener(new WindowAdapter() {
       public void windowClosing(WindowEvent e) {
         System.exit(0);
       }
      });

    f.getContentPane().add(imagePanel, BorderLayout.CENTER);
```

```
    f.setSize(new Dimension(200,200));
    f.setVisible(true);
   }
  public static Image db2getImage (String productNumber)
    {
      String system          = "AS400WS";
      Connection connection   = null;
      Image image = null;
    try {
        // Load the AS/400 Toolbox for Java JDBC driver.
        DriverManager.registerDriver(new
        com.ibm.as400.access.AS400JDBCDriver());

        // Get a connection to the database.  Since we do not
        // provide a user id or password, a prompt will appear.
        connection = DriverManager.getConnection ("jdbc:as400://" + system);

        PreparedStatement stmt = connection.prepareStatement(
        "SELECT product_picture " +
        "FROM teamxx.prodmast01 " +
        "WHERE PRODUCT_NUMBER = CAST(? AS SRLNUMBER)"); 1
        stmt.setString(1, productNumber); 2
        ResultSet rs = stmt.executeQuery();
        while (rs.next()) {
        Blob pictblob = rs.getBlob(1); 3
          long length = pictblob.length();
          ImageIcon imageicon = new ImageIcon(pictblob.getBytes(0, (int)
          length)); 4
          image = imageicon.getImage();
         }
 }
    catch (Exception e) {
      System.out.println ();
      System.out.println ("ERROR: " + e.getMessage());
     }
 return image;
 }
}

class ImagePanel extends JPanel {
    Image image;

    public ImagePanel(Image image) {
        this.image = image;
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g); //paint background

        //Draw image at its natural size.
        g.drawImage(image, 0, 0, this);

      }
}
```

Notes for ImageDisplayer.java

1. The Product_Number column in the PRODMAST01 table is of user defined type SRLNUMBER. Because the implicit casting is not supported in the WHERE clause, we need to explicitly cast the parameter marker to the SRLNUMBER UDT.
2. The `setString` method of the PreparedStatement class is used to set the parameter to the Product_Number passed by the invoking process.
3. The Blob object is created. At this time, the variable `pictblob` contains a logical pointer to the BLOB value stored in the Product_Picture column. Note that the UDT Picture was implicitly cast to its source type BLOB(1M) on the I/O operation. Therefore, no explicit casting is needed, and we can use `getBlob` method on the `rs` object.
4. We need to materialize the BLOB data before we can display it on the workstation. We use the `getBytes` method on the Blob object for this purpose.

The imageicon object now contains a copy of all of the bytes in the BLOB value.

You can use the `getBytes` method on a BLOB object to materialize only a fragment of the BLOB object. The first argument of this method is used to specify the starting byte, while the second argument tells how many bytes should be returned.

To compile the `ImageDisplayer.java` program, type the following command at the DOS prompt:

```
javac ImageDisplayer.java
```

To execute the program, type the following command:

```
java ImageDisplayer 00001
```

Note, that the string value '00001' was passed as the Product_Number parameter. The results are shown in Figure 91.

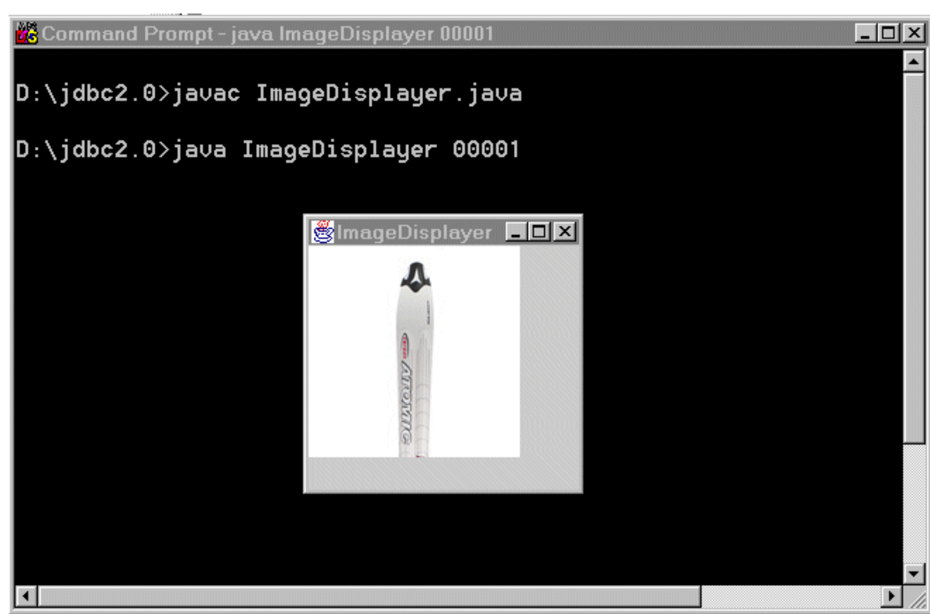


Figure 91. Using Java to display DB2 UDB for AS/400 BLOBs

5.1.2.2 Storing a Blob object in the database

You can use the `setBlob` method on a BLOB object to store it in the DB2 UDB for AS/400 database. The following code snippet illustrates this approach:

```
...
Blob pictblob = rs.getBlob("PRODUCT_PICTURE"); 1
...
// Prepare UPDATE statement.
PreparedStatement stmt = connection.prepareStatement(
"UPDATE teamxx.prodmast01" +
" SET PRODUCT_PICTURE = ? WHERE PRODUCT_NUMBER = CAST( ? AS SRLNUMBER)");
// Set the first parameter marker to a blob object
stmt.setBlob(1, pictblob); 2
// Set the second parameter marker to a String
stmt.setString(2, productNumber);
// Execute the SQL statement
stmt.executeUpdate(); 3
```

Notes on Blob object storage code

1. The Blob object is retrieved from the PRODMAST01 table. We can now use this object within our Java application. For example, we could crop the retrieved product picture. The next two steps demonstrate how to update the table with this changed object.
2. The pictblob object is passed as the input parameter to the prepared statement object `stmt`.
3. The Blob value pointed by pictblob is now stored in PRODMAST01 table.

The pictblob Blob object must exist in your Java application before you can execute the `setBlob` method on it. The sql.java package defines the Blob as a public interface, so you cannot instantiate it in your application. Instead, you need to use `getBlob` method on `ResultSet`, `CallableStatement`, or `PreparedStatement` to get access to the Blob data or you can provide your own implementation.

The `setBlob` method is capable of sending large amounts of data. You can also accomplish this task by setting a Blob parameter marker to a Java input stream. The following code example shows how to load Blob data into a AS/400 table using the `setBinaryStream` method. This approach is useful if you have to construct the BLOB object in your application and then upload it to the database for persistent storage. The LoadPicture program accepts two parameters: name of a file on the workstation that contains the product picture, and the product number for the given picture. The program reads the content of the file and stores it as a Blob object in the AS/400 database.

```
import java.sql.*;
import java.io.*;

public class LoadPicture
{
 public static void main (String[] args)
  {
   String system           = "AS400WS";
   Connection connection   = null;
   try {
       File file = new File(args[0]);
       int  fileLength = (int)file.length();
       InputStream fin = new FileInputStream(file); 1
       // Load the AS/400 Toolbox for Java JDBC driver.
       DriverManager.registerDriver(new com.ibm.as400.access.AS400JDBCDriver());
       // Get a connection to the database.  Since we do not
       // provide a user id or password, a prompt will appear.
       connection = DriverManager.getConnection ("jdbc:as400://" + system);
       DatabaseMetaData dmd = connection.getMetaData ();

       // Prepare UPDATE statement.
       PreparedStatement stmt = connection.prepareStatement(
       "UPDATE " + collectionName + dmd.getCatalogSeparator() + tableName +
       " SET PRODUCT_PICTURE = ? WHERE PRODUCT_NUMBER = CAST( ? AS SRLNUMBER)");
       // Set the first parameter marker to a binary input stream
       stmt.setBinaryStream(1, fin, fileLength); 2
       // Set the second parameter marker to a String
       stmt.setString(2, args[1]);
       // Execute the SQL statement
       stmt.executeUpdate(); 3

    }
catch (Exception e) {
    System.out.println ();
    System.out.println ("ERROR: " + e.getMessage());
 }
finally {
  // Clean up.
  try {
       if (connection != null)
       connection.close ();
      }
```

```
   catch (SQLException e) {
            // Ignore.
          }
  }
  System.exit (0);
 }
}
```

Notes for LoadPicture.java

1. We use the instance of `FileInputStream` to obtain the content of the picture file located in the workstation's file system.
2. The Blob parameter marker is set to input stream.
3. At the SQL statement execution, the JDBC driver repeatedly calls to the fin input stream to transmit the Blob content to the database.

5.1.3 Using a Clob object

An SQL Clob is mapped by the JDBC driver into a Java Clob object. The interfaces `ResultSet`, `CallableStatement`, and `PreparedStatement` support methods `getClob` and `setClob` that can be used to manipulate the CLOB data. These interfaces also support `setAsciiStream`, and `setCharacterStream` methods that allow you to input a stream as a Clob value. Additionally, you can use `getAsciiStream` and `getCharecterStream` methods on a Clob object to materialize it as an input stream.

5.1.3.1 Creating and materializing a Clob object

The following Java program shows how to retrieve a Clob value from an AS/400 table. The QueryClob class connects to the AS/400 using the AS/400 Toolbox for Java JDBC driver and retrieves two columns, Product_Number and Product_Description, from the PRODMAST01 table. Then, it iterates through all rows in the result set. For every row, it materializes the Product_Description data as a Clob object and then manipulates the object to retrieve the color of a product.

```
import java.sql.*;
public class ClobQuery
{
  public static void main (String[] parameters)
   {
    String system           = "AS400WS";
    Connection connection   = null;

    try {

        // Load the AS/400 Toolbox for Java JDBC driver.
        DriverManager.registerDriver(new
                      com.ibm.as400.access.AS400JDBCDriver());

       // Get a connection to the database.  Since we do not
       // provide a user id or password, a prompt will appear.
       connection = DriverManager.getConnection ("jdbc:as400://" + system);

       // Allocate the statement and execute the query.
       Statement stmt = connection.createStatement ();

       ResultSet rs = stmt.executeQuery (
       "SELECT Product_Number, Product_Description " +
       "FROM TEAMXX.PRODMAST01");
       // Iterate throught the rows in the result set and output
       // the columns for each row.
       while (rs.next ()) {
              String prdnum = rs.getString(1);
              System.out.print(prdnum + "  ");
              Clob prddesc = rs.getClob(2); 1
              if (prddesc != null)
               {
```

```
                long length = prddesc.length();
                String desc = prddesc.getSubString(0, (int) length); 2
                int startcolor = desc.indexOf((String) "Color:")+ 6;
                int stopcolor = desc.indexOf((String)"Best For:",startcolor);
                System.out.print(desc.substring(startcolor, stopcolor - 1));
               }
             else
               {
                System.out.print("NULL");
               }
               System.out.println ();
          }
         }
       catch (Exception e) {
            System.out.println ();
            System.out.println ("ERROR: " + e.getMessage());
         }
       finally {

            // Clean up.
            try {
                if (connection != null)
                    connection.close ();
            }
            catch (SQLException e) {
                // Ignore.
            }
         }
        System.exit (0);
    }
}
```

Notes for ClobQuery.java

1. A Clob object is created. The `prddesc` variable contains a logical pointer to the Clob value stored in the Product_Description column. Note that this column was specified with a user defined type `PRDDESC`. This UDT was implicitly cast to its source built-in type CLOB(50k) on the I/O operation.
2. The `getSubString` method can be used to materialize all or part of the Clob value as a String object. It was used to materialize the complete content of the `Product_Description` column.

As mentioned earlier, you may also materialize the Clob value as a byte stream containing Unicode characters. The following code snippet illustrates how to use the `getCharacterStream` method on a Clob value to accomplish this task:

```
...
File file = new File(args[0]);
OutputStream fout = new FileOutputStream(file); 1
...
byte[] clobBytes = new byte[MAX_FILE_SIZE];
char[] clobData = new char[MAX_FILE_SIZE];
PreparedStatement stmt = connection.prepareStatement(
"SELECT product_description FROM TEAMXX.PRODMAST01 " +
"WHERE PRODUCT_NUMBER = CAST( ? as SRLNUMBER)");
String productNumber = new String(args[1]);
stmt.setString(1, productNumber);
ResultSet rs = stmt.executeQuery();
while (rs.next()) {
  Clob prddesc = rs.getClob(1); 2
  Reader rin = prddesc.getCharacterStream(); 3
  int max = rin.read(clobData); 4
  for (int cnt = 0; cnt < max; cnt = cnt + 1) {
       clobBytes[cnt] = (byte)clobData[cnt]; 5
    }
  fout.write(clobBytes); 6
 }
```

Code example notes

1. The FileOutputStream class is used to write the Clob data retrieved from the AS/400 system to a file on the workstation.

2. The Clob object is created. The underlying UDT PRDDESC was implicitly cast to its source type CLOB(50k).
3. The Clob value is materialized as a stream of Unicode characters.
4. The `read` method reads the Unicode characters from the stream into a character array. The variable `max` contains the number of characters retrieved from the database (length of Clob).
5. We copy the description from the character array to the byte array to write the content out to the output stream.
6. We write the Clob data to a workstation file using the `fout` output stream.

5.1.3.2 Storing Clob in the database

A `PreparedStatement` object supports the `setClob` method, which can be used to store the Clob data in the DB2 UDB for AS/400 database. The following code snippet illustrates this approach:

```
...
Clob prddesc = rs.getClob("PRODUCT_PICTURE"); 1
...
// Prepare UPDATE statement.
PreparedStatement stmt = connection.prepareStatement(
"UPDATE teamxx.prodmast01" +
" SET PRODUCT_DESCRIPTION = ? WHERE PRODUCT_NUMBER = CAST( ? AS SRLNUMBER)");
// Set the first parameter marker to a blob object
stmt.setClob(1, prddesc); 2
// Set the second parameter marker to a String
stmt.setString(2, productNumber);
// Execute the SQL statement
stmt.executeUpdate(); 3
```

Code sample notes

1. The Clob object is retrieved from the database.
2. The `setClob` method on the `stmt` object allows the Clob data to be passed as an input parameter. The `setClob` method requires an existing Clob object as the second parameter. We set this parameter to the Clob retrieved in step 1.
3. The Clob is now stored in the AS/400 table.

The `setClob` method can be used to upload large amounts of data. The alternate approach is to load a large Clob object by setting a Clob parameter marker to a Java input stream. The following code snippet shows how to read a text file from the workstation and upload it to the AS/400 as a Clob.

```
...
File file = new File(args[0]);
int  fileLength = (int)file.length();
InputStream fin = new FileInputStream(file); 1
...
// Prepare UPDATE statement.
PreparedStatement stmt = connection.prepareStatement(
"UPDATE TEAMXX.PRODMAST01 " +
" SET PRODUCT_DESCRIPTION = ? WHERE PRODUCT_NUMBER = CAST( ? AS SRLNUMBER)");
// Set the first parameter marker to a binary input stream
stmt.setAsciiStream(1, fin, fileLength); 2
// Set the second parameter marker to a String
stmt.setString(2, args[1]);
// Execute the SQL statement
stmt.executeUpdate(); 3
```

Code sample notes

1. We use an instance of `FileInputStream` to read the product description from a workstation’s file.

2. The Clob parameter marker is set to an input stream. We need to cast the parameter marker to the appropriate UDT in the SQL statement.
3. The JDBC driver repeatedly calls to the `fin` input stream to transmit the Clob content to the database.

5.1.4 Using metadata

Metadata is useful when you write programs that use advanced database features like complex object support. The JDBC defines two metadata interfaces: DatabaseMetaData and ResultSetMetaData. A DataBaseMetaData object provides comprehensive information about the database. A ResultSetMetaData object retrieves information about the columns in a ResultSet object.

The following code example illustrates how to use the metadata interfaces to get the column information for our test table PRODMAST01. It also retrieves the descriptions of all UDTs defined in the TEAMxx collection.

```
import java.sql.*;

public class GetMetaData {

 public static void main(String args[]) {
      Connection con;
      Statement stmt;
      String system           = "AS400WS";
      String collectionName   = "TEAMXX";
      String tableName = "PRODMAST01";

      try {
           DriverManager.registerDriver(new
                com.ibm.as400.access.AS400JDBCDriver());
           con = DriverManager.getConnection ("jdbc:as400://" + system);
           DatabaseMetaData dmd = con.getMetaData (); 1
           stmt = con.createStatement();

           ResultSet rs = stmt.executeQuery("select * from " +
                collectionName + dmd.getCatalogSeparator() + tableName);

           ResultSetMetaData rsmd = rs.getMetaData(); 2
           int numberOfColumns = rsmd.getColumnCount(); 3
           for (int i = 1; i <= numberOfColumns; i++) {
                String colName = rsmd.getColumnName(i);
                String tblName = rsmd.getTableName(i);
                int type = rsmd.getColumnType(i);
                String name = rsmd.getColumnTypeName(i);4
                boolean caseSen = rsmd.isCaseSensitive(i);
                boolean writable = rsmd.isWritable(i);
                System.out.println("Information for column " + colName);
                System.out.println("    Column is in table " + tblName);
                System.out.println("        Column type is " + type);
                System.out.println("    DBMS name for type is " + name);
                System.out.println("    Is case sensitive:  " + caseSen);
                System.out.println("    Is possibly writable:  " + writable);
                System.out.println("");
           }

          int[] types = {Types.DISTINCT};
           ResultSet rsUDT = dmd.getUDTs(null, collectionName, "%", types); 5

           while (rsUDT.next()) {
                System.out.println("UDT catalog " + rsUDT.getString(1));
                System.out.println("UDT schema  " + rsUDT.getString(2));
                System.out.println("Type name   " + rsUDT.getString(3)); 6
                System.out.println("Class name  " + rsUDT.getString(4)); 7
                System.out.println("Data type   "  + rsUDT.getString(5));
                System.out.println("Remarks     "  + rsUDT.getString(6));
           }
           stmt.close();
           con.close();
        } catch(SQLException ex) {
           System.err.println("SQLException: " + ex.getMessage());
```

```
        }
         System.exit (0);
       }
}
```

Notes for GetMetaData.java

1. We create a DatabaseMetaData object that contains information about DB2 UDB for AS/400. We use the `getMetaData` method on the Connection object for this purpose.
2. The `rs` object contains the data retrieved from the PRODMAST01 table with the SELECT statement. Now we can get information about the columns in the `rs ResultSet` by creating a ResultSetMetaData object. We use the `getMetaData` method on the `rs` object for this purpose.
3. The `getColumnCount` method on the ResultSetMetaData object created in step 2 is used here to find out how many columns the result set has. Our SELECT statement retrieved all of the columns in the PRODMAST01 table, so the value of the numberOfColumns variable is set to 5.
4. We iterate through the `rsmd` columns to print out the detailed metadata information about each particular column in the PRODMAST01 table. For instance, the `getColumnTypeName` method is used to find out a column's data type name. The example data retrieved for the Product_Number column is shown here:

   ```
   Information for column PRODUCT_NUMBER
       Column is in table
       Column type is 1
       DBMS name for type is CHAR
       Is case sensitive:  true
       Is possibly writable:  false
   ```

   Note, that the `getColumnTypeName` method reports the name of the source built-in data type name, rather than the UDT name for the Product_Number column.
5. The `getUDTs` method on the DatabaseMetaData object gets a description of the UDTs defined in a particular schema. This method accepts four parameters:
   - `catalog`: A string object representing a catalog name. Set this parameter to null for the DB2 UDB for AS/400 database.
   - `schemaPattern`: A string object representing a schema name pattern. We use TEAMxx to indicate that we want to retrieve the UDTs definitions from this particular schema (library). You can set this parameter to null to retrieve all UDTs without any schema name restrictions.
   - `typeNamePattern`: A string object representing type name pattern. We use `%` to indicate that we want to retrieve all UDTs definitions in the TEAMxx schema.
   - `types[]`: An array representing the data types to be retrieved. We set it to `Types.DISTINCT` to indicate that we want only user distinct types definitions. Other database platforms may support other values, such as `Types.STRUCT`.

   The `rsUDT` result set object contains one row for each UDT found in the TEAMxx schema. Each row of this ResultSet object has six columns containing a type catalog, type schema, UDT's type name (as defined on the CREATE table statement), Java class name to represent given UDT, generic JDBC data type as defined in `java.sql.Types`, and remarks.

6. The third column of the `rsUDT` result set contains the SQL type name for a given UDT.
7. The fourth column of the `rsUDT` contains a String object giving a class name in the Java programming language for this UDT. The example data retrieved for the PRDDESC distinct type is shown here:

   ```
   UDT catalog AS400WS
   UDT schema  TEAMXX
   Type name   PRDDESC
   Class name  com.ibm.as400.access.AS400JDBCClob
   Data type   2001
   Remarks     UDT sourced from CLOB(50K)
   ```

   Note that the Clob interface is implemented by the AS/400 Toolbox class called com.ibm.as400.access.AS400JDBCClob.

5.2 Using complex objects in CLI or ODBC

In this section, we explain how to use complex objects with the DB2 Call Level Interface (CLI). Since the CLI specification is based on the ODBC, the discussion presented here should be relevant to the ODBC support. However, we didn't test the code samples listed in this section with the AS/400 ODBC driver.

5.2.1 DB2 CLI application flow

The DB2 Call Level Interface allows applications to access the data in the DB2 family of database management systems (DBMS) using Structured Query Language (SQL) as a standard for accessing data. Using this interface, a single application can access different DBMS. This allows the application developer to develop, compile, and ship an application without targeting the specific database. The DB2 Call Level Interface is an alternative to an embedded dynamic SQL. On the AS/400 system, this interface is available to any of the ILE languages.

A DB2 CLI application can be broken down into a set of tasks. Each task may be composed of several lines of code and may use a number of DB2 CLI functions. The sample applications included in this section demonstrate only a small subset of all CLI functions available on the AS/400 system. Refer to *DB2/400 Advanced Database Functions*, SG24-4249, for more details on CLI programming.

5.2.2 Passing LOB to a stored procedure written in CLI

The following C code illustrates how to pass a BLOB value as an input parameter to the CLI stored procedure. The BLOB is the first parameter passed to the RTVPRDNB3 procedure and is defined as INPUT. You could use this procedure as part of your Web store application, which presents online customers with a range of product pictures. Then, customers can click on the product they want to purchase. The stored procedure is then used to find out the product number for the item which was clicked.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "sqlcli.h" 1

...
typedef struct
{
  unsigned long length;
  char data[ 1048576 ];
```

```
} BLOB1M; 2

BLOB1M     bin_ProductPicture; 3

...
void main( int argc, char **argv )
{
 ...

     memcpy( ( void * )&bin_ProductPicture, argv[ 1 ], 1048580 ); 4
...
```

C code notes

1. To use DB2 CLI functions in your C programs, you must include the header file called *"sqlcli.h"*. This include file is contained in QSYSINC library. The OS/400 option Openness Includes needs to be installed on the AS/400 system for this library to be present.
2. The BLOB1M structure is declared. This structure has two elements, the current length of the BLOB object and the data buffer, which can contain up to 1 MB of binary data.
3. The bin_ProductPicture variable contains the picture passed to the stored procedure by the invoking process.
4. The content of the first parameter is copied into the bin_ProductPicture variable. Note that, in the C calling convention, the second argument passed to the program object constitutes the first parameter passed to the stored procedure. The first argument of the program, pointed by argv[0], is always set to the called program name, RTVPRDNB3 in this case. When copying the BLOB parameter into a variable, make sure that you copy both data length and data buffer.

Once the parameters we passed and the initial CLI environment were successfully created, we can implement the business logic of the stored procedure. Let's take a closer look at the fun_Process function, which is the core of the RTVPRDNB3 program. The most interesting (and tricky) part of this function is the code, which illustrates how to bind a BLOB parameter using the SQLBindParam function. Refer to A.7, "RtvPrdNbr3: External stored procedure written in CLI" on page 222, for a complete code listing.

```
SQLRETURN fun_Process()
{
  short Pictture_Ind = 0;

   printf( "Attempting to allocate handle to statement\n" );

  nml_ReturnCode = SQLAllocStmt( nml_HandleToDatabaseConnection,
                                 &nml_HandleToSqlStatement ); 1
 {
     printf( "Could not allocate handle to statement\n" );
    fun_PrintError( SQL_NULL_HSTMT );
    printf( "Terminating\n" );
     return SQL_ERROR;
    } 2

    strcpy( chs_SqlStatement01, "select product_number " );
    strcat( chs_SqlStatement01, "from teamxx.prodmast01 " );
    strcat( chs_SqlStatement01, "where " );
    strcat( chs_SqlStatement01, "product_picture = " );
    strcat( chs_SqlStatement01, " cast( ? as  TEAMXX.PICTURE)") 3;

    nml_ReturnCode = SQLPrepare( nml_HandleToSqlStatement,
                                 chs_SqlStatement01,
                                 SQL_NTS ); 4
     if ( nml_ReturnCode != SQL_SUCCESS )
     {
       ...
```

```
    }

   nmi_PcbValue = bin_ProductPicture.length;
   nml_ReturnCode = SQLBindParam( nml_HandleToSqlStatement,
                                  1,
                                  SQL_BLOB,
                                  SQL_BLOB,
                                  sizeof( bin_ProductPicture ),
                                  0,
                                  ( SQLPOINTER ) bin_ProductPicture.data,
                                  ( SQLINTEGER *) &nmi_PcbValue );5
    if ( nml_ReturnCode != SQL_SUCCESS )
  {
      ...
  }

 nml_ReturnCode = SQLExecute( nml_HandleToSqlStatement );
  if ( nml_ReturnCode != SQL_SUCCESS )
  {
        ...
  }

   nml_ReturnCode = SQLBindCol( nml_HandleToSqlStatement,
                                1,
                                SQL_CHAR,
                                ( SQLPOINTER ) chs_ProductNumber,
                                sizeof( chs_ProductNumber ),
                                ( SQLINTEGER * ) &nmi_PcbValue ); 6
  if ( nml_ReturnCode != SQL_SUCCESS )
  {
    ...
  }

 nml_ReturnCode = SQLFetch( nml_HandleToSqlStatement );
  if ( nml_ReturnCode != SQL_SUCCESS )
  {
   ....
  }
  else
  {
   return SQL_SUCCESS;
  }
}
```

Code listing notes

1. The SQL statement handle is allocated. This handle is used to pass SQL requests to the DB2 UDB for AS/400 database engine.
2. This a typical error-handling routine, which is used to catch SQL error conditions returned from the database. This routine is used after each execution of a CLI function to make sure that there are no pending SQL errors.
3. The text of the SQL request is assembled here. Note the use of the parameter marker. This marker is used to bind the BLOB value passed from the invoking process. We need to explicitly cast the parameter marker to the `TEAMXX.PICTURE` UDT, because the implicit casting is not supported in the WHERE clauses.
4. The SQL statement is prepared. Notice that the CLI uses dynamic SQL under the covers.
5. To bind application variables to parameter markers, the application must call `SQLBindParam()` or `SQLSetParam()`. Both functions are the same and are included for compatibility. The sample application provides the following parameters to the SQLBindParam function:
   - `nml_HandleToSqlStatement`: This is the handle to the SQL statement that contains the parameter markers.
   - `1`: This is the number of the parameter marker to which you want to bind the application variable. We bind the `chs_ProductNumber` variable to the first (and only) parameter marker. If you have more parameter markers in your

SQL statement, you need to call the SQLBindParam function for each of them. The parameter markers are counted from left to right, starting with 1.

- `SQL_BLOB`: This is the data type of the application variable as it is defined in C. This is the data type of the parameter passed by the invoking process.
- `SQL_BLOB`: This is the SQL data type of the application variable.
- `sizeof( bin_ProductPicture )`: This is the precision or length of the application variable. In the case of BLOB variables, this is the variable size in bytes. For this parameter you need to pass the size of the bin_ProductPicture structure.
- `0`: This is the scale of the application variable. In data types other than zoned packed decimals, this is unused. In case of packed and zoned decimals, this is the number of digits to the right of the decimal point.
- `( SQLPOINTER ) bin_ProductPicture.data`: This is a pointer to the buffer that actually contains the data to be used at the execution time. For this parameter, we pass the pointer to the data buffer containing the BLOB object.
- `( SQLINTEGER *) &nmi_PcbValue`: This is an integer pointer that, for the BLOB variable, points to a location containing the exact length of the BLOB data. The nmi_PcbValue was set to bin_ProductPicture.length just before the `SQLBindParam` was called.

6. After the SQL statement is successfully executed, we bind the value of the column returned by the run time to an application variable. The `chs_ProductNumber` variable contains the product number for the first item in the table, which has the same product picture as the picture passed by the invoking process as a search parameter.

   Note: The PRODUCT_NUMBER column was implicitly cast from the SRLNUMBER UDT to its underlying source type of CHARACTER(5). The DB2 CLI, like any other high level programing interface, is not aware of the UDTs, so UDTs are implicitly converted to their appropriate source data types during the INPUT/OUTPUT operations.

The following CL command compiles our sample CLI stored procedure:

```
CRTBNDC PGM(DPOBJECT/RTVPRDNBR3) SRCFILE(DPSOURCE/QCSRC) OUTPUT(*PRINT)
DBGVIEW(*ALL)
```

Since CLI is not using embedded SQL, the DB2 for AS/400 Development Kit is *not* required on your development machine. Once the program object is successfully created, we register the stored procedure with the following SQL statement:

```
create procedure TEAMXX/RTVPRDNB3( IN ProductPicture BLOB(1M),
                                   OUT ProductNumber CHAR(5) ) 1
language        C
specific        RTVPRD0003
deterministic
external name   DPOBJECT/RTVPRDNBR3
general;
```

SQL statement note

1. The stored procedure is defined with two parameters. The BLOB object is passed by value.

For the external stored procedure, you can specify the input parameter as a locator to the value rather than the actual value. You can use the AS LOCATOR clause only if the input parameter has a LOB data type or a distinct type based on a LOB data type. The AS LOCATOR clause is not allowed for SQL procedures.

5.2.3 Calling the CLI stored procedure

Once the RTVPRDNB3 external stored procedure is successfully registered in the system catalogs, it can be called from any interface that supports SQL CALL statement. The following embedded SQL code example illustrates how to call the procedure and how to pass LOB value as one of the parameters:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

EXEC SQL INCLUDE SQLCA;

EXEC SQL BEGIN DECLARE SECTION;
     SQL TYPE IS BLOB(1M) bin_ProductPicture;
     char chs_ProductNumber[ 5 ];
EXEC SQL END DECLARE SECTION;

void main( int argc, char **argv )
{
     EXEC SQL WHENEVER NOT FOUND GOTO badnews;
     EXEC SQL WHENEVER SQLERROR GOTO badnews;

     EXEC SQL DECLARE
     cur_Picture
     CURSOR FOR
     select Product_Picture
     from
     prodmast01; 1

     EXEC SQL OPEN cur_Picture;

     do
     {
        EXEC SQL FETCH cur_Picture into :bin_ProductPicture; 2
        EXEC SQL SET :chs_ProductNumber = '     ';
        EXEC SQL CALL RTVPRDNB3( :bin_ProductPicture, :chs_ProductNumber); 3
        printf( "The product number - %s\n", chs_ProductNumber );
     } while ( sqlca.sqlcode != 100 );
     exit(0);
     badnews:
     EXEC SQL CLOSE cur_Picture;
     exit( 1 );
}
```

Notes for CALLRPNBR3 C embedded SQL program

1. The SQL cursor is defined. We use this cursor to retrieve product pictures from the `prodmast01` table. The PRODUCT_PICTURE column is of user defined type PICTURE.
2. A PRODUCT_PICTURE value is fetched from the table. The value is implicitly cast from the PICTURE UDT into the sourced data type (BLOB in this case) before it is assigned to the bin_ProductPicture host variable.
3. The retrieved value, now stored in bin_ProductPicture host variable, is passed by value to the stored procedure.

5.2.4 Retrieving LOBs in CLI

In this section, we describe how to use the CLI to retrieve LOB data. We coded another stored procedure called RTVPRDNBR4. The procedure accepts two parameters: `chs_ProductNumber` as an INPUT parameter and `bin_ProductPicture` as an OUTPUT parameter. This time, we use the procedure to retrieve the product

picture for the given product number. The product number is passed by the invoking process. We focus our attention on the most important portion of the source code:

```
typedef struct
{
  unsigned long length;
  char data[ 1048576 ];
} BLOB1M;

BLOB1M     bin_ProductPicture;
...
SQLRETURN fun_Process()
{
 ...
  strcpy( chs_SqlStatement01, "select product_picture " );
  strcat( chs_SqlStatement01, "from prodmast01 " );
  strcat( chs_SqlStatement01, "where " );
  strcat( chs_SqlStatement01, "product_number = cast (? as SRLNUMBER)" ); 1

  nml_ReturnCode = SQLPrepare( nml_HandleToSqlStatement,
                               chs_SqlStatement01,
                                 SQL_NTS );
    if ( nml_ReturnCode != SQL_SUCCESS )
    {
      ...
    }

    nml_ReturnCode = SQLBindParam( nml_HandleToSqlStatement,
                                   1,
                                   SQL_CHAR,
                                   SQL_CHAR,
                                   sizeof( chs_ProductNumber ),
                                   0,
                                   ( SQLPOINTER ) chs_ProductNumber,
                                   ( SQLINTEGER * ) &nmi_PcbValue ); 2
   if ( nml_ReturnCode != SQL_SUCCESS )
   {
     ...
   }

  nml_ReturnCode = SQLExecute( nml_HandleToSqlStatement );
   if ( nml_ReturnCode != SQL_SUCCESS )
   {
    ...
   }

 nml_ReturnCode = SQLBindCol( nml_HandleToSqlStatement,
                              1,
                              SQL_BLOB,
                              ( SQLPOINTER ) bin_ProductPicture.data,
                              sizeof( bin_ProductPicture ),
                              ( SQLINTEGER * ) &nmi_PcbValue ); 3
 if ( nml_ReturnCode != SQL_SUCCESS )
  {
    ...
  }

    nml_ReturnCode = SQLFetch( nml_HandleToSqlStatement );
    if ( nml_ReturnCode != SQL_SUCCESS )
   {
    ...
   }
   else
   {
     return SQL_SUCCESS;
   }
}
```

Notes for RTVPRDNBR4 CLI stored procedure

1. The SQL request is assembled here. The PRODUCT_NUMBER column is of user defined type SRLNUMBER, so we need to explicitly cast the parameter marker to the appropriate type.
2. We use `SQLBindParam` function to bind the parameter marker.

3. To bind a column to an application variable, the application must call the `SQLBindCol` function. The sample application provides the following parameters to this function:
   - `nml_HandleToSqlStatement`: This is the handle to the SQL statement that contains the column.
   - `1`: We retrieve `Product_Picture` as the first and only column in the result set.
   - `SQL_BLOB`: This is the SQL data type of the application variable.
   - `( SQLPOINTER ) bin_ProductPicture.data`: This is a pointer to the buffer where the retrieved picture is stored at the fetch time.
   - `sizeof( bin_ProductPicture )`: This is the size of the buffer that stores the data retrieved from the column. Note that, in case of BLOB data type, you need to pass the size of `bin_ProductPicture` structure.
   - `( SQLINTEGER * ) &nmi_PcbValue`: This is an integer pointer that points to a location containing the length of the BLOB data returned at fetch time.

Chapter 6. DataLinks

This chapter describes:

- The role of DataLinks in applications and their use relative to LOBs
- The generic components of DataLinks
- The AS/400 operational environment to support the DataLinks components
- The creation of DataLinks in DB2 Universal Database for AS/400
- The considerations for working with DataLinks in DB2 UDB for AS/400
- An overview of working with DataLinks in an heterogeneous environment
- Backup/recovery considerations for DataLinks

6.1 A need for DataLinks

Chapter 2, “Large object support in DB2 UDB for AS/400” on page 3, described the potential role that large objects can play in modern applications. In particular, with the growth of Internet-based applications, the desire for organizations to capture and retain the interest of potential customers is driving the need to include types of data beyond the simple structured interface presented by characters and numerics. This new breed of unstructured data includes images, sound recordings, video clips, and complex text.

Large objects, in the form of BLOBs, CLOBS, and DBCLOBS, are now supported as data types for inclusion in DB2 Universal Database for AS/400 tables. Although the DB2 Universal Database architecture defines 2 GB as the maximum size of a LOB, the current OS/400 V4R4 implementation limits the size to 15 MB.

While a majority of unstructured data that an application needs to use is likely to fall below the 2 GB, or even the 15 MB limit, some will undoubtedly be larger. Video recordings are a prime example of data that can be very large. These file objects will need to overcome that limit.

As a further scenario, consider a user with thousands of file objects, for example, video recordings, images, or sound bites, stored on a hard drive of a PC server or in the Integrated File System of their AS/400. These files may simply be there for ease of storage, with hierarchical structure of file system directories being well suited to stream file management. Additionally, they may be currently used by PC based applications, such as audio and video players and graphics and drawing packages. New application requirements may then arise, which are best fulfilled by using an SQL table, to contain information about these file objects, for example, title, length, creation date, artist, and so forth. However, since the user already has the objects stored in a file directory, they may be reluctant to transfer them into the SQL table as LOB columns. Furthermore, it may not be feasible to move them from the file system if they need to be accessed by PC applications.

The DataLink data type extends the types of data that can be stored in an SQL table. The principle behind a DataLink is that the actual data stored in the table column is only a pointer to an object residing in a file system on any file server that supports DataLinks. The file object can be any file type. The method used to resolve the file object in an application is to store this pointer in the form of a Uniform Resource Locator (URL). This URL can use any of the following formats:

© Copyright IBM Corp. 2000

- file:
- http:
- https:

This means that a row in a table can be used to contain information about the file object in columns of traditional data types, and the object itself can be referenced using the DataLink data type. An application can use new SQL scalar functions to retrieve the name of the server on which the file object is stored and the path to it. The application can then hand control to software more appropriate in handling streaming data, for example a browser, to retrieve the object. This approach also has the advantage of deferring the physical movement of potentially very large objects from the server until needed by the client application. The access of such objects through the file system is also likely to provide better performance than through the relational database management system (RDBMS).

However, there are a number of important considerations if the RDBMS is to be used to effectively manage unstructured data that is stored in a file system. The two major considerations are:

- There has to be some relationship between the relational data and the file data.
- This relationship must be managed, particularly in the areas of data integrity, data access control, and data backup/recovery to ensure high-application availability.

Relational database management systems provide the robust environment that is lacking in file systems by applying that environment to the DataLinks.

Figure 92 shows a relational table with the LOB data actually stored in columns within each row along with the traditional structured data columns.

Figure 93 shows the same data, but the LOB data is stored as DataLinks within each row. Each DataLink column points to a file server, for example, the AS/400 Integrated File System, and a directory and file object within that server.

SOLD	ONHAND	RATING	ARTIST	TITLE	COVER	VIDEO	MUSIC	SCRIPT
234	59	PG-13	Arnold	The Exter minator				
13	45	R	Kevin	Dancing with Bulls				
1295	209	G	Glenn	101 Doll Imitations				
379	112	G	Buzz	Toy Glory				

Figure 92. Large objects in tables: The LOB approach

SOLD	ONHAND	RATING	ARTIST	TITLE	COVER	VIDEO	MUSIC	SCRIPT
234	59	PG-13	Arnold	The Exterminator	file://AS400WS/covers/ext.jpg	file://AS400WS/videos/ext.mpg	file://AS400WS/music/ext.wav	file://AS400WS/script/ext.lwp
13	45	R	Kevin	Dancing with Bulls	file://AS400WS/covers/dbull.jpg	file://AS400WS/videos/dbull.mpg	file://AS400WS/music/dbull.wav	file://AS400WS/script/dbull.lwp
1295	209	G	Glenn	101 Doll Imitations	file://AS400WS/covers/di101.jpg	file://AS400WS/videos/di101.mpg	file://AS400WS/music/di101.wav	file://AS400WS/script/di101.lwp
379	112	G	Buzz	Toy Glory	file://AS400WS/covers/toyg.jpg	file://AS400WS/videos/toyg.mpg	file://AS400WS/music/toyg.wav	file://AS400WS/script/toyg.lwp

Figure 93. Large objects in tables: The DataLink approach

There are a number of additional benefits from the DataLinks approach. File systems, including the AS/400 Integrated File System (IFS), are able to store any type of stream file. The current scope includes all of the types referenced in Figure 93 and more. However, technological advances will, over time, no doubt give birth to new ways of storing complex unstructured data using new file types. In other words, using the file system approach to unstructured data storage provides a high degree of future proofing to applications.

In summary, any application that could benefit from significant content management capabilities and robust and secure file management would be a candidate for deploying DataLinks. Examples include:

- Web-based electronic commerce
- Intranet applications
- Applications with links to computer-aided design and manufacturing (CAD/CAM)
- Library and asset management applications (for example, entertainment industry, medical applications using X-rays, and so on)

6.2 DataLinks components

DataLink support on the AS/400 system is comprised of four major components:

- The DataLink data type
- The DataLink File Manager (DLFM)
- The DataLink filter
- DBMS/DLFM APIs

Figure 94 summarizes the DataLinks components and the API interfaces they use to communicate.

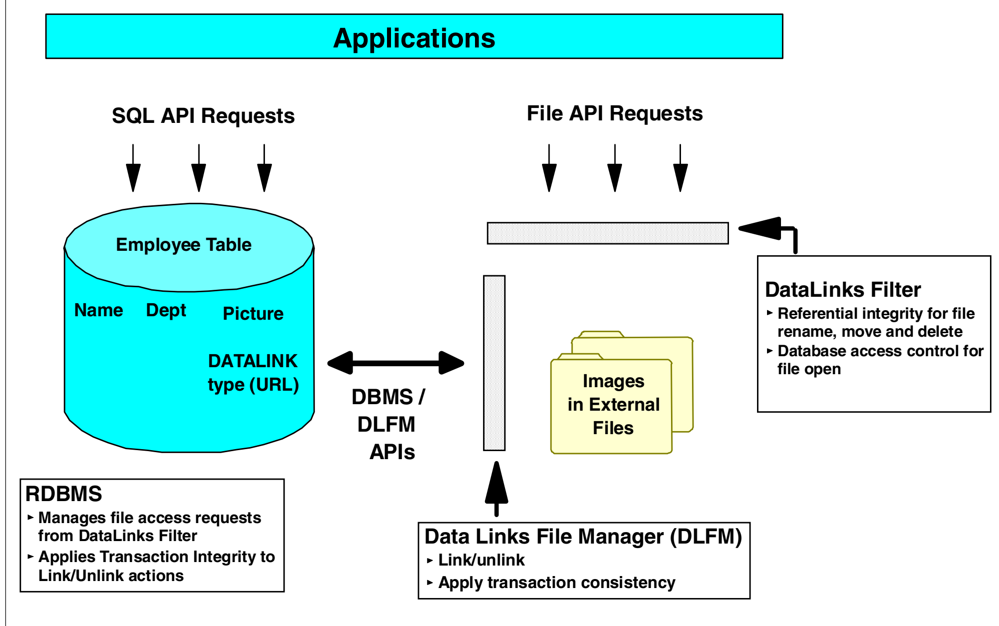


Figure 94. DataLinks components summary

6.2.1 DataLink data type

The DataLink data type is new to DB2 Universal Database for AS/400 in V4R4 with the Database Fixpack. When you use Operations Navigator to create a table, the data type can be found in the drop-down list box when you insert a column as illustrated in Figure 95.

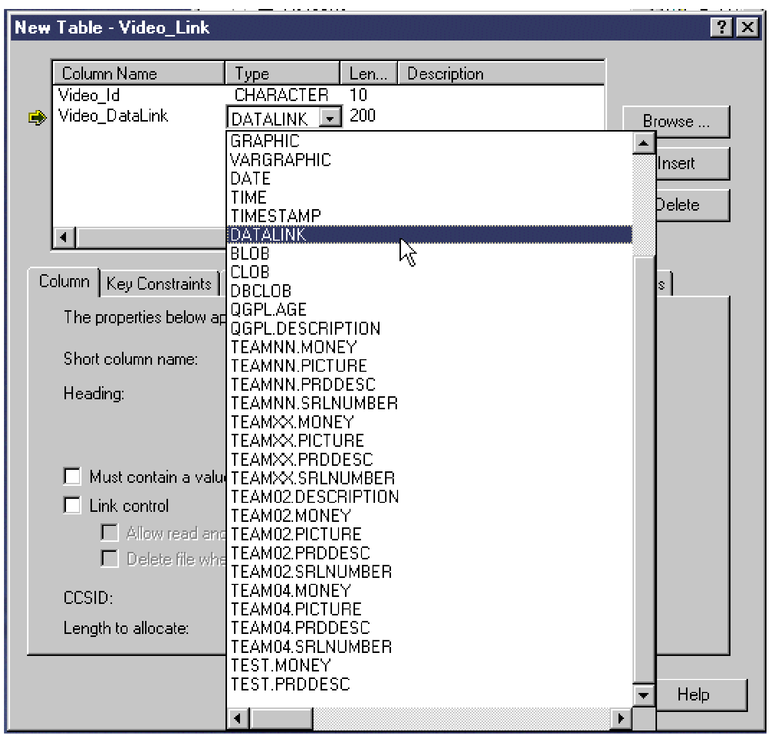


Figure 95. Inserting a column of data type DataLink

The SQL statements that support the data type are `CREATE TABLE` and `ALTER TABLE`. The only default value that you can specify for a column of type DataLink is null. Because the DataLink is not compatible with any host variable data type, the only interface that allows access is SQL. The underlying format of the data in a DataLink column is character format, and you can use a number of SQL scalar functions to retrieve the Datalink value in this format. When you insert or update data in a DataLinks column, you must use the DLVALUE scalar function with SQL `INSERT` or `UPDATE`.

6.2.2 DataLink file manager

The DataLink File Manager (DLFM) is the core component of the support for DataLinks on any platform. It controls and maintains the status of the links between the RDBMS tables and their associated file system objects. It does this by creating and maintaining metadata for each table and file. On the AS/400 system, this metadata is stored in a number of tables in the QDLFM collection (library). Figure 96 on page 152 shows the objects in the QDLFM library.

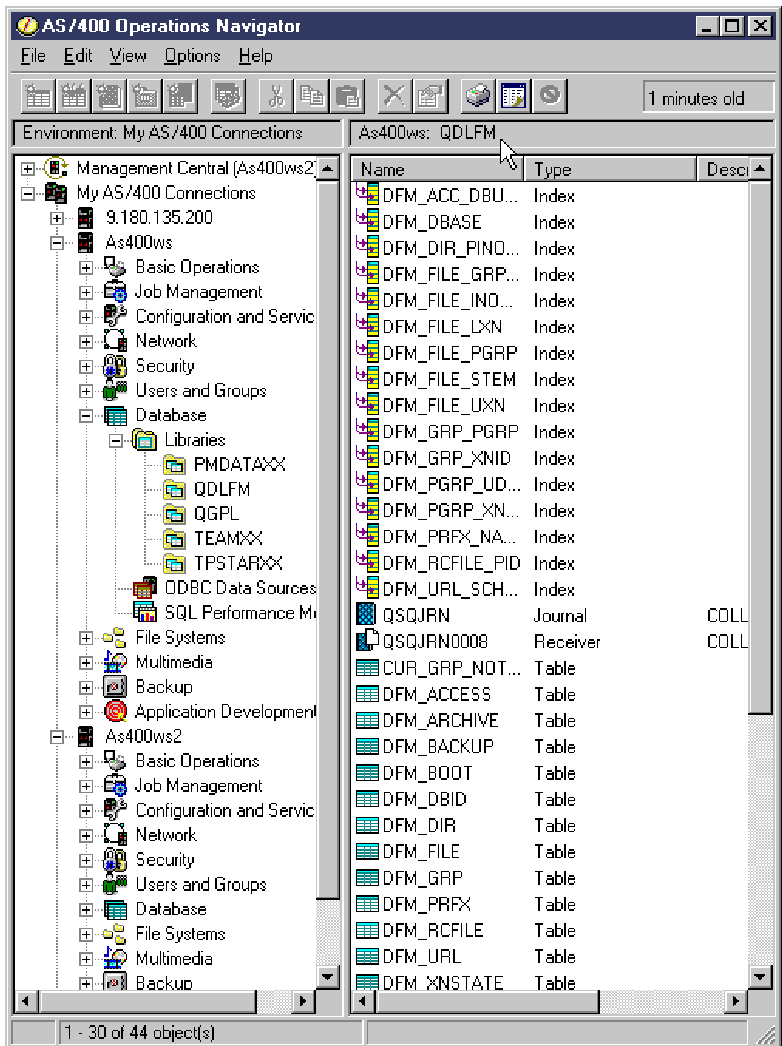


Figure 96. DLFM objects in library QDLFM

Important

No explicit user interface is provided to these objects. We strongly recommend that you do not make any changes to their content. However, if you view the content of some of the tables, you will find useful information on the setup and state of the DataLink environment. Some examples are shown in 6.3.2.4, “Additional DataLink management interfaces” on page 161.

The DLFM handles the linking and unlinking of files with tables. Because it is using DB2 Universal Database for AS/400 tables (in library QDLFM) for managing the environment, it can also manage the integrity of those links through commitment control by treating link and unlink actions as transactions.

One of the most important aspects of the DataLinks architecture is that it is designed so that the DLFM can reside on a remote AS/400 system running V4R4 or higher. It achieves this by using a standard set of APIs between the RDBMS and DLFM components. This approach allows relational tables on one system to

link to files on the same or another system, either locally or remotely. This flexible approach allows the files to reside on the most appropriate system from an overall application standpoint. Such flexibility can also aid performance and minimize network costs by allowing file servers to be positioned close to the end users, enabling files to be delivered over shorter distances. Figure 97 shows an example of the type of DataLink environment that could be deployed. Relational DB2 Universal Database for AS/400 tables on the Rochester system are linked to:

- Files in directories in its own Integrated File System
- Files in directories on the London AS/400s Integrated File System

A DLFM is running on both systems.

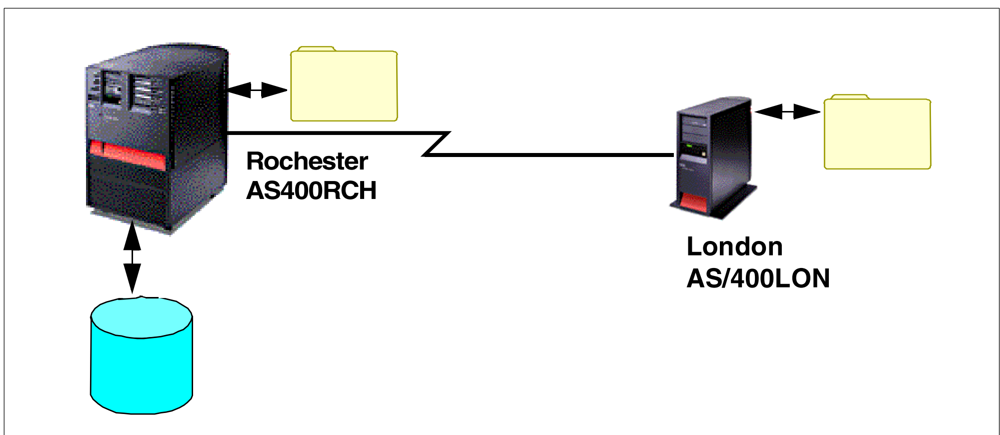


Figure 97. Distributed heterogeneous DLFM environment

6.2.3 DataLink filter

The DataLink filter is an internal function that is invoked whenever a program attempts operations, through a file system interface, on a file that is within a directory registered as containing linked files. It determines if the file is actually linked and, depending on the attempted operation, may also check if the user is authorized to access the file and open it. If it finds that the file is linked, it will impose a form of referential constraint and reject any attempt to move, rename, or delete the file. This aspect is covered in more detail in 6.4.6, “Using the DataLink access control token” on page 186.

The DataLink Filter is invoked regardless of whether a file is linked or unlinked. Because invoking the DataLink Filter generates resource overhead, it is only executed when the file being accessed is in a registered directory or in a directory path below a registered directory. This is covered in more detail in 6.3.2.1, “Adding a prefix” on page 157.

6.2.4 APIs

There are essentially three API interfaces in the DataLinks environment:

- The interface to the relational table. This is through SQL and uses new scalar functions to work with the DataLink in the table rows. No OS/400 native I/O interface is provided to the DataLink data type.
- The interface to objects in the file system from file API requests. Access to linked or unlinked files residing in a registered directory is intercepted by the

DataLinks Filter. Access can be directly from file system programs and utilities or, in the case of the AS/400 IFS, from an AS/400 ILE C program.

- The interface between the RDBMS and the DLFM. These APIs allow the RDBMS to communicate link and unlink activities to the DLFM, and the DLFM to communicate file access requests to the RDBMS if the option has been taken to use the RDBMS to control file access.

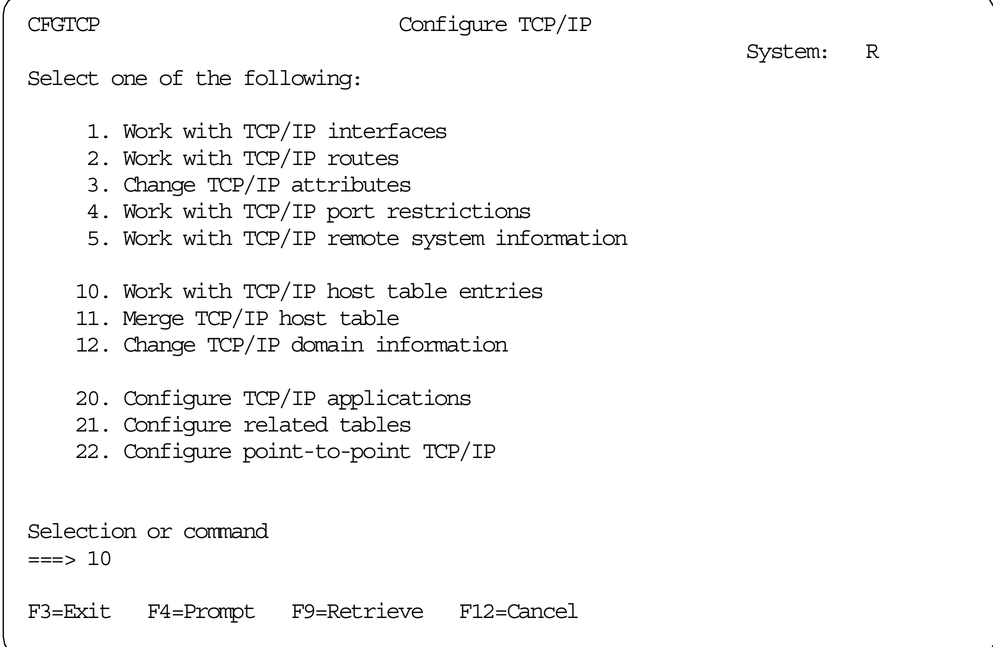
6.3 DataLinks system configuration

A number of basic configuration tasks are necessary to enable the DataLinks environment to be defined.

You must configure TCP/IP on all systems that you want to participate in the environment. That is, those that will host the SQL tables in which DataLink columns are created, and those that will host the file objects to be linked. In the case of a single AS/400 system where IFS files are going to be linked to DB2 Universal Database for AS/400 tables, the configuration is a single process. The URL in the DataLink column used to reference the file object contains the name of the file server. You must configure this name or register a TCP/IP name server. Enter the command:

```
CFGTCP
```

Then, enter option `10` as shown in Figure 98.



```
CFGTCP                          Configure TCP/IP
                                                             System:   R
Select one of the following:

     1. Work with TCP/IP interfaces
     2. Work with TCP/IP routes
     3. Change TCP/IP attributes
     4. Work with TCP/IP port restrictions
     5. Work with TCP/IP remote system information

    10. Work with TCP/IP host table entries
    11. Merge TCP/IP host table
    12. Change TCP/IP domain information

    20. Configure TCP/IP applications
    21. Configure related tables
    22. Configure point-to-point TCP/IP


Selection or command
===> 10

F3=Exit   F4=Prompt   F9=Retrieve   F12=Cancel
```

Figure 98. Adding the TCP/IP server name

Figure 99 shows the next screen displayed. If the system is not already configured, type `1` and the IP address on the top line, and press Enter to add a new entry. The next screen allows you to type in the name of the AS/400 server.

If the IP address is configured, but the AS/400 system name you want to use in your DataLink columns is not, type `2` next to the appropriate IP address and press

Enter. The next screen allows you to enter an additional server name. In our case, this is AS400RCH.

```
                   Work with TCP/IP Host Table Entries
                                                                   System:   RCH
Type options, press Enter.
  1=Add   2=Change   4=Remove   5=Display   7=Rename

     Internet         Host
Opt  Address          Name

  2  1.1.12.30        AS23
                      AS400WS
     1.1.92.31        AS400LON
     1.11.208.1       AS89
     127.0.0.1        LOOPBACK
                      LOCALHOST



                                                                         B
F3=Exit   F5=Refresh   F6=Print list   F12=Cancel   F17=Position to
```

Figure 99. Adding the IP server name: IP address already configured

Next, you must ensure that the AS/400 system that will host the relational tables with the DataLink columns has a relational database directory entry. You can define this system as *LOCAL in the system's relational database directory by running the CL command:

```
WRKRDBDIRE
```

The screen shown in Figure 100 is displayed.

```
                Work with Relational Database Directory Entries

Position to  . . . . . .

Type options, press Enter.
  1=Add   2=Change   4=Remove   5=Display details   6=Print details

        Relational          Remote
Option  Database            Location                 Text

        AS400RCH            *LOCAL






                                                                         Bottom
F3=Exit   F5=Refresh   F6=Print list   F12=Cancel
(C) COPYRIGHT IBM CORP. 1980, 1999.
```

Figure 100. Adding the relational database directory entry (WRKRDBDIRE)

If there is no entry with a remote location name of *LOCAL, you must add this entry for the local AS/400 system. You should use the AS/400 system name that you used for the TCP/IP server as the relational database name. This enables the

DLFM to communicate with DB2 Universal Database for AS/400 within the local AS/400 system.

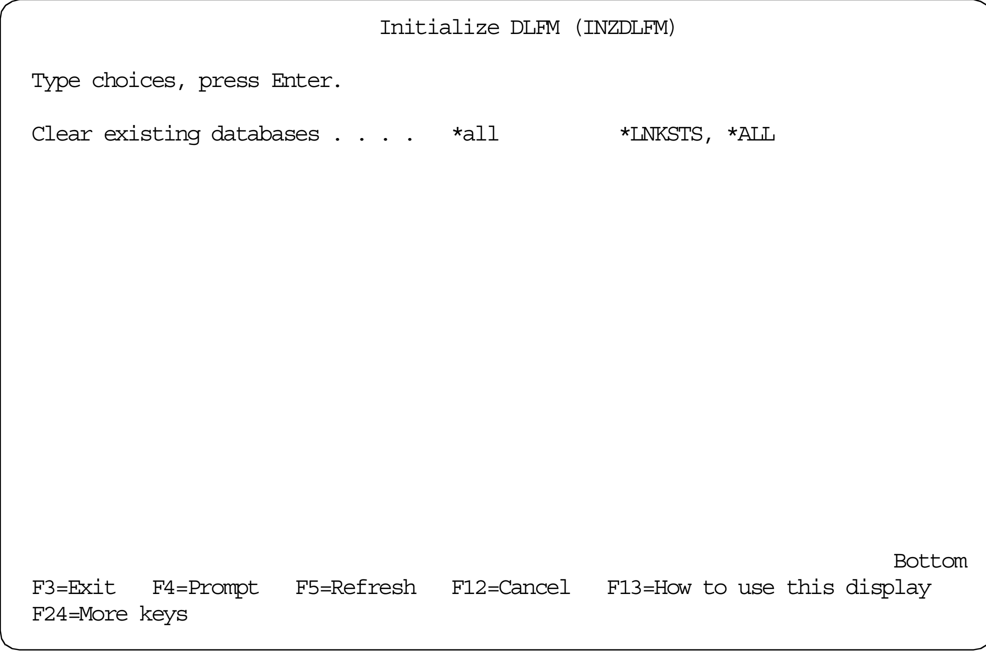
Note that the *LOCAL entry in the RDB directory is required only on the system where the tables reside. In a distributed environment, if the DLFM server is running on the system with no linked tables, there is no need for either the *LOCAL RDB entry or the remote entry for the system where the linked tables reside.

6.3.1 Initializing the DLFM server

In 6.2.2, “DataLink file manager” on page 151, we explained how the DLFM manages the environment by keeping and maintaining metadata in a number of tables in the QDLFM collection. These tables must be set up and initialized. To do this, you must run the CL command `INZDLFM` as follows:

```
INZDLFM *ALL
```

The prompted command is shown in Figure 101.



```
                          Initialize DLFM (INZDLFM)

Type choices, press Enter.

Clear existing databases . . . .   *all          *LNKSTS, *ALL

                                                                      Bottom
F3=Exit   F4=Prompt   F5=Refresh   F12=Cancel   F13=How to use this display
F24=More keys
```

Figure 101. Initializing the DLFM tables (INZDLFM)

The `INZDLFM` command with the *LNKSTS parameter clears the tables in the QDLFM library that contain the link status of DataLinks. You then need to relink all your files.

Attention

You should use this command with *caution*. Under normal circumstances, its use should be viewed as a one-time exercise. If you run it again once the environment has been set up and Datalinks have been created, the system attempts to re-create the internal DLFM tables and delete any rows of data that they contain. You will then be faced with re-creating the entire environment and relinking your files. You are *strongly* advised to add the backup of the QDLFM library to your existing installation backup policies. If you should accidentally rerun the command, you will have to re-IPL the system. This is because DLFM mounts the file directories that are registered to it (refer to 6.3.2.1, “Adding a prefix” for more details) at IPL time, and rerunning the initialization causes the mount points to be lost.

6.3.2 DLFM configuration

Once the generic DLFM environment has been initialized, you begin to customize it for your own application environment. This involves defining information to DLFM about your specific relational table and file system entities that will be included in the DLFM configuration. You should note that *IOSYSCFG special authority is needed to perform these administrative functions.

6.3.2.1 Adding a prefix

DLFM needs to know the file system directories where it will find file objects to be linked and which will come under its management and control. For example, if you wanted to define the directory "Videos" as one containing files to be linked, you use the following CL command example:

```
ADDPFXDLFM PREFIX('/videos')
```

Figure 102 on page 158 shows it as a prompted command.

```
                         Add Prefix to DLFM (ADDPFXDLFM)

Type choices, press Enter.

Prefix:                          PREFIX
  Name . . . . . . . . . . . . .              > '/videos'




                          + for more values
Source file  . . . . . . . . . . SRCFILE
  Library  . . . . . . . . . . .                   *LIBL
Source member  . . . . . . . . . SRCMBR








                                                                           Bottom
F3=Exit   F4=Prompt   F5=Refresh   F12=Cancel   F13=How to use this display
F24=More keys
```

Figure 102. ADDPFXDLFM command prompt

Note

You may use the format '/videos' or '/videos/' for the PREFIX parameter, but you should be consistent in their usage. The directory that you are registering *must* exist at the time of registration. The SRCFILE and SRCMBR parameters allow you to pre-build the prefix names in a separately maintained source file member for input to the command. Each prefix name would occupy a line in the source member.

When rows containing DataLinks columns are inserted into a table, if the referenced files are to be linked, the DLFM checks to ensure that the files exist and that they are within a file directory that is a registered prefix or within a sub-directory of a registered prefix. For example, if the directory TEAMXX has been registered as follows:

```
ADDPFXDLFM PREFIX('/teamxx')
```

then files in any of the following paths are valid candidates for linking:

- /teamxx
- /teamxx/multimedia
- /teamxx/multimedia/sound_bites

To minimize the performance overhead incurred when the DLFM checks the registered prefixes, you should keep the number of prefixes to a minimum. For optimum manageability, you should keep files to be linked in sub-directories within the directories defined as prefixes, not within the registered directories themselves. This allows you to manipulate those sub-directories without affecting the mount points that have been set up at IPL. Therefore, using the above example, if you wanted to replace the complete set of sound bite files in the

sub-directory '/sound_bites', you could simply delete the complete sub-directory and restore the new version. Because '/sound_bites' is not a registered prefix, deleting it will not affect the file mount point.

Note

There is a command to remove prefixes (see 6.3.2.4, “Additional DataLink management interfaces” on page 161). However, this would not be a commonly used function since prefixes can only be removed if there are no linked files anywhere in the directory path within and below the prefix directory.

6.3.2.2 Adding a host database

DLFM needs to know the AS/400 systems and the libraries within those systems where relational tables will be found that need to link to files in a file system. Note that both the local AS/400 system and any remote AS/400 system that may generate link requests must be known to DLFM. For example, you have a local system, AS400RCH, and a remote system, AS400LON, with libraries as indicated in Table 10.

Table 10. Host database registration example

AS/400 system	Libraries with tables to be linked
AS400RCH	MULTIM01 MULTIM02
AS400LON	IMAGMAST

The local AS400RCH system tables link to files in the local IFS and the remote system’s IFS. The remote system tables only link to files in its own IFS. To register the necessary host database information on system AS400RCH, you use the following CL commands:

```
ADDHDBDLFM HOSTDBLIB((MULTIM01)(MULTIM02)) HOSTDB(AS400RCH)
```

Note

The value of the HOSTDB parameter must be set to the relational database name that you used for the *LOCAL RDB entry on the AS400RCH system.

On the AS400LON system, use the following command:

```
ADDHDBDLFM HOSTDBLIB(IMAGMAST) HOSTDB(AS400LON)
ADDHDBDLFM HOSTDBLIB((MULTIM01)(MULTIM02)) HOSTDB(AS400RCH)
```

Note

London's system will have link requests coming from Rochester (remote) and London (local). You need to register libraries from both systems. For the local system, you use the HOSTDB name, as specified for the *LOCAL RDB entry on the AS400LON machine. Similarly, for the remote system, you use the HOSTDB name, as specified for the *LOCAL RDB entry on the *AS400RCH* machine. In other words, to register libraries on the remote system, you need to know the name of the relational database for the *LOCAL entry *on the remote system*.

Note also that, in this scenario, there is no need for the AS400RCH RDB entry on the AS400LON machine. However, this entry may be required for some other functionality, such as DRDA and/or DDM. To summarize, the RDB entries required by the DataLink interface on the Rochester system are shown in Table 11.

Table 11. RDB entries on the AS400RCH system

Relational database	Remote location
AS400RCH	*LOCAL

The RDB entries on the London system are shown in Table 12.

Table 12. RDB entries on the AS400LON system

Relational database	Remote location
AS400LON	*LOCAL

The prompted ADDHDBDLFM command is shown in Figure 103.

```
                    Add Host Database to DLFM (ADDHDBDLFM)

 Type choices, press Enter.

 Host database library:          HOSTDBLIB
   Name . . . . . . . . . . . . .              > MULTIM01

   Name . . . . . . . . . . . . .              > MULTIM02
                           + for more values
 Host database instance . . . . . HOSTDBINST     QSYS
 Host database  . . . . . . . . . HOSTDB         AS400RCH



 Source file  . . . . . . . . . . SRCFILE
   Library  . . . . . . . . . . .                  *LIBL
 Source member  . . . . . . . . . SRCMBR



                                                                    Bottom
 F3=Exit   F4=Prompt   F5=Refresh   F12=Cancel   F13=How to use this display
 F24=More keys
```

Figure 103. ADDHDBDLFM command prompt

Note

The ADDHDBDLFM command has an additional parameter, the host database instance (HOSTDBINST). This is always QSYS on the AS/400 system and does not need to be specified as the parameter defaults to this value. It is present for compatibility with DB2 Universal Database on other platforms which, unlike DB2 Universal Database for AS/400, supports multiple database instances.

The SRCFILE and SRCMBR parameters allow you to pre-build the host database names in a separately maintained source file member for input to the command. The format of the input should be: HOSTDBLIB HOSTDBINST HOSTDB. Using Figure 103 as an example, the source file member would consist of the two following entries:

```
MULTIM01 QSYS AS400RCH
MULTIM02 QSYS AS400RCH
```

6.3.2.3 Additional configuration commands

The functions of registering prefixes and host databases are both provided through CL commands. Sometimes, you may also want to remove registered prefixes. However, this administrative function is not accessed through CL commands but through the QShell interactive interface. The following example shows how to use the `dfmadmin` command in the QShell. To use the command, type the following statement into an OS/400 command prompt, and press Enter:

```
QSH
```

You are now presented with the QShell command interface. To remove the prefix '/teamzz,' which you accidentally mis-typed, run the following QShell command:

```
dfmadmin -del_prefix
```

Press Enter. When prompted, type:

```
/teamzz
```

Press Enter. Press Enter again to terminate the command.

Attention

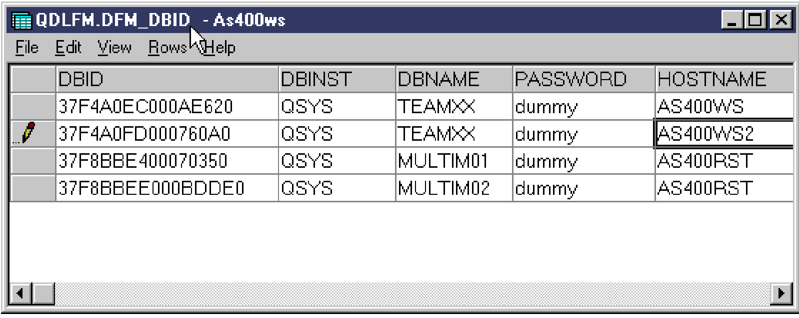
Exercise caution if using the QShell commands. Their use should be restricted to deleting inappropriate prefixes or host database names. For example, a particular directory prefix is no longer needed and it does not contain any linked files, or a library referenced in the host database entries is being replaced by a new library.

6.3.2.4 Additional DataLink management interfaces

Viewing the content of some of the database tables in the QDLFM library through the Operations Navigator Quick View function can be a useful way of determining the current state of the DataLink environment. The tables in QDLFM on which you should focus are:

```
DFM_DBID
DFM_FILE
DFM_PRFX
```

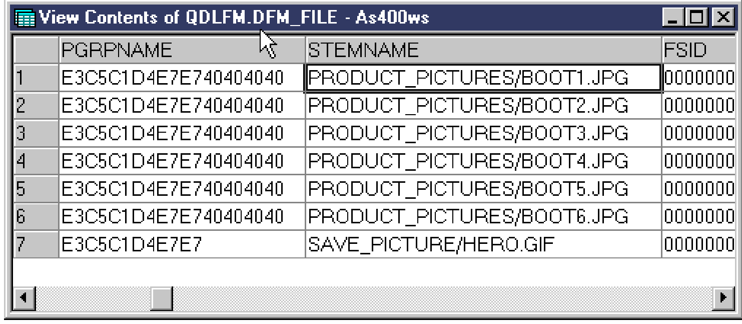
From the Operations Navigator window, right-click on the **DFM_DBID** table icon, and select **Quick View** to display the rows in the table. You should *not* double-click on the table, since this opens it for update, exposing you to the danger of accidentally overwriting or deleting values in the table rows. You see a display similar to that in Figure 104. It contains one row for each library that has been registered with the DLFM and the name of the system on which that library resides.



DBID	DBINST	DBNAME	PASSWORD	HOSTNAME
37F4A0EC000AE620	QSYS	TEAMXX	dummy	AS400WS
37F4A0FD000760A0	QSYS	TEAMXX	dummy	AS400WS2
37F8BBE400070350	QSYS	MULTIM01	dummy	AS400RST
37F8BBEE000BDDE0	QSYS	MULTIM02	dummy	AS400RST

Figure 104. Table DFM_DBID in QDLFM library: Viewed with Operations Navigator

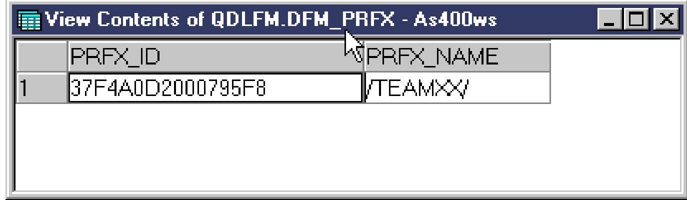
Repeat the Quick View operation on the table DFM_FILE. The Results window should resemble that shown in Figure 105. This table has one row for each file that is linked and includes the directory path to the file.



	PGRPNAME	STEMNAME	FSID
1	E3C5C1D4E7E740404040	PRODUCT_PICTURES/BOOT1.JPG	0000000
2	E3C5C1D4E7E740404040	PRODUCT_PICTURES/BOOT2.JPG	0000000
3	E3C5C1D4E7E740404040	PRODUCT_PICTURES/BOOT3.JPG	0000000
4	E3C5C1D4E7E740404040	PRODUCT_PICTURES/BOOT4.JPG	0000000
5	E3C5C1D4E7E740404040	PRODUCT_PICTURES/BOOT5.JPG	0000000
6	E3C5C1D4E7E740404040	PRODUCT_PICTURES/BOOT6.JPG	0000000
7	E3C5C1D4E7E7	SAVE_PICTURE/HERO.GIF	0000000

Figure 105. Table DFM_FILE in QDLFM library: Viewed with Operations Navigator

Finally, repeat the Quick View operation on the table DFM_PRFX. You see a Results window similar to that shown in Figure 106. This has one row for each prefix that has been registered with the DLFM.



	PRFX_ID	PRFX_NAME
1	37F4A0D2000795F8	/TEAMXX/

Figure 106. Table DFM_PRFX in QDLFM library: Viewed with Operations Navigator

6.3.3 Starting the DLFM server

Once the generic DLFM environment has been initialized, you are ready to start the DLFM. It must be started on any systems that contain file objects to be linked. On the AS/400 system, the DLFM is, in fact, a TCP/IP server job. To start it, run the following CL command:

```
STRTCPSVR SERVER(*DLFM)
```

The screen shown in Figure 107 should be displayed. You will notice that this screen is not a conventional command display. The reason for this is that the DLFM server is started through the OS/400 QShell interface, and it is the interactive shell interface that is displayed. Once the DLFM server has started, you must press Enter to terminate the QShell terminal session and return to the OS/400 command interface.

Once started, the DLFM would normally be permanently active. However, to terminate it in a controlled way, there is the following CL command:

```
ENDTCPSVR SERVER(*DLFM)
```

Once execution of this command has completed, you receive a the message:

```
DLFM server ended
```

```
Create detach session message queue.
  DLFM server started.
  Press ENTER to end terminal session.









===>

F3=Exit F4=End of File F6=Print F9=Retrieve F17=Top
F18=Bottom  F19=Left   F20=Right F21=User Window
```

Figure 107. Starting the DLFM server jobs

When the DLFM has successfully started, there will be a number of jobs active in the OS/400 QSYSWRK subsystem. These are shown in Figure 108 and Figure 109 on page 164.

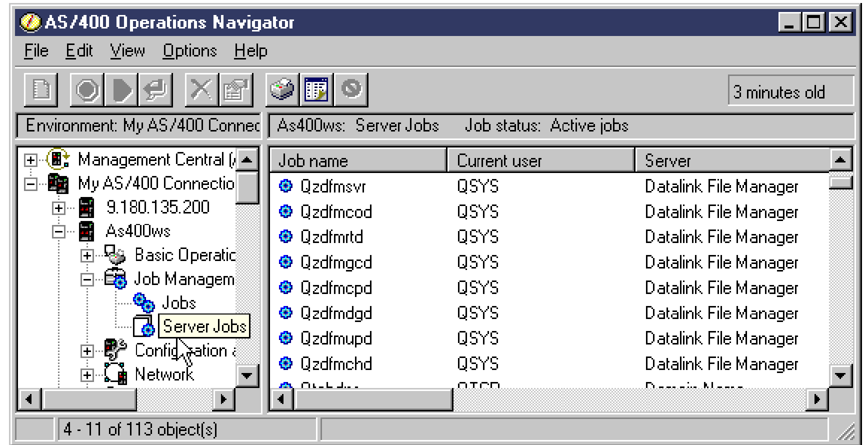


Figure 108. DLFM server jobs in Operations Navigator

```
                          Work with Active Jobs                      AS400WS
                                                            09/24/99  15:23:29
CPU %:     .0     Elapsed time:   00:00:00     Active jobs:   197

Type options, press Enter.
  2=Change   3=Hold   4=End   5=Work with   6=Release   7=Display message
  8=Work with spooled files   13=Disconnect ...

Opt  Subsystem/Job  User        Type  CPU %  Function        Status
       QZDFMCHD     QSYS        BCI      .0                   TIMW
       QZDFMCOD     QSYS        BCI      .0                   DEQW
       QZDFMCPD     QSYS        BCI      .0                   SIGW
       QZDFMDGD     QSYS        BCI      .0                   DEQW
       QZDFMGCD     QSYS        BCI      .0                   SIGW
       QZDFMRTD     QSYS        BCI      .0                   DEQW
       QZDFMSVR     QSYS        BCH      .0                   TIMW
       QZDFMUPD     QSYS        BCI      .0                   DEQW

                                                                      Bottom
Parameters or command
===>
F3=Exit   F5=Refresh       F7=Find      F10=Restart statistics
F11=Display elapsed data   F12=Cancel   F23=More options   F24=More keys
```

Figure 109. DLFM server jobs in subsystem QSYSWRK

6.4 Using DataLinks with SQL

Once you have registered the prefixes and host database names, and the DLFM server has been started, you can start to link to objects in the file system.

You begin by defining and creating the tables you need in the libraries that have been registered to the DLFM. There are a number of options to be considered as part of the table creation process as you define the DataLinks columns. These are covered in detail in 6.4.1, “DataLink options: General” on page 165. The options you choose will be governed primarily by the nature of the applications that will be using the DataLinks. It is important to note that the DataLinks architecture defines a number of possible attributes for creating, processing, and managing linked files. V4R4 delivers the first stage of the implementation for DB2

Universal Database for AS/400 and, therefore, delivers a subset of those attributes.

6.4.1 DataLink options: General

The DataLinks architecture defines the following attributes that are summarized in Table 13 on page 167:

- **Linktype:** The only link type currently defined is the *URL*.
- **Link Control:** This is the basic attribute that defines whether file system objects will be linked to a DataLink row in an RDBMS table. The choice is between *No Link Control* and *File Link Control***:**
  - *No Link Control*: When rows are inserted into the table, there would be no links established to the file objects referenced in the DataLink column. No check is made to verify that the file server can be accessed or that the file object being referenced even exists. However, the syntax of the URL is validated. While the No Link Control option still provides value in terms of new application potential, it does not enable you to benefit from the management and integrity control provided by the File Link Control option.
  - *File Link Control:* When a row is inserted into the table, the DLFM immediately attempts to establish a link to the referenced file object. The file server must be accessible, and the file object must exist. Once the link has been established, the DLFM maintains control of the link through its metadata. A file object may only be linked to one table row. However, a table row may contain multiple DataLink columns as long as each is linked to a different file object. Once a file has been linked, it may not be moved, deleted, or renamed. Deleting the table row unlinks the associated file. Updating the DataLink value in the table row causes the original referenced file to be unlinked while the new referenced file is linked.
- **Integrity:** This attribute controls the level of data integrity between the database server and the file server. The two options are *Integrity All* and *Integrity Selective***:**
  - *Integrity All*: Any linked file object referenced by a DataLink column is considered to be under the control of the RDBMS, and attempts to rename, delete, or move the file object from a file system interface are rejected.
  - *Integrity Selective*: Any linked file object referenced by a DataLink column is considered to be under the control of the RDBMS only if the file server has the DataLinks Filter installed. *This option is not supported by V4R4 of DB2 Universal Database for AS/400*.
- **Read Permission:** This defines where file object read access is controlled. The choices are *Read Permission FS* and *Read Permission DB***:**
  - *Read Permission FS*: The file system controls whether a user has the necessary authority to perform a read operation on a linked file system object. No prior access to the associated RDBMS table is required.
  - *Read Permission DB*: The RDBMS controls whether a user may perform a read operation on a linked file system object. Assuming the file system object has been given no public access authority, it can only be read by first accessing the DataLink value in the database table and retrieving an access control token. This is covered in more detail in 6.4.6, “Using the DataLink access control token” on page 186.

- **Write Permission:** This defines whether a user can write to the file object. The choices are *Write Permission FS* and *Write Permission Blocked*:
  - *Write Permission FS*: The file system controls whether a user has the necessary authority to perform a write operation to a linked file system object. No prior access to the associated RDBMS table is required.
  - *Write Permission Blocked*: A file system object cannot be written to through any interface because it is owned by the DLFM. V4R4 of DB2 Universal Database for AS/400 enforces this option if Read Permission DB has been selected.
- **Recovery:** This attribute specifies whether point-in-time recovery of linked files will be supported. The two options are *Recovery Yes* and *Recovery No***:**
  - *Recovery Yes*: Point-in-time recovery is achieved by the RDBMS ensuring that backup copies of the linked files are made as needed. It is only valid when Integrity All and Write Permission Blocked are also specified. *This option is not* supported by V4R4 of DB2 Universal Database for AS/400.
  - *Recovery No*: Point-in-time recovery is not supported.
- **On Unlink**: This attribute determines the action to be taken when the RDBMS controls write operations to a linked file (Write Permission Blocked), and the file is unlinked either through the DataLink value in the associated table row being updated or the row being deleted. Note that updating a row's DataLink value effectively deletes the current file link and replaces it with a new file link. The option is not applicable when write operations are controlled by the file system (Write Permission FS). The options are *On Unlink Restore* and *On Unlink Delete*:
  - *On Unlink Restore*: When a file is unlinked, this option will ensure that the file's ownership and permissions are restored to their state at the time that the file was linked. If the owner no longer exists in the file system, a default owner may be established, but this action depends on the particular file system involved. Apart from only being a valid option when Write Permission Blocked is also specified, Integrity All is also a prerequisite.
  - *On Unlink Delete*: When a file is unlinked, it is automatically deleted. This option is only valid when Read Permission DB and Write Permission Blocked are also specified.

Table 13. Architected DataLink attributes: Permissible combinations

Link control	Integrity	Read permission	Write permission	Recovery	On unlink
N/A	N/A	N/A	N/A	N/A	N/A
FILE	ALL	FS	FS	NO	N/A
FILE	ALL	FS	BLOCKED	NO	RESTORE
FILE	ALL	FS	BLOCKED	YES	RESTORE
FILE	ALL	DB	BLOCKED	NO	RESTORE
FILE	ALL	DB	BLOCKED	NO	DELETE
FILE	ALL	DB	BLOCKED	YES	RESTORE
FILE	ALL	DB	BLOCKED	YES	DELETE
FILE	SELECTIVE	FS	FS	NO	N/A

Note:

- N/A means not applicable
- The shaded rows indicate the combination of options that are supported by V4R4 of DB2 Universal Database for AS/400.

6.4.2 DataLink options: DB2 Universal Database for AS/400

You can see from Table 13 that V4R4 of DB2 Universal Database for AS/400 supports a subset of the architected options. We now review how those options are defined and implemented.

Tables that are to contain rows with DataLink columns can be created through the Operations Navigator or through a 5250 session and Interactive SQL. To use the Operations Navigator interface, you must right-click on a library object and select **New->Table**. The New Table dialog appears as shown in Figure 110.

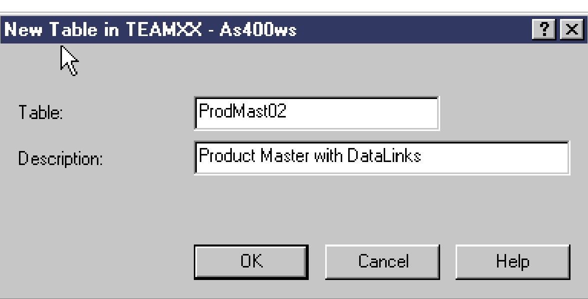


Figure 110. New table dialog

Once you type the name of your new table and its optional description and click **OK**, the table definition can begin. You start inserting columns into the table, defining the appropriate data type for each column from the drop-down list box in the normal way. Figure 111 on page 168 shows a DataLink type column about to be inserted.

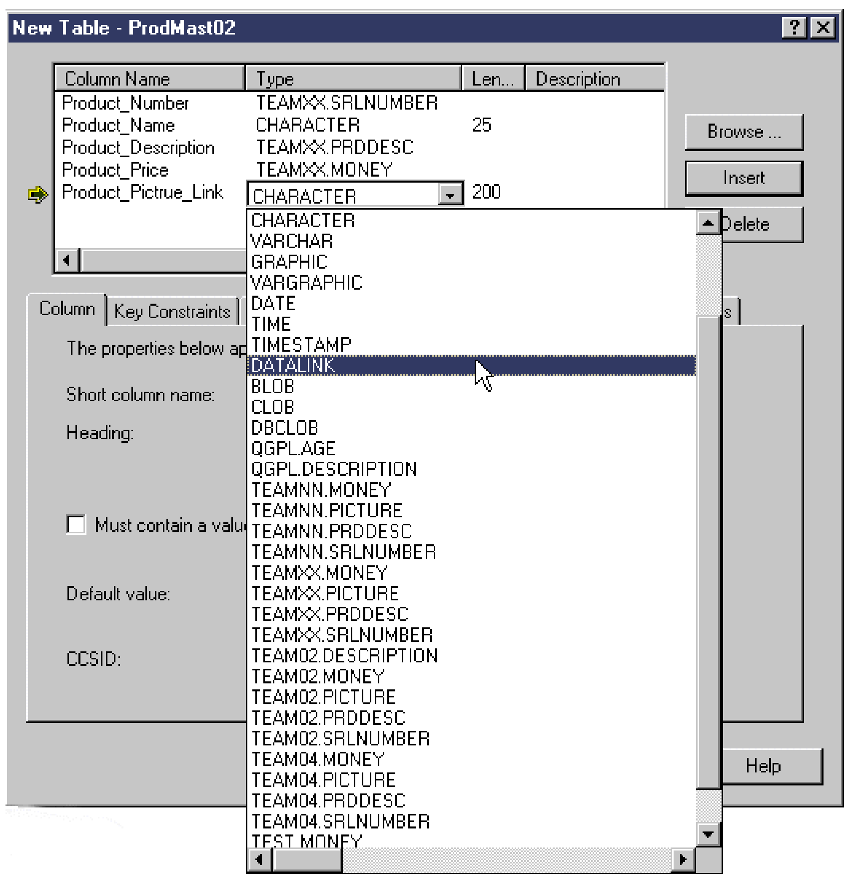


Figure 111. Inserting a DataLink column

If you do not specify a length for the column, a default value of 200 is applied. Make sure that the specified length is sufficient to contain both the largest expected URL and any DataLink comment. The maximum length that can be specified is 32718. Once the DataLink column has been inserted, a number of options must be considered for that column, some of which are specific to the DataLink data type. These options could be grouped under the general description of "Link Control". The Link Control options you select determines if file system objects are linked to this table and how the links will be managed with regards to access control and integrity.

Although a linked file cannot be referenced by more than one table row, a table row may contain more than one DataLink column. However, each of those columns must link to a different file. Figure 112 shows the state of the display after you have selected the DataLink data type and decided on its length. We now look at the remaining options for a DataLink column on the display.

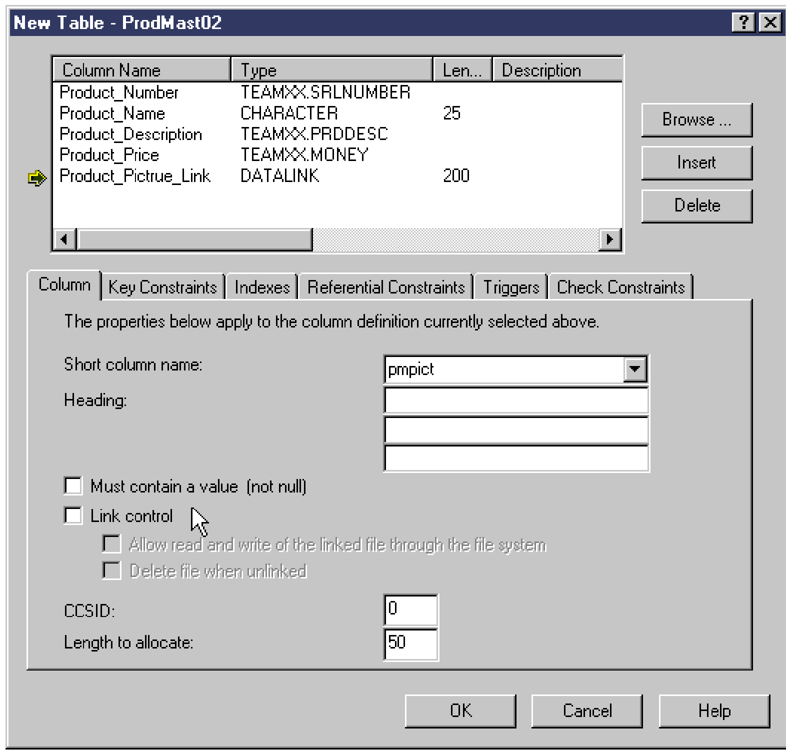


Figure 112. Create table: DataLink column display

A DataLink column can be defined as not null. There are two valid default values: null and DLVALUE('','URL','').

The Length to allocate box has a value of 50. This specifies the default fixed-length space to be allocated for the DataLink column in each row. Column values with lengths less than or equal to the allocated length are stored in the fixed-length portion of the row. Column values with lengths greater than the allocated value are stored in the variable-length portion of the row. Column values stored in the variable-length portion of the row require additional input/output operations to retrieve. The allocated value may range from 0 to the maximum length of the string.

The Link control check box can be left unchecked. This would result in the table being created, but when rows are inserted, there would be no links established to the file objects referenced in the DataLink column. In fact, the DLFM does not become involved, and no check is made to verify that the file server can be accessed or that the file object being referenced even exists. However, the syntax of the URL is validated. This option corresponds to the first row of attributes in Table 13 on page 167. While the 'No Link Control' option still provides value in terms of new application potential, it does not enable you to benefit from the management and integrity control provided by the File Link Control option. However, it allows a linked file to be referenced by a DataLink value in more than one table. If you now check the Link control check box, the display changes look like those shown in Figure 113 on page 170.

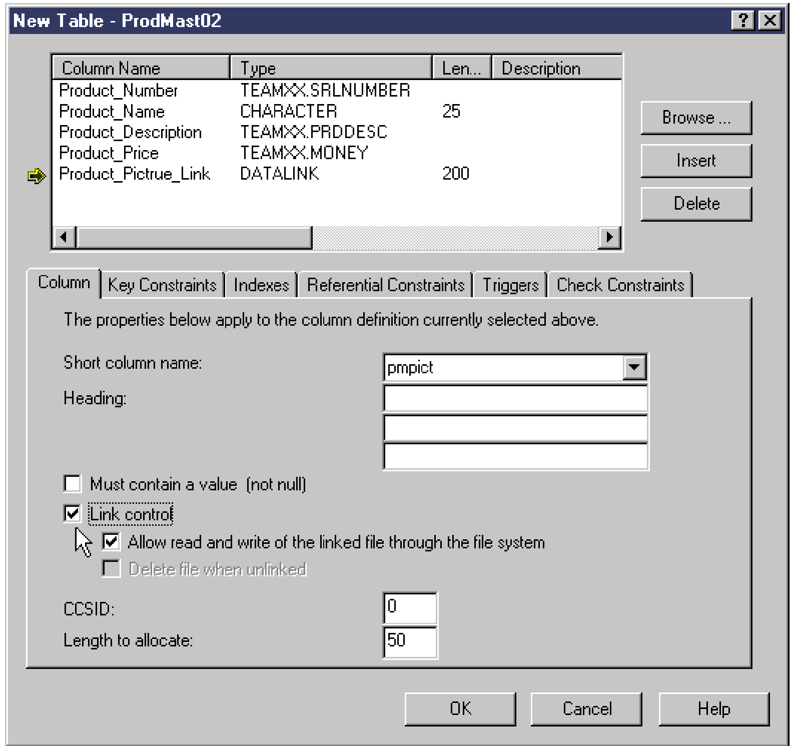


Figure 113. Create table: DataLink column link control Read FS/Write FS

Refer to Figure 119 on page 175 for the equivalent SQL statement.

When you specify Link control, the check box Allow read and write of the linked file through the file system is no longer grayed out and is checked by default. If you now create the table by pressing the OK button, it is created with the DataLink option READ PERMISSIONS FS/WRITE PERMISSIONS FS. This means that the file system controls access to the associated file objects. However, attempts to move, delete, or rename the file while it is linked are always denied because the data integrity is enforced. Attempts by a program to perform read and write operations directly on the file are allowed if all the appropriate authorities are in place. There is no requirement to retrieve an access control token from the database table (see 6.4.6, “Using the DataLink access control token” on page 186, for a detailed explanation). This option corresponds to the second row of attributes in Table 13 on page 167.

However, if you un-check the Allow read and write of the linked file through the file system check box, the display appears as shown in Figure 114.

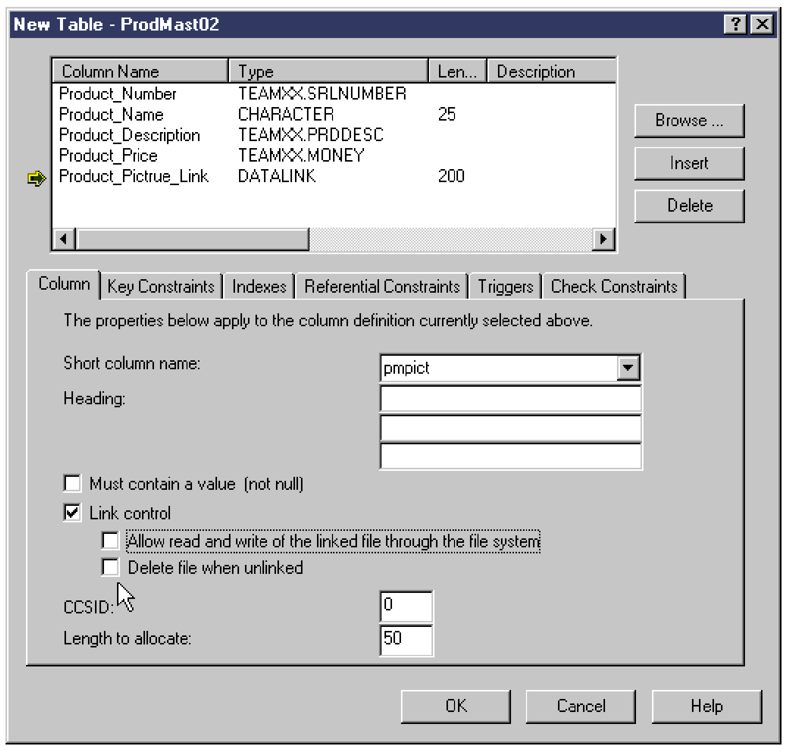


Figure 114. Create table: DataLink column link control Read DB/Write Blocked

The equivalent SQL statement looks like the following example:

```
CREATE TABLE ProdMast02
(Product_Number         FOR COLUMN PMNBR     TEAMXX/SRLNUMBER NOT NULL WITH DEFAULT,
Product_Name            FOR COLUMN PMNAM     CHAR(25) NOT NULL WITH DEFAULT,
Product_Description     FOR COLUMN PMDESC    TEAMXX/PRDDESC,
Product_Price           FOR COLUMN PMPRIC    TEAMXX/MONEY,
Product_Picture_Link    FOR COLUMN PMPICT    DATALINK(200)
LINKTYPE URL
FILE LINK CONTROL
INTEGRITY ALL
READ PERMISSION DB
WRITE PERMISSION BLOCKED
RECOVERY NO
ON UNLINK RESTORE);
```

This indicates implicitly that you wish to create the table with the attributes READ PERMISSION DB/WRITE PERMISSION BLOCKED. This means that DB2 Universal Database for AS/400 controls access to the associated file objects. This is achieved by transferring ownership of each file object to the DLFM (user profile QDLFM) at the time that a table row is inserted and the link is established. However, attempts to move, delete, or rename the file while it is linked are always denied because the data integrity is enforced. Attempts by a program to perform write operations on the file are always rejected regardless of the permissions in place for the file. Attempts to perform read operations directly on the file will be honored if the user has sufficient permissions in place for that file. However, as the intention of this link option is to have the database control read access to the file objects, you should always ensure that the files to be linked have no public access permissions defined. Then, read operations are only successful if the program first obtains an access control token from the database by reading the

associated table row. Refer to 6.4.6, “Using the DataLink access control token” on page 186, for a detailed explanation.

You can display the file permissions by using Operations Navigator to display the IFS files. Select a file, right-click on it, and select **Permissions** from the drop-down box. Figure 115 shows the ownership of a file in the AS/400 IFS before it has been linked. You see that it is owned by user TEAMXX.

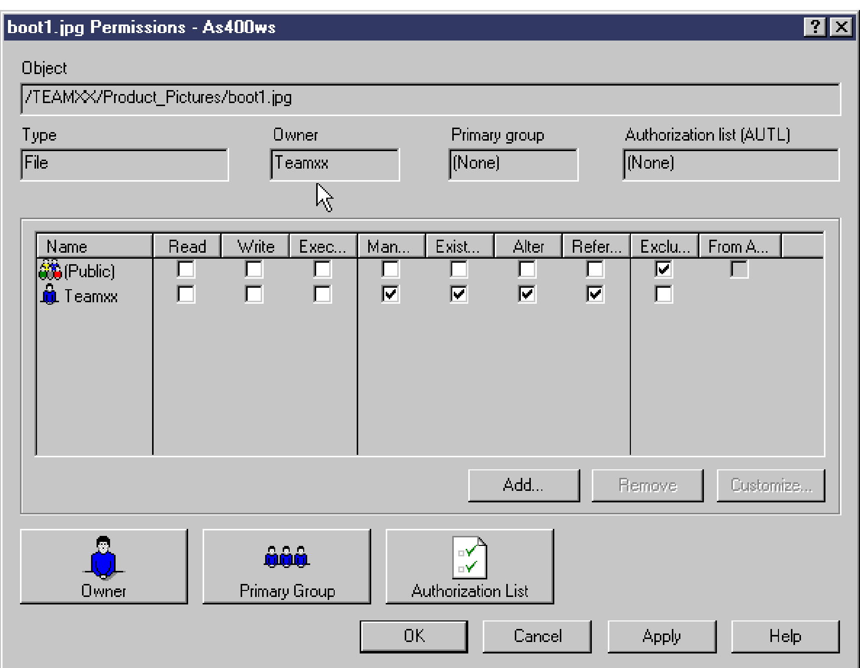


Figure 115. File ownership: Before linking

Figure 116 shows the ownership of the same file after it has been linked to a table row where the table was created with the option Read Permission DB.

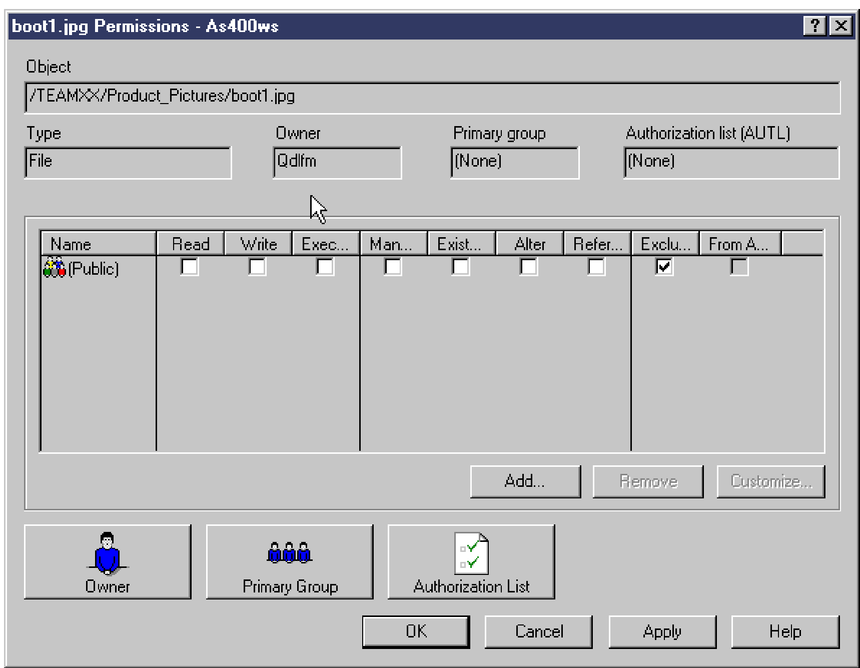


Figure 116. File ownership: After linking

You will see that the owner of the file is now QDLFM, the user profile of the DataLinks File Manager. However, when ownership is changed to QDLFM, details about the previous ownership are saved.

Referring back to Figure 114 on page 171, the final check box option is the one labeled 'Delete file when unlinked'. If you place a check mark in this box, you establish the option ON UNLINK DELETE when the table is created. Note that the check box is grayed out if the file system is controlling authorities for read and write operations. This is because it is not logical for unlink actions caused by database activity to operate on file objects when the file system is managing write access to those objects. If you select On Unlink Delete when a table row is deleted, the associated file is unlinked and then deleted from the file system. The same action occurs when a table row is updated, because an update is executed as a delete of the existing row followed by an insert of the new row. This option corresponds to the sixth row of attributes in Table 13 on page 167.

If you leave the check box empty, you are implicitly indicating that you wish to create the table with the option ON UNLINK RESTORE. When a file is unlinked, this option ensures that the file’s ownership and permissions are restored to their state at the time that the file was linked. If the owner no longer exists in the file system, ownership is given to the default owner, which is QDFTOWN on AS/400. This option corresponds to the fifth row of attributes in Table 13 on page 167.

You must exercise caution if you use the On Unlink Delete option. We *strongly* advise that you use the On Unlink Restore option unless an application can significantly benefit from the delete action. For example, you may have a Web application that allows potential customers to listen to CDs and watch video clips from an online catalog. While the CD or video are popular, you are using the I integrity of DataLinks to prevent the CD sound bites and video clips from being deleted. However, you would want to maintain the catalog so that, when a CD or video is no longer current or popular, it is removed. The On Unlink Delete option

would ease the maintenance of the catalog by automatically deleting the CD sound bites and video clips when the row is deleted from the catalog table in the database. Figure 117 summarizes the V4R4 DB2 Universal Database for AS/400 link control options.

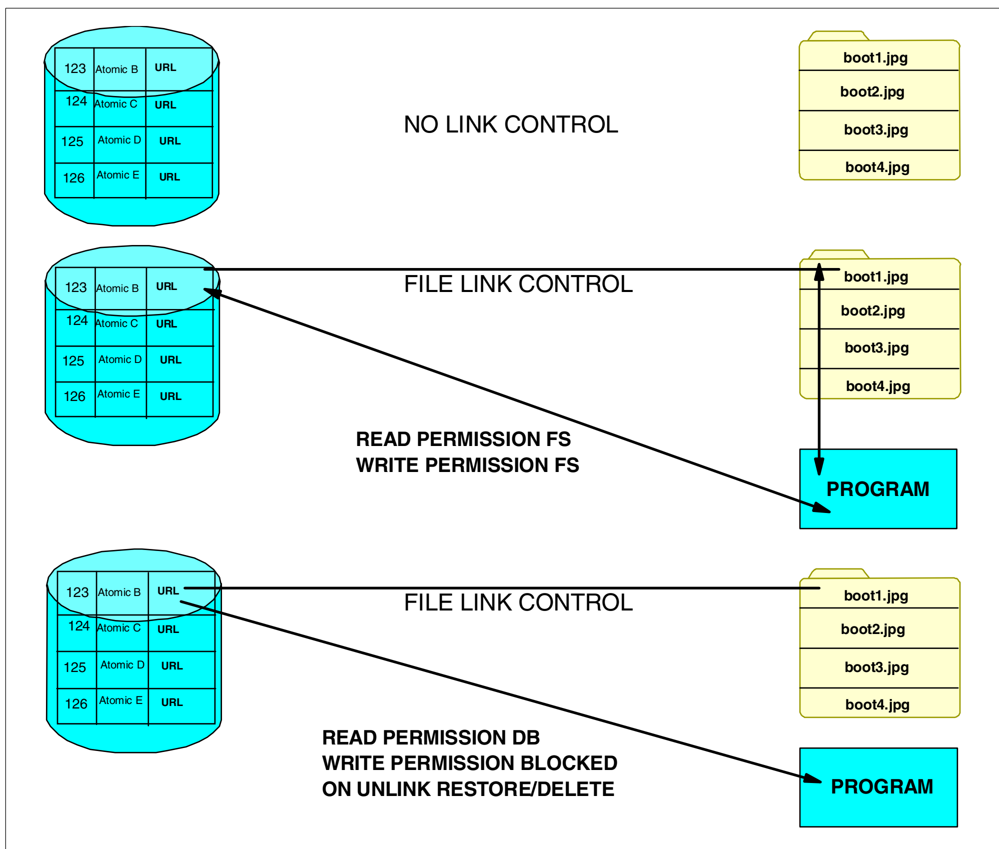


Figure 117. Summary of DB2 Universal Database for AS/400 link control options

When you create a table, there are a number of other table properties that can be defined. These properties include Key Constraints, Indexes, Referential Constraints, and so forth.

DataLink fields have special considerations. They may not be used as key fields in an index. Consequently, they may not be used to define key constraints or referential constraints. Any attempt to define an index, key constraint, or referential constraint will result in an error message.

Triggers may be defined for tables containing DataLinks columns.

Because DataLink columns cannot be compared with other columns or literals, they cannot have check constraints defined on them. An attempt to define a check constraint results in the SQL0401 error message.

Operations Navigator now provides a more complete interface than the 5250 interface for database management activities. However, for those who need to use the 5250 interface, the SQL constructs to enable tables to be created with DataLinks are shown in Figure 118 and Figure 119. For the sake of clarity, these are depicted using the Operations Navigator Run SQL Scripts interface.

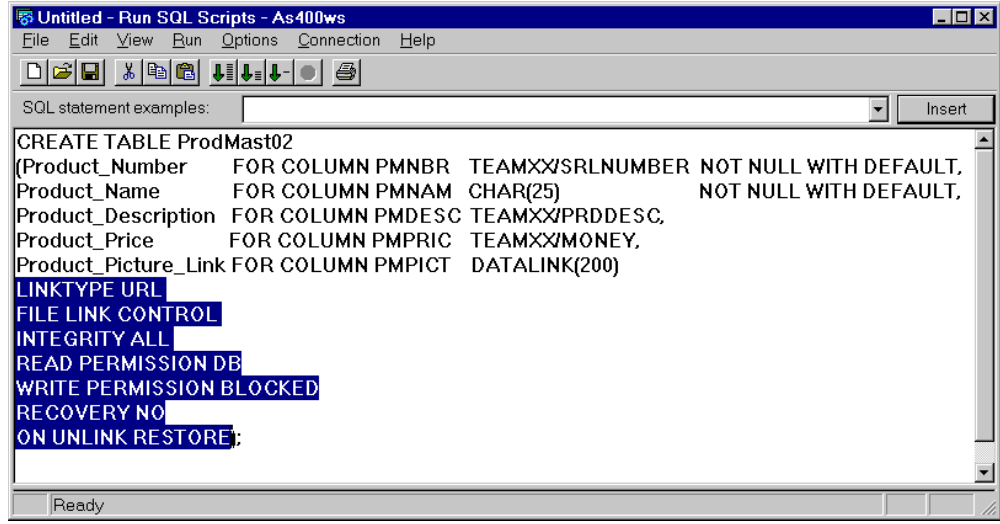


Figure 118. DataLink column with read permission DB/Write permission blocked

Note

- If the On Unlink Delete option is required, simply substitute ON UNLINK RESTORE with ON UNLINK DELETE.
- LINKTYPE URL is currently the only link type supported by the architecture.

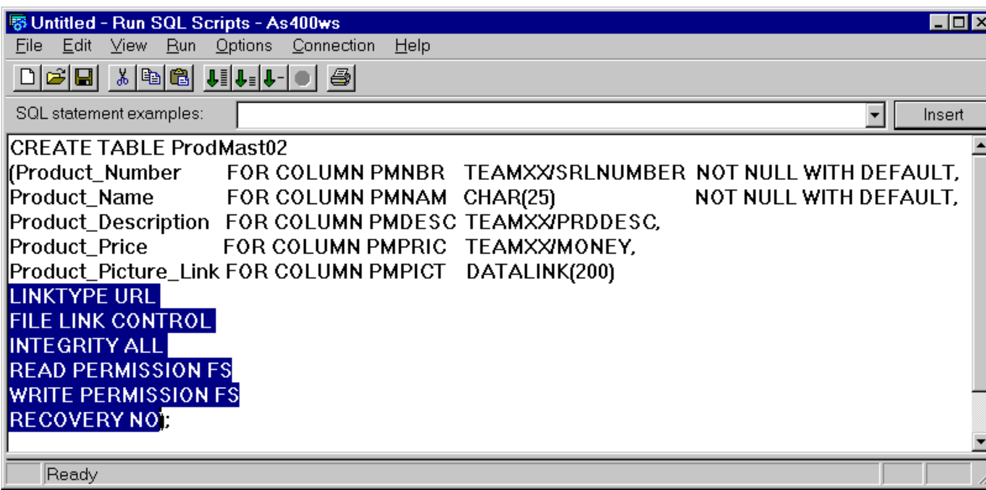


Figure 119. DataLink column with read permission FS/Write permission FS

Figure 120 on page 176 shows an alternative, shorthand definition to that in Figure 119. MODE DB2OPTIONS is used to define a default set of options and is functionally equivalent to:

```
INTEGRITY ALL
READ PERMISSION FS
WRITE PERMISSIN FS
RECOVERY NO
```

This notation is currently the only mode option that has been defined and is provided by DB2 Universal Database for AS/400 for compatibility with the other DataLink capable platforms.

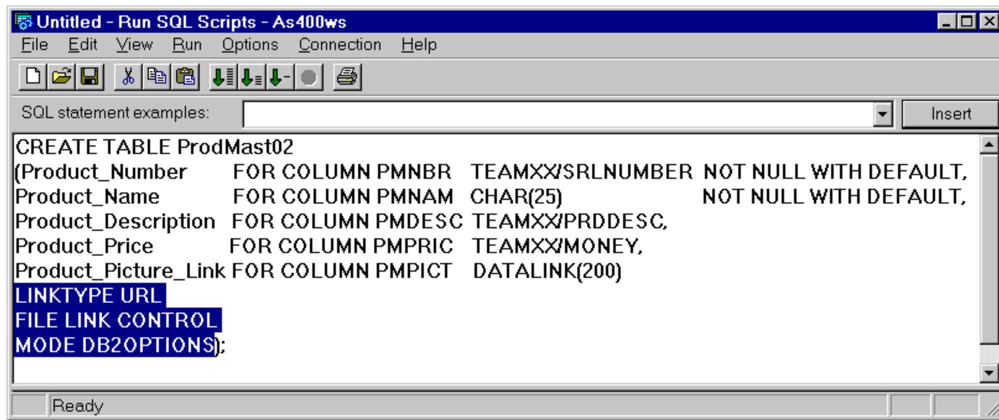


Figure 120. Create table with DataLinks: SQL (Mode DB2Options)

If a table with a datalink column is created with File Link Control, that table must be journaled. This is because the link operation operates on two separate entities, the table and the file, and they are considered to be part of one transaction in order to maximize integrity. Because a table and its linked files could be on different physical servers, two-phase commitment control is exploited in order to extend that transaction integrity.

If you attempt to insert data into a table with a DataLink that has been created with File Link Control, and you have failed to start journaling that table, you receive an SQL7008 error message (for example, `SQL7008 - PICTU00001 in TEAMXX not valid for operation`). It is worth remembering that placing your tables in a collection rather than a library will automate journaling for those tables. However, you must not forget the need to manage the journal receivers to prevent them from growing in an uncontrolled way. The journal entry for a DataLink column is written in the normal way with the row content appearing as characters as shown in Figure 121.

```
                              Display Journal Entry

Object . . . . . . . :   PICTU00001      Library  . . . . . . :   TEAMXX
Member . . . . . . . :   PICTU00001      Sequence . . . . . . :   387
Code . . . . . . . . :   R  - Operation on specific record
Type . . . . . . . . :   PT - Record added
Incomplete data  . . :   No

            Entry specific data
Column      *...+....1....+....2....+....3....+....4....+....5
00001      'A é00001URL 00043        FILE://AS400WS/teamxx/Fun'
00051      '_pictures/fish.gifThis is The Fish                '
00101      '                                                  '
                                                                  More...

           Null value indicators
Field      *...+....1....+....2....+....3....+....4....+....5
00001     >00<
                                                                   Bottom
Press Enter to continue.

F3=Exit   F6=Display only entry specific data
F10=Display only entry details   F12=Cancel   F24=More keys
```

Figure 121. Detailed journal entry: DataLink row insert

V4R4 of Operations Navigator does not provide a function for viewing the link attributes of DataLink column within a table. However, the DSPFFD CL command has been updated to display the information in a 5250 session. The last page of the displayed output will look similar to the example shown in Figure 122 on page 178.

```
                               Display Spooled File
 File  . . . . . :   QPDSPFFD
Page/Line    1/47
 Control . . . . .
Columns      1 - 130
 Find  . . . . . .

*...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
....+....9....+....0....+....1....+....2....+....3
   Alternative name . . . . . . . . . . . . : PRODUCT_PRICE
   User defined-type name . . . . . . . . . : MONEY
   User defined-type library name . . . . . : TEAMXX
    Allows the null value
     PMPICT      DATALINK       200      224        71         Both
PRODUCT_PICTURE_LINK
   Alternative name . . . . . . . . . . . . : PRODUCT_PICTURE_LINK
   Variable length field -- Allocated length :    50
   Datalink link control . . . . . . . . . . : File
   Datalink integrity . . . . . . . . . . . : All
   Datalink read permission . . . . . . . . : Database
   Datalink write permission . . . . . . . . : Blocked
   Datalink recovery . . . . . . . . . . . . : No
   Datalink unlink control . . . . . . . . . : Restore
    Allows the null value
   Default value . . . . . . . . . . . . . . :
      *NULL
      Coded Character Set Identifier  . . . . . :      37
                                                                          Bottom
 F3=Exit    F12=Cancel    F19=Left    F20=Right    F24=More keys
```

Figure 122. DSPFFD output for a table with a DataLink column

6.4.3 Data manipulation examples

Once you have created a table with one or more DataLink columns, you can use an SQL scalar function to insert or update data.

Figure 123 shows an example of a table insert. DLVALUE is the scalar function. It is overloaded to accept one, two, or three parameters.

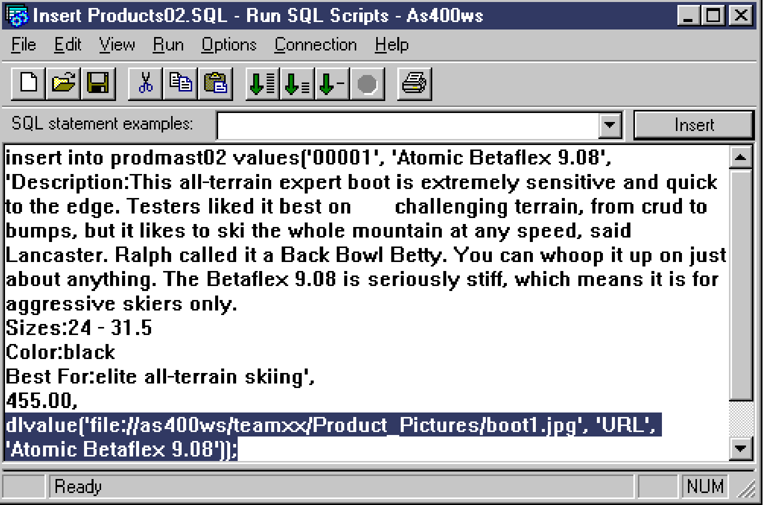


Figure 123. Insert with DLVALUE DataLink scalar function

The example shows all three parameters being passed to the function. These parameters are:

- The data location, for example:

  `'file://as400ws/teamxx/Product_Pictures/boot1.jpg'`

- The link type, for example: `'URL'` (currently only 'URL' is supported). This argument is optional. If not specified, it will be set to 'URL'.

- A comment, for example:

  `'Atomic Betaflex 9.08'.`

  The comment is optional. If it is not specified, it is set to an empty string.

The full file path comprises the file server name, the registered prefix, the sub-directory path below the prefix, and the file name. The file must exist at the time the insert operation is executed if the table was created with file link control.

Using the DataLink data from the example in Figure 123, other valid parameter combinations provided by function overloading of the DLVALUE scalar function are:

- dlvalue('file://as400ws/teamxx/Product_Pictures/boot1.jpg')
- dlvalue('', 'URL', 'Atomic Betaflex 9.08')

In the case where only the link type (URL) and the comment are provided, the file path is defined as a zero-length string. The resulting row in the table would contain empty link attributes, and no file link would exist. This can be used, for example, to create rows in the table that are place-holders for files that, as yet, do not exist. Figure 124 shows an example of inserting a row into a table where the DataLink column does not specify a data location parameter. Using the Operations Navigator Quick View function, Figure 125 shows the contents of the table after the successful insert operation. The newly inserted row has an empty DataLink column.

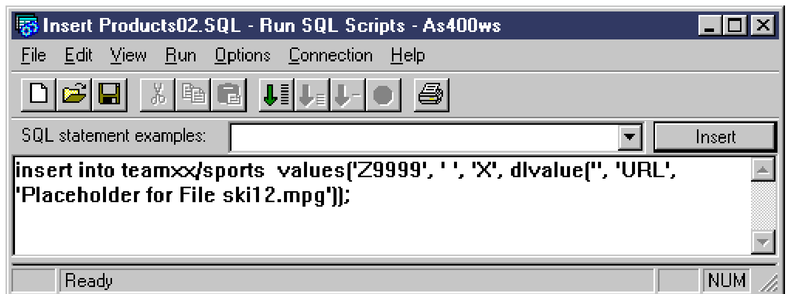


Figure 124. DLVALUE function overloading

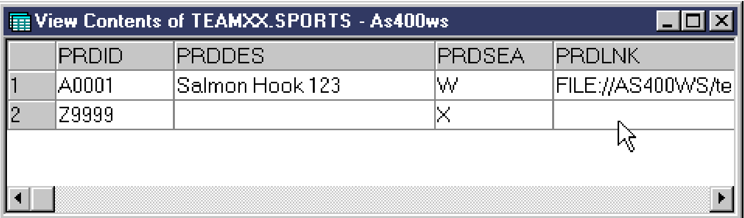


Figure 125. Table with empty DataLink column

However, there is one important consideration when inserting DataLink values into a table. A linked file cannot be referenced by more than one table row, that is, there is a one-to-one relationship between a file and the table to which it is linked. However, another table may include a linked file in its DataLink column value if that table was created with no link control.

The Update statement works in the same way with the DLVALUE scalar function. However, an update to a row with a linked file is treated as a delete of the existing row followed by an insert of a new row. The effect on the linked file is to unlink the existing file and link the new file. In 6.4.1, “DataLink options: General” on page 165, the On Unlink Delete option is discussed in detail. However, to reiterate a word of caution, if On Unlink Delete is the option you choose for a table, the Update operation causes the file referenced by the row being updated to be deleted from the file system. Figure 126 shows an example of an Update operation. In this example file, ski12.mpg is linked to the Sports table immediately after the existing row has been deleted and the new row inserted.

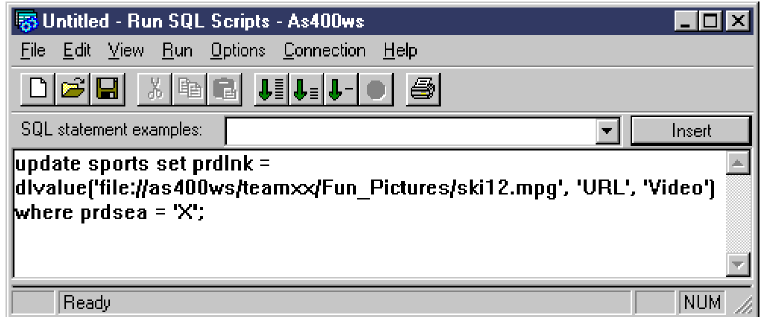


Figure 126. Update with DLVALUE DataLink scalar function

Note

Since the update is really a delete followed by an insert, you need to specify link type and comment values again. If you omit them in the update statement, they will be set to their respective default values.

When you insert a row with a DataLink into a table using the DLVALUE scalar function, the DataLink content that is actually stored is an encapsulated value. It contains a logical reference from the database to a file stored externally to the database within a file system. The encapsulated value is comprised of the following elements:

- Link Type: Currently, only type URL is supported.
- Scheme (Optional): For Link Type URL file:, http:, and https: are supported. Note that the scheme simply aids the DLFM in locating and validating the file object to be linked.There is no requirement for the presence of any particular Web server product.
- File Server Name: The complete server address.
- File Path: The directory and file name hierarchy within the file server

- Access Control Token: Generated dynamically (see 6.4.6, “Using the DataLink access control token” on page 186, for more details).
- Comment (Optional): Up to 254 characters of description.

Once you insert data into a table, you must use an SQL scalar function to retrieve data from the encapsulated DataLink value. The valid scalar functions are covered in 6.4.4, “DataLink SQL scalar functions” on page 182.

Other common types of SQL data manipulation you may use include Group By and Order By. Neither grouping nor ordering is allowed based on a DataLink column. Figure 127 shows the error message you receive if you attempt Order By on a DataLink.

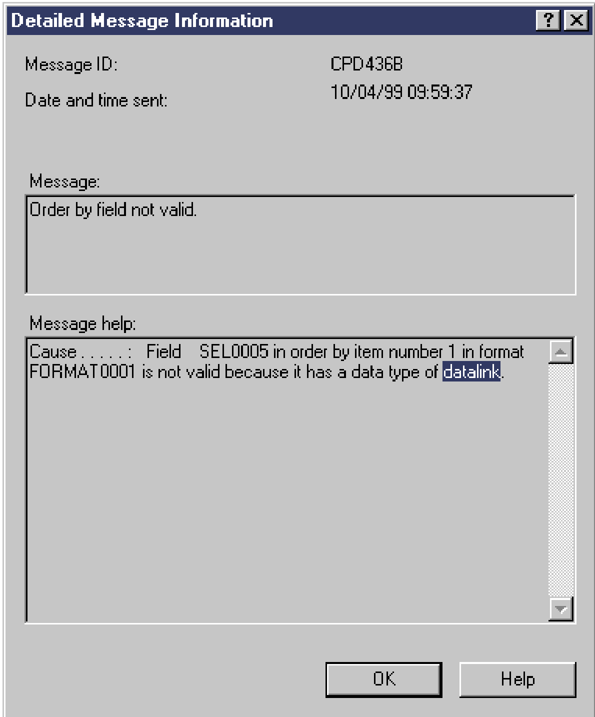


Figure 127. Order by on DataLink column

6.4.4 DataLink SQL scalar functions

In addition to the DLVALUE scalar function, a number of others are provided in order to extract data from the encapsulated DataLink. They are summarized in Table 14.

Table 14. DataLinks SQL scalar functions

Scalar function name	Data type returned	Data returned
DLVALUE	DATALINK	N/A (For Insert and Update)
DLCOMMENT	VARCHAR(254)	Comment
DLLINKTYPE	VARCHAR(4)	Link Type (Only URL currently supported)
DLURLCOMPLETE	VARCHAR	Server Name+ Full Directory Path + Access Control Token
DLURLPATH	VARCHAR	Full Directory Path + Access Control Token
DLURLPATHONLY	VARCHAR	Full Directory Path
DLURLSCHEME	VARCHAR(20)	FILE or HTTP or HTTPS
DLURLSERVER	VARCHAR	Server name

As an example, if you run the SQL statements from the Operations Navigator Run SQL Scripts window, as shown in Figure 128, the output appears similar to the example displayed in Figure 129 through Figure 132.

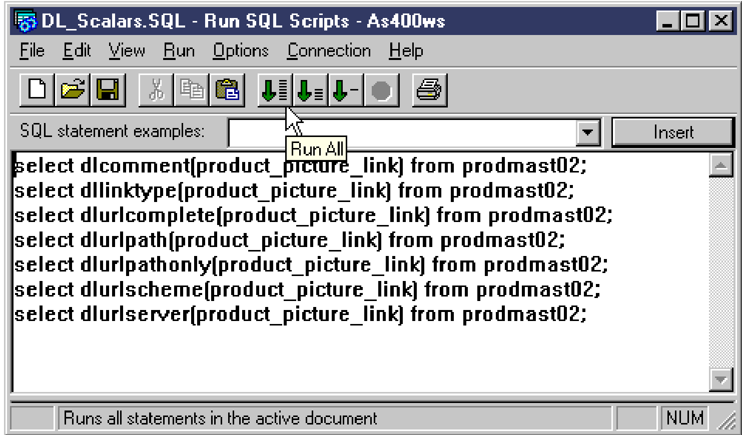


Figure 128. DataLink SQL scalar functions script

Figure 129. Result set from the DLURLCOMPLETE scalar function

In Figure 129, as well as returning the file server name (AS400WS) and the full file directory path (/teamxx/Product_Pictures/boot1.jpg), you can also see the access control token immediately before the file name.

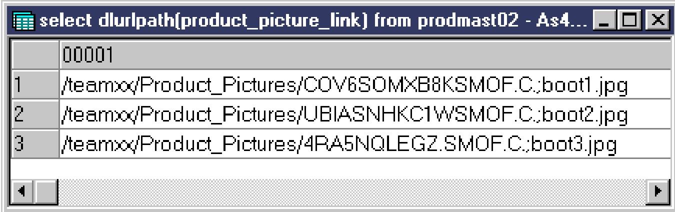


Figure 130. Result set from the DLURLPATH scalar function

Figure 130 shows that DLURLPATH omits the file server name, while Figure 131, DLURLPATHONLY, also omits the access control token.

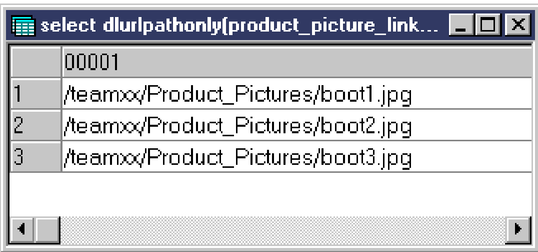


Figure 131. Result set from the DLURLPATHONLY scalar function

Finally, Figure 132 shows the simple scalar functions, DLURLSCHEME and DLURLSERVER, that merely return the URL scheme (file:, http:, or https:) and the file server name, respectively.

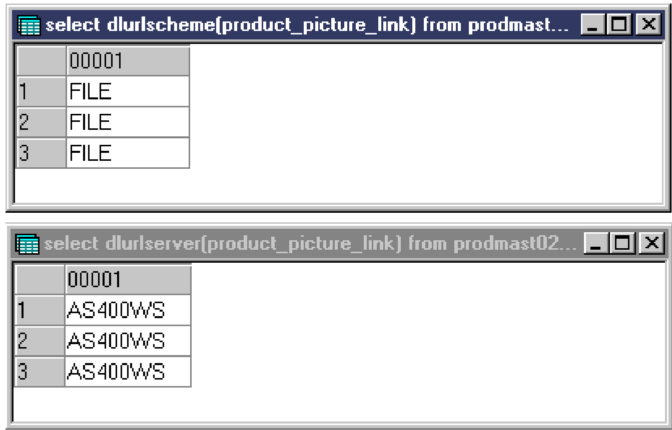


Figure 132. Result sets from DLURLSCHEME and DLURLSERVER scalar functions

6.4.5 Using the DataLink in dynamic Web pages

To achieve better scalability of your Internet software solution, you usually split the application server running the Web server from the database server. For performance reasons, you also want to store the files referred in the Web pages on the Web server. Now, you may ask the question: How does the DataLink

support fit into this picture? In this section, we explain how to take advantage of the DataLinks for building dynamic Web pages.

Let’s suppose that the database containing all the products we want to sell over the Internet reside in the library TEAMXX on the AS400RCH database server, and that the product picture files reside on the AS400LON Web server machine. The product picture files are linked to appropriate rows in the PRODMAST03 table on the AS400RCH machine. The following procedure outlines the major steps required to set up our application environment:

1. The PRODMAST03 table containing detailed product information was created with the following SQL statement:

   ```
   CREATE TABLE ProdMast03
   (Product_Number            FOR COLUMN PMNBR      TEAMXX/SRLNUMBER  NOT NULL WITH DEFAULT,
   Product_Name               FOR COLUMN PMNAM      CHAR(25)          NOT NULL WITH DEFAULT,
   Product_Description        FOR COLUMN PMDESC     TEAMXX/PRDDESC,
   Product_Price              FOR COLUMN PMPRIC     TEAMXX/MONEY,
   Product_Picture_Link       FOR COLUMN PMPICT     DATALINK(200)
   LINKTYPE URL
   FILE LINK CONTROL
   INTEGRITY ALL
   READ PERMISSION FS
   WRITE PERMISSION FS
   RECOVERY NO);
   ```

   Note that we use file system permission for read and write options for the DataLink column.

2. The linked objects, which are product pictures in this case, were copied to the `/teamxx/images/` IFS directory on the Web server system AS400LON. To enhance the Web server security, all files located in this directory have the PUBLIC permission set to *EXCLUDE. At the same time, we added the *RX permission for the QTMHHTTP profile so that the HTTP server jobs running on the AS400LON system can access the image files and serve them to the clients.

3. The DLFM environment was initialized on the AS400LON system with the `INZDLFM` command. The directory prefix was set up with the following CL command:

   ```
   ADDPFXDLFM PREFIX(('/teamxx/'))
   ```

   The host database was set up as follows:

   ```
   ADDHDBDLFM HOSTDBLIB((TEAMXX)) HOSTDB(AS400RCH)
   ```

   The DLFM server was started with the following CL command:

   ```
   STRTCPSVR SERVER(*DLFM)
   ```

4. The product details were inserted into the PRODMAST03 table on the database server AS400RCH. An example of the insert statement is shown as follows:

   ```
   Insert into teamxx/prodmast02 values('00001','Solomon X-scream Series',
   'Description:A solid, reliable performer for experts. A sense of freedom and
   speed when turning. For playing with the terrain and improvising at high
   speed.
   Sizes:179 - 195
   Color:yellow
   Best For:all but the gnarliest terrain',
   730.00,
   dlvalue('file://as400lon/teamxx/images/xscr_pr.gif', 'URL', 'Solomon
   Xscream'));
   ```

Note that the URL value in the DataLink column points to the remote system AS400LON.

Now, our Web Shop application can generate dynamic HTML pages on the fly, fetching the required product data from the AS400RCH system. To retrieve a product picture file name for a given product number, we could use the following SQL statements:

```
CONNECT TO AS400RCH 1
....
SELECT dlurlpath(Product_Picture_Link) INTO :src FROM prodmast03 WHERE
Product_number = '00001' 2
```

SQL statement notes

1. Since the product database resides on the remote system, we can use DRDA to connect to the remote database. Note that, in this scenario, we need an RDB directory entry for the AS400RCH system on the AS400LON machine. Refer to *DB2/400 Advanced Database Functions*, SG24-4249, for details on setting up the DRDA environment.
2. The DLURLPATH scalar function is used to retrieve the full file directory path for a given product into the `src` host variable. We can now use this variable to generate an appropriate IMG HTML tag:

   ```
   <IMG alt="Product Inline Image" src="/teamxx/images/xscr_pr.gif">
   ```

There are several advantages of using DataLinks in this scenario:

- The product picture files on the Web server machine are safe. Nobody, even with QSECOFR authority, can move, rename, or delete linked files.

  Note that unlinked objects in the /teamxx/images/ directory can still be manipulated by a user who has proper authority. However, this is only true when the DLFM server is up and running. When the server is down, no manipulation of the objects in the prefixed directory is allowed, because the file system cannot verify whether any of these objects is linked. For example, deleting an object in the /teamxx/images/ directory could compromise the integrity of the PRODMAST03 file.
- Although the image files logically belong to the PRODMAST03 table, they are physically stored on the machine where they are needed.
- IFS APIs, rather than SQL, are used to serve potentially large objects to the client Web browser.

Figure 133 on page 186 shows our example HTML page using linked image files.

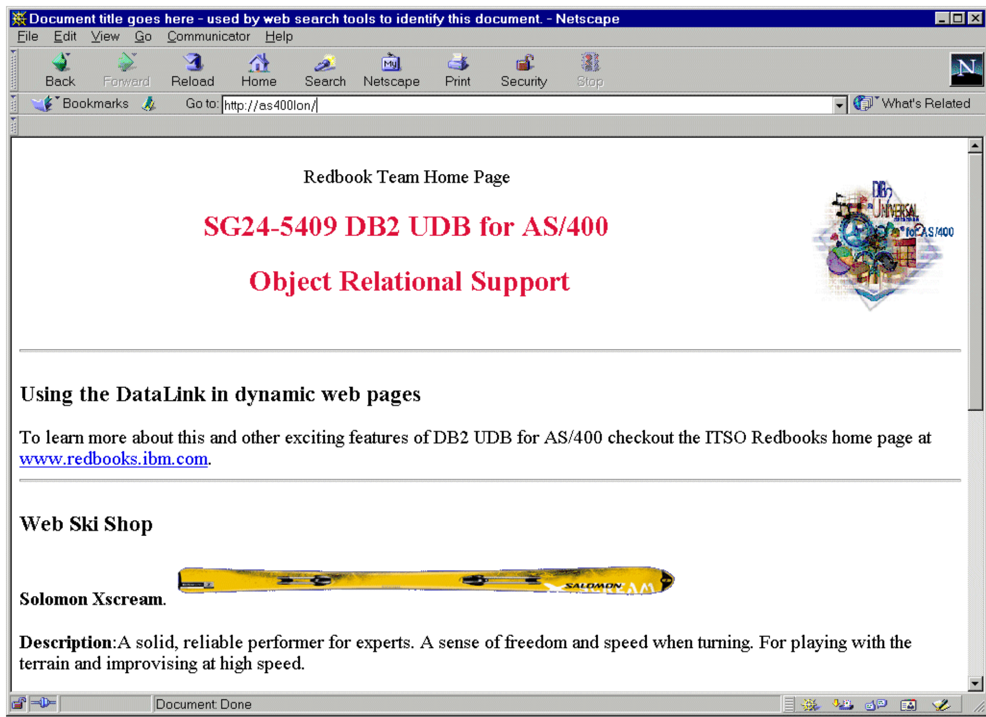


Figure 133. Using linked image files in HTML pages

6.4.6 Using the DataLink access control token

Section 6.4.1, "DataLink options: General" on page 165, provides detailed coverage of the link control options defined in the DataLinks architecture and those implemented in V4R4 of DB2 Universal Database for AS/400. In particular, Figure 114 on page 171 and Figure 118 on page 175 show the option for creating the table with the link option of Read Permission DB/Write Permission Blocked. You select this option when you want your application to control access to associated file objects. More specifically, you want your application to be able to read file system objects but not to write to or update them. However, if you define the PUBLIC file permissions or ownership properties of a file to allow read access, your application will bypass the database access control. Therefore, you should always ensure that the files to be linked have no public access permissions defined. Then, read operations will only be successful if the program first obtains an access control token from the database by reading the associated table row, and the DataLink Filter validates that token. The DLURLPATH scalar function may be used to retrieve the full directory path along with the control token. This section illustrates how to retrieve and use the token.

We coded programs in C with embedded SQL to test both the Read Permission DB/Write Permission Blocked and the Read Permission FS/Write Permission FS environments. The programs also tested the integrity of both environments by attempting various operations on linked files.

6.4.6.1 READPM02 program

The details are for program READPM02, Test Read Permission DB/Write Permission Blocked. The table is PRODMAST02 and the file boot1.jpg. The file was originally defined with no public authority. The DataLink column definition looks like the following example:

```
Product_Picture_LinkFOR COLUMN PMPICTDATALINK(200)
LINKTYPE URL
FILE LINK CONTROL
INTEGRITY ALL
READ PERMISSION DB
WRITE PERMISSION BLOCKED
RECOVERY NO
ON UNLINK RESTORE
```

The program steps are:

1. Read file directly in the IFS.
2. Move the file to another directory.
3. Update the file (coded in the program as an append).
4. Delete the file.
5. Read the file after reading the table row to which it is linked and executing several scalar functions.

Figure 134 shows the output from the program after running the first four steps. Figure 135 on page 188 shows the output from the program after executing step five. The numbered lines are explained in the notes that follow each figure.

```
    DB2 Universal Database for AS/400
   DataLink Test Program: TEAMXX/READPM02
   Read Permission DB/Write Permission Blocked

   Hit Enter to continue...
 >
   Attempting file system operations on: /teamxx/Product_Pictures/boot1.jpg

 1 Read failed: /teamxx/Product_Pictures/boot1.jpg: Permission denied.
 2 Move failed: /teamxx/Product_Pictures/boot1.jpg -> /boot1.jpg: Improper link.
 3 Append failed: /teamxx/Product_Pictures/boot1.jpg: Permission denied.
 4 Delete failed: /teamxx/Product_Pictures/boot1.jpg: Object is a Datalink object.


 Hit Enter to continue...
 ===>

 F3=Exit F4=End of File F6=Print F9=Retrieve F17=Top
 F18=Bottom  F19=Left   F20=Right F21=User Window
```

Figure 134. Executing program READPM02: Direct file operations on boot1.jpg

Notes on READPM02 program

1. The direct read operation failed. It is because of Read Permission DB and no public read access for the boot1.jpg file, and no control token was passed on the open file request.
2. The move operation failed because the file is linked, and data integrity is enforced by the DataLink Filter.
3. The direct write operation failed because the DataLinks were created with Write Permission Blocked.
4. The delete operation failed because the file is linked, and data integrity is enforced by the DataLink Filter.

```
Selecting Product_Picture_Link from teamxx/prodmast02
    where Product_Number = '00001'

 5 dlcomment(Product_Picture_Link) =
    Atomic Betaflex 9.08
 5 dllinktype(Product_Picture_Link) =
    URL
 5 dlurlcomplete(Product_Picture_Link) =
    FILE://AS400WS/teamxx/Product_Pictures/SS3AYIS;JG2A;F.CC.;boot1.jpg
 6 dlurlpath(Product_Picture_Link) =
    /teamxx/Product_Pictures/SS3AYIS;JG2A;F.CC.;boot1.jpg
 5 dlurlpathonly(Product_Picture_Link) =
    /teamxx/Product_Pictures/boot1.jpg

 7 Readok:/teamxx/Product_Pictures/SS3AYIS;JG2A;F.CC.;boot1.jpg10530bytesread

  Press ENTER to end terminal session.

===>

F3=Exit F4=End of File F6=Print F9=Retrieve F17=Top
F18=Bottom  F19=Left   F20=Right F21=User Window
```

Figure 135. Executing program READPM02: Read of boot1.jpg with control token

5. Various SQL scalar functions used to retrieve the DataLink value from the row in the PRODMAST02 table.
6. The SQL scalar function actually used in the program to retrieve the access control token, file directory path, and file name. The access control token value is highlighted.
7. Opening the file for read access succeeded because the access control token was passed to the file open operation.

The following snippets of the program code highlight the most significant parts of the program with regards to working with the DataLinks. The full program listing can be found in Appendix A, “Source code listings” on page 215.

```
1/  Compile:     CRTSQLCI OBJ(TEAMXX/READPM02) SRCFILE(TEAMXX/QCSRC) +
/                 SRCMBR(READPM02) OPTION(*NOGEN)
/               CRTBNDC PGM(TEAMXX/READPM02) SRCFILE(QTEMP/QSQLTEMP) +
/                 SRCMBR(READPM02) SYSIFCOPT(*IFSIO)
```

Note 1: The comment lines show the method of program compilation. Note the SYSIFCOPT parameter, which directs the created object to use the IFS for stream I/O operations.

```
2{
  exec sql include SQLCA;

  /* declare host variables */
  exec sql begin declare section;
    char link_comment[255];
    char link_type[5];
    struct VARCHAR1
    {
      short length;
      char  data[200];
    } link_url_complete;
    struct VARCHAR2
```

```
    {
      short length;
      char  data[200];
    } link_url_path;
    struct VARCHAR3
    {
      short length;
      char  data[200];
    } link_url_path_only;
    char where_value[6];
  exec sql end declare section;

  char file_name[FILENAME_MAX];
  char ren_file_name[FILENAME_MAX];
  char mov_file_name[FILENAME_MAX];

  exec sql whenever sqlerror go to sqlexit;
  exec sql set path teamxx;
```

Note 2: Declaring the host variables.

```
 3/* initialize the datalink file name */
  strcpy(file_name, "/teamxx/Product_Pictures/boot1.jpg");
```

Note 3: Establishing the path and file name of the IFS file to be processed. Refer to the execution time results shown in Figure 136 on page 192.

```
 4 /* read file */
  file_read(file_name);

  /* move file to new location */
  file_move(file_name, mov_file_name);

  /* update file */
  file_append(file_name);

  /* delete file */
  file_delete(file_name);
```

Note 4: Attempting operations directly on the file.

```
 5 exec sql
    select
      dlcomment(Product_Picture_Link),
      dllinktype(Product_Picture_Link),
      dlurlcomplete(Product_Picture_Link),
      dlurlpath(Product_Picture_Link),
      dlurlpathonly(Product_Picture_Link)
    into
      :link_comment,
      :link_type,
      :link_url_complete,
      :link_url_path,
      :link_url_path_only
    from
      teamxx/prodmast02
    where
      Product_Number = srlnumber(:where_value);

  /* null terminate the varchar host variables */
    link_url_path.data[link_url_path.length]           = '\0';
```

Note 5: Retrieve DataLink values into host variables using scalar functions.

```
 6 /* read file using access control token */
  file_read(link_url_path.data);

  return 0;
}
```

Note 6: Attempt to read the IFS file with the access control token. The host variable 'link_url_path' contains the value obtained from the DLURLPATH SQL scalar function. This value includes the access control token and must be passed to the file open operation. If no token is present, or the token is invalid (for example, it is a previously retrieved token that has expired), the file open fails. Refer to Figure 137 on page 193 for the execution time results.

```
7/*-------------------------------------------------------------------
/
/  Description: Read a file. Display number of bytes read.
/
/  Usage:       file_name       name of the file to read
/               returns         -1 on success
/                                0 on failure
/
*/
int file_read(char* file_name)
{
  FILE* read_file;
  char buf[BUF_SIZE+1];
  int read_count;
  long read_total;
  char perror_message[FILENAME_MAX+128];

  sprintf(perror_message, "Read failed: %s", file_name);
  read_total = 0;

  if ((read_file = fopen(file_name,"rb")) == NULL)
  {
    perror(perror_message);
    return 0;
  }
  while ((read_count =
    fread(buf, sizeof(char), BUF_SIZE, read_file)) > 0)
  {
    read_total += read_count;
  }
  if (fclose(read_file) == EOF)
  {
    perror(perror_message);
    return 0;
  }
  printf("Read ok: %s %ld bytes read\n", file_name, read_total);
  return -1;
}
```

Note 7: Set up the direct file read operation.

```
8/*-------------------------------------------------------------------
/
/  Description: Append EOF to a file.
/
/  Usage:       file_name       name of the file to append EOF to
/               returns         -1 on success
/                                0 on failure
/
*/
int file_append(char* file_name)
{
  FILE* append_file;
  char perror_message[FILENAME_MAX+128];

  sprintf(perror_message, "Append failed: %s", file_name);

  /* make sure that the file exists first */
  if ((append_file = fopen(file_name,"rb")) == NULL)
  {
    perror(perror_message);
    return 0;
  }
  fclose(append_file);
```

```
  if ((append_file = fopen(file_name,"ab")) == NULL)
  {
    perror(perror_message);
    return 0;
  }
  fputc(EOF, append_file);
  if (fclose(append_file) == EOF)
  {
    perror(perror_message);
    return 0;
  }
  printf("Append ok: %s\n", file_name);
  return -1;
}
```

Note 8: Set up the direct file update operation.

```
9/*-------------------------------------------------------------------
/
/  Description: Move a file. File only renamed if no or same path
/               supplied in dest_file_name.
/
/  Usage:       src_file_name   old name of the file to move
/               dest_file_name  new name of the file to move
/               returns         -1 on success
/                                0 on failure
/
*/
int file_move(char* src_file_name, char* dest_file_name)
{
  char perror_message[FILENAME_MAX+FILENAME_MAX+128];

  sprintf(
    perror_message, "Move failed: %s -> %s",
    src_file_name, dest_file_name);

  if (rename(src_file_name, dest_file_name))
  {
    perror(perror_message);
    return 0;
  }
  printf("Move ok: %s -> %s\n", src_file_name, dest_file_name);
  return -1;
}
```

Note 9: Set up the direct file move operation.

READPM03 program

For the program READPM03, we used Test Read Permission FS/Write Permission FS. The table is PRODMAST03, and the file is boot4.jpg. The file was defined with *RWX public authority. The DataLink column definition is shown here:

```
Product_Picture_Link FOR COLUMN PMPICTDATALINK(200)
LINKTYPE URL
FILE LINK CONTROL
INTEGRITY ALL
READ PERMISSION FS
WRITE PERMISSION FS
RECOVERY NO
```

The program steps are:

1. Read file directly in the IFS.
2. Move the file to another directory.
3. Update the file (coded in the program as an append).
4. Delete the file.

Figure 136 shows the output from the program after executing the four steps. The numbered lines are explained in the notes that follow.

```
                    DB2 Universal Database for AS/400
  DataLink Test Program: TEAMXX/READPM03
  Read Permission FS/Write Permission FS

  Hit Enter to continue...
>
  Attempting file system operations on: /teamxx/Product_Pictures/boot4.jpg

  1 Read ok: /teamxx/Product_Pictures/boot4.jpg 12094 bytes read
  2 Move failed: /teamxx/Product_Pictures/boot4.jpg -> /boot4.jpg: Improper link.
  3 Append ok: /teamxx/Product_Pictures/boot4.jpg
  4 Delete failed: /teamxx/Product_Pictures/boot4.jpg: Object is a Datalink object.

  Press ENTER to end terminal session.
   ===>

F3=Exit F4=End of File F6=Print F9=Retrieve F17=Top
F18=Bottom  F19=Left   F20=Right F21=User Window
```

Figure 136. Executing program READPM03: Direct file operations on boot4.jpg

Notes on the READPM03 program

1. The direct read operation succeeded because the public file permissions for boot4.jpg are *RWX and the DataLink column attribute READ PERMISSION FS rather than READ PERMISSION DB.
2. The move operation failed because the file is linked, and data integrity is enforced by the DataLink Filter.
3. The direct write operation succeeded because the DataLinks were created with Write Permission FS, and the public file permissions for boot4.jpg are *RWX.
4. The delete operation failed because the file is linked, and data integrity is enforced by the DataLink Filter.

The READPM03 program is identical to READPM02, except the code to access the database table and retrieve DataLinks values have been removed. The full program listing can be found in Appendix A, “Source code listings” on page 215.

In summary, when a file has been linked, any attempt to move, delete, rename, or update that file is denied by the DataLink Filter.

When a DataLink has been created with the option READ PERMISSIONS DB/WRITE PERMISSIONS BLOCKED, you are allowing the database to control access to associated file objects by transferring ownership of the files to the DLFM (user QDLFM). An attempt to read the file after first using an SQL scalar function to retrieve the access control token from the linked database table is permitted by the DataLink Filter.

When a DataLink has been created with the option READ PERMISSIONS FS/WRITE PERMISSIONS FS, the file system controls access to the associated file objects. Attempts by a program to perform read and write operations directly

on the file are allowed if all the appropriate authorities are in place. There is no need to retrieve an access control token from the database table.

The value of the access control token is not stored permanently in the DataLink value within the table row. It is generated dynamically when a scalar function is executed to retrieve it. Once retrieved, it eventually expires to prevent a user from storing it permanently for later use. Currently, the expiration time is set to two weeks. Figure 137 shows an example of retrieving the token for each of three table rows with the SQL scalar function DLURLPATH and then retrieving it again a few seconds later.

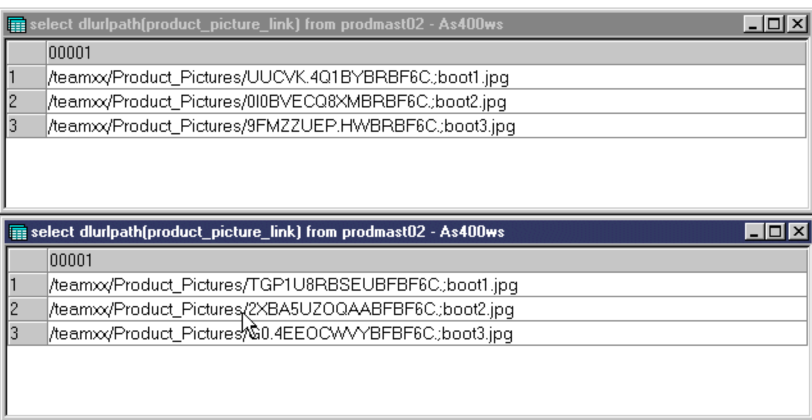


Figure 137. Access control token: Dynamic generation

Notice that the value of the tokens has changed in the short time it took to re-read the table rows.

6.5 Native interface considerations

As previously stated, because the DataLink is not compatible with any host variable data type, the only interface that allows access is SQL. However, we conducted a number of tests to identify what native activities, if any, would be allowed with tables containing DataLink columns. Our conclusion is that, while DataLink columns cannot be used in applications using native I/O access, the tables in which they reside can be used by defining a logical file over the underlaying table that omits the DataLink columns.

The tests were based on a table, SPORTS, which contains three columns of character data type and a single DataLink column. Figure 138 on page 194 shows the table properties. Both the table name and the column names have been kept short to simplify the native file and RPG program coding. The table was created with File Link Control and Read Permissions FS/Write Permissions FS, although these characteristics should have no bearing on the native interface capabilities.

An RPG program, SPORTRPG1, was written to read a row from the SPORTS table, check the product season code column (PRDSEA) for the character 'W' and, if it is equal to 'W', add a row to the output table WINTER containing all the input columns, including a DataLink column. WINTER was created with the option No Link Control.

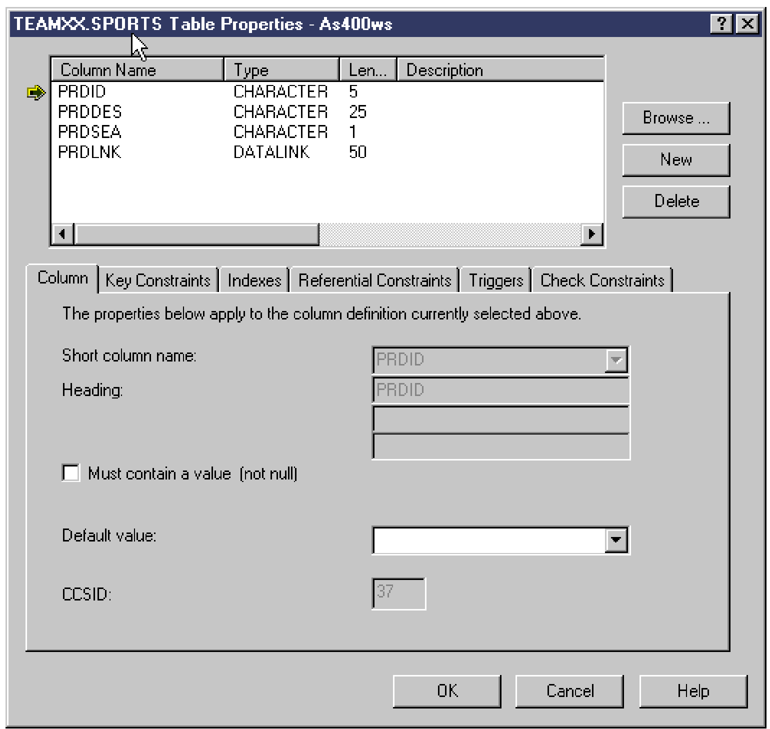


Figure 138. Table SPORTS for native tests

The program was compiled, and the compilation listing is shown in here:

```
5769RG1 V4R4M0  990521              IBM RPG/400                     TEAMXX/SPORTRPG1     10/06/99
15:56:28      Page       1
 Compiler . . . . . . . . . . . . . . :   IBM RPG/400
 Command Options:
   Program  . . . . . . . . . . . . . :   TEAMXX/SPORTRPG1
   Source file  . . . . . . . . . . . :   TEAMXX/QRPGSRC
   Source member  . . . . . . . . . . :   SPORTRPG1
   Source listing options . . . . . . :   *SOURCE     *XREF       *GEN        *NODUMP     *NOSECLVL
*NOSRCDBG   *NOLSTDBG
   Generation options . . . . . . . . :   *NOLIST     *NOXREF     *NOATR      *NODUMP     *NOOPTIMIZE
   Source listing indentation . . . . :   *NONE
   Type conversion options  . . . . . :   *NONE
   Sort sequence  . . . . . . . . . . :   *HEX
   Language identifier  . . . . . . . :   *JOBRUN
   SAA flagging . . . . . . . . . . . :   *NOFLAG
   Generation severity level  . . . . :   9
   Print file . . . . . . . . . . . . :   *LIBL/QSYSPRT
   Replace program  . . . . . . . . . :   *NO
   Target release . . . . . . . . . . :   *CURRENT
   User profile . . . . . . . . . . . :   *USER
   Authority  . . . . . . . . . . . . :   *LIBCRTAUT
   Text . . . . . . . . . . . . . . . :   *SRCMBRTXT
   Phase trace  . . . . . . . . . . . :   *NO
   Intermediate text dump . . . . . . :   *NONE
   Snap dump  . . . . . . . . . . . . :   *NONE
   Codelist . . . . . . . . . . . . . :   *NONE
   Ignore decimal data error  . . . . :   *NO
   Allow null values  . . . . . . . . :   *NO
 Actual Program Source:
   Member . . . . . . . . . . . . . . :   SPORTRPG1
   File . . . . . . . . . . . . . . . :   QRPGSRC
   Library  . . . . . . . . . . . . . :   TEAMXX
   Last Change  . . . . . . . . . . . :   10/06/99  15:56:26
   Description  . . . . . . . . . . . :   RPG Program to Read PF with DataLink
 5769RG1 V4R4M0  990521              IBM RPG/400                     TEAMXX/SPORTRPG1     10/06/99
15:56:28      Page       2
 SEQUENCE                                                                         IND    DO    LAST
PAGE   PROGRAM
```

```
 NUMBER   *...1....+....2....+....3....+....4....+....5....+....6....+....7...* USE   NUM  UPDATE
LINE   ID
                    S o u r c e   L i s t i n g
         H                                                                                *****
      100  FSPORTS  IF  E                   DISK                                       10/06/99
      200  F            SPORTS                             KRENAMESPREC                10/06/99
            RECORD FORMAT(S):  LIBRARY TEAMXX FILE SPORTS.
                     EXTERNAL FORMAT SPORTS RPG NAME SPREC
      300  FWINTER  O   E                   DISK                                       10/06/99
      400  F            WINTER                             KRENAMEWINREC               10/06/99
            RECORD FORMAT(S):  LIBRARY TEAMXX FILE WINTER.
                     EXTERNAL FORMAT WINTER RPG NAME WINREC
  A000000   INPUT  FIELDS FOR RECORD SPREC FILE SPORTS FORMAT SPORTS.
  A000001                                       1   5 PRDID
  A000002                                       6  30 PRDDES
  A000003                                      31  31 PRDSEA
      500  C          NEXTR     TAG                                                    10/06/99
      600  C                    READ SPREC                    60                 3     10/06/99
      700  C          *IN60     DOWEQ*OFF                                         B001  10/06/99
      800  C          PRDSEA    IFEQ 'W'                                          B002  10/06/99
      900  C                    WRITEWINREC                                        002  10/06/99
     1000  C                    ENDIF                                             E002  10/06/99
     1100  C                    READ SPREC                    60                 3  001  10/06/99
     1200  C                    ENDDO                                             E001  10/06/99
     1300  C                    SETON                       LR               1          10/06/99
  B000000   OUTPUT FIELDS FOR RECORD WINREC FILE WINTER FORMAT WINTER.
  B000001                            PRDID      5  CHAR    5
  B000002                            PRDDES    30  CHAR   25
* 6074             FIELD PRDLNK HAS A DATA TYPE OR ATTRIBUTE THAT IS NOT SUPPORTED.
           * * * * *   E N D   O F   S O U R C E   * * * * *
         A d d i t i o n a l   D i a g n o s t i c   M e s s a g e s
* 7086       100   RPG PROVIDES BLOCK OR UNBLOCK SUPPORT FOR FILE SPORTS.
* 7150             RECORD SPORTS IN FILE SPORTS CONTAINS NULL-CAPABLE FIELDS.
* 7154             IGNORED VARIABLE-LENGTH FIELDS IN RECORD SPORTS OF FILE SPORTS.
* 7150             RECORD WINTER IN FILE WINTER CONTAINS NULL-CAPABLE FIELDS.
* 7154             IGNORED VARIABLE-LENGTH FIELDS IN RECORD WINTER OF FILE WINTER.
 5769RG1 V4R4M0  990521               IBM RPG/400                     TEAMXX/SPORTRPG1     10/06/99
15:56:28      Page       3
                    C r o s s   R e f e r e n c e
 File and Record References:
       FILE/RCD    DEV/RCD     REFERENCES (D=DEFINED)
    01  SPORTS      DISK           100D
         SPREC      SPORTS         100D A000000      600     1100
    02  WINTER      DISK           300D
         WINREC     WINTER         300D     900  B000000
 Field References:
       FIELD       ATTR    REFERENCES (M=MODIFIED D=DEFINED)
       *IN60       A(1)        700
* 7031  NEXTR       TAG         500D
       PRDDES      A(25)   A000002D B000002D
       PRDID       A(5)    A000001D B000001D
       PRDSEA      A(1)    A000003D     800
       *OFF        LITERAL     700
       'W'         LITERAL     800
 Indicator References:
       INDICATOR  REFERENCES (M=MODIFIED D=DEFINED)
       *IN           700
       LR           1300M
       60            600M     700     1100M
      * * * * *   E N D   O F   C R O S S   R E F E R E N C E   * * * * *
 5769RG1 V4R4M0  990521               IBM RPG/400                     TEAMXX/SPORTRPG1     10/06/99
15:56:28      Page       4
                    M e s s a g e   S u m m a r y
* QRG6074 Severity:  40   Number:    1
         Message . . . . :   Field data type from an externally-described
           file is not supported. The file is ignored.
* QRG7031 Severity:  00   Number:    1
         Message . . . . :   The Name or indicator is not referenced.
* QRG7086 Severity:  00   Number:    1
         Message . . . . :   RPG handles blocking function for file. INFDS
           updated only when blocks of data transferred.
* QRG7150 Severity:  00   Number:    2
         Message . . . . :   The record format contains null-capable fields.
* QRG7154 Severity:  00   Number:    2
         Message . . . . :   The record format contains variable length
           fields. Variable length fields ignored.
      * * * * *   E N D   O F   M E S S A G E   S U M M A R Y   * * * * *
```

```
 5769RG1 V4R4M0  990521              IBM RPG/400                       TEAMXX/SPORTRPG1     10/06/99
15:56:28      Page       5
                           F i n a l   S u m m a r y
 Message Count:  (by Severity Number)
           TOTAL    00     10     20     30     40     50
             7      6      0      0      0      1      0
 Program Source Totals:
    Records . . . . . . . . . . :   13
    Specifications  . . . . . . :   13
    Table Records . . . . . . . :   0
    Comments  . . . . . . . . . :   0
 Compile stopped. Severity level 40 errors found in file.
          * * * * *  E N D  O F  C O M P I L A T I O N  * * * * *
```

You can see that the compilation failed because of the presence of a DataLink field in the output file. However, the DataLink field in the input file was ignored.

Using another RPG program, SPORTRPG2, we tried to perform exactly the same processing as SPORTRPG1, except the program adds a row to a different table, WINTER2, which only has the Product Code (PRDID) and Product Description (PRDDES) columns defined. It does not have a DataLink column.

The program was compiled, and the listing is shown in here:

```
5769RG1 V4R4M0  990521              IBM RPG/400                       TEAMXX/SPORTRPG2     10/06/99
16:17:43      Page       1
 Compiler . . . . . . . . . . . . . :   IBM RPG/400
 Command Options:
   Program  . . . . . . . . . . . . :   TEAMXX/SPORTRPG2
   Source file  . . . . . . . . . . :   TEAMXX/QRPGSRC
   Source member  . . . . . . . . . :   SPORTRPG2
   Source listing options . . . . . :   *SOURCE     *XREF       *GEN        *NODUMP     *NOSECLVL
*NOSRCDBG   *NOLSTDBG
   Generation options . . . . . . . :   *NOLIST     *NOXREF     *NOATR      *NODUMP     *NOOPTIMIZE
   Source listing indentation . . . :   *NONE
   Type conversion options  . . . . :   *NONE
   Sort sequence  . . . . . . . . . :   *HEX
   Language identifier  . . . . . . :   *JOBRUN
   SAA flagging . . . . . . . . . . :   *NOFLAG
   Generation severity level  . . . :   9
   Print file . . . . . . . . . . . :   *LIBL/QSYSPRT
   Replace program  . . . . . . . . :   *NO
   Target release . . . . . . . . . :   *CURRENT
   User profile . . . . . . . . . . :   *USER
   Authority  . . . . . . . . . . . :   *LIBCRTAUT
   Text . . . . . . . . . . . . . . :   *SRCMBRTXT
   Phase trace  . . . . . . . . . . :   *NO
   Intermediate text dump . . . . . :   *NONE
   Snap dump  . . . . . . . . . . . :   *NONE
   Codelist . . . . . . . . . . . . :   *NONE
   Ignore decimal data error  . . . :   *NO
   Allow null values  . . . . . . . :   *NO
 Actual Program Source:
   Member . . . . . . . . . . . . . :   SPORTRPG2
   File . . . . . . . . . . . . . . :   QRPGSRC
   Library  . . . . . . . . . . . . :   TEAMXX
   Last Change  . . . . . . . . . . :   10/06/99  16:17:41
   Description  . . . . . . . . . . :   RPG Program to Read PF with DataLink
 5769RG1 V4R4M0  990521              IBM RPG/400                       TEAMXX/SPORTRPG2     10/06/99
16:17:43      Page       2
 SEQUENCE                                                                            IND    DO    LAST
PAGE   PROGRAM
 NUMBER   *...1....+....2....+....3....+....4....+....5....+....6....+....7...* USE   NUM  UPDATE
LINE   ID
                          S o u r c e   L i s t i n g
          H                                                                                      *****
      100  FSPORTS  IF  E                    DISK                                           10/06/99
      200  F            SPORTS                            KRENAMESPREC                      10/06/99
            RECORD FORMAT(S):  LIBRARY TEAMXX FILE SPORTS.
                     EXTERNAL FORMAT SPORTS RPG NAME SPREC
      300  FWINTER2 O   E                    DISK                                           10/06/99
      400  F            WINTER2                           KRENAMEWINREC                     10/06/99
            RECORD FORMAT(S):  LIBRARY TEAMXX FILE WINTER2.
                     EXTERNAL FORMAT WINTER2 RPG NAME WINREC
  A000000   INPUT  FIELDS FOR RECORD SPREC FILE SPORTS FORMAT SPORTS.
```

```
 A000001                                       1   5 PRDID
 A000002                                       6  30 PRDDES
 A000003                                      31  31 PRDSEA
     500  C          NEXTR     TAG                                                  10/06/99
     600  C                    READ SPREC                  60                3      10/06/99
     700  C          *IN60     DOWEQ*OFF                                        B001   10/06/99
     800  C          PRDSEA    IFEQ 'W'                                         B002   10/06/99
     900  C                    WRITEWINREC                                       002   10/06/99
    1000  C                    ENDIF                                            E002   10/06/99
    1100  C                    READ SPREC                  60                3   001   10/06/99
    1200  C                    ENDDO                                            E001   10/06/99
    1300  C                    SETON                    LR              1              10/06/99
 B000000   OUTPUT FIELDS FOR RECORD WINREC FILE WINTER2 FORMAT WINTER2.
 B000001                         PRDID      5  CHAR    5
 B000002                         PRDDES    30  CHAR   25
         * * * * *   E N D   O F   S O U R C E   * * * * *
        A d d i t i o n a l   D i a g n o s t i c   M e s s a g e s
* 7086       100   RPG PROVIDES BLOCK OR UNBLOCK SUPPORT FOR FILE SPORTS.
* 7086       300   RPG PROVIDES BLOCK OR UNBLOCK SUPPORT FOR FILE WINTER2.
* 7150             RECORD SPORTS IN FILE SPORTS CONTAINS NULL-CAPABLE FIELDS.
* 7154             IGNORED VARIABLE-LENGTH FIELDS IN RECORD SPORTS OF FILE SPORTS.
* 7150             RECORD WINTER2 IN FILE WINTER2 CONTAINS NULL-CAPABLE FIELDS.
 5769RG1 V4R4M0  990521              IBM RPG/400                  TEAMXX/SPORTRPG2     10/06/99
16:17:43      Page       3
                         C r o s s   R e f e r e n c e
 File and Record References:
       FILE/RCD    DEV/RCD     REFERENCES (D=DEFINED)
    01  SPORTS      DISK           100D
          SPREC     SPORTS         100D A000000      600     1100
    02  WINTER2     DISK           300D
          WINREC    WINTER2        300D     900  B000000
 Field References:
       FIELD       ATTR    REFERENCES (M=MODIFIED D=DEFINED)
       *IN60       A(1)        700
* 7031  NEXTR       TAG         500D
       PRDDES      A(25)   A000002D B000002D
       PRDID       A(5)    A000001D B000001D
       PRDSEA      A(1)    A000003D     800
       *OFF        LITERAL     700
       'W'         LITERAL     800
 Indicator References:
       INDICATOR  REFERENCES (M=MODIFIED D=DEFINED)
       *IN           700
       LR           1300M
       60            600M     700     1100M
      * * * * *   E N D   O F   C R O S S   R E F E R E N C E   * * * * *
 5769RG1 V4R4M0  990521              IBM RPG/400                  TEAMXX/SPORTRPG2     10/06/99
16:17:43      Page       4
                         M e s s a g e   S u m m a r y
* QRG7031 Severity:  00   Number:    1
          Message . . . . :   The Name or indicator is not referenced.
* QRG7086 Severity:  00   Number:    2
          Message . . . . :   RPG handles blocking function for file. INFDS
            updated only when blocks of data transferred.
* QRG7150 Severity:  00   Number:    2
          Message . . . . :   The record format contains null-capable fields.
* QRG7154 Severity:  00   Number:    1
          Message . . . . :   The record format contains variable length
            fields. Variable length fields ignored.
      * * * * *   E N D   O F   M E S S A G E   S U M M A R Y   * * * * *
 5769RG1 V4R4M0  990521              IBM RPG/400                  TEAMXX/SPORTRPG2     10/06/99
16:17:43      Page       5
                           F i n a l   S u m m a r y
 Message Count:  (by Severity Number)
           TOTAL    00     10     20     30     40     50
             6       6      0      0      0      0      0
 Program Source Totals:
    Records . . . . . . . . . . :   13
    Specifications  . . . . . . :   13
    Table Records . . . . . . . :   0
    Comments  . . . . . . . . . :   0
 PRM has been called.
 Program SPORTRPG2 is placed in library TEAMXX. 00 highest severity. Created on 10/06/99 at 16:17:45.
         * * * * *   E N D   O F   C O M P I L A T I O N   * * * * *
```

The DataLink field on the input file was ignored, and the compilation succeeded.

The next step was to execute the program, resulting in the error message shown in Figure 139 and Figure 140.

```
                         Additional Message Information

Message ID . . . . . . :   CPF428A       Severity . . . . . . . :   40
Message type . . . . . :   Escape
Date sent  . . . . . . :   10/06/99      Time sent  . . . . . . :   16:20:02

Message . . . . :   Open of member SPORTS file SPORTS in TEAMXX failed.
Cause . . . . . :   Member SPORTS file SPORTS in library TEAMXX was not opened
  because of error code 2. The error codes and their meanings are:
     1 -- The format for file SPORTS contains one or more large object fields
  and the open request did not indicate that large object fields could be
  processed by the user of the open.
     2 -- The format for file SPORTS contains one or more data link fields and
  the open request did not indicate that data link fields could be processed
  by the user of the open.
     3 -- The format for file SPORTS contains one or more user defined data
  type fields and the open request did not indicate that user defined data
  type fields could be processed by the user of the open.
                                                                       More...
Press Enter to continue.

F3=Exit   F6=Print   F9=Display message details   F12=Cancel
F21=Select assistance level
```

Figure 139. Table with DataLink input to RPG program: Error

```
                         Additional Message Information

Message ID . . . . . . :   CPF428A       Severity . . . . . . . :   40
Message type . . . . . :   Escape

     4 -- A user-defined type for a field for the file does not exist.
Recovery  . . . :   Either specify a different file, use the DSPFFD command to
  determine what user-defined type is missing, change the open request to
  indicate that the specified field type can be processed, or change the
  program to use embedded SQL to process the file. Then try your request
  again. These field types are fully supported only through SQL. Therefore, if
  you do not have the DB2 Query Manager and SQL Development Tool Kit for
  AS/400 product, your program may not be able to access file SPORTS.






                                                                        Bottom
Press Enter to continue.

F3=Exit   F6=Print   F9=Display message details   F12=Cancel
F21=Select assistance level
```

Figure 140. Table with DataLink input to RPG program: Recovery

Even though the program compiled successfully as a result of ignoring the DataLink column, as soon as an attempt was made to open the file that contained the DataLink, the program failed. We attempted to recompile the program with the option CVTOPT(*VARCHAR), but the compilation failed because the DataLink column was no longer ignored.

In an attempt to avoid the DataLink problem, we created a Logical File, SPORTLF1, with a DataLink field defined. The following listing shows the CRTLF output:

```
5716SS1 V4R4M0  990521               Data Description            TEAMXX/SPORTLF1       10/06/99
15:43:43        Page    1
 File name . . . . . . . . . . . . . . . . . . . . . :  SPORTLF1
   Library name  . . . . . . . . . . . . . . . . . . :  TEAMXX
 File attribute  . . . . . . . . . . . . . . . . . . :  Logical
 Source file containing DDS  . . . . . . . . . . . . :  QDDSSRC
   Library name  . . . . . . . . . . . . . . . . . . :  TEAMXX
 Source member containing DDS  . . . . . . . . . . . :  SPORTLF1
 Source member last changed  . . . . . . . . . . . . :  10/06/99  15:43:31
 Source listing options  . . . . . . . . . . . . . . :  *SOURCE    *LIST      *NOSECLVL  *NOEVENTF
 DDS generation severity level . . . . . . . . . . . :  20
 DDS flagging severity level . . . . . . . . . . . . :  00
 File type . . . . . . . . . . . . . . . . . . . . . :  *DATA
 Authority . . . . . . . . . . . . . . . . . . . . . :  *LIBCRTAUT
 Replace file  . . . . . . . . . . . . . . . . . . . :  *NO
 Text  . . . . . . . . . . . . . . . . . . . . . . . :
 Compiler  . . . . . . . . . . . . . . . . . . . . . :  IBM AS/400 Data Description Processor
                                     Data Description Source
 SEQNBR  *...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8  Date
    100                 R SPREC                    PFILE(TEAMXX/SPORTS)                10/06/99
    200       A           PRDID                                                        10/06/99
    300       A           PRDDES                                                       10/06/99
    400       A           PRDSEA                                                       10/06/99
    500       A           PRDLNK                                                       10/06/99
*                            CPD7426-*****
                        * * * * *  E N D   O F   S O U R C E   * * * * *
 5716SS1 V4R4M0  990521               Data Description            TEAMXX/SPORTLF1       10/06/99
15:43:43        Page    2
                                         Expanded Source
                                                                                         Field
Buffer position
 SEQNBR *...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8 length
Out     In
    100                 R SPREC                    PFILE(TEAMXX/SPORTS)
    200                 PRDID         5A  B      COLHDG('PRDID')                       5
1       1
    300                 PRDDES       25A  B      COLHDG('PRDDES')                     25
6       6
    400                 PRDSEA        1A  B      COLHDG('PRDSEA')                      1
31      31
                * * * * *  E N D   O F   E X P A N D E D   S O U R C E   * * * * *
 5716SS1 V4R4M0  990521               Data Description            TEAMXX/SPORTLF1       10/06/99
15:43:43        Page    3
                                            Messages
    ID       Severity  Number
* CPD7426      30        1      Message . . . . :   Field length too large for data type.
 5716SS1 V4R4M0  990521               Data Description            TEAMXX/SPORTLF1       10/06/99
15:43:43        Page    4
                                         Message Summary
    Total          Informational          Warning        Error       Severe
                      (0-9)              (10-19)       (20-29)      (30-99)
      1                 0                    0            0            1
* CPF7302      40               Message . . . . :   File SPORTLF1 not created in library TEAMXX.
                        * * * * *  E N D   O F   C O M P I L A T I O N   * * * * *
```

The creation failed because the DataLink field was an unacceptable length, even though this had been defined as only 50. We then attempted to create the Logical File, SPORTLF2, over the SPORT table with the DataLink field omitted. The CRTLF listing is shown here:

```
5716SS1 V4R4M0  990521               Data Description            TEAMXX/SPORTLF2       10/06/99
15:43:48        Page    1
 File name . . . . . . . . . . . . . . . . . . . . . :  SPORTLF2
   Library name  . . . . . . . . . . . . . . . . . . :  TEAMXX
 File attribute  . . . . . . . . . . . . . . . . . . :  Logical
 Source file containing DDS  . . . . . . . . . . . . :  QDDSSRC
   Library name  . . . . . . . . . . . . . . . . . . :  TEAMXX
 Source member containing DDS  . . . . . . . . . . . :  SPORTLF2
 Source member last changed  . . . . . . . . . . . . :  10/06/99  15:43:41
 Source listing options  . . . . . . . . . . . . . . :  *SOURCE    *LIST      *NOSECLVL  *NOEVENTF
 DDS generation severity level . . . . . . . . . . . :  20
```

```
 DDS flagging severity level . . . . . . . . . . . . :   00
 File type . . . . . . . . . . . . . . . . . . . . . :   *DATA
 Authority . . . . . . . . . . . . . . . . . . . . . :   *LIBCRTAUT
 Replace file  . . . . . . . . . . . . . . . . . . . :   *NO
 Text  . . . . . . . . . . . . . . . . . . . . . . . :
 Compiler  . . . . . . . . . . . . . . . . . . . . . :   IBM AS/400 Data Description Processor
                                     Data Description Source
 SEQNBR  *...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8  Date
    100                R SPREC                   PFILE(TEAMXX/SPORTS)                 10/06/99
    200       A          PRDID                                                        10/06/99
    300       A          PRDDES                                                       10/06/99
    400       A          PRDSEA                                                       10/06/99
                       * * * * *  E N D   O F   S O U R C E   * * * * *
 5716SS1 V4R4M0  990521              Data Description            TEAMXX/SPORTLF2        10/06/99
15:43:48       Page    2
                                        Expanded Source
                                                                                       Field
Buffer position
 SEQNBR *...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8 length
Out     In
    100                R SPREC                   PFILE(TEAMXX/SPORTS)
   200                 PRDID         5A  B      COLHDG('PRDID')                        5
1       1
   300                 PRDDES       25A  B      COLHDG('PRDDES')                      25
6       6
   400                 PRDSEA        1A  B      COLHDG('PRDSEA')                       1
31      31
                 * * * * *  E N D   O F   E X P A N D E D   S O U R C E   * * * * *
 5716SS1 V4R4M0  990521                   Data Description              TEAMXX/SPORTLF2
10/06/99 15:43:48        Page    3
                                          Message Summary
    Total          Informational          Warning        Error       Severe
                       (0-9)              (10-19)       (20-29)      (30-99)
        0                0                   0             0            0
* CPC7301      00              Message . . . . :   File SPORTLF2 created in library TEAMXX.
                         * * * * *  E N D   O F   C O M P I L A T I O N   * * * * *
```

The Logical File was successfully created. We then created an additional RPG program, SPORTRPG3, which defined the SPORTLF2 logical file as input and the WINTER2 table as output, in other words, no DataLink fields defined on input or output, but the underlying input table with a DataLink column. The compilation listing is shown here:

```
5769RG1 V4R4M0  990521              IBM RPG/400                    TEAMXX/SPORTRPG3    10/06/99
18:14:21      Page       1
 Compiler . . . . . . . . . . . . . . :   IBM RPG/400
 Command Options:
   Program  . . . . . . . . . . . . . :   TEAMXX/SPORTRPG3
   Source file  . . . . . . . . . . . :   TEAMXX/QRPGSRC
   Source member  . . . . . . . . . . :   SPORTRPG3
   Source listing options . . . . . . :   *SOURCE     *XREF       *GEN        *NODUMP     *NOSECLVL
*NOSRCDBG   *NOLSTDBG
   Generation options . . . . . . . . :   *NOLIST     *NOXREF     *NOATR      *NODUMP     *NOOPTIMIZE
   Source listing indentation . . . . :   *NONE
   Type conversion options  . . . . . :   *NONE
   Sort sequence  . . . . . . . . . . :   *HEX
   Language identifier  . . . . . . . :   *JOBRUN
   SAA flagging . . . . . . . . . . . :   *NOFLAG
   Generation severity level  . . . . :   9
   Print file . . . . . . . . . . . . :   *LIBL/QSYSPRT
   Replace program  . . . . . . . . . :   *YES
   Target release . . . . . . . . . . :   *CURRENT
   User profile . . . . . . . . . . . :   *USER
   Authority  . . . . . . . . . . . . :   *LIBCRTAUT
   Text . . . . . . . . . . . . . . . :   *SRCMBRTXT
   Phase trace  . . . . . . . . . . . :   *NO
   Intermediate text dump . . . . . . :   *NONE
   Snap dump  . . . . . . . . . . . . :   *NONE
   Codelist . . . . . . . . . . . . . :   *NONE
   Ignore decimal data error  . . . . :   *NO
   Allow null values  . . . . . . . . :   *NO
 Actual Program Source:
   Member . . . . . . . . . . . . . . :   SPORTRPG3
   File . . . . . . . . . . . . . . . :   QRPGSRC
   Library  . . . . . . . . . . . . . :   TEAMXX
   Last Change  . . . . . . . . . . . :   10/06/99  18:14:17
```

```
  Description  . . . . . . . . . . :   RPG Program to Read PF with DataLink
 5769RG1 V4R4M0  990521               IBM RPG/400                      TEAMXX/SPORTRPG3    10/06/99
18:14:21      Page       2
 SEQUENCE                                                                         IND    DO    LAST
PAGE   PROGRAM
 NUMBER   *...1....+....2....+....3....+....4....+....5....+....6....+....7...*  USE   NUM  UPDATE
LINE   ID
                      S o u r c e   L i s t i n g
        H                                                                                  *****
      100  FSPORTLF2IF  E                  DISK                                          10/06/99
            RECORD FORMAT(S):  LIBRARY TEAMXX FILE SPORTLF2.
                     EXTERNAL FORMAT SPREC RPG NAME SPREC
      200  FWINTER2 O   E                  DISK                                          10/06/99
      300  F            WINTER2                     KRENAMEWINREC                        10/06/99
            RECORD FORMAT(S):  LIBRARY TEAMXX FILE WINTER2.
                     EXTERNAL FORMAT WINTER2 RPG NAME WINREC
  A000000   INPUT  FIELDS FOR RECORD SPREC FILE SPORTLF2 FORMAT SPREC.
  A000001                                      1   5 PRDID
  A000002                                      6  30 PRDDES
  A000003                                     31  31 PRDSEA
      400  C          NEXTR     TAG                                                    10/06/99
      500  C                    READ SPREC                   60                 3      10/06/99
      600  C          *IN60     DOWEQ*OFF                                          B001   10/06/99
      700  C          PRDSEA    IFEQ 'W'                                           B002   10/06/99
      800  C                    WRITEWINREC                                         002   10/06/99
      900  C                    ENDIF                                              E002   10/06/99
     1000  C                    READ SPREC                   60                 3   001   10/06/99
     1100  C                    ENDDO                                              E001   10/06/99
     1200  C                    SETON                     LR                 1            10/06/99
  B000000   OUTPUT FIELDS FOR RECORD WINREC FILE WINTER2 FORMAT WINTER2.
  B000001                            PRDID      5  CHAR    5
  B000002                            PRDDES    30  CHAR   25
          * * * * *   E N D   O F   S O U R C E   * * * * *
        A d d i t i o n a l   D i a g n o s t i c   M e s s a g e s
* 7086       100   RPG PROVIDES BLOCK OR UNBLOCK SUPPORT FOR FILE SPORTLF2.
* 7086       200   RPG PROVIDES BLOCK OR UNBLOCK SUPPORT FOR FILE WINTER2.
* 7150             RECORD SPREC IN FILE SPORTLF2 CONTAINS NULL-CAPABLE FIELDS.
* 7150             RECORD WINTER2 IN FILE WINTER2 CONTAINS NULL-CAPABLE FIELDS.
 5769RG1 V4R4M0  990521               IBM RPG/400                      TEAMXX/SPORTRPG3    10/06/99
18:14:21      Page       3
                     C r o s s   R e f e r e n c e
 File and Record References:
       FILE/RCD    DEV/RCD     REFERENCES (D=DEFINED)
    01  SPORTLF2    DISK           100D
          SPREC                    100D A000000      500     1000
    02  WINTER2     DISK           200D
          WINREC    WINTER2        200D     800  B000000
 Field References:
       FIELD       ATTR    REFERENCES (M=MODIFIED D=DEFINED)
       *IN60       A(1)        600
* 7031  NEXTR       TAG         400D
       PRDDES      A(25)   A000002D B000002D
       PRDID       A(5)    A000001D B000001D
       PRDSEA      A(1)    A000003D     700
       *OFF        LITERAL     600
       'W'         LITERAL     700
 Indicator References:
       INDICATOR  REFERENCES (M=MODIFIED D=DEFINED)
       *IN           600
       LR           1200M
       60            500M     600     1000M
      * * * * *   E N D   O F   C R O S S   R E F E R E N C E   * * * * *
 5769RG1 V4R4M0  990521               IBM RPG/400                      TEAMXX/SPORTRPG3    10/06/99
18:14:21      Page       4
                     M e s s a g e   S u m m a r y
* QRG7031 Severity:  00   Number:    1
          Message . . . . :   The Name or indicator is not referenced.
* QRG7086 Severity:  00   Number:    2
          Message . . . . :   RPG handles blocking function for file. INFDS
            updated only when blocks of data transferred.
* QRG7150 Severity:  00   Number:    2
          Message . . . . :   The record format contains null-capable fields.
      * * * * *   E N D   O F   M E S S A G E   S U M M A R Y   * * * * *
 5769RG1 V4R4M0  990521               IBM RPG/400                      TEAMXX/SPORTRPG3    10/06/99
18:14:21      Page       5
                       F i n a l   S u m m a r y
 Message Count:  (by Severity Number)
            TOTAL    00     10     20     30     40     50
```

```
           5      5      0      0      0      0      0
Program Source Totals:
   Records . . . . . . . . . . :   12
   Specifications  . . . . . . :   12
   Table Records . . . . . . . :   0
   Comments  . . . . . . . . . :   0
PRM has been called.
Program SPORTRPG3 is placed in library TEAMXX. 00 highest severity. Created on 10/06/99 at 18:14:21.
         * * * * *   E N D   O F   C O M P I L A T I O N   * * * * *
```

The compilation was successful. Program SPORTRPG3 was then executed, and it also ran successfully.

In summary, while DataLink columns cannot be used in applications using native I/O techniques, regardless of the programming language, the tables in which they reside can be used by defining a logical file over the underlying data that omits the DataLink column. To gain access to the DataLink columns, you have to use SQL interface.

6.6 DataLinks management considerations

To use the DataLink environment in the most effective way, you are linking files in file systems with tables in the RDBMS. Most application requirements dictate that, in addition to maintaining the integrity of the environment, you also need to ensure the highest availability. Therefore, it is important that you pay attention to managing the DataLink environment and, in particular, the backup and restoration requirements.

Important

To be able to manipulate any object residing in the prefixed directory, the DLFM server must be up and running. This also applies to save and restore activities.

6.6.1 Backup and recovery procedures

Consider the AS/400-only environment. On the AS/400 system, you are dealing with two distinct data storage systems, each with its own support software. The relational tables reside in DB2 Universal Database for AS/400, while the file objects reside in the IFS. Each has its own set of CL commands to handle the save and restore of data. Currently, there is no direct linkage or communication between those two command sets. If you save a table using the SAVOBJ command, there is no facility to automatically save linked files with the SAV command. Therefore, you must manually manage the synchronization of backup copies of related table and file objects.

DB2 Universal Database for AS/400 provides assistance when tables and their linked files are restored to the system. It tracks the status of the links through the DLFM metadata. It also helps with reconciling the links. We ran tests to cover the following scenarios where a table and a linked file have been deleted from the system and have to be restored from a backup copy:

- Restore the table before restoring the linked files.
- Restore the linked file before restoring the table.

Note that it is not possible to delete linked files before the associated table has been dropped due to the integrity rules applied by the DataLinks Filter.

Two save files were created, one to receive the backup copy of the saved table and the other the backup copy of the linked file. Figure 141 shows the script used to run the first test.

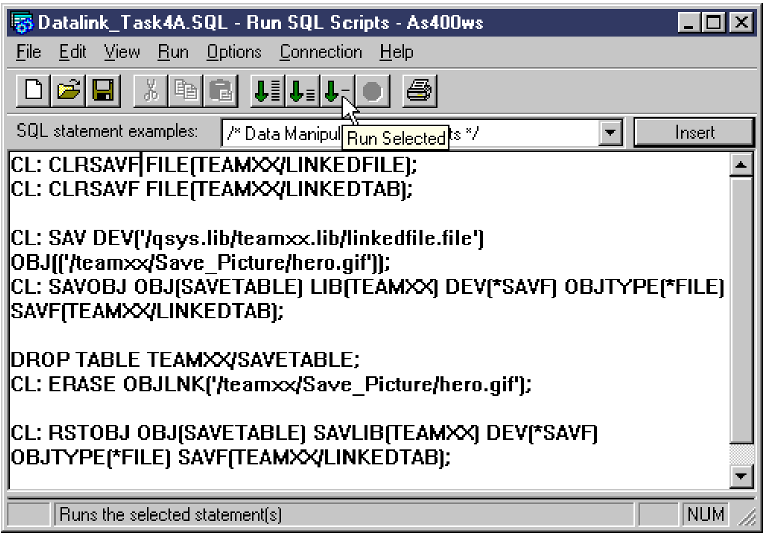


Figure 141. Script for save/restore exercise: Restore table before file

The statements perform the following steps:

1. Clear the save file for the linked file.
2. Clear the save file for the table.
3. Save the table to the save file.
4. Save the linked file to the save file.
5. Drop the table from the system.
6. Delete the linked file from the system.
7. Restore the table.

By running the DSPFD CL command for the table that has just been restored and paging down the resulting displays, you see a screen similar to the one shown in Figure 142 on page 204. This display is shown for a table called SAVETABLE in library TEAMXX. Look for the line "File is in link pending status". This shows that this table has at least one linked file object that is currently not present on the system. The DLFM has determined the link pending status from the metadata it maintains in the QDLFM library. As a result, it also marks the table as read-only. Any attempt to insert, update, or delete rows is rejected until the pending links are reconciled.

```
                              Display Spooled File
 File  . . . . . :   QPDSPFD                                  Page/Line   1/58
 Control . . . . .                                            Columns     1 - 130
 Find  . . . . . .
 *...+....1....+....2....+....3....+....4....+....5....+....6....+....7....
+....8....+....9....+....0....+....1....+....2....+....3
   Allow delete operation . . . . . . . . . . : ALWDLT   *YES
   Record format level check . . . . . . . . . : LVLCHK   *YES
   Access path . . . . . . . . . . . . . . . . :       Arrival
   Access path size  . . . . . . . . . . . . . : ACCPTHSIZ *MAX1TB
   Maximum record length . . . . . . . . . . . :         225
   File is currently journaled . . . . . . . . :        Yes
   Current or last journal . . . . . . . . . . :       TEAMXX_JRN
     Library . . . . . . . . . . . . . . . . . :       TEAMXX
   Journal images  . . . . . . . . . . . . . . : IMAGES   *AFTER
   Journal entries to be omitted . . . . . . . : OMTJRNE   *NONE
   Last journal start date/time  . . . . . . . :       09/23/99 19:10:53
   File is in link pending status  . . . . . . :        Yes
 Access Path Description
   Access path . . . . . . . . . . . . . . . . :       Arrival
   Sort Sequence . . . . . . . . . . . . . . . : SRTSEQ   *HEX
   Language identifier . . . . . . . . . . . . : LANGID   ENU
 Member Description
   Member  . . . . . . . . . . . . . . . . . . : MBR      SAVETABLE
    Member level identifier . . . . . . . . . :       0990923190615

More...
 F3=Exit F12=Cancel F19=Left F20=Right F24=More keys
```

Figure 142. DSPFD of table: Link pending status after file restore

A new CL command has been introduced in V4R4 of OS/400. This is the Work with Physical File DataLinks (WRKPFDL) command. When you run the following command for the same SAVETABLE table, you should see a display similar to the one shown in Figure 143:

```
WRKPFDL FILE(TEAMXX/SAVETABLE)
```

This shows that the field PICTU00001 is in Link Pending status. PICTU00001 is the system-derived short name for the DataLink column Picture_Link. This CL command displays the status of all DataLink columns defined for a table.

```
                    Work with Physical File DataLinks

Type options, press Enter.
  2=Reconcile   6=Display

                                                          Link
Opt     File           Library        Field          Pending
        SAVETABLE      TEAMXX         PICTU00001       YES















                                                                          Bottom
Parameters for option 6 or command
===>
F3=Exit   F5=Refresh     F9=Retrieve   F12=Cancel   F15=Sort by
F16=Repeat position to   F17=Position to
```

Figure 143. WRKPFDL TEAMXX/SAVETABLE: Link pending

If you type option 6 next to the SAVETABLE table and press Enter, you should see a display like the example shown in Figure 144. The only additional information displayed is the name of the RDBMS server.

```
                         Display Physical File Member
File . . . . . . :   QDL_000001         Library  . . . . :   QTEMP
Member . . . . . :   QDL_000001         Record . . . . . :   1
Control  . . . . .                      Column . . . . . :   1
Find . . . . . . .
*...+....1....+....2....+....3....+....4....+....5....+....6....+....7
                           Display DataLink File Attributes
File name  . . . . . . . . . . . . . . . :  SAVETABLE
Library name . . . . . . . . . . . . . . :  TEAMXX

Field name . . . . . . . . . . . . . . . :  PICTU00001
Link pending . . . . . . . . . . . . . . :  Yes
Server names . . . . . . . . . . . . . . :  AS400WS
                          ****** END OF DATA ******
```

Figure 144. DataLink file attributes for TEAMXX/SAVETABLE

With the table in link pending status, we attempted to perform a write operation on the table by trying to delete a row. The result was the error shown in Figure 145 on page 206.

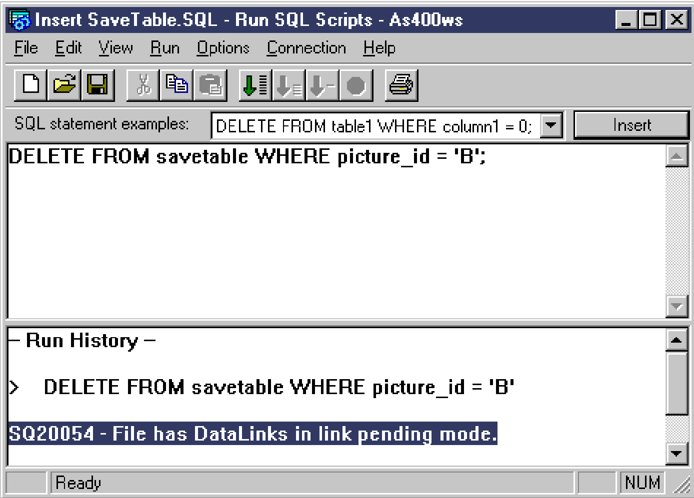


Figure 145. Delete from table in link pending status: Error message

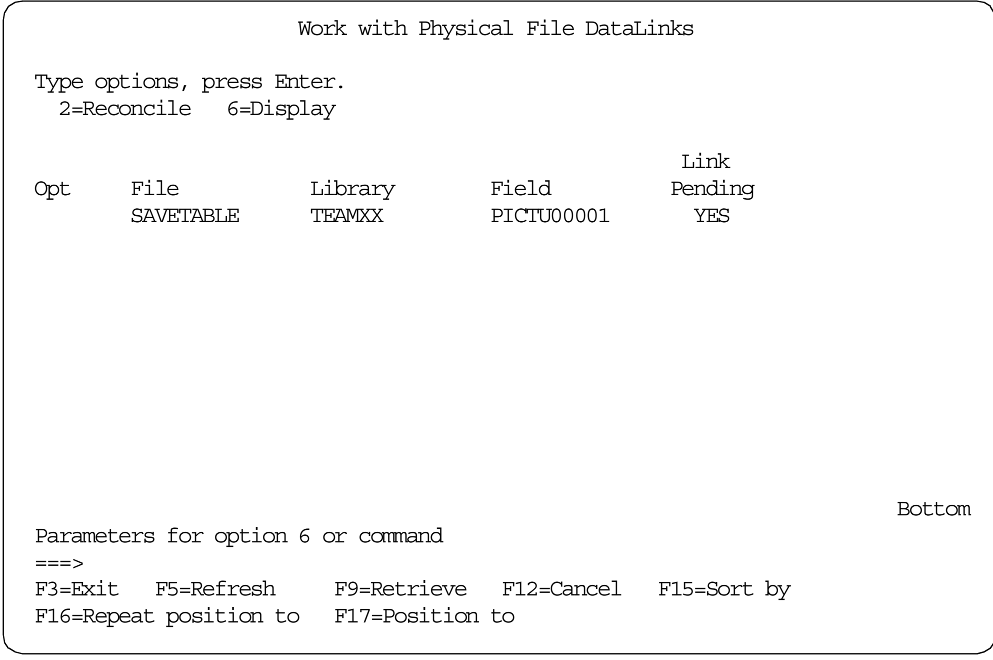
Before the pending links could be reconciled, we restored the file object from the save file by using the RST CL command:

```
RST DEV('/qsys.lib/teamxx.lib/linkedfile.file')
OBJ(('TEAMXX/Save_Picture/hero.gif'))
```

We then ran the following command once again:

```
WRKPFDL FILE(TEAMXX/SAVETABLE)
```

The display shown in Figure 146 appeared. This shows that field PICTU00001 is still in Link Pending status, even though the linked file has been restored.



```
                         Work with Physical File DataLinks

Type options, press Enter.
  2=Reconcile   6=Display

                                                          Link
Opt     File          Library       Field          Pending
        SAVETABLE     TEAMXX        PICTU00001       YES

                                                                      Bottom
Parameters for option 6 or command
===>
F3=Exit   F5=Refresh     F9=Retrieve   F12=Cancel   F15=Sort by
F16=Repeat position to   F17=Position to
```

Figure 146. WRKPFDL TEAMXX/SAVETABLE: Link pending after file restore

We then entered option 2 (Reconcile) in the Opt field on the SAVETABLE line. Specifying the reconcile option does not actually perform the reconciliation at this stage. It simply marks the table as being eligible for reconciliation. Another new CL command has been provided in V4R4 of OS/400 to actually perform the reconciliation. This command is Edit DataLink File Attributes (EDTDLFA). Running this command resulted in a display like the example shown in Figure 147.

This shows that the table SAVETABLE in library TEAMXX is in a "Link Pending" status and has been marked for reconciliation. It now provides you with an opportunity to actually perform the reconciliation.

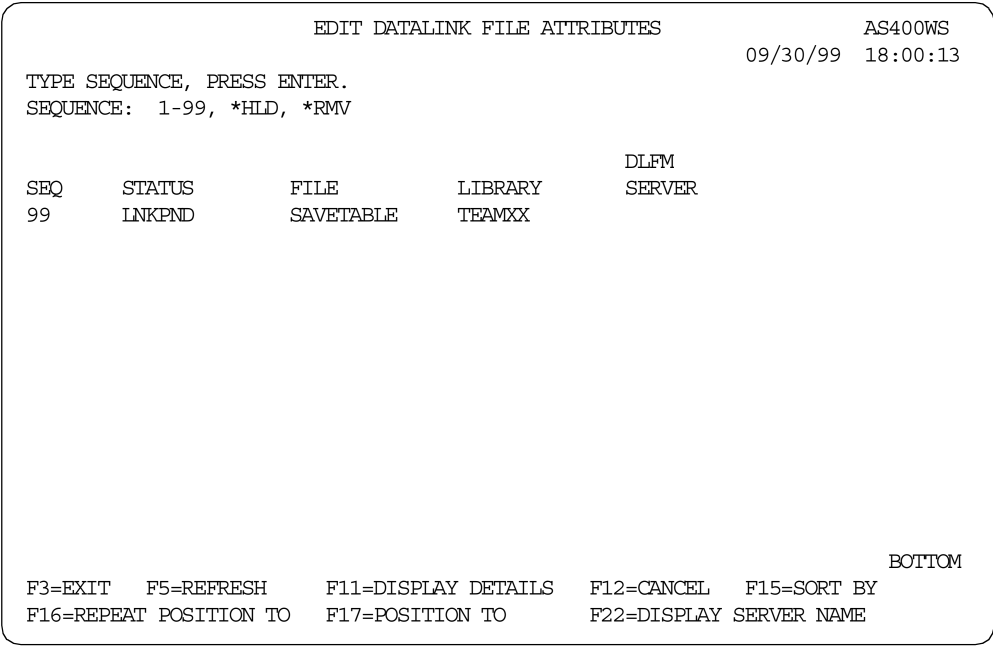


Figure 147. EDTDLFA display: Status LNKPND

On this display, you over-type the value in the SEQ column with any value between 01 and 98 inclusive and press the Enter key. The display should now appear like the one shown in Figure 148 on page 208.

```
                         EDIT DATALINK FILE ATTRIBUTES                AS400WS
                                                             09/30/99  18:07:17
TYPE SEQUENCE, PRESS ENTER.
SEQUENCE:  1-99, *HLD, *RMV

                                                   DLFM
SEQ     STATUS        FILE          LIBRARY        SERVER
98      READY         SAVETABLE     TEAMXX

                                                                        BOTTOM
F3=EXIT   F5=REFRESH     F11=DISPLAY DETAILS   F12=CANCEL   F15=SORT BY
F16=REPEAT POSITION TO   F17=POSITION TO       F22=DISPLAY SERVER NAME
```

Figure 148. EDTDLFA display: Status READY

The DLFM is ready to attempt to reconcile any pending links. The sequence number can be used to prioritize the order in which table reconciliation is executed when there are several tables to be processed that may have pending links to thousands of files.

If you refresh the display, it should appear like the example in Figure 149.

```
                         EDIT DATALINK FILE ATTRIBUTES                  ASM23
                                                             09/30/99  18:44:01
TYPE SEQUENCE, PRESS ENTER.
SEQUENCE:  1-99, *HLD, *RMV

                                                   DLFM
SEQ     STATUS        FILE          LIBRARY        SERVER

  (No DataLinks to display)

                                                                        BOTTOM
F3=EXIT   F5=REFRESH     F11=DISPLAY DETAILS   F12=CANCEL   F15=SORT BY
F16=REPEAT POSITION TO   F17=POSITION TO       F22=DISPLAY SERVER NAME
```

Figure 149. EDTDLFA display: Links reconciled

Note

If you press F5 very quickly, you may see the Status field displayed as RUN. This indicates that the reconciliation is still in progress. Redisplaying the link pending status of the table by running the DSPFD command resulted in a display like the one shown in Figure 150.

```
                              Display Spooled File
 File  . . . . . :   QPDSPFD
 Control . . . . .
 Find  . . . . . .

*...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
...
   Allow delete operation  . . . . . . . . . . : ALWDLT     *YES
   Record format level check . . . . . . . . . : LVLCHK     *YES
   Access path . . . . . . . . . . . . . . . . :            Arrival
   Access path size  . . . . . . . . . . . . . : ACCPTHSIZ  *MAX1TB
   Maximum record length . . . . . . . . . . . :              225
   File is currently journaled . . . . . . . . :            Yes
   Current or last journal . . . . . . . . . . :            TEAMXX_JRN
     Library . . . . . . . . . . . . . . . . . :            TEAMXX
   Journal images  . . . . . . . . . . . . . . : IMAGES     *AFTER
   Journal entries to be omitted . . . . . . . : OMTJRNE    *NONE
   Last journal start date/time  . . . . . . . :            09/30/99  15:51:06
   File is in link pending status  . . . . . . :            No
  Access Path Description
   Access path . . . . . . . . . . . . . . . . :            Arrival
   Sort Sequence . . . . . . . . . . . . . . . : SRTSEQ     *HEX
   Language identifier . . . . . . . . . . . . : LANGID     ENU
  Member Description
   Member  . . . . . . . . . . . . . . . . . . : MBR        SAVETABLE
     Member level identifier . . . . . . . . . :            0990930154509

 F3=Exit   F12=Cancel   F19=Left   F20=Right   F24=More keys
```

Figure 150. DSPFD of TEAMXX/SAVETABLE: Link pending status after reconciliation

Rerunning the WRKPFDL command also confirmed that the DataLink column is no longer in Link Pending status as shown in Figure 151 on page 210.

```
                    Work with Physical File DataLinks

 Type options, press Enter.
   2=Reconcile   6=Display

                                                      Link
 Opt     File           Library        Field         Pending
         SAVETABLE      TEAMXX         PICTU00001      NO










                                                                        Bottom
 Parameters for option 6 or command
 ===>
 F3=Exit   F5=Refresh     F9=Retrieve   F12=Cancel   F15=Sort by
 F16=Repeat position to   F17=Position to
```

Figure 151. WRKPFDL TEAMXX/SAVETABLE: After link reconciliation

We then ran the second test. This was identical to the first test except that the file was restored to the system before the table to which it was linked. Displaying the link pending status of the table with the DSPFD command immediately after it was restored resulted in the display shown in Figure 152. This shows that the table was restored, and its file links were automatically reconciled. Rerunning the WRKPFDL command also confirmed that the table was not in link pending status as shown in Figure 153.

```
                         Display Spooled File
File  . . . . . :   QPDSPFD
Control . . . . .
Find  . . . . . .

*...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8.
   Allow delete operation  . . . . . . . . . . : ALWDLT     *YES
   Record format level check . . . . . . . . . : LVLCHK     *YES
   Access path . . . . . . . . . . . . . . . . :            Arrival
   Access path size  . . . . . . . . . . . . . : ACCPTHSIZ  *MAX1TB
   Maximum record length . . . . . . . . . . . :              225
   File is currently journaled . . . . . . . . :            Yes
   Current or last journal . . . . . . . . . . :            TEAMXX_JRN
     Library . . . . . . . . . . . . . . . . . :            TEAMXX
   Journal images  . . . . . . . . . . . . . . : IMAGES     *AFTER
   Journal entries to be omitted . . . . . . . : OMTJRNE    *NONE
   Last journal start date/time  . . . . . . . :            09/30/99  15:51:06
   File is in link pending status  . . . . . . :            No
 Access Path Description
   Access path . . . . . . . . . . . . . . . . :            Arrival
   Sort Sequence . . . . . . . . . . . . . . . : SRTSEQ     *HEX
   Language identifier . . . . . . . . . . . . : LANGID     ENU
 Member Description
   Member  . . . . . . . . . . . . . . . . . . : MBR        SAVETABLE
     Member level identifier . . . . . . . . . :            0990930154509

F3=Exit F12=Cancel F19=Left F20=Right F24=More keys
```

Figure 152. DSPFD of TEAMXX/SAVETABLE: Link pending status after table restore

```
                      Work with Physical File DataLinks

 Type options, press Enter.
   2=Reconcile   6=Display

                                                          Link
 Opt     File            Library         Field           Pending
         SAVETABLE       TEAMXX          PICTU00001        NO

                                                                         Bottom
 Parameters for option 6 or command
 ===>
 F3=Exit   F5=Refresh     F9=Retrieve   F12=Cancel   F15=Sort by
 F16=Repeat position to   F17=Position to
```

Figure 153. WRKPFDL TEAMXX/SAVETABLE: No link pending

The two tests show that the integrity of the DataLinks environment can be maintained by either restoring the tables or the linked files first. However, we *strongly* advise that you base your normal recovery policy on restoring the files first. This approach avoids placing the tables into the Link Pending status and, therefore, removes the need for the links to be reconciled.

There are other save/restore and copy considerations for tables with DataLinks columns. The save of such a table with a target release prior to OS/400 V4R4 is not supported.

6.7 Using DataLinks in a heterogeneous environment

Disclaimer

At publication time, there were some known interopability issues and problems between the different IBM DB2 DataLink managers. Please check the following Web site for the latest status on these issues before implementing a cross-platform DataLink solution: `http://www.as400.ibm.com/db2/dlinkinter.htm`

In 6.2.2, "DataLink file manager" on page 151, we described the DataLinks architecture and how it was designed in a way that allowed the DLFM to reside on any file server with a supporting operating system and RDBMS. Currently, support is provided by DB2 Universal Database for AS/400 V4R4 and by DB2 Universal Database V6 running on Windows NT or AIX. This permits the type of heterogeneous environment shown in Figure 154, where DB2 Universal Database tables on any of the three platforms can be linked to file objects residing in any of the three different file systems.

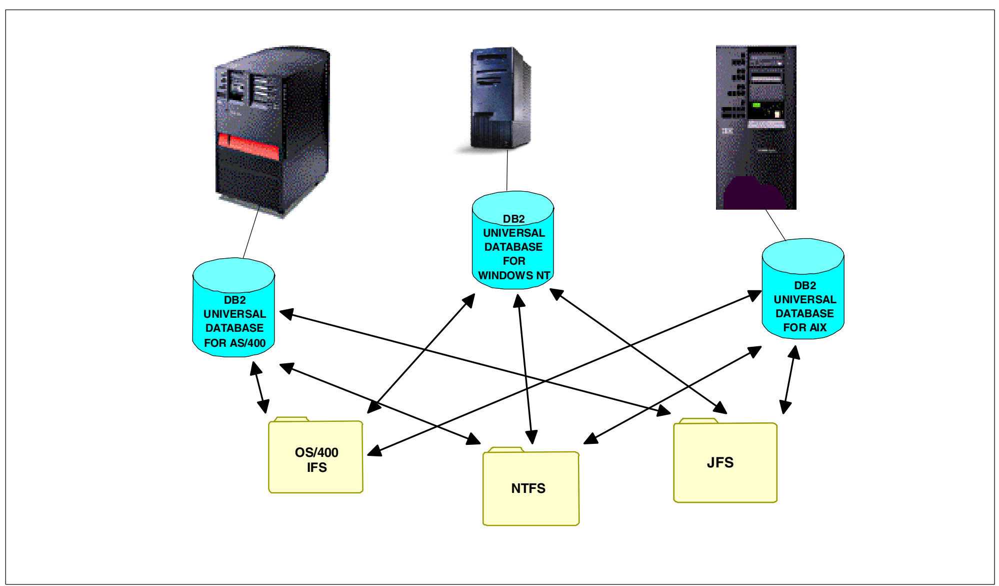


Figure 154. The IBM heterogeneous DataLink server environment

6.7.1 DataLinks Manager for Windows NT and for AIX

This section describes the different components that make up a database system that is using DB2 DataLinks Manager for Windows NT and for AIX. The Windows NT and the AIX versions are functionally identical except for where they interact with the underlying file systems, NT File System (NTFS) and Journaled File System (JFS) respectively. These components include the:

- DataLinks Server
- DB2 Universal Database Server
- DB2 Client

The DataLinks Server is comprised of the following components:

- **DataLinks File Manager (DLFM):** The DLFM has identical functions to the DB2 Universal Database for AS/400 DLFM. It registers all the files on a particular DataLinks server that are linked to a DB2 database. It receives and processes link-file and unlink-file messages arising from SQL INSERT, UPDATE, and DELETE statements that reference a DATALINK column. For each linked file, the DLFM logically tracks the database instance, the fully qualified table name, and the column name referred to in the SQL statement. However, unlike DB2 Universal Database for AS/400, it also tracks previously linked files, if they were linked to a DATALINK column for which the RECOVERY=YES option was specified, during table creation. This allows DB2 to provide point-in-time roll-forward recovery for any file that is specified by a DATALINK column. The *Recovery Yes* option is not supported by V4R4 of DB2 Universal Database for AS/400.
- **Data Links Filesystem Filter (DLFF):** Filter commands to ensure that linked files are not deleted, renamed, or the file's attributes are not changed. Optionally, it also filters commands to ensure that proper access authority exists.
- **DB2 (Logging Manager):** This is a Logging Manager that contains the DLFM_DB database. It provides equivalent function to the QDLFM library on the AS/400 system. This database contains registration information about databases that can connect to a Data Links server (equivalent to the Host Database entries in the AS/400 QDLFM table dfm_dbid), and the sharename of the drives that are managed by a DLFF (equivalent to the prefix entries in the AS/400 QDLFM table dfm_prfx). The DLFM_DB database also contains information about files that have been linked, unlinked, or backed up on a Data Links server (the AS/400 QDLFM library equivalent is the table dfm_file, except this does not track file backup activity). This database is created during the installation of DB2 Data Links Manager.

  Unlike V4R4 of DB2 Universal Database for AS/400, DB2 DataLinks Manager can provide point-in-time roll-forward recovery on the Data Links server (if the RECOVERY=YES option was specified during table creation) for any linked file that is specified by a DATALINK column. The files can be backed up on a disk or using ADSTAR Distributed Storage Manager (ADSM). The files that are linked via a DATALINK column are ensured to be backed up when your database is backed up.

The DB2 Universal Database Server is the location of the main database where the DataLinks server is registered. It contains the table that includes the DATALINK data type. No sharing is required between a DB2 server and a DataLinks Server. All communication is done through a port reserved for communications. The remote DB2 Universal Database server can only be participating in a single-partitioned database system. Unlike DB2 Universal Database for AS/400s MultiSystem option, DB2 DataLinks Manager does not support interaction with partitioned database systems.

A DB2 Client connects to a DB2 server as normal. In the case of Windows NT, a remote client can share a drive under the control of a DataLinks Filesystem Filter

that is installed on a DataLinks server. This way, the client can directly access the files on the DataLinks server. AIX provides this capability with a Network File System (NFS) mount of the file system under the control of the DataLinks Filesystem Filter. This is equivalent to the way that AS/400 provides simultaneous access to DB2 Universal Database for AS/400 tables and Integrated File System files through, for example, the C programming language.

The DataLink environment and components for Windows NT are shown in Figure 155 and for AIX in Figure 156.

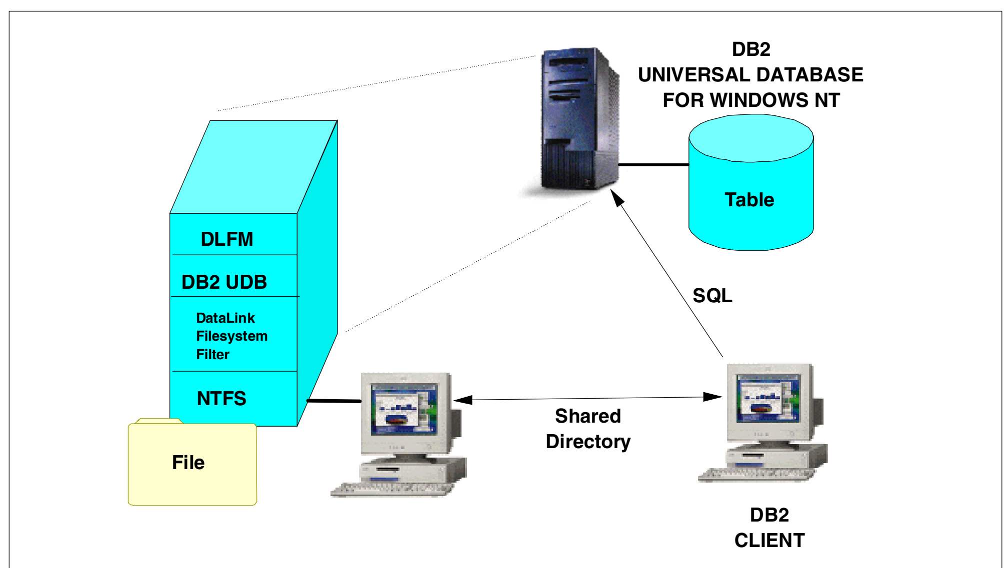


Figure 155. DataLink environment for Windows NT

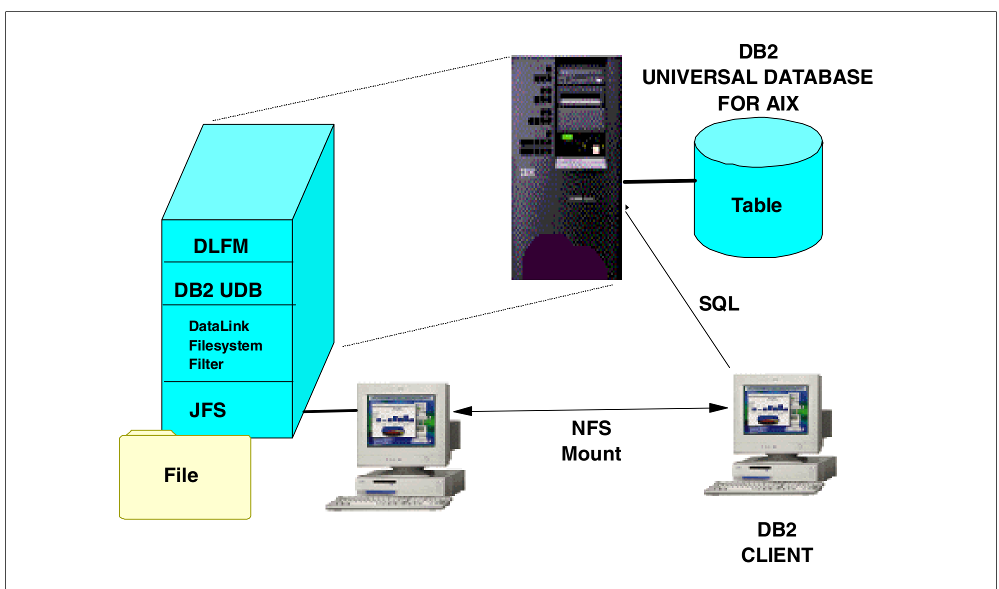


Figure 156. DataLink environment for AIX

Appendix A. Source code listings

This appendix contains detailed example programs implementing functions and concepts covered in this redbook. The logic and programmimg techniques used in the programs listed here are thoroughly explained in the relevant sections of the redbook.

Important information

These example programs have not been subjected to any formal testing. They are provided "as is"; they should be used for reference only. Please refer to the Appendix B, “Special notices” on page 229.

A.1 UDTLABA: Using UDTs

```
/*--------------------------------------------------------------------
/
/  File:         UDTLABA
/
/  Description: AS/400 DB2 UDT test program
/
/  Usage:        CALL TEAMXX/UDTLABA
/
/  Author:       Mark Endrei
/
/  Complile:     CRTSQLCI OBJ(TEAMXX/UDTLABA) SRCFILE(TEAMXX/QCSRC) +
/                  SRCMBR(UDTLABA) COMMIT(*NONE)
/                CRTPGM PGM(TEAMXX/UDTLABA) MODULE(TEAMXX/UDTLABA)
/
/  Copyright (c) 1999 IBM Corp.
/  All Rights Reserved.
/
*/

#include <stdio.h>
#include <decimal.h>


/*--------------------------------------------------------------------
/
/  Description: Main program
/
/  Usage:        CALL TEAMXX/UDTLABA
/                returns         -1 on success
/                                 0 on failure
/
*/
int main(int argc, char** argv)
{
  exec sql include SQLCA;

  /* host variable declarations */
  decimal(11,2) dec_price_in  = 88.88d;
  decimal(11,2) dec_price_out =  0.00d;

  printf("\n");
  printf("AS/400 DB2 UDB UDT Lab Test Program: %s\n", argv??(0??));
  printf("\n");

  printf(
    "/* host variable declaration */\n" \
    "decimal(11,2) dec_price_in  = 88.88d;\n" \
    "\n" \
    "/* implicit cast on assignment from decimal into money */\n" \
    "exec sql\n" \
    "  update prodmast01 set product_price = :dec_price_in\n" \
    "  where product_number = srlnumber('00001');\n" \
    "\n" \
```

```
    "Use Operations Navigator to view current\n" \
    "product_price for product_number 00001.\n" \
    "\n" \
    "Then hit Enter key to continue...\n");
  getchar();

  /* implicit cast on assignment from decimal into money */
  exec sql
    update prodmast01 set product_price = :dec_price_in
    where product_number = srlnumber('00001');

  if (SQLCODE != 0)
  {
    printf("SQL Error, SQLCODE = %d\n", SQLCODE);
  }

  printf(
    "SQL statement executed.\n" \
    "\n" \
    "Use Operations Navigator to view updated\n" \
    "product_price for product_number 00001.\n" \
    "\n" \
    "Then hit Enter key to continue...\n");
  getchar();

  printf(
    "/* host variable declaration */\n" \
    "decimal(11,2) dec_price_out =  0.00d;\n" \
    "\n" \
    "/* implicit cast on assignment from money into decimal */\n" \
    "exec sql\n" \
    "  select product_price into :dec_price_out from prodmast01\n" \
    "  where product_number = srlnumber('00001');\n" \
    "\n" \
    "Hit Enter key to continue...\n");
  getchar();

  /* implicit cast on assignment from money into decimal */
  exec sql
    select product_price into :dec_price_out from prodmast01
    where product_number = srlnumber('00001');

  if (SQLCODE != 0)
  {
    printf("SQL Error, SQLCODE = %d\n", SQLCODE);
  }

  printf(
    "SQL statement executed.\n" \
    "\n" \
    "value assigned to host variable :dec_price_out = %D(11,2)\n" \
    "\n", dec_price_out);

  return -1;

}
```

A.2 UDTLABB: Casting UDTs

```
/*--------------------------------------------------------------------
/
/  File:        UDTLABB
/
/  Description: AS/400 DB2 UDT test program
/
/  Usage:       CALL TEAMXX/UDTLABB
/
/  Author:      Mark Endrei
/
/  Complile:    CRTSQLCI OBJ(TEAMXX/UDTLABB) SRCFILE(TEAMXX/QCSRC) +
/                 SRCMBR(UDTLABB) COMMIT(*NONE)
/               CRTPGM PGM(TEAMXX/UDTLABB) MODULE(TEAMXX/UDTLABB)
/
/  Copyright (c) 1999 IBM Corp.
/  All Rights Reserved.
/
```

```
*/

#include <stdio.h>
#include <decimal.h>

#define HOST_STRUCT_SIZE 10


/*--------------------------------------------------------------------
/
/  Description: Main program
/
/  Usage:       CALL TEAMXX/UDTLABB
/               returns         -1 on success
/                                0 on failure
/
*/
int main(int argc, char** argv)
{
  exec sql include SQLCA;

  /* host variable declarations */
  long int_price_in  = 111;
  _Packed struct {
    char number??(5??);
    char name??(25??);
    long int_price_out;
  } product_rec??(10??);
  struct { short ind??(3??); } product_ind??(10??);

  int i;

  printf("\n");
  printf("AS/400 DB2 UDB UDT Lab Test Program: %s\n", argv??(0??));
  printf("\n");

  printf(
    "/* host variable declaration */\n" \
    "long int_price_in = 111;\n" \
    "\n" \
    "/* implicit cast on assignment from long integer into money */\n" \
    "exec sql\n" \
    "  insert into prodmast01 (product_number, product_name, product_price)\n" \
    "  values( '00004', 'New product', :int_price_in);\n" \
    "\n" \
    "Hit Enter key to insert row...\n");
  getchar();

  /* implicit cast on assignment from long integer into money */
  exec sql
    insert into prodmast01 (product_number, product_name, product_price)
    values( '00004', 'New product', :int_price_in);

  if (SQLCODE != 0)
  {
    printf("SQL Error, SQLCODE = %d\n", SQLCODE);
  }

  printf(
    "SQL statement executed.\n" \
    "\n" \
    "Hit Enter key to continue...\n");
  getchar();

  printf(
    "/* host variable declaration */\n" \
    "_Packed struct {\n" \
    "  char number??(5??);\n" \
    "  char name??(25??);\n" \
    "  long int_price_out;\n" \
    "} product_rec??(10??);\n" \
    "struct { short ind??(3??); } product_ind??(10??);\n" \
    "\n" \
    "/* declare and open fetch cursor */\n" \
    "exec sql\n" \
    "  declare c1 cursor for\n" \
    "  select product_number, product_name, product_price from prodmast01;\n" \
    "exec sql open c1;\n" \
```

```
    "/* implicit cast on assignment from money into long integer */\n" \
    "exec sql\n" \
    "  fetch c1 for 10 rows into :product_rec indicator :product_ind;\n" \
    "\n" \
    "Hit Enter key to continue...\n");
  getchar();

  /* declare and open fetch cursor */
  exec sql
    declare c1 cursor for
    select product_number, product_name, product_price from prodmast01;
  exec sql open c1;
  /* implicit cast on assignment from money into long integer */
  exec sql
    fetch c1 for 10 rows into :product_rec indicator :product_ind;

  if (SQLCODE != 0)
  {
    printf("SQL Error, SQLCODE = %d\n", SQLCODE);
  }

  printf(
    "SQL statement executed.\n" \
    "\n" \
    "values assigned to host variable array:\n" \
    "\n");
  for (i=0; i<sqlca.sqlerrd??(2??); i++)
  {
    printf("product_rec??(%d??).number        = %5.5s\n",
      i, product_rec??(i??).number);
    printf("product_rec??(%d??).name          = %25.25s\n",
      i, product_rec??(i??).name);
    printf("product_rec??(%d??).int_price_out = %d\n",
      i, product_rec??(i??).int_price_out);
  }
  printf(
    "\n" \
    "Hit Enter key to delete new row...\n");
  getchar();

  exec sql
    delete from prodmast01 where product_number = srlnumber('00004');

  if (SQLCODE != 0)
  {
    printf("SQL Error, SQLCODE = %d\n", SQLCODE);
  }

  return -1;

}
```

A.3 PictCheck: External UDF

```
#define GIF_HEADER_LENGTH 6
#define BMP_HEADER_LENGTH 2
#define GIF_FUNCTION "ISGIF"
#define BMP_FUNCTION "ISBMP"

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sqludf.h>

int fun_CheckHeader( char *, int, char * );

typedef struct
{
        unsigned long length;
        char          data??( 1 ??);
} BLOB1M;


void SQL_API_FN fun_CheckPictureType( BLOB1M         *str_ProductPicture,
                SQLUDF_INTEGER *nmi_IsCorrect,
                SQLUDF_NULLIND *nms_InputNullIndicator01,
```

```
                SQLUDF_NULLIND *nms_OutputNullIndicator01,
                SQLUDF_CHAR    sqludf_sqlstate??( SQLUDF_SQLSTATE_LEN + 1 ??),
                SQLUDF_CHAR    sqludf_fname??( SQLUDF_FQNAME_LEN + 1 ??),
                SQLUDF_CHAR    sqludf_fspecname??( SQLUDF_SPECNAME_LEN + 1 ??),
                SQLUDF_CHAR    sqludf_msgtext??( SQLUDF_MSGTEXT_LEN + 1 ??) )
{
     char       chr_GifHeader87??( GIF_HEADER_LENGTH ??) = { 0x47,
                                                             0x49,
                                                             0x46,
                                                             0x38,
                                                             0x37,
                                                             0x61 };
     char       chr_GifHeader89??( GIF_HEADER_LENGTH ??) = { 0x47,
                                                             0x49,
                                                             0x46,
                                                             0x38,
                                                             0x39,
                                                             0x61 };
     char       chr_BmpHeader??( BMP_HEADER_LENGTH ??)   = { 0x42, 0x4D};
     char       *chr_FunctionResolution;
     int        nmi_CompareResult01 = 0;
     int        nmi_CompareResult02 = 0;


     if ( *nms_InputNullIndicator01 == -1 )
     {
        *nms_OutputNullIndicator01 = -1;
        return;
     }

    chr_FunctionResolution = strstr( sqludf_fname, GIF_FUNCTION );

    if ( chr_FunctionResolution != NULL )
    {
       nmi_CompareResult01 = fun_CheckHeader( str_ProductPicture->data,
                                              GIF_HEADER_LENGTH,
                                              chr_GifHeader87 );
       nmi_CompareResult02 = fun_CheckHeader( str_ProductPicture->data,
                                              GIF_HEADER_LENGTH,
                                              chr_GifHeader89 );

       if ( ( nmi_CompareResult01 == 1 ) || ( nmi_CompareResult02 == 1 ) )
       {
          *nmi_IsCorrect = 1;
          *nms_OutputNullIndicator01 = 0;
       }
       else
       {
          *nmi_IsCorrect = 0;
          *nms_OutputNullIndicator01 = 0;
       }

       return;
    }

    chr_FunctionResolution = strstr( sqludf_fname, BMP_FUNCTION );

    if ( chr_FunctionResolution != NULL )
    {
       nmi_CompareResult01 = fun_CheckHeader( str_ProductPicture->data,
                                              BMP_HEADER_LENGTH,
                                              chr_BmpHeader );

       if ( nmi_CompareResult01 == 1 )
       {
          *nmi_IsCorrect = 1;
          *nms_OutputNullIndicator01 = 0;
       }
       else
       {
          *nmi_IsCorrect = 0;
          *nms_OutputNullIndicator01 = 0;
       }

       return;
    }

    *nms_OutputNullIndicator01 = -1;
```

```
    strcpy( sqludf_sqlstate, "38501" );
    strcpy( sqludf_msgtext, "Unregistered function" );
    return;
}
```

A.4 ChkHdr

```
#define MAX_HEADER_SIZE 10

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int fun_CheckHeader( char *chr_HeaderData,
             int nmi_HeaderLength,
             char *chr_HeaderFormat )
{
  char chr_HeaderString[ MAX_HEADER_SIZE ];
  int nmi_CompareResult;

  memcpy( chr_HeaderString, chr_HeaderData, nmi_HeaderLength );
 nmi_CompareResult = memcmp( chr_HeaderString, chr_HeaderFormat, nmi_HeaderLength );

  if ( nmi_CompareResult != 0 )
  {
    return 0;
  }
  else
  {
    return 1;
  }
}
```

A.5 RunGetPicture: Testing GetPicture UDF

```
#include <stdio.h>

EXEC SQL INCLUDE SQLCA;

EXEC SQL BEGIN DECLARE SECTION;
         SQL TYPE IS BLOB(1M)  bin_ProductPicture;
         SQL TYPE IS CLOB(50K) chs_ProductDescription;
         char chs_ProductNumber??( 5 ??);
         char chs_Description??( 1024 ??);
EXEC SQL END DECLARE SECTION;

void main( int argc, char **argv )
{
     EXEC SQL WHENEVER NOT FOUND GOTO BadNews;

     strcpy( chs_ProductNumber, argv??(1 ??) );
     printf( "The product number - %s\n", chs_ProductNumber );

     EXEC SQL
          select Product_Description
          into :chs_ProductDescription
          from
          prodmast01
          where
          product_number = SRLNUMBER( :chs_ProductNumber );

     EXEC SQL
          DECLARE cur_Picture CURSOR FOR
          Select GetPicture( :chs_ProductDescription,
                             Product_Description,
                             Product_Picture )
          from
          prodmast01;

     EXEC SQL open cur_Picture;

     bin_ProductPicture.length = 0;
     strcpy( bin_ProductPicture.data, " " );
```

```
     EXEC SQL fetch cur_Picture into :bin_ProductPicture;

     while ( sqlca.sqlcode != 100 )
     {
           printf( "\n" );
           if ( bin_ProductPicture.length != 0 )
           {
              printf( "Values returned by GetPicture( CLOB, PRDDESC, " );
              printf( "PICTURE ): \n" );
              printf( "The picture length - %d\n", bin_ProductPicture.length );
              printf( "The picture data - %s\n", bin_ProductPicture.data );
           }
           else
           {
              printf( "The GetPicture function( CLOB, PRDDESC, PICTURE ) " );
              printf( "returned NULL\n" );
           }

           bin_ProductPicture.length = 0;
           strcpy( bin_ProductPicture.data, " " );

           EXEC SQL fetch cur_Picture into :bin_ProductPicture;
     }

     BadNews:
           EXEC SQL close cur_Picture;
           return;
}
```

A.6 Rating: External UDF using SCRATCHPAD

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <decimal.h>
#include <sqludf.h>

typedef struct
{
   decimal( 15, 5 ) *nmpd_LargeValue;
   long             nml_RequiredRating;
   long             nml_ValuesStored;
} str_ScratchPad;

void SQL_API_FN fun_Rating( decimal( 11, 2 )  *nmpd_InputMoneyValue,
     SQLUDF_INTEGER    *nml_InputRequiredRank,
     decimal( 11, 2 )  *nmpd_OutputMoneyValue,
     SQLUDF_NULLIND    *nms_InputNullIndicator01,
     SQLUDF_NULLIND    *nms_InputNullIndicator02,
     SQLUDF_NULLIND    *nms_OutputNullIndicator01,
     SQLUDF_CHAR       sqludf_sqlstate[ SQLUDF_SQLSTATE_LEN + 1 ],
     SQLUDF_CHAR       sqludf_fname[ SQLUDF_FQNAME_LEN +1 ],
     SQLUDF_CHAR       sqludf_fspecname[ SQLUDF_SPECNAME_LEN + 1 ],
     SQLUDF_CHAR       sqludf_msgtext[ SQLUDF_MSGTEXT_LEN + 1 ],
     SQLUDF_SCRATCHPAD *sqludf_scratchpad,
     SQLUDF_CALL_TYPE  *sqludf_call_type )
{
     str_ScratchPad   *str_SPad;
     str_ScratchPad   **ptr_AlignmentPointer;
     decimal( 11, 2 ) nmpd_LowestValue, nmpd_Temp;
     int              nmi_Counter;
     long             nml_Temp;

/* Get the address of the data part of the scratchpad and align the */
/* pointer for the scratchpad to the 16 byte boundary */
     ptr_AlignmentPointer = ( ( str_ScratchPad ** )( sqludf_scratchpad ) ) + 1;
     str_SPad = ( str_ScratchPad * ) ptr_AlignmentPointer;

     if ( *sqludf_call_type == -1 )
     {
        if ( ( *nms_InputNullIndicator02 != 0 ) ||
             ( *nml_InputRequiredRank < 0 ) )
        {
           strcpy( sqludf_sqlstate, "38601" );
           strcpy( sqludf_msgtext, "Incorrect rank value specified" );
```

```
            *nms_OutputNullIndicator01 = -1;
            return;
         }

         str_SPad->nml_RequiredRating = *nml_InputRequiredRank;
         str_SPad->nml_ValuesStored   = 0;
         nml_Temp = *nml_InputRequiredRank * sizeof( decimal( 15, 5 ) );
         str_SPad->nmpd_LargeValue = ( decimal( 15, 5 ) * )
                                      malloc( *nml_InputRequiredRank *
                                              sizeof( decimal( 11, 2 ) ) );
      }


      if ( *sqludf_call_type == 1 )
      {
         free( str_SPad->nmpd_LargeValue );
      }


      if ( *sqludf_call_type < 1 )
      {
         if ( *nms_InputNullIndicator01 == 0 )
         {
            nmpd_LowestValue = *nmpd_InputMoneyValue;

            for ( nmi_Counter = 0;
                  nmi_Counter < str_SPad->nml_ValuesStored;
                  nmi_Counter++ )
            {
                if ( str_SPad->nmpd_LargeValue[ nmi_Counter ] <
                     nmpd_LowestValue )
                {
                   nmpd_Temp = nmpd_LowestValue;
                   nmpd_LowestValue = str_SPad->nmpd_LargeValue[nmi_Counter];
                   str_SPad->nmpd_LargeValue[ nmi_Counter ] = nmpd_Temp;
                }
            }

            if ( str_SPad->nml_ValuesStored < str_SPad->nml_RequiredRating )
            {
               str_SPad->nml_ValuesStored++;
               str_SPad->nmpd_LargeValue[str_SPad->nml_ValuesStored - 1]
               = nmpd_LowestValue;
            }
         }

         if ( str_SPad->nml_ValuesStored < str_SPad->nml_RequiredRating )
         {
            *nms_OutputNullIndicator01 = -1;
            return;
         }
         else
         {
            *nmpd_OutputMoneyValue =
            str_SPad->nmpd_LargeValue[ str_SPad->nml_RequiredRating - 1 ];
            *nms_OutputNullIndicator01 = 0;
            return;
         }
      }
}
```

A.7 RtvPrdNbr3: External stored procedure written in CLI

```
#define SQL_MAX_PWD_LENGTH 10
#define SQL_MAX_STM_LENGTH 255

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "sqlcli.h"

#define SQL_MAX_UID_LENGTH 10
SQLRETURN fun_Connect( void );
SQLRETURN fun_DisConnect( void );
SQLRETURN fun_ReleaseEnvHandle( void );
```

```
SQLRETURN fun_ReleaseDbcHandle( void );
SQLRETURN fun_ReleaseStmHandle( void );
SQLRETURN fun_Process( void );
void fun_PrintError( SQLHSTMT );

typedef struct
{
   unsigned long length;
   char data[ 1048576 ];
} BLOB1M;

SQLRETURN  nml_ReturnCode;
SQLHENV    nml_HandleToEnvironment;
SQLHDBC    nml_HandleToDatabaseConnection;
SQLHSTMT   nml_HandleToSqlStatement;
SQLINTEGER nmi_PcbValue;
SQLCHAR    chs_SqlStatement01[ SQL_MAX_STM_LENGTH + 1 ];
SQLCHAR    chs_ProductNumber[ 5 ];
BLOB1M     bin_ProductPicture;


void main( int argc, char **argv )
{
     SQLRETURN  nml_ConnectionStatus;
     char chs_OrderNumber[ 5 ];

     nml_ConnectionStatus = fun_Connect();
     if ( nml_ConnectionStatus == SQL_SUCCESS )
     {
        printf( "Connection Succeeded\n" );
     }
     else
     {
        printf( "Connection Failed\n" );
        exit( -1 );
     }

     memcpy( ( void * )&bin_ProductPicture, argv[ 1 ], 1048580 );
     nml_ConnectionStatus = fun_Process();

     if ( nml_ConnectionStatus == SQL_SUCCESS )
     {
        strncpy( argv[ 2 ], chs_ProductNumber, sizeof( chs_ProductNumber ) );
     }

     nml_ConnectionStatus = fun_DisConnect();
     if ( nml_ConnectionStatus == SQL_SUCCESS )
     {
        printf( "DisConnect Succeeded\n" );
        exit( 0 );
     }
     else
     {
        printf( "DisConnect Failed\n" );
        exit( -1 );
     }
}


SQLRETURN fun_Connect()
{
    SQLCHAR    chs_As400System[ SQL_MAX_DSN_LENGTH ];
    SQLCHAR    chs_UserName[ SQL_MAX_UID_LENGTH ];
    SQLCHAR    chs_UserPassword[ SQL_MAX_PWD_LENGTH ];

    printf( "Attempting to connect\n" );

    nml_ReturnCode = SQLAllocEnv( &nml_HandleToEnvironment );
    if ( nml_ReturnCode != SQL_SUCCESS )
    {
       printf( "Error allocating environment handle\n" );
       fun_PrintError( SQL_NULL_HSTMT );
       printf( "Terminating\n" );
       return SQL_ERROR;
    }

    printf( "Please enter the name of the As/400 system\n" );
    gets( chs_As400System );
```

```
    printf( "Please enter User Id for Log On\n" );
    gets( chs_UserName );
    printf( "Please enter password for Log On\n" );
    gets( chs_UserPassword );

    nml_ReturnCode = SQLAllocConnect( nml_HandleToEnvironment,
                                      &nml_HandleToDatabaseConnection );

    if ( nml_ReturnCode != SQL_SUCCESS )
    {
       printf( "Error allocating databse connection handle\n" );
       fun_PrintError( SQL_NULL_HSTMT );
       nml_ReturnCode = fun_ReleaseEnvHandle();
       printf( "Terminating\n" );
       return SQL_ERROR;
    }

    nml_ReturnCode = SQLConnect( nml_HandleToDatabaseConnection,
                                 chs_As400System,
                                 SQL_NTS,
                                 chs_UserName,
                                 SQL_NTS,
                                 chs_UserPassword,
                                 SQL_NTS );
    if ( nml_ReturnCode != SQL_SUCCESS )
    {
       printf( "Could not connect to system %s\n", chs_As400System );
       fun_PrintError( SQL_NULL_HSTMT );
       nml_ReturnCode = fun_ReleaseDbcHandle();
       nml_ReturnCode = fun_ReleaseEnvHandle();
       printf( "Terminating\n" );
       return SQL_ERROR;
    }
    else
    {
       return SQL_SUCCESS;
    }
}


SQLRETURN fun_Process()
{
    short Pictture_Ind = 0;

    printf( "Attempting to allocate handle to statement\n" );

    nml_ReturnCode = SQLAllocStmt( nml_HandleToDatabaseConnection,
                                   &nml_HandleToSqlStatement );
    if ( nml_ReturnCode != SQL_SUCCESS )
    {
       printf( "Could not allocate handle to statement\n" );
       fun_PrintError( SQL_NULL_HSTMT );
       printf( "Terminating\n" );
       return SQL_ERROR;
    }

    strcpy( chs_SqlStatement01, "select product_number " );
    strcat( chs_SqlStatement01, "from teamxx.prodmast01 " );
    strcat( chs_SqlStatement01, "where " );
    strcat( chs_SqlStatement01, "product_picture = " );
    strcat( chs_SqlStatement01, "cast( ? as TEAMXX.PICTURE)" );

    nml_ReturnCode = SQLPrepare( nml_HandleToSqlStatement,
                                 chs_SqlStatement01,
                                 SQL_NTS );
    if ( nml_ReturnCode != SQL_SUCCESS )
    {
       printf( "Could not prepare SQL statement\n" );
       fun_PrintError( nml_HandleToSqlStatement );
       nml_ReturnCode = fun_ReleaseStmHandle();
       printf( "Terminating\n" );
       return SQL_ERROR;
    }

    nmi_PcbValue = bin_ProductPicture.length;
    nml_ReturnCode = SQLBindParam( nml_HandleToSqlStatement,
                                   1,
                                   SQL_BLOB,
```

```
                                     SQL_BLOB,
                                     sizeof( bin_ProductPicture ),
                                     0,
                                     ( SQLPOINTER ) bin_ProductPicture.data,
                                     ( SQLINTEGER *) &nmi_PcbValue );
    if ( nml_ReturnCode != SQL_SUCCESS )
    {
       printf( "Could not bind SQL statement\n" );
       fun_PrintError( nml_HandleToSqlStatement );
       nml_ReturnCode = fun_ReleaseStmHandle();
       printf( "Terminating\n" );
       return SQL_ERROR;
    }

    nml_ReturnCode = SQLExecute( nml_HandleToSqlStatement );
    if ( nml_ReturnCode != SQL_SUCCESS )
    {
       printf( "Could not execute the SQL statement\n" );
       fun_PrintError( nml_HandleToSqlStatement );
       nml_ReturnCode = fun_ReleaseStmHandle();
       printf( "Terminating\n" );
       return SQL_ERROR;
    }

    nml_ReturnCode = SQLBindCol( nml_HandleToSqlStatement,
                                 1,
                                 SQL_CHAR,
                                 ( SQLPOINTER ) chs_ProductNumber,
                                 sizeof( chs_ProductNumber ),
                                 ( SQLINTEGER * ) &nmi_PcbValue );
    if ( nml_ReturnCode != SQL_SUCCESS )
    {
       printf( "Could not bind columns of the cursor\n" );
       fun_PrintError( nml_HandleToSqlStatement );
       nml_ReturnCode = fun_ReleaseStmHandle();
       printf( "Terminating\n" );
       return SQL_ERROR;
    }

    nml_ReturnCode = SQLFetch( nml_HandleToSqlStatement );
    if ( nml_ReturnCode != SQL_SUCCESS )
    {
       printf( "Could not fetch from the SQL cursor\n" );
       fun_PrintError( nml_HandleToSqlStatement );
       nml_ReturnCode = fun_ReleaseStmHandle();
       printf( "Terminating\n" );
       return SQL_ERROR;
    }
    else
    {
       return SQL_SUCCESS;
    }
}


SQLRETURN fun_DisConnect()
{
    printf( "Attempting to disconnect\n" );

    nml_ReturnCode = SQLDisconnect( nml_HandleToDatabaseConnection );
    if ( nml_ReturnCode != SQL_SUCCESS )
    {
       printf( "Failed to disconnect\n" );
       fun_PrintError( SQL_NULL_HSTMT );
       printf( "Terminating\n" );
       return 1;
    }
    else
    {
       printf( "Successfully disconnected\n" );
    }

    nml_ReturnCode = fun_ReleaseDbcHandle();
    nml_ReturnCode = fun_ReleaseEnvHandle();

    return nml_ReturnCode;
}
```

```
SQLRETURN fun_ReleaseEnvHandle()
{
     printf( "Attempting to release handle to environment\n" );
     nml_ReturnCode = SQLFreeEnv( nml_HandleToEnvironment );
     if ( nml_ReturnCode != SQL_SUCCESS )
     {
        printf( "Could not release handle to environment\n" );
        fun_PrintError( SQL_NULL_HSTMT );
        return SQL_ERROR;
     }
     else
     {
        printf( "Successfully released handle to environment\n" );
        return SQL_SUCCESS;
     }
}


SQLRETURN fun_ReleaseDbcHandle()
{
     printf( "Attempting to release handle to database connection\n" );
     nml_ReturnCode = SQLFreeConnect( nml_HandleToDatabaseConnection );
     if ( nml_ReturnCode != SQL_SUCCESS )
     {
        printf( "Could not release handle to database connection\n" );
        fun_PrintError( SQL_NULL_HSTMT );
        return SQL_ERROR;
     }
     else
     {
        printf( "Successfully released handle to database connection\n" );
        return SQL_SUCCESS;
     }
}


SQLRETURN fun_ReleaseStmHandle()
{
     printf( "Attempting to release handle to SQL statement\n" );

     nml_ReturnCode = SQLFreeStmt( nml_HandleToSqlStatement, SQL_CLOSE );
     if ( nml_ReturnCode != SQL_SUCCESS )
     {
        printf( "Could not release handle to SQL statement\n" );
        fun_PrintError( nml_HandleToSqlStatement );
        return SQL_ERROR;
     }
     else
     {
        printf( "Successfully released handle to SQL statement\n" );
        return SQL_SUCCESS;
     }
}


void fun_PrintError( SQLHSTMT nml_HandleToSqlStatement )
{
     SQLCHAR     chs_SqlState[ SQL_SQLSTATE_SIZE ];
     SQLINTEGER  nmi_NativeErrorCode;
     SQLCHAR     chs_ErrorMessageText[ SQL_MAX_MESSAGE_LENGTH + 1 ];
     SQLSMALLINT nmi_NumberOfBytes;

     nml_ReturnCode = SQLError( nml_HandleToEnvironment,
                                nml_HandleToDatabaseConnection,
                                nml_HandleToSqlStatement,
                                chs_SqlState,
                                &nmi_NativeErrorCode,
                                chs_ErrorMessageText,
                                sizeof( chs_ErrorMessageText ),
                                &nmi_NumberOfBytes );

     if ( nml_ReturnCode != SQL_SUCCESS )
     {
        printf( "Could not retrieve error information\n" );
        return;
     }
```

```
    printf( "SqlState - %s\n", chs_SqlState );
    printf( "SqlCode  - %d\n", nmi_NativeErrorCode );
    printf( "Error Message:\n" );
    printf( "%s\n", chs_ErrorMessageText );
}
```

Appendix B. Special notices

This publication is intended to help programmers, analysts, and database administrators to implement DB2 UDB for AS/400. The information in this publication is not intended as the specification of any programming interfaces that are provided by DB2 UDB for AS/400. See the PUBLICATIONS section of the IBM Programming Announcement for DB2 UDB for AS/400, for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The information about non-IBM ("vendor") products in this manual has been supplied by the vendor and IBM assumes no responsibility for its accuracy or completeness. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Any pointers in this publication to external Web sites are provided for convenience only and do not in any manner serve as an endorsement of these Web sites.

Any performance data contained in this document was determined in a controlled environment, and therefore, the results that may be obtained in other operating environments may vary significantly. Users of this document should verify the applicable data for their specific environment.

© Copyright IBM Corp. 2000

Reference to PTF numbers that have not been released through the normal distribution process does not imply general availability. The purpose of including these reference numbers is to alert IBM customers to specific information relative to the implementation of the PTF when it becomes available to each customer according to the normal IBM PTF distribution process.

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

AIX	AS/400
AT	C/400
COBOL/400	CT
DB2	DRDA
IBM®	Netfinity
Operating System/400	OS/2
OS/400	RPG/400
RS/6000	SP
SQL/400	System/390
XT	400

The following terms are trademarks of other companies:

Tivoli, Manage. Anything. Anywhere.,The Power To Manage., Anything. Anywhere.,TME, NetView, Cross-Site, Tivoli Ready, Tivoli Certified, Planet Tivoli, and Tivoli Enterprise are trademarks or registered trademarks of Tivoli Systems Inc., an IBM company, in the United States, other countries, or both. In Denmark, Tivoli is a trademark licensed from Kjøbenhavns Sommer - Tivoli A/S.

C-bus is a trademark of Corollary, Inc. in the United States and/or other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States and/or other countries.

PC Direct is a trademark of Ziff Communications Company in the United States and/or other countries and is used by IBM Corporation under license.

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States and/or other countries.

UNIX is a registered trademark in the United States and other countries licensed exclusively through The Open Group.

SET and the SET logo are trademarks owned by SET Secure Electronic Transaction LLC.

Other company, product, and service names may be trademarks or service marks of others.

Appendix C. Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

C.1 IBM Redbooks publications

For information on ordering these ITSO publications see "How to get IBM Redbooks" on page 233.

- *Building AS/400 Client/Server Applications with Java*, SG24-2152
- *DB2/400 Advanced Database Functions*, SG24-4249
- *DB2/400: Mastering Data Warehousing Functions*, SG24-5184
- *AS/400 Client Access Express for Windows: Implementing V4R4M0*, SG24-5191
- *Developing Cross-Platform DB2 Stored Procedures*, SG24-5485

C.2 IBM Redbooks collections

Redbooks are also available on the following CD-ROMs. Click the CD-ROMs button at http://www.redbooks.ibm.com/ for information about all the CD-ROMs offered, updates and formats.

CD-ROM Title	Collection Kit Number
System/390 Redbooks Collection	SK2T-2177
Networking and Systems Management Redbooks Collection	SK2T-6022
Transaction Processing and Data Management Redbooks Collection	SK2T-8038
Lotus Redbooks Collection	SK2T-8039
Tivoli Redbooks Collection	SK2T-8044
AS/400 Redbooks Collection	SK2T-2849
Netfinity Hardware and Software Redbooks Collection	SK2T-8046
RS/6000 Redbooks Collection (BkMgr Format)	SK2T-8040
RS/6000 Redbooks Collection (PDF Format)	SK2T-8043
Application Development Redbooks Collection	SK2T-8037
IBM Enterprise Storage and Systems Management Solutions	SK3T-3694

C.3 Other resources

These publications are also relevant as further information sources:

- *IBM DB2 Universal Database Application Development Guide*, SC09-2845
- *DB2 UDB for AS/400 SQL Programming*, SC41-5611
- *DB2 UDB for AS/400 SQL Reference*, SC41-5612
- White, Seth. *JDBC API Tutorial and Reference, Second Edition*. Addison-Wesley Publishing, Co., 1999 (ISBN: 0-2014332-81).

© Copyright IBM Corp. 2000

C.4 Referenced Web sites

These Web sites are also relevant as further information sources:

- Visit the IBM Redbooks home page at: `http://www.redbooks.ibm.com` for announcements about upcoming redbooks, redpieces, and full redbook downloads and ordering information.
- The reference tool *Net.Data Administration and Programming Guide for OS/400* is available for download from the Web at: `http://www.as400.ibm.com/products/netdata/docs/doc.htm`
- Modification 2 of the AS/400 Toolbox for Java is available for download from the Web at: `http://www.ibm.com/as400/toolbox`
- For information regarding interopability issues and problems between the different IBM DB2 DataLink managers, and for DB2 information in general, check the following Web site: `http://www.as400.ibm.com/db2/dlinkinter.htm`

How to get IBM Redbooks

This section explains how both customers and IBM employees can find out about IBM Redbooks, redpieces, and CD-ROMs. A form for ordering books and CD-ROMs by fax or e-mail is also provided.

- **Redbooks Web Site** http://www.redbooks.ibm.com/

  Search for, view, download, or order hardcopy/CD-ROM Redbooks from the Redbooks Web site. Also read redpieces and download additional materials (code samples or diskette/CD-ROM images) from this Redbooks site.

  Redpieces are Redbooks in progress; not all Redbooks become redpieces and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

- **E-mail Orders**

  Send orders by e-mail including information from the IBM Redbooks fax order form to:

	e-mail address
In United States	usib6fpl@ibmmail.com
Outside North America	Contact information is in the “How to Order” section at this site: http://www.elink.ibmlink.ibm.com/pbl/pbl

- **Telephone Orders**

United States (toll free)	1-800-879-2755
Canada (toll free)	1-800-IBM-4YOU
Outside North America	Country coordinator phone number is in the “How to Order” section at this site: http://www.elink.ibmlink.ibm.com/pbl/pbl

- **Fax Orders**

United States (toll free)	1-800-445-9269
Canada	1-403-267-4455
Outside North America	Fax phone number is in the “How to Order” section at this site: http://www.elink.ibmlink.ibm.com/pbl/pbl

This information was current at the time of publication, but is continually subject to change. The latest information may be found at the Redbooks Web site.

IBM Intranet for Employees

IBM employees may register for information on workshops, residencies, and Redbooks by accessing the IBM Intranet Web site at http://w3.itso.ibm.com/ and clicking the ITSO Mailing List button. Look in the Materials repository for workshops, presentations, papers, and Web pages developed and written by the ITSO technical professionals; click the Additional Materials button. Employees may access MyNews at http://w3.ibm.com/ for redbook, residency, and workshop announcements.

© Copyright IBM Corp. 2000

IBM Redbooks fax order form

Please send me the following:

Title	Order Number	Quantity

First name　Last name

Company

Address

City　Postal code　Country

Telephone number　Telefax number　VAT number

☐ Invoice to customer number

☐ Credit card number

Credit card expiration date　Card issued to　Signature

We accept American Express, Diners, Eurocard, Master Card, and Visa. Payment by credit card not available in all countries. Signature mandatory for credit card payment.

List of abbreviations

CPU Central Processing Unit

DBMS Database Management System

CLI Call Level Interface

DDL Data Definition Language

DDS Data Definition Specification

DML Data Manipulation Language

IBM International Business Machines Corporation

ILE Integrated Language Environment

ITSO International Technical Support Organization

LOB Large Object

I/O Input/Output

ODBC Open Database Connectivity

OLAP On-line Analytical Processing

OLTP On-line Transaction Processing

RRA Relative Record Address

RDBMS Relational Database Management System

SEU Screen Edit Utility

SLIC System License Internal Code

SQL Structured Query Language

UDF User Defined Function

UDT User Defined Type

© Copyright IBM Corp. 2000

Index

A

activation group, *CALLER 101
AS LOCATOR clause 143
AS/400 Toolbox for Java 129

B

bind, external UDFs 101
BLOB 5
 using in Java 130
Blob
 code example in Java 130
 parameter marker 133
Blob interface 130
Blob object
 creating and materializing 130
 storing in the database 132

C

Call Level Interface 139
casting functions 28
CL command
 ADDHDBDLFM 159
 ADDPFXDLFM 157
 CRTCMOD 101
 CRTSRVPGM 101
 DSPFD 203
 DSPFFD 60, 177
 EDTDLFA 207
 ENDTCPSVR 163
 INZDLFM 156
 STRTCPSVR 163
 WRKPFDL 204
 WRKRDBDIRE 155
CLASSPATH variable 130
CLI 139
 code example 139
 compile and bind 142
 retrieving LOBs 143
 stored procedure 139
CLOB 5
Clob object
 code example 135
 creating and materializing 134
 storing in the database 136
 Unicode 135
 using 134
column function 78
commitment control 12, 18
compile
 CLI program 142
 external UDFs 101
 Java program 132
complex objects 1, 129
control token 165, 186, 190
 code example 186
CREATE FUNCTION statement 78
 FINAL CALL clause 107
 SCRATCHPAD clause 104
 STATIC DISPATCH clause 90
CREATE PROCEDURE statement 142

D

Data Links Filesystem Filter (DLFF) 213
data type precedence list 40
data type promotion 40
Datalink APIs 153
DataLink File Manager 151
Datalink Filter 153
DataLinks
 architecture 150
 attributes 165
 code examples 186
 configuration 153, 154
 considerations 174
 control token 165, 186
  code example 186
 definition 147
 delete ON UNLINK 166
 in dynamic web pages 183
 journal function 176
 link control 165
 link pending 205
 ON UNLINK DELETE explained 173
 read permission 165
 reconcile pending link 206
 save and restore 202
 scalar function DLVALUE 178
 SQL examples 175
 SQL scalar functions 182
 using as column type 168
 write permission 166
DataLinks File Manager (DLFM) 213
DB2 (Logging Manager) 213
DB2SQL parameter style 96
 coding example 104
DBCLOB 5
debugging 119
display pictures 130
DLFM 151
 adding host database 159
 adding prefix 157
 initialize 156
 on remote system 152
DLFM server job 156
 checking the status 163
 starting 163
 stopping 163
DLURLCOMPLETE scalar function 182
DLURLPATH scalar function 183
DLURLPATHONLY scalar function 183
DLURLSERVER scalar function 183
DLVALUE scalar function 178
 overloading example 179

© Copyright IBM Corp. 2000

dropping UDF 118

E
error handling in UDF 96
explicit casting 35, 141
  of parameter markers 131
external UDF 71, 95
  error handling 101
  null indicator 100
  parameter styles 95

F
FINAL CALL clause 107
full file path 179
function overloading 72, 179
  code example 110
function parameters 90
function path 72
function resolution 71
  code example 108
function selection algorithm 76
function signature 72

H
HTTP 184

I
IFS 149
  file permission 172
  mount points 158
implicit casting 38, 141
  host variables 42
  in Java 131
input stream in Java 136
input stream using in Java 133
Integrated File System 149

J
Java 2 platfrom 129
Java input stream 133, 136
JDBC 2.0 129
JDK 1.2.2 130
join 45

L
link pending status 205
LOB
  commitment control 18
  comparing 19
  compatibility of data types 8
  definition 4
  maximum size 6
  native interface 22
  passing as a parameter 143
  supported built-in functions 21
  triggers 25
  using as column type 7
  using in CLI 139
LOB file reference
  CCSID conversion 14
  definition 12
  file options 14
LOB locator
  commitment control 12
  declaring 9
  definition 8
LOBs
  using in Net.Data 25

M
Metadata
  using in Java 137
metadata
  code example in Java 137
  retrieving column information 137

N
native I/O 202
native interface 22, 49, 193
Net.Data 25

O
Operations Navigator 2, 28, 85, 150, 167
  Run SQL Scripts utility 7

P
parameter markers
  BLOB 141
  Blob 133
  casting 131
parameter matching 74
parameter promotion 74, 112, 113
parameter styles in external UDFs 95
pending link reconcilation 206
picture, load into database 133
precedence of data types 76

R
RDB entry 155
RPG code example 202
Run SQL Scripts utility 78

S
save and restore for UDTs 60
scalar function 78
scratchpad 104
  code example 105
secondary thread 119
SET PATH statement 73
sourced UDF 70, 78
  arithmetic operators 83
  code example 79, 82, 84

column functions 82
scalar function 78
SQL naming convention 72
SQL parameter style 95
coding example 99
SQL UDF 70, 85
code example 87
STATIC DISPATCH clause 90
strong typing 34
Swing GUI 130
system catalog
SYSCOLUMNS 58
SYSPARMS 117
SYSROUTINES 113, 116
SYSTYPES 57
system naming convention 72

T

Toolbox for Java 129
triggers 25

U

UDF
code example 92
compile and bind 101
debugging 119
definition 69
dropping 118
error handling 96
external 71, 95
function overloading 72
function path 72
function signature 72
LOB parameters 90
LOB return value 91
resolving 71
return value 91
save and restore 119
sourced 70, 78
SQL 70, 85
UDF_TIME_OUT parameter 127
UDT parameters 90
UDT return value 91
UDT
casting functions 28
changing definition 34
comparing 44
creating 28
data type promotion 40
definition 27
dropping 64
explicit casting 35
implicit casting 38
joining on UDT columns 45
native I/O 49, 54
native interface 49
save and restore 60, 62
strong typing 34
system catalog 57
using as column type 31, 33
using in CLI 142
using in Java 137
Unicode 135

IBM Redbooks evaluation

DB2 UDB for AS/400 Object Relational Support
SG24-5409-00

Your feedback is very important to help us maintain the quality of IBM Redbooks. **Please complete this questionnaire and return it using one of the following methods:**

- Use the online evaluation form found at http://www.redbooks.ibm.com/
- Fax this form to: USA International Access Code + 1 914 432 8264
- Send your comments in an Internet note to redbook@us.ibm.com

Which of the following best describes you?
_ **Customer** _ **Business Partner** _ **Solution Developer** _ **IBM employee**
_ **None of the above**

Please rate your overall satisfaction with this book using the scale:
(1 = very good, 2 = good, 3 = average, 4 = poor, 5 = very poor)

Overall Satisfaction __________

Please answer the following questions:

Was this redbook published in time for your needs? Yes___ No___

If no, please explain:

What other Redbooks would you like to see published?

Comments/Suggestions: (THANK YOU FOR YOUR FEEDBACK!)

© Copyright IBM Corp. 2000

SG24-5409-00

Printed in the U.S.A.

DB2 UDB for AS/400 Object Relational Support