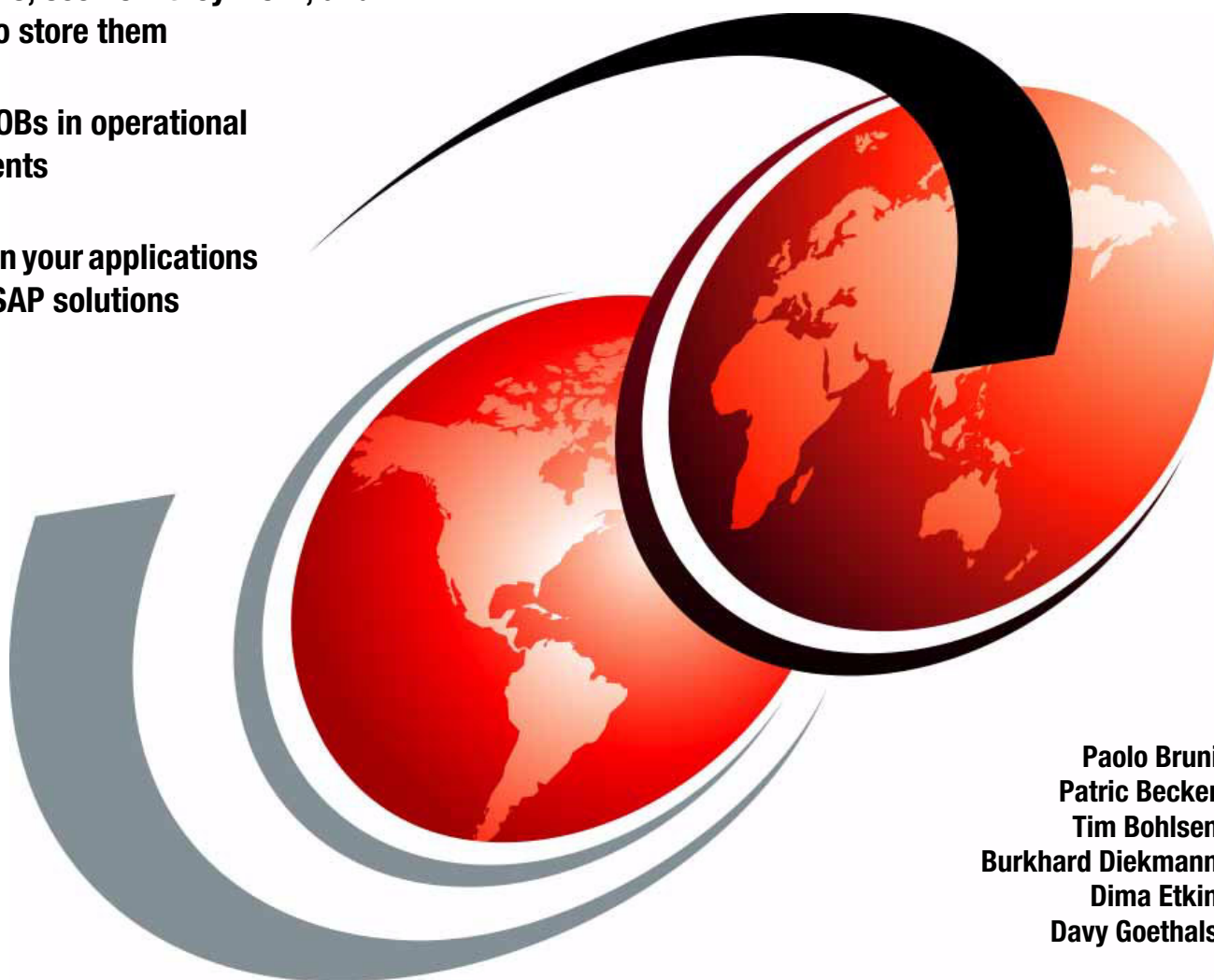


LOBs with DB2 for z/OS: Stronger and Faster

Define LOBs, see how they work, and
see how to store them

Manage LOBs in operational
environments

Use LOBs in your applications
and with SAP solutions



Paolo Bruni
Patric Becker
Tim Bohlsen
Burkhard Diekmann
Dima Etkin
Davy Goethals

Redbooks



International Technical Support Organization

LOBs with DB2 for z/OS: Stronger and Faster

November 2006

Note: Before using this information and the product it supports, read the information in “Notices” on page xv.

First Edition (November 2006)

This edition applies to IBM DB2 for z/OS Version 8 (program number 5625-DB2) and IBM DB2 Version 9.1 for z/OS (program number 5635-DB2).

Note: This book is based on a pre-GA version of DB2 Version 9.1 for z/OS and may not apply when the product becomes generally available. We recommend that you consult the product documentation or follow-on versions of this IBM Redbook for more current information.

Contents

Figures	vii
Tables	ix
Examples	xi
Notices	xv
Trademarks	xvi
Preface	xvii
The team that wrote this IBM Redbook	xvii
Become a published author	xx
Comments welcome	xx
Chapter 1. Introduction	1
1.1 Object-relational in DB2 for z/OS	2
1.2 Changes with DB2 9	3
1.3 DB2 for z/OS and large objects	4
1.4 The IBM Redbook contents	4
1.5 Pointers to LOB functions after DB2 Version 6	4
Chapter 2. Large objects with DB2	9
2.1 Introduction to LOB data types	10
2.2 The LOB table spaces	12
2.2.1 Single LOB column table space	12
2.2.2 Multiple LOB column table space	12
2.2.3 Partitioned LOB table space	13
2.2.4 The full LOB implementation structure	14
2.3 LOB locators	16
2.3.1 Purpose of LOB locators	16
2.3.2 Different types of LOB locators	17
2.4 LOB file reference variables	18
2.4.1 DB2-generated file reference variable constructs	19
2.4.2 Language support for LOB file reference variables	19
2.4.3 File local/client support	21
Chapter 3. Creating LOBs	23
3.1 Alternatives in defining LOBs	24
3.1.1 Example of automatic creation of objects	24
3.1.2 Using CURRENT RULES STD	29
3.1.3 Manual creation of objects	31
3.1.4 Adding a LOB column to an existing table	37
3.2 Defining ROWIDs	39
3.2.1 Creating the ROWID column	40
3.3 LOBs and LOG activity	44
3.3.1 LOGGED and NOT LOGGED attributes	44
3.3.2 Logging for all LOB sizes	50
3.4 Additional considerations for creating LOB objects	52
3.4.1 Data conversion	52
3.4.2 Buffer pools and LOB table spaces	56

3.4.3	Locking with LOBs	56
3.4.4	Buffer pool and page size considerations	57
3.4.5	DSSIZE for LOB table spaces	58
3.4.6	GBPCACHE parameter	59
3.4.7	Impact on cursors fetching LOB values	60
3.5	LOBs are different DB2 objects	60
3.6	Physical layout of LOBs	62
Chapter 4.	Using LOBs	65
4.1	Language considerations	66
4.1.1	LOB host variables, locators, and file reference variables	66
4.1.2	Use of a double or triple SQLDA in dynamic SQL	67
4.1.3	Working with LOBs in JDBC and SQLJ applications	68
4.1.4	Specific SQL support for LOBs	69
4.1.5	Functions such as XML2CLOB	72
4.1.6	Stored procedures	73
4.2	LOB locators	73
4.2.1	Getting to know LOB locators	74
4.2.2	Examples of using locators	78
4.3	DRDA LOB flow optimization	79
4.3.1	DB2 Universal Java Driver	81
4.4	Feeding a LOB column	83
4.4.1	Loading a LOB column using LOAD or the cross loader	83
4.4.2	Inserting LOBs using the host application	85
4.4.3	DB2 for Linux, UNIX and Windows import	93
4.5	Locking	94
4.5.1	Locking for LOBs with DB2 V8	95
4.5.2	Locking for LOBs with DB2 9	101
4.6	Unloading LOBs	107
4.6.1	Unloading a LOB using an application	107
4.6.2	Using FETCH CONTINUE	113
4.6.3	Finding the nth occurrence of a string	119
4.7	Updating LOBs	120
4.7.1	Deleting a specific part of a LOB	120
4.7.2	Updating a specific part of a LOB	121
4.7.3	Updating the entire LOB value	122
4.8	General best practices	122
Chapter 5.	SAP usage of LOBs	125
5.1	Overview of SAP usage of LOBs	126
5.1.1	Some history of SAP LOB usage	126
5.1.2	Basic architecture	126
5.1.3	Connectivity	127
5.1.4	Why use LOBs	127
5.1.5	SAP usage of LOBs in terms of number and size	128
5.2	ABAP and Dynpro source and Load	129
5.3	Programming techniques for the ABAP stack	133
5.3.1	Basic locator access	133
5.3.2	CLI Streaming Interface	134
5.4	Optimization techniques and query rewrite	136
5.4.1	Local LOB buffer	136
5.4.2	Retrieve length and maximal data with locator	137
5.4.3	Optimizing the free locator statement	137

5.4.4	Comparison of different techniques using SGEN	138
5.4.5	Chaining	138
5.5	Programming techniques with JDBC	140
5.6	Data Dictionary considerations	143
5.6.1	ABAP stack	143
5.6.2	Java stack	144
5.6.3	DSNZPARMs for DB2 V8	144
5.6.4	ROWID	144
5.7	Unicode	147
5.8	Some points of SAP LOB usage with CCMS	148
5.9	Portability aspects	149
5.10	Monitoring and tracing	150
5.11	Database interface layer profile parameters	154
5.12	Performance measurements	155
5.12.1	Locks and SELECT	155
5.12.2	Locks and INSERT	157
5.12.3	UPDATE improvement	157
Chapter 6.	Utilities with LOBs	159
6.1	UNLOAD	160
6.2	DSNTIAUL	169
6.3	LOAD	171
6.3.1	Loading LOB data as normal data columns	171
6.3.2	Loading LOB data using file reference variables	172
6.3.3	Using the cross loader	172
6.3.4	Impact of logging	173
6.4	COPY	177
6.5	COPYTOCOPY	180
6.6	QUIESCE	181
6.7	REPORT	182
6.8	RUNSTATS	183
6.9	REORG	185
6.10	RECOVER and REBUILD	195
6.11	CHECK DATA	199
6.12	CHECK LOB	204
6.13	CHECK INDEX	207
6.14	REPAIR	208
6.15	DSN1COPY and DSN1PRNT	210
Chapter 7.	Data administration with LOBs	211
7.1	LOBs in the DB2 catalog	212
7.1.1	Catalog definitions for LOBs	212
7.1.2	LOBs defined in DB2 catalog	217
7.1.3	Real Time Statistics	218
7.2	Recovery strategies and considerations	219
7.2.1	LOGGED base table space with LOGGED LOB table space	219
7.2.2	LOGGED base table space with NOT LOGGED LOB table space	230
7.2.3	NOT LOGGED base table space with NOT LOGGED LOB table space	232
7.2.4	LOBs and SYSTEM RECOVERY	233
7.2.5	Conclusions on recovery of LOB data	234
7.3	Altering tables containing LOB columns	235
Chapter 8.	Performance with LOBs	237
8.1	LOB materialization	238

8.1.1 The different cases of materialization	239
8.1.2 Materialization avoidance techniques	241
8.2 Virtual storage management for LOBs	242
8.2.1 DB2 subsystem parameters for LOBs.	243
8.3 Buffer pools and group buffer pools	244
8.3.1 Virtual buffer pools	244
8.3.2 Considerations for a data sharing environment	246
8.4 Logging with LOBs	248
8.5 Accessing LOBs	248
8.5.1 Reading LOBs.	248
8.5.2 Writing LOBs.	248
8.6 Comparing SQL accounting profiles	249
8.7 Important I/O aspects	250
8.8 IFCID enhancements for LOBs	251
8.9 DRDA LOB flow optimization performance	255
8.10 LOB recommendations for performance	256
Appendix A. Additional material	259
Locating the Web material	259
Using the Web material	259
System requirements for downloading the Web material	261
How to use the Web material	261
Related publications	263
IBM Redbooks	263
Other publications	263
Online resources	264
How to get IBM Redbooks	265
Help from IBM	265
Abbreviations and acronyms	267
Index	271

Figures

2-1 Association between base table and auxiliary table	12
2-2 Base table with two LOB columns and the two associated LOB table spaces	13
2-3 Partitioned base table containing two LOB columns.	14
2-4 Association between base table, ROWID, auxiliary table, and LOB table space	15
2-5 Assigning a LOB locator for a LOB value	17
2-6 DSNTIPE installation panel.	22
3-1 Catalog description for table BOOK_BASE_TABLE	33
3-2 SYSIBM.SYSCOLUMNS contents for TBNAME= BOOK_AUX_TABLE	36
3-3 Index keys for an auxiliary index.	37
3-4 LOB structure	39
3-5 Applying changes to a LOB table space created with NOT LOGGED	49
3-6 Mixed client/server environment with different data types	53
3-7 Differences between code pages 37 and 500.	54
3-8 LOB value spanned over pages using chunks and non-chunks.	62
3-9 LOB data pages chunked together using a space map and LOB map pages	63
4-1 Concurrent LOB access using LOB locators	75
4-2 Primary chain of locators containing secondary chains	77
4-3 Progressive reference return of LOB data	81
4-4 An example of LOAD with file reference variables	84
4-5 Secondary chain of locators	91
4-6 Primary chain of locators containing secondary chains	91
4-7 Lock escalation on LOBs	95
4-8 SELECT lock sequence	96
4-9 SELECT lock sequence using uncommitted read.	97
4-10 Insert lock sequence	98
4-11 Delete lock sequence	98
4-12 Locks acquired by mass delete.	99
4-13 Lock sequence when updating a LOB column	100
4-14 Lock escalation for UR readers.	102
4-15 Accessing a LOB without a locator reference using ISOLATION (CS).	111
4-16 Processing a LOB using a locator reference	112
5-1 Overview of SAP Web AS 6.40 on DB2 for z/OS (c) SAP AG; 2006	127
5-2 Total space usage for SAP MCOD system (c) SAP AG; 2006	129
5-3 REPOLOAD structure (c) SAP AG; 2006	130
5-4 REPOSRC structure (c) SAP AG; 2006	131
5-5 Transaction SGEN (c) SAP AG; 2006	132
5-6 SQL Trace for simple select using locators (c) SAP AG; 2006.	133
5-7 SQL Trace for update using locator (c) SAP AG; 2006	134
5-8 SELECT statement with LOB locators (c) SAP AG; 2006.	134
5-9 Modified statement (extended DA) on REPOLOAD (c) SAP AG; 2006	137
5-10 Starting the dbsl trace (c) SAP AG; 2006	151
5-11 SAP Performance Optimizer Tool (c) SAP AG; 2006	153
5-12 SAP Performance Optimizer Tool DB Trace (c) SAP AG; 2006.	153
5-13 SAP R3load test case elapsed time (c) SAP AG; 2006	156
5-14 R3load test case reduced locks (c) SAP AG; 2006	156
5-15 R3load test case reduced network round-trips (c) SAP AG; 2006	157
5-16 Time to update a row when increasing the numbers of updated rows (c) SAP AG; 2006	

6-1	Heavily updated LOB table space.	187
6-2	LOB Table space REORG with DB2 V8 and prior	188
6-3	LOB Table space REORG with DB2 9	188
6-4	Fragmented LOB table space	192
6-5	Non-fragmented LOB table space	193
8-1	DB2 materialization overview illustration.	239
8-2	LOB Materialization In user address space in case of data retrieval	240
8-3	LOB materialization with INSERT	241
8-4	DSNTIPD installation panel	243
8-5	Buffer pool strategies	245
8-6	Separation of LOB buffer pools.	246
8-7	LOB group buffer pool.	247
8-8	Accounting trace report for LOB and VARCHAR applications	250
8-9	Striped versus non-striped LOB table spaces.	251
8-10	Performance of LOB Progressive Streaming	255

Tables

1-1	The LOBs functions that have been introduced after Version 6	5
2-1	Typical average size for large objects	10
2-2	Maximum number of LOB columns by partitions	16
2-3	DB2-generated construct	19
2-4	File option constants	20
3-1	Attributes for implicit database creation	27
3-2	Default values for implicitly created table spaces	27
3-3	Base table space created using automatic object creation	28
3-4	LOB table space created using automatic object creation	28
3-5	Auxiliary index created using automatic object creation	29
3-6	LOB table space created using CURRENT RULES STD	30
3-7	Auxiliary index created using CURRENT RULES STD	30
3-8	LOG column values scenarios	47
3-9	SYSIBM.SYSCOPY values for LOGGED attribute changes	48
3-10	DB2 Unicode support - Additional useful resources	55
3-11	Choosing a LOB page size that minimizes getpages	57
3-12	Choosing a LOB page size for LOBs that are all the same size	58
3-13	Primary and secondary quantity with LOBs	58
3-14	Summary of data set, partition, and partitioned table space sizes	59
4-1	SQLTYPE and SQLLEN of LOB columns or LOB host variables in the SQLDA	68
4-2	Casting large objects	72
4-3	Where large objects come from and how	83
5-1	LOB objects in one SAP system (c) SAP AG; 2006	128
5-2	Total size of all LOB columns (c) SAP AG; 2006	129
5-3	Details of Repo tables' LOB content (c) SAP AG; 2006	132
5-4	SGEN runs on different optimization levels (c) SAP AG; 2006	138
5-5	DSNZPARMs for V8 (c) SAP AG; 2006	144
5-6	Comparison of different databases for aspects of LOBs (c) SAP AG; 2006	150
5-7	dbsl_lib profile parameters for LOB handling (c) SAP AG; 2006	155
6-1	Impact of logging if base table space is LOGGED	173
6-2	Impact of logging if base table space is LOGGED	191
6-3	Resetting pending states using REPAIR	209
8-1	LOB linked subsystem parameters	243
8-2	Current IFCIDs providing information regarding LOBs	252

Examples

2-1 Host variable definitions for LOB locators in COBOL	17
2-2 What the DB2 precompiler makes of LOB locators.	18
2-3 Host variable definition for BLOB file reference variable in COBOL	20
2-4 What the DB2 precompiler makes of BLOB file reference variable	20
3-1 DDL for a base table resulting in automatic object creation	25
3-2 Resulting objects for automatic object creation	25
3-3 DDL for a base table for manual object creation.	31
3-4 Catalog description for a hidden ROWID for LOBs.	32
3-5 Retrieving a generated ROWID value at INSERT time.	33
3-6 Content of LOB indicator columns	34
3-7 DDL for a LOB table space	34
3-8 DDL for an auxiliary table	35
3-9 DDL for an auxiliary table containing data of one base table partition	35
3-10 DDL for an auxiliary Index.	36
3-11 Displaying a database for LOBs	37
3-12 Adding a ROWID column	38
3-13 Adding a LOB column	38
3-14 DDL for a table containing a ROWID column	41
3-15 Inserting a row in CUSTOMER table	41
3-16 ROWID value of a table created with ROWID column	41
3-17 DB2 9 DSN1PRNT of ROWID in hex value	41
3-18 DB2 V8 DSN1PRNT of ROWID in hex value	42
3-19 ALTER TABLE adding a ROWID column	42
3-20 ROWID value of a table where a ROWID column was added	42
3-21 DB2 9 DSN1PRNT of ROWID in hex value after adding and updating a row.	43
3-22 DB2 V8 DSN1PRNT of ROWID in hex value after adding and updating a row	43
3-23 ROWID values of updated and inserted columns	43
3-24 DB2 9 DSN1PRNT of updated and inserted ROWIDs in hex value	43
3-25 DB2 V8 DSN1PRNT of updated and inserted ROWIDs in hex value	44
3-26 DISPLAY DATABASE sample output	47
3-27 DSN1LOGP output for logged LOB insert	50
3-28 DSN1LOGP output for not logged LOB insert	51
4-1 Usage example of XML2CLOB function	72
4-2 Example output from XML2CLOB function	73
4-3 Syntax for FREE locator and HOLD locator	76
4-4 Loading LOB data using File Reference Variable.	85
4-5 Rows tied together in one variable	87
4-6 Host variable declaration for a LOB column	87
4-7 What the DB2 precompiler makes of your host variable declaration	87
4-8 Inserting a single LOB value using one host variable	87
4-9 Pseudo code inserting LOBs with one locator chain.	88
4-10 Pseudo code inserting LOBs with multiple locator chains.	92
4-11 IMPORT command	93
4-12 EXPORT command	94
4-13 Comparison of RLSN and LSN to reclaim space	104
4-14 Unloading LOB data using a file reference variable	107
4-15 What the DB2 precompiler generates for LOB files	108
4-16 Unloading LOB data using one host variable	109

4-17	Unloading a LOB using locators	110
4-18	FETCH CONTINUE with dynamic SQL	115
4-19	FETCH CONTINUE with static SQL	118
4-20	Finding a specific occurrence of a string	119
4-21	Delete Chapter 8 of book CLOB using locators	120
4-22	Updating a part of a CLOB	121
4-23	Inserting new text at a particular position	122
5-1	CLI trace for REPOLOAD Select using SQLGetSubString() (c) SAP AG; 2006	135
5-2	Pseudo code for chaining (c) SAP AG; 2006	138
5-3	Chaining on insert (c) SAP AG; 2006	139
5-4	R3load log files for dbs_db2_chaining=20 (c) SAP AG; 2006	139
5-5	R3load log files for dbs_db2_chaining=0 (c) SAP AG; 2006	139
5-6	LOB access with JDBC driver (c) SAP AG; 2006	140
5-7	Insert using progressive reference (c) SAP AG; 2006	142
5-8	Table definition for test table TSTLOBDDL in XML format (c) SAP AG; 2006	145
5-9	DDL for LOB table DB2 V8 (c) SAP AG; 2006	146
5-10	DDL for LOB table DB2 9 (c) SAP AG; 2006	146
5-11	Bind command for UNIX system (c) SAP AG; 2006	147
5-12	DBSL trace displaying all LOB data (c) SAP AG; 2006	151
5-13	Locks taken by simple SELECT on LOB table V8 (c) SAP AG; 2006	154
5-14	Locks taken by simple SELECT on LOB table with DB2 9 (c) SAP AG; 2006	154
6-1	DDL for table ##T.NORMEN00	160
6-2	Unload LOB data as normal data columns with UNLOAD TABLESPACE	161
6-3	UNLOAD exceeding a 32 KB row size	162
6-4	Unload LOB data as normal data columns with UNLOAD TABLE	162
6-5	Unload LOB data as normal data columns in DELIMITED format	162
6-6	Unload LOB data as normal data columns in DELIMITED format	163
6-7	Unload LOB data as normal data columns in truncated format	163
6-8	Unload LOB data as normal data columns in truncated format	163
6-9	Unload LOB data to a PDS	165
6-10	Unload LOB data to a PDS	165
6-11	Contents of the SYSREC file	166
6-12	Unload LOB data to a PDSE	166
6-13	Contents of the SYSREC file	167
6-14	Unload LOB data to a HFS directory	167
6-15	Contents of the HFS directory /u/DB9B	167
6-16	Contents of the SYSREC file	168
6-17	DSNTIAUL with SQL parameter	169
6-18	DSNTIAUL output with SQL parameter	169
6-19	DSNTIAUL with LOBFILE parameter	170
6-20	DSNTIAUL output with LOBFILE parameter	171
6-21	LOAD LOB data with input from DSNTIAUL	174
6-22	LOAD LOB data with input from DSNTIAUL	174
6-23	Entries in SYSIBM.SYSTABLESPACE	175
6-24	LOAD with LOG YES on NOT LOGGED table spaces	175
6-25	LOAD LOB input file not found	176
6-26	LOAD LOB data with cross loader	176
6-27	COPY LOB data	179
6-28	Common START_RBA in SYSIBM.SYSCOPY	179
6-29	CONCURRENT COPY of LOB data	180
6-30	CONCURRENT COPY entries in SYSIBM.SYSCOPY	180
6-31	CONCURRENT COPY of LOB data to one dump data set	180
6-32	CONCURRENT COPY entries in SYSIBM.SYSCOPY with FILTERDDN	180

6-33 COPYTOCOPY	181
6-34 Primary and COPYTOCOPY entries in SYSIBM.SYSCOPY	181
6-35 QUIESCE a base table space and all LOB table spaces	182
6-36 QUIESCE entries in SYSIBM.SYSCOPY	182
6-37 QUIESCE a table space set	182
6-38 REPORT TABLESPACESET on automatic created objects.	183
6-39 REPORT TABLESPACESET report.	183
6-40 REPORT RECOVERY	183
6-41 RUNSTATS on LOB data	184
6-42 REORG SHRLEVEL NONE of one LOB table space	186
6-43 REORG SHRLEVEL NONE of all LOB objects.	186
6-44 REORG SHRLEVEL REFERENCE	189
6-45 REORG SHRLEVEL REFERENCE output.	189
6-46 REORG SHRLEVEL REFERENCE with LISTDEF.	190
6-47 REORG INDEX auxiliary index.	190
6-48 RTS DSNACCOX query for REORG TABLESPACE	194
6-49 RTS DSNACCOX query for REORG INDEX	194
6-50 Recover table spaces and indexes	198
6-51 Recover table spaces and rebuild indexes	198
6-52 REBUILD SHRLEVEL CHANGE of the indexes	199
6-53 Using RECOVER to reallocate a single VSAM cluster of a LOB table space.	199
6-54 Examples of CHECK DATA	203
6-55 Examples of CHECK LOB	206
6-56 Example of CHECK INDEX using a LISTDEF	208
6-57 DUMP or DELETE an entire LOB value	209
7-1 DDL for table ##T.NORMEN00.	212
7-2 Select from SYSIBM.SYSAUXRELS	213
7-3 Select from SYSIBM.SYSCOLUMNS	214
7-4 Select from SYSIBM.SYSLOBSTATS	215
7-5 Select from SYSIBM.SYSTABLEPART	215
7-6 Select from SYSIBM.SYSTABLES	215
7-7 Select from SYSIBM.SYSTABLESPACE	217
7-8 DB2 9, automatic creation of objects	217
7-9 Select from SYSIBM.SYSTABLESPACE	217
7-10 DB2 catalog query	217
7-11 Creating a common recoverable point of consistency using COPY	219
7-12 Common START_RBA in SYSIBM.SYSCOPY.	219
7-13 SPUFI delete.	220
7-14 Delete VSAM clusters	220
7-15 RECOVER to current point in time	220
7-16 RECOVER to current point in time	220
7-17 Display database command	221
7-18 Creating a common recoverable point of consistency using QUIESCE	222
7-19 SPUFI delete.	222
7-20 Point in time recovery to a common recoverable point of consistency	222
7-21 Point in time recovery to a recoverable quiesce point	222
7-22 Point in time recovery with consistency in DB2 9	224
7-23 SPUFI delete.	225
7-24 Point in time recovery of base table only	225
7-25 Base table space in ACHKP	225
7-26 CHECK DATA SHRLEVEL REFERENCE AUXERROR REPORT.	226
7-27 Result of CHECK DATA AUXERROR REPORT	226
7-28 CHECK DATA SHRLEVEL CHANGE AUXERROR REPORT	226

7-29	CHECK DATA AUXERROR INVALIDATE	227
7-30	Point in time recovery of the LOB table space	227
7-31	State of the table spaces afterwards.	227
7-32	State of the table spaces after CHECK DATA	228
7-33	SPUFI delete.	228
7-34	Point in time recovery of LOB table space only	229
7-35	CHECK DATA AUXERROR INVALIDATE SHRLEVEL REFERENCE.....	229
7-36	Identification and Invalidation of orphan LOBs	229
7-37	Use of REPAIR to delete orphan LOBs	230
7-38	SPUFI Insert	230
7-39	LOB table space in AUXW	231
7-40	CHECK LOB output	231
7-41	SPUFI update invalid LOB	232
7-42	RECOVER of NOT LOGGED objects.	232
8-1	Omegamon XE for DB2 Performance Expert output	242
8-2	LOB buffer pool allocation size	246

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described 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:

IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.


This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

Redbooks (logo)  ™	DFSMSdss™	Language Environment®
ibm.com®	DFSMSHsm™	MVS™
iSeries™	DRDA®	OS/390®
z/OS®	Enterprise Storage Server®	Parallel Sysplex®
zSeries®	ESCON®	PR/SM™
z9™	FlashCopy®	QMFT™
AIX®	FICON®	Redbooks™
CICS®	Geographically Dispersed Parallel Sysplex™	RACF®
Database 2™	GDPS®	System/390®
DB2 Connect™	IBM®	Tivoli®
DB2 Extenders™	IBM WebSphere® MQ	WebSphere®
DB2 Universal Database™	IMS™	
DB2®		

The following terms are trademarks of other companies:

SAP, R/3, mySAP, mySAP.com, xApps, xApp, SAP NetWeaver, and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP AG in Germany and in several other countries all over the world.

EJB, Java, Java Naming and Directory Interface, JDBC, JVM, J2EE, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Visual Studio, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel, Intel logo, Intel Inside logo, and Intel Centrino logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others. Data contained in this document serves informational purposes only. National product specifications may vary

Preface

The requirements for a database management system (DBMS) have included support for very large and complex data objects.

DB2® UDB for OS/390® Version 6 introduced the support for large objects (LOBs): they can contain text documents, images, or movies, and can be stored directly in the DBMS with sizes up to 2 gigabytes per object and 65,536 TB for a single LOB column in a 4,096 partition table. The introduction of these new data types has implied some changes in the administration processes and programming techniques. The Redbook *Large Objects with DB2 for z/OS and OS/390*, SG24-6571, introduced and described the usage of LOBs with DB2 for z/OS® at Version 7 level.

Major enhancements for LOB manipulation have been introduced with DB2 UDB for z/OS Version 8 and DB2 Version 9.1 for z/OS (DB2 9 in this IBM Redbook). These enhancements include performance functions such as the avoidance of LOB locks and DRDA® LOB flow optimization, usability functions such as file reference variables, FETCH CONTINUE, and the automatic creation of objects. DB2 utilities provide integrated support with LOAD and UNLOAD, cross-loader, REORG, CHECK DATA, and CHECK LOB.

In this IBM® Redbook, we provide a totally revised description of the DB2 functions for LOB support as well as useful information about how to design and implement them. We also offer examples of their use, programming considerations, and the enhanced processes used for their administration and maintenance. We also detail how SAP solutions use LOBs.

This IBM Redbook replaces the previous IBM Redbook *Large Objects with DB2 for z/OS and OS/390*, SG24-6571, for DB2 Version 8 and Version 9.1.

The team that wrote this IBM Redbook

This IBM Redbook was produced by a team of specialists from around the world working at the International Technical Support Organization, San Jose Center.

Paolo Bruni is a DB2 Information Management Project Leader at the ITSO, San Jose Center. He has authored several IBM Redbooks™ about DB2 for z/OS and related tools, and has conducted workshops and seminars worldwide. During Paolo's many years with IBM in development and in the field, his work has been mostly related to database systems.

Patric Becker is an Application Programming Department Manager with Sparkassen Informatik GmbH and Co. KG in Germany. He has over eight years experience with DB2 and holds a degree in Computer Science from FOM University of Applied Sciences in Essen. Since joining the company in 1997, Patric has been responsible for several high availability, Customer Relationship Management, DB2, and IMS™ applications. For DB2, he has also assumed the role of database administrator and he was involved in evaluating and applying the new functions of DB2 for OS/390 Version 6 and Version 7. He is also co-author of the IBM Redbooks *DB2 for z/OS Using Large Objects*, SG24-6571, and *DB2 UDB for z/OS: Application Design for High Performance and Availability*, SG24-7134.

Tim Bohlson is an SAP, DB2, and z/OS Specialist. He has worked in various locations worldwide with SAP with DB2 for z/OS customers during the last eight years, as well as instructing for IBM Learning Services on this topic during this time. He has ten years experience with SAP and 19 years experience in the IT industry. Prior to working with SAP

R/3, Tim was an MVS™ System Programmer in Australia for five years, specializing in large system performance and automation. He holds an Honors degree in Computer Engineering from Newcastle University. Tim is also co-author of the IBM Redbook *DB2 UDB for z/OS V8: Through the Looking Glass and What SAP Found There*, SG24-7088.

Burkhard Diekmann is a Senior Developer in the SAP NetWeaver Database Platform organization at SAP's Headquarters in Walldorf. He joined SAP in 1996 and has ten years of experience in DB2 interfacing SAP applications. His areas of expertise include SAP database interface, SAP middleware, and technology development.

Dima Etkin is a DB2 System Administrator and a senior DB2 Consultant in Israel. He has ten years of experience in the DB2 System Administration field, as well as instructing for BLUE Education Center (Israel) and IBM Learning Services (Israel). His areas of expertise include DB2 administration and system tuning.

Davy Goethals is a Belgian systems engineer. He works for Arcelor Technologies, a subsidiary of Arcelor, a steel producing company. He has experience as a DB2 system administrator since DB2 Version 1 in 1985. He participated in multiple DB2 ESP and QPP programs from DB2 V2.1 up to DB2 V7. Davy is also co-author of the IBM Redbook *DB2 for z/OS and OS/390 Version 7 Using the Utilities Suite*, SG24-6289. Currently, he is DB2 team leader within the Arcelor Technologies z/OS department, located in Dunkerque, France, responsible for supporting more than 45 DB2 systems for steel plants all over Europe.

A photo of the team is in Figure 1. Patric Becker is missing from the group picture.



Figure 1 Left to right: Davy, Dima, Burkhard, Tim and Paolo in SVL

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

Rich Conway
Bob Haimowitz

Emma Jacobs
Leslie Parham
Deanna Polm
Sangam Racherla
International Technical Support Organization

Terry Allen
Jeff Berger
Ben Budiman
Larry Cowdery
Bill Franklin
Koshy John
Jeff Josten
Brandon Lee
Li-Mey Lee
Chao-Lin Liu
Bruce McAlister
Patrick Malone
Roger Miller
Esther Mote
Haakon Roberts
Akira Shibamiya
Bryan Smith
Bart Steegmans
James Teng
Frances Villafuerte
Maryela Weihrauch
Allen Yang
Jay Yothers
Maureen Zoric
IBM DB2 for z/OS Development, Silicon Valley Lab, USA

Muni Bandlamoori
Roger Lo
Manfred Olschanowsky
Helmut Roesner
Yeong Soong
IBM/SAP Integration Center, IBM Silicon Valley Lab, USA

Brenda Beane
Seewah Chan
System Performance Evaluation Test, Poughkeepsie, USA

Namik Hrle
Johannes Schuetzner
IBM Systems and Technology Group, Germany

Bernhard Heiningner
Jennifer Johnson
Bernd Kohler
Peter Mohrholz
SAP AG, Walldorf, Germany

Rick Butler
BMO Financial Group, Toronto, Canada

Become a published author

Join us for a two- to six-week residency program! Help write an IBM Redbook dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You'll have the opportunity to team with IBM technical professionals, Business Partners, and Clients.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you'll develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

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

- Use the online **Contact us** review IBM Redbook form found at:

ibm.com/redbooks

- Send your comments in an E-mail to:

redbooks@us.ibm.com

- Mail your comments to:

IBM Corporation, International Technical Support Organization
Dept. HYTD Mail Station P099
2455 South Road
Poughkeepsie, NY 12601-5400



Introduction

The implementation of object-relational in DB2 for z/OS allows you to define custom data types and functions based on the previously existent ones. Some of the data objects you want to model might well be very large and complex. DB2 for z/OS Version 6 has introduced the foundation of object-relational extension, with, on one hand, support for large objects (LOBs), and, on the other one, support for user-defined functions (UDFs), user-defined distinct types (UDTs), and triggers. In this chapter, we outline the implemented functions for object-relational, including extenders, we mention the main changes in DB2 9 for z/OS, and we introduce the LOB-related topics described in this IBM Redbook.

This chapter contains:

- ▶ Object-relational in DB2 for z/OS
- ▶ Changes with DB2 9
- ▶ DB2 for z/OS and large objects
- ▶ The IBM Redbook contents
- ▶ Pointers to LOB functions after DB2 Version 6

1.1 Object-relational in DB2 for z/OS

The object extensions introduced in DB2 UDB for OS/390 Version 6 and beyond offer the benefits of object-oriented technology while increasing the strength of your relational database with an enriched set of data types and functions:

- Large objects

The VARCHAR, VARGRAPHIC, and VARBINARY DB2 data types have a size limit of 32 KB. Although this can be sufficient for small- to medium-size data, applications often need to store large text documents. They might also need to store a wide variety of additional data types such as audio, video, drawings, mixed text, graphics, and images. DB2 provides data types to store these large data objects (LOBs) as strings of up to 2 GB in size.

LOBs are well suited to represent large, complex structures in DB2 tables. You can make effective use of multimedia by storing objects such as complex documents, videos, images, and voice.

- User-defined types

User-defined types (UDTs), like built-in data types, describe the data that is stored in columns of tables where the instances (or objects) of these data types are stored. They ensure that only those functions and operators that are explicitly defined on a distinct type can be applied to its instances.

A distinct type is a user-defined data type that shares its internal representation with a built-in data type, but it is considered to be a separate and incompatible type for semantic purposes. For example, you might want to define a picture type or an audio type which has different semantics but uses the built-in data type BLOB for its internal representation.

- User-defined functions

User-defined functions (UDFs), like built-in functions or operators, support the manipulation of distinct type instances (and built-in data types) in SQL queries.

The built-in functions that are supplied with DB2 are a useful set of functions, but they might not satisfy all of your requirements. You can write user-defined functions to meet the specific needs for your installation. For example, a built-in function can perform a calculation you need, but the function does not accept the distinct types you want to pass to it. You can then define a function based on a built-in function, called a *sourced user-defined function*, that accepts your distinct types. You might need to perform another calculation in your SQL statements for which there is no built-in function. In that situation, you can define and write an external user-defined function.

New and extended built-in functions improve the power of the SQL language with a lot of new built-in functions, extensions to existing functions, and sample user-defined functions.

- Triggers

Triggers bring application logic into the database. Triggers automatically execute a set of SQL statements whenever a specified event occurs. These statements can validate and edit database changes, read and modify the database, and invoke functions that perform operations inside and outside the database.

For more information about UDFs, UDTs, and triggers, refer to *DB2 for z/OS Application Programming Topics*, SG24-6300, and *DB2 for z/OS Data Integrity*, SG24-711.

- Extenders

When you ordered DB2 V8, as part of the obvious base product, you also received the following extenders:

- IAV Extenders - Image, Audio, and Video extenders
- Text Extender
- XML Extender

Text Extender adds full-text retrieval to SQL queries by making use of features available in DB2 that let you store unstructured text documents of up to 2 GB in databases.

You can use the DB2 Extenders™ feature of DB2 to store and manipulate image, audio, video, and text objects. The extenders automatically capture and maintain object information and provide a rich body of APIs.

These extenders define new data types and functions using DB2's built-in support for user-defined types and user-defined functions. You can couple any combination of these data types, that is, image, audio, and video, with a text search query.

The extenders exploit DB2's support for large objects of up to 2 GB, and for triggers that provide integrity checking across database tables ensuring the referential integrity of the multimedia data.

1.2 Changes with DB2 9

The main change in this area is DB2's native *XML support*.

DB2 V8 had started including XML functions into the DB2 engine by integrating XML publishing functions.

DB2 9 for z/OS goes way beyond by introducing a new infrastructure to support XML. It lets your client applications manage XML data in DB2 tables. You can store well-formed XML documents in their hierarchical form, and retrieve all or portions of those documents. Because the stored XML data is fully integrated into the DB2 database system, you can access and manage the XML data by leveraging DB2 functionality.

To efficiently manage traditional SQL data types and XML data, DB2 uses two distinct storage mechanisms. However, the underlying storage mechanism that is used for a given data type is transparent to the application. The application does not need to explicitly specify which storage mechanism to use, or to manage the physical storage for XML and non-XML objects. For more information about XML support, see *DB2 Version 9.1 for z/OS XML Extender Administration and Programming*, SC18-9857, and *DB2 Version 9.1 for z/OS XML Guide*, SC18-9858.

The XML Extender is still shipped with DB2 9 for z/OS, but not the other (Text and IAV) Extenders. The XML Extender requires Language Environment® and the IBM XML Parser for z/OS, C++ Edition to be available at run time. The required parser is provided with Release 6 of the XML Toolkit for z/OS (5655-J51). XML Extender is deprecated in DB2 V9 for z/OS and may be removed in the future. Consider using the XML native support.

DB2 9 for z/OS also introduces *INSTEAD OF triggers*.

They provide a mechanism to unify the target for all read/write access by an application while permitting separate and distinct actions to be taken for the individual read and write actions. INSTEAD OF triggers are triggers that are processed instead of the update, delete, or insert statement that activates the trigger. Unlike other forms of triggers that are defined only on tables, INSTEAD OF triggers can only be defined on views. For more information about INSTEAD OF TRIGGERS, see *DB2 Version 9.1 for z/OS Application Programming and SQL Guide*, SC18-9841.

1.3 DB2 for z/OS and large objects

Large objects can contain pictures, images, text-documents, or even movies, and can be stored directly in DB2 with sizes up to 2 GB per object. Because you might have 254 data sets and 4,096 partitions, that is approximately 65,536 TB for one column.

Normally, large objects are used and manipulated through graphical user interfaces (GUI) via a workstation. So with the implementation of LOBs, we can exploit the server capacity that DB2 for z/OS provides today while displaying and feeding them through a user friendly interface.

The introduction of these data types, with their peculiarities, have shaken the DBAs' worlds and made them adjust to the new, larger environment and new programming techniques. Today, with DB2 9 level of support for LOBs, we get high performance and full operational management.

1.4 The IBM Redbook contents

In this IBM Redbook, we describe the various data types and the management functions introduced from DB2 Version 6 up to DB2 Version 9.1 for z/OS for LOBs support. We provide useful information about how to design, create, and manage LOBs, and we offer examples of their use, programming considerations, and the new processes which are necessary for their implementation, administration, and maintenance.

This IBM Redbook is based on DB2 Version 9.1 for z/OS, but throughout the book, we point out the differences between functionality and syntax of V9 and the previous versions of DB2 for z/OS (mostly V8, but also V7). Notice that all of the examples are given in DB2 9 for z/OS new syntax, but the old syntax is still supported and that some of the new DB2 9 level functions have been made available in V8 and even V7 by APARs.

- ▶ Chapter 2, "Large objects with DB2" on page 9, introduces the new data types and the new infrastructure designed to support the large objects.
- ▶ Chapter 3, "Creating LOBs" on page 23, provides examples of defining LOBs and their associated DB2 data elements.
- ▶ Chapter 4, "Using LOBs" on page 65, gives details about how to load and manipulate LOBs using locators and file reference variables.
- ▶ Chapter 5, "SAP usage of LOBs" on page 125, is a peek into the way this widely used set of solutions exploits LOB for its application structures.
- ▶ Chapter 6, "Utilities with LOBs" on page 159, shows how the individual utilities, old and new, have been enriched to deal with the characteristics of LOBs.
- ▶ Chapter 7, "Data administration with LOBs" on page 211, puts together what you have learned so far and looks at more general scenarios of data administration with LOBs.
- ▶ Chapter 8, "Performance with LOBs" on page 237, is a section with preliminary information about LOBs measurements and general best practice recommendations.

1.5 Pointers to LOB functions after DB2 Version 6

In Table 1-1 on page 5, we provide a summary of the most important enhancements to LOB support that have been introduced after V6 and where you can find their descriptions.

Table 1-1 The LOBs functions that have been introduced after Version 6

Function	Where described	Version 7	Version 8	Version 9.1
JDBC™ functions and DB2 catalog changes for Java™ support requiring LOBs in DB2 catalog	See Chapter 3 of <i>DB2 UDB Server for OS/390 and z/OS Version 7 Presentation Guide</i> , SG24-6121.	Y	Y	Y
DB2 Extenders using LOBs	See Chapter 4 of <i>DB2 UDB Server for OS/390 and z/OS Version 7 Presentation Guide</i> , SG24-6121.	Y	Y	XML Extender only
LISTDEF to allow LOAD and UNLOAD to support LOBs < 32 KB and account for them with a 4 byte length field	See Chapter 5 of <i>DB2 UDB Server for OS/390 and z/OS Version 7 Presentation Guide</i> , SG24-6121.	Y	Y	Y
File reference variables in LOAD and UNLOAD	See 6.3, “LOAD” on page 171 and 6.1, “UNLOAD” on page 160.	UK13720 (PK22910)	UK13721 (PK22910)	Y
LOB support in the cross-loader	See 6.3.3, “Using the cross loader” on page 172.	UK03226 (PQ90263)	UK03227 (PQ90263)	Y
INSERT and UPDATE performance	See 5.12, “Performance measurements” on page 155.	UK15036 (PK22887)	UK15037 (PK22887) OPEN PK25241	Y (integrated)
ROWID transparency	See 4.25 of <i>DB2 UDB for z/OS Version 8: Everything You Ever Wanted thing to Know, ... and More</i> , SG24-6079.		Y	Y
Virtual storage change (LOBVALS and LOBVALA)	See 2.28 of <i>DB2 UDB for z/OS Version 8: Everything You Ever Wanted thing to Know, ... and More</i> , SG24-6079, and 8.2.1, “DB2 subsystem parameters for LOBs” on page 243.		Y	Y
XML2CLOB	See 5.4 of <i>DB2 UDB for z/OS Version 8: Everything You Ever Wanted thing to Know, ... and More</i> , SG24-6079, and 4.1.5, “Functions such as XML2CLOB” on page 72.		Y	Y

Function	Where described	Version 7	Version 8	Version 9.1
SQL stored procedures and LOB variables	See 8.3.7 of <i>DB2 UDB for z/OS Version 8: Everything You Ever Wanted thing to Know, ... and More</i> , SG24-6079, and <i>DB2 for z/OS Stored Procedures: Through the CALL and Beyond</i> , SG24-7083.		Y	Y
Java API enhanced for LOBs	See 5.1.13 of <i>DB2 UDB for z/OS Version 8: Everything You Ever Wanted thing to Know, ... and More</i> , SG24-6079 and <i>DB2 for z/OS Stored Procedures: Through the CALL and Beyond</i> , SG24-7083.		Y	Y
Long SQL statements as CLOBs. Using dynamic SQL, SQL statements up to 2 MB are passed to DB2 as a CLOB or DBCLOB, because a “normal” character string can only be up to 32 KB	See 2.37 of <i>DB2 UDB for z/OS Version 8: Everything You Ever Wanted thing to Know, ... and More</i> , SG24-6079.		Y	Y
CHECK LOB sort enhancement: SYSUT1 and SORTOUT DD statement for sort input and output are no longer needed	See 9.14.2 of <i>DB2 UDB for z/OS Version 8: Everything You Ever Wanted thing to Know, ... and More</i> , SG24-6079.		Y	Y
New samples for LOBs	See the DB2 Installation Guide for each DB2 Version and Appendix A, “Additional material” on page 259.		Y	Y
Avoidance of LOB locks	See 4.5.2, “Locking for LOBs with DB2 9” on page 101.			Y
DRDA LOB flow optimization	See 5.3.2, “CLI Streaming Interface” on page 134.			Y
FETCH CONTINUE	See 4.6.2, “Using FETCH CONTINUE” on page 113.			Y
Automatic creation of objects	See 3.1.1, “Example of automatic creation of objects” on page 24.			Y
Removed requirement that 1 GB or greater LOBs be in a NOT LOGGED LOB table space	See 3.3.1, “LOGGED and NOT LOGGED attributes” on page 44.			Y
REORG SHRLEVEL REFERENCE for a LOB table space	See 6.9, “REORG” on page 185.			Y
Online CHECK LOB	See 6.12, “CHECK LOB” on page 204.			Y

Function	Where described	Version 7	Version 8	Version 9.1
Online CHECK DATA	See 6.11, "CHECK DATA" on page 199.			Y
DSNTEJ7 now creates the sample LOB database using LOAD utility and file reference variables	See the <i>DB2 9 for z/OS Installation Guide</i> .			Y
DSNTIAUL to extract LOB data into files and generate a LOAD statement	See 6.2, "DSNTIAUL" on page 169.			Y



Large objects with DB2

In this chapter, we provide introductory information about large objects, starting with the data types that were introduced in DB2 V6 for LOB support, the general structure of LOB table spaces as well as the LOB manipulation components, including the new 2.4, “LOB file reference variables” on page 18 introduced with DB2 9 for z/OS.

This chapter contains:

- ▶ Introduction to LOB data types
- ▶ The LOB table spaces
- ▶ LOB locators
- ▶ LOB file reference variables

2.1 Introduction to LOB data types

Multimedia application environments rely on many types of large data objects. Those objects can be large text documents, X-ray images, audio messages, pictures, and many other types of images.

Table 2-1 shows some representative sizes of various objects. These are outlined just for comparison purposes. Overall, they just give you some idea of the storage requirements of various types of objects that you may have to deal with.

Table 2-1 Typical average size for large objects

Object	From	To
Bank checks	30 KB	40 KB
Small image	30 KB	50 KB
Large image	200 KB	3 MB
Color image	20 MB	40 MB
Radiology image	40 MB	60 MB
Video	.5 GB/hour	-
Feature length movie	1 GB/hour	-
High resolution video	3 GB/hour	-
High resolution movie	5 GB/hour	6 GB
High definition TV	720 GB/hour	-

The data types provided by DB2, such as VARCHAR or VARBINARY (V9), are not large enough to hold this amount of data, because of their limit of 32 KB. LOB support is based on the set of data types which were introduced with DB2 V6. With large object support, DB2 stores and manipulates data objects that are much larger.

DB2 provides data types to store large data objects (LOBs), which support strings of up to 2 GB in size, well beyond the 32 KB supported by a *varchar* column. Techniques for storing and retrieving these huge amounts of data have also been created within DB2.

The LOB data types allow you to store directly in DB2 tables objects in size of up to 2 GB, and 65,536 Terabytes (TB) per LOB column. Their characteristics depend on the way they are stored in your DB2 subsystem:

- ▶ *Character Large Objects (CLOBs)*
- ▶ *Binary Large Objects (BLOBs)*
- ▶ *Double-byte Character Large Objects (DBCLOBs)*

For internal structure support, DB2 also uses the data type:

- ▶ *ROWIDs*

DB2 also provides host variables and data types that are used for manipulating LOBs:

- ▶ *LOB locators*
- ▶ *LOB file reference variables*

Locators and file reference variables are described later at 2.3, “LOB locators” on page 16 and 2.4, “LOB file reference variables” on page 18.

We now briefly introduce the data types available in DB2 for LOB support.

CLOBs

A character large object (CLOB) is a varying-length string with a maximum length of 2,147,483,647 bytes (2 gigabytes minus 1 byte). A CLOB is designed to store large SBCS data or mixed data, such as lengthy documents. For example, you can store information such as an employee resume, the script of a play, or the text of a novel in a CLOB. Alternatively, you can store such information in UTF-8 in a mixed CLOB. A CLOB is a varying-length character string, which can be thought of as a varchar field of (almost) unlimited length.

Most text documents stored on other platforms cannot be converted to a CLOB value easily because most known editing applications save data using their own format. This format usually contains an interspersed amount of control information used for font types, font sizes, and layout purposes. If you want to store this data as a CLOB value and access it via DB2 functions such as SUBSTR or POSSTR, you might have to convert the data from its format on your client platform to a plain or tagged text file before inserting into a CLOB column. You can consider using a CLOB column for storing large text documents. If you plan to store your PC documents in DB2 for z/OS without converting them, because you want to access them from your client applications for further processing, you need to store them as BLOBs to avoid loss of font and layout control information.

BLOBs

A binary large object (BLOB) is a varying-length string that has a maximum length of 2,147,483,647 bytes (2 gigabytes minus 1 byte). A BLOB contains a binary string representing binary data, which is a sequence of bytes and typically unreadable. A BLOB is designed to store various data such as pictures, voice, and mixed media. BLOBs can also store structured data for use by distinct types and user-defined functions.

Although BLOB strings and FOR BIT DATA character strings might be used for similar purposes, the two data types are incompatible. The BLOB function or CAST function can be used to change a FOR BIT DATA character string into a BLOB string.

Normal character strings defined with the FOR BIT DATA option cannot be assigned a CCSID, and the same rule applies to BLOB strings.

DBCLOBs

A double-byte character large object (DBCLOB) is a varying length string of double-byte characters that can be up to 2 GB long. Using a DBCLOB column, you can store up to 1,073,741,823 double-byte characters in a single DBCLOB value. A DBCLOB is used to store large double-byte character set (DBCS) character-based data such as documents written with a double-byte CCSID. A DBCLOB is considered to be a graphic string. You can find double-byte CCSIDs for languages such as Japanese (extended Katakana or Katakana-Kanji), Korean, or Chinese (simplified or traditional).

ROWIDs

The ROWID data type (and column) definition was introduced with DB2 V6 to uniquely and permanently identify a row in a table. To understand the role of the ROWID, and why DB2 creates a value for a ROWID column whenever a row is inserted in a table containing this new data type, you must first understand the overall picture of the DB2 objects involved in supporting LOBs, as described first at 2.2, “The LOB table spaces” on page 12, and then in more detail at 3.2, “Defining ROWIDs” on page 39.

2.2 The LOB table spaces

CLOBs, BLOBs, and DBCLOBs are the data types provided by DB2 for storing large objects. DB2 also provides different techniques to accommodate storing these potentially huge amounts of data within the DB2 physical structures. In fact, a large object (LOB) in DB2, even though it can be compared to a more familiar string element, such as a CHAR or a VARCHAR column in terms of type of contents, can reach the size of 2 GB minus one byte (2,147,483,647 bytes). This requires different storing and handling techniques in order to minimize the impact on usability and performance.

2.2.1 Single LOB column table space

Generally, non-LOB columns are stored in what we refer to as a *base table*. LOBs belong to a base table and are related to it, but they are not stored in the same table with the other non-LOB columns, they are stored in a LOB table space. The table, where the rows of a LOB column live and which is embedded in the LOB table space, is called an *auxiliary table*. See Figure 2-1. Each LOB value is assigned to a different page.

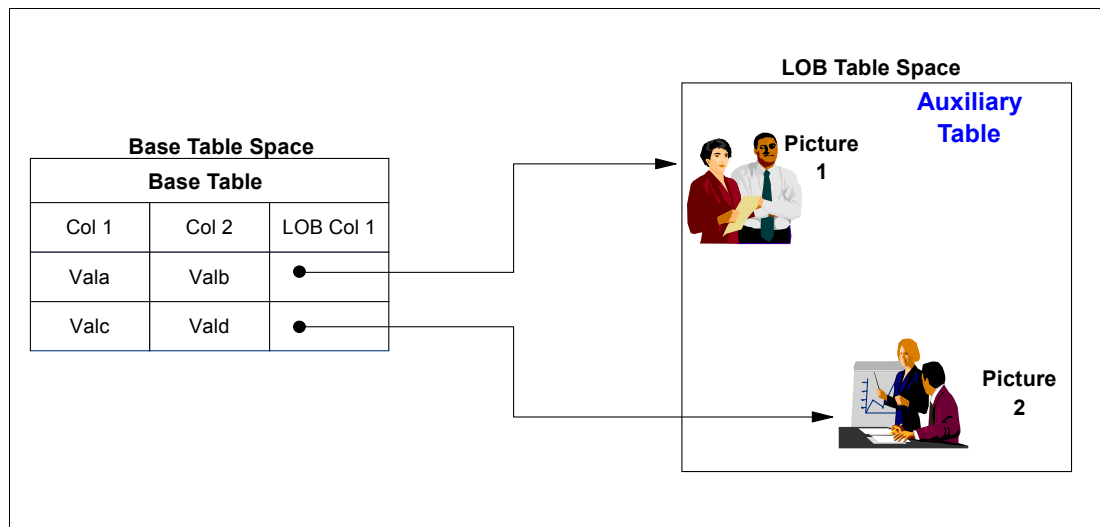


Figure 2-1 Association between base table and auxiliary table

2.2.2 Multiple LOB column table space

Each LOB column in a base table requires its own LOB table space, so LOB columns are also separated from other LOB columns belonging to the same base table. Each LOB table space and auxiliary table will contain the values of the same LOB column. It is important to know that each page can only contain a LOB or a portion of a LOB; it never contains values of two LOBs or two LOB columns. For a pictorial view of the different LOB columns and their associated LOB table spaces, see Figure 2-2 on page 13.

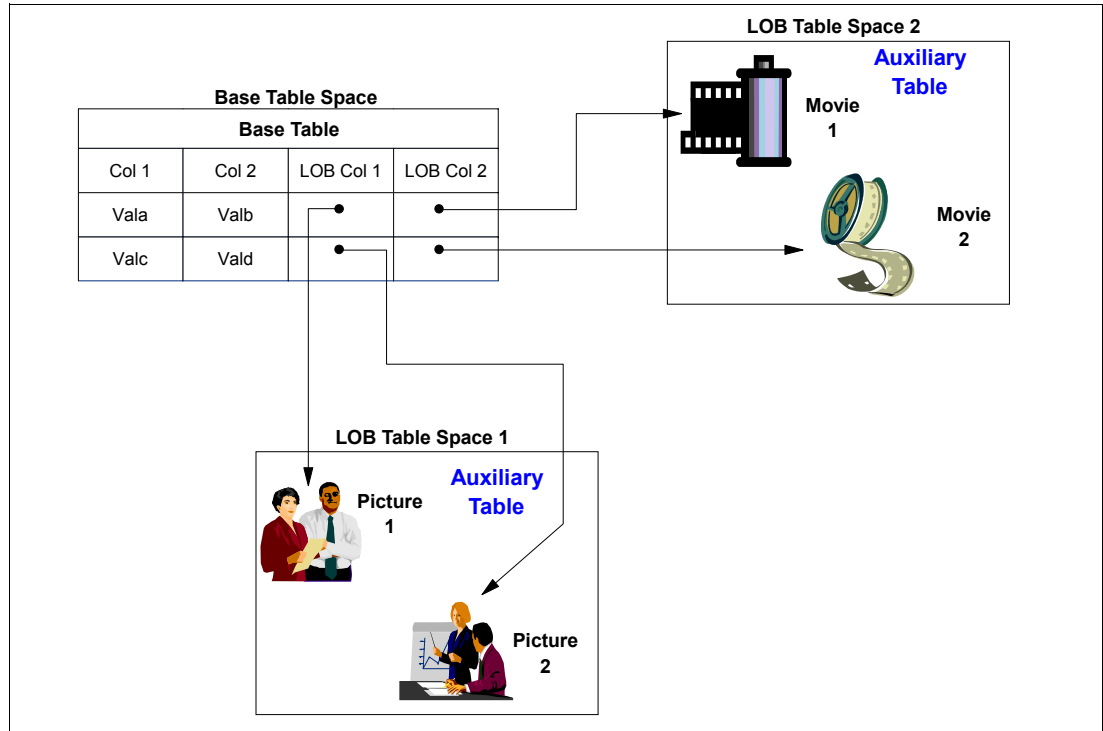


Figure 2-2 Base table with two LOB columns and the two associated LOB table spaces

2.2.3 Partitioned LOB table space

If the base table is partitioned, every LOB column in every partition has its own LOB table space. This means up to 4,096 LOB table spaces per LOB column if the base table has 4,096 partitions. Figure 2-3 on page 14 gives you a better understanding of a partitioned multi-LOB column base table and the required LOB table spaces.

The LOB column itself is referred to as an *auxiliary column*, because it is not stored in the base table. The rows in the base table are associated to the LOB columns residing in the auxiliary table in the LOB table space using the ROWID as a pointer from the base table to the auxiliary table. For further information about ROWIDs, see 3.2, “Defining ROWIDs” on page 39.

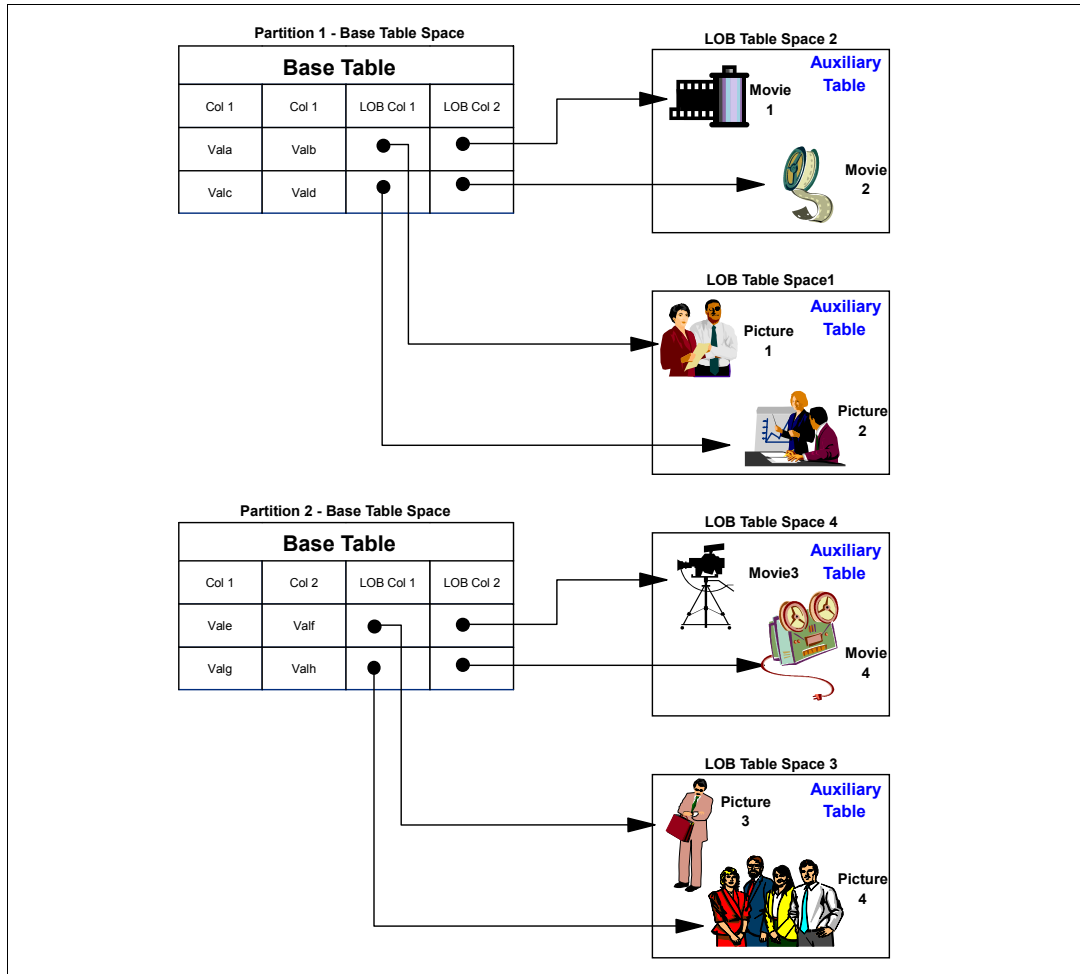


Figure 2-3 Partitioned base table containing two LOB columns

2.2.4 The full LOB implementation structure

In order to quickly locate the proper LOB value in the auxiliary table, an auxiliary index must be created on the auxiliary table. This index includes the ROWID as its only column. Additionally, a LOB indicator is also stored in the base table. You can find more information about this indicator in “A few more details about the base table” on page 31. See Figure 2-4 on page 15 for a summary picture of all the objects mentioned here and how they work together.

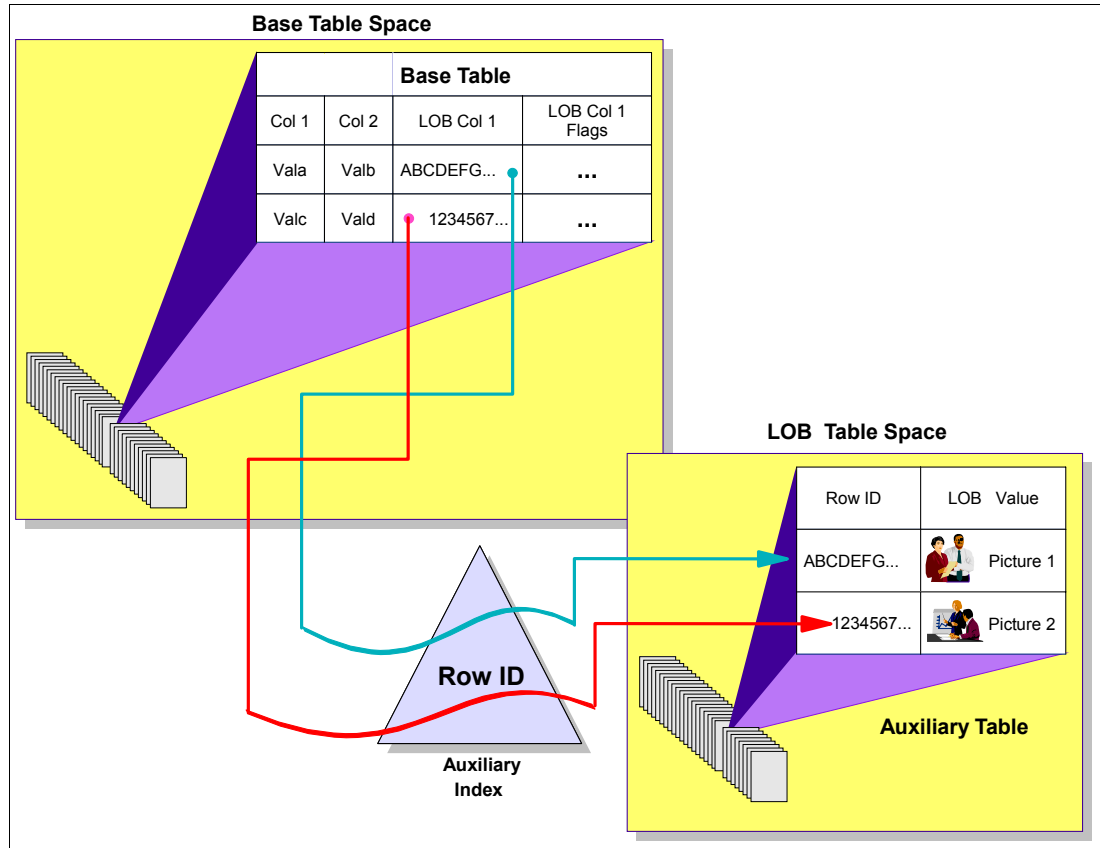


Figure 2-4 Association between base table, ROWID, auxiliary table, and LOB table space

An important reason for the separation of LOB columns is performance. The decision was made during the design phase of large object support, based on experiences using middleware systems and LOBs. Assuming table space scans on the base table, LOB values do not have to be processed during these scans. Probably most of the scanning time would be spent in scanning LOB columns if the LOB columns resided in the same table as the non-LOB columns.

Based on the nature of LOBs, they can be larger than the biggest available page size, which is still 32 KB in DB2 9 and is also valid for LOB table spaces. Therefore, a single LOB value can span pages.

Because each data set of a LOB table space can grow up to 64 GB, and there can be up to 254 data sets per table space, the maximum size of a non-partitioned LOB table space is 16,256 GB (16 TB) in total. Because the number of partitions can grow up to 4,096 in the base table, there are 4,096 single LOB table spaces, each holding up to 16 TB of data as a maximum size. This gives a grand total of 4,096 x 16 TB = 65,536 TB available for a single column of LOB data.

A single database can hold a maximum of 65,535 objects, or to be more specific, X'FFFF' object identifiers (OBDs).

Regardless of the number of partitions, you use one OBD for each auxiliary relationship per LOB column. Furthermore, DB2 uses 5 OBDs per LOB column per partition. Therefore, this gives us a maximum of 3 LOB columns for a 4,096 partition table, not exceeding 65,535 OBDs. According to these values:

$$3 + n + 5np \leq 65535$$

with n as the number of LOB columns in your base table and p as the number of partitions. The formula gives you the number of partitions and LOB columns that can reside inside one database.

The values are summarized in Table 2-2.

Table 2-2 Maximum number of LOB columns by partitions

Partitions	Data sets	Maximum number of LOB columns
250	12,250	48
1,000	13,000	12
4,096	16,384	3

2.3 LOB locators

In general, when the application is accessing data, DB2 has to deal with the materialization of data on the layers involved: auxiliary storage, central storage, data buffers, and the movement of data across them. The materialization of LOB values consists of placing LOB data in contiguous storage. Because LOB values can be very large, DB2's actions are inspired by the objective of avoiding materialization of LOB data until it becomes absolutely necessary. Furthermore, if the application is running on a client machine, distinct from the database server, the transfer of LOB values from the server to the client adds a sizable time and resource consumption. Application programs typically process LOB values a piece at a time, rather than as a whole. For all of those cases where an application does not need (or want) the entire LOB value stored in application memory, the application can reference a LOB value using the large object locator (LOB locator) and avoid materialization.

A *LOB locator* is a host variable with a value that represents a single LOB instance in the database server. LOB locators have been developed to provide users with a mechanism by which they could easily manipulate very large objects in application programs without requiring them to store the entire LOB value in the application's memory.

You can consider locators roughly as cursors that reference remote data. Locators, and the structures associated to them, are contained in virtual storage blocks allocated by DB2 in the DBM1 address space. The LOB data referenced by the locators is allocated by DB2 in memory structures, which are dynamically created for the duration of the request.

2.3.1 Purpose of LOB locators

A LOB locator is a value generated by DB2 when a LOB string is assigned to a host variable that was previously specified as a LOB locator in your application program. Every LOB locator, also called *host-identifier*, is a special type of SQL object materialized as a four-byte token used to represent the LOB. This representation allows the value of the large object strings to be contained in the host-identifier, rather than the actual string of bytes of the large object. Figure 2-5 on page 17 provides an example of LOB locator usage.

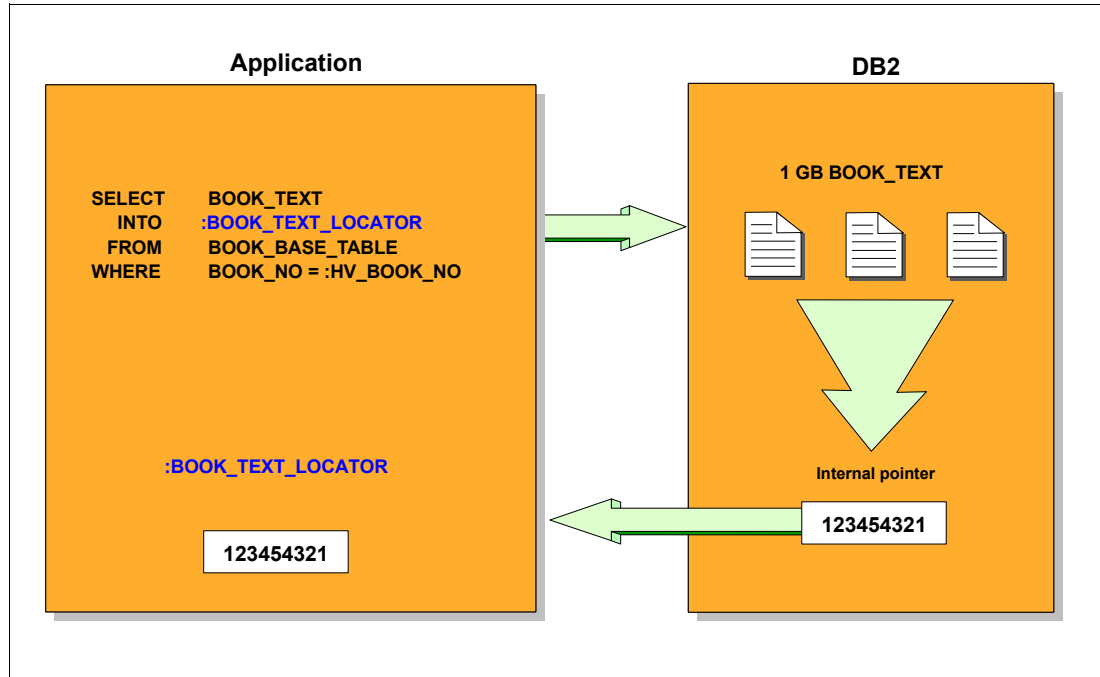


Figure 2-5 Assigning a LOB locator for a LOB value

A LOB locator is an association between a host variable and its corresponding LOB value in DB2 at a point in time. The application only accesses the locator while the entire LOB value resides in DB2 and is not propagated to the application program. Using this technique, an application does not need to acquire a buffer large enough to contain the entire LOB value. Instead, the application deals only with LOB locators, therefore largely reducing the amount of resources to be allocated. The definition and usage are not mandatory; however, considerations on performance soon lead you to the conclusion that it might be a better idea for the applications to use them, because locators dramatically reduce the movement of data between the different address spaces involved in LOB data management as well as greatly reducing connectivity issues.

2.3.2 Different types of LOB locators

Each LOB format (BLOB, CLOB, and DBCLOB) has its own type of locator, so currently there are three different types of LOB locators:

- ▶ Binary Large Object locator, which is associated with large binary strings
- ▶ Character Large Object locator, which is associated with large character strings
- ▶ Double-byte Character Large Object locator, which is associated with large graphic strings

The definition of a LOB locator depends on the language you choose for developing programs for LOB usage. The syntax for defining LOB locators in COBOL is shown in Example 2-1.

Example 2-1 Host variable definitions for LOB locators in COBOL

01	BLOB-LOCATOR	USAGE	IS SQL TYPE IS BLOB-LOCATOR.
01	CLOB-LOCATOR	USAGE	IS SQL TYPE IS CLOB-LOCATOR.
01	DBCLOB-LOCATOR	USAGE	IS SQL TYPE IS DBCLOB-LOCATOR.

The DB2 precompiler converts the locator structure into the COBOL structure as reported in Example 2-2 on page 18.

Example 2-2 What the DB2 precompiler makes of LOB locators

01 BLOB-LOCATOR	PIC S9(9) COMP.
01 CLOB-LOCATOR	PIC S9(9) COMP.
01 DBCLOB-LOCATOR	PIC S9(9) COMP.

The locator 4-byte value is then stored in a host variable; the program, as already shown in Figure 2-5 on page 17, can use it to refer to a LOB value. Even if every LOB locator shows up identically for all definitions of host variables, DB2 knows the associated LOB type and does not let you use a locator with a different type of LOB. If you define a CLOB locator and try to use it for a BLOB, SQLCODE -171 is issued.

You can only use LOB locators inside an application program; you cannot deal with them interactively. This means that you cannot use them, for instance, with SPUFI or QMF™.

For more information about LOB locators, refer to 4.2, “LOB locators” on page 73 and 8.1.2, “Materialization avoidance techniques” on page 241.

LOB locator disadvantages

There are some disadvantages for using LOB locators. Their coding is pretty cumbersome, and in the long run, they can cause problems in the application. The code becomes complicated and hard to maintain. The need to chain the locators when dealing with extremely large objects and limited storage in a user address space can cause the program to abend due to lack of storage in a user address space.

Though LOB locators are a good alternative to LOB materialization, we can safely state that a better alternative was introduced to replace them in most circumstances. When no complicated manipulations on LOBs are required, or the application is not for usage in a distributed environment, we recommend that you program using LOB file reference variables as described in 2.4, “LOB file reference variables” on page 18.

2.4 LOB file reference variables

The purpose of file reference variables is to import or export data between a LOB column and an external file outside of the DB2 system. In the past, if you used a host variable to materialize the entire LOB in the application, your application would not only need adequate storage but would also incur poor performance, because the file I/O time would not be overlapped with any DB2 processing or network transfer time.

Locator variables used in conjunction with the SUBSTR function can be used to overlap the file I/O time with DBM1 processing time or network transfer time and also to avoid materializing the whole LOB in the application. However, there is still some CPU overhead to transfer pieces of the LOB between DBM1 and the application.

LOB file reference variables accomplish the same function using less CPU time and avoiding the use of any application storage for the LOBs. LOB file references are also easier to use than locator variables. LOB file reference variables are supported within a DB2 for z/OS system or in a distributed configuration between DB2 for z/OS subsystems.

In DB2 9, three new SQL host variables have been introduced:

- ▶ **BLOB_FILE**
- ▶ **CLOB_FILE**
- ▶ **DBCLOB_FILE**

File reference host variables can be used in applications to insert a LOB from a file into a DB2 table or to select a LOB from a DB2 table into a file. They can be used to update a LOB from a file as well. When you use a file reference variable, you can select or insert an entire LOB value without contiguous application storage to contain the entire LOB. In other words, LOB file reference variables move LOB values from the database server to an application or from an application to the database server without going through the application's memory. Furthermore, LOB file reference variables bypass the host language limitation on the maximum size allowed for dynamic storage to contain a LOB value.

2.4.1 DB2-generated file reference variable constructs

For each LOB file reference variable that an application declares, DB2 generates an equivalent construct that uses the host language data types. When an application refers to a LOB file reference variable, the application must use the equivalent construct that DB2 generates. If the construct is not used, the DB2 precompiler issues an error. The construct describes properties of the file as shown in Table 2-3.

Table 2-3 DB2-generated construct

Data type	BLOB, CLOB, or DBCLOB. This property is specified when the variable is declared using the BLOB_FILE, CLOB_FILE, or DBCLOB_FILE data type.
File name	This property must be specified by the application program at run time. The file name property can have the following values: <ul style="list-style-type: none"> ► The complete path name of the file. We recommend this. ► A relative file name. If a relative file name is provided, it is appended to the current path of the client process. A file should be referenced only once in a file reference variable.
File name length	This property must be specified by the application program at run time.
File options	An application must assign one of the file options to a file reference variable before the application can use that variable. File options are set by the INTEGER value in a field in the file reference variable construct. One of the following values must be specified for each file reference variable: <ul style="list-style-type: none"> ► Input (from application to database): SQL-FILE-READ. A regular file that can be opened, read, and closed. ► Output (from database to application): SQL-FILE-CREATE. If the file does not exist, a new file is created. If the file already exists, an error is returned. SQL-FILE-OVERWRITE. If the file does not exist, a new file is created. If the file already exists, it is overwritten. SQL-FILE-APPEND. If the file does not exist, a new file is created. If the file already exists, the output is appended to the existing file.
Data length	The length, in bytes, of the new data written to the file, provided by DB2 server.

2.4.2 Language support for LOB file reference variables

Encoding scheme

The encoding scheme CCSID of the file name is based on the application's encoding scheme. The CCSID of the LOB (contents of the file) can be set by the application by using the SQL DECLARE host variable CCSID statement if it is different from the application encoding scheme. DB2 performs any character conversion required prior to inserting LOB data into a DB2 table or placing a LOB into a file.

Programming support for LOB file reference variables

You can declare a LOB file reference variable or a LOB file reference array for applications that are written in C, COBOL, PL/I, and Assembler. The LOB file reference variables do not contain LOB data. They represent a file that contains LOB data. Database queries, updates, and inserts can use file reference variables to store or retrieve column values. As with other host variables, a LOB file reference variable can have an associated indicator variable.

The definition of a LOB file reference variable depends on the language you choose for developing programs for LOB usage. The syntax for defining LOB file reference variables in COBOL is shown in Example 2-3.

Example 2-3 Host variable definition for BLOB file reference variable in COBOL

```
01 MY-BLOB-FILE      SQL TYPE IS BLOB-FILE.
```

The DB2 precompiler converts the declaration into the COBOL structure as reported in Example 2-4.

Example 2-4 What the DB2 precompiler makes of BLOB file reference variable

```
01 MY-BLOB-FILE.  
  49 MY-BLOB-FILE-NAME-LENGTH PIC S9(9) COMP-5.  
  49 MY-BLOB-FILE-DATA-LENGTH PIC S9(9) COMP-5.  
  49 MY-BLOB-FILE-FILE-OPTION PIC S9(9) COMP-5.  
  49 MY-BLOB-FILE-NAME PIC X(255).
```

Table 2-4 shows the precompiler generated file option constant declarations. You can use these constants to set the file option variable when you use file reference host variables.

Table 2-4 File option constants

Constant name	Constant value
SQL-FILE-READ	2
SQL-FILE-CREATE	8
SQL-FILE-OVERWRITE	16
SQL-FILE-APPEND	32

NULL indicator support for file reference variables

Like all other host variables, a LOB file reference variable can have an associated indicator variable. Indicator variables for LOB file reference behave differently from indicator variables of normal host variables. Because the file name can never be NULL for either input or output, a negative indicator variable value indicates that the LOB value represented by the file reference variable is NULL.

When a NULL value of a LOB column is returned from the database, the indicator variable is set and the file associated with the variable is not opened by DB2.

When a NULL value is set by an application to place a LOB file reference into a DB2 LOB column, the file is not opened for writing.

ODBC API support of LOB file reference

Two new APIs are added to support LOB file reference.

The new APIs are:

- **SQLBindFileToCol**

SQLBindFileToCol is used to bind a LOB column in a result set to a file reference allowing the column data to be transferred into a file when the row is fetched.

- **SQLBindFileToParam**

SQLBindFileToParam is used to bind a parameter marker to a file reference allowing the data from the file to be transferred into a LOB column.

An ODBC application can use either the statement attributes or the keyword `CURRENTAPPENSCHM` to override the default CCSID setting. If both are specified, the statement attributes override the setting of `CURRENTAPPENSCHM` in the INI file. ODBC converts the input file name to the application's encoding scheme before passing it to DB2 for processing.

2.4.3 File local/client support

The DB2 support of LOB file reference is on the local side (non-distributed environment) or on the client side (distributed environment) in a DB2 to DB2 (both z/OS) connection. The file referenced by a LOB file must reside on the system on which program is running, or it must be accessible by the system on which the program is running. If a remote stored procedure that issues an SQL statement that uses a LOB file reference is called, the file must reside on the system on which the stored procedure is running or be accessible by the system on which a stored procedure is running.

A LOB file reference cannot be used in the stored procedure or user-defined function as an input or output parameter. If a LOB file reference is used in the stored procedure or user-defined function as an input or output parameter, `SQLCODE -104` is issued.

DB2 9 for z/OS SQL supports sequential files (DSORG=PS), Hierarchical File System (HFS) files, Partition Data Set (PDS) files, and Partition Data Set Extended (PDSE) files.

Utilities support of LOB file reference

LOAD and UNLOAD utilities support the capability of LOB file reference variables introduced in DB2 9, and support is retrofitted to V7 and V8 with APARs PK10278 and PK22910. More information is available in Chapter 6, "Utilities with LOBs" on page 159.

Restriction: Utilities do not support sequential (BSAM or QSAM) data sets for LOB file reference variables.

DSNZPARMs affected

DB2 9 introduces the new parameter `MAXOFILR` (MAX OPEN FILE REFS) to control the maximum number of data sets that can be open concurrently for processing of LOB file reference. The value setting for the parameter appears on the installation panel `DSNTIPE` as shown on Figure 2-6 on page 22.

Though the default value is 100, the highest number in the range is also effectively limited to the setting of the `CTHREAD` (MAX USERS) parameter. This limitation exists because a thread can hold at most one file open using file references. With z/OS 1.7, `DSMAX` can be 64K and even get up to 100K, as you note below.

DSNTIPE	INSTALL DB2 - THREAD MANAGEMENT	
===>		
Check numbers and reenter to change:		
1 DATABASES	===> 100	Concurrently in use
2 MAX USERS	===> 200	Concurrently running in DB2
3 MAX REMOTE ACTIVE	===> 200	Maximum number of active database access threads
4 MAX REMOTE CONNECTED	===> 10000	Maximum number of remote DDF connections that are supported
5 MAX TSO CONNECT	===> 50	Users on QMF or in DSN command
6 MAX BATCH CONNECT	===> 50	Users in DSN command or utilities
7 SEQUENTIAL CACHE	===> BYPASS	3990 storage for sequential IO. Values are SEQ or BYPASS.
8 MAX KEPT DYN STMTS	===> 5000	Maximum number of prepared dynamic statements saved past commit points
9 CONTRACT THREAD STG	===> NO	Periodically free unused thread stg
10 MANAGE THREAD STORAGE	===> NO	Manage thread stg to minimize size
11 LONG-RUNNING READER	===> 0	Minutes before read claim warning
12 PAD INDEXES BY DEFAULT	===> NO	Pad new indexes by default
13 MAX OPEN FILE REFS	===> 100	Maximum concurrent open data sets
PRESS: ENTER to continue RETURN to exit HELP for more information		

Figure 2-6 DSNTIPE installation panel

Note: APARs PK13287 and PK29281 and z/OS 1.7 GRS SPE allow support of up to 100,000 open data sets.

Because some storage needs to be allocated for file processing, the parameter also has some effect on the consumption of storage below the 2 GB bar.



Creating LOBs

In this chapter, we provide information about how to define and load *LOBs*. We discuss the following topics:

- ▶ Alternatives in defining LOBs
- ▶ Defining ROWIDs
- ▶ LOBs and LOG activity
- ▶ Additional considerations for creating LOB objects
- ▶ LOBs are different DB2 objects
- ▶ Physical layout of LOBs

3.1 Alternatives in defining LOBs

In this section, we show how LOBs are created through practical examples.

We provide you with sample DDL needed for the creation of all LOB dependent objects including a base table, a LOB table space, an auxiliary table, and an auxiliary index. We also discuss the way DB2 stores the data at page level and how buffer pools are associated with a LOB table space. We point out the differences between DB2 V8 and DB2 9.

In our example, we assume that we need to set up a new table containing non-LOB columns and two LOB columns: a CLOB and a BLOB column. We create a base table containing information about a book, storing the book itself as a CLOB, and the image of the book cover as a BLOB.

DB2 supports three different approaches for creating all of the necessary objects for a LOB environment:

- ▶ Automatic creation of objects by DB2

The automatic creation of objects is supported starting with DB2 9.

If you do not specify an IN clause or a database name on a CREATE TABLE statement, DB2 assumes automatic creation of all necessary objects. For a CREATE TABLE statement, DB2 implicitly creates a database and a table space to hold the table. If the containing table space is implicitly created, DB2 creates all of the system required objects for the user. For any base table containing at least one LOB column, the objects automatically created are the LOB table space, the auxiliary table, and the auxiliary index.

- ▶ Setting CURRENT RULES special register

You can let DB2 create all necessary LOB objects by setting CURRENT RULES special register value to STD.

- ▶ Manual creation of objects

In this case, you have the most flexibility and the opportunity to comply with existing naming standards.

We look at the automatic creation of objects provided by DB2 9 closely, then we investigate the CURRENT RULES special register, and finally, we look at the manual creation of objects. For ROWID definition, see 3.2, “Defining ROWIDs” on page 39.

3.1.1 Example of automatic creation of objects

DB2 9 provides the ability for the user to only create the base table, including the definition of all LOB columns. When you decide to let DB2 create all the necessary objects for you, the only item that you have to define is the base table containing one or more LOB columns.

By omitting the IN clause on your CREATE TABLE statement for the base table, DB2 creates the following objects for you implicitly if at least one LOB column is specified:

- ▶ Database for the base table space and the LOB table space
- ▶ Table spaces for the base table and the auxiliary table
- ▶ Enforcing primary key index
- ▶ Enforcing unique key index
- ▶ Index on ROWID column on the base table if ROWID GENERATED BY DEFAULT was specified

- Auxiliary table
- Auxiliary index

If no LOB column is specified, DB2 does not create any auxiliary objects.

Note: Automatic creation of objects does not apply to explicitly partitioned tables. An implicitly created table space always defaults to a partitioned-by-growth table space.

Note: If the ROWID column is omitted during the CREATE TABLE statement for the base table, DB2 generates a hidden ROWID column for you. See “A few more details about the base table” on page 31 for a detailed description.

See Example 3-1 for a DDL to invoke automatic object creation. This creates a table named BOOK_BASE_TABLE with four columns: two fixed length character (CHAR) columns and two LOB columns. The IN *database-name.table-space-name* clause defined the table space. The absence of the IN clause now triggers the automatic creation mechanism.

Example 3-1 DDL for a base table resulting in automatic object creation

```
CREATE TABLE BOOK_BASE_TABLE
( BOOK_NO          CHAR(10)          NOT NULL WITH DEFAULT
, DESCRIPTION      CHAR(32)          NOT NULL WITH DEFAULT
, BOOK_TEXT        CLOB(500M)        NOT NULL
, BOOK_COVER       BLOB(1M)          NOT NULL );
```

If you use the DDL shown in Example 3-1, DB2 creates the objects listed in Example 3-2.

Example 3-2 Resulting objects for automatic object creation

```
DSNT360I  -DB9B *****
DSNT361I  -DB9B *  DISPLAY DATABASE SUMMARY
              *    GLOBAL
DSNT360I  -DB9B *****
DSNT362I  -DB9B      DATABASE = DSN00050  STATUS = RW
              DBD LENGTH = 4028
DSNT397I  -DB9B
NAME      TYPE PART  STATUS              PHYERRLO PHYERRHI CATALOG  PIECE
-----
BOOKRBAS TS      0001 RW
L99TVXZL LS              RW
L99TW33X LS              RW
IB001ITW IX              RW
IB00KRBO IX              RW
*****  DISPLAY OF DATABASE DSN00050 ENDED  *****
DSN9022I  -DB9B DSNTDDIS 'DISPLAY DATABASE' NORMAL COMPLETION
```

DB2 implicitly creates two LOB table spaces named L99TVXZL and L99TW33X, and the auxiliary indexes named IB001ITW and IBOOKRBO. Because we have not specified a base table space, a partitioned-by-growth table space named BOOKRBAS is implicitly created as well. All objects are placed in database DSN00050.

Tip: Use REPORT utility to find out about the names DB2 has chosen after you have retrieved the table space name for the base table using SYSIBM.SYSTABLES. For details, see 6.7, “REPORT” on page 182.

Note that even if you create more than one table inside one unit of work using automatic creation of objects, all tables are placed in different databases; therefore, they are also placed in different table spaces.

Parts of the names are chosen randomly, so that issuing the DDL in your system most likely results in different names of the affected objects.

Note: You cannot create your own table in an implicitly created table space nor can you create your own table space in an implicitly created database.

Dropping implicitly created objects

If DROP TABLE is issued and the residing table space was implicitly created by DB2, DB2 drops all the related system required objects that are explicitly or implicitly created, including:

- ▶ Enforcing primary key index
- ▶ Enforcing unique key index
- ▶ Index on ROWID column on the base table if applicable
- ▶ Auxiliary table
- ▶ Auxiliary index
- ▶ Table spaces for the base table and the auxiliary table

If a table has a LOB column defined and the base table or the base table space is dropped, the LOB table space is implicitly dropped.

If the base table space is explicitly created and the LOB table space is implicitly created, DROP is allowed.

Note: If the base table is dropped, all dependent objects created implicitly, including the base table space, are dropped, too. If the last object inside the automatically created database is dropped, the database still remains in your DB2 subsystem and can be explicitly dropped by using a DROP DATABASE statement if you prefer.

Implicit databases

If you do not specify a database name, DB2 implicitly creates a database and generates a name for it. If the number of existing implicitly created databases reaches 60 000, DB2 wraps around and uses an existing database that has been implicitly created by DB2 instead of implicitly creating a new one. Each database that is created implicitly by DB2 can contain multiple table spaces. Trying to create your own objects in automatically created databases or table spaces is not allowed and results in SQLCODE -2035, because this involves one or more implicitly created objects.

Restriction: Only an implicitly created table space is allowed to be created in an implicitly created database.

If you issue a CREATE TABLE statement without specifying a database name, DB2 does not roll back a successful implicitly created database if any failure occurs during the table or table space creation in this database.

Note: On an implicit database creation, DB2 uses SYSIBM as the database creator.

Table 3-1 on page 27 shows the attributes used for automatic database creation.

Table 3-1 Attributes for implicit database creation

NAME	BUFFERPOOL	INDEXBP
DSNnnnnn (nnnnn = 00001 - 60000)	Default: 4 KB	Default: IDXBPOOL setting
STOGROUP	IMPLICIT	ENCODING_SCHEME
SYSDEFLT	'Y'	Default: DECP setting
SBCS_CCSID	DBCS_SSID	MIXED_CCSID
Default: DECP setting	Default: DECP setting	Default: DECP setting

The names of implicitly created databases start with DSN followed by exactly five digits, providing a name range from DSN00001 to DSN60000. If your number of implicitly created databases exceeds 60 000, DB2 looks for an existing DSN00001 and creates all the necessary objects in it. If DSN00001 does not exist, DB2 creates a DSN00001 database and creates the objects in it. DB2 then continues with DSN00002 the next time DB2 objects are created implicitly, and DB2 resumes the incrementing of the last digits, either creating a database or adding another set of objects to the existing database.

Implicit table spaces

If you do not specify the IN clause or a table space name in your CREATE TABLE statement, DB2 implicitly creates a partitioned-by-growth table space for you.

When DB2 implicitly creates a table space, the table spaces in an implicitly created database are to be created as partitioned-by-growth. A partitioned-by-growth table space uses 4 GB as DSSIZE, 256 for MAXPARTITIONS, and LOCKSIZE ROW as its default values.

The DSNZPARM parameter IMPDSDEF corresponds to DEFINE YES / DEFINE NO on a CREATE TABLESPACE statement and allows you to specify whether the underlying data set is to be defined when a table space is created in an implicitly created database.

Parameter IMPTSCMP lets you choose if you want the implicitly created table space using compression or not. The MGEXTSZ subsystem parameter specifies whether to use a sliding scale for optimizing secondary extent allocations for DB2-managed data sets.

See Table 3-2 for default values you can influence for implicitly created table spaces.

Table 3-2 Default values for implicitly created table spaces

DEFINE YES / NO	COMPRESS YES / NO	Optimizing secondary extent allocations
Default specified using IMPDSDEF	Default specified using IMPTSCMP	Default specified using MGEXTSZ

A buffer pool is associated with the created table space according to the record size. The next larger page size can be chosen if the maximum record size reaches approximately 90% of the capacity of the smaller page size when the default buffer pool is not large enough to support future extensions.

The implicitly created table space CCSID is the same as the table CCSID if it is specified in the CREATE TABLE statement. Otherwise, the CCSID associated with the table space is set as the DECP default CCSID value.

See Table 3-3 for the default values used by DB2 for the automatic creation of our base table space. The only parameter you can influence using automatic object creation is the buffer pool specified in DSNZPARM TBSBPOOL, which specifies the default buffer pool for user data and, which was set to BP0 in our system. The other attributes used for implicitly created objects cannot be changed and can only be altered after the object is created.

Table 3-3 Base table space created using automatic object creation

TSNAME	BUFFERPOOL	LOCKSIZE	LOG	CLOSE
BOOKRBAS	BP0	ROW	YES	YES
DSSIZE	PQTY/SQTY	FREEPAGE	PCTFREE	GBPCACHE
4194304	3 / -1	0	5	
MAXROWS	MAXPARTITIONS	SEGSIZE		
255	256	4		

The SQTY of -1 indicates that DB2 uses a sliding scale algorithm to allocate an appropriate amount of space as a secondary extent. See *Disk storage access with DB2 for z/OS*, REDP-4187, for details about the sliding algorithm used for extent allocation.

Implicit auxiliary table space

You can find the parameters used by DB2 during automatic object creation for the LOB table space reported in Example 3-4. Again, note the BUFFERPOOL has been picked up from TBSBPOOL, which is BP0 in our test system.

Table 3-4 LOB table space created using automatic object creation

TSNAME	BUFFERPOOL	LOCKSIZE	LOG	CLOSE
L99TVXZL	BP0	ANY	YES	YES
DSSIZE	PQTY/SQTY	FREEPAGE	PCTFREE	GBPCACHE
4194304	50 / -1	0	0	

Again, the only controlling parameter is the buffer pool specified in DSNZPARM TBSBPOOL. All other parameters can only be altered after the object is created. The name of the LOB table space is an 8-character string every time DB2 creates it for you. For the name of the table space, the first character is an "L", followed by seven random characters.

Because implicitly created base table spaces and LOB table spaces use the same buffer pool as specified using DSNZPARM TBSBPOOL, both table spaces use the same page size. Altering a table space buffer pool to a different page size is not supported and requires a DROP TABLESPACE and CREATE TABLESPACE statement. Note that dropping the base table space also drops the auxiliary table and the auxiliary index.

The TBSBPOOL default is 4 KB. This value is generally acceptable for the base table but could be considered small for the LOB table. If you have small LOBs, for instance, a 4,000 byte LOB maximum size, because only one LOB can be stored in one page, larger pages waste disk storage. So, this default is good. If you have large LOBs with recent zSeries® hardware and software, the I/O performance difference between a 4 KB page and a 32 KB page is disappearing with MIDAW, so the default is acceptable. However, if you have large LOBs and no MIDAW, it might be worthwhile assigning them to large pages and large CI

sizes and considering striping for best performance. See also 3.4.2, “Buffer pools and LOB table spaces” on page 56.

Note: Automatic creation of objects only supports partition-by-growth table spaces.

Implicit auxiliary table

Because an auxiliary table is the only table residing in an auxiliary table space, it only contains the information about which LOB column is stored inside. No additional parameters are available, because the CREATE AUXILIARY TABLE only specifies the name of the LOB column to be stored inside.

Implicit auxiliary index

An implicitly created auxiliary index on the auxiliary table has the parameters assigned as reported in Table 3-5.

Table 3-5 Auxiliary index created using automatic object creation

IXNAME	IXSPACE	BUFFERPOOL	CLOSE
IBOOK_BOOK_99TWUYZ	IBOOKRBO	BP0	YES
PIECESIZE	PQTY/SQTY	FREEPAGE	PCTFREE
4194304	-1 / -1	0	10

One more time, the only parameter you can influence is the index buffer pool specified in DSNZPARM IDXBPOOL. All other attributes used for implicitly created objects cannot be changed. The name of the auxiliary index is also 18 characters long. The first character of the name is an 'I'. The next ten characters are the first ten characters of the name of the auxiliary table. The last seven characters are generated randomly.

The most important reason for DB2 coming up with cryptic names for implicitly created objects is to avoid naming collisions with already existing objects. You should be aware of these conventions so that you can plan for any consequences caused by your existing naming conventions, such as SMS ACS routine refinement and so forth.

3.1.2 Using CURRENT RULES STD

CURRENT RULES is a data server register where you can specify whether certain SQL statements are executed in accordance with DB2 rules or the rules of the SQL standard. The valid values are 'DB2' and 'STD'. If the server is not the local DB2, the initial value of the register is 'DB2'. Otherwise, the initial value is the same as the value of the SQLRULES bind option. You can change the value of the register by executing the statement SET CURRENT RULES. See the *DB2 UDB for z/OS Version 8 SQL Reference*, SC18-7426, for details on the scope of CURRENT RULES.

In this section, we examine the impact on CREATE and DROP when using the CURRENT RULES STD setting. In general, we recommend using CURRENT RULES DB2, because this allows you more flexibility in setting up the LOB environment more appropriately to your needs and in complying with naming standards.

CREATE necessary LOB objects

If you specify CURRENT RULES DB2, you have to create all required objects by yourself. You can then specify all the parameters mentioned in “Creating the LOB table space” on page 34, and in the following sections.

If you specify CURRENT RULES STD, DB2 creates all the needed auxiliary objects at the time it processes the CREATE statement for your base table.

Note that you do not have to explicitly specify a ROWID column, DB2 creates it for you, regardless if you use STD or DB2 as a value for CURRENT RULES special register. However, CURRENT RULES STD implies that DB2 creates an index for a ROWID column that is defined with GENERATED BY DEFAULT.

DB2 creates the LOB table space, the auxiliary table, and the auxiliary index for you. If the auxiliary objects defined in Example 3-1 on page 25 were created using CURRENT RULES STD, they would look like Table 3-6 for the LOB table space and Table 3-7 for the auxiliary index.

Table 3-6 LOB table space created using CURRENT RULES STD

TSNAME	BUFFERPOOL	LOCKSIZE	LOG	CLOSE
L9WY757H	BP32K	ANY	YES	YES
DSSIZE	PQTY/SQTY	FREEPAGE	PCTFREE	GBPCACHE
4 GB	3 / -1	0	0	

The name of the LOB table space is an 8-character string every time DB2 creates it for you. The first character is an L, followed by seven random characters. The auxiliary table is created as PAOLO.CLOB_DOCUM1LXA0LID. The table name of the auxiliary table is an 18-character string. The first five characters of the name are the first five characters of the name of the base table. The second five characters are the first five characters of the name of the LOB column. The last eight characters are randomly generated. If a base table name or a LOB column name is fewer than five characters, DB2 adds underscore characters to the name to pad it to a length of five characters.

The buffer pool for the auxiliary table space is the default buffer pool for user data defined in your system for 32 KB data pages.

Table 3-7 Auxiliary index created using CURRENT RULES STD

IXNAME	IXSPACE	BUFFERPOOL	CLOSE
ICLOB_DOCUM9WBL NG3	ICLOBRDO	BP0	YES
PIECESIZE	PQTY/SQTY	FREEPAGE	PCTFREE
4194304	-1 / -1	0	10

The name of the auxiliary index is also 18 characters long. The first character of the name is an I. The next ten characters are the first ten characters of the name of the auxiliary table. The last seven characters are generated randomly. The index has the COPY NO attribute, therefore, full image copies and recover utilities are not allowed.

You can also modify an existing table for holding a LOB column using the ADD column option in the ALTER TABLE statement. First, you ADD the ROWID column to your designated base table, then, at the time you ADD the appropriate LOB column with the ADD LOB column statement, DB2 also creates all the needed auxiliary objects, if CURRENT RULES STD is chosen. If CURRENT RULES DB2 is not chosen, you have to create the objects on your own.

DROP base object

Whenever you drop a base table or a base table space, the associated objects are dropped depending on the way they were created. If you have created the auxiliary objects using CURRENT RULES STD, the LOB table space is implicitly dropped when you drop the base table or the base table space. The auxiliary table and the auxiliary index are dropped as well. Dropping the LOB table space or the auxiliary table is not allowed if they were created using CURRENT RULES STD.

If you have created the auxiliary objects by yourself, only the auxiliary table and the auxiliary index are dropped when you either drop the base table space or the base table. The LOB table space remains in your system.

Note that dropping the auxiliary table space is not allowed when it is used to store LOB data. Dropping the auxiliary table is allowed.

3.1.3 Manual creation of objects

DB2 still allows you to create all of the necessary objects for a LOB environment on your own.

Creating the base table

The first step is creating the base table in a table space already existing in the same database where the LOB table spaces are stored. There are no special suggestions for a base table space: it is just a normal table space. But we recommend that you have only one base table in a base table space. This simplifies your recovery procedures by dealing independently with each LOB table space.

The base table for our example is created with the DDL statement reported in Example 3-3.

Example 3-3 DDL for a base table for manual object creation

```
CREATE TABLE BOOK_BASE_TABLE
( BOOK_NO          CHAR(10)          NOT NULL WITH DEFAULT
, DESCRIPTION      CHAR(32)          NOT NULL WITH DEFAULT
, BOOK_TEXT        CLOB(500M)        NOT NULL
, BOOK_COVER       BLOB(1M)          NOT NULL )
IN LOBDB.BASETS;
```

Note that a ROWID column is omitted and created as a hidden column by DB2.

A few more details about the base table

DB2 currently requires that a ROWID column is included in tables which have one or more LOB columns. The additional column containing the data type ROWID simply contains unique values related to the auxiliary tables in the LOB table spaces. DB2 generates a hidden ROWID column if a ROWID column is not explicitly specified, as shown in Example 3-3. This column is not included in the result set of a SELECT * from a base table, but by selecting the column explicitly by name, you can retrieve its content.

If DB2 detects a LOB column in your CREATE TABLE statement, the definition of a ROWID column is added implicitly as shown in Example 3-4 on page 32. This is called *ROWID transparency* and was introduced in DB2 V8.

Example 3-4 Catalog description for a hidden ROWID for LOBs

NAME	COLTYPE	HIDDEN	LENGTH	UPDATES	DEFAULT
DOCUMENT_NR	CHAR	N	10	Y	Y
DESCRIPTION	CHAR	N	32	Y	Y
DOCUMENT	BLOB	N	4	Y	N
DB2_GENERATED_ROWID_FOR_LOBS	ROWID	P	17	N	A

DB2 creates the column with a name of DB2_GENERATED_ROWID_FOR_LOBS nn . DB2 appends nn only if the column name already exists in the table, replacing nn with 00 and incrementing by 1 until the name is unique within the row. The implicitly added column is appended to the end of the row after all of the other explicitly defined columns.

For the DB2-generated column containing the ROWID, the value of 'P' for the attribute 'HIDDEN' indicates that the ROWID column is not visible in SQL statements except for explicit reference by column name. Updates for that specific column are not allowed, and it is also created with the GENERATED ALWAYS clause.

Note: The ROWID transparency enhancement includes the capability of hiding the ROWID column from DML and DDL. This way, applications running on other platforms that do not have a ROWID data type can avoid the special code to handle ROWID and use the same code path for all platforms.

A new message warns about possible conflicting specifications in the definition:

```
DSNT408I  SQLCODE = -857, ERROR:  CONFLICTING OPTIONS HIDDEN ROWID HAVE BEEN
SPECIFIED
```

The data type ROWID is stored as a VARCHAR (17) column in the base table. It is implicitly one part of the unique index for each auxiliary table containing the LOB columns in the LOB table spaces, where the LOB columns are stored.

Even if a base table contains more than one LOB column, only one ROWID column is needed. So, all LOB columns in one row are associated with the same ROWID value. Regarding this procedure, the ROWID column is a unique and permanent identifier for each row in the base table. A ROWID is not the same as a record identifier (RID), which is internally used by DB2 to reflect the position of a row in a table. But you can find the RID as a part of the externalized ROWID when you select the ROWID column.

There are three ways of assigning a ROWID to a base table, and these are described in more detail in 3.2, "Defining ROWIDs" on page 39.

In case a ROWID is generated by DB2 at the time when you insert your data, you might need to immediately determine the value that has been generated and inserted into the table for you. The INSERT with SELECT statement provides this capability, enhancing the usability and power of SQL. The associated benefits include reduced network costs and simplified procedural logic in stored procedures, because you are decreasing the number of SQL calls from within your application. See Example 3-5 on page 33 for an example of how to retrieve a generated ROWID value during INSERT processing. Note that the example is based on the DDL provided in Example 3-3 on page 31.

Example 3-5 Retrieving a generated ROWID value at INSERT time

```
SELECT DB2_GENERATED_ROWID_FOR_LOBS FROM FINAL TABLE
  (INSERT INTO BOOK_BASE_TABLE
   (BOOK_NO, DESCRIPTION)
   VALUES
   ('SG24-7270-00', 'LOBs - Stronger and Faster'));
```

If there is a requirement in your application to retrieve the rows in the same sequence that they are inserted, the application can use the INPUT SEQUENCE keywords with the ORDER BY clause of the SELECT statement. For more detailed information about this topic, refer to *DB2 UDB for z/OS Version 8: Everything You Ever Wanted to Know, ... and More*, SG24-6079.

In the table definitions of the new data types CLOB (500 MB) and BLOB (1 MB) in Example 3-3 on page 31, DB2 implicitly puts two LOB indicators into the base table definition. Only indicator columns are stored in the base table in place of the actual LOB columns.

Note that the only supported default value for a LOB column is NULL.

For CLOBs, you can also specify the parameters FOR SCBS, MIXED, or BIT DATA. A CCSID EBCDIC or ASCII can also be specified for CLOBs. For BLOBs and DBCLOBs, this is not supported, because BLOBs contain binary data and DBCLOBs have a graphic CCSID associated with them.

A LOB indicator for a LOB column consists of six bytes, and it provides useful information about the stored LOB to DB2 when it accesses the column. The LOB indicators are stored like VARCHAR (4) columns, resulting in a total of 6 bytes, including the length field. Figure 3-1 illustrates the catalog information for the created table.

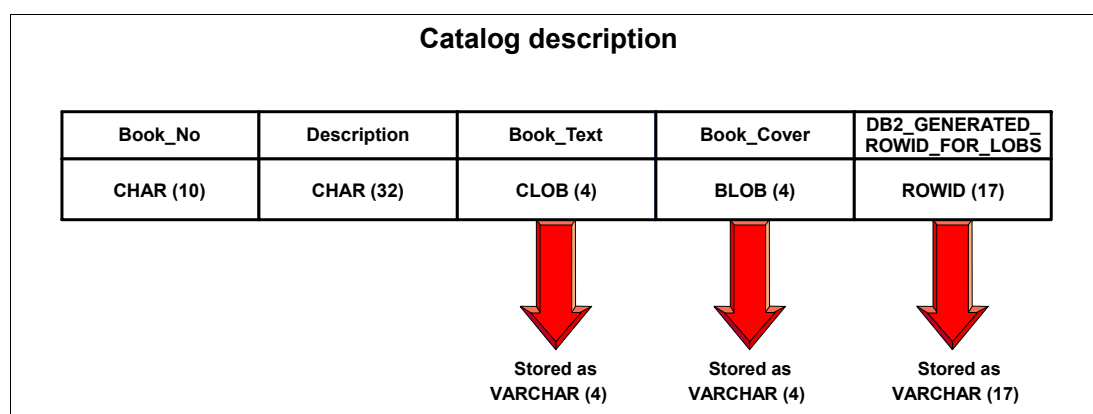


Figure 3-1 Catalog description for table BOOK_BASE_TABLE

The LOB indicator bytes are made up of a 2-byte length field, a 2-byte flag, and a 2-byte string containing the number of the currently stored version of the LOB. The flag bytes contain a NULL flag, which has the information about a NULL, or even a NOT NULL value assigned to a specific LOB value. Further information retrieved from the flag bytes is the zero length flag, which indicates that a LOB column does not contain any data. Invalid LOB values are also marked invalid using the flag bytes. By referring to this information, DB2 does not have to read the auxiliary table when any of these conditions are true. The remaining two bytes contain the current version of the LOB value. This is stored to detect mismatch situations between the base table and the auxiliary table. For further information about possible mismatches, see 6.12, “CHECK LOB” on page 204.

See Example 3-6 for a brief description of the content of a LOB indicator column.

Example 3-6 Content of LOB indicator columns

Length (Hex)	Flags (Binary)	Version (Hex)
00 04	1110 0000 0000 0000	00 02

At this stage, we have only created the base table using manual object creation. Trying to access the base table now, by using SELECT, UPDATE, INSERT, or DELETE, results in DB2 issuing SQLCODE -747. Access to the base table is not allowed, because the base table is marked as incomplete if at least one dependent object is not defined.

The information when a table definition is incomplete is stored in column TABLESTATUS in SYSIBM.SYSTABLES. A value of 'L' indicates a missing auxiliary index or auxiliary table for a LOB column. When the content of the value is 'R', you have used the GENERATED BY DEFAULT clause for your ROWID column and have not yet created the required single column unique index on that particular column in your base table. A value of blank states a complete table definition.

Note: You cannot access the base table using SQL until all dependent objects (LOB table space, auxiliary table, and the auxiliary index) are defined.

Creating the LOB table space

After the base table is created, we define the LOB table space containing the LOB column. In our example, we need two LOB table spaces, one for the CLOB column and the other one for the BLOB column.

Both LOB table spaces can be created using the sample statement shown in Example 3-7.

Example 3-7 DDL for a LOB table space

```
CREATE LOB TABLESPACE BLOBATS1
IN LOBDB
USING STOGROUP          BLOBSGTS
      PRIQTY             1000000
      SECQTY              1000000
NOT LOGGED
LOCKSIZE                LOB
BUFFERPOOL              BP32K
DSSIZE                  64G
GBPCACHE                SYSTEM;
```

Note: The table space containing the base table has to be in the same database as every associated LOB table space. If the base table space has the NOT LOGGED attribute, the same is mandatory for the associated auxiliary table space. It is overridden by DB2 to NOT LOGGED if specified otherwise.

The syntax for creating the auxiliary table space in DB2 9 has changed from 'LOG YES' to 'LOGGED' and from 'LOG NO' to 'NOT LOGGED'. For compatibility reasons, the syntax used in former versions before DB2 9 is still supported.

The keyword LOB tells DB2 to create the table space with the new format. You cannot store LOB values in any other than LOB table spaces (for instance, the generic LARGE table space). Specifying the free space parameter (FREESPACE and PCTFREE) has no influence with LOBs.

The second LOB table space holding the CLOB column is created in a similar way.

Every LOB column needs its own LOB table space. Partitioning of LOB table spaces is not allowed, but they are divided in pagesets in accordance with the partitioned base table definition.

Compression is not supported for LOB table spaces. For a more detailed description of how the data is stored in a LOB table space, see 3.6, “Physical layout of LOBs” on page 62.

Note: You cannot create a LOB table space inside a work file database.

Creating the auxiliary table

The third step is creating an auxiliary table, holding the data of a LOB column. There can only be one auxiliary table per LOB table space.

The DDL for creating an auxiliary table in a LOB table space is shown in Example 3-8.

Example 3-8 DDL for an auxiliary table

```
CREATE AUXILIARY TABLE BLOB_AUX_TABLE_1
  IN              LOBDB.BLOBATS1
  STORES          BOOK_BASE_TABLE
  COLUMN          BOOK_COVER;
```

This statement creates the auxiliary table BLOB_AUX_TABLE_1 in the LOB table space created in “Creating the LOB table space” on page 34. The other auxiliary table storing the CLOB column is created in a similar way.

There is no need to specify column names or column types for auxiliary tables. Using the STORES clause tells DB2 what column of which base table you want to store in the created auxiliary table. The associated table consists of only one column, the LOB column.

Note: A LOB table space and its associated base table space must be stored in the same database. Otherwise, SQLCODE -764 is issued.

If the referenced base table is partitioned, there must be a LOB table space and an auxiliary table for each LOB column in each partition of the base table. In this case, Example 3-9 shows sample DDL for creating the auxiliary table.

Example 3-9 DDL for an auxiliary table containing data of one base table partition

```
CREATE AUXILIARY TABLE BLOB_AUX_TABLE_1
  IN              LOBDB.BLOBATS1
  STORES          BOOK_BASE_TABLE
  COLUMN          BOOK_COVER
  PART            n;
```

The PART clause indicates which partition’s BOOK_COVER column you want to store in this auxiliary table, where *n* is the number of the partition.

FIELDPROCs, EDITPROCs, VALIDPROCs, and check constraints cannot be defined on LOB columns.

How DB2 locates the assigned LOB values

We have only defined one column for the auxiliary table, but DB2 requires two more columns for the table: a column containing the ROWIDs of the base table and one column storing the current version number of the LOB. We report the information stored in the catalog for the three columns of the auxiliary table in Figure 3-2. Using this redundant data, DB2 is able to quickly locate rows in the auxiliary table. In order to do so, we need to define another and the last new object: the auxiliary index.

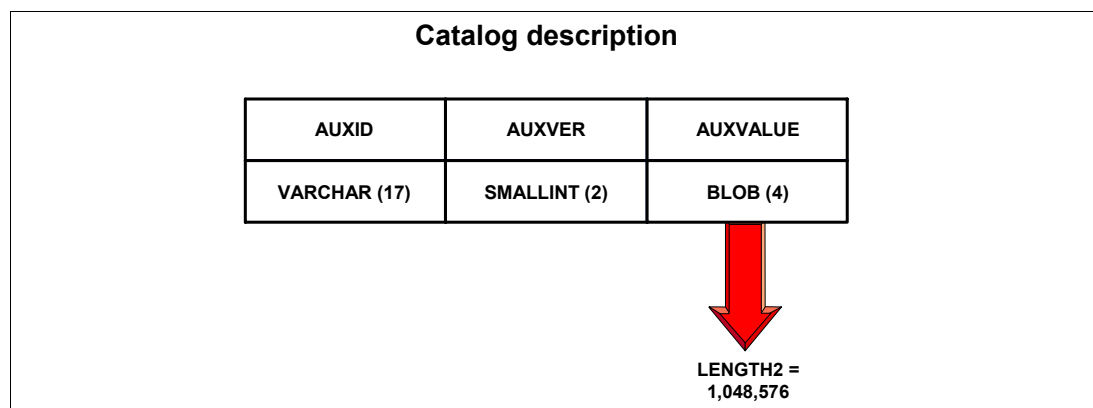


Figure 3-2 *SYSIBM.SYSCOLUMNS* contents for *TBNAME= BOOK_AUX_TABLE*

The auxiliary index

The last step when creating LOBs is to define an auxiliary index for the LOB table space, as shown in Example 3-10. An auxiliary table must have exactly one index, and there cannot be any additional column in the index.

Example 3-10 *DDL for an auxiliary Index*

```
CREATE UNIQUE INDEX LOBDB.BLOBAIX1
ON      BLOB_AUX_TABLE_1
USING   STOGROUP      BLOBSGIX
        PRIQTY        100
        SECQTY        10;
```

No index columns are defined within this index, because DB2 automatically creates the key definition. The index definition consists of a two key value: The 17-byte system generated ROWID stored as VARCHAR, and a 2-byte version of the LOB stored as SMALLINT, for 21 bytes in total (including the 2-byte length field for VARCHAR columns). No index keys can be defined. If you even try to specify a key column, DB2 issues SQLCODE -767, missing or invalid column specification for an index. Figure 3-3 on page 37 shows the key columns of an auxiliary index.

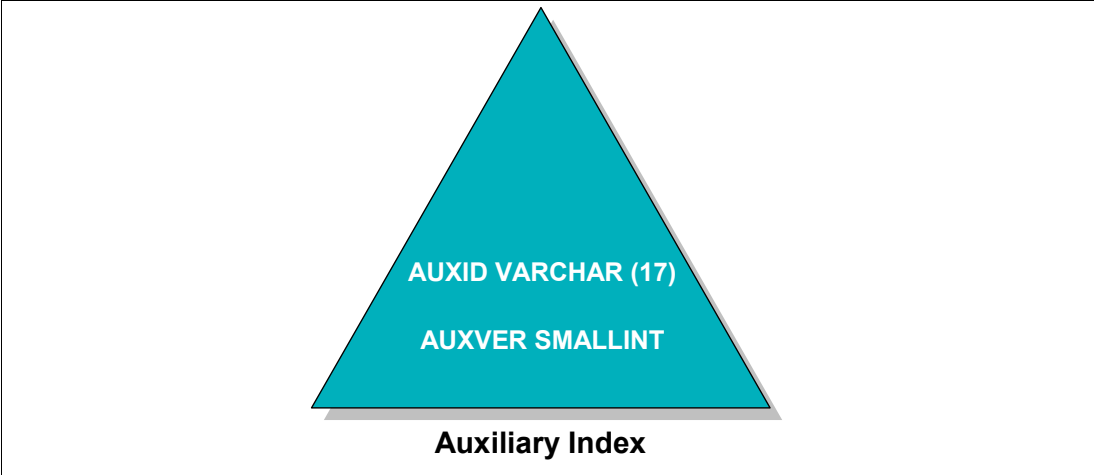


Figure 3-3 Index keys for an auxiliary index

Note: You cannot define an index on a LOB column.

DB2 V7 uses the index on the auxiliary table to locate a LOB value for a particular row containing a LOB within the base table.

An index defined on an auxiliary table is automatically defined as a unique index, fed by a ROWID from the base table. The buffer pool that you might want to assign to this index does not need any special considerations, because the index only contains two relatively small columns.

Displaying LOB objects

For our base table containing one BLOB column and one CLOB column, the display database looks like Example 3-11.

Example 3-11 Displaying a database for LOBs

```

DSNT360I  -DB9B *****
DSNT361I  -DB9B *   DISPLAY DATABASE SUMMARY
              *   GLOBAL
DSNT360I  -DB9B *****
DSNT362I  -DB9B DATABASE = LOBDB  STATUS = RW
              DBD LENGTH = 4028
DSNT397I  -DB9B
NAME      TYPE PART  STATUS          PHYERRLO PHYERRHI CATALOG  PIECE
-----
BASETS    TS          RW
BLOBATS1  LS          RW
CLOBATS1  LS          RW
BLOBAIX1  IX          RW
CLOBAIX1  IX          RW
*****  DISPLAY OF DATABASE LOBDB      ENDED      *****

```

3.1.4 Adding a LOB column to an existing table

Another possible situation is the need for adding a particular LOB column to an already existing table; this table becomes the base table for your LOB columns.

Before a LOB column can be added, we have to be sure that a column of data type ROWID is already in the table. If it is not, we have to add it using the statement reported in Example 3-12.

Example 3-12 Adding a ROWID column

```
ALTER TABLE NEW_LOB_TABLE
ADD ROW_ID ROWID NOT NULL GENERATED ALWAYS;
```

A table can only have one ROWID column, and you cannot add a ROWID column to a created temporary table.

Important: If you add a LOB column prior to a ROWID column, DB2 generates a hidden ROWID column.

As described in 3.2, “Defining ROWIDs” on page 39, we recommend that you always generate ROWIDs by using DB2’s mechanism. Specifying GENERATED BY DEFAULT generates a ROWID only if no value for the ROWID column is provided while inserting into the table. If a value for a ROWID column is provided, DB2 takes it and inserts the value into the base table. By providing a value for a ROWID column, for instance, when moving data across two DB2 subsystems, it is unlikely, but it might happen, that DB2 generates a ROWID that is already stored in the table. Because the GENERATED BY DEFAULT clause always requires a unique index on that column, an insert with a DB2-generated ROWID value can result in SQLCODE -803. In this case, it is your responsibility to resolve the duplication.

Once a column of data type ROWID is added, you can proceed with creating the LOB columns using the alter table statement shown in Example 3-13.

Example 3-13 Adding a LOB column

```
ALTER TABLE NEW_LOB_TABLE
ADD PICTURE BLOB(1M) NOT NULL WITH DEFAULT;
```

Adding a LOB column is not allowed for created temporary tables. The same is true when adding ROWID columns.

Instead of specifying BLOB (1M), such as in this example, you can use the ALTER to define any other possible LOB column type to DB2.

After you have added the designated LOB columns to the base table, you should continue with the same actions shown earlier starting with “Creating the auxiliary table” on page 35. Remember that only one ROWID column is needed, even if you want to add more than one LOB column to the base table.

Note: When you add one or more LOB columns to the base table, the table is marked incomplete until all dependent objects are created.

We recommend setting up your LOB environment using manual creation of all necessary objects if you want to influence the parameters that are involved. For most other needs, use automatic object creation to reduce the overhead of creating all the necessary objects by yourself since it fits most needs.

3.2 Defining ROWIDs

The ROWID data type (and column) definition was introduced with DB2 V6 to uniquely and permanently identify a row in a table. To understand the role of the ROWID, and why DB2 creates a value for a ROWID column whenever a row is inserted in a table containing this new data type, you must first understand the overall picture of the DB2 objects involved in supporting LOBs, as illustrated in Figure 3-4.

LOB data is contained in a LOB column, which is conceptually part of the base table, but it is physically stored in a separate table. Because it is not part of the base table, it is called a LOB table or auxiliary table. The auxiliary table resides in a separate LOB table space.

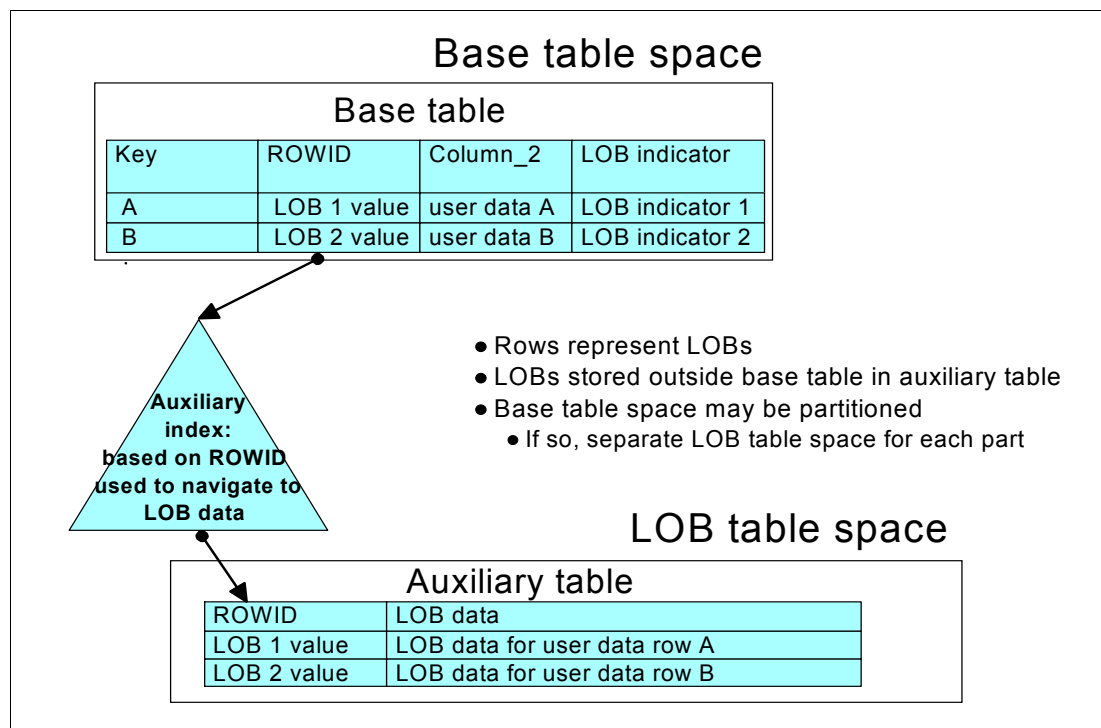


Figure 3-4 LOB structure

A base table can be associated with many LOB or auxiliary columns of different types and lengths. Each auxiliary column is stored in its own auxiliary LOB table in its own LOB table space. An auxiliary index must be created on every auxiliary table before it can be used. To create a base table that contains a LOB column, you must define a ROWID column. The ROWID acts as, but it is not, a bidirectional pointer to the LOB data associated with the particular row: the LOB column values are associated with the proper base table row in both directions using the base table row's ROWID value. The auxiliary index, whose key is based on the ROWID, is used to navigate to LOB data associated with the row.

If a base table that contains LOB data is partitioned, you create a separate LOB table space and auxiliary table for each LOB column in each partition. A LOB table space can have a page size of 4, 8, 16, or 32 KB. Because the length of a LOB can exceed 32 KB, it is clear that a LOB can span physical pages, and because there is only one row per page, with LOBs, rows can span pages. LOB pages only contain one LOB value, or row.

Starting from DB2 V8, users have the choice of defining a ROWID explicitly or leaving it to be defined implicitly by DB2 during creation of the table. This functionality is frequently referred to as *ROWID transparency* feature.

3.2.1 Creating the ROWID column

There are three different ways of actually defining a column to be a ROWID data type in a CREATE TABLE statement:

- ▶ Omitting an explicit ROWID definition while a LOB column is contained in your CREATE TABLE statement
- ▶ Using **COLNAME ROWID GENERATED ALWAYS**
- ▶ Using **COLNAME ROWID GENERATED BY DEFAULT**

If you omit an explicit ROWID definition while a LOB column is present in your CREATE TABLE statement, DB2 creates a ROWID column using GENERATED ALWAYS method for you.

Using the GENERATED ALWAYS keyword, DB2 always generates a ROWID when inserting a row. Applications and users are not allowed to insert a ROWID.

If you use GENERATED BY DEFAULT, users and applications can supply a value for a ROWID column as long as the value was previously generated by DB2 and a unique, single column index that exists on the ROWID column. DB2 checks that the value you are going to insert is a valid ROWID. It is not sufficient to provide unique numbers yourself. You should only use this parameter when inserting data from another table for the purpose of moving data.

The recommended usage is GENERATED ALWAYS, either explicitly specified or implicitly by omitting ROWID from the CREATE TABLE statement.

As mentioned, you have to create a unique index on the ROWID column when you specify GENERATED BY DEFAULT.

Make sure that it is not possible to use the GENERATED ALWAYS clause before implementing GENERATED BY DEFAULT, because the additional index on a table can increase your response time for inserting and deleting transactions on the base table. The index is not affected by an UPDATE statement, because the ROWID is not updateable. If you try to update a ROWID column, DB2 issues SQLCODE -151, because the catalog description indicates that this column cannot be updated.

Important: When you specify GENERATED BY DEFAULT for a ROWID column, make sure that a single column unique index exists on your ROWID column. ROWID values can never contain null values, so the ROWID column has to be defined as NOT NULL.

Be aware that a ROWID column implies some restrictions, preventing the values in the column from being manipulated:

- ▶ Users are not allowed to update a ROWID column.
- ▶ Null values cannot be assigned to ROWID columns.
- ▶ EDITPROCs, FIELDPROCs, and CHECK CONSTRAINTs are not allowed for ROWIDs.
- ▶ It is not allowed to load a single partition or a range of partitions if a column of data type ROWID is part of the partitioning key.

A value for ROWID is stored in the base table data page even if all LOB columns are NULL. The ROWID column is stored as a VARCHAR (17) column. The length of a ROWID column as described in the LENGTH column of catalog table SYSCOLUMNS is the internal length, which is 17 bytes. The length as described in the LENGTH2 column of catalog table SYSCOLUMNS is the external length, which is 44 bytes. A value for a ROWID is never subject to character conversion, because it is considered to contain BIT data.

To give you a better understanding of the different occurrences of a ROWID, consider the following scenarios:

Case 1: A new table is created including a ROWID column

First, we have a look at a table where the CREATE TABLE statement already contains a ROWID column (for all examples, it does not matter if the ROWID column is defined using GENERATED ALWAYS or GENERATED BY DEFAULT) as shown in Example 3-14.

Example 3-14 DDL for a table containing a ROWID column

```
CREATE TABLE CUSTOMER
  ( CUSTNO    CHAR(10)    NOT NULL WITH DEFAULT
    , CUSTNAME CHAR(32)    NOT NULL WITH DEFAULT
    , ROW_ID   ROWID      NOT NULL GENERATED ALWAYS );
```

Now you insert a row into the CUSTOMER table, as shown in Example 3-15, without providing the ROWID column, because it is generated by DB2 at insert time.

Example 3-15 Inserting a row in CUSTOMER table

```
INSERT INTO CUSTOMER (CUSTNO, CUSTNAME)
VALUES ('0000001406', 'REGINA RICHARDSON');
```

After a new table is created, all CUSTNOs being inserted are associated with a unique ROWID value. After you insert a row (at this point in time, a ROWID is associated with the inserted row), a SELECT CUSTNO,CUSTNAME,ROW_ID returns the result set shown in Example 3-16.

Example 3-16 ROWID value of a table created with ROWID column

CUSTNO	CUSTNAME	ROW_ID
0000001406	REGINA RICHARDSON	D6DE97EE118FD4252104015C5630010000000004201

The generated ROWID is externalized as a 44 byte value (40 bytes of data plus length fields), but stored as VARCHAR (17).

Example 3-17 shows the output of the DSN1PRNT utility of the HEX values for the ROWID with DB2 9 for z/OS.

Example 3-17 DB2 9 DSN1PRNT of ROWID in hex value

```
002C D6DE97EE 118FD425 2104015C 5630
```

Note that the value 0100000000004201 is generated at SELECT time.

In DB2 9 for z/OS, the value of 002C in our example gives you the offset within the row of the variable length column containing the ROWID. It varies, depending on the other columns defined in your table, since DB2 9 introduces Reordered Row Format (RRF), a performance improvement for access to data in tables that contain columns of varying length. The format in

which the row is stored in the table is changed from V8 to optimize column location for data retrieval and for predicate evaluation.

In DB2 V8, the output of DSN1PRNT is similar to the output in Example 3-18.

Example 3-18 DB2 V8 DSN1PRNT of ROWID in hex value

000E D6DE97EE 118FD425 2104015C 5630

In DB2 V8, the value *000E* declares the length of the ROWID column, which is 000E in HEX and 14 in decimal.

Comparing the information stored in the DB2 catalog for DB2 V8 and DB2 9, ROWIDs are stored as VARCHAR (17) columns: in both versions of DB2, there are three bytes left for future extensions of ROWIDs.

Case 2: Adding a ROWID column using ALTER TABLE

Things look different if a ROWID column is added when a table already contains many rows. Consider the table mentioned in Example 3-14 on page 41 without a ROWID column and that the table already contains lots of rows. Let us issue an ALTER TABLE statement as shown in Example 3-19.

Example 3-19 ALTER TABLE adding a ROWID column

ALTER TABLE EXAMPLE ADD ROW_ID ROWID NOT NULL GENERATED ALWAYS;

Note: A ROWID column must be defined using NOT NULL. When you add a ROWID column, the NOT NULL attribute contradicts the normal usage of ALTER. In fact when ALTERing non-ROWID columns, you must specify NOT NULL WITH DEFAULT.

The ALTER TABLE statement does not affect any columns stored in the table, so no ROWID is stored up to now. If you do a SELECT on CUSTNO,CUSTNAME,ROW_ID...FETCH FIRST 1 ROW, you receive the result set of Example 3-20.

Example 3-20 ROWID value of a table where a ROWID column was added

CUSTNO	CUSTNAME	ROW_ID
0000001406	REGINA RICHARDSON	4200000080000106000D020100000000004201

The ROWID now only consists of 38 bytes, compared to the previously mentioned ROWID of 44 bytes.

If you use DSN1PRNT again to look into your table space, you do not find any value for the ROWID column inside your table space. When you select the ROWID value, it is only generated at SELECT time, and stored inside your host variable. You can select the ROWID for a specific row several times, and the value in the ROWID column never changes.

The first time you update the row, the ROWID is physically stored in the table, with the same value delivered to you by DB2 when you selected the row before.

This behavior is the same for DB2 V8 and DB2 9.

When you use DSN1PRNT again after you have performed an UPDATE on the row, in DB2 9 you see the results of Example 3-21 on page 43.

Example 3-21 DB2 9 DSN1PRNT of ROWID in hex value after adding and updating a row

```
002C 42000000 80000106 000D02
```

Again, the value *002C* declares the offset of the ROWID column. If other VARCHAR columns are stored in your table after the ROWID, other offsets follow the *002C* before the ROWID value begins.

You can find the output of DSN1PRNT in DB2 V8 for the same scenario in Example 3-22.

Example 3-22 DB2 V8 DSN1PRNT of ROWID in hex value after adding and updating a row

```
000B 42000000 80000106 000D02
```

In case you use DB2 V8, the value of *000B* declares the length of the ROWID column, which means 11 in a decimal value. Adding the two byte length field, as usual, we now have a ROWID column made of 13 bytes.

So, if an already existing row is updated and no ROWID value is stored for the updated row up to now, the new ROWID is at least 11 bytes plus the two byte length field.

Let us now have a close look at the new rows, which are inserted after the ROWID column was added to the table. All rows that are inserted after the ROWID column was added to the table have a ROWID of length of 14 bytes plus two additional bytes for the length field.

The first row shown in Example 3-23 represents the ROWID value for an *old* row being updated after ROWID was added to the table. It is important to know that this row already had been in the table before the ROWID column was added. The second row shows the ROWID value for a new row inserted after the ROWID column was added.

Example 3-23 ROWID values of updated and inserted columns

CUSTNO	CUSTNAME	ROW_ID
0000001406	MRS. RICHARDSON	4200000080000106000D020100000000004201
0000001423	MR. KLINGEN	647667FA918FDE192104015C56300100000000004202

If a number of *n* rows have been in the table before a ROWID column is added, there is a number of *n* different ROWIDs after ALTER. Each “updated” row existing before the ALTER has its own unique value.

In Example 3-24, you can find the DSN1PRNT output for both ROWID values:

Example 3-24 DB2 9 DSN1PRNT of updated and inserted ROWIDs in hex value

```
002C 42000000 80000106 000D02
002C 647667FA 918FDE19 2104015C 5630
```

In conclusion, it is possible that no ROWID value is stored in the table space even if a ROWID column exists in the table, and you can find two different lengths of ROWIDs even if they are stored in the table, depending on when the ROWID column was added to the table and whether or not the row existed at that time.

Be careful when altering a table to have a ROWID column, because pre-ALTER rows might never have a value that can be used by code that relies on ROWID value to go directly to the row. ROWID should then be selected and stored somewhere and used for future access when contained in a host variable. After CREATE TABLE, INSERT *n* ROWS, ALTER ADD

ROWID, you SELECT ROWIDs and store them, then the application selects data using the ROWIDs retrieved by the previous SELECT.

The output in DB2 V8 is in Example 3-22 on page 43, again preceded by the current length fields *000B* and *000E*.

Example 3-25 DB2 V8 DSN1PRNT of updated and inserted ROWIDs in hex value

```
000B 42000000 80000106 000D02
000E 647667FA 918FDE19 2104015C 5630
```

ROWID is a data type that can be used outside LOBs. ROWID gives you guaranteed unique values. This can be useful for applications and table designs where artificial keys have to be generated by the application to ensure uniqueness of a particular row. Using a ROWID column, DB2 is able to handle this special requirement for you. The ROWID behavior, that the first bytes in general appear to be pseudo-random, has made a ROWID column a solution for a partitioning key, which spreads your data randomly across partitions. However, with the separation of attributes for indexes in DB2 V8 and the introduction of new partition definitions in DB2 9, there are other means to obtain similar results. Using a ROWID as a partitioning key is not a good idea when you have to process your data in the order of another key value.

3.3 LOBs and LOG activity

Logging LOB data can become an issue for your DB2 subsystem. The amount of data to be logged grows according to the actual size of the manipulated LOBs and the number of LOBs being processed in parallel. Prior to DB2 9, LOB auxiliary table spaces were the only table spaces which allowed the LOG YES\NO option, and this option was disabled for LOBs exceeding 1 GB. This logging attribute for the LOB table space was therefore independent of the implied LOG YES attribute of the correspondent base table space.

Two major changes are introduced by DB2 9 in new function mode:

- ▶ LOGGED and NOT LOGGED attributes for all table spaces
- ▶ Logging for all LOB sizes, up to the maximum of 2 GB -1

3.3.1 LOGGED and NOT LOGGED attributes

In DB2 9, the CREATE TABLESPACE syntax is changed from LOG YES/NO to LOGGED/NOT LOGGED. The old syntax is still supported for LOB tables spaces for compatibility reasons. For more information:

- ▶ LOGGED

Specifying LOGGED for a LOB table space tells DB2 to log almost all data manipulations on LOB columns stored in the associated LOB table space, except delete operations. When you delete a LOB value, no LOB data is written to the log, only LOB system pages are written to the log data sets, because a LOB delete internally is translated into only deallocation of the pages where the LOB is stored. The deallocation of a LOB is flagged in the space map pages for all pages containing the LOB information, including the LOB map pages. For more information about the structure of a LOB table space, refer to 3.6, “Physical layout of LOBs” on page 62.

When you insert a LOB, DB2 writes the entire LOB value to the log. The entire LOB is also written to the log even when you update a LOB value, because updates consist of one singleton delete (where DB2 does not write data in the log) and one insert into the auxiliary table.

Depending on the size of your LOBs and the frequency with which you regularly insert or update them, the data to be logged can grow rapidly. When you plan to use the LOGGED option, make sure that writing the redo log records does not become a critical factor!

If your logging becomes I/O-constrained, you can benefit from striping the log data sets.

► **NOT LOGGED**

To prevent your system from the possible overhead caused by logging large amounts of LOB data, DB2 9 allows you to turn off logging for all table spaces, including LOB table spaces, base, and auxiliary. NOT LOGGED for the LOB table space tells the DB2 subsystem to suppress writing redo log records for every LOB column in your LOB table space. The force at commit protocol ensures that LOB values persist after a unit of work is committed, because NOT LOGGED LOB values are written at COMMIT. LOB data associated with a LOB table space defined with LOGGED option is written to disk, like for other data, when buffer pool thresholds are reached.

LOGGED is the default value when you create your LOB table space.

For information about the logging impact when running the Load utility, see 6.3, “LOAD” on page 171.

Logging combinations of base table space and LOB table spaces

With DB2 V8, you had the LOB base table logged and the LOB table space possibly not logged. In DB2 9, however, a LOB table space logging attribute is not completely independent of its associated base table logging attribute. It is a requirement that if the base table space has the NOT LOGGED logging attribute, all associated LOB table spaces must also have the NOT LOGGED logging attribute.

Therefore, the logging options for the base table space and the LOB table spaces are somewhat connected, as shown in this paragraph.

If the base table space has the LOGGED attribute, the logging attribute of the LOB table space continues to be independent of the base table space. In this case, the LOB table space can have either a LOGGED or a NOT LOGGED attribute. Furthermore, the LOB table space logging attribute can be altered without restriction. This was also true previously to DB2 9.

However, if the base table space has a NOT LOGGED attribute, the LOB table space must also have a NOT LOGGED attribute. The LOB table space can acquire the NOT LOGGED attribute either independently or as linked to its associated base table space's attribute. When the base table space has a NOT LOGGED attribute, the LOB table space attribute might not be altered to LOGGED (DB2 issues SQLCODE -763, SQLSTATE 560A1), regardless of how it was acquired.

To alter the LOGGED attribute, you use the ALTER TABLESPACE command. The detailed syntax is in *DB2 Version 9.1 for z/OS SQL Reference*, SC18-9854.

Altering LOGGED to NOT LOGGED

When a base table space logging attribute is altered from LOGGED to NOT LOGGED, all associated LOB table spaces with the LOGGED logging attribute are also implicitly altered to force their logging attribute to NOT LOGGED to match the base table space. When a LOB table space logging attribute is implicitly altered in this way, its logging attribute is said to be linked to the base table logging attribute.

It remains NOT LOGGED and linked to the base table space logging attribute until the base table space logging attribute is altered back to LOGGED.

Altering NOT LOGGED to LOGGED

A LOB table space containing LOBs can have its logging attribute explicitly altered from NOT LOGGED to LOGGED only if its logging attribute is not linked to the base table logging attribute (SQLCODE -763, SQLSTATE 560A1).

A LOB table space can have its logging attribute implicitly changed from NOT LOGGED to LOGGED when it is linked to the base table logging attribute.

Should the base table space logging attribute be subsequently altered back to LOGGED, all linked LOB table spaces are also implicitly altered to return their attribute to LOGGED. At this point, the table spaces are no longer linked. When a LOB table space logging attribute is linked to its base table space logging attribute, the link can also be broken by explicitly altering the LOB table space attribute to NOT LOGGED, even though it already has been implicitly given the NOT LOGGED attribute when its base table space attribute was altered to NOT LOGGED. While it might seem redundant to alter a NOT LOGGED LOB table space to NOT LOGGED, this is provided to allow you to break the link between the logging attributes of a LOB table space and its base table space.

If the LOG column of the SYSIBM.SYSTABLESPACE catalog table record for a LOB table space has the value of 'X', it means that the logging attributes of the LOB table space and its base table space are linked, and that the logging attribute of both table spaces is NOT LOGGED. To break the link, alter the base table space attribute back to LOGGED, which results in both table spaces' logging attribute being changed back to LOGGED.

Table 3-8 shows the progression of the DB2 catalog SYSIBM.SYSTABLESPACE LOG column values through a set of ALTER TABLESPACE statements that alter the logging attribute of a base table space and two LOB table spaces associated with a table in the base table space. The values are:

- ▶ Y - Logged
- ▶ N - Not Logged
- ▶ X - Not Logged, linked to base

In our scenario, we assume that initially all table spaces have been created using the LOGGED attribute.

Table 3-8 LOG column values scenarios

ALTER	LOGGED attribute			
	Base table space	LOB table space 1	LOB table space 2	Notes
Initially all LOGGED	Y	Y	Y	
LOB 2 NOT LOGGED	Y	Y	N	
Base to NOT LOGGED	N	X	N	LOB 1 linked to base
LOB 2 to LOGGED	N	X	N	Rejected, SQLCODE -763
Base to LOGGED	Y	Y	N	LOB 1 link dissolved
LOB 2 to LOGGED	Y	Y	Y	
Base to NOT LOGGED	N	X	X	LOB 1 and LOB 2 linked to base
Base to logged	Y	Y	Y	LOB 1 and LOB 2 link dissolved
Base to NOT LOGGED	N	X	X	LOB 1 and LOB 2 linked to base
LOB 2 to NOT LOGGED	N	X	N	LOB 2 link dissolved
Base to LOGGED	Y	Y	N	LOB 1 link dissolved

If the table space is opened for update while it has the NOT LOGGED attribute, the table space is placed in the Informational Copy Pending state. The DISPLAY DATABASE ADVISORY command is enhanced to display the Informational Copy Pending state for table spaces in the database as shown in Example 3-26.

Example 3-26 DISPLAY DATABASE sample output

```

DSNT360I  -DB9B *****
DSNT361I  -DB9B *   DISPLAY DATABASE SUMMARY
              *   ADVISORY
DSNT360I  -DB9B *****
DSNT362I  -DB9B      DATABASE = DSN8D91L  STATUS = RW
              DBD LENGTH = 8066
DSNT397I  -DB9B
NAME      TYPE PART  STATUS                PHYERRLO PHYERRHI CATALOG  PIECE
-----
TPIQUQ01  TS   001   RW,AUXW
TPIQUQ01  TS   002   RW,AUXW
TPIQUQ01  TS   003   RW,AUXW
TPIQUQ01  TS   004   RW,ICOPY

```

Other CATALOG tables

The following other DB2 catalog tables are influenced by the change of LOGGED attribute:

► **SYSIBM.SYSLGRNX**

DB2 does not maintain SYSLGRNX entries for NOT LOGGED objects.

► **SYSIBM.SYSCOPY**

The new column LOGGED is added to the SYSIBM.SYSCOPY catalog table. The values can be:

- Y - to indicate the logging attribute is LOGGED.
- N - to indicate the logging attribute is NOT LOGGED.
- blank - to indicate that the row was inserted prior to DB2 9.

For a non-LOB table space or an index space, this is an indication that the logging attribute is LOGGED. No assumptions are made about the logging attribute of LOB table spaces.

Table 3-9 reflects the contents of the ICTYPE and STYPE columns in the SYSIBM.SYSCOPY catalog table.

Table 3-9 SYSIBM.SYSCOPY values for LOGGED attribute changes

Action	LOGGED	ICTYPE	STYPE
CREATE TABLESPACE	Y	C	L
CREATE TABLESPACE	N	C	O
ALTER TABLESPACE	Y	A	L
ALTER TABLESPACE	N	A	O

The LRSN in these SYSIBM.SYSCOPY records reflects the point in the log at which the logging attribute was altered.

When updates on a NOT LOGGED object are rolled back or the thread is canceled prior to commit, the space state of base tables and auxiliary indexes is changed to RECP (Recovery Pending) and the affected pages are added to the LPL list. To remove these objects from the LPL and reset recover pending (RECP), use any of the following:

- The RECOVER utility, to recover either to most recent recoverable point or to a prior image copy.
- LOAD REPLACE or LOAD REPLACE PART, either with an input data set to repopulate the table, or without one so that INSERT can repopulate the table.
- Drop and recreate the table space and repopulate the table.
- Use Delete without a WHERE clause or new TRUNCATE statement (restrictions apply).

For more information about the recovery issues, see 7.2, “Recovery strategies and considerations” on page 219.

See 7.1, “LOBs in the DB2 catalog” on page 212 for more information about the recognition of LOBs in the DB2 catalog.

DB2 rollback without undo log records

Because new pages are allocated while inserting a LOB, they are simply deallocated if DB2 does a rollback. If a LOB is deleted, pages are also simply reallocated as available in the LOB table space. Because updating a LOB means deleting a LOB (deallocating pages) and inserting a new one (allocating new pages), at rollback time, already allocated pages are deallocated and previously deallocated pages are reallocated again. This mechanism is known as Shadow Copy Recovery, see “Shadow Copy Recovery” on page 100.

Because of allocation and deallocation of data pages, no UNDO logs are written for LOBs, regardless of which LOG parameter you use.

Header pages, space map pages, and the new LOB map pages are logged even though NOT LOGGED is specified. They are backed out using the same procedure used for other types of table spaces. The same rule applies if an application terminates abnormally after changing the database. Even in this case, those changes made in the current unit of work are backed out along with all other changes, too.

If NOT LOGGED is specified, DB2 does not log any changes applied to any LOBs in the associated LOB table space. Therefore, recovery of LOB table spaces can be more difficult than recovering any other table space. Let us assume the scenario depicted in Figure 3-5 on page 49.

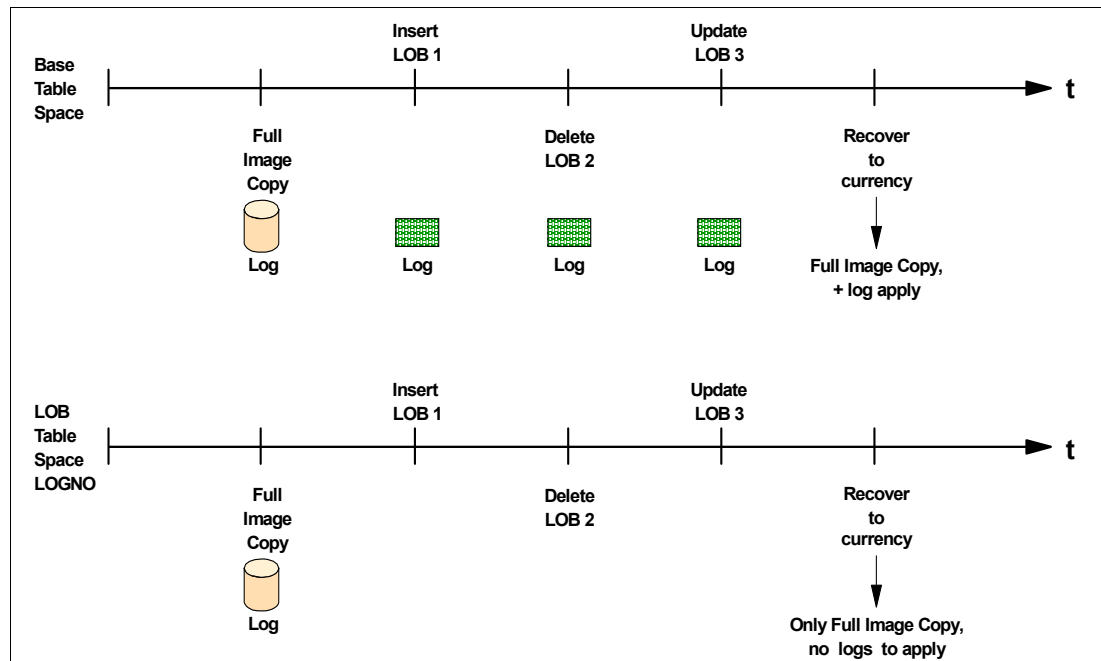


Figure 3-5 Applying changes to a LOB table space created with NOT LOGGED

This can be a DB2 V8 scenario where you have chosen not to log the LOB table space, while the base table is logged anyway. A full Image Copy is taken of a base table space and the associated LOB table spaces. After completion of the full Image Copies, one LOB is inserted, another one is deleted, and a third one is updated. Assume now that you have to recover the base table space and the LOB table space after all updates are done because of a system failure. For the base table, DB2 can apply all necessary changes from the log. Recovering the LOB table space for DB2 is more difficult with a specified NOT LOGGED option.

Inserted LOB

After using the full image copy, DB2 notices that a new LOB was inserted without writing any logs about its value. So the new LOB is marked invalid in the auxiliary table.

Deleted LOB

DB2 knows from the logged system pages what has happened and marks the pages formerly used by LOB number 2 as available. So recovery is possible for deleted LOBs.

Updated LOB

Since UPDATE consists of DELETE and INSERT, the pages are deallocated in the LOB table space. Not having log records to apply for the new LOB value, it is also marked as invalid in the LOB table space.

In our case, the LOB table space is set in a new AUX WARNING (AUXW) state for the inserted and updated LOB values. Trying to access an invalid LOB value results in SQLCODE -904. For more information about recovery scenarios, see 7.2, “Recovery strategies and considerations” on page 219.

If you decide to use the NOT LOGGED option on the base table, make sure that image copies of the base table space and all its auxiliary table spaces you take are in sync.

Recommendation: COPY base table space, COPY YES indexes, and COPY LOB table spaces (and XML table spaces) in the same COPY invocation with SHRLEVEL REFERENCE to ensure that they all share the same recoverable point.

COPY does not allow SHRLEVEL(CHANGE) on NOT LOGGED table spaces.

3.3.2 Logging for all LOB sizes

The limitation that logging was possible only for LOBs under 1 GB is removed with DB2 9. From the very beginning, this limitation was introduced in order to prevent excessive I/O on the LOG data sets.

Customers demand high availability of their data. In order to achieve this goal, all objects must have an option to be recovered to any point in time, thus, making logging a crucial aspect.

To reduce the volume of logging, you can specify NOT LOGGED in your CREATE LOB TABLESPACE statement or alter the table space to the NOT LOGGED attribute. This suppresses writing redo records. There are no UNDO records for LOB updates (except for system pages, space map) even with LOG YES. DB2 always inserts the new LOB value at a different place and deletes the old LOB at commit, marking the old space as free.

To give you an idea of the amount of data being written to the log, look at Example 3-27, which shows the DSN1LOGP output for a LOB insert into a LOGGED table space.

Example 3-27 DSN1LOGP output for logged LOB insert

```
0000893EABE2  URID(0000893EABE2)  LRSN(BF3BF2C96624)
                TYPE(UR CONTROL)  SUBTYPE(BEGIN UR)

0000893EB068  URID(0000893EABE2)  LRSN(BF3BF2C966C8)  DBID(014A)
                OBID(0007)  PAGE(00000007)  TYPE( UNDO REDO )
                SUBTYPE(ALLOC/DEALLOC OF SPACE IN LOB SPACE MAP PAGE)
                CLR(NO)  PROCNAME(DSNODEAL)
```



```

0000893EB1B4 URID(0000893EABE2) LRSN(BF3BF2C966CB) DBID(014A)
              OBID(000B) PAGE(00000003) TYPE( REDO )
              SUBTYPE(TYPE 2 INDEX UPDATE) CLR(NO)
              PROCNAME(DSNKPDGB)

0000893EB2A6 URID(0000893EABE2) LRSN(BF3BF2C966CB) DBID(014A)
              OBID(000B) PAGE(00000003) TYPE( UNDO REDO )
              SUBTYPE(TYPE 2 INDEX UPDATE) CLR(NO)
              PROCNAME(DSNKDLE )

0000893EB321 URID(0000893EABE2) LRSN(BF3BF2C966CE) DBID(014A)
              OBID(0007) PAGE(00000005) TYPE( UNDO REDO )
              SUBTYPE(ALLOC/DEALLOC OF SPACE IN LOB SPACE MAP PAGE)
              CLR(NO) PROCNAME(DSNOALLO)

0000893EB3AF URID(0000893EABE2) LRSN(BF3BF2C966CF) DBID(014A)
              OBID(0007) PAGE(00000006) TYPE( UNDO REDO )
              SUBTYPE(ALLOC/DEALLOC OF SPACE IN LOB SPACE MAP PAGE)
              CLR(NO) PROCNAME(DSNOALLO)

0000893EB41F URID(0000893EABE2) LRSN(BF3BF2C966D0) DBID(014A)
              OBID(0007) PAGE(00000085) TYPE( REDO )
              SUBTYPE(LOB MAP CHANGE) CLR(NO) PROCNAME(DSNOFLMP)

0000893EB4E8 URID(0000893EABE2) LRSN(BF3BF2C966D1) DBID(014A)
              OBID(0007) PAGE(00000085) TYPE( REDO )
              SUBTYPE(LOB DATA PAGE CHANGE) CLR(NO)
              PROCNAME(DSNOLINS)
              .
              .
              .
000089452DC8 URID(0000893EABE2) LRSN(BF3BF2C9674B) DBID(014A)
              OBID(0007) PAGE(000000EB) TYPE( REDO )
              SUBTYPE(LOB DATA PAGE CHANGE) CLR(NO)
              PROCNAME(DSNOLINS)

0000894601CB URID(0000893EABE2) LRSN(BF3BF2C96FE9)
              TYPE(UR CONTROL) SUBTYPE(BEGIN COMMIT1)

000089460235 URID(0000893EABE2) LRSN(BF3BF2C96FED)
              TYPE(UR CONTROL) SUBTYPE(PHASE 1 TO 2)

000089460269 URID(0000893EABE2) LRSN(BF3BF2C97061)
              TYPE(UR CONTROL) SUBTYPE(END COMMIT2)

```

On the other hand, the NOT LOGGED table space update looks as shown in Example 3-28. You can also see the pageset indicated as Information Copy pending.

Example 3-28 DSN1LOGP output for not logged LOB insert

```

0000894FAD8D URID(0000894FAD8D) LRSN(BF3BF3D57D02)
              TYPE(UR CONTROL) SUBTYPE(BEGIN UR)

0000894FB1BA URID(0000894FAD8D) LRSN(BF3BF3D5900E) DBID(014A)
              OBID(0007) TYPE( REDO )

```

SUBTYPE(DBE TABLE WITH EXCEPTION DATA)

REDO: DSN8D91L.DSN8S91L
ICOPY

0000894FB239	URID(0000894FAD8D) LRSN(BF3BF3D59010) DBID(014A) OBID(0007) PAGE(00000005) TYPE(UNDO REDO) SUBTYPE(ALLOC/DEALLOC OF SPACE IN LOB SPACE MAP PAGE) CLR(NO) PROCNAME(DSNODEAL)
0000894FB2C7	URID(0000894FAD8D) LRSN(BF3BF3D59034) DBID(014A) OBID(0007) PAGE(00000001) TYPE(UNDO REDO) SUBTYPE(LOB HIGH LEVEL SPACE MAP PAGE UPDATE) CLR(NO) PROCNAME(DSNODEAL)
0000894FB307	URID(0000894FAD8D) LRSN(BF3BF3D5905A) DBID(014A) OBID(0007) PAGE(00000006) TYPE(UNDO REDO) SUBTYPE(ALLOC/DEALLOC OF SPACE IN LOB SPACE MAP PAGE) CLR(NO) PROCNAME(DSNODEAL)
0000894FB421	URID(0000894FAD8D) LRSN(BF3BF3D59061) DBID(014A) OBID(0007) PAGE(00000005) TYPE(UNDO REDO) SUBTYPE(ALLOC/DEALLOC OF SPACE IN LOB SPACE MAP PAGE) CLR(NO) PROCNAME(DSNOALLO)
0000894FB4C3	URID(0000894FAD8D) LRSN(BF3BF3D59061) DBID(014A) OBID(0007) PAGE(00000001) TYPE(UNDO REDO) SUBTYPE(LOB HIGH LEVEL SPACE MAP PAGE UPDATE) CLR(NO) PROCNAME(DSNOALLO)
0000894FB503	URID(0000894FAD8D) LRSN(BF3BF3D59064) DBID(014A) OBID(0007) PAGE(00000015) TYPE(REDO) SUBTYPE(LOB MAP CHANGE) CLR(NO) PROCNAME(DSNOFLMP)
0000894FCAB2	URID(0000894FAD8D) LRSN(BF3BF3D5BCD9) TYPE(UR CONTROL) SUBTYPE(BEGIN COMMIT1)
0000894FD155	URID(0000894FAD8D) LRSN(BF3BF3D5BE6C) TYPE(UR CONTROL) SUBTYPE(PHASE 1 TO 2)
0000894FD189	URID(0000894FAD8D) LRSN(BF3BF3D5BEEC) TYPE(UR CONTROL) SUBTYPE(END COMMIT2)

3.4 Additional considerations for creating LOB objects

In this section, we describe other important options available when creating LOBs.

3.4.1 Data conversion

DB2 for z/OS is often used as the enterprise server of large client server systems. In these environments, character representations can vary on clients and servers across different platforms and across many different geographic locations. In this scenario, you might also want to deal with data conversion topics. Large objects can also be subject to conversion,

depending on the type of data you plan to store in your large objects. See Figure 3-6 on page 53.

One area where this sort of environment exists is in data centers of multinational companies and even more in e-business environments. In both of these examples, a geographically diverse group of users interact with a central server, storing and retrieving data.

Today, there are hundreds of different encoding systems. No single encoding could contain enough characters: for example, the European Union alone requires several different encoding schemes to cover all of its languages. Even for a single language such as English, no single encoding was adequate for all the letters, punctuation, and technical symbols in common use.

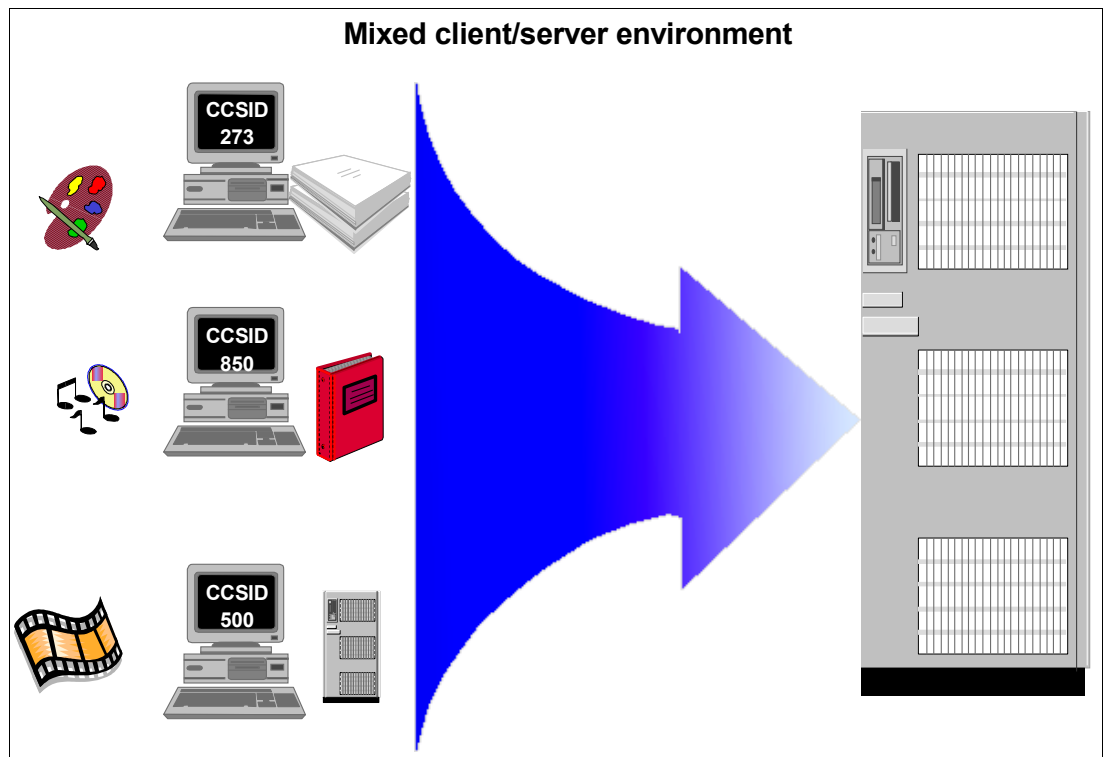


Figure 3-6 Mixed client/server environment with different data types

These encoding systems also conflict with one another. That is, two encoding schemes can use the same codes for two different characters, or use different codes for the same character. Figure 3-7 gives you a good example of the described case.

HEX DIGITS 1ST 2ND	4-	5-	6-	7-	8-	9-	A-	B-	C-	D-	E-	F-	HEX DIGITS 1ST 2ND	4-	5-	6-	7-	8-	9-	A-	B-	C-	D-	E-	F-
-0	SP190000	SM230000	SP190000	LO190000	LO200000	SM190000	SM170000	SM170000	SM170000	SM170000	SM170000	SM170000	-0	SP190000	SM230000	SP190000	LO190000	LO200000	SM190000	SM170000	SM170000	SM170000	SM170000	SM170000	SM170000
-1	SP190000	SM230000	SP190000	LO190000	LO200000	SM190000	SM170000	SM170000	SM170000	SM170000	SM170000	SM170000	-1	SP190000	SM230000	SP190000	LO190000	LO200000	SM190000	SM170000	SM170000	SM170000	SM170000	SM170000	SM170000
-2	SP190000	SM230000	SP190000	LO190000	LO200000	SM190000	SM170000	SM170000	SM170000	SM170000	SM170000	SM170000	-2	SP190000	SM230000	SP190000	LO190000	LO200000	SM190000	SM170000	SM170000	SM170000	SM170000	SM170000	SM170000
-3	SP190000	SM230000	SP190000	LO190000	LO200000	SM190000	SM170000	SM170000	SM170000	SM170000	SM170000	SM170000	-3	SP190000	SM230000	SP190000	LO190000	LO200000	SM190000	SM170000	SM170000	SM170000	SM170000	SM170000	SM170000
-4	SP190000	SM230000	SP190000	LO190000	LO200000	SM190000	SM170000	SM170000	SM170000	SM170000	SM170000	SM170000	-4	SP190000	SM230000	SP190000	LO190000	LO200000	SM190000	SM170000	SM170000	SM170000	SM170000	SM170000	SM170000
-5	SP190000	SM230000	SP190000	LO190000	LO200000	SM190000	SM170000	SM170000	SM170000	SM170000	SM170000	SM170000	-5	SP190000	SM230000	SP190000	LO190000	LO200000	SM190000	SM170000	SM170000	SM170000	SM170000	SM170000	SM170000
-6	SP190000	SM230000	SP190000	LO190000	LO200000	SM190000	SM170000	SM170000	SM170000	SM170000	SM170000	SM170000	-6	SP190000	SM230000	SP190000	LO190000	LO200000	SM190000	SM170000	SM170000	SM170000	SM170000	SM170000	SM170000
-7	SP190000	SM230000	SP190000	LO190000	LO200000	SM190000	SM170000	SM170000	SM170000	SM170000	SM170000	SM170000	-7	SP190000	SM230000	SP190000	LO190000	LO200000	SM190000	SM170000	SM170000	SM170000	SM170000	SM170000	SM170000
-8	SP190000	SM230000	SP190000	LO190000	LO200000	SM190000	SM170000	SM170000	SM170000	SM170000	SM170000	SM170000	-8	SP190000	SM230000	SP190000	LO190000	LO200000	SM190000	SM170000	SM170000	SM170000	SM170000	SM170000	SM170000
-9	SP190000	SM230000	SP190000	LO190000	LO200000	SM190000	SM170000	SM170000	SM170000	SM170000	SM170000	SM170000	-9	SP190000	SM230000	SP190000	LO190000	LO200000	SM190000	SM170000	SM170000	SM170000	SM170000	SM170000	SM170000
-A	SP190000	SM230000	SP190000	LO190000	LO200000	SM190000	SM170000	SM170000	SM170000	SM170000	SM170000	SM170000	-A	SP190000	SM230000	SP190000	LO190000	LO200000	SM190000	SM170000	SM170000	SM170000	SM170000	SM170000	SM170000
-B	SP190000	SM230000	SP190000	LO190000	LO200000	SM190000	SM170000	SM170000	SM170000	SM170000	SM170000	SM170000	-B	SP190000	SM230000	SP190000	LO190000	LO200000	SM190000	SM170000	SM170000	SM170000	SM170000	SM170000	SM170000
-C	SP190000	SM230000	SP190000	LO190000	LO200000	SM190000	SM170000	SM170000	SM170000	SM170000	SM170000	SM170000	-C	SP190000	SM230000	SP190000	LO190000	LO200000	SM190000	SM170000	SM170000	SM170000	SM170000	SM170000	SM170000
-D	SP190000	SM230000	SP190000	LO190000	LO200000	SM190000	SM170000	SM170000	SM170000	SM170000	SM170000	SM170000	-D	SP190000	SM230000	SP190000	LO190000	LO200000	SM190000	SM170000	SM170000	SM170000	SM170000	SM170000	SM170000
-E	SP190000	SM230000	SP190000	LO190000	LO200000	SM190000	SM170000	SM170000	SM170000	SM170000	SM170000	SM170000	-E	SP190000	SM230000	SP190000	LO190000	LO200000	SM190000	SM170000	SM170000	SM170000	SM170000	SM170000	SM170000
-F	SP190000	SM230000	SP190000	LO190000	LO200000	SM190000	SM170000	SM170000	SM170000	SM170000	SM170000	SM170000	-F	SP190000	SM230000	SP190000	LO190000	LO200000	SM190000	SM170000	SM170000	SM170000	SM170000	SM170000	SM170000

Figure 3-7 Differences between code pages 37 and 500

It is likely that LOBs are being moved to the mainframe from other platforms and even other geographies, because more businesses are spread geographically all over the world. In these cases, a verification of the needed data conversion is absolutely necessary, unless you decide to use the Unicode support as introduced with DB2 Version 8.

Different LOBs are suitable for different tasks. BLOBs are designed to contain binary data. As such, they have no CCSID associated with them. CLOBs and DBCLOBs are designed to contain text data. CLOBs have the normal single byte and mixed CCSIDs associated with them, while DBCLOBs have the graphic CCSID associated with them.

The old mechanism of code page translations is still valid in most of the cases. For encoding and decoding, DB2 first accesses the SYSIBM.SYSSTRINGS table. If the conversion cannot be resolved, DB2 turns to z/OS Unicode Conversion Services. This mechanism is constantly updated and improved to make the conversion more efficient. See *DB2 UDB for z/OS Version 8 Performance Topics*, SG24-6465, for more information.

Unicode provides a unique number for almost every character, on any platform, in any language, and by any program. The Unicode character encoding standard is a character-encoding scheme that includes characters from almost all living languages. It is an implementation of the ISO-10646 standard.

There are several popular implementations of the Unicode standard such as:

- ▶ UCS-2 - Universal Character Set coded in 2 octets.
- ▶ UCS-4 - Universal Character Set coded in 4 octets. This becomes UTF-32.
- ▶ UTF-8 - Unicode Transformation Format for 8 bit (ASCII safe Unicode). Characters are encoded in 1 to 6 bytes.
- ▶ UTF-16 - Unicode Transformation Format for 16 bits. The format is a superset of UCS-2 and contains an encoding form that allows more than 64 KB characters to be represented.

DB2 V7 support for Unicode provides the most popular implementations of Unicode: UTF-8 and UTF-16.

- ▶ CHAR, VARCHAR, LONG VARCHAR, and CLOB data for SBCS data is stored as ASCII (7 bit) CODE CCSID 367.
- ▶ CHAR, VARCHAR, LONG VARCHAR, and CLOB data for mixed data is stored as UTF-8 (Unicode CCSID 1208).
- ▶ GRAPHIC, VARGRAPHIC, LONG VARGRAPHIC, and DBCLOB data is stored as UTF-16 (Unicode CCSID 1200).

If you are working with character string data in UTF-8, you should be aware that ASCII characters are encoded into one byte lengths. However, non-ASCII characters, for example, Japanese characters, are encoded into two or three byte lengths in a multiple-byte character code set (MBCS). Therefore, if you define an 'n' bytes length character column, you can store strings anywhere from 'n/3' to 'n' characters depending on the ratio of ASCII to non-ASCII character code elements. DB2 *cannot* use the table SYSIBM.SYSSTRINGS for conversion to and from Unicode CCSIDs. Instead, DB2 uses z/OS Conversion Services to manage all of the conversions to and from Unicode CCSIDs.

Use the links in Table 3-10 on page 55 for additional information about Unicode with DB2:

Table 3-10 DB2 Unicode support - Additional useful resources

Title	Link
Unicode site	http://www.unicode.org
The Unicode character code charts	http://www.unicode.org/charts
<i>DB2 Version 9.1 for z/OS SQL Reference</i> , SC18-9854, Section on Character Conversion	http://www.ibm.com/software/data/db2/zos/v9books.html
<i>DB2 Version 9.1 for z/OS Installation Guide</i> , GC18-9846, Appendix A. Character conversion	http://www.ibm.com/software/data/db2/zos/v9books.html
<i>DB2 9 for z/OS Internationalization Guide</i>	http://www.ibm.com/software/data/db2/zos/v9books.html
<i>z/OS support for Unicode: Unicode Services</i> , SA22-7649	http://publibz.boulder.ibm.com/epubs/pdf/iea2un30.pdf
DB2 V7 and V8 Unicode support	ftp://ftp.software.ibm.com/software/db2storedprocedure/db2zos390/techdocs/F10.pdf
Unicode performance in DB2 for z/OS	http://www.idug.org/idug/_journalpdfarea/IDUG_V11N02.pdf
All for One and One for All - Part 1	http://www.idug.org/idug/_journalpdfarea/IDUG_V10N03.pdf
All for One and One for All - Part 2	http://www.idug.org/idug/_journalpdfarea/IDUG_V11N01.pdf

Note: Character conversion with LOBs causes LOB materialization. Avoid it. For more information and materialization avoidance tips, refer to 8.1, “LOB materialization” on page 238.

3.4.2 Buffer pools and LOB table spaces

If you want to store huge amounts of data in your LOBs, you should consider using separate storage groups and buffer pools from your other data in order to avoid interference resulting in a possible impact on performance. This is because a single LOB value is able to use up to 2 GB of your buffer pool if assigned to the same buffer pool as the non-LOB table spaces. If all LOBs are going to be read, the content of the buffer pool can mainly consist of LOBs.

If your buffer pool, which is dedicated to hold your LOB values, can only hold a small number of your average LOBs, DB2 might be idle by flushing out the buffer pool values, because reading or updating a new LOB value updates the values inside the buffer pool. Therefore, a small buffer pool can be a good choice to avoid the overhead of recycling the buffer pool's data pages with a large LOB column value, which most likely are not being referenced by someone else before they are erased again from the buffer pool. For further discussion about the performance implications of buffer pool choice for LOB objects, see 8.3, "Buffer pools and group buffer pools" on page 244.

3.4.3 Locking with LOBs

There are at least two different types of lock sizes you can choose between. Valid parameters are LOCKSIZE LOB and LOCKSIZE TABLESPACE. Choosing LOCKSIZE ANY implies the use of LOCKSIZE LOB for DB2. From DB2 Version 6 until DB2 V8, DB2 uses two types of locks for ensuring a LOB's integrity: the S-LOB and the X-LOB locks. They are very similar to the common S- and X-Locks. There are no U-LOB locks, because of a different update mechanism taking place for LOBs. See 3.3.1, "LOGGED and NOT LOGGED attributes" on page 44 for a description of updating techniques for LOBs.

There is no support for Uncommitted Read (UR) on LOBs. Assuming LOCKSIZE LOB, selecting a LOB acquires an S-LOB lock on the accessed LOB, even if you use ISOLATION (UR) in a package or WITH UR in your SQL statement. This is because of the new table space format where physical correctness has to be guaranteed while retrieving the LOB value, which might be spanned over many pages. Acquiring S-LOB locks prevents the application from retrieving only partial LOB data.

Deleting a LOB also requests an S-LOB lock on a specific LOB. But how does this work with other transactions selecting the same LOB? A transaction can delete a LOB that another transaction is reading at the same time, but the space is not reused until all readers of the LOB have committed their work.

So S-LOB locks do not prevent a LOB from being deleted.

Inserting a LOB acquires an X-LOB lock on the new LOB, and the lock is released at COMMIT as usual. If a LOB is locked by an X-LOB lock, no readers can access the LOB before it is committed.

Because every LOB lock is like a row lock, the number of acquired locks can increase dramatically, especially when mass-updates occur on LOBs, or many LOBs are inserted using subselect.

Beginning with DB2 9, the entire locking technique for LOBs has changed, basically using locks on the base table to guarantee consistency of the LOBs your application accesses. DB2 ensures using a comparison technique of log record sequence from the oldest reader in the system to make sure that no deleted LOB data pages are reused with new LOB values before the last reader of this particular value has been removed.

You can find more details about locking for LOBs in 4.5, "Locking" on page 94.

Note: Starting with DB2 9, LOBs up to 2 GB in size are eligible for logging, so all LOBs are able to be logged. Previously, the limit was 1 GB.

3.4.4 Buffer pool and page size considerations

You can assign 4 KB, 8 KB, 16 KB, or 32 KB buffer pools to your LOB table space. But which one to use depends on several considerations.

Choosing a page size is the common trade-off between minimizing the number of getpages, which simply maximizes performance, and not wasting space. Because one data page in a LOB table space never stores data of more than one LOB value, the space not used by the LOB value in the last page remains unused. To estimate a good average size of a LOB value, use the formula:

$$\text{Size of a LOB} = (\text{Average length of all LOBs}) * 1.1$$

Use the general rule shown in Table 3-11 to choose a page size that minimizes the number of getpages.

Table 3-11 Choosing a LOB page size that minimizes getpages

Average LOB size (ALS)	Recommended page size for LOB table space
ALS < 4 KB	4 KB
4 KB < ALS < 8 KB	8 KB
8 KB < ALS < 16 KB	16 KB
16 KB < ALS	32 KB

Using this technique, a LOB value of 22 KB means 10 KB of unused space, but only one getpage request for DB2. Therefore, you have to analyze your data to determine what is best for you and your LOBs. Normally, you have 32 KB data pages assigned to your LOB table space, because you should use LOBs only for real large objects.

By using Modified Indirect Addressing Words (MIDAW), the impact of performance for getpages using different page sizes can be minimized. IBM z/OS 1.7 support for IBM z9™ systems includes a new function that supports MIDAW. The z9 implements a new function for channel programming called modified indirect addressing words. MIDAWs can be used to move data over FICON® and ESCON® channels. For FICON channels, this support can provide substantially better response time while increasing overall channel bandwidth. MIDAW's exploitation by z/OS is expected to improve performance for some DB2 table scan, DB2 sequential prefetch, BSAM, and extended-format data set operations by reducing system overhead for I/O requests, with no application changes. Therefore, the page size for the LOB table space can become less important from a performance point of view.

For more information, see *How does the MIDAW facility improve the performance of FICON channels using DB2 and other workloads?*, REDP-4201.

If your LOB data is all the same size, it might be easier to choose a page size that makes more efficient use of data pages without impacting performance. For LOBs that are all the same size, consider Table 3-12, which maximizes space savings without sacrificing performance.

Table 3-12 Choosing a LOB page size for LOBs that are all the same size

LOB size (LS)	Recommended page size for LOB table space
LS < 4 KB	4 KB
4 KB < LS < 8 KB	8 KB
8 KB < LS < 12 KB	4 KB
12 KB < LS < 16 KB	16 KB
16 KB < LS < 24 KB	8 KB
24 KB < LS < 32 KB	32 KB
32 KB < LS < 48 KB	16 KB
48 KB < LS	32 KB

Also, see buffer pool considerations and buffer pool thresholds (*DWQT* and *VDWQT*) in 8.3, “Buffer pools and group buffer pools” on page 244, for more information about this topic.

If you have decided on the page size you want to use, DB2 allocates at least *n* KB of space even if you tell DB2 to allocate less than the minimum amount of space for that page size. Table 3-13 provides information about the minimum space requirements for primary (PRIQTY) and secondary (SECQTY) quantity, and how they might change from your specification. This reflects the DB2 expectation that LOB objects are going to be used as they are intended, that is, for LARGE objects, and prepares the environment accordingly.

Table 3-13 Primary and secondary quantity with LOBs

PRIQTY and SECQTY		
Specification in KB	Page size	Resulting allocation in KB
< 200	4 KB	200
< 400	8 KB	400
< 800	16 KB	800
< 1 600	32 KB	1,600
> 4,194,304	Any	4,194,304

Note: The maximum value allowed for PRIQTY is 64 GB (67,108,864 KB).

3.4.5 DSSIZE for LOB table spaces

DSSIZE represents the data set size value passed to DB2 to indicate the maximum allowable size. With the combination of the *DSSIZE* parameter and the extended addressability function of SMS-managed VSAM data sets, data sets can grow up to a size of 64 GB.

Table 3-14 summarizes the partition and partitioned table space sizes for the current DB2 versions.

Table 3-14 Summary of data set, partition, and partitioned table space sizes

DB2 version	Number of partitions	Maximum size each	Total maximum size
V8 and 9	4,096	64 GB	65,536 TB
Note: DSSIZE specifies the maximum size for each partition, or for LOB table spaces, for each data set. If you specify DSSIZE, you must also specify Numparts, Maxpartitions, or LOB clause. DSSIZE and SMS-managed table spaces are required.			

Since each LOB table space can consist of 254 data sets, you can reach the maximum amount of 16,256 GB (approximately 16 TB) for a single LOB column in one LOB table space. Considering a partitioned base table with 4,096 partitions, these 16,256 GB can occur up to 4,096 times (one LOB table space for each partition of the base table), so you can have 66,584,576 GB or 65,536 TB for one single LOB column.

For LOB table spaces, if DSSIZE is not specified, the default for the maximum size of each data set is 4 GB. The maximum number of data sets is 254. This means a maximum value of 1,016 GB (approximately 1 TB) for a non-partitioned base table and 258,064 GB (approximately 258 TB) for a partitioned base table consisting of 254 partitions.

The maximum amount of space for PRIQTY and SECQTY that you can specify is at least 4,194,304 KB. So, make sure that you have specified PRIQTY, and n times SECQTY, if allocated, is about 64 GB. The value of n depends on the version of DFSMS running in your system. Extended addressability for VSAM data sets, leading to a maximum file size of 64 GB, was introduced in DFSMS Version 1 Release 5. To benefit from the maximum size provided for LOBs, DFSMS Version 1 Release 5 is required.

Note: If DSSIZE is greater than 4 GB, make sure that this data set belongs to a DFSMS class that is defined with the extended addressability attribute, and also, that the automatic class selection routine associates this data set with this data class. Otherwise, DB2 is not able to allocate the requested space, and it issues SQLCODE -904.

3.4.6 GBPCACHE parameter

When defining or altering a table space definition, in a data sharing environment, GBPCACHE specifies what pages of the table space or partition are written to the group buffer pool. In a non-data-sharing environment, this parameter is ignored. The values are:

- **CHANGED**

When there is inter-DB2 R/W interest on the table space or partition, updated pages are written to the group buffer pool.

- **ALL**

Indicates that pages are to be cached in the group buffer pool as they are read in from DASD.

- **NONE**

Indicates that no pages are to be cached to the group buffer pool.

- **SYSTEM**

The SYSTEM parameter was added for LOBs to prevent LOB data from flooding the global buffer pool and still help with performance. SYSTEM Indicates that only changed system pages within the LOB table space are to be cached to the group buffer pool. A system page is a space map page or any other page that does not contain actual data values. SYSTEM is the default for a LOB table space.

In DB2 V8, in a data sharing environment, GBPCACHE SYSTEM is recommended for large objects.

In DB2 9, in a data sharing environment, because of the changes in LOB locks management (see 3.4.3, “Locking with LOBs” on page 56), you might want to consider specifying CHANGED.

Note that when you change an option by specifying ALTER TABLESPACE GBPCACHE in a data sharing environment, the table space or partition must be in the stopped state when the statement is executed.

See 8.3, “Buffer pools and group buffer pools” on page 244, for more information.

3.4.7 Impact on cursors fetching LOB values

In general, an application program declares a cursor and fetches the cursor in a specific section of an application program. The type of variables you fetch the cursor into cannot vary, even if you fetch the same cursor in two different sections of your application program for whatever reasons. Fetching a LOB column now gives you the flexibility to fetch into a large object host variable or into a large object locator variable. DB2 allows you to make a decision of the kind of variable you want to fetch the LOB value into, depending on the current value of special register CURRENT RULES at the time the cursor is opened.

Using CURRENT RULES DB2

- ▶ After the cursor is opened, if the first FETCH executed uses a LOB locator to access a LOB column, no subsequent FETCH for that cursor can fetch that LOB column into a LOB host variable.
- ▶ After the cursor is opened, if the first FETCH executed uses a LOB host variable to access a LOB column, no subsequent FETCH for that cursor can fetch that LOB column into a LOB locator.

Using CURRENT RULES STD

- ▶ After the cursor is opened, and after the first FETCH executed, regardless of where the LOB value is fetched into (LOB locator or LOB host variable), a subsequent FETCH for that cursor can FETCH the LOB column into either a LOB locator or a LOB host variable.

Although a CURRENT RULES special register set to STD gives you more flexibility to let your application program switch between both methods, you can get better performance if you use a value of DB2 in distributed environments. When you use the STD option, the server has to send and receive network messages for each FETCH to indicate whether the data being transferred is a LOB locator or a LOB value. With the DB2 option, the server knows the size of the LOB data after the first FETCH, so an extra message about the LOB data size is unnecessary. The server can send multiple blocks of data to the requester at one time, which reduces the total time for data transfer.

Using the STD option, DB2 cannot make any assumptions about what the requesting application might want on the next fetch.

3.5 LOBs are different DB2 objects

In this section, we highlight the main differences between LOBs and standard DB2 objects.

Lock sizes

With LOBs, DB2 uses a different approach to store huge amounts of data that can span over many pages. Common locking techniques acquire locks on table spaces, partitions, tables, pages, or rows. Considering a LOB table space, there is only one table space, containing one table, which possibly holds several LOB values. DB2 does not acquire a lock on many LOB data pages, nor on the entire table space. Therefore, DB2 uses two additional lock sizes to efficiently handle locking for large objects before DB2 9:

- ▶ Shared LOB lock (**S-LOB lock**)
- ▶ Exclusive LOB lock (**X-LOB lock**)

These types of locks are also acquired by the Internal Resource Lock Manager (IRLM). LOB locks are not related to any pages at all. DB2 takes both lock types explicitly by a combination of the LOB table space, the associated ROWID, and the LOB version number.

In general, DB2 9 no longer uses LOB locks to ensure data integrity but locks on the base table space and information about the oldest reader accessing your LOB data. For a more detailed description about locking for large objects and lock sequences in DB2 V8 and DB2 9, see 4.5, “Locking” on page 94.

Update mechanisms for LOBs

When you update a large object, DB2 uses an updating technique that is different from the updating technique for standard objects in DB2. Updating a LOB for DB2 means deallocation of used data pages, and allocating and inserting new data pages, which contain the new LOB value. For lock-related information about this topic, see “Locks with UPDATE” on page 99, and for rollback-related information, see “Shadow Copy Recovery” on page 100.

Large objects and indexes

For obvious reasons, a LOB column in the auxiliary table is not indexable. Prior to DB2 8, the maximum length of an index column is 255 bytes. Beginning with DB2 8, the maximum key length is extended from 255 bytes to 2 000 bytes but LOBs are potentially even bigger. Therefore, you cannot create an index on a LOB column. With DB2 9, you can create an index on an expression from a LOB column.

Compression

DB2 does not allow you to specify COMPRESS YES for LOB table spaces. Most objects stored in a BLOB column could already be compressed anyway, such as JPEG pictures or ZIP folders. Regardless of compression for a LOB table space, you can specify COMPRESS YES for the table space containing the base table.

EDITPROCs, FIELDPROCs, and VALIDPROCs

LOB values cannot be compared, except with the LIKE predicate, and because they are not stored along with other columns, they are not available to any database procedures such as EDITPROCs, FIELDPROCs, or VALIDPROCs. Note that although they are not available to these types of procedures, they are available to any triggers that are defined on the base table.

Check constraints

A check constraint cannot reference a LOB column. Those values are not designed to be eligible to work with constraints, because they are too large to check when changing a row's content in a table.

Data capture and data propagation

Even if LOB values have been defined as NOT LOGGED or LOG NO in DB2 V8, WebSphere® Information Integrator Q Replication is able to replicate those values. The capture program reads the LOB descriptor to determine if any data in the LOB column has changed or not, and places an indicator in the capture data table. When the apply program reads the indicator, it then copies the entire LOB value, not just the changed parts of the LOB value. The apply program always copies the most current version of a LOB column directly from the source table, in our case, the auxiliary table. So it replicates only full LOBs, parts of a LOB are not replicated.

For details about replicating LOBs, refer to *WebSphere Information Integrator Q Replication: Fast Track Implementation Scenarios*, SG24-6487.

3.6 Physical layout of LOBs

LOB table spaces have a completely different format compared to other table spaces. Because a LOB entry in a LOB column can span pages, pages have to be *chunked* together. A *chunk* is referred to as 16 pages of contiguous space acquired in a LOB table space. Depending on your size of data, a certain number of chunks are allocated. A single LOB value can be stored using many chunks also using page allocations of fewer than 16 contiguous pages depending on the space available. Figure 3-8 on page 62 illustrates the distribution of pages for a single LOB value. The LOB in our example is stored in a chunk containing pages 1 to 16, pages 21 to 28, another chunk beginning at page 49, and the last four single pages starting at page 83. Keeping the LOB well chunked helps in LOB processing. See 6.9, “REORG” on page 185 for need to REORG LOBs.

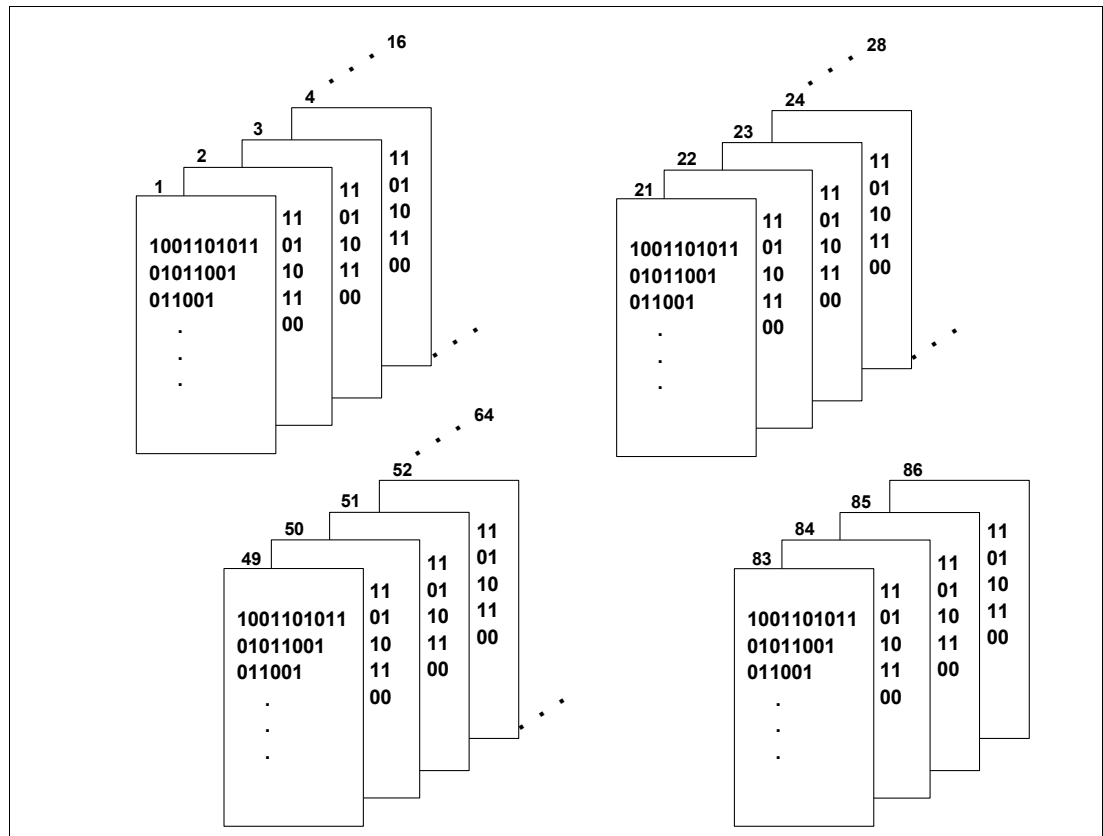


Figure 3-8 LOB value spanned over pages using chunks and non-chunks

LOBs can be stored in chunks of 16 pages and also can be stored using page allocations of fewer than 16 pages (21 - 28 and 83 - 86 in our example). After inserting, deleting, and updating LOB values, the pages can be nearly everywhere.

LOB table space organization

Storing LOBs causes the table spaces containing their values to be organized in a slightly different way in order to support pageset structures for columns spanning the maximum available page size.

As in every other table space, a LOB table space has only one header page. The header page contains DB2 internal control information which is needed when you access your LOB data. The new type of pages are LOB space map pages. They are structured like a multi-level index, which is an index containing several levels of leaf pages and which is basically a pointer to chunks and pages. You can find at least one LOB map page for every version of every single LOB value in your LOB table space. When we talk about deallocation of LOB pages, the space map pages have set the *invalid flag* for all pages containing the LOB information, which includes both LOB map pages and LOB data pages.

LOB map pages contain descriptive information about LOB values. The chunk information itself is stored in LOB map pages, which point to the associated data pages for a LOB value. After reading the information stored in the LOB map pages, DB2 knows where to find the information it has to retrieve to access a particular LOB value. See Figure 3-9 for possible LOB map page information about a single value.

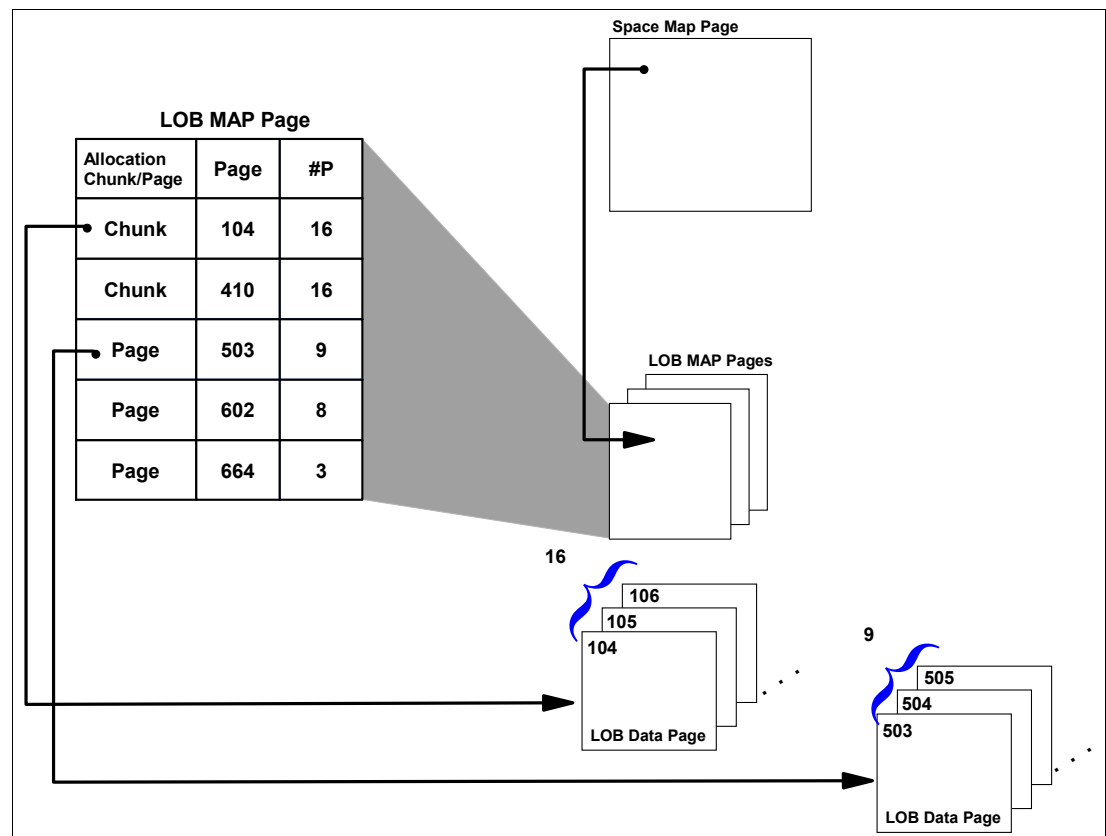


Figure 3-9 LOB data pages chunked together using a space map and LOB map pages

You can think of two different allocation units that a LOB map page points to. First, it can point to a chunk of data pages, which is nothing more than 16 pages of contiguous space (indicated by #P = 16 in Figure 3-9). As soon as DB2 uses partial chunks to store parts of the LOB value, it contains the page number where the allocation starts and the number of contiguous pages used.

For a detailed description of LOB system pages, refer to the licensed documentation *DB2 Version 9.1 for z/OS Diagnosis Guide and Reference*, LY37-3218.

Note: When using DSN1PRNT, you might also see pages referring to LOB values that have been already deleted and whose space was not reused up to now.



Using LOBs

This chapter discusses basic considerations when starting to use LOBs in applications.

This chapter contains the following:

- ▶ Language considerations
- ▶ LOB locators
- ▶ DRDA LOB flow optimization
- ▶ Feeding a LOB column
- ▶ Locking
- ▶ Unloading LOBs
- ▶ Updating LOBs
- ▶ General best practices

4.1 Language considerations

If you want to use LOBs in your application programs, there are certain differences to take into account when comparing LOBs to the other DB2 data types. You can handle LOBs in “classic” host languages such as COBOL, PL/I, and REXX but also in more recent languages such as JAVA.

4.1.1 LOB host variables, locators, and file reference variables

When you write applications to manipulate LOB data, you need to declare host variables to hold the LOB data, LOB locator, or LOB file reference variables to point to the LOB data. You can declare LOB host variables, LOB locators, and LOB file reference variables in Assembler, C, C++, COBOL, Fortran, and PL/I. For each host variable, locator, or file reference variable of SQL type BLOB, CLOB, or DBCLOB that you declare, DB2 generates an equivalent declaration that uses host language data types. When you refer to a LOB host variable, LOB locator, or LOB file reference variable in an SQL statement, you must use the variable that you specified in the SQL type declaration. When you refer to the host variable in a host language statement, you must use the variable that DB2 generates.

DB2 supports host variable declarations for LOBs with lengths of up to 2 GB - 1. However, the size of a LOB host variable is limited by the restrictions of the host language and the amount of storage available to the program.

To retrieve LOB data from a DB2 table, you can define host variables that are large enough to hold all of the LOB data. This requires your application to allocate large amounts of storage, and requires DB2 to move large amounts of data, which can be inefficient or impractical. Instead, you can use LOB locators. LOB locators let you manipulate LOB data without retrieving the data from the DB2 table. Using LOB locators for LOB data retrieval is a good choice in the following situations:

- ▶ When you move only a small part of a LOB to a client program
- ▶ When the entire LOB does not fit in the application’s memory
- ▶ When the program needs a temporary LOB value from a LOB expression but does not need to save the result
- ▶ When performance is important

A LOB locator is associated with a LOB value or expression, not with a row in a DB2 table or a physical storage location in a table space. Therefore, after you select a LOB value using a locator, the value in the locator normally does not change until the current unit of work ends. However, the value of the LOB itself can change. If you want to remove the association between a LOB locator and its value before a unit of work ends, execute the `FREE LOCATOR` statement. To keep the association between a LOB locator and its value after the unit of work ends, execute the `HOLD LOCATOR` statement. After you execute a `HOLD LOCATOR` statement, the locator keeps the association with the corresponding value until you execute a `FREE LOCATOR` statement or the program ends.

Like host variables, each LOB locator can have an associated indicator variable. When you select a LOB column using a LOB locator and the LOB column contains a null value, the indicator variable is set to a negative value. However, the value in the LOB locator itself is not changed and still contains the old value. Therefore, when you use LOB locators to retrieve data from columns that can contain null values, define indicator variables for the LOB locators, and first check the indicator variables after you fetch data into the LOB locators. If an indicator variable is negative after a fetch operation, you cannot use the value in the LOB locator.

In a host application, starting with DB2 9, you can use a file reference variable of type BLOB_FILE, CLOB_FILE, or DBCLOB_FILE to insert a LOB from a file into a DB2 table or to select a LOB from a DB2 table into a file. When you use a file reference variable, you can select or insert an entire LOB value without contiguous application storage to contain the entire LOB. In other words, LOB file reference variables move LOB values from the database server to an application or from an application to the database server without going through the application's memory.

Furthermore, LOB file reference variables bypass the host language limitation on the maximum size allowed for dynamic storage to contain a LOB value. You can declare a LOB file reference variable or a LOB file reference array for applications that are written in C, COBOL, PL/I, and Assembler. The LOB file reference variables do not contain LOB data; they represent a file that contains LOB data. Database queries, updates, and inserts can use file reference variables to store or retrieve column values. As with other host variables, a LOB file reference variable can have an associated indicator variable.

4.1.2 Use of a double or triple SQLDA in dynamic SQL

If you write a dynamic SQL program, and you want to be able to retrieve LOB values using a SQLDA, you must use a double SQLDA or triple SQLDA.

The SQLDA is a collection of variables that is required for the execution of the SQL DESCRIBE statement, and can be optionally used by the PREPARE, OPEN, FETCH, EXECUTE, and CALL statements. The meaning of the information in an SQLDA depends on the context in which it is used. For DESCRIBE and PREPARE INTO, DB2 sets fields in the SQLDA to provide information about the columns of the result set or table to the application program. For OPEN, EXECUTE, FETCH, and CALL, the application program must set the fields in the SQLDA to provide DB2 with information about the host variables of the program.

An SQLDA can contain a variable number of occurrences of SQLVAR, each of which is a set of fields that describes one column in the result table of a SELECT statement. If your program wants to be able to retrieve n columns, you should at least allocate $2*n$ SQLVARS if you want to be able to retrieve LOB columns (double SQLDA) and $3*n$ SQLVARS (triple SQLDA) if you want to be able to retrieve LOB columns and have both column names and column labels in your SQLDA (by using the USING BOTH option in your PREPARE or DESCRIBE statement).

The base SQLVAR contains the following variables when set by DB2 during DESCRIBE or PREPARE:

- ▶ SQLTYPE: Indicates the data type of the column and whether it can contain null values
- ▶ SQLLEN: The length attribute of the column
- ▶ SQLDATA: The CCSID of the column
- ▶ SQLIND: Reserved
- ▶ SQLNAME: The name of the column

For LOBs, the following SQLTYPES are returned in the base SQLVAR of the SQLDA as shown in Table 4-1 on page 68. An even value of SQLTYPE means the column does not allow nulls, and an odd value means the column does allow nulls. For LOB columns, the SQLLEN field is always zero (only half word) and the actual length of the LOB columns can be found in the SQLLONGLEN field of the extended SQLVAR.

Table 4-1 *SQLTYPE and SQLLEN of LOB columns or LOB host variables in the SQLDA*

SQLTYPE	Data type	SQLLEN
404/405	BLOB	0
408/409	CLOB	0
412/413	DBCLOB	0
916/917	BLOB_FILE	267
920/921	CLOB_FILE	267
924/925	DBCLOB_FILE	267

For LOBs, the extended SQLVAR contains following variables:

- ▶ SQLLONGLEN: The length attribute of the LOB column
- ▶ SQLDATALEN: Not used

The application program should set the following fields in the base SQLVAR during OPEN, EXECUTE, FETCH, and CALL:

- ▶ SQLTYPE: The data type of the host variable and whether an indicator variable is provided
- ▶ SQLLEN: The length attribute of the host variable. Always 0 for LOBs
- ▶ SQLDATA: The address of the host variable
- ▶ SQLIND: The address of the indicator variable if SQLTYPE is odd
- ▶ SQLNAME: The CCSID of the host variable

For LOBs, the extended SQLVAR should be set as follows:

- ▶ SQLLONGLEN: The length attribute of the host variable.
- ▶ SQLDATALEN: If the value of this field is null, the actual length of the LOB is stored in the 4 bytes immediately before the start of the data, and SQLDATA in the base SQLVAR points to the first byte of the field length. The actual length indicates the number of bytes for a BLOB or CLOB, and the number of characters for a DBCLOB. If the value of this field is not null, the field contains a pointer to a 4-byte long buffer that contains the actual length in bytes (even for DBCLOBs) of the data in the buffer pointed to from the SQLDATA field in the matching base SQLVAR.

4.1.3 Working with LOBs in JDBC and SQLJ applications

The IBM DB2 Driver for JDBC and SQLJ includes all of the LOB support in the JDBC 3.0 and earlier specifications. This driver also includes support for LOBs in additional methods and for additional data types:

- ▶ DRDA LOB flow optimization: If the database server supports progressive streaming, the IBM DB2 Driver for JDBC and SQLJ can use progressive streaming to retrieve data in LOB columns. With progressive streaming, the database server dynamically determines the most efficient mode in which to return LOB data, based on the size of the LOBs. Refer to 4.3, “DRDA LOB flow optimization” on page 79 for a detailed description.
- ▶ LOB locator support: The IBM DB2 Driver for JDBC and SQLJ can use LOB locators to retrieve data in LOB columns. You should use LOB locators only if the database server does not support progressive streaming. See 4.2, “LOB locators” on page 73 for more information.

As in any other language, a LOB locator in a Java application is associated with only one DB2 subsystem. You cannot use a single LOB locator to move data between two different DB2 subsystems. To move LOB data between two DB2 subsystems, you need to materialize the LOB data when you retrieve it from a table in the first DB2 subsystem and then insert that data into the table in the second DB2 subsystem.

In addition to the methods in the JDBC specification, the IBM DB2 Driver for JDBC and SQLJ includes LOB support in the following methods:

- ▶ You can specify a BLOB column as an argument of the following *ResultSet* methods to retrieve data from a BLOB column: *getBinaryStream* and *getBytes*
- ▶ You can specify a CLOB column as an argument of the following *ResultSet* methods to retrieve data from a CLOB column: *getAsciiStream*, *getCharacterStream*, *getString*, and *getUnicodeStream*
- ▶ You can use the following *PreparedStatement* methods to set the values for parameters that correspond to BLOB columns: *setBytes* and *setBinaryStream*
- ▶ You can use the following *PreparedStatement* methods to set the values for parameters that correspond to CLOB columns: *setString*, *setAsciiStream*, *setUnicodeStream*, and *setCharacterStream*
- ▶ You can retrieve the value of a JDBC CLOB parameter using the following *CallableStatement* method: *getString*
- ▶ You can use the following *ResultSet* methods to retrieve data from a ROWID column: *getBytes* and *getObject*
- ▶ You can use the following *PreparedStatement* methods to set a value for a parameter that is associated with a ROWID column: *setBytes* and *setObject*

Restriction: If you are using IBM DB2 Driver for JDBC and SQLJ type 2 connectivity, you cannot call a stored procedure that has DBCLOB OUT or INOUT parameters.

4.1.4 Specific SQL support for LOBs

In this section, we describe the enhancements to the SQL language that allow you to handle LOB values efficiently.

There are some differences between SQL functions when accessing LOBs and when accessing other normal columns stored in a table. Logically, a row in the base table also contains the LOB value, and this is true from the application's point of view. Physically, DB2 stores them in two different table spaces. You cannot access the auxiliary table via SQL where DB2 stores the LOB values. DB2 protects the auxiliary table by issuing appropriate negative SQLCODE (for instance -766) when you try to access it directly by using SQL.

In general, LOBs can be referenced in all of the string functions with the exception of those that relate to date and time. LOBs have the same set of restrictions as other long strings. Some functions cannot be used on LOB columns for obvious reasons. You cannot use the following functions on LOB columns:

- ▶ GROUP BY clause
- ▶ HAVING clause
- ▶ ORDER BY clause
- ▶ SELECT DISTINCT clause
- ▶ Column function
- ▶ Datetime function
- ▶ DECIMAL or NULLIF function

- ▶ WHEN clause of a CASE expression
- ▶ Subselect of a UNION without the ALL keyword

Most of the restrictions on LOB values are due to the fact that LOB values cannot be compared, except with the LIKE predicate.

LOB functions

One way of manipulating large objects without retrieving the entire object is to use functions. Many of the string functions also work with LOBs. The built-in functions allow you to concatenate strings, get a substring, find the LOB length, find the position in the LOB of a search string, and cast the LOB to another type. UDFs can be used as well.

The list of available LOB functions includes:

- ▶ CONCAT
- ▶ SUBSTR
- ▶ LENGTH
- ▶ POSSTR
- ▶ IFNULL
- ▶ VALUE / COALESCE
- ▶ Casts
- ▶ UDFs
- ▶ LIKE within predicates

In the following sections, we discuss the most important of the functions listed above.

How CONCAT operates

The CONCAT function allows you to put two strings together so that they end up as one string. You can do the same with your host language, but depending on what language you use, there are some limits within which you cannot move easily. For example, you might not be able to concatenate two host variables each containing 80 MB of data, because the resulting variable would be bigger than the maximum allowed size of a variable, as allowed by Enterprise COBOL for z/OS. But be aware not to use the CONCAT function every time you want to string two short variables together (such as two variables of 80 bytes each), because invocation of SQL is more expensive than a single MOVE statement in COBOL, for example. Try to avoid unnecessary use of this function and use simple statements in your host-language as long as you can afford it.

An SQL statement without any data access such as this one uses about 7,500 machine instructions more than the same MOVE statement without any SQL invocation. The amount of data being moved is nearly the same, so you can assume the mentioned overhead for concatenating variables using SQL compared to a MOVE statement.

A few details about SUBSTR

Your application programs can use the SUBSTR function to retrieve part of a LOB value. The string delivered to your application can consist of up to 2,147,483,647 bytes, so there are no limits in accessing parts of your LOBs. When your declared LOB column is smaller than 2 GB, the maximum value returned by the SUBSTR function is the declared size of your large object.

Note: Specifying SUBSTR for a LOB value forces DB2 to read the LOB value until it finds the start position of your SUBSTR statement. Depending on the value of the start position, the statement can affect performance.

The built-in function POSSTR

You can use the POSSTR function to locate the starting position of one string within another string. POSSTR returns the first occurrence of one string. The string you want to locate (the *search string*) in the source string can consist of up to 4,000 bytes, which can also be represented by a host variable. The POSSTR function returns the position of your search string for BLOB and CLOB values. For DBCLOBs, a returned position is a DBCS character. For finding the second, third, or n^{th} occurrence of a string using POSSTR, refer to “The built-in function POSSTR” on page 71.

IFNULL, VALUE, and COALESCE

When you use the IFNULL function in your select statement, you can tell DB2 which value you want it to return to your application when an accessed LOB value is null. For example, you can return text saying ‘value unknown’ to your application if a particular LOB value is null. You can also return the value of an already assigned LOB locator, probably pointing to a picture saying “Image not found”.

```
IFNULL (:LOB-LOCATOR, 'unknown value')
IFNULL (:LOB-LCATOR, :LOCATOR-IMAGE-NOT-FOUND)
```

IFNULL is identical to the COALESCE and VALUE functions, except that IFNULL is limited to two arguments instead of multiple arguments. Because a NULL indicator is one of the two stored flags in the base table, DB2 does not need to access the auxiliary table at all, if a LOB column is stored as a null value.

Using CAST for LOBs

You can also use CAST functions on your LOB values or LOB locators, even to convert your current LOB value into another value. They can be used to get around some of the restrictions on LOB values. To give you an idea of CASTing between data types, assume the following statement:

```
SELECT LOB FROM BASE_TABLE
WHERE SUBSTR (LOB,1,11) = 'IBM Redbook'
```

You could expect that the statement returns to you all of the LOB values, which start with ‘IBM Redbook’. Because large objects are subject to long string column restrictions, DB2 issues SQLCODE -134, complaining about an improper use of a long string column. To solve this problem, you can replace the statement using the following syntax:

```
SELECT LOB FROM BASE_TABLE
WHERE CHAR (SUBSTR (LOB,1,11)) = 'IBM Redbook'
```

This statement delivers the result you could have expected when you issued the first statement.

Table 4-2 on page 72 shows the list of the allowed LOB conversions.

Table 4-2 Casting large objects

BLOB	CLOB	DBCLOB
CHAR VARCHAR CLOB GRAPHIC VARGRAPHIC DBCLOB BLOB	CHAR VARCHAR CLOB GRAPHIC (*) VARGRAPHIC (*) DBCLOB (*)	CHAR (**) VARCHAR (**) CLOB (**) GRAPHIC VARGRAPHIC DBCLOB
(*) CAST is only supported if the data is Unicode. (**) CAST is only supported if the data is Unicode. The result length for these casts is 3 * LENGTH (<i>graphic string</i>)		

The reason for the length growing up to three times with DBCLOB conversions into a CHAR, VARCHAR, or CLOB column is to allow for expansion when the data is converted from UTF-16 to UTF-8. A character that takes two bytes to represent in UTF-16, can take three bytes in UTF-8.

How does LIKE operate on LOBs

Depending on the value specified for LIKE search arguments, LOB Manager takes different actions. These actions correspond to common LIKE operations used for non-LOB values. When you issue LIKE 'Chapter 8%', only the first nine bytes of a LOB value according to qualifying base table rows are scanned to verify the result. A LIKE '%Chapter 8%' clause can be worse, because now DB2 has to scan the entire LOB value until it finds the first occurrence of the string to qualify the LOB value for your SQL statement. This kind of statement can be very time consuming, depending on your average LOB size and the number of touched rows in the base table.

In general, after issuing an SQL statement against the base table, all base table columns appearing in the WHERE clause of your statement are checked before a condition on the auxiliary table is checked, even if the columns in the base table are not indexed. This avoids unnecessary scans of your LOB values and only those LOB values are scanned, which already have qualified from the base table point of view.

4.1.5 Functions such as XML2CLOB

DB2 V8 provided the function XML2CLOB to allow the retrieval of data from an XML data type. The resulting encoding scheme of the character set is UTF-8. The nodes of the XML data are traversed and the result returned as a character string.

An example of usage of the XML2CLOB function is shown in Example 4-1, where we also use the XMLELEMENT to create the XML object for conversion.

Example 4-1 Usage example of XML2CLOB function

```

SELECT c.id,XML2CLOB(
      XMLELEMENT ( NAME "ID Number",
                    c.cname || ' ' || c.status
                  ) ) AS "Company"
FROM companies c;

```

The query can result in the following output shown in Example 4-2 on page 73.

Example 4-2 Example output from XML2CLOB function

id	Company
0002344	Onesteel Ltd
0012311	Intradata Ltd

DB2 9 introduces the XMLSERIALIZE function to produce the XML data in string format, and this replaces the XML2CLOB functionality. The XML2CLOB function however remains for compatibility.

4.1.6 Stored procedures

All stored procedures support LOBs as parameters. When LOBs are passed as an argument to a stored procedure or a user function, they are materialized.

Although LOB file reference variables are supported in host languages, they were not implemented in native SQL language. LOB file references are not data types, just constructs, thus, they cannot be used in native SQL stored procedures. If you still want to use them in your stored procedures, you have to code your stored procedure in any other supported language and pass the file name as a character parameter to a stored procedure.

We recommend using LOBs in stored procedures to manipulate the LOBs within stored procedures, and avoid externalizing them. One example of bad programming would be retrieving LOBs to the program using a stored procedure and not a standard SELECT SQL statement. In this case, the object is materialized twice - once in DBM1 region above the 2 GB bar and once in the program address space. The whole LOB is moved between DBM1 address space to WLM stored procedures address space and back to DBM1.

Note that, with DB2 9, stored procedures implemented with Stored Procedure language execute in the DBM1 address space, rather than in address spaces managed by WLM.

4.2 LOB locators

You can use locators everywhere in your application program where you can access a LOB value by itself. The idea behind locators is that an application program only deals with a reference to a particular LOB and DB2 performs the real operations on the LOB value. Using this method, the LOB value is not stored in the application's memory. To deal with a locator instead of the entire value, an application program simply selects the LOB column into a LOB locator host variable, not into the generated LOB host variable. By selecting into a locator variable, DB2 assigns a 4-byte value to the locator, which is delivered to the application. The entire LOB value is not retrieved or delivered to the application. By selecting the whole LOB into a locator variable, you avoid materialization of the LOB, because DB2 does not have to access every page of the LOB, it just keeps the reference.

For example, when selecting a LOB value, an application program could select the entire LOB value and place it into a host variable of the same size as the LOB (which is acceptable if the application program is going to process the entire LOB value at once), or it could instead select the LOB value into a LOB locator. Then, using the LOB locator, the application program can issue further database operations on the LOB value (such as applying scalar functions SUBSTR, CONCAT, LENGTH, doing an assignment, searching the LOB with LIKE or POSSTR, or applying UDFs against the LOB) by supplying the locator as input. The resulting output of the locator operation, for example, the amount of data assigned to the application's host variable, would then typically be only a subset of the input LOB value.

Once a locator is set to a particular LOB value, there is no action you can take within the database to change that value from the application's point of view until the locator is freed. You can free a locator by issuing a `FREE LOCATOR` statement or by completing the current unit of work.

But how does this work? Locators do not force extra copies of the data in order to provide this function, but maintain a relationship to the underlying data pages. The data remains consistent despite other activity in the system, with the underlying mechanism varying according to the DB2 version.

DB2 V8 uses an S-LOB lock when you assign a locator to a specific LOB value (see 4.5.1, “Locking for LOBs with DB2 V8” on page 95 for a description of LOB locks); the LOB value is released only at `COMMIT` time or when you explicitly free the locator using the `FREE LOCATOR` statement. Therefore, the data pages containing the LOB value cannot be overwritten with different content as long as they are referenced.

When you issue the `HOLD LOCATOR` statement, an assigned locator survives the current unit of work and is valid until the thread terminates or a `FREE LOCATOR` statement is passed to DB2. Either one of these events releases the S-LOB lock taken on the LOB value in the auxiliary table.

DB2 9 does not allow space reuse for LOB data pages before the oldest reader in the system has completed its workload using RLSN comparison techniques; therefore, your application can access the referenced LOB value to finish its work. You can find more details in 4.5.2, “Locking for LOBs with DB2 9” on page 101.

Note: In DB2 V8, the space in a LOB data page can be reused if the S-LOB lock is released using, for example, a `FREE LOCATOR` statement. In DB2 9, DB2 relies on the oldest reader in the system to decide if LOB data pages can be reused, because there is no more S-LOB lock in the auxiliary table space for `SELECT` operations, except for the unconditional lock for UR readers.

4.2.1 Getting to know LOB locators

You assign a locator to a LOB value when selecting a LOB column into a LOB locator. Using this method, DB2 detects that the entire LOB is going to be selected into a locator column, and therefore, it does not materialize the associated LOB value and also does not provide the LOB value to your application program. Instead, DB2 provides a value for your locator into your locator host variable where your application is selecting the LOB value into. You can use a LOB locator anywhere a LOB value can be used, for instance, in expressions where you would normally use an entire LOB value.

Can the LOB value change while a locator is assigned

Simply speaking, yes, it can. But it does not change for your application program, once you have associated a locator to a particular LOB value. So what is DB2 doing when you have a locator assigned to a LOB value (remember, a locator represents a LOB value at a point in time) and someone else is updating or deleting the LOB? DB2 ensures that the locator still represents the original value that existed at the time the locator was set. See Figure 4-1 on page 75.

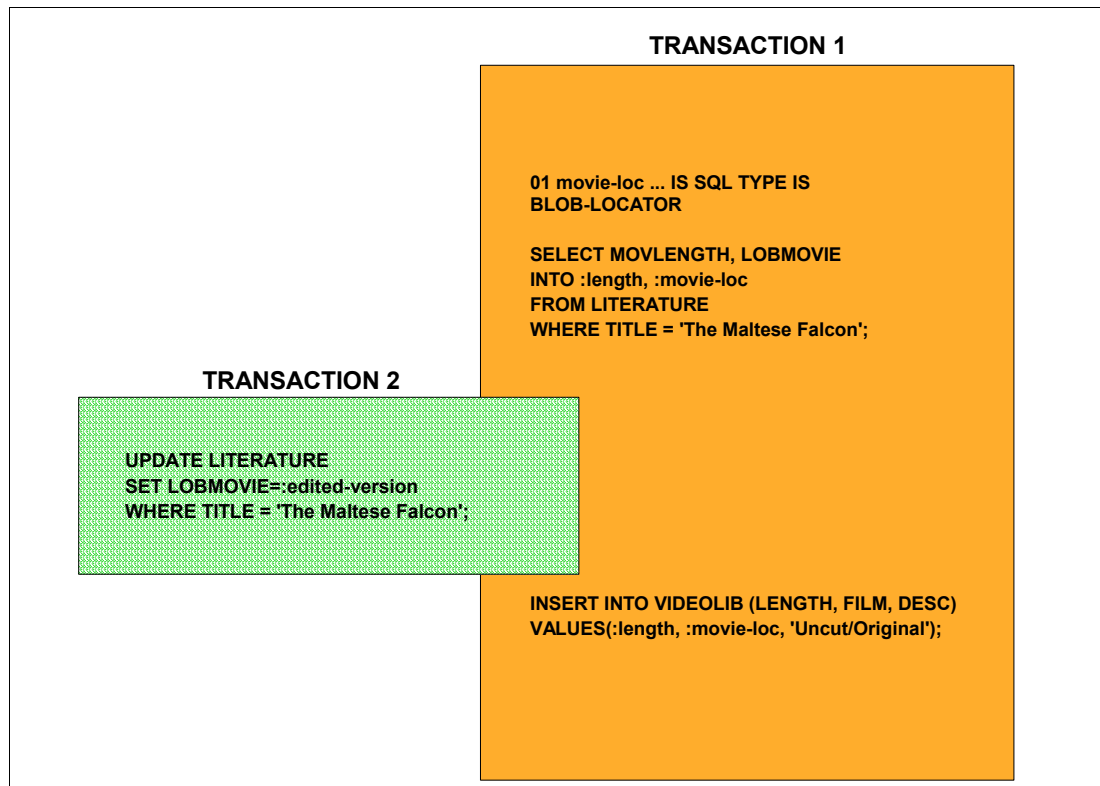


Figure 4-1 Concurrent LOB access using LOB locators

In this example, transaction 1 selects a non-LOB column into a host variable and also a LOB column into a locator variable. After the SELECT is completed in transaction 1, transaction 2 updates the LOB column, which is already referenced by a locator in transaction 1. After the LOB is updated by transaction 2, a new value is inserted into the VIDEOLIB table in transaction 1, using the previously assigned locator for the LOB value by DB2. But the value referenced by the locator is still the same as before the updating transaction has updated the LOB value. How is this possible?

Operations on the original LOB value have definitely no effect on the value referenced by a locator.

Let us first have a look at LOB delete. A DELETE statement only deallocates pages in the auxiliary table where the LOB data is stored. But the data still remains in the pages; it is not deleted.

Now let us have a close look at how a LOB is updated. Updating a LOB consists of DELETE and INSERT, so the pages are deallocated and the data still remains in the auxiliary table.

The INSERT statement cannot reuse the previously deallocated pages because of the different mechanisms DB2 uses to prevent space reuse for currently accessed data regarding locators.

The conclusion is that these data pages to which a locator is pointing cannot be allocated again, even if they are going to be deallocated by a DELETE statement, as long as an S-LOB lock persists on the referenced LOB value in DB2 V8 or a reader possibly accesses the value in DB2 9.

Using locators across multiple units of work

A LOB locator is only a mechanism used to refer to a LOB value during a unit of work. Ordinarily, a locator is freed, implying release of acquired space in DBM1 if the locator is not referenced further, when the application COMMITs its data or the associated thread terminates. Acquired locks are also released at COMMIT time as usual. To increase locator durations beyond COMMIT points, a HOLD locator statement can be issued. This statement increases the duration of the specified locator, and the locks held on the locator, until a FREE locator statement is issued for the same locator. SQL ROLLBACK also frees every locator having the hold property. If no FREE locator statement is issued, the locator is valid until the thread terminates. In Example 4-3, you can find the syntax of FREE locator and HOLD LOCATOR statements.

Example 4-3 Syntax for FREE locator and HOLD locator

Hold beyond COMMIT:

```
EXEC SQL
    HOLD LOCATOR :HV-LOCATOR-1, :HV-LOCATOR-2
END-EXEC
```

Free the locator if it is not needed any more:

```
EXEC SQL
    FREE LOCATOR :HV-LOCATOR-1, :HV-LOCATOR-2
END-EXEC
```

A locator is freed when one of the following conditions occur:

- ▶ An SQL FREE LOCATOR statement occurs
- ▶ An SQL ROLLBACK statement occurs
- ▶ The associated thread terminates

If a locator has the hold property, it survives the SQL COMMIT statement. Without the hold property, it does not survive the SQL COMMIT statement. A locator obtains the hold property by the SQL HOLD LOCATOR statement.

If you receive SQLCODE -423 (INVALID LOCATOR VALUE) after you have specified more than one host variable for a FREE LOCATOR statement, only those locators up to the invalid locator are freed. According to this result, when you receive SQLCODE -423 after issuing a HOLD LOCATOR statement, all locators listed in the statement after the first invalid locator are not held.

Can I free a referenced locator

Assume two locators as shown in Figure 4-2 on page 77, where LOCATOR-2 references LOCATOR-1. When you now free LOCATOR-1 using the FREE LOCATOR statement, it does not release any DB2 resources in this case, because LOCATOR-2 still refers to LOCATOR-1. The locator is only freed externally. This means that the application cannot refer to it; when it does, it probably gets SQLCODE -423 indicating an invalid locator. Internally, DB2 keeps LOCATOR-1 around as long as it is referenced.

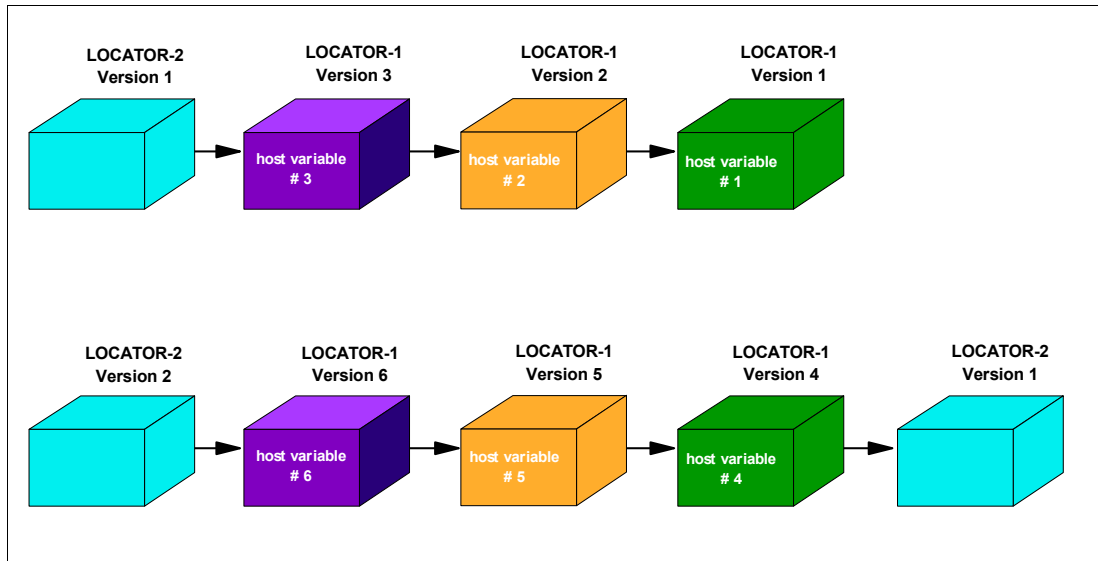


Figure 4-2 Primary chain of locators containing secondary chains

When to use locators

If LOBs are not too large, they can be managed as other data. LOBs can be retrieved, inserted, and updated, just like any other type of data. Everything works well as long as you have enough main storage space to store the LOB. If you think of a CLOB containing a book's text and of an application which has to extract only a single chapter of the book, without the use of locators you have to retrieve the whole object (potentially up to 2 GB of data) in your application and then to extract the data you need. Because of their size, LOBs might be unmanageable for a single application. Huge amounts of storage might be needed to buffer their values, and it might not be possible to acquire contiguous buffers of sufficient size, because this storage must reside within the region size of the address space in which the application program is running. To give you an easy method to deal with LOB columns, DB2 provides locators to make LOB access manageable.

Using a locator can be a good choice in the following situations:

- ▶ Only a part of a LOB is needed by the application.
- ▶ Only a part of a LOB is moved to a client.
- ▶ The entire LOB does not fit into the application's memory.
- ▶ A temporary LOB value is needed, but it does not have to be stored in DB2.

You cannot use any of the locators mentioned above in mathematical operations. The reason is the need to prevent the locators from getting corrupted by the application.

Materialization when using locators

If you do not use the locator technique to access your LOBs, then DB2 has to materialize the value represented by a large object when you access it. In this case, DB2 materializes the LOB by moving it through the buffer pool into the user's address space. Basically, materialization is avoided when you select LOBs or parts of their values into locators. When an application retrieves a LOB value, whether it is assigned to a locator or not, it goes through the buffer pool into the user address space. As soon as a LOB value has to be materialized and a locator is involved, materialization of a LOB takes place in DBM1 above the bar.

This mainly occurs when you update a LOB value by using locators. In this case, the DBM1 address space only contains the LOB's control structure. DB2 tries to avoid materialization wherever it is possible, but if it is no longer possible, it materializes a certain value in DBM1

above the bar or in the user's allied address space, depending on the way a LOB is accessed. For more information about materialization, see 8.1, "LOB materialization" on page 238.

Locators and expressions

LOB locators can also represent more than just base values. They can also represent the value associated with a LOB expression. For example, a LOB locator can represent the value associated with:

```
EXEC SQL
  SET LOCATOR = SUBSTR (LOB1, :START1, :LENGTH1) CONCAT
                  SUBSTR (LOB2, :START2, :LENGTH2)
END-EXEC
```

The same statement can also be issued even if LOB1 and LOB2 are referenced by locators. You are also able to use other LOB functions for further reference in an expression. Regarding this implementation, DB2 provides you with the opportunity to build new locators using other locators and expressions of LOBs or locators. This gives you the flexibility to deal with large objects in various scenarios without allocating huge amounts of storage when you deal with LOBs.

Using locators and SUBSTR considerations

The use of the SQL SUBSTR function to retrieve pieces of the LOB value (for example, VALUES(SUBSTR(:xClobLocator, :xFrom, :xLength)) INTO :szClob:sIndicator) means that the requester must account for the possibility for a partial character in the last few bytes of the chunk of data (due to code page conversion) when it sets the position for the next chunk. Locators, in the current design, can remain active for longer than necessary when not explicitly freed, consuming valuable server resources. For locators referencing a column of a Unicode table in a non-Unicode database, a UTF-8 to database code page conversion is required at the server, and the data is sent to the requester in the database code page with the possibility of data loss. Current workarounds to this problem have their disadvantages, including the possibility of double conversion for certain code pages.

4.2.2 Examples of using locators

A LOB locator is also allowed to represent LOB expressions, this means substrings of an entire LOB value. A *LOB expression* is defined as any expression that refers to a LOB column or results in a LOB data type. LOB functions can also be part of a LOB expression. Also, LOB expressions can reference other LOB locators, which does not simplify this topic at all. A LOB expression is any string expression that contains a LOB value. It is possible to associate the result of a LOB expression to a LOB locator. Because a LOB locator can be used anywhere a LOB value can be used, the LOB expression associated with a LOB locator could make reference to other LOB locators.

Concatenating two strings using locators

When you plan to use any string expressions on a LOB value, you can also use a locator instead of the entire LOB value. For example, when you try to concatenate two LOBs of 10 MB each, you do not have to read them both into two large host variables and string them together, you can instead let DB2 do the work for you.

Assume that LOCATOR-1 is pointing to a CLOB value containing a 10 MB string, and LOCATOR-2 is pointing to another CLOB value of another 10 MB. After assigning both locators, you can simply issue this to string them together:

```
EXEC SQL
  SET :LOCATOR-3 = CONCAT (:LOCATOR-1, :LOCATOR-2)
END-EXEC
```

The new value of both strings, now a total 20 MB, is assigned to the new locator LOCATOR-3, to which you can now refer. To store the new value in a table, you can insert a LOB using the locator as a reference in an insert statement or perform other operations such as SUBSTR.

Referring to a block of data inside of a CLOB

If you want to refer to a single chapter in a document, you can do this also by simply using locators. The first thing an application has to do is to assign a locator to a particular LOB value. The starting position of the chapter you want to refer to is easily determined by the integer value resulting from the POSSTR function as follows:

```
EXEC SQL
  SET :START-POSITION = POSSTR (:LOB-LOCATOR, 'Chapter 8')
END-EXEC
```

You can determine the end of the chapter you want to deal with in the same way using the POSSTR function. In our example, the position of the beginning of Chapter 9:

```
EXEC SQL
  SET :END-POSITION = POSSTR (:LOB-LOCATOR, 'Chapter 9')
END-EXEC
```

After issuing this statement, the locator END-LOCATOR points to the beginning of Chapter 9. When your application wants to deal with Chapter 8 directly, you can set the content of the chapter to a host variable that is large enough to contain the entire text for Chapter 8. Otherwise, simply set the string containing Chapter 8 to a new LOB locator as shown below:

```
EXEC SQL
  SET :CHAPTER-8-LOCATOR = SUBSTR (:LOB-LOCATOR
                                   ,:START-POSITION
                                   ,:END-POSITION - START-POSITION)
END-EXEC
```

The SUBSTR function uses the LOB locator (instead of the whole LOB value) as a string-expression. The START-POSITION represents a starting-position, and the expression END-POSITION - START-POSITION is provided as a length value.

Note Try to avoid the use of string expressions (such as SUBSTR) of BLOB values, because binary documents might not be usable in parts, such as pictures, executable files, or even movies. The use of parts for further processing depends on the type of data you store in your LOB columns.

4.3 DRDA LOB flow optimization

Database applications increasingly store character data in LOB columns and use distributed environments. LOB columns provide additional capacity for the data to grow, as compared to varchar or long varchar columns. LOB columns may be used to store small character strings, serialized Java objects, and XML documents.

The processing of LOBs in a distributed environment with Java Universal Driver on the client side has been optimized for the retrieval of larger amounts of data. This *dynamic data format* is only available for the JCC T4 driver (Type 4 Connectivity). The Call Level Interface (CLI) of

DB2 for Linux®, UNIX® and Windows® also has this client-side optimization. Many applications effectively use locators to retrieve LOB data regardless of the size of the data being retrieved. This mechanism incurs a separate network flow to get the length of the data to be returned, so that the requester can determine the proper offset and length for SUBSTR operations on the data to avoid any unnecessary blank padding of the value. For small LOB data, returning the LOB value directly instead of using a locator would be more efficient, that is, the overhead of the underlying LOB mechanisms can tend to overshadow the resources required to achieve the data retrieval.

For these reasons, LOB (and XML) data retrieval in DB2 9 has been enhanced so that it is more effective for small and medium size objects, and still efficient in its use of locators to retrieve large amounts of data. For small LOBs, the performance should approximate that of retrieving a varchar column of comparable size. This functionality is known as *progressive streaming*. Within the overall dynamic data format of progressive streaming, *progressive reference* is the mechanism that supports the category of large LOB data retrieval.

With the JCC Type 4 driver, a LOB value is associated to one of three categories depending on its size:

- ▶ Small LOBs - DRDA (server) default is 32 KB, the driver can override it by setting smaller values.
- ▶ Medium LOBs - Greater than 32 KB and up to 1 MB size.
- ▶ Large LOBs - Greater than 1 MB and up to 2 GB size.

The large threshold is set as DRDA parameter MEDDTASZ, whose default is 1 MB. This threshold should be set to the maximum storage size the application region can handle, but not less than 32 KB. There is no means to override the small threshold (currently set for performance reasons at 12 KB) set by the Type 4 driver.

Based on the size, the decision is made how to transfer the data. The structure of the data query block varies according to LOB size. When small-sized LOBs are used, they are treated exactly as VARCHAR type and transferred as part of the row, thus gaining performance close to that of retrieving a varchar. When medium-sized LOBs are used, the non-LOB data is placed in the query data block and the LOBs are placed in overflow blocks. These blocks are called *externalized data blocks*. This way, all LOBs are retrieved at once and cached on a client for subsequent processing. For large LOBs, it was found that locators are still the most efficient flow method. Thus, the locators are transmitted in a data query block with the rest of the data, avoiding a need to materialize the entire LOB at once. This explanation is depicted in Figure 4-3 on page 81.

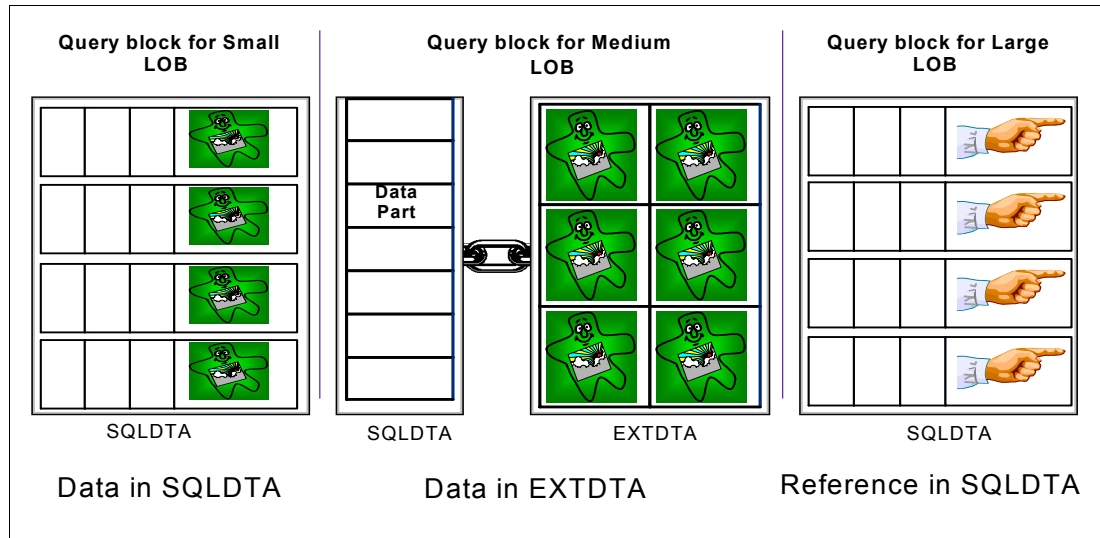


Figure 4-3 Progressive reference return of LOB data

Note: Data streaming is architected in DRDA but is not implemented in a peer to peer DB2 connection where it requires the Java Universal Driver on the client side.

Inserting data LOBs are streamed to DB2 by the client and completely materialized on the server side. The object is “assembled” in the DDF address space and then moved on to the DBM1 address space for further processing.

This mechanism provides a very adaptable flow mechanism to seamlessly change the logic used to retrieve data between that suitable for *real* LOBs, where the data length is indeed large, and data stored within LOB columns that does not really satisfy the description of large. This is particularly useful for data stores where the length of data is subject to wide variations, with DB2 adapting on the fly to minimize the effort to serve the workload it receives.

4.3.1 DB2 Universal Java Driver

Two new datasource properties are introduced in the JCC V3 driver (*progressiveStreaming* and *streamBufferSize*), and the default value for an existing datasource property (*fullyMaterializeLobData*) is changed. The changes are as follows:

progressiveStreaming property

This property introduces the ability for the server to dynamically determine the most efficient mode in which to return LOB or XML data as introduced in 4.3, “DRDA LOB flow optimization” on page 79. When this dynamic data format is enabled, the locator’s life span is the scope of the cursor (cursor-based reference) as opposed to the scope of the transaction. The server frees the reference accordingly at the close (implicit or explicit) of the cursor, or it might free the reference earlier after the rows associated with the reference are returned. A new mechanism is also provided for the requester to retrieve sequential chunks of the LOB data, with the progression of the reference (position in the data) maintained at the server.

Even though random access is not supported on the underlying DRDA progressive reference, the driver still supports random access (by using reset and skip) in JDBC API. However, using random access API on an underlying progressive reference has performance implications.

DB2BaseDataSource.progressiveStreaming can be set to *DB2BaseDataSource.NOT_SET* (default), *DB2BaseDataSource.YES*, or *DB2BaseDataSource.NO*. If the property setting is *NOT_SET* (the default), progressive references are enabled whenever the server supports progressive references; otherwise, progressive references are not enabled by the driver. If the server does not support progressive references, then *DB2BaseDataSource.progressiveStreaming* is ignored.

If progressive references are enabled, then JDBC *executeQuery()* requests the dynamic data format for LOB and XML data with *OUTOVR*OPT = *OUTOVR*NON and *DYN*DTAFMT (Dynamic Data Format) is set to 0xF1 (True). Subsequent *CNT*QRY (Continue Query) requests can then set *FRE*PRVREF (Free Previous References) to enable automatic closure of the locator when there is cursor movement (that is, freeing locators previously returned in complete rows and row sets).

If progressive streaming is disabled, the LOB data is either fully materialized data or locators are used as they were prior to this feature.

Externally, progressive references are closed on cursor movement or cursor closure, but not on stream exhaustion. To summarize, if progressive streaming is disabled, then LOB data is returned as it was prior to this feature, including locators with a transaction life span and support for random access. If progressive streaming is enabled, then LOB and XML data is implicitly closed by the driver upon cursor movement or cursor closure.

streamBufferSize

DB2BaseDataSource.streamBufferSize determines the size of the driver's staging areas for chunking LOB or XML streams, regardless of whether progressive or SQL locators are used. When progressive streaming is requested, data is "inlined" if the stream buffer can accommodate the data size. This setting corresponds to *DRDA* MEDDTASZ. There is no user external for *SML*DTASZ which takes the driver default (currently set at 12 KB). From an external's perspective, the stream chunking size is orthogonal to whether or not data is transported in *QRY*DTA or *EXT*DTA.

fullyMaterializeLobData

DB2BaseDataSource.fullyMaterializeLobData in JCC V3 is ignored by the driver whenever progressive references are enabled. When progressive streaming is used, materialization is controlled by the *DB2BaseDataSource.streamBufferSize* property and the *DB2BaseDataSource.fullyMaterializeLobData* setting is ignored. Notice that progressive reference becomes the default in JCC V3 (against servers that support it), so applications that rely on *DB2BaseDataSource.fullyMaterializeLobData* have to explicitly change the property setting for *DB2BaseDataSource.progressiveStreaming* to *DB2BaseDataSource.NO*. Some indication of the performance advantages can be seen from test cases where the LOB length is varied around the *streamBufferSize* threshold, which effectively compares the old non-progressive mechanism with the new progressive streaming, using varying size LOB workloads.

Figure 8-10 on page 255 shows the effect of varying LOB data sizes with differing flow mechanisms. For small LOB sizes, the longest run time comes from using LOB locators. It is somewhat more efficient in terms of elapsed time to materialize the LOB instead of using a locator, because a degree of overhead is removed from DB2, and a reduction occurs in the number of data flows. Using progressive streaming further improves the performance by further eliminating overheads involved with invoking LOB specific flow mechanisms.

4.4 Feeding a LOB column

Creating the environment for storing LOB values is one thing, providing the values is another. There are at least two different possible locations where the large objects are originating before they are inserted into a LOB column. The choice of how to bring them up to your host environment depends on their location. Even if you have lots of large files already stored in your middleware, you can also consider uploading them using FTP to your host, and then access them with applications running on the mainframe. Applications running in a distributed environment can use the same technique as z/OS applications for inserting LOBs, using distributed relational database architecture (DRDA) connections (for instance, through DB2 Connect™), open database connection (ODBC) calls, or Java using SQLJ or JDBC. Table 4-3 summarizes the alternatives of where LOBs can come from and how to feed them into your LOB columns.

Table 4-3 Where large objects come from and how

Where is the data?	Methods to feed LOBs
Client	Cross-loader, Application, DB2 for LUW import, or DB2 Extenders
Host	LOAD utility or Application

4.4.1 Loading a LOB column using LOAD or the cross loader

Depending on the way the LOB data is provided, there are three ways that the load utility can be used to load LOB data:

- ▶ Loading LOB data as normal data fields from the LOAD input file
- ▶ Using file reference variables when each LOB value is stored in a separate input file
- ▶ Using the cross loader

Note: You always LOAD the data into the base table and DB2 automatically stores the LOB data in the associated auxiliary tables.

Loading LOB data as normal data columns

This method can be used when the LOB values are already stored in the LOAD input file together with the other data fields of the base table. Because the maximum record length of a sequential file in z/OS is 32,760 bytes, this method can only be used to LOAD LOB columns of just < 32 KB. The sum of the length of all normal data fields and LOB fields in the input file cannot exceed 32 KB.

In most cases, this method is only used if the LOBs in the table are small LOBs (SLOBs) with an actual length smaller than 32 KB. The defined length of the LOB column on the CREATE TABLE statement can exceed 32 KB.

The provided input file can be the result of:

- ▶ UNLOAD utility
- ▶ DSNTIAUL utility
- ▶ File generated by another application

Loading LOB data using file reference variables

This method is used when the LOB values are stored in separate input files. The normal input file contains the data for the non-LOB columns of the base table and the names of the LOB files. The sum of the length of all normal data fields and the LOB file names cannot exceed 32

KB. See Figure 4-4 for a simple example of LOAD with file reference variables. More examples are provided in 6.3, "LOAD" on page 171.

LOB LOAD using File Reference Variables

LOAD is changed to allow an input field value to contain the name of file containing a LOB column value. The LOB is loaded from that file.

```
//SYSREC DD *
"000001","UN.DB1.TS1.RESUME(AI3WX3JT)","UN.DB1.TS1.PHOTO(AI3WX3JT)"
"000002","UN.DB1.TS1.RESUME(AI3WX5BS)","UN.DB1.TS1.PHOTO(AI3WX5BS)"
"000003","UN.DB1.TS1.RESUME(AI3WX5CC)","UN.DB1.TS1.PHOTO(AI3WX5CC)"
"000004","UN.DB1.TS1.RESUME(AI3WX5CK)","UN.DB1.TS1.PHOTO(AI3WX5CK)"

LOAD DATA FORMAT DELIMITED
  INTO TABLE MY_EMP_PHOTO_RESUME
  (EMPNO CHAR,
   RESUME VARCHAR CLOBF,
   PHOTO VARCHAR BLOBF)
```

new syntax

Figure 4-4 An example of LOAD with file reference variables

The LOB input files can be any of these types:

- ▶ A sequential file
- ▶ A member of a PDS or PDSE
- ▶ A HFS file on a HFS directory
- ▶ A z/FS file

The LOB input file contains the entire LOB value, and the name of this file is stored in the normal load input file as a CHAR or VARCHAR field. So instead of containing the whole LOB value, the normal input file now only contains a file name which in most cases does not cause the sequential file to hit the 32 KB limit.

Additional keywords have been added to the CHAR and VARCHAR field-specifications of the LOAD utility to support a file name as the input for the actual LOB value:

- ▶ BLOBF: The input field contains the name of a file with a BLOB value.
- ▶ CLOBF: The input field contains the name of a file with a CLOB value.
- ▶ DBCLOBF: The input field contains the name of a file with a DBCLOB value.

In case of CLOBF and DBCLOBF, CCSID conversions are done when the CCSID of the input data is different than the CCSID of the table space. (EBCDIC, ASCII, Unicode, or CCSID keywords might have been specified for the input data. The default is EBCDIC input data). In case of BLOBF, no conversions are done.

When the input field of a BLOBF,CLOBF, or DBCLOBF is NULL, the resulting LOB value is NULL (null indicator field for the CHAR or VARCHAR field specified in the NULLIF keyword is hex FF).

The provided input files can be the result of:

- ▶ UNLOAD utility (PDS members, PDSE members, or HFS files)
- ▶ DSNTIAUL utility (sequential files)
- ▶ File generated by another application

Using the cross loader

The cross loader can be used to load LOB data that resides in an other table on the same or another location. The other table can be on any DRDA server or the result of a request with WebSphere Information Integrator (the follow-on product to Datajoiner). Use a three part table name when accessing remote data with the first qualifier being the remote location.

The data that is loaded is the result set of a SQL SELECT statement specified in the CURSOR specification of the LOAD utility.

When you use the cross loader function in DB2 9 (or in DB2 V7 or V8 with the PTF for APAR PQ90263 applied), the LOB value can be greater than 32 KB. For this method, DB2 uses a separate buffer for LOB data and only stores 8 bytes per LOB column in the cursor result set buffer. The sum of the lengths of the non-LOB columns plus the sum of 8 bytes per LOB column cannot exceed 32 KB. The separate LOB buffer resides in storage above the 16 MB line and is limited by the available memory above the 16 MB line.

For more information about using the cross loader with LOB data, see 6.3.3, “Using the cross loader” on page 172.

4.4.2 Inserting LOBs using the host application

To handle this situation, and assuming that the LOB data is already available in your OS/390 environment, besides using the LOAD utility, you can also use an application program which performs the inserts of your LOBs.

You have to distinguish between three different scenarios:

- ▶ Inserting LOBs using file reference variables, introduced in DB2 9
- ▶ Inserting LOBs using a *host variable* large enough to hold the entire LOB *value*
- ▶ Inserting LOBs using a *host variable* **not** large enough to hold the entire LOB value, and introducing the use of *locators*

Inserting LOBs using a file reference variable

The recommended and most likely the easiest way to feed your LOB columns from inside your application is by simply passing the file to DB2 and inserting it into the LOB column.

Example 4-4 shows you the pseudo code of inserting a LOB value using a file reference variable.

By using a file reference variable, you do not have to ensure that the amount of memory you might need to hold the entire LOB value is available in your user address space, because the LOB value is passed through DBM1 before it is written to a file.

Example 4-4 Loading LOB data using File Reference Variable

```
*****
010700      * COBOL DECLARATIONS FOR TABLE PAOLO.BLOB_BASE_TABLE      *
010800      *****
010900
011000      01 BASE-TABLE-AREA.
011100          05 LO-DOCUMENT-NR          PIC X(10).
011200          05 LO-DESCRIPTION          PIC X(32).
011210
011220      01 LOBDATA                      USAGE IS SQL TYPE
011230                                IS BLOB-FILE.
011300
....
```

```

.....
014200 *****
014300 *
014400 *      INITIALIZATION OF USED VARIABLES
014500 *
014900
015000      INITIALIZE INTERNAL-VARIABLES
015100      BASE-TABLE-AREA
015200      LOBDATA
015500
015600      MOVE '0000000001'          TO LO-DOCUMENT-NR
015700      MOVE 'FIRST LARGE OBJECT' TO LO-DESCRIPTION
015710      MOVE 'PAOLOR1.DB2V8.PDF' TO LOBDATA-NAME
015720      MOVE 17                  TO LOBDATA-NAME-LENGTH
015730      MOVE SQL-FILE-READ        TO LOBDATA-FILE-OPTION
.....
.....
021500 *****
021600      INSERT-BASE-TABLE SECTION.
021700 *****
021800 *
021900 *      INSERT ROW INTO BLOB_BASE_TABLE
022000 *
022100
022200
022300      EXEC SQL
022400          INSERT INTO PAOLO.BLOB_BASE_TABLE
022500              (DOCUMENT_NR
022600               ,DESCRIPTION
022700               ,DOCUMENT)
022800          VALUES
022900              (:LO-DOCUMENT-NR
023000              ,:LO-DESCRIPTION
023100              ,:LOBDATA)
023200      END-EXEC

```

Note: Make sure that your DSNZPARMs LOBVALA and LOBVALS are set appropriately by using file reference variables to handle large files.

Important: Using a file reference variable requires that the source file has a record format of VB. Using a file with record format FB results in SQLCODE -452 with reason 12.

Inserting LOBs using a host variable

Inserting a LOB already available at the host using a host variable is the second way to feed your LOBs. There are two main techniques for how to move a LOB value to your host variable. The first one requires a data set containing the entire LOB value in one row. The only thing the application has to do is read a file, move the value to the assigned host variable, provide the correct value to the length field, and execute the SQL INSERT statement. The second technique consists of having a data set where a certain number of rows have to be tied together to build the entire LOB. This might be a more common method because of the maximum MVS allowed record length of 32,760 bytes, which is really a *small* LOB. The rows can be tied together in one variable as shown in the pseudo code in Example 4-5 on page 87.

Example 4-5 Rows tied together in one variable

```
MOVE 0                                TO HV-LOB-LENGTH
PERFORM UNTIL EOF-INPUT
  READ FILE INTO INPUTRECORD

  IF NOT EOF-INPUT
    MOVE LENGTH OF INPUTRECORD        TO LENGTH
    MOVE INPUTRECORD                  TO HV-LOB-DATA (HV-LOB-LENGTH + 1:LENGTH)
    ADD LENGTH                        TO HV-LOB-LENGTH
  END-IF
END-PERFORM
```

The definition of a host variable, large enough to contain an entire LOB value, is reported in Example 4-6. You need to include it in your WORKING-STORAGE SECTION of your application program if you try to insert a 1 MB CLOB, as in our example.

Example 4-6 Host variable declaration for a LOB column

```
01 HV-LOB          USAGE IS SQL TYPE
                   IS CLOB (1M).
```

Using this syntax, DB2 generates the definition reported in Example 4-7 for your application program.

Example 4-7 What the DB2 precompiler makes of your host variable declaration

```
01 HV-LOB.
02 HV-LOB-LENGTH      PIC S9(9) COMP.
02 HV-LOB-DATA.
  49 FILLER           PIC X(32767).
    [repeated 32 times]
  49 FILLER           PIC X(32).
```

The last filler is used to match exactly the requested host variable size of 1 MB as declared for the CLOB. Be aware that your application only defines host variables in a size that you are allowed to acquire, regarding your JCL and your system. You must acquire buffers large enough to store your LOBs. This can be difficult for very large LOBs. For example, Enterprise COBOL for z/OS has significantly raised the limits to support DB2 applications, for example, the maximum data-item size has been raised to 128 MB, which is also the compiler limit for WORKING STORAGE SECTION. The maximum PICTURE clause and the maximum OCCURS integer have been raised to 134,217,727. Because of this limitation, for LOBs bigger than 128 MB, you have to use another method for feeding your LOBs. See “Inserting LOBs using locators” on page 88 for details. The application has to ensure the delivery of a correct value in the length field of your LOB host variable.

Once the host variable contains the entire LOB, you can simply issue an SQL INSERT on the base table to pass your LOB to DB2. You do not need to worry about storing LOBs in the auxiliary table, because this is DB2’s job. From the application programmer’s point of view, there is only one table containing all of the columns that are being used for LOB processing. The statement shown in Example 4-8 is a possible way for you to insert your LOB data.

Example 4-8 Inserting a single LOB value using one host variable

```
EXEC SQL
  INSERT INTO BASE_TABLE
    (COL1, COL2, LOB)
  VALUES
    (:HV-COL1 ,:HV-COL2 ,:HV-LOB)
```

We recommend that an application commits after completing the unit of work while inserting a LOB, because COMMIT releases locks taken during insert and frees allocated storage in use by the LOB. A sample program showing how to insert a LOB using one host variable is included for BLOBs and CLOBs in the additional material described in Appendix A, “Additional material” on page 259.

The only restriction is that inserting LOBs bigger than the maximum supported variable size of your programming language is not possible using this method. If you want to insert a LOB larger than the maximum supported variable size of your application language, refer to the section “Inserting LOBs using locators” on page 88.

Note: Make sure that you use the largest host variable you can afford to keep the length of a locator chain as small as possible before you reach your maximum LOB size, because the intended use of locator chains is not dealing with a large number of interrogations. For example, it is definitely better to use 100 times a 1 MB host variable in a locator chain than using 10,000 times a 10 KB host variable to achieve the same result.

Inserting LOBs using locators

If you are not able to acquire a buffer large enough to hold your LOB data, you have to move it in pieces into a DB2 LOB column. This technique is supported by using LOB locators. We show two examples of pseudo code using locators. In the first one, we append values in single chain passing through an intermediate locator until we are ready to insert the full LOB; in the second one, we use instead multiple chains of locators.

Single locator chain

For this case, see the pseudo coding reported in Example 4-9 for a suggestion about inserting large LOBs.

Example 4-9 Pseudo code inserting LOBs with one locator chain

Definitions:

HV-LOB	USAGE IS SQL TYPE IS CLOB (10M)
LOB-LOCATOR-1	USAGE IS SQL TYPE IS CLOB-LOCATOR
LOB-LOCATOR-2	USAGE IS SQL TYPE IS CLOB-LOCATOR

Pseudo-Code:

```
MOVE 0 TO HV-LOB-LENGTH
EXEC SQL
    SET :LOB-LOCATOR-1 = ''
END-EXEC

PERFORM UNTIL EOF-INPUT
    READ FILE INTO INPUTRECORD

    IF NOT EOF-INPUT
        PERFORM BUILD-HOST-VARIABLE
    END-IF
END-DO

PERFORM FINAL-INSERT

:BUILD-HOST-VARIABLE
    MOVE LENGTH OF INPUTRECORD      TO LENGTH
```

```

MOVE INPUTRECORD          TO HV-LOB-DATA (HV-LOB-LENGTH + 1:LENGTH)
ADD LENGTH                TO HV-LOB-LENGTH

IF HV-LOB-LENGTH > 10000000 THEN
    PERFORM APPEND-LOCATOR

    MOVE 0                TO HV-LOB-LENGTH
    MOVE SPACE            TO HV-LOB-DATA
END-IF

:APPEND-LOCATOR
EXEC SQL
    SET :LOB-LOCATOR-2 = CONCAT (:LOB-LOCATOR-1, :HV-LOB)
END-EXEC

EXEC SQL
    FREE LOCATOR :LOB-LOCATOR-1
END-EXEC

MOVE LOB-LOCATOR-2        TO LOB-LOCATOR-1

:FINAL-INSERT
IF HV-LOB-LENGTH > 0 THEN
    PERFORM APPEND-LOCATOR
END-IF

EXEC SQL
    INSERT INTO BASE_TABLE
        (KEYCOL1, COL2, LOB)
    VALUES
        (:HV-KEYCOL1, :HV-COL2, :LOB-LOCATOR-1)
END-EXEC

EXEC SQL
    FREE LOCATOR :LOB-LOCATOR-1
END-EXEC

```

At the top of Example 4-9 on page 88, you can find the declaration of a CLOB host variable. In this case, the size is 10 MB, because we assume 10 MB as the maximum usable host variable size in our environment. The associated locators are also defined at the beginning of the pseudo-code.

The first initialization of LOB-LOCATOR-1 is done by its first reference in the CONCAT statement. Then the application reads the input file in a loop and it builds a host variable using the rows from the input file until an end-of-file condition occurs. After reading one input record, it is appended to our 10 MB CLOB host variable. After the host variable is nearly filled up (CURRENT-SIZE > 10000000, the correct value depends on the possible length of your input records), the host variable is appended to LOB-LOCATOR-1 using the CONCAT function of DB2 to the already existing LOB value, where LOB-LOCATOR-1 is referring to and set to LOB-LOCATOR-2. The locator is not assigned to a specific table yet, only the reference to a value inside of DB2 is created. Using this technique, LOB materialization takes place outside of the application in DBM1 for building the entire LOB.

If the host variable is filled with some data even after an end-of-file condition is detected, the content of the host variable is applied one last time to the locator. After all records are appended to your LOB locator (ensure that the correct record length is passed to DB2), the application finally inserts the LOB, using the locator to provide the needed host variable for the LOB value.

Attention should be paid to the FREE LOCATOR statement. If you do not FREE the used locators, DB2 tends to keep them around in buffers allocated in the DBM1 address space where the locators reside, and this can cause problems in already constrained virtual storage environments. The intent of the FREE LOCATOR statement is to release the allocated virtual storage space used by the locator itself as well as the virtual storage buffers in DBM1 containing the data referenced by the locator. Unless the LOCATOR has been defined with HOLD, an SQLCOMMIT also frees the locator. If you do not free the locators after using them, the storage remains allocated until COMMIT.

Make sure to issue the FREE LOCATOR statement for each append within the concatenation loop as well; this decreases a counter used by DB2 to control the structure built to reference the first locator. This minimizes the risk of causing virtual storage problems when inserting many LOBs without issuing a COMMIT between your LOB insert statements. We recommend that you COMMIT after each single LOB is inserted by your application. The MOVE simply moves the value of our second locator to the first locator, which makes the first locator available again, but with the value of the second locator, which we use temporarily for the concatenation.

We have used a final insert statement at the end of the sample program, because we want to avoid long lock durations on the base table. Another technique is to insert the first input record into the table, assign a locator to the LOB value, and start building the locator for a final update on the LOB column using the locator. When you plan to use the updating method, be aware that you hold a lock on the base table and on the auxiliary table in DB2 V8. In DB2 9, inserting partial LOB data can result in data corruption for UR readers accessing the incomplete LOB data. The exclusive lock on the base table can prevent access by other users to the base table depending on the lock size you use.

Error-handling routines and possible data movement to non-file-variables are not mentioned in the example above.

A sample program showing how to insert a LOB using a 10 MB host variable and a single locator chain is included in the sample files in Appendix A, “Additional material” on page 259.

Multiple locator chains

The method described in “Single locator chain” on page 88 is fast, but there is also one disadvantage while using it. Every pass through the loop acquires a new LOB locator also pointing to a previous one. Using this method, DB2 builds a chain of locators. Each locator is associated to a control structure that describes how to construct the new LOB value by concatenating a value copied from HV-LOB-DATA into DBM1 with the contents of the identified LOB locator. This particular use of LOB locators does not hurt in terms of additional storage acquired in the DBM1 address space, the agent's LOB storage limit, and the system's storage limit. But it is not too hard to imagine situations where you could still run out of space very quickly without proper locator management.

Each time a concatenation is added to the chain, the entire chain is interrogated using recursion. Let us consider the chains we build using the concatenation technique as *secondary chains*. See Figure 4-5 on page 91 for a better understanding of what happens in DBM1 when you use the appending mechanism as mentioned above.

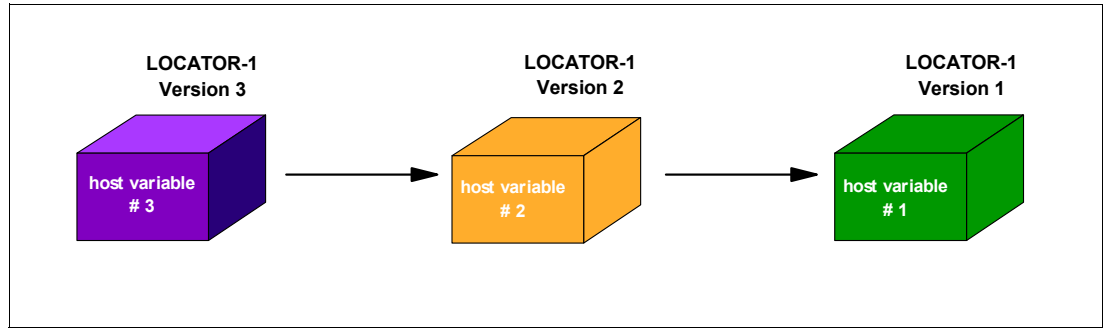


Figure 4-5 Secondary chain of locators

After assigning the first version of a locator to a host variable, the locator only contains the first host variable (Version 1). After issuing the first concatenation of `:LOCATOR-1 = CONCAT (:LOCATOR-1, :HV-LOB)`, the second version of the same locator is created. Version 2 of the locator contains the new host variable and points to Version 1 of the same locator. So DB2 builds a chain of locators. Appending data to a locator becomes more expensive when the chain grows, because of the recursive interrogation technique we mentioned above. You can especially run into a long chain when you build a large object of comparatively small input records only using locators. This is not the recommended way to feed LOBs, so use a reasonably sized host variable to minimize the length of the locator chain.

To improve performance, we can build a *primary chain* containing the previously built secondary chain to reduce the number of recursions when appending a new host variable to LOCATOR-1. So DB2 only has to go through the long chain containing all host variables when we append a secondary chain to a primary chain. See Figure 4-6 for a visualization of primary and secondary chains.

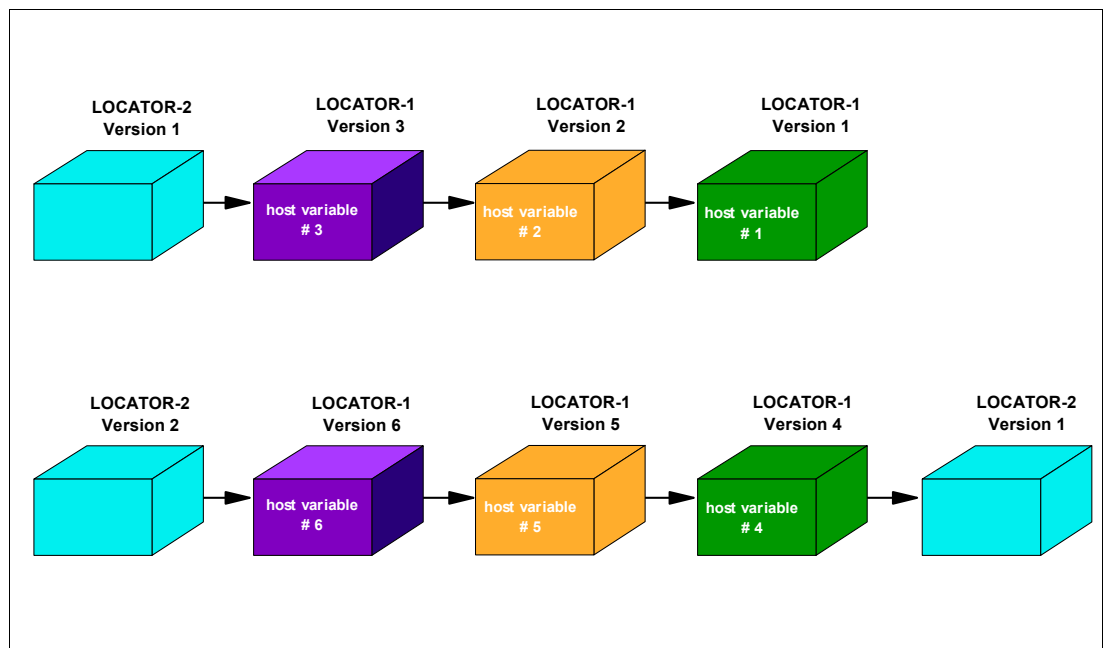


Figure 4-6 Primary chain of locators containing secondary chains

As soon as a secondary chain becomes very large, we can source it out to a primary chain to reduce the level of recursion we have to go through when we append another host variable to our secondary chain. You can use this method when appending big host variables to a single

locator does not satisfy your performance requirements after a large number of concatenations for a secondary chain.

The pseudo code reported in Example 4-10 shows you how to perform the same insert as above including a technique dealing with primary and secondary chains.

Example 4-10 Pseudo code inserting LOBs with multiple locator chains

Definitions:

```
HV-LOB          USAGE IS SQL TYPE IS CLOB (100K)
LOB-LOCATOR-1   USAGE IS SQL TYPE IS CLOB-LOCATOR
LOB-LOCATOR-2   USAGE IS SQL TYPE IS CLOB-LOCATOR
```

Additional instruction before reading the 1st INPUTRECORD:

```
EXEC SQL
  SET :LOCATOR-2 = ''
END-EXEC
```

Pseudo-Code:

```
:APPEND-LOCATOR
  ADD 1          TO APPEND-LOCATOR-COUNTER

EXEC SQL
  SET :LOB-LOCATOR-1 = CONCAT (:LOB-LOCATOR-1, :HV-LOB)
END-EXEC

IF APPEND-LOCATOR-COUNTER = 1000 THEN
  EXEC SQL
    SET :LOB-LOCATOR-2 = CONCAT (:LOB-LOCATOR-2, :LOB-LOCATOR-1)
  END-EXEC

  EXEC SQL
    SET :LOB-LOCATOR-1 = ''
  END-EXEC

  MOVE 0          TO APPEND-LOCATOR-COUNTER
END-IF

:FINAL-INSERT
IF HV-LOB-LENGTH > 0 THEN
  PERFORM APPEND-LOCATOR
END-IF

IF APPEND-LOCATOR-COUNTER > 0 THEN
  EXEC SQL
    SET :LOB-LOCATOR-2 = CONCAT (:LOB-LOCATOR-2, :LOB-LOCATOR-1)
  END-EXEC
END-IF

EXEC SQL
  INSERT INTO BASE_TABLE
    (KEYCOL1, COL2, LOB)
  VALUES
    (:HV-KEYCOL1, :HV-COL2, :HV-LOB-LOCATOR-1)
END-EXEC

EXEC SQL
  FREE LOCATOR :LOB-LOCATOR-1, :LOB-LOCATOR-2
```

The difference between these two examples is the locator management in the APPEND-LOCATOR subroutine. After 1,000 times of appending a host variable (which means nearly 100 MB of data in our example), we create a primary chain by issuing SET :LOCATOR-2 = CONCAT (:LOCATOR-2, :LOCATOR-1). After LOCATOR-1 was outsourced successfully, we can reset it to point to an empty string to start building a new secondary chain.

It is a good programming practice to free both locators after the final insert statement in order to release the virtual storage allocated out of the DBM1 address space, and make it available for further usage in the application. Otherwise, the storage remains allocated until the implicit or explicit COMMIT releases the entire allocated storage.

Before you start coding your application, consider that every invocation of SQL takes a certain amount of time. So try to keep SQL calls to a minimum number and exploit as much storage as you can get for your host variable. This results in a reduction of SQL calls and leads you to a better performing application. But, also be aware of storage allocation issues if you do not free your locators.

Important: Do not make extensive use of LOB locators by concatenating small pieces of storage. Instead, concatenate the largest host variable you can afford to acquire to benefit from performance and reduce the risk of virtual storage constraints.

A sample program showing how to insert a LOB using a 10 MB host variable and two locator chains is included in the sample files described in Appendix A, “Additional material” on page 259.

4.4.3 DB2 for Linux, UNIX and Windows import

If you have DB2 or DB2 Connect installed on a UNIX or Windows platform, you can upload files containing LOBs that are stored on this platform to the host using the DB2 IMPORT command.

The DB2 export and import utilities allow you to move data from a DRDA server database to a file on the DB2 Connect workstation, and the reverse. You can then use the data with any other application or relational database management system that supports this export or import format.

For LOB data, it operates the same way as the DB2 LOAD utility on the host. Each LOB value must be stored as a separate file on a LOB directory and the normal input file contains the non-LOB data and the names of the LOB files.

In Example 4-11, we show an example of an IMPORT command that loads the host table X.MYTABLE from the DB2 Connect workstation. This command can be issued from within the command line processor (CLP) after the connection to the host DB2 system has been established using the CONNECT statement.

Example 4-11 IMPORT command

```
import from filename of ixf lobs from lobpath insert into X.MYTABLE
```

Explanations of Example 4-11 on page 93 are:

- ▶ **filename**: Contains the name of the IXF input file that contains the data to be loaded (data of the base table columns and the names of the LOB files). If the path is omitted, the current working directory is used.
- ▶ IXF: File format of the input file supported by DB2 on the host (IXF is integrated exchange format, in which most of the table attributes are saved).
- ▶ **lobpath**: Specifies the path where the LOB files are stored. The names of the LOB data files are stored in the main input file in the column that is loaded into the LOB column.
- ▶ **into**: Specifies the host table or host view to be loaded.

The best way to create a working base for the IMPORT command is to first EXPORT a similar table from the host to the workstation using an EXPORT command as shown in Example 4-12. Similarly, you first have to CONNECT to the host DB2 system.

Example 4-12 EXPORT command

```
export to filename of ixf lobs to lobpath lobfile lobfile modified by  
lobsinsefiles select * from X.MYTABLE
```

The explanations for Example 4-12 are:

- ▶ **filename**: Contains the name of the IXF file to which data is to be exported. If the complete path to the file is not specified, the export utility uses the current directory and the default drive as the destination.
- ▶ **lobpath**: Specifies a path to a directory in which the LOB files are to be stored.
- ▶ **lobfile**: Contains the base file name for the LOB files. When creating LOB files during an export operation, file names are constructed by appending the current base name from this list to the current path from lobpath and then appending a 3-digit sequence number. For example, if the current LOB path is the directory c:\u\davy\lobpath, the LOB files created are c:\u\davy\lobpath\lobfile.001.lob, c:\u\davy\lobpath\lobfile.002.lob, and so forth.
- ▶ **lobsinsefiles**: Required parameter to specify that the LOBs are stored in separate files on the lobpath.
- ▶ **From**: Specifies a host table or host view containing the columns to unload from the host.

Afterwards, you can create a similar IXF file on the workstation and edit it with the names of the LOB files you want to upload.

See the DB2 for Linux, UNIX and Windows manuals for complete information about the IMPORT and EXPORT commands.

Tip: Be sure that the input file does not contain values for the ROWID column. These are rejected by DB2 on the host when the ROWID column was defined there as GENERATED BY DEFAULT.

4.5 Locking

Locking for LOBs has significantly changed in DB2 9; therefore, the information about LOB's locking techniques with DB2 V8 and DB2 9 are mentioned in two separate sections.

For LOB locking with DB2 V8, see 4.5.1, "Locking for LOBs with DB2 V8" on page 95. Locking for LOBs beginning with DB2 9 is described at 4.5.2, "Locking for LOBs with DB2 9" on page 101. Note that the different locking mechanisms introduced in DB2 9 were not retrofitted to DB2 V8.

4.5.1 Locking for LOBs with DB2 V8

Locking considerations for LOBs are different from normal locking techniques for other DB2 objects. Because a single LOB can grow up to 2 GB, and therefore its data can span over a lot of pages, a new mechanism was introduced to ensure row consistency and data integrity while accessing LOB values to try to minimize the impact on the accessed data. This is done using a combination of base table locks and LOB locks.

For the following examples and explanations, we assume LOCKSIZE ANY. If you specify other lock sizes in your environment, the locked objects might differ.

The LOB lock

A LOB lock is a type of lock which is used to provide concurrent access on LOBs. These new types of locks support access to large objects for reading and updating applications without interfering with each other. You can find a shared LOB lock (S-LOB lock) and an exclusive LOB lock (X-LOB lock) as lock types for large objects in your system. There is no need for an update LOB lock, because updating a LOB value always means deleting and inserting a value, so that there are not any direct updates on LOB data pages. A LOB lock always locks an individual LOB, and it comes with a corresponding lock on the base table to ensure row consistency. A LOB lock only occurs as a column lock on a single value.

There is no locking for untouched LOBs, because a data page cannot contain values of more than one single LOB value. But if you look at lock escalation mechanisms or if you explicitly specify LOCKSIZE TABLESPACE, you can find more locks than the number accessed by your application. Locks on LOB values are only escalated to the table space level, because there is only one table in each LOB table space which can be ignored for locking purposes. See Figure 4-7.

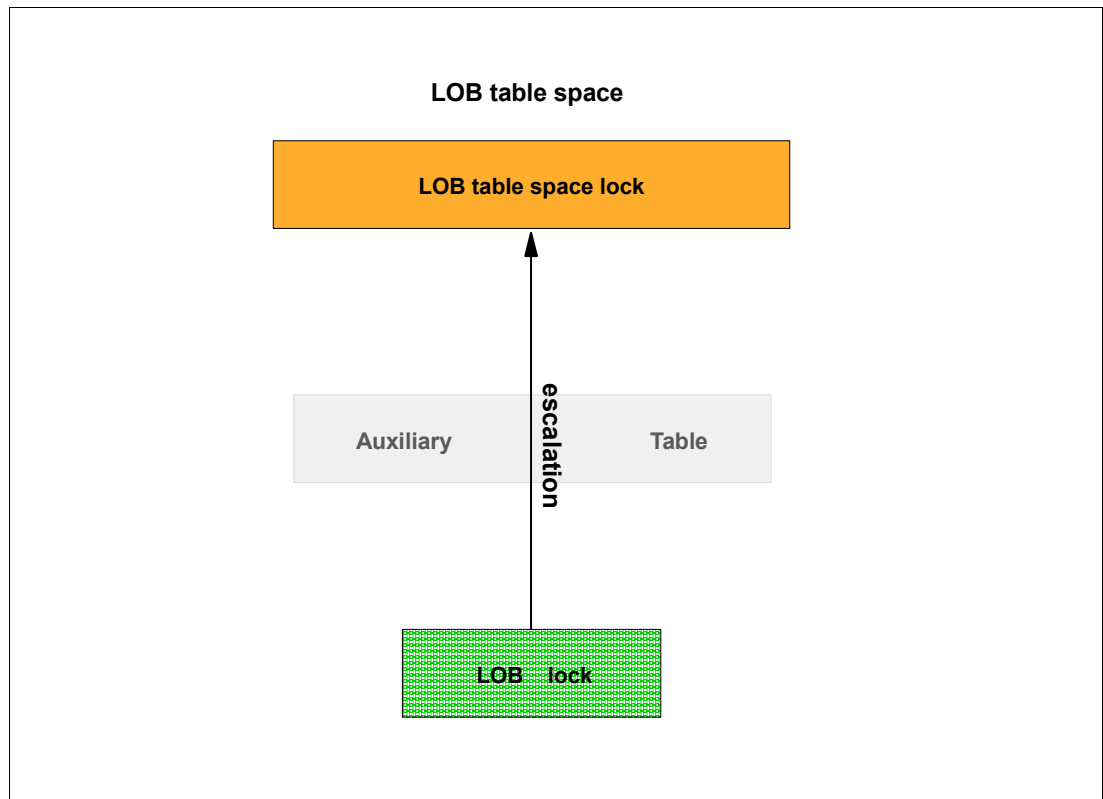


Figure 4-7 Lock escalation on LOBs

Locks with SELECT

Locking, when selecting LOBs, differs from selecting other data types in DB2. There is also a difference in terms of accesses because the data describing the LOB is stored in a base table, while the LOB value itself is stored in an auxiliary table. When you select a LOB value into your host variable, you code your SQL statement as if accessing just the base table. Accessing the auxiliary table directly using SQL is not supported. The first thing DB2 does is access the base table to find out which row in the auxiliary table it has to access.

For this access to the base table, an intent-share (IS) lock is needed on the table space containing the base table, another one on the base table itself, and a share (S) lock on the page accessed while retrieving or validating the rows of the base table. After the particular LOB is located, a IS-lock on the LOB table space and a single S-LOB lock on the LOB value are acquired to ensure row consistency. The lock is held by DB2 until the whole value is delivered to your host variable, so that no updating transactions can interfere with your application and the LOB does not disappear while your application is reading it. Figure 4-8 gives you an idea of how DB2 serializes locks when accessing a single LOB value.

The numbers indicated in the figures in this section reflect the sequence in which the locks are taken.

Using lock avoidance or uncommitted read does not lock the accessed row in the base table while you are retrieving the LOB value. In this case, the part of a 'logical row' (containing the entry in the base table and the associated LOB value) which resides in the base table can be updated by concurrent transactions while you retrieve the associated LOB value.

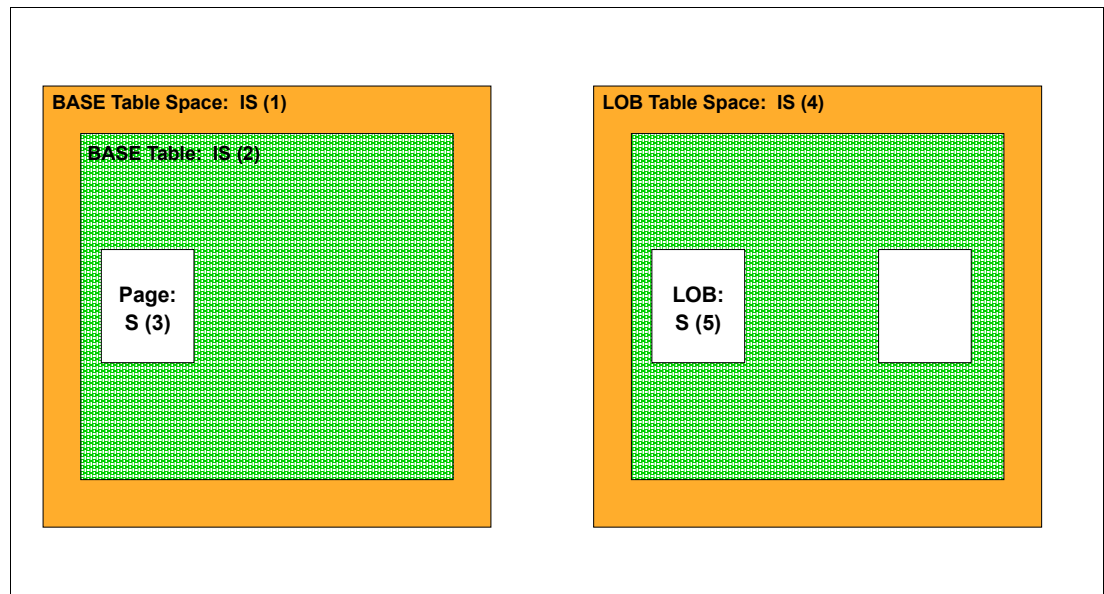


Figure 4-8 SELECT lock sequence

Note: This differs from the general DB2 lock mechanism, where no parts of a row can be updated while it is delivered to your application. Using lock avoidance or uncommitted read delivers the row as it was at the point in time when your SQL statement was issued.

Using ISOLATION (UR) for LOBs

You can use ISOLATION (UR) for your application, but DB2 has to request an S-LOB lock to ensure data integrity while the application retrieves the data. But what is DB2 doing when you access your LOB data even with ISOLATION (UR)? When using uncommitted read, first an S-lock is acquired on the base table to prevent you from being hit by mass delete statements. We refer to this type of lock as an *S mass delete lock*. The next lock to be requested is an intent-share (IS) lock on the LOB table space and an S-LOB lock on the LOB itself. An S-LOB lock has to be requested to protect the LOB value from updates or deletes while the application retrieves the value. As mentioned before, a LOB value can span several pages and it takes time to retrieve it; during this time, no other transaction is allowed to change its value. Using S-LOB locks, even while using uncommitted read, prevents LOB data from getting corrupted by someone else during the delivery process to your application by DB2. Figure 4-9 shows LOB lock serialization when you use uncommitted read.

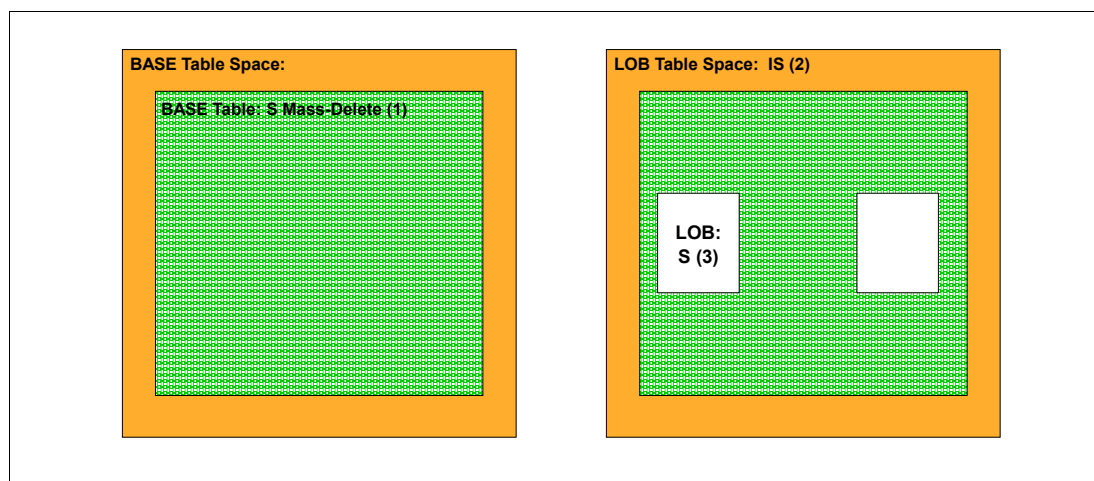


Figure 4-9 *SELECT* lock sequence using uncommitted read

Locks with INSERT

Inserting a LOB from a locking point of view requires more locks than inserting data in a normal table, because there are two tables involved every time during an insert process. However, DB2 only gets a lock on the base table if it is segmented. In general, at first, an intent exclusive (IX) lock is taken on the base table space and the base table. The same type of lock is acquired for the LOB table space after establishing the first two locks. After the three IX locks are established, an exclusive (X) lock is taken on the data pages which are used for inserting the LOB value. The last lock is an X-lock on the page or row in the base table where the new row is stored. The reason why insert applications get an X-LOB lock is to prevent access by readers before the LOB is completely inserted. The actions performed by DB2 when you insert a LOB value from the locking point of view are shown in Figure 4-10 on page 98.

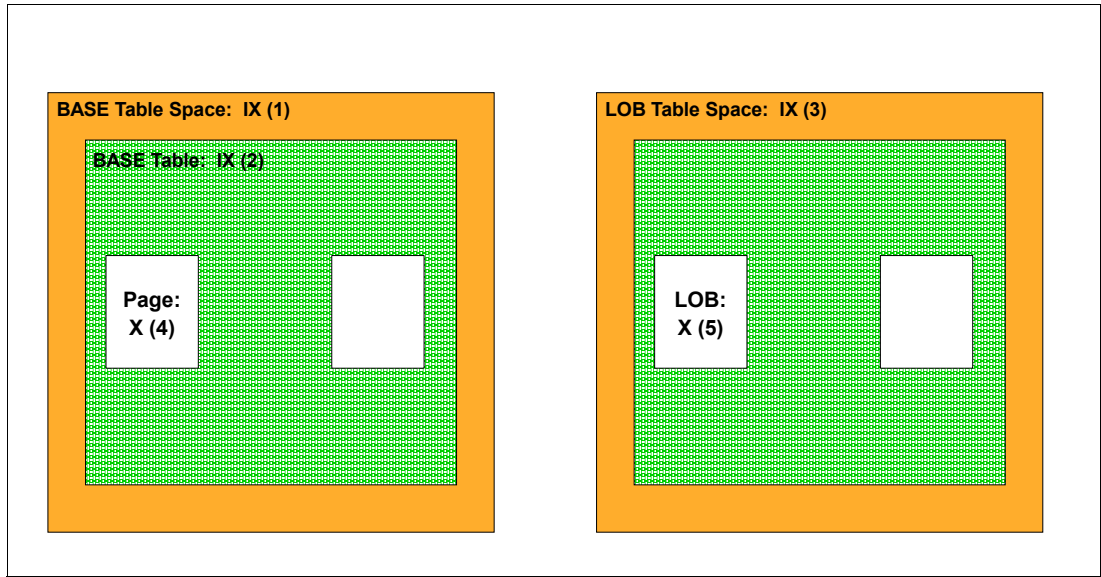


Figure 4-10 Insert lock sequence

The IX LOB locks are held until the application commits or the thread terminates.

Locks with DELETE

In this section, we differentiate between the locking operations occurring with a singleton delete and a mass delete.

Deleting a single LOB value

A LOB delete requires more locks than just those on the LOB table space, as shown in Figure 4-11.

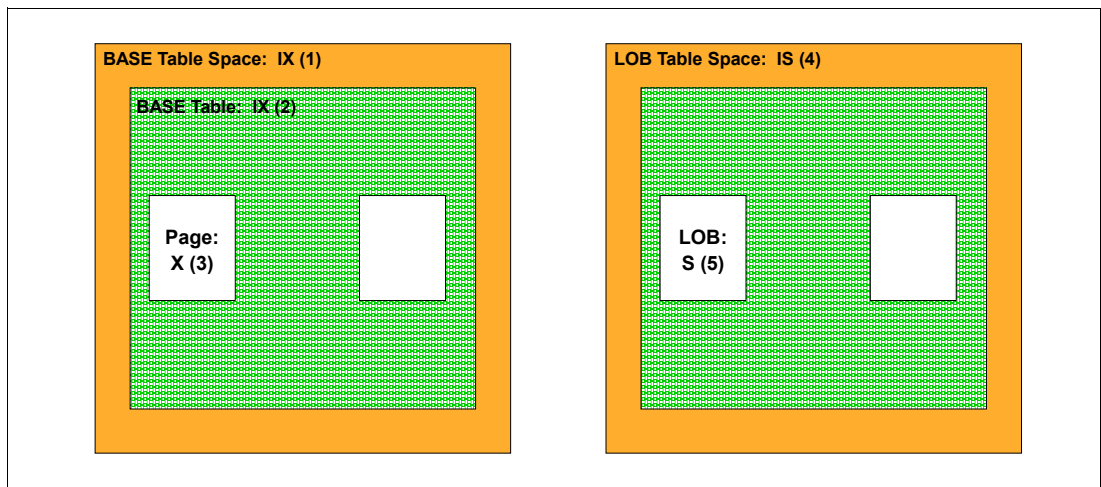


Figure 4-11 Delete lock sequence

After establishing the usual IX locks on the base table space and the base table (if segmented), only a intent-share (IS) lock is requested for the LOB table space. The page containing the associated data in the base table is also provided with an X-lock. There is no IX lock request for the LOB table space, because deleting a LOB value is implemented by deallocating the allocated pages used by the LOB, and not directly updating the data pages. For this reason, the requested lock on the LOB is only a shared LOB (S-LOB) lock. If an

S-LOB lock exists for a LOB value, the deallocation is done at DELETE time, but the space can only be reused if all S-LOB locks held by readers (see “Locks with SELECT” on page 96 for locks requested by LOB readers) are released. So deleting applications acquire S-LOB locks to reserve space in case of a rollback.

Even if the deleting application commits the unit of work or its thread ends, the COMMIT is done and the LOB is only accessible for the thread currently reading the LOB value.

Comparison with a mass delete

A mass delete also places the common IX-lock as shown in the examples above. For the base table, DB2 requires an X-lock and also a mass-delete X-lock on the base table. An X-lock for the LOB table space is acquired, too. See Figure 4-12 for a brief overview of locks taken at mass-delete time.

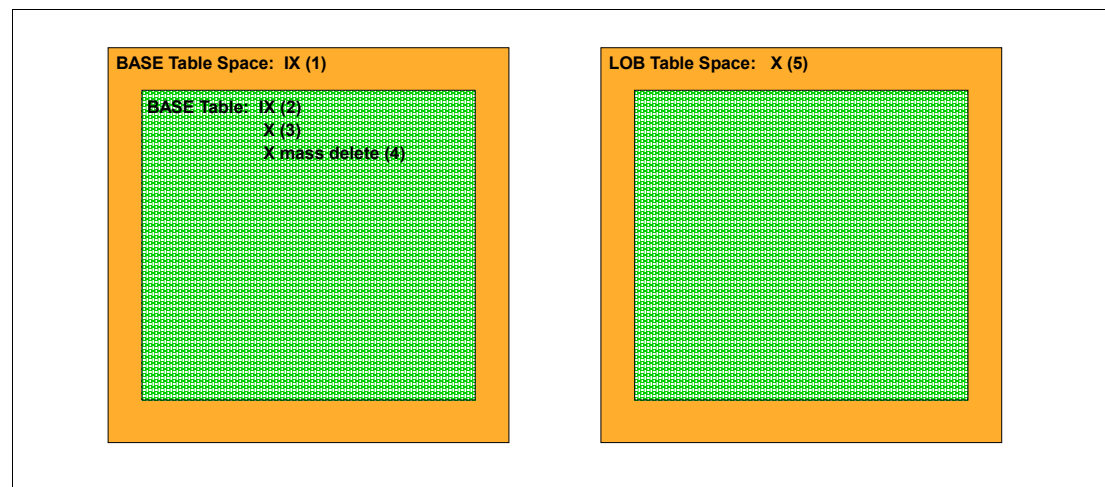


Figure 4-12 Locks acquired by mass delete

Locks with UPDATE

Discussing what happens when a LOB value is updated by an application program is quite easy after having seen how locks are established at delete and insert times. A LOB update always consists of a single LOB delete and a LOB insert. You can see these actions only serialized when updating a LOB column. After DB2 takes both IX-locks on the base table space and the base table, it also requests an X-lock for a particular row in the base table. After all locks are established by the internal resource lock manager (IRLM), DB2 acquires an IS-lock on the LOB table space and an S-LOB lock to delete the old LOB. When the LOB is deleted, the new row is inserted after changing the IS-lock on the LOB table space to an IX-lock, and an X-LOB lock is set for the new LOB value. You can find the sequence illustrated in Figure 4-13 on page 100.

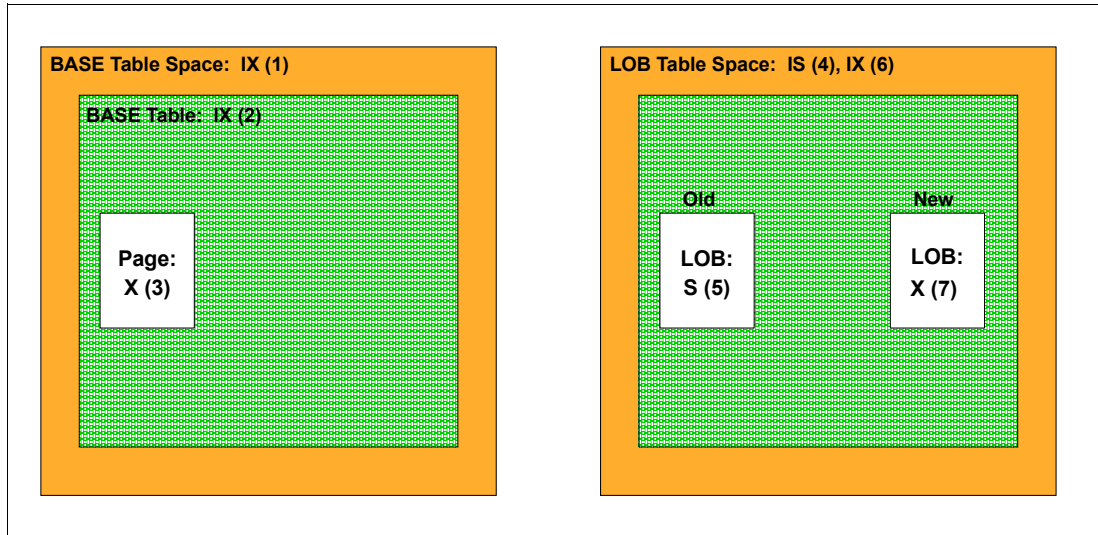


Figure 4-13 Lock sequence when updating a LOB column

Additional information about locking

In this section, we add some considerations about LOB locking.

Conflicts between SELECTers and DELETers

After you have read the previous sections, you can ask about concurrency of selectors and deleters. While readers and deleters both take S-LOB locks, they can coexist without interfering with each other. The pages where the LOB data resides are only reused if there are no more S-LOB locks on the LOB, which simply means that nobody accesses the LOB at this particular point in time.

Shadow Copy Recovery

By shadow copy recovery, we mean the deallocation and reallocation of data pages in a LOB table space as they have been flagged in the space map pages. When you delete a LOB value, the deletion deallocates pages and logs only the deallocation. The LOB data is left in the pages within the LOB table space as a shadow (dormient) copy of the deleted data. A rollback simply results in reallocation of the previously deleted pages. The S-LOB lock taken at delete prevents a later insertion from reusing the space deallocated by the deletion. The S-LOB lock does not prevent deletions of that LOB but ensures that the space allocated to the LOB is not reused until the LOB lock is released. If the LOB is deleted, the shadow copy of that version of the LOB persists until all LOB locks on that LOB version are released.

When no LOB locks are taken

There are some situations where DB2 knows that it does not make sense to ask IRLM for a lock. There are at least four instances in DB2 V8 when no LOB locks are acquired:

- ▶ Selecting a LOB value that is *NULL* or zero length
- ▶ Deleting a LOB value that is *NULL* or zero length
- ▶ Inserting a LOB value that is *NULL* or zero length
- ▶ Updating a LOB value that is *NULL* or zero length to zero length or *NULL*

Changes on base table rows are logged as usual, only indicator column changes are logged within the base table log record.

Summary of locks with DB2 V8

Locks are mainly used to determine whether space can be reused for deleted LOBs, not for concurrency control.

Locks are held until commit, and held across commit for held cursors or held locators:

- ▶ Insert/Update - X LOB lock
 - ▶ Select/Delete - S LOB lock
 - There is no lock avoidance.
 - ISOLATION UR skips uncommitted inserts.
- S locks, if acquired, are held until commit.

Recommendations for V8:

- ▶ Increase LOCKMAX value to avoid lock escalation.
- ▶ Issue frequent commits, free the locators, and release held cursors to reduce the number of held LOB locks.
- ▶ In data sharing, specify GBPCACHE SYSTEM to cache space map pages to improve performance without flooding the global buffer pool with LOB data pages.

4.5.2 Locking for LOBs with DB2 9

One of the most important improvements for LOBs introduced in DB2 9 is the different locking technique compared to previous versions of DB2. Before DB2 9, DB2 acquired a LOB lock in order to:

- ▶ Serialize access to the LOB table space
- ▶ Determine whether deallocated space can be reallocated and reused

Prior to DB2 9, readers have acquired S-LOB locks to prevent their storage from getting reclaimed while reading the LOB value. The space was only reclaimed by DB2 if all of the readers were gone and had released their locks.

Even if it does not look obvious at first sight, isolation UR readers also could cause lock escalations on LOB table spaces with pre-DB2 9 LOB locking mechanisms. Therefore, several performance implications for concurrently running processes might have occurred.

Figure 4-7 on page 95 describes the process of a possible lock escalation with isolation UR access to LOB data.

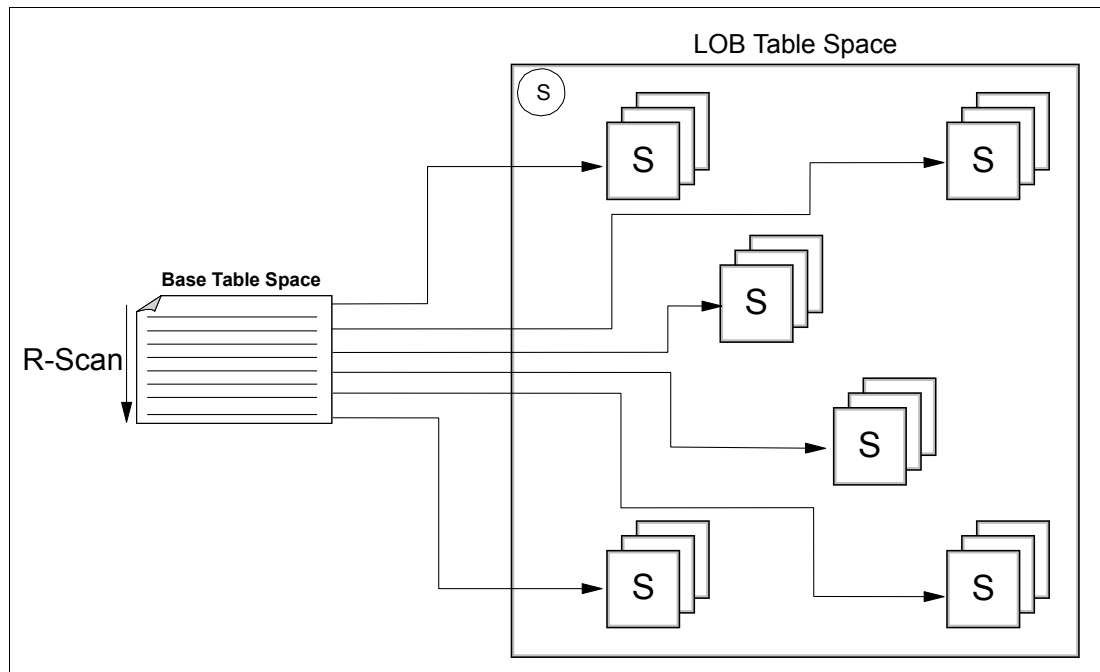


Figure 4-14 Lock escalation for UR readers

A thread accesses the base table using isolation level UR on a relational scan to avoid any locking conflicts. To ensure data integrity of LOB data while reading them, even an isolation UR reader has to acquire an S-LOB lock to ensure data consistency between a base table row and its corresponding LOB value (see “Using ISOLATION (UR) for LOBs” on page 97 for more information). Assuming the UR reader reads more than one LOB without committing frequently, which is typically the case for UR readers, the number of LOB locks can frequently exceed the maximum limitation for the affected table space and DB2 has to escalate the lock scope to the LOB table space level.

Another problem facing UR readers relies on the fact that DB2 performs a base row update and its corresponding LOB update in two separate stages, which implies UR readers might retrieve the new data from the base table row but are unable to fetch its corresponding LOB data, which can even be partial LOB data in a data sharing environment. The reason for retrieving partial LOB data in data sharing is related to physical LOB storage spanning multiple pages. Because the LOB pages and the LOB's index pages can be written to the global buffer pool (GBP) in any order, it is possible for an UR reader on a different member to see partial LOB data.

What is so exciting about locking for LOBs in DB2 9

DB2 9 reduces and almost eliminates the acquisition of LOB locks on every LOB operation. For UPDATE, DELETE, and SELECT operations, the current LOB lock is removed. The lock duration of S-LOB lock for UR readers has been optimized and shortened from commit duration to autorel duration (see “Autorel mode” on page 103). Beginning with DB2 9, you should no longer experience lock escalations on your LOB table spaces.

The enhanced locking mechanism for LOBs minimizes the use of LOB locks and provides better management to resolve the serialization process between isolation UR readers and concurrently running INSERT and UPDATE operations. In general, now there are only X-LOB locks for the period of time when a LOB value is obviously inconsistent, which is the case during INSERT or UPDATE processing of a single LOB value.

DB2 9 now allows you to serialize the access of a LOB table space by only maintaining the locks on a base table row or a base table space data page. This base table lock basically blocks readers with isolation levels CS, RS, and RR except readers using UR who are using a lock avoidance scheme. For a LOB table space, DB2 now rarely takes locks to ensure data integrity.

There are two new lock duration modes introduced in DB2 9:

- **Manual duration mode**

For INSERT and UPDATE operations, DB2 must continue to hold an X-LOB lock on the LOB data. The locks on the LOB table space pages are released immediately after completing the INSERT or UPDATE statement of the corresponding LOB table space pages and its auxiliary index. This is called *manual duration mode*. Access to LOB data by non-UR readers is furthermore prohibited due to an exclusive lock on the base table space row or data page. In a data sharing environment, prior to unlocking, DB2 flushes out the changed data of LOB data pages and its index pages to the global buffer pool. These operations ensure that any UR reader in other DB2 subsystems can access a complete copy of your LOB data.

Changing to manual duration mode can significantly shorten the time that a LOB lock is held and improve the overall lock contention.

- **Autorel mode**

For SELECT operations, DB2 relies on the lock taken on the base table to ensure the access of a correct copy of LOB data. As soon as your application uses isolation UR, if LOB columns are to be selected, the UR reader must acquire a S-LOB lock to serialize with concurrent INSERT and UPDATE operations. The lock is immediately released after it was acquired because the purpose was just to check that no other lock had been established. This is called *autorel mode*.

We refer to a third lock mode called *commit duration*, which releases the locks at COMMIT time. This lock mode is used by DB2 V8 for INSERTs.

The basics for concurrency improvements

For concurrency management, DB2 keeps track of a global value called Read Log Sequence Number (Read LSN or RLSN), containing the information about the oldest read claim in each unit of recovery in the system:

- In data sharing, the RLSN is maintained at buffer pool level.
- In non-data sharing, the RLSN is maintained at page set level.

The read claim is released at COMMIT time or when a thread is deallocated.

DB2 9 stores a LRSN in data sharing mode and a relative byte address (RBA) in non-data sharing mode for each deallocated page under one lower space map page, containing the information when a data page was last updated. Note that this group LRSN is not effective to represent the status of each data page covered by this space map.

We assume page locking in the following discussion.

Space reuse in LOB table spaces

Reusing space in a LOB table space is an issue when you insert or update your LOB data, because deleting a LOB or removing parts from a LOB value results in the deallocation of the old LOB data pages and the space might then be reused for a new LOB value. Before LOB data pages can be reused, DB2 has to guarantee that the LOB data residing in the data pages, which are going to be reused, is no longer accessed by any other thread in the

system. This means that before space reuse can be invoked, all readers of LOB values of the affected LOB data pages have to have completed their access.

For space reuse in a LOB table space, DB2 9 removes the use of LOB locks previously required to determine which deallocated LOB storage can be reused for new allocations.

DB2 now depends on a comparison of the read LRSN and delete LRSN to determine the availability of already deallocated storage.

When a unit of recovery (UR) starts accessing a LOB table space, it first needs to acquire a read claim, this is the read LRSN. DB2 keeps tracks of this read LRSN for each UR and knows the read LRSN of the oldest one. By knowing the delete LRSN of the page considered for reuse, DB2 can use the comparison between the delete LRSN and the oldest read LRSN to determine if the UR is committed or not. When DB2 deletes a LOB, it marks all the pages for that LOB to be deallocated on the status bit and stores the deleted LRSN for each page on the space map page.

Before V9, the delete LRSN is one value on the low level space map page and it indicates the latest deletion of any page belonging to this low level space map page. One or more low-level space map pages refer to all data pages of one single LOB column. This LRSN comparison does not give enough information to reuse a page which was already committed. A LOB LOCK must be used.

In V9, DB2 stores the delete LRSN on each page. This allows DB2 to use the LRSN comparison method to determine if the page can be reused.

See Example 4-13 for a brief description.

Example 4-13 Comparison of RLSN and LSN to reclaim space

RLSN of oldest read claim : X'FEDCBA098765'
LSN in LOB page : X'1234567890AB'

RLSN > LSN of page considered for reuse

Yes: Page can be reused, reader started after the deallocated page was committed

RLSN of oldest read claim : X'1234567890AB'
LSN in LOB page : X'FEDCBA098765'

RLSN > LSN of page considered for reuse

No: Page cannot be reused because of possible active reference

In the first case, the RLSN of the oldest reader in the system is greater than the LSN contained in the LOB page. This means that the pages considered for reuse were deallocated and committed before the reader has started accessing LOB data; therefore, it is safe to reuse the data pages for a new LOB value.

In the second case, the RLSN of the oldest reader in the system is less than the LSN contained in the LOB page. This implies that the page considered for reuse was deallocated after the oldest reader has started accessing that particular LOB data. As a consequence, the affected LOB data page cannot be reused since reusing the data page can result in corrupting data for the application currently accessing its value.

Locks with INSERT

In case you INSERT a row containing a LOB value, you get an X-lock on the base table data page. While your data is moved to the auxiliary table where your LOB column resides, you also get an X-LOB lock in manual duration mode. Free LOB data pages are determined as explained in “Space reuse in LOB table spaces” on page 103. If no reusable data pages are available, DB2 allocates a set of new LOB data pages. After the INSERT statement has finally moved your data to your LOB columns, all X-LOB locks are released because your LOB data is now consistent.

Since LOB values can only be accessed using SQL statements on the base table which is still holding an X-lock on the affected data page or row, your LOB data can only be accessed by UR readers at that particular point in time. The X-lock on the base table data page is released at COMMIT time.

Important: In case your applications insert partial LOB data to append the remaining bytes afterwards, DB2 cannot ensure data integrity since the LOB can be accessed by UR readers after the originating INSERT statement succeeds.

Locks with UPDATE

From the operational point of view, updates on LOB values still consist of deallocating the old data pages and allocating new data pages for the new LOB value. The same is true for reclaiming space during LOB updates as it is for inserting LOB values.

First, every UPDATE statement gets an X-lock on the base table data page. Passing the new LOB value to DB2 behaves in almost the same way as when you insert a new LOB value; X-LOB locks on the auxiliary table are acquired in manual duration mode until the insert of the new LOB value is complete. All X-LOB locks are released as soon as the UPDATE statement completes. The current LRSN/RBA is stored in the low-level space map pages pointing to that particular version of the LOB that is being deallocated.

From a DB2 point of view, data integrity can now be guaranteed, because only UR readers can read the base table and access consistent LOB data. The X-lock on the base table data page is released at COMMIT time.

Important: In case your applications update LOB data partially and do not replace the entire LOB value using an UPDATE statement, DB2 allows UR readers to access inconsistent LOB values from an application point of view.

Locks with DELETE

Each DELETE statement holds an X-lock on the base table space data page it references. The data pages containing the associated LOB value in the LOB table space are deallocated as they were in former versions of DB2 using the information in the space map pages during DELETE processing. The LOB value itself can still be accessed by readers that were already referencing the LOB value before the pages were deallocated. No LOB lock is taken. Statements currently accessing deallocated LOB data pages can complete their LOB reference since no additional check is performed once the LOB is accessed through the space map pages. But again, the current LRSN/RBA is stored in the low-level space map pages pointing to that particular LOB value. Further reference to the LOB data pages is prohibited because of the row being removed from the base table.

Locks with SELECT

Depending on the isolation level you use for your application, DB2 can use lock avoidance on the base table. If lock avoidance is not an option, DB2 acquires an S-lock on the base table

data page that you reference. Using this common technique, data consistency can be guaranteed for the data residing in the base table. If the data is consistent in the base table, the LOB data is also consistent and can be accessed. Once your application accesses the LOB data for a specific row through the base table, data can be retrieved even if the LOB data pages are deallocated by another transaction during retrieval time. Only the next reference to the LOB itself through the base table fails if a concurrently running DELETE statement deletes the corresponding row from the base table.

Locks with SELECT UR

Using uncommitted read on the base table, data integrity among the base table row is guaranteed. For access to LOB values corresponding to the base table rows, DB2 requests an unconditional S-LOB lock on each LOB data that is to be selected. Requesting an unconditional lock in our case means that an S-LOB lock is requested and immediately released afterwards before retrieving the LOB value. The reason for this behavior is to serialize with concurrent INSERT or UPDATE operations holding an X-LOB lock on the LOB data pages, making sure that no inconsistent LOB data from a DB2 point of view can be accessed.

SKIP LOCKED DATA clause

DB2 9 also introduces the SKIP LOCKED DATA clause, allowing you to bypass locked rows on your SELECT statements as on searched UPDATE and DELETE operations.

You can use the SKIP LOCKED DATA clause to minimize transaction suspensions as well as to avoid the possibility of deadlocks. Using this clause, all locked rows, which cannot be accessed to complete your SQL statement, are skipped.

The SKIP LOCKED DATA clause applies to the base table, the considerations on LOB locks seen in the previous sections apply only to the LOB table rows selected from the base table, not the skipped rows.

Summary of locks with DB2 9

A short summary of how locks are established is as follows:

- ▶ INSERT operations require an X-lock on each LOB data page to be inserted with a manual duration instead of a commit duration (lock and unlock sequence).
- ▶ UPDATE operations require an X-lock on each LOB data to be updated but with a manual duration instead of commit duration.
- ▶ DELETE operations require no LOB lock but an X-lock on the base table.
- ▶ SELECT operations require no LOB lock while accessing the LOB data.
- ▶ UR readers request a unconditional S-LOB lock on each LOB data that is to be selected. This is to serialize with concurrent INSERT or UPDATE operations. The lock is released immediately.
- ▶ Under INSERT or UPDATE operations, DB2 now depends on the Read LSN to decide the availability of each deallocated page, therefore, no LOB lock is required.
- ▶ In data sharing, DB2 needs to flush LOB data and index pages to GBP before releasing the LOB lock.

Recommendation for DB2 9:

- ▶ In data sharing, consider specifying GBPCACHE CHANGE to avoid flushing changed pages to disk.

4.6 Unloading LOBs

Dealing with all kinds of large objects is challenging from the application's point of view. You face new challenges in dealing with objects bigger than most common data objects. In this section, we discuss unloading LOBs to store them in a data set, and also application programming techniques to manipulate LOBs without retrieving them. We use the term unloading rather than just reading, because the normal usage of LOBs would be to place the content of the read data into another data container of some sort.

DB2 9 introduces file reference variables supporting you to unload LOB data to a sequential file, PDS, PDSE data set, or a HFS file. Refer to 2.4, "LOB file reference variables" on page 18 for a detailed description of the usage of File Reference Variables. Unloading and loading LOB values using a file reference variable is the recommended way in DB2 9 compared to using LOB locators.

Before DB2 9, you could use application logic or the UNLOAD utility to externalize your LOB data to an external file. The UNLOAD utility now also supports rows containing LOB values and, therefore, exceeding a total size of 32 KB. You can find more information about the UNLOAD utility in 6.1, "UNLOAD" on page 160.

You can find examples for inserting LOB data through an application in 4.4.2, "Inserting LOBs using the host application" on page 85.

4.6.1 Unloading a LOB using an application

Unloading a LOB value into a certain data set is easy when compared to the manipulation techniques described in 4.7, "Updating LOBs" on page 120. In general, the method you use to unload LOB values depends on the version of DB2 you use and on the maximum available size of a host variable. If you are able to allocate a host variable that is large enough to contain the maximum size of a LOB value that you want to unload, unloading is easier than using locators.

Case 1: Unloading a LOB using a file reference variable

If your system runs DB2 9, the recommended way to unload a LOB column is using a file reference variable, pointing to the destination data set.

For using file reference variables, DB2 9 introduces three new types of SQL host variables:

- ▶ BLOB-FILE
- ▶ CLOB-FILE
- ▶ DBCLOB-FILE

These SQL host variables can be used either to insert a LOB from a file or to select a LOB value from DB2 to a file. Using this technique, an entire LOB value can be selected or inserted without being required to acquire a contiguous piece of storage to hold the entire LOB value. The LOB value is not passed through your application's memory, therefore bypassing the limitation of your host language on the maximum size allowed for host variables. Figure 4-16 on page 112 provides the pseudo code for using a file reference variable to unload a LOB value.

Example 4-14 Unloading LOB data using a file reference variable

Definitions:

```
01 LOBDATA          USAGE IS SQL TYPE IS BLOB-FILE
```

Pseudo-Code:

```

MOVE 'PAOLO.SG247270.PDF' TO LOBDATA-NAME
MOVE 18 TO LOBDATA-NAME-LENGTH
MOVE SQL-FILE-CREATE TO LOBDATA-FILE-OPTION

EXEC SQL
  SELECT LOB
  INTO   :LOBDATA
  FROM   BASE_TABLE
  WHERE  KEYCOL1 = :HV-KEYCOL1
END-EXEC

```

By providing a variable with SQL TYPE IS BLOB-FILE, the precompiler generates the structure as reported in Example 4-15.

Example 4-15 What the DB2 precompiler generates for LOB files

```

01 LOBDATA.
  49 LOBDATA-NAME-LENGTH PIC S9(9) COMP-5 SYNC.
  49 LOBDATA-DATA-LENGTH PIC S9(9) COMP-5.
  49 LOBDATA-FILE-OPTION PIC S9(9) COMP-5.
  49 LOBDATA-NAME PIC X(255).

[...]

77 SQL-FILE-READ      PIC S9(9) COMP-4 VALUE +2.
77 SQL-FILE-CREATE    PIC S9(9) COMP-4 VALUE +8.
77 SQL-FILE-OVERWRITE PIC S9(9) COMP-4 VALUE +16.
77 SQL-FILE-APPEND    PIC S9(9) COMP-4 VALUE +32.

```

The LOBDATA-NAME contains the name of the file where the LOB is going to be unloaded and it can be provided at run time. LOBDATA-NAME-LENGTH contains the length of the LOBDATA-NAME itself. DB2 always creates a variable length file with LRECL 27,994. The LOBDATA-DATA-LENGTH is set to the length of the new data written to the file and is ignored as input. The values of fields SQL-FILE-READ, SQL-FILE-CREATE, SQL-FILE-OVERWRITE, and SQL-FILE-APPEND can be used as input for LOBDATA-FILE-OPTION to tell DB2 how to deal with the specified output file:

- ▶ SQL-FILE-READ for a regular file that can be opened, read, and closed.
- ▶ SQL-FILE-CREATE creates a new file. If the file already exists, an error is returned.
- ▶ SQL-FILE-OVERWRITE for existing files with the specified name. If the file exists, it is overwritten; otherwise, DB2 creates the file.
- ▶ SQL-FILE-APPEND for existing files with the specified name. If the file exists, new data is appended; otherwise, DB2 creates the file.

You can still use these variables to move them to the FILE-OPTION field since DB2 generates them at precompile time. For more information about file reference variables, refer to 2.4, “LOB file reference variables” on page 18.

Note: DB2 always uses variable length data sets with LRECL 27,994 for CLOBs, BLOBs, and DBCLOBs when you unload your LOB values using a file reference variable.

A sample program showing how to unload a LOB using a file reference variable for BLOBs and CLOBs is included in the files described in Appendix A, “Additional material” on page 259.

Case 2: Unloading a LOB using one host variable

Assume you want to unload a 10 MB CLOB value of a text document. The first thing you have to verify is the format in which the data is needed, because you cannot write the data to a single row in a data set. So the application has to split up the data into the requested file format. In our example, we assume an output file of LRECL 1,024 using record format FB. So the application selects the entire LOB value into a host variable and writes the value of the host variable in 1,024 byte pieces to an output data set. Example 4-16 provides a pseudo code to perform a LOB unload using an application using one host variable.

Example 4-16 Unloading LOB data using one host variable

Definitions:

```
HV-LOB          USAGE IS SOL TYPE IS CLOB (10M)
```

Pseudo-Code:

```
EXEC SQL
  SELECT LOB
  INTO   :HV-LOB
  FROM   BASE_TABLE
  WHERE  KEYCOL1 = :HV-KEYCOL1
END-EXEC
```

PERFORM WRITE-DATA

```
:WRITE-DATA
```

MOVE 1

TO BYTE-COUNTER

```
PERFORM UNTIL BYTE-COUNTER > HV-LOB-LENGTH
```

```
MOVE HV-LOB-DATA (BYTE-COUNTER:OUTPUT-FILE-LENGTH)
```

TO OUTPUT-RECORD

WRITE OUTPUT-RECORD

ADD OUTPUT-FILE-LENGTH

TO BYTE-COUNTER

END-PERFORM

This solution works fine as long as the maximum LOB value is not bigger than the largest size of a host variable that you are allowed to acquire. If you are not able to acquire a host variable as big as the size of your maximum LOB value, you can use locators for unloading your data as we discuss in Case 3.

A sample program showing how to unload a LOB using a host variable for BLOBs and CLOBs is included in the files described in Appendix A, “Additional material” on page 259.

Case 3: Unloading an entire LOB using locators

When you are unable to define a host variable that is the size of the LOB data that you want to unload, you can use a different technique. A method to unload huge amounts of data is using locators to reference a whole LOB value. The application only retrieves a part of a LOB, which can be easily written to an output file. After processing the retrieved part, the next part is retrieved and also written to an output file. We assume the same file attributes as in Example 4-16. Example 4-17 on page 110 gives you an idea how to unload a LOB value using locators when you are only allowed to use a 1 MB host variable and your LOBs are larger than 1 MB.

Example 4-17 Unloading a LOB using locators

Definitions:

```
HV-LOB          USAGE IS SQL TYPE IS CLOB (1M)
LOB-LOCATOR-1   USAGE IS SQL TYPE IS CLOB-LOCATOR
```

Pseudo-Code:

```
EXEC SQL
  SELECT LENGTH(LOB), LOB
  INTO   :CURRENT-LENGTH, :LOB-LOCATOR-1
  FROM   BASE_TABLE
  WHERE  KEYCOL1 = :HV-KEYCOL1
END-EXEC

COMPUTE AMOUNT-FULL-SUBSTR  = CURRENT-LENGTH / MAX-VAR-SIZE
COMPUTE AMOUNT-REMAIN-SUBSTR = CURRENT-LENGTH - (AMOUNT-FULL-SUBSTR * MAX-VAR-SIZE)

PERFORM VARYING SUBSTR-COUNTER FROM 1 BY 1 UNTIL SUBSTR-COUNTER > AMOUNT-FULL-SUBSTR
EXEC SQL
  SET :HV-LOB = SUBSTR(:LOB-LOCATOR-1, 1, MAX-VAR-SIZE)
END-EXEC

PERFORM WRITE-DATA
END-PERFORM

IF AMOUNT-REMAIN-SUBSTR > 0 THEN
  EXEC SQL
    SET :HV-LOB = SUBSTR(:LOB-LOCATOR-1, 1, :AMOUNT-REMAIN-SUBSTR)
  END-EXEC

  PERFORM WRITE-DATA
END-IF

EXEC SQL
  FREE LOCATOR :LOB-LOCATOR-1
END-EXEC
```

Before we start retrieving parts of the LOB, the application sets a locator to a LOB value in order not to allow changes to the LOB value during the unit of work. Otherwise, in DB2 V8, the content of the LOB can change while the application processes the LOB data. In this case, we acquire a locator to refer to a *frozen* LOB value for our unit of work. The idea behind retrieving values using a locator is the same idea we use when we insert LOB value, try using large host variables to reduce the number of your SQL calls.

A sample program showing how to unload a LOB using a host variable of 1 MB and locators for BLOBs and CLOBs is included in the files described in Appendix A, “Additional material” on page 259.

The variable AMOUNT-FULL-SUBSTR tells your program how often it has to perform the SUBSTR to retrieve a part of the LOB value that is the size of your largest host variable. For the remaining bytes of the LOB, we calculate AMOUNT-REMAIN-SUBSTR to retrieve those bytes, which are not covered by the previously issued statements. After retrieving the results of each SUBSTR function, the data is written to a file using the WRITE-DATA subroutine as mentioned in Case 1.

This example for retrieving your LOB data avoids materialization of the LOB, because DB2 knows where the parts you want to retrieve using the SUBSTR function are stored. Therefore,

DB2 uses the LOB pageset structure to locate the data pages you want to retrieve. For a detailed description of the LOB pageset structure, see 3.6, “Physical layout of LOBs” on page 62.

Figure 4-15 shows you how a delete from another transaction can affect your unit of work when you do not use a locator technique to *freeze* a LOB value from your application’s view, or when you do not use isolation levels that can protect you from this situation. In our specific example, we assume cursor stability (CS) as the isolation level with CURRENTDATA(NO) for lock avoidance and RELEASE(COMMIT).

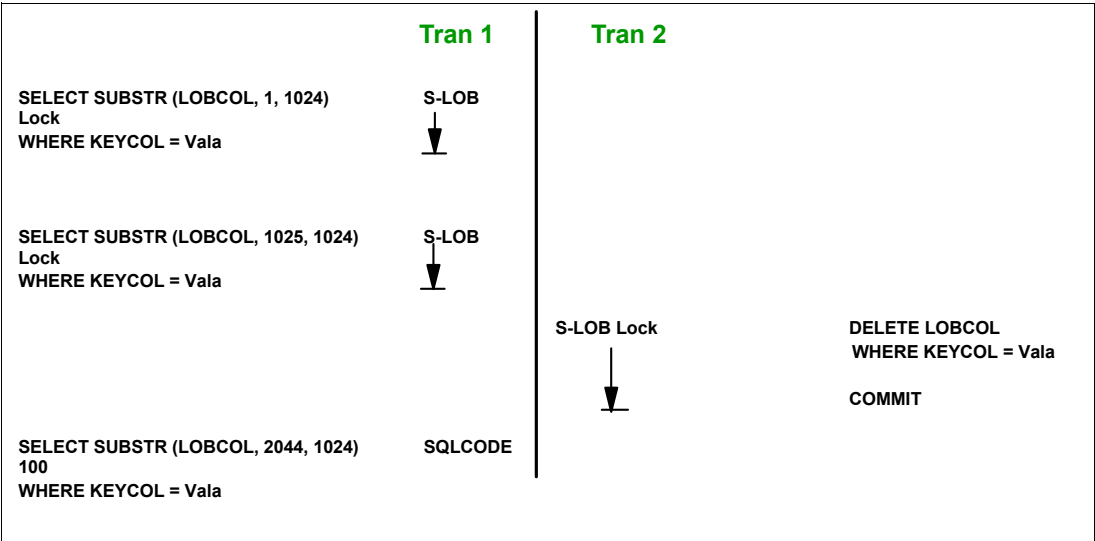


Figure 4-15 Accessing a LOB without a locator reference using ISOLATION (CS)

Note: Note that in DB2 9, no S-LOB locks are acquired for SELECT statements. Refer to 4.5.2, “Locking for LOBs with DB2 9” on page 101 for more details.

Every SELECT SUBSTR acquires an S-LOB lock for the time that it takes to retrieve the requested value. As soon as DB2 has delivered the value (depending on your current settings for DB2 locking), it releases the S-LOB lock. The LOB lock is taken again by DB2 as soon as the next SELECT SUBSTR statement is issued. When you use this method, a second transaction is able to delete the LOB which is currently processed by another transaction.

To avoid this kind of situation, use a locator to *freeze* the object that you currently access. When your application assigns a locator to a particular LOB value, in DB2 V8 the lock is held by DB2 until you explicitly free the locator or issue DB2 COMMIT as shown in Figure 4-16 on page 112.

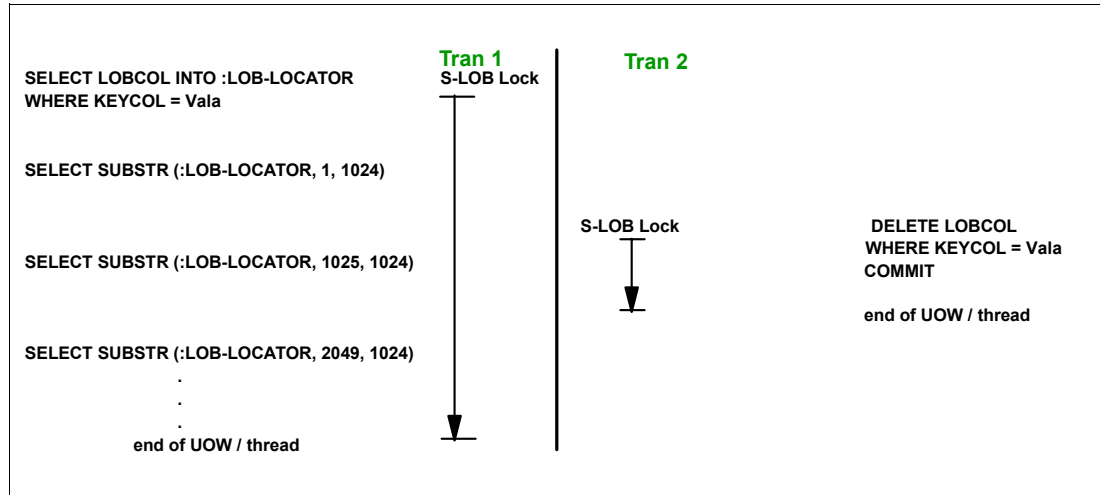


Figure 4-16 Processing a LOB using a locator reference

Transaction two acquires an S-LOB lock which is compatible with the S-LOB lock already held by transaction one. After transaction two issues a COMMIT and its thread terminates, the LOB is only visible for transaction one. The pages are finally deallocated when transaction one releases its S-LOB lock for the accessed value.

Note: In DB2 9, there is no S-LOB lock taken for LOB locators, but once referenced by a LOB locator, the LOB data pages can still be referenced even if the pages are deallocated by another DELETE statement. DB2 does not reuse the LOB's data pages before the locator is finally freed.

Case 4: Unload parts of a LOB using locators

In some cases, your application might want to retrieve only a known portion of a LOB value. In this case, you do not have to retrieve the entire LOB value, you can retrieve just parts of it. You can retrieve parts with and without locators. If you want to have a LOB's part in your host variable for further processing, you can use the SELECT in Case 1 including the LOB function to retrieve only a part of the value. But if even the part you want to retrieve is too big for one single host variable, you can use the pseudo code mentioned in Case 3 to retrieve only the data you need.

Cursors and LOB values

When you plan to vary between the host variables into which you fetch your LOB values, using a cursor (such as FETCH into host variable or FETCH into LOB locator variable), make sure that the CURRENT RULES special register has the correct value. CURRENT RULES STD lets you switch between the host variables into which you fetch the cursor. For more information about CURRENT RULES, see 3.4.7, "Impact on cursors fetching LOB values" on page 60.

List prefetch for LOBs

With respect to the size of LOBs, the only prefetch for LOBs that DB2 performs is for a single LOB value. If a LOB occupies more than one page, LOB Manager prefetches up to a chunk of pages at a time for the LOB value. You have to look at prefetching for LOBs as a different dimension from prefetching normal rows to reduce I/O overhead as much as possible.

4.6.2 Using FETCH CONTINUE

FETCH CONTINUE is an extension to the standard FETCH SQL statement introduced by DB2 9, which provides a convenient method for applications to read from tables that contain LOBs or XML columns, when the actual length of the LOB or XML value is not known or is so large that the application cannot materialize the entire LOB in memory. The declared maximum length for a LOB column is frequently much larger than the typical LOBs that are inserted into those columns.

Prior to DB2 9, applications that used embedded SQL to read from tables containing LOB columns typically had to declare or allocate a piece of storage that was equal in size to the maximum defined storage size for the LOB column. This frequently causes a shortage of virtual memory in certain configurations.

Suppose that the application employs the technique of using a fixed size buffer to fetch an XML or LOB column, but the predefined area is not large enough. DB2 writes as much data as possible to the output buffer and truncates the rest. The truncated portion of the data then cannot be retrieved without repositioning the cursor and prefetching the entire row, but with a larger buffer area for the LOB or XML item. Assuming that the application has the ability to dynamically allocate storage for a new buffer, it would need to CLOSE, reOPEN, and reposition the cursor, then FETCH with the larger buffer. Besides performance concerns with this approach, you would need to issue SELECT LENGTH(LOB_COLUMN) to have returned the length of the LOB value. When truncation occurs on output for other character-based and graphic-based data types, DB2 provides the actual length of the data item through the indicator host variable, if one was provided. However, the indicator variable is assumed to be defined as a two-byte signed integer, limiting the capacity to 32,767. As a result, DB2 does not provide the actual length for LOBs. What is needed is the capability for DB2 to pass that actual length back to the application when truncation occurs for LOB and XML columns. So, if the application has no way of knowing how large to allocate the new buffer, it still has success in the second FETCH.

LOB locators are one way to avoid having to preallocate space for the entire LOB, but they have some problems as well, including slower performance in some cases, excessive resource consumption at the server, and more complex coding.

SQL Driver programs, such as the JDBC driver (when using type-2 connectivity), the native ODBC driver, or even DSNTEP2 and SPUFI that connect locally to DB2, are especially susceptible to virtual storage constraint problems when handling LOBs of unknown length, particularly in environments that support multiple concurrent user connections.

This problem is compounded in DB2 9 with the introduction of XML objects. There is no defined maximum for XML columns. There is an architectural limit of 2 GB, but that limit is impractical for use in declaring or allocating a buffer.

Solution

The FETCH CONTINUE standard FETCH SQL statement extension allows an application to do a FETCH against a table that contains LOB or XML columns, using a buffer that might not be large enough to hold the entire LOB or XML value. If any of the fetched LOB or XML columns do not fit, DB2 returns information about which columns were truncated and what the actual length is. To enable this behavior on the FETCH, the application must add the WITH CONTINUE clause. The application is then able to use that actual length information to allocate a larger target buffer, and to then execute a FETCH statement with the CONTINUE clause to retrieve the remaining data for those columns. Alternatively, the application can be coded to handle large LOBs or XML values by "streaming" the data. That is, the application can use a fixed-size buffer, and following the initial FETCH, perform repeated FETCH CONTINUE operations to retrieve the large LOB or XML value, one piece at a time. Note that

the application is not required to consume all LOB or XML objects in their entirety. It can FETCH the next row at any time, even if truncated columns remain in the current row.

This technique is primarily useful for applications that retrieve an entire LOB or XML column and process it before fetching another row for the cursor or closing the cursor. (Note, however, that it is not required to fetch the entire column before moving to the next row.)

Note: For applications that need to perform "random access" to parts of a LOB, using functions, such as LENGTH, SUBSTR, and POSSTR the use of LOB locators, is still recommended.

How does it work - basic design

FETCH CONTINUE provides two syntax extensions on FETCH:

► FETCH WITH CONTINUE

It is just like a regular FETCH, but it tells DB2 how to react when truncation occurs on output of LOB or XML column. If the truncation occurs, DB2 preserves the rest of the data and remembers the position of the truncation point.

DB2 returns the truncated part of the LOB and the total length of the whole LOB in the LENGTH part of LOB host variable construct. The SQLCA SQLWARN1 field contains 'W' as a warning for truncation.

► FETCH CURRENT CONTINUE

Tells DB2 to continue fetching from the truncation point. The CURRENT keyword tells DB2 to stay on the same row. This statement can be executed as many times as needed until all of the LOBs are processed. Each subsequent FETCH CURRENT CONTINUE returns the size of the remaining portion of LOB in the LENGTH part of the LOB host variable construct. When the last portion is fetched, the SQLWARN1 field is cleared and the SQLCODE and SQLSTATE are 0.

The next time you try to FETCH CURRENT CONTINUE, you receive SQLCODE -20411, stating "A FETCH CURRENT CONTINUE OPERATION WAS REQUESTED FOR <cursor> BUT THERE IS NO PRESERVED, TRUNCATED, DATA TO RETURN."

Possible error messages

The following error codes appear when something has gone wrong:

- -20411 - All the situations where a FETCH CURRENT CONTINUE is attempted, but there were no truncated columns from the previous FETCH.
 - No truncation occurred.
 - FETCH CURRENT CONTINUE one too many times.
 - Cursor is open but not positioned on the row.
- -225 - FETCH CONTINUE was attempted for cursor opened for multi-row FETCH.

Restrictions

The following restrictions might appear:

- FETCH CONTINUE is not supported with multi-row fetch (SQLCODE -225, SQLSTATE 42872, or this might be caught by the precompiler with DSNH104I).
- No intervening operations on this cursor are allowed between FETCH and FETCH CURRENT CONTINUE.
- FETCH CONTINUE only preserves truncated data for result set columns of type BLOB, CLOB, DBCLOB, or XML, and only when the output host variable data type is the appropriate data type.

Note: The following is allowed with FETCH CONTINUE:

- Scrollable Cursors

The FETCH CURRENT CONTINUE functionality can be used with scrollable cursors as well. The FETCH operation can specify WITH CONTINUE even for backward, relative, and absolute fetches. Following such FETCH operations, the FETCH CURRENT CONTINUE statement can retrieve any truncated data.

- Allocated Cursors

A FETCH CURRENT CONTINUE statement can reference an allocated cursor associated with a stored procedure result set. Likewise, the FETCH against an allocated cursor can use the WITH CONTINUE clause.

- Cursors declared WITH HOLD

For a cursor that is held across a commit operation, a FETCH CURRENT CONTINUE following the commit can retrieve LOB data that was truncated on a FETCH that occurred before the commit operation.

Usage high-level examples

There are two recommended ways of using FETCH CONTINUE:

- Dynamically allocate the appropriate storage size
- Stream the data through a single fixed-size buffer

Dynamically allocate the appropriate storage size

Use the initial FETCH to fetch into a preallocated buffer of a moderate size. If the returned data item is too large to fit in that buffer, use the length information returned by DB2 to allocate just the right amount of storage and use one FETCH CONTINUE statement to retrieve the remainder of the data. This method requires that the programming language allow for dynamic storage allocation. This method also requires that the application build and manage its own DESCRIPTOR area (SQLDA) for fetching from the cursor and use the form of the FETCH and FETCH CURRENT CONTINUE statements with the INTO DESCRIPTOR :SQLDA clause. This is standard programming for many dynamic SQL programs.

The details of the FETCH CONTINUE processing are best introduced by a simple example. Example 4-18 uses dynamic SQL and manipulation of the SQLDA descriptor area by the application (as opposed to precompiler-generated SQLDA manipulation). In this example, the application uses, at most, two fetch operations to retrieve the LOB values. On the first fetch operation, it fetches these columns into a moderate size buffer. In cases where that buffer is not large enough, it receives accurate length information from DB2, so it can then allocate the appropriate amount of storage and then retrieve the remaining, unreturned data for the truncated columns.

Example 4-18 FETCH CONTINUE with dynamic SQL

Assumptions:

Table exists created as following:

```
CREATE TABLE T1
  (C1 INT,
   C2 CLOB(100M),
   C3 CLOB(32K));
```

There is a row in the table T1 where:

- C1 - valid integer
- C2 - 10MB object

C3 - 32KB object

Program Flow:

```
[1] EXEC SQL DECLARE CURSOR1 CURSOR FOR DYNSQLSTMT1;
    EXEC SQL PREPARE DYNSQLSTMT1 FROM 'SELECT * FROM T1';
[2] EXEC SQL DESCRIBE DYNSQLSTMT1 INTO DESCRIPTOR :SQLDA;
[3] EXEC SQL OPEN CURSOR1;
[4] Prepare for FETCH:
    Allocate data buffers (32K for each CLOB)
    Set data pointers and lengths in SQLDA.
[5] EXEC SQL FETCH WITH CONTINUE CURSOR1 INTO DESCRIPTOR :SQLDA;
[6] if truncation occurred on any LOB column
    loop through each column
        if column is LOB and was truncated
            allocate larger buffer area for any truncated columns, move
            first piece into larger area
            reset data pointers, length fields in SQLDA
        endif
    endloop
[7] EXEC SQL FETCH CURRENT CONTINUE CURSOR1 INTO DESCRIPTOR :SQLDA;
    endif
Work with returned data...
[8] EXEC SQL FETCH WITH CONTINUE CURSOR1 INTO DESCRIPTOR :SQLDA;
[9] EXEC SQL CLOSE CURSOR1;
```

Description:

1. The application declares a cursor for a dynamic SQL statement, then prepares a SELECT statement which retrieves LOB columns of different sizes.
2. The application DESCRIBES the statement. This populates the SQLDA with the initial data type and length information.
3. The application opens the cursor.
4. The application prepares for the FETCH by allocating storage to receive each of the output columns. For the LOB and XML columns, it allocates 32,767 bytes. This is an arbitrary size. A larger, or smaller size could be used. This example assumes that the programming language being used allows for dynamic storage allocation. The application then completes the SQLDA setup in preparation for the FETCH. It sets the SQLDATA pointers to point at each allocated buffer area and sets the SQLLONGLEN field to 32,767. It can optionally set each SQLDATALEN field to point at a 4-byte length field to hold the LOB output length.
5. The application issues the FETCH request using the new WITH CONTINUE clause to tell DB2 to manage LOB and XML truncation on output differently, as described below. After the FETCH, the buffers contain the complete data for C1 (the integer) and C3 (it fits in the 32 KB buffer). Because the data of C2 is greater than 32 KB, truncation occurs for this column. The FETCH returns a truncation warning - SQLWARN1 is set to 'W', and SQLCODE +20141 might be returned.

Because the FETCH CONTINUE flag is on for truncated column C2, DB2 performs the following actions:

The amount of data written to the data buffer equals the length specified by the SQLLONGLEN field in the secondary SQLVAR minus possibly 4 bytes depending on whether SQLDATALEN is NULL or not.

The remaining data remains materialized (cached) at the server, and it can be retrieved by the application using `FETCH CURRENT CONTINUE` immediately following the current `FETCH`. If the data contains multi-byte characters, a partial character might result at the truncation point because the data is truncated on a byte boundary.

The required buffer length is reported in one of two places. If the `SQLDATALEN` field in the secondary `SQLVAR` is not `NULL`, it contains a pointer to a 4-byte long buffer that contains the required buffer length in bytes (even for `DBCLOB`). If the `SQLDATALEN` field is `NULL`, the required buffer length (in bytes for `CLOB` and `BLOB`, in characters for `DBCLOB`) is stored in the first 4 bytes of the buffer pointed at by the `SQLDATA` field in the base `SQLVAR`. A required buffer length is the length of buffer space required to hold the entire data value; therefore, it includes the amount of data already written to the data buffer. For this example, assume that the `SQLDATALEN` pointer is not null.

6. The application checks the result of the `FETCH` and processes the returned data. If any data has been truncated, then the `SQLWARN1` field in the `SQLCA` is set to 'W'. `SQLCODE +20141` is returned when a LOB value is truncated and the length of the value that was truncated is too large to be returned in the indicator variable. The indicator variable contains a value of 32K.

However, checking `SQLWARN1` is the best way to check for truncation. In this example, we know that truncation has occurred. So, the application loops through each output column to find the truncated columns. For the LOB columns, it does this by comparing the value pointed at by `SQLDATALEN` with the value in `SQLLONGLEN` in the corresponding secondary `SQLVAR`. If the `SQLDATALEN` value is greater, truncation has occurred. The application then uses the value pointed at by `SQLDATALEN` to allocate a new buffer. It then copies the first piece of data into the new buffer and resets the `SQLDATA` pointer to point just past that new data piece. `SQLLONGLEN` is then set to the new buffer length minus the length of the first chunk (32,767 in this case).

Alternatively, the application could have set the `SQLDATALEN` pointer to zero. In that case, the processing would be similar, except that DB2 would place the actual length into the first four bytes of that data output area pointed at by `SQLDATA`.

7. The application issues a `FETCH CURRENT CONTINUE`. DB2 then processes the `SQLDA`, ignores `SQLVARs` for columns that are neither LOB nor XML, and finds that there is data cached for C2. DB2 then writes the data to the provided host variables in the same way that it would for a normal `FETCH` operation, but begins at the truncation point.

The application then processes the returned data in the data buffers. In this example, the application allocated the buffer sizes for the `FETCH CURRENT CONTINUE` to be successful. However, if the one of the data buffers was still too small, DB2 would again set the truncation warnings and lengths as described on the `FETCH WITH CONTINUE` step.

8. This `FETCH` operation just fetches the next row of data in the result set. In this example, the application consumed all of the LOB data and did not leave any truncated data. But if it had, this `FETCH` would make that data unavailable. The application is not required, when using this continue capability, to consume all of the truncated data. When the cursor is moved to the next row, or closed, that data is then unavailable. Steps 4 through 8 can be repeated until the application does not request any more rows, the application closes the cursor, or in the case of non-scrolling cursors, there are no more rows of the result set available.
9. The application closes the cursor. Similarly, if there had been any truncated LOB columns that had not been fully fetched, DB2 would discard the remaining data.

Stream the data through a single fixed-size buffer

After the original `FETCH`, if there is more data remaining, use as many subsequent `FETCH CONTINUE` statements as necessary to consume the data, using the same buffer area. This assumes that the data in the buffer is processed after each `FETCH` or `FETCH CONTINUE`

operation. For example, it is written to a file, or piped to another tool. See Example 4-19 below for a more detailed description of the processing.

Example 4-19 FETCH CONTINUE with static SQL

Assumptions:

Table exists created as following:

```
CREATE TABLE T1
  (C1 INT,
   C2 CLOB(100M),
   C3 CLOB(32K));
```

There is a row in the table T1 where:

```
C1 - valid integer
C2 - 10MB object
C3 - 32KB object
```

Program Flow:

```
[1] EXEC SQL BEGIN DECLARE SECTION
    DECLARE CLOBHV SQL TYPE IS CLOB(32767);
    EXEC SQL END DECLARE SECTION;

[2] EXEC SQL DECLARE CURSOR1 CURSOR FOR SELECT C2 FROM T1;
[3] EXEC SQL OPEN CURSOR1;
[4] EXEC SQL FETCH WITH CONTINUE CURSOR1 INTO :CLOBHV;
[5] if (sqlcode >= 0) + sqlcode <> 100
    loop until LOB is completely fetched (no truncation occurred - compare
                                         returned length to provided buffer
                                         length or examine the contents of
                                         SQLWARN1)
        write current piece of data to output file
[6]     EXEC SQL FETCH CURRENT CONTINUE CURSOR1 INTO :CLOBHV;
    endloop
endif
[7] EXEC SQL CLOSE CURSOR1;
```

Description:

1. The application declares a CLOB host variable that it uses to fetch the CLOB into.
2. The application declares a cursor for a static SQL SELECT statement that retrieves one CLOB column from the table.
3. The application opens the cursor.
4. The application issues the FETCH request. It uses the WITH CONTINUE clause on the FETCH to enable subsequent FETCH CURRENT CONTINUE operations. The precompiler generates the code that sets up the appropriate indicators in the RDI parameter block.

DB2 sees that FETCH WITH CONTINUE was specified and processes column C2 accordingly:

- The amount of data written to the data buffer equals the length specified by the SQLLONGLEN field in the secondary SQLVAR minus 4 bytes. The remaining data remains materialized (cached) at the server, and can be retrieved by the application using FETCH CURRENT CONTINUE immediately following the current FETCH. If the data contains multi-byte characters, a partial character might result at the truncation point because the data is truncated on a byte boundary.

- The precompiler-generated code does not use the SQLDATALEN field, so the required buffer length is reported (in bytes for CLOB and BLOB, in characters for DBCLOB) in the first 4 bytes of the buffer pointed at by the SQLDATA field in the base SQLVAR. The required buffer length is the length of buffer space required to hold the entire data value, therefore, it includes the amount of data already written to the data buffer.
5. The application checks for a successful fetch and then enters a loop in which it writes the buffer contents out to an external file, then checks if truncation occurred. To check for truncation, the application first checks the SQLWARN1 field to see if it is set to 'W'. If so, that means that at least one column was truncated. To check each column, the application must compare the length returned in the first 4 bytes of the output data with the length of the buffer that it provided (this is still set in SQLLONGLEN). If there was truncation, it executes the FETCH CURRENT CONTINUE statement to get the next piece of data. This is repeated until the LOB column is completely fetched. The check for truncation involves comparing the integer value in the first 4 bytes of the data buffer with the length of the input host variable.
 6. When doing the FETCH CURRENT CONTINUE, the application uses a direct host variable reference in the INTO clause. If there had been other host variables in the original SELECT list, those would have had to have been specified in the INTO clause as well.

To process the FETCH CURRENT CONTINUE statement, DB2 writes data to the output host variables in the same way that FETCH does, but beginning at the truncation point. DB2 only writes out data for LOB or XML columns that were previously truncated. Other columns are ignored. The application processes the returned data in the data buffers. In this case, the application allocated the required sizes for the FETCH CURRENT CONTINUE to be successful. However, if the LOB data buffer is still too small, DB2 would again set the truncation warnings and lengths as described on the FETCH step. One difference is that the length returned in the first 4 bytes on the FETCH CURRENT CONTINUE statement is equal to the length of the data from the truncation point to the end.
 7. After the loop, the application closes the cursor. If there had been truncated columns with unfetched data remaining, the unfetched data would have been discarded.

4.6.3 Finding the nth occurrence of a string

Since the POSSTR function provides you with the ability to find the first occurrence of a string, you probably want to find the second or third position of your search string. DB2 allows you to use POSSTR function to succeed anyway, but you have to combine it with SUBSTR, because with both functions, you are able to quickly find the position you need. Except that from the application's point of view, it is a bit more difficult than finding the first position. Example 4-20 provides a possible solution for searching a LOB value for the position of a search string you really need.

Example 4-20 Finding a specific occurrence of a string

```
EXEC SQL
  SET :POS = POSSTR (:LOB-LOCATOR, :SEARCH-STRING)
END-EXEC

[determine if correct position is returned]

IF WRONG-POSITION THEN
  MOVE 0                                TO FINAL-POS
                                      START-POS

  PERFORM UNTIL CORRECT-POSITION
    ADD POS                            TO FINAL-POS
```

```

COMPUTE POS-START = FINAL-POS + STRING-LENGTH

EXEC SQL
  SET :POS = POSSTR (SUBSTR (:LOB-LOCATOR, :POS-START), :SEARCH-STRING)
END-EXEC

[determine if correct position is returned]
END-PERFORM
END-IF

```

The first POSSTR statement returns as usual the first search string position in the LOB value 'hiding' behind a LOB locator.

Beginning with DB2 9, you can also use POSSTR function to search your host variable instead searching a value inside of DB2. Note that you add the additional overhead of invoking SQL for using POSSTR instead of using the appropriate functions of your host programming language.

4.7 Updating LOBs

In this section, we look at different ways of manipulating a LOB without retrieving it.

In most common scenarios, you would only need to manipulate CLOBs. Manipulating BLOBs can result in unusable binary data if important parts are removed or updated. The updated binary value might become unreadable for the application using this data. As an example, just think of updating parts of a JPEG picture or an MPEG movie. Cutting out or updating some data in the movie might destroy it for further use.

4.7.1 Deleting a specific part of a LOB

To delete a specific part of a LOB value, the first step is to locate the start and the end positions for your delete. In our example, we are going to delete 'Chapter 8' of our book CLOB. The first step our application should do is to assign a locator on the LOB we want to update. After the locator is set, we use the POSSTR statement to figure out the position of the beginning of 'Chapter 8'. The position of the end of 'Chapter 8' in our book is marked by the string 'Chapter 9'. When our application knows both positions, it can assign a new locator using the SUBSTR function to point to the beginning of the book up to the beginning of 'Chapter 8' minus one byte, concatenating the rest of the book beginning at the end position of 'Chapter 8', which is the beginning of 'Chapter 9'. After the new locator is established, the LOB is finally updated. See Example 4-21 for pseudo code performing the actions as mentioned above.

Example 4-21 Delete Chapter 8 of book CLOB using locators

Definitions:

LOB-LOCATOR-1	USAGE IS SQL TYPE IS CLOB-LOCATOR
LOB-LOCATOR-2	USAGE IS SQL TYPE IS CLOB-LOCATOR
START-POSITION	PIC S9(9) USAGE IS BINARY
END-POSITION	PIC S9(9) USAGE IS BINARY

Pseudo-Code:

```
EXEC SQL
```

```

SELECT LOB
INTO   :LOB-LOCATOR-1
FROM   BASE_TABLE
WHERE  KEYCOL1 = :HV-KEYCOL1
END-EXEC

EXEC SQL
  SET :START-POSITION = POSSTR (:LOB-LOCATOR-1, 'Chapter 8')
END-EXEC

EXEC SQL
  SET :END-POSITION = POSSTR (:LOB-LOCATOR-1, 'Chapter 9')
END-EXEC

EXEC SQL
  SET :LOB-LOCATOR-2 = SUBSTR (:LOB-LOCATOR-1, 1, :START-POSITION - 1) CONCAT
                        SUBSTR (:LOB-LOCATOR-1, :END-POSITION)
END-EXEC

EXEC SQL
  UPDATE BASE_TABLE
  SET LOB      = :LOB-LOCATOR-2
  WHERE KEYCOL1 = :HV-KEYCOL1
END-EXEC

EXEC SQL
  FREE LOCATOR :LOB-LOCATOR-1, :LOB-LOCATOR-2
END-EXEC

```

A sample program showing how to delete a specific part of a LOB using locators is included in BLOB and CLOB sample files described in Appendix A, “Additional material” on page 259.

4.7.2 Updating a specific part of a LOB

When you consider updating your LOB values in your application, you can build the new content of the LOB in the same way that we have shown for insert cases in 4.4.2, “Inserting LOBs using the host application” on page 85. The other way to apply necessary updates consists of locator usage. Using locators, you can simply replace parts of your LOB value or even delete them. Depending on the size of your LOBs and the number of updates you want to perform on a single LOB value, you have to decide when you want to use locators and when to replace a whole LOB value by building it in the application’s memory.

You can easily compare updating a part of a LOB with deleting a part of a LOB as shown in Case 1, except that the new locator is set up in a different way.

After determining the start and end positions of your update, the only difference to Example 4-22 is the changed assignment of LOCATOR-2.

Example 4-22 Updating a part of a CLOB

```

EXEC SQL
  SET :LOB-LOCATOR-2 = SUBSTR (:LOB-LOCATOR-1, 1, :START-POSITION - 1) CONCAT
                        :NEW-TEXT                                     CONCAT
                        SUBSTR (:LOB-LOCATOR-1, :END-POSITION)
END-EXEC

```

So LOCATOR-2 is made of the former text referenced by LOCATOR-1 up to the start position minus one byte, the NEW-TEXT variable which can consist either of a host variable or a LOB locator, and the remaining text referenced by LOCATOR-1 from your end position up to the end of the LOB value.

Using the technique mentioned above, you are also able to insert certain new text at a particular position in your CLOB. The only thing you have to figure out is the position where you want to insert the text in your CLOB.

A sample program showing how to update a specific part of a LOB using locators is included in the BLOB and CLOB sample files described in Appendix A, “Additional material” on page 259.

Let us now assume that you want to insert more text at the end of ‘Chapter 8’. For this reason, you only have to find a position in the LOB where you want to place the new text value. After you have determined the correct position by using POSSTR, you can assign a new locator to the complete new value of the LOB. Example 4-23 shows you how you can perform an insert of new text to a known position in a LOB.

Example 4-23 Inserting new text at a particular position

```
EXEC SQL
  SET :LOB-LOCATOR-2 = SUBSTR (:LOB-LOCATOR-1, 1, :START-POSITION - 1) CONCAT
                        :NEW-TEXT                                     CONCAT
                        SUBSTR (:LOB-LOCATOR-1, :START-POSITION)
END-EXEC
```

In this example, the NEW-TEXT variable can also be a host variable or another LOB locator.

4.7.3 Updating the entire LOB value

Updating an entire LOB value can also be done by using the techniques as we describe them for INSERT in 4.4.2, “Inserting LOBs using the host application” on page 85. Note that there is no difference for DB2 LOB management between replacing a number of bytes in one data page inside a LOB value and replacing the entire LOB value. In both cases, all LOB data pages are deallocated and the new value is inserted again into free LOB data pages. This means that even updating a single byte in a certain LOB value results in a much higher number of pages (and possibly I/O) involved than just the small number of pages needed to locate the affected part of the LOB and replacing it. Additionally, the elapsed time for updating a small part of a LOB value forces the entire LOB to be inserted again, and with logging activated, the elapsed time of your transaction can increase again.

An UPDATE statement to a LOB value is allowed to contain:

- ▶ The LOB value itself inside a host variable
- ▶ The reference to a LOB locator
- ▶ A file reference variable in DB2 9

4.8 General best practices

Depending on the version of DB2 that you use, the advice for best practices is different because the new functions of DB2 9 can simplify your life with LOBs. First, we list general advice for all versions of DB2 before we have to distinguish between best practices for DB2 V8 and DB2 9 users regarding the improved functionality of DB2 9. We suggest:

- ▶ If only a small number of your LOB values can fit into the *dedicated buffer pool*, use a relatively small buffer pool for your LOB table space to avoid inefficient usage of your system resources. Just make sure the buffer pool is large enough to avoid disabling list prefetch because of a buffer shortage condition.
- ▶ Always use LOG YES or the LOGGED keyword to avoid recovery complications unless you are really looking for a challenge.
- ▶ In data sharing, use GBPCACHE SYSTEM:
 GBPCACHE SYSTEM parameter was added for LOBs to prevent LOB data from flooding your group buffer pool. Specifying GBPCACHE SYSTEM caches only the LOB control (system) information in the group buffer pool.
 In a data sharing environment, GBPCACHE SYSTEM is recommended for large objects. See 8.3, “Buffer pools and group buffer pools” on page 244 for more information.
- ▶ If you use IBM WebSphere® MQ to store large messages into LOB table spaces in a shared-queue environment, use GBPCACHE CHANGED because it is recommended for volatile LOB data and in DB2 9.
- ▶ Use the LOAD utility with file reference variables to insert your LOB data into the target table space outside of your applications.
- ▶ Use the UNLOAD utility with file reference variables to unload your LOB data into data sets outside of your applications.

See Table 1-1 on page 5 for the required maintenance for supporting LOAD and UNLOAD of LOB data greater than 32 KB. In general, you should check APAR II13767 to verify the recommended maintenance when using LOBs.

If you are using DB2 V8

With DB2 V8:

- ▶ Do not misuse LOBs to disable logging.
- ▶ Use large host variables to insert your LOB data from inside your applications.
- ▶ Use large host variables with locator chains to insert your LOB data from inside your applications if a host variable is not large enough to hold the entire LOB value.

If you are using DB2 9

With DB2 9:

- ▶ Use file reference variables to insert your LOB data from inside your applications.
- ▶ Use file reference variables to unload your LOB data using your applications.
- ▶ Use SHRLEVEL CHANGE with CHECK DATA and CHECK LOB to improve concurrency and availability.
- ▶ Use REORG SHRLEVEL REFERENCE to reclaim free space in the LOB table space while ensuring access to your LOB data by read applications.
- ▶ Consider changing from GBPCACHE SYSTEM to CHANGED.



SAP usage of LOBs

This chapter discusses the usage of LOBs from the application point of view and in particular from the perspective of SAP. This is of interest because SAP is one of the largest software companies in the world in terms of market capitalization. It is also the largest business application and Enterprise Resource Planning (ERP) solution software provider worldwide in terms of revenue and an important driver of DB2 business and functionalities.

From the DB2 perspective, the SAP applications can be looked at as just one example of how LOBs are used, or, if you prefer, misused. The points discussed in this chapter, while of course, are in some way specific to SAP, however, can be of interest to any application or program which uses LOBs and is running on DB2 for z/OS. Most features, ideas, or techniques discussed in this chapter can be easily extended to other applications or programs.

This chapter starts with a short overview of the history of LOB usages by SAP. Then, we turn our attention to the different programming techniques used to access LOBs.

We highlight the ways in which LOB usage differs from what could be expected from a more native DB2 point of view. There are inferences that can be drawn from the experiences and design of the SAP applications that apply to LOB usage in a more general context.

We give some details of different optimization techniques which have been implemented into the SAP database interface to boost the performance of LOB access. These techniques are not particular to any application but can be used by any other persistency layer. Some of the ideas have been implemented within the database itself in Version 9.1 of DB2 for z/OS. This is discussed in 5.3.2, “CLI Streaming Interface” on page 134.

Traces and different possibilities to monitor LOB access are also discussed in some length.

We also briefly discuss the improvements of DB2 9 with respect to the usage of LOBs by SAP.

5.1 Overview of SAP usage of LOBs

SAP stores data in the persistent database layer that conforms to the definition of *Large Objects*. Prior to the support of LOB column types by DB2 for z/OS, SAP used application techniques to store this data into the standard column types of DB2. With LOB columns becoming available, a number of these database types have been changed to use LOBs. This change simplifies the database interface and simplifies code, reducing maintenance and potential for errors. The various usage and performance-related issues surrounding this change provide practical examples that should be applicable to everyone using LOB columns.

It should be noted that the SAP Web Application Server model allows the application programmers to abstract from the underlying data model, with DDL being generated and executed independently from the programmers. Programs treat access to the database as a logical interface, independent of the underlying DBMS. Hence, the considerations described in this chapter are derived from the experiences across all SAP implementations on the DB2 for z/OS platform, because the usage of LOBs is inherent in the solution.

5.1.1 Some history of SAP LOB usage

SAP started to use LOBs with its Web Application Server (SAP Web AS) Version 6.10 in 2001; specifically, LOBs were used for ABAP reports and Dynpros. ABAP stands for Advanced Business Application Programming, the SAP-created computer language. Web Dynpro for ABAP (WD4A, WDA) is the SAP standard user interface technology for developing Web applications in the ABAP environment. Because this usage is still the most prominent and has many implications, it is discussed in a separate section.

SAP Web AS 6.10 started to use BLOB and CLOB as the database types that correspond to the ABAP types Rawstring and String.

As of SAP NetWeaver 2004s, SAP on DB2 for z/OS began supporting Unicode. The Unicode implementation for SAP data in DB2 for z/OS is UTF 16. The corresponding data type for CLOB was DBCLOB.

With the SAP NetWeaver 2004s, SAP introduced its J2EE™ Engine. The SAP Java application server always, as all Java programs do, uses Unicode. The persistency for string-like objects and for long byte arrays is achieved by the use of DBCLOBs and BLOBs.

5.1.2 Basic architecture

Figure 5-1 on page 127 is intended to give an overview of the basic architecture of the SAP Web Application Server.

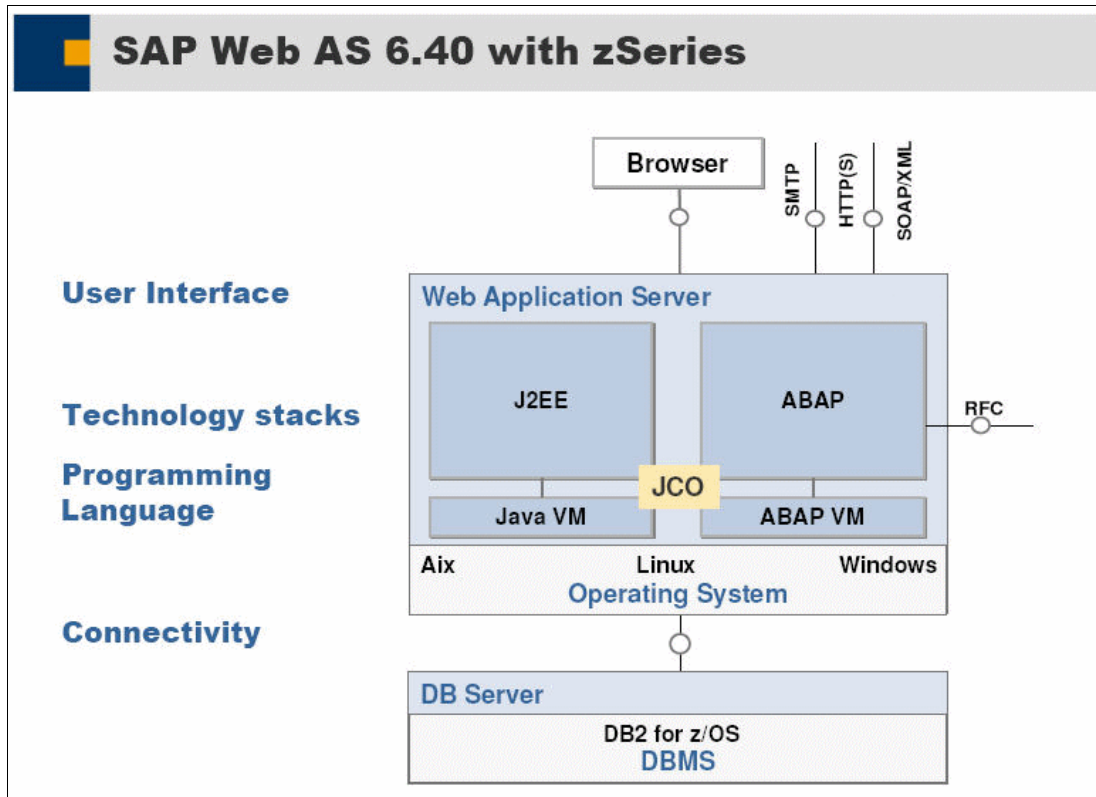


Figure 5-1 Overview of SAP Web AS 6.40 on DB2 for z/OS (c) SAP AG; 2006

The SAP Web AS consists of two stacks, the ABAP stack and the Java stack. The ABAP stack is based on a kernel written in C or C++. This kernel interacts with the database by means of a database interface library, which is referred to as `lib_dbs1`. The Java stack uses a JDBC driver as connectivity.

5.1.3 Connectivity

SAP has used different connectivity middleware in the past. In this section, we give a short overview and explain why we only discuss distributed connections in this chapter. With SAP NetWeaver 2004s, the Integrated Call Level Interface (ICLI) connectivity was phased out and DB2 Connect was introduced. ICLI is a protocol that was delivered with z/OS; it was used by SAP in prior releases and essentially provided a subset of DRDA-like connectivity. SAP NetWeaver 2004s is also the last release to support the application server running natively on z/OS. Starting with release SAP NetWeaver 2004s, connectivity is only supported using Distributed Data Facility (DDF) on the server side and DB2 Connect on the client side. This client-side software can be either the CLI interface of DB2 Connect or the DB2 JDBC Universal Driver (JCC) for the Java stack. Therefore, we only discuss distributed connections in this chapter.

5.1.4 Why use LOBs

SAP introduced LOBs to simplify data management. If the burden to handle the length of an object is shifted to the database, the interfaces to the database become easier to code and to maintain, thus, reducing the number of problems and therefore the maintenance effort required at the customer side.

(c) SAP AG; 2006

Also, SAP expects a performance improvement in the long run because less code is executed on the Application Server and potentially fewer SQL statements are executed on the database.

Prior to the arrival of LOBs, logical SAP objects with a large amount of data attached to them had to be blocked into 32 KB chunks. Thus, for one logical object several corresponding rows would exist on the database. Assurance of consistency for write and read (under a heavy load) is then a much more difficult task. If one piece of data is changed, several rows on the database have to be locked.

Currently, not all interfaces have been changed to use LOBs. For example, the SAP Cluster interface is still using blocked objects today. The reason is that the current implementation performs very well and is very critical. Also, migrating to a new data format makes an unload and load necessary. As the data volume can be huge, this is currently not feasible because this would induce long down times during the upgrade.

In 5.2, “ABAP and Dynpro source and Load” on page 129, we discuss the REPOLOAD and REPOSRC tables as examples for SAP LOB tables. The interface to these tables is less critical than the Cluster interface, because the REPOLOAD table is buffered on the Application Server. In a production environment that has gone through the initial startup phase, the REPOLOAD and REPOSRC tables are not accessed in an extensive manner. However, this is not true for a development system.

5.1.5 SAP usage of LOBs in terms of number and size

To provide an overview, we start by giving the numbers of objects used in one SAP NetWeaver 2004s system where most usage types (business process specific applications) are deployed. You can see that while the Java stack is rather small in terms of columns and tables, it has relatively more LOB tables than the ABAP stack. Because the Java component of SAP will grow in the future, LOB usage will become even more important.

Table 5-1 LOB objects in one SAP system (c) SAP AG; 2006

Number of:	ABAP stack	Java stack
DBCLOB columns	525	62
BLOB columns	246	46
Columns (total)	602,348	1,164
Tables (total)	51,329	190

In this section, we give an example of the total sizes of LOB columns in a Multiple Components in One Database (MCOD). See Table 5-2 on page 129.

An MCOD system contains more than one SAP system. More details can be found in *SAP on DB2 Universal Database for OS/390 and z/OS: Multiple Components in One Database (MCOD)*, SG24-6914. Each SAP system stores its data in a different schema in the same DB2 subsystem. In the example, there are three SAP Enterprise systems within one database.

(c) SAP AG; 2006

Table 5-2 Total size of all LOB columns (c) SAP AG; 2006

Column type	Total size of all columns in KB
BLOB	21,302,815
CLOB	5,042
DBCLOB	32,236

These numbers can be compared to the total size of the system as shown Figure 5-2.

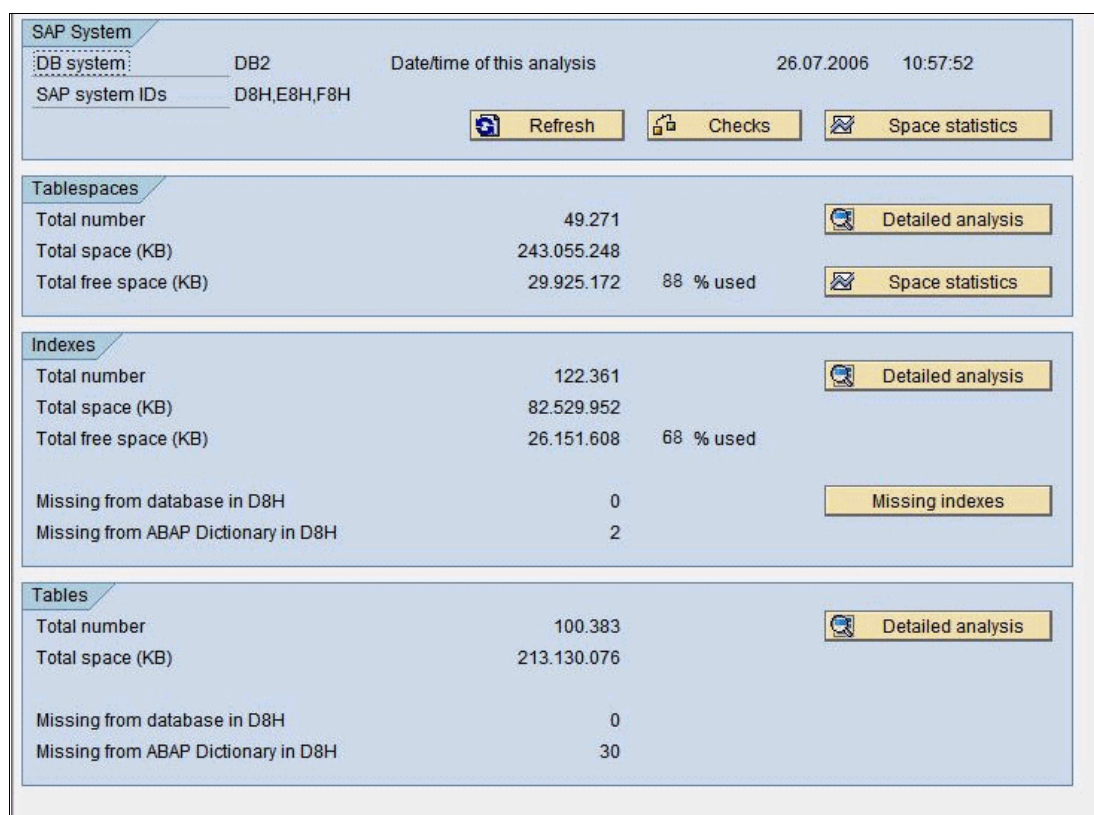


Figure 5-2 Total space usage for SAP MCODE system (c) SAP AG; 2006

5.2 ABAP and Dynpro source and Load

Advanced Business Application Programming (ABAP) is a high-level programming language created by SAP. It is an application-specific fourth-generation language. ABAP provides a high level of abstraction from the basic database level. For example, ABAP supports the concept of logical databases. It is positioned as the language for programming SAP's Web Application Server. SAP Dynpro is an SAP screen that allows the user to interact with the application. The Dynpro forms the heart of any transaction.

The ABAP report sources are stored in a table named REPOSRC, and the generated reports are stored in a table named RELOAD. For Dynpros, the tables are named DYNPSOURCE and DYNPLOAD, respectively.

Because most of the SAP system and all of the application logic are coded into ABAP reports and Dynpros, these tables are the heart of an SAP system. Every time a report is modified, created, transported, compiled, or executed while it is not yet in the local (SAP) buffer, LOBs are read from or written to the database. Concepts such as local SAP buffering and transporting programs are SAP specific topics beyond the scope of this IBM Redbook. As a result, the requirements for stability, robustness, and performance for LOB objects are the same or higher as for every other data type.

In Figure 5-3, we display the structure of the REPOLOAD table using SAP's Data Dictionary Tools. The columns that map to a LOB column in this case are LDATA and QDATA, with an SAP data type of RAWSTRING.

Field	Key	Initi	Data element	Data Type	Length	Deci	Short Text
PROGNAME	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	PROGNAME	CHAR	40		0 ABAP Program Name
R3STATE	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	R3STATE	CHAR	1		0 ABAP: Program Status (Active, Saved, Transported...)
MACH	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	MACH	INT2	5		0 ABAP: Host ID
UNAM	<input type="checkbox"/>	<input checked="" type="checkbox"/>	UNAM	CHAR	12		0 Last changed by
UDAT	<input type="checkbox"/>	<input checked="" type="checkbox"/>	RDIR_UPDATE	DATS	8		0 Changed On
UTIME	<input type="checkbox"/>	<input checked="" type="checkbox"/>	DDTIME	TIMS	6		0 Dictionary: time of last change
L_DATALG	<input type="checkbox"/>	<input checked="" type="checkbox"/>	DATALG	INT4	10		0 ABAP/4: Length of a program component
Q_DATALG	<input type="checkbox"/>	<input checked="" type="checkbox"/>	DATALG	INT4	10		0 ABAP/4: Length of a program component
SDAT	<input type="checkbox"/>	<input checked="" type="checkbox"/>	DD10_SDATE	DATS	8		0 ABAP: Date of last generation
STIME	<input type="checkbox"/>	<input checked="" type="checkbox"/>	DD10_STIME	TIMS	6		0 ABAP: Time of last generation
MINOR_VERS	<input type="checkbox"/>	<input checked="" type="checkbox"/>	MINOR_VERS	INT2	5		0 MINOR_VERS in REPOLOAD
MAJOR_VERS	<input type="checkbox"/>	<input checked="" type="checkbox"/>	MAJOR_VERS	INT4	10		0 MAJOR_VERS in REPOLOAD
LDATA	<input type="checkbox"/>	<input type="checkbox"/>		RAWSTRING	0		0 Load (Compressed)
QDATA	<input type="checkbox"/>	<input type="checkbox"/>		RAWSTRING	0		0 Line Reference (Not Compressed)

Figure 5-3 REPOLOAD structure (c) SAP AG; 2006

In Figure 5-4 on page 131, we display part of the structure of the REPOSRC table. The table has 34 columns in total. The column that maps to a LOB column in this case is DATA, with an SAP data type of RAWSTRING.

Table 5-3 Details of Repo tables' LOB content (c) SAP AG; 2006

	REPOLOAD LDATA	REPOLOAD QDATA	REPOSRC DATA
SUM	3,390,540,497	1,607,356,266	2,362,162,619
AVG length	24,865.54	11,788.03	1,561.80
Total number of rows	136,355	136,355	1,512,436
Total < 32 KB	114,032	126,566	1,505,927
Total (32 KB - 64 KB)	10,208	3,709	5,134
Total (64 KB - 1 MB)	12,114	6,080	1,375
Total >= 1 M	1	0	0
Total > 100 M	0	0	0

A good test case to check LOB performance is to run transaction SGEN (see Figure 5-5). This transaction can be used to generate some or all ABAP reports. It is mainly called after an upgrade or an install, because these operations either invalidate the load or just provide the sources. The program reads data from REPOSRC (and others) where the majority of data is in a LOB column, and updates data in the LOB columns of REPOLOAD (and others). As such it is a very effective test case of LOB performance, and is also able to produce repeatable loads with the same underlying data manipulation.

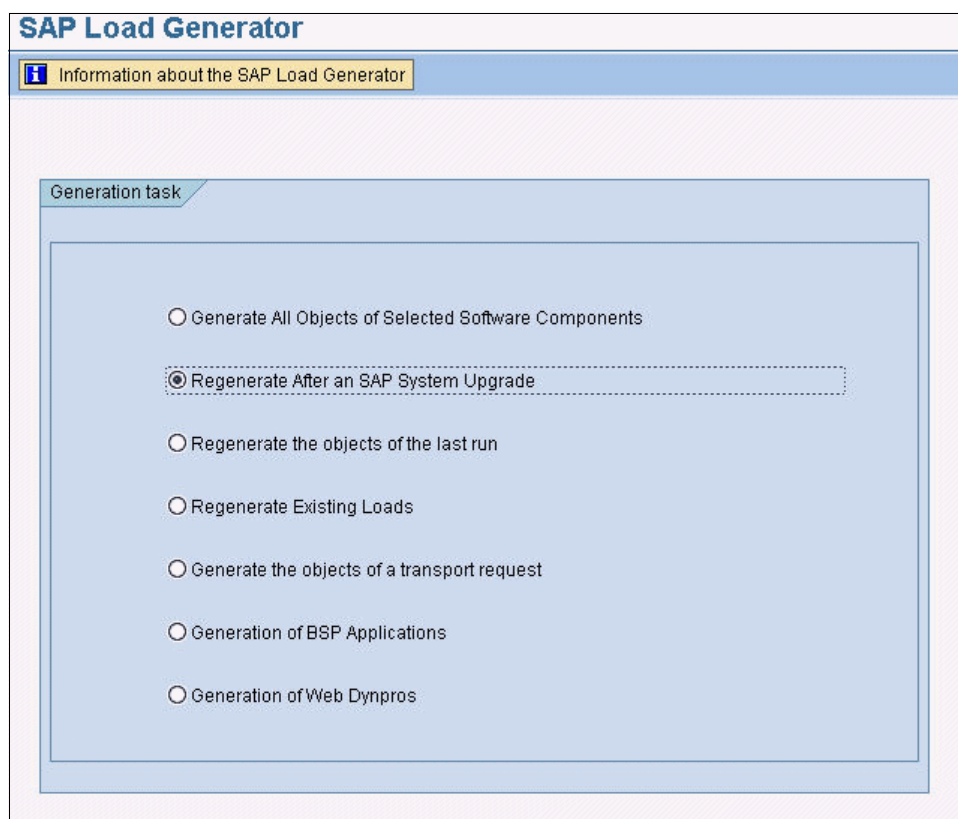


Figure 5-5 Transaction SGEN (c) SAP AG; 2006

(c) SAP AG; 2006

Another good test case is to load the REPOLOAD table via R3load SAP loading tool. This tool uses mass insert without any application logic as the SAP transport tools and R3trans do.

5.3 Programming techniques for the ABAP stack

In this section, we discuss how LOBs are accessed on the ABAP stack. The Java stack is discussed in 5.5, “Programming techniques with JDBC” on page 140.

5.3.1 Basic locator access

Statements can access LOBs either in one chunk or piece, or they can access them one piece at a time or piece-wise. The piece-wise access to LOBs is achieved by locators. For a SELECT statement instead of the data, a locator is retrieved and the locator is then used for fetching the data. All statements using locators operate on a dummy table. In an SAP system, the special table “#LOB” is used for this purpose. Equivalent table #LOBU is used for Unicode systems.

A typical example of how LOBs are retrieved using locators is shown in Figure 5-6. The trace is taken using SAP transaction ST05 (user trace).

<div> <div>Trace List</div> <div> <div>Trace List</div> <div>Edit</div> <div>Goto</div> <div>System</div> <div>Help</div> </div> </div> <div> <div>DDIC information</div> <div>Explain</div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> </div>									
<div>Transaction SE38</div> <div>Work process no 0</div> <div>Proc.type DIA</div> <div>Client 000</div> <div>User D022204</div> <div>TransGUID 44C68A6A8A120048020000000A11C96D</div> <div>Date 25</div>									
Duration	Obj. name	Op.	Recs.	RC	Statement				
2	REPOLOAD	REOPEN		0	SELECT WHERE "PROGNAME" = 'SAPLSSEL' AND "R3STATE" = 'A' AND "MACH" = 324 FOR FETCH ONLY				
55.130	REPOLOAD	FETCH	1	0					
3	#LOB	REOPEN		0	SELECT FROM "#LOB" FOR FETCH ONLY WITH UR				
2.292	#LOB	FETCH	1	0					
597	#LOB	CLOSE	0	0					
1	REPOLOAD	EXECSTA		0	LOB_GET_PIECE used SAPBCM.#LOB (cursor=16, type=BL0B, locator=CF7E5AE8, length=8)				
2	#LOB	REOPEN		0	SELECT FROM "#LOB" FOR FETCH ONLY WITH UR				
2.913	#LOB	FETCH	1	0					
629	#LOB	CLOSE	0	0					
1	REPOLOAD	EXECSTA		0	LOB_GET_PIECE used SAPBCM.#LOB (cursor=16, type=BL0B, locator=CF7E5AE8, length=32000)				
3	#LOB	REOPEN		0	SELECT FROM "#LOB" FOR FETCH ONLY WITH UR				
3.217	#LOB	FETCH	1	0					
757	#LOB	CLOSE	0	0					
2	REPOLOAD	EXECSTA		0	LOB_GET_PIECE used SAPBCM.#LOB (cursor=16, type=BL0B, locator=CF7E5AE8, length=32000)				
3	#LOB	REOPEN		0	SELECT FROM "#LOB" FOR FETCH ONLY WITH UR				
3.304	#LOB	FETCH	1	0					
558	#LOB	CLOSE	0	0					
1	REPOLOAD	EXECSTA		0	LOB_GET_PIECE used SAPBCM.#LOB (cursor=16, type=BL0B, locator=CF7E5AE8, length=32000)				
3	#LOB	REOPEN		0	SELECT FROM "#LOB" FOR FETCH ONLY WITH UR				
3.020	#LOB	FETCH	1	0					
884	#LOB	CLOSE	0	0					
1	REPOLOAD	EXECSTA		0	LOB_GET_PIECE used SAPBCM.#LOB (cursor=16, type=BL0B, locator=CF7E5AE8, length=32000)				
2	#LOB	REOPEN		0	SELECT FROM "#LOB" FOR FETCH ONLY WITH UR				
2.916	#LOB	FETCH	1	0					
573	#LOB	CLOSE	0	0					
2	REPOLOAD	EXECSTA		0	LOB_GET_PIECE used SAPBCM.#LOB (cursor=16, type=BL0B, locator=CF7E5AE8, length=32000)				
3	#LOB	REOPEN		0	SELECT FROM "#LOB" FOR FETCH ONLY WITH UR				
2.866	#LOB	FETCH	1	0					
727	#LOB	CLOSE	0	0					
1	REPOLOAD	EXECSTA		0	LOB_GET_PIECE used SAPBCM.#LOB (cursor=16, type=BL0B, locator=CF7E5AE8, length=32000)				
2	#LOB	REOPEN		0	SELECT FROM "#LOB" FOR FETCH ONLY WITH UR				
2.807	#LOB	FETCH	1	0					
660	#LOB	CLOSE	0	0					

Figure 5-6 SQL Trace for simple select using locators (c) SAP AG; 2006

Now, we give you an example of how LOBs are updated with locators. For the UPDATE, first a locator is retrieved. The INSERT statements work the same way. Next the data is attached to the locator. Finally, the locator is used in the statement as shown in Figure 5-7 on page 134. The locator statements in this example use table #LOBU because it is a Unicode system.

DECLARE	0	0	UPDATE SET "DATA" = ? WHERE "PROGNAME" = ? AND "R3STATE" = ?
PREPARE	0	0	UPDATE SET "DATA" = ? WHERE "PROGNAME" = ? AND "R3STATE" = ?
DECLARE	0	0	SELECT FROM "#LOBU" FOR FETCH ONLY WITH UR
PREPARE	0	0	SELECT FROM "#LOBU" FOR FETCH ONLY WITH UR
OPEN	0	0	SELECT FROM "#LOBU" FOR FETCH ONLY WITH UR
FETCH	1	0	
CLOSE	0	0	
EXECSTA	0	0	LOB_GET_LOCATOR used #LOBU (cursor=205, type=BLOB, locator=D4643398)
DECLARE	0	0	SELECT FROM "#LOBU" FOR FETCH ONLY WITH UR
PREPARE	0	0	SELECT FROM "#LOBU" FOR FETCH ONLY WITH UR
OPEN	0	0	SELECT FROM "#LOBU" FOR FETCH ONLY WITH UR
FETCH	1	0	
CLOSE	0	0	
EXECSTA	0	0	LOB_PUT_PIECE used #LOBU (cursor=205, type=BLOB, in_locator=D4643398, out_locator=D262359E, length=9)
REOPEN	0	0	SELECT FROM "#LOBU" FOR FETCH ONLY WITH UR
FETCH	1	0	
CLOSE	0	0	
EXECSTA	0	0	LOB_PUT_PIECE used #LOBU (cursor=205, type=BLOB, in_locator=D262359E, out_locator=D060379C, length=389)
REOPEN	0	0	SELECT FROM "#LOBU" FOR FETCH ONLY WITH UR
FETCH	1	0	
CLOSE	0	0	
EXECSTA	0	0	LOB_PUT_PIECE used #LOBU (cursor=205, type=BLOB, in_locator=D060379C, out_locator=DE6E3992, length=0)
EXECSTM	1	0	UPDATE SET "DATA" = <BLOB> WHERE "PROGNAME" = 'ZPMTEST' AND "R3STATE" = 'I'

Figure 5-7 SQL Trace for update using locator (c) SAP AG; 2006

The data is attached to the locator by a select. See Figure 5-8.

SQL Statement	
SELECT	
CAST(? AS BLOB) CAST(? AS BLOB)	
FROM	
"#LOB" FOR FETCH ONLY WITH UR	
Variable	
A0(BL,6)	= <BLOB>
A1(BL,6)	= <BLOB>

Figure 5-8 SELECT statement with LOB locators (c) SAP AG; 2006

If it is a non-Unicode system, table "#LOB" is used. The first host variable, the question mark character (?), is the locator on input (that is, the locator for the data already transferred to DB2). The second host variable (?) is BLOB data. The select returns a new locator pointing to the new data.

Up to release SAP NetWeaver 2004s, the lib_dbsl basically uses these direct calls to locators as explained above. However, several optimizations are built on top of this. These are explained in 5.4, "Optimization techniques and query rewrite" on page 136. In 5.3.2, "CLI Streaming Interface" on page 134, we discuss an alternative using the CLI streaming APIs.

5.3.2 CLI Streaming Interface

Instead of using direct locator calls, you can use the DB2 CLI streaming interface. The application uses CLI APIs SQLGetSubString() for piece-wise read and SQLParamData() and SQLPutData() to insert and update data.

(c) SAP AG; 2006

SQLGetSubString() is used to retrieve a portion of a large object value, referenced by a large object locator that has been returned from the server (returned by a fetch or a previous SQLGetSubString() call) during the current transaction.

SQLParamData() is used in conjunction with SQLPutData() to send long data in pieces. It can also be used to send fixed-length data at execution time.

SQLPutData() is called following an SQLParamData() call returning SQL_NEED_DATA to supply parameter data values. This function can be used to send large parameter values in pieces.

We give you an example of a typical sequence of calls from the DB2 Connect CLI trace in Example 5-1. Note that some unimportant calls have been omitted to keep the trace reasonably small and improve readability. First, the Select from REPOLOAD is prepared, then the input and output variables are bound (omitted), the cursor is opened (omitted), the fetch is executed, and the locator is retrieved and then used to SQLGetLength() to get the length of the LOB and in SQLGetSubString() to retrieve the data.

Example 5-1 CLI trace for REPOLOAD Select using SQLGetSubString() (c) SAP AG; 2006

```

SQLExtendedPrepare( hStmt=1:19 )
    ---> Time elapsed - +2.000000E-005 seconds
( pszSqlStr="SELECT "LDATA" FROM "REPOLOAD" WHERE "PROGNAME" = ? AND "R3STATE" = ? AND
"MACH" = ? FOR FETCH ONLY WITH CS ", cbSqlStr=109, cPars=3, sStmtType=SQL_CLI_STMT_SELECT,
cStmtAttrs=1( Attribute=1, piStmtAttr=SQL_CURSOR_HOLD, pvParams=0 )
SQLExtendedPrepare( )
****
SQLFetch( hStmt=1:19 )
    ---> Time elapsed - +1.900000E-005 seconds
    sqlccsend( Handle - 0000004609406048 )
    sqlccsend( ulBytes - 143 )
    sqlccsend( ) rc - 0, time elapsed - +4.000000E-006
    sqlccrecv( )
    sqlccrecv( ulBytes - 95 ) - rc - 0, time elapsed - +1.934000E-003
( iRow=1, iCol=1, fCType=SQL_C_BLOB_LOCATOR, rgbValue=BD09179B, pcbValue=4, piIndicator=4 )

SQLFetch( )
    <--- SQL_SUCCESS Time elapsed - +2.011000E-003 seconds
    *****
SQLGetLength( hStmt=1:20, fCType=SQL_C_BLOB_LOCATOR, iLocator=-1123477605,
pcbLocator=&0000000112b006e4, piIndicatorValue=<NULL pointer> )
    ---> Time elapsed - +8.000000E-006 seconds
( Row=1, iPar=1, fCType=SQL_C_BLOB_LOCATOR, rgbValue=BD09179B )
    sqlccsend( Handle - 0000004609406048 )
    sqlccsend( ulBytes - 132 )
    sqlccsend( ) rc - 0, time elapsed - +5.000000E-006
    sqlccrecv( )
    sqlccrecv( ulBytes - 92 ) - rc - 0, time elapsed - +1.084000E-003
( Row=1, iPar=2, fCType=SQL_C_LONG, rgbValue=79057 )
SQLGetLength( pcbLocator=79057 )
    <--- SQL_SUCCESS Time elapsed - +1.230000E-003 seconds
    *****
SQLGetSubString( hStmt=1:20, fCType=SQL_C_BLOB_LOCATOR, iLocator=-1123477605, iFrom=1,
iLength=9, fCTypeTarget=SQL_C_BINARY, rgbValue=&0000000113063ff0, cbValueMax=9,
pcbValue=&0ffffffffffffbe9c, piIndicatorValue=&0xffffffffffffbd64 )
    ---> Time elapsed - +8.000000E-006 seconds
( Row=1, iPar=1, fCType=SQL_C_BLOB_LOCATOR, rgbValue=BD09179B )
    sqlccsend( Handle - 0000004609406048 )
    sqlccsend( ulBytes - 132 )
    sqlccsend( ) rc - 0, time elapsed - +4.000000E-006

```

(c) SAP AG; 2006

```

        sqlccrecv( )
        sqlccrecv( ulBytes - 92 ) - rc - 0, time elapsed - +2.895000E-003
( Row=1, iPar=2, fCType=SQL_C_LONG, rgbValue=79057 )
( Row=1, iPar=1, fCType=SQL_C_BLOB_LOCATOR, rgbValue=BD09179B )
( Row=1, iPar=2, fCType=SQL_C_LONG, rgbValue=1 )
( Row=1, iPar=3, fCType=SQL_C_LONG, rgbValue=9 )
        sqlccsend( Handle - 0000004609406048 )
        sqlccsend( ulBytes - 148 )
        sqlccsend( ) rc - 0, time elapsed - +8.000000E-006
        sqlccrecv( )
        sqlccrecv( ulBytes - 111 ) - rc - 0, time elapsed - +1.802000E-003
( Row=1, iPar=4, fCType=SQL_C_BINARY, rgbValue=x'FFA42B0400121F9D02', pcbValue=9,
piIndicatorPtr=9 )

```

These APIs can only be used with the new MERGE statement available with DB2 9 for z/OS. In DB2 V8, the MERGE statement has to be emulated by a series of Update and Insert statements. The SQLPutData is not suited for this, because the data has to be sent prior to the insert and update.

Progressive streaming with DB2 Connect CLI

With DB2 Connect 9.1, the SQLGetData API implicitly uses progressive references against DB2 9 for z/OS. The default setting is that any LOB smaller than 1 MB is buffered back to the client during the fetch; anything larger uses the progressive references. This value is controlled with the LobCacheSize keyword or the SQL_ATTR_LOB_CACHE_SIZE connection or statement attribute. See also “Progressive streaming with the Java Universal Driver” on page 141 and 4.3, “DRDA LOB flow optimization” on page 79 for detailed explanations about progressive streaming.

With progressive streaming, we expect the CLI streaming interface to be as fast as or better than the currently used interface. Once this is implemented and lib_dbsl has switched to CLI streaming, the interface will be considerably less complex and easier to maintain.

5.4 Optimization techniques and query rewrite

Many problems with the performance of LOBs result from the fact that most applications or application developers regard LOBs not so much as large objects but rather as objects with unknown size. With LOB objects, the responsibility to handle the size of an object is shifted from the application code to the database. In particular, this is the case in the Java world where the Java String object is treated in the persistency layer as a CLOB. Obviously, there is a price to pay for this commodity in the form of performance. Ideally, the performance of small LOBs should not differ from that of a VARCHAR type field. This can be achieved by some programming techniques explained below. Some of these techniques have been implemented in DB2, and they are also discussed in this section.

5.4.1 Local LOB buffer

The SAP database interface can retrieve and write LOB data in several pieces or in one piece. Typically, very large LOBs should be retrieved and written piece-wise (streaming). For every piece-wise operation, a locator statement is executed, causing a network flow. To avoid the execution of locator statements for small pieces (typically less than 64 KB), these pieces are buffered in a local LOB buffer.

When fetching, the LOB buffer is filled up ahead to its total length or to the maximum length of the LOB, so that fetches which follow can be served from the buffer. (c) SAP AG; 2006

When writing, the pieces are not directly written to the database but into the buffer. The buffer is flushed to the database once it is filled or the last piece is written.

5.4.2 Retrieve length and maximal data with locator

If the LOB is small, it is much faster to retrieve the complete data instead of using a locator. For this reason, the database layer contains an optimization to rewrite statements on LOB tables so that the statement retrieves, for every fetch, the length of the LOB, the data up to the local LOB buffer length, and a locator. If the LOB is larger than the buffer size, the locator is used to fetch the remaining part of the LOB. If the LOB is smaller than the buffer size, the locator can be freed immediately. See Figure 5-9.

SQL Statement
<pre> SELECT length("LDATA") , "LDATA" , "LDATA" FROM "REPOLOAD" WHERE "PROGNAME" = ? AND "R3STATE" = ? AND "MACH" = ? FOR FETCH ONLY WITH CS </pre>
Variable
<pre> A0(CH,40) = SAPLSSEL A1(CH,1) = A A2(I4,3) = 324 </pre>

Figure 5-9 Modified statement (extended DA) on REPOLOAD (c) SAP AG; 2006

In this case, we retrieve the length, the data, and the locator. The data and locator appear both as LDATA. The data type specifies the difference. Obviously this technique requires a manipulation of the statement itself. Because the database layer under normal circumstances does not parse the statement, this technique is restricted to special interfaces, the so-called *trusted interfaces*, and several well known statements on SAP report and Dynpro source and Load tables.

The parameter `dbs/db2/use_eda` of the `dbsl_lib` profile controls this *extended description area* feature. It is called *extended* because it includes the length and the data. If `use_eda` is set to 0, the extended da feature is turned off.

5.4.3 Optimizing the free locator statement

Locators are implicitly freed at the end of the logical unit of work (LUW). Therefore, the FREE LOCATOR statements can be optimized by simply omitting them. If, however, there are too many locators used in one LUW, this can cause problems, because the resources connected to the locators are not released in this case. For this reason, the locators to be freed are buffered. There are two situations when the locators in the buffer are freed:

- If the locator free buffer is filled, all locators are released using just one statement: "FREE LOCATOR ?,...,?". The buffer can hold up to 1,000 locators. This number can be adjusted by the profile parameter `dbs/db2/lob_free_buffer`.

- The amount of storage connected to the locators is tracked. If this storage exceeds a given threshold (normally 1 GB), then the locators are freed. The threshold can be adjusted by profile parameter `db2/db2/max_lob_free_length` (in MB). Note that the same number of host variables are always used for the `FREE LOCATOR` statement, so that only one slot in the statement cache is used. To fill up the missing host variables, the last locator is repeated. DB2 issues an `SQLCODE -423` (Invalid locator value). This code is ignored by the database interface.

At every Commit or Rollback, the buffer is reset. With DB2 9, the new concept of the progressive reference is introduced. These references are implicitly freed when the cursor is closed. Therefore, the free locator buffer becomes obsolete with DB2 9.

5.4.4 Comparison of different techniques using SGEN

In Table 5-4, we display total execution time of SGEN runs on different optimization levels. The runs have been done on the `SAP_BASIS` component of an SAP NetWeaver 2004s system using an AIX® Application Server. There are 34,789 objects in one run. These measurements are not accurate performance evaluations, but they can be used to get a rough idea about the impact of different optimizations. The values should not be read in absolute numbers, but rather in ratios.

Table 5-4 SGEN runs on different optimization levels (c) SAP AG; 2006

Optimization	Execution time of SGEN in seconds
No query rewrite, no local LOB buffer	1,1603
No query rewrite, local LOB buffer = 64 KB	8,855
Query rewrite, local LOB buffer = 64 KB	6,671

5.4.5 Chaining

It is possible to have a reduction of network flows with CLI array input chaining. *CLI array input chaining* is a feature that, when enabled, causes requests for the execution of prepared insert, update, and delete statements to be held and queued at the client until the chain is ended. Once the chain has been ended, all of the chained `SQLExecute()` requests at the client are then sent to the server in a single network flow.

Chaining avoids LOBs or sparsely filled long variable length columns filling up the internal SAP buffer for array operations. The effect is that data gets buffered within the CLI layer, reducing the network flow and the number of SQL statements executed.

The following sequence of events (presented as pseudo code) is an example of how CLI array input chaining can reduce the number of network flows to the server.

The statement attribute `SQL_ATTR_CHAINING_BEGIN` is set in order to turn chaining on. To turn chaining off, the `SQL_ATTR_CHAINING_END` attribute is set. See Example 5-2.

Example 5-2 Pseudo code for chaining (c) SAP AG; 2006

```
db2rc = SQLSetStmtAttr(*stmth_p, SQL_ATTR_CHAINING_BEGIN, (SQLPOINTER) 1, SQL_IS_UIINTEGER );
db2rc = SQLEXPETE ( stmth ) ;
db2rc = SQLEXPETE ( stmth ) ;
....
db2rc = SQLSetStmtAttr(*stmth_p, SQL_ATTR_CHAINING_END, (SQLPOINTER) 1, SQL_IS_UIINTEGER );
```

Example 5-3 displays a simple case of chaining.

(c) SAP AG; 2006

Example 5-3 Chaining on insert (c) SAP AG; 2006

```
5.125 CO2MAP REEXEC 13 0 DELETE WHERE "ID" = 1105
11 CO2MAP EXECSTA 0 0 START CHAIN (cursor=70, statement=INSERT INTO "CO2MAP" VALUES( ? , ?
, ? , ? , ? , ? ) )
20 CO2MAP REEXEC 0 0 INSERT VALUES( 1105 , 18 , 2 , 1105 , 0 , 0 )
13 CO2MAP REEXEC 0 0 INSERT VALUES( 1105 , 19 , 3 , 1105 , 0 , 0 )
13 CO2MAP REEXEC 0 0 INSERT VALUES( 1105 , 21 , 4 , 1105 , 0 , 0 )
13 CO2MAP REEXEC 0 0 INSERT VALUES( 1105 , 22 , 5 , 1105 , 0 , 0 )
13 CO2MAP REEXEC 0 0 INSERT VALUES( 1105 , 24 , 9 , 1105 , 0 , 0 )
13 CO2MAP REEXEC 0 0 INSERT VALUES( 1105 , 27 , 11 , 1105 , 0 , 0 )
13 CO2MAP REEXEC 0 0 INSERT VALUES( 1105 , 30 , 14 , 1105 , 0 , 0 )
13 CO2MAP REEXEC 0 0 INSERT VALUES( 1105 , 32 , 14 , 1105 , 0 , 0 )
12 CO2MAP REEXEC 0 0 INSERT VALUES( 1105 , 34 , 14 , 1105 , 0 , 0 )
12 CO2MAP REEXEC 0 0 INSERT VALUES( 1105 , 36 , 16 , 1105 , 0 , 0 )
13 CO2MAP REEXEC 0 0 INSERT VALUES( 1105 , 38 , 16 , 1105 , 0 , 0 )
13 CO2MAP REEXEC 0 0 INSERT VALUES( 1105 , 40 , 26 , 1105 , 0 , 0 )
32 CO2MAP REEXEC 0 0 INSERT VALUES( 1105 , 43 , 36 , 1105 , 0 , 0 )
2.863 CO2MAP EXECSTA 13 0 STOP CHAIN (cursor=70, statement=INSERT INTO "CO2MAP" VALUES( ? ,
? , ? , ? , ? , ? ) )
```

The first number to the right gives the execution time in microseconds. You can see that the inserts are very fast, because they are just client-side operations. The chain buffer is then flushed to the database at chain end time.

One restriction to CLI chaining is that it can only be applied to one statement at a time; if several statements are open at the same time, then the chain buffer must be flushed before a new statement can execute.

Chaining helps with mass insert, for example, when loading the REPOLOAD table. It does not help with single LOB operations such as those used by transaction SGEN.

Chaining is always used for LOBs. In the other cases, the chaining behavior is governed by the dbsl profile parameter `dbs_db2_chaining`. If the number of rows fitting into the internal buffer is less than or equal to the value set in `dbs_db2_chaining`, chaining is used. The internal buffer usually has a size of 32 KB. The default for `dbs_db2_chaining` is 20. To turn chaining completely off, it must be set to 0.

The effect on LOB insert is shown in Example 5-4 where a copy of the REPOLOAD table is loaded with all LOBs chained.

Example 5-4 R3load log files for `dbs_db2_chaining=20` (c) SAP AG; 2006

```
(DDL) Info: Deleting data from ZZNPSOURCE
Time : 20060124 104145
(DB) INFO: ZZNPSOURCE deleted/truncated
(IMP) INFO: import of ZZNPSOURCE completed (34834 rows) #20060124 104238
(DB) INFO: disconnected from DB
R3load: job completed
R3load: END OF LOG: 20060124 104238
```

The total time for the import is 53 seconds.

The case where no LOBs are chained is shown in Example 5-5.

Example 5-5 R3load log files for `dbs_db2_chaining=0` (c) SAP AG; 2006

```
(DDL) Info: Deleting data from ZZNPSOURCE
Time : 20060124 122423
```

(c) SAP AG; 2006

```
(DB) INFO: ZZNPSOURCE deleted/truncated
(IMP) INFO: import of ZZNPSOURCE completed (34834 rows) #20060124 122542
(DB) INFO: disconnected from DB
R3load: job completed
R3load: END OF LOG: 20060124 122542
```

The total time for the import is 79 seconds. This yields an improvement of about 33%.

5.5 Programming techniques with JDBC

Connectivity for the Java stack is achieved by means of a JDBC driver. We discuss only the IBM DB2 Driver for JDBC and SQLJ (we refer to it as the JCC driver), which is the only JDBC driver used by SAP for both DB2 for z/OS and DB2 for Linux, UNIX, and Windows. With the JCC driver, all members of the DB2 family can be accessed. In Example 5-6, we give samples for operations on LOBs using a JDBC driver. Sample code can be downloaded as described in Appendix A, "Additional material" on page 259.

Example 5-6 LOB access with JDBC driver (c) SAP AG; 2006

```
....
String insertStmt = "insert into " + TEST_TABLE
    + " (FCHAR4, FINT, FCLOB, FBLOB) " + "values (?, ?, ?, ?)";

try {
    PreparedStatement ps = getConnection().prepareStatement(insertStmt);

    ps.setString(1, "A");
    ps.setInt(2, 1);
    char[] cArray = { ' ', 'B', 'C' };
    Reader r = new CharArrayReader(cArray);
    ps.setCharacterStream(3, r, cArray.length);

    byte[] bArray = { 'D', 'E', 'F' };
    ByteArrayInputStream bais = new ByteArrayInputStream(bArray);
    ps.setBinaryStream(4, bais, bArray.length);
    int cnt = ps.executeUpdate();

    ps.setString(1, "XYZ");
    ps.setInt(2, 3);
    Reader ucReader = new CharArrayReader(ucArray);
    ps.setCharacterStream(3, ucReader, ucArray.length);
    ps.setBinaryStream(4, null, 0);
    cnt = ps.executeUpdate();
    ....

    Reader r = null;
    InputStream is = null;
    String query = "select FCLOB, FBLOB from " + TEST_TABLE
        + " " + "where FCHAR4 = 'A' and FINT = 1";
    try {
        Statement stmt = getConnection().createStatement();
        ResultSet rs = stmt.executeQuery(query);
```

(c) SAP AG; 2006

```

        if (rs.next()) {
            r = rs.getCharacterStream(1);
            r.read(cArray2);
            is = rs.getBinaryStream(2);
            is.read(bArray2);
            ....
        }
    }

```

For the JCC driver, two properties are important with respect to LOBs when running on DB2 V8:

- ▶ **fullyMaterializeLobData**

LOB locator support

The DB2 Universal JDBC Driver can use LOB locators to retrieve data in LOB columns. To cause JDBC to use LOB locators to retrieve data from LOB columns, you need to set the `fullyMaterializeLobData` property to false.

- ▶ **fullyMaterializeInputStreams**

The property above, `fullyMaterializeLobData`, only affects the streams coming *from the server to the client*, while the property `fullyMaterializeInputStreams` only affects the streams going *from the client to the server*.

To use locators with the Java stack, both properties need to be set because the JCC driver is not reentrant. Otherwise, it would be possible that a stream is inserted that is constructed as a read of a LOB field from the database.

DB2LobTableCreator utility

If LOB locators are used to access DBCLOB or CLOB columns, the JCC driver uses the database tables SYSIBM.SYSDUMMYU, SYSIBM.SYSDUMMYA, and SYSIBM.SYSDUMMYE for fetching data.

This can be achieved by either of the following:

- ▶ On the DB2 for z/OS servers:

Customize and run job DSNTIJSG when you install a new DB2 9 or job DSNTIJMS for post-installation. These jobs are located in data set prefix.SDSNSAMP.

- ▶ On the client:

Run the `com.ibm.db2.jcc.DB2LobTableCreator` utility against each of the DB2 for z/OS servers. See DB2LobTableCreator utility for details.

Normally, this task is done on the server with job DSNTIJMS. For more information, go to:

<http://publib.boulder.ibm.com/infocenter/db2luw/v9/index.jsp?topic=/com.ibm.db2.udb.apdv.java.doc/doc/r0023715.htm>

Progressive streaming with the Java Universal Driver

If the JCC driver of DB2 Connect is used against DB2 9 for z/OS, progressive streaming and progressive references are enabled by default. See also “Progressive streaming with DB2 Connect CLI” on page 136.

With Progressive Streaming, the JCC driver materializes if the LOBs are small, and it uses locators if the LOB is large (by default, larger than 1 MB). This resolves a problem with the LOB access on DB2 V8. In a distributed environment, it is good to materialize LOBs from the performance point of view.

(c) SAP AG; 2006

However, huge LOBs must be read piece-wise, because otherwise, the client runs out of memory. In the SAP Portal, clients can store in principle any amount of data, for example, they can store complete videos with more than 500 MB. Because with DB2 V8, there is only the global property `fullyMaterializeLobData`, which applies to all statements executed using this connection, locators must always be used to retrieve any LOB within connections which access large LOBs.

The Progressive streaming property specifies whether the JDBC driver uses progressive streaming when progressive streaming is supported on the database server. This is the case for DB2 9 but not V8. The value of the `streamBufferSize` parameter determines whether the data is materialized when it is returned.

Valid values are `DB2BaseDataSource.YES` (1) and `DB2BaseDataSource.NO` (2). If the `progressiveStreaming` property is not specified, the `progressiveStreaming` value is `DB2BaseDataSource.NOT_SET` (0).

If the connection is to a database server that supports progressive streaming, and the value of `progressiveStreaming` is `DB2BaseDataSource.YES` or `DB2BaseDataSource.NOT_SET`, the JDBC driver uses progressive streaming to return both LOBs and XML data. Therefore, by default progressive streaming is enabled.

If the value of `progressiveStreaming` is `DB2BaseDataSource.NO` or the database server does not support progressive streaming, the way in which the JDBC driver returns LOB or XML data depends on the value of the `fullyMaterializeLobData` property.

The `streamBufferSize` property specifies the size, in bytes, of the JDBC driver buffers for chunking LOB or XML data. The JDBC driver uses the `streamBufferSize` value whether or not it uses progressive streaming. The default is 1 MB.

If the JDBC driver uses progressive streaming, LOB or XML data is materialized if it fits in the buffers and the driver does not use the `fullyMaterializeLobData` property.

With the change to progressive streaming, attention has to be paid to the changed behavior when using progressive references as compared to the use of locators with V8 `fullyMaterializeLobData=false`. The progressive reference is freed when the cursor is closed, while in V8, it was only freed at commit. Example 5-7 shows a snippet from the code to be adapted for use with progressive references.

Example 5-7 Insert using progressive reference (c) SAP AG; 2006

```
Clob c1 = null;
Blob b1 = null;
// retrieve values via progressive reference
// assuming length of FLOB and FBLOB is greater than streamBufferSize
String query =
    "select FCLOB, FBLOB from TSTLOBTAB where FCHAR4 = 'A' and FINT = 1";

Statement stmt = getConnection().createStatement();
ResultSet rs = stmt.executeQuery(query);

if (rs.next()) {
    c1 = rs.getClob(1);
    b1 = rs.getBlob(2);
}

rs.close(); // this will not work in DB2 9 with
           // Progressive Streaming
```

(c) SAP AG; 2006

```

// DB2 will issue SQLCode -423

// update the second row with the LOB field values of the 1st row
String updStmt =
    "update TSTLOBTAB set FCLOB = ?, FBLOB = ? where FCHAR4 = 'B' and FINT = 2";
ps = getConnection().prepareStatement(updStmt);
ps.setClob(1, cl);
ps.setBlob(2, bl);

ps.executeUpdate();
ps.close();
// rs.close(); // DB2 9: close ResultSet here!

```

5.6 Data Dictionary considerations

In this section, we give you an overview about how LOB tables and their auxiliary objects are created and named. The naming conventions outlined here have been chosen by SAP and are enforced by its database interface layer, out of direct client control. These standards are not required by non-SAP applications, but are a good example to follow, helped by DB2 V8 allowing long names.

5.6.1 ABAP stack

The following database objects are created for each LOB column and partition on the ABAP stack if running on DB2 V8:

- ▶ The *LOB table space* is created in the base table's database with the name L[TABNAME5][LK]
 - Where:
 - [TABNAME5] = first 5 characters of the table's name
 - [LK] = 2 random characters ([A-Z] [0-9])
- ▶ The default storage attributes are as follows:
 - STOGROUP = table's default data STOGROUP
 - PRIQTY 200 SECQTY 10240 GBPCACHE SYSTEM
 - BUFFERPOOL BP40 LOG YES LOCKMAX 1000000 LOCKSIZE LOB
- ▶ For the *auxiliary table*, the naming convention is #[COLNAME14][MNO]
 - Where:
 - [COLNAME14] = first 14 characters of the column's name
 - [MNO] = 3 random characters ([A-Z][0-9])
- ▶ The *index on auxiliary table* is created with the same name as the auxiliary table using the following storage parameters:
 - STOGROUP = table's default index STOGROUP
 - PRIQTY 16 SECQTY 10240 FREEPAGE 10 PCTFREE 10 GBPCACHE CHANGED
 - BUFFERPOOL BP40 PIECESIZE 2097152 K

For example, if the non-partitioned table LOBTSTTAB has a LOB column named TESTLOBCOL, the naming of the related LOB objects could be:

- ▶ LOB table space LLOBTS5A
- ▶ Auxiliary table #TESTLOBCOL8ZH with index #TESTLOBCOL8ZH

(c) SAP AG; 2006

LOB fields are always created with a length of 1 GB, because this is the maximum length which can be logged in V8. With DB2 9, this length is increased to the maximum size of a LOB, 2 GB-1 bytes, and the SAP Data Dictionary is adapted accordingly.

The LOB table spaces are always created with BP40. This buffer pool allocation is again an SAP specific choice that is part of a broader buffer pool management scheme. It does, however, demonstrate a broad recommendation to separate LOB objects into distinct separate buffer pools because of their different attributes. More information about buffer pool considerations can be found in 8.3, “Buffer pools and group buffer pools” on page 244.

5.6.2 Java stack

The following database objects are created for each LOB column and partition on the JAVA stack. Auxiliary objects to store the LOB data are created as they are in the ABAP stack with the following differences:

- ▶ The table name is always used without a namespace prefix.
- ▶ Primary and secondary quantities are not specified. DB2 Space Extend Management is used (sliding allocation).
- ▶ The column name is used without truncation.

5.6.3 DSNZPARMs for DB2 V8

In this section, we show the recommended SAP-related DSNZPARMs for an SAP system.

Table 5-5 shows the additional values for DB2 V8.

Table 5-5 DSNZPARMs for V8 (c) SAP AG; 2006

Parameter	Value	Remark
LOBVALA	1000000K	The size of the user storage for LOB values (in KB). The recommended value is 1 GB (1,000,000 KB).
LOBVALS	50000M	The size of the system storage for LOB values (in MB). The recommended value is 50 GB (50,000 MB).

5.6.4 ROWID

DB2 V8 has introduced the concept of transparent ROWID. Prior to V8, the ROWID was visible to INSERT or SELECT statements, which did not specify a field list. It can be advantageous to normalize SELECT and INSERT statements to statements without a field list, because this cuts down the number of statements against a table and thus reduces stress on statement cache. This technique can be only used in SAP Unicode systems because the non-Unicode tables might have been created on DB2 V7 and then migrated to DB2 V8.

Even with a transparent ROWID, the SAP data dictionary has to hide the ROWID field, because it is visible with catalog SELECTs. SAP in practice hides the visibility of this column from the ABAP programmers for the tables containing LOB columns. This is important for application portability across SAP systems using different underlying DBMSs.

(c) SAP AG; 2006

Sample DDL for CREATE TABLE statement

As an example, we create a very simple table with three columns and two LOB columns on the Java stack. The definition is given in Example 5-8 in XML format, as used by the Java data dictionary.

Example 5-8 Table definition for test table TSTLOBDDL in XML format (c) SAP AG; 2006

```
<?xml version="1.0"?>
<Dbtable name="TSTLOBDDL" creation-date="">
  <properties>
    <author> </author>
    <description language=""></description>
  </properties>
  <predefined-action></predefined-action>
  <position-is-relevant></position-is-relevant>
  <deployment-status></deployment-status>
  <columns>
    <column name = "F1">
      <position>1</position>
      <dd-type>string</dd-type>
      <java-sql-type>VARCHAR</java-sql-type>
      <length>10</length>
      <decimals>0</decimals>
      <is-not-null>true</is-not-null>
      <default-value> 1 </default-value>
    </column>
    <column name = "F2">
      <position>17</position>
      <dd-type>binary</dd-type>
      <java-sql-type>BLOB</java-sql-type>
      <length>0</length>
      <decimals>0</decimals>
      <is-not-null>false</is-not-null>
      <default-value></default-value>
    </column>
    <column name = "F3">
      <position>14</position>
      <dd-type>string</dd-type>
      <java-sql-type>CLOB</java-sql-type>
      <length>1334</length>
      <decimals>0</decimals>
      <is-not-null>false</is-not-null>
      <default-value></default-value>
    </column>
  </columns>
  <primary-key>
    <tablename>TSTLOBDDL</tablename>
    <columns>
      <column>F1</column>
    </columns>
  </primary-key>
</Dbtable>
```

With DB2 V8, this results in the DDL shown in Example 5-9.

(c) SAP AG; 2006

```
SET CURRENT RULES = 'DB2';
CREATE TABLESPACE "TSTLOBD" IN "JDOXXTPB"
USING STOGROUP SAPR3L FREEPAGE 20 PCTFREE 16
GBPCACHE CHANGED DEFINE YES BUFFERPOOL BP2
LOCKSIZE ROW LOCKMAX 1000000 CLOSE YES
COMPRESS YES MAXROWS 255 SEGSIZE 20 CCSID UNICODE;
COMMIT;
CREATE TABLE "TSTLOBDDL" (
"F1" VARGRAPHIC(10) DEFAULT ' 1' NOT NULL,
"F2" BLOB (1G) ,
"F3" DBCLOB (500M) );
IN JDOXXTPB.TSTLOBD CCSID UNICODE;
COMMIT;
CREATE LOB TABLESPACE LTSTLOUX IN JDOXXTPB
USING STOGROUP SAPR3L LOG YES LOCKMAX 0 GBPCACHE
SYSTEM LOCKSIZE LOB DEFINE YES BUFFERPOOL BP40;
COMMIT;
CREATE AUX TABLE "#F2UX6" IN JDOXXTPB.LTSTLOUX
STORES "TSTLOBDDL" COLUMN "F2" CREATE INDEX "#F2UX6" ON "#F2UX6"
USING STOGROUP SAPR3L FREEPAGE 10 PCTFREE 10 GBPCACHE
CHANGED PIECESIZE 2097152 K DEFINE YES BUFFERPOOL BP40;
COMMIT;
CREATE LOB TABLESPACE LTSTLO1W IN JDOXXTPB USING STOGROUP SAPR3L
LOG YES LOCKMAX 0 GBPCACHE SYSTEM LOCKSIZE LOB DEFINE YES BUFFERPOOL BP40;
COMMIT;
CREATE AUX TABLE "#F31W0" IN JDOXXTPB.LTSTLO1W STORES "TSTLOBDDL" COLUMN "F3";
CREATE INDEX "#F31W0" ON "#F31W0" USING STOGROUP SAPR3L FREEPAGE 10
PCTFREE 10 GBPCACHE CHANGED PIECESIZE 2097152 K DEFINE YES BUFFERPOOL BP40;
COMMIT;
CREATE UNIQUE INDEX "#TSTLOBDDL8P" ON "TSTLOBDDL" ( "F1" ASC )
NOT PADDED USING STOGROUP SAPR3L
FREEPAGE 20 PCTFREE 16 GBPCACHE CHANGED
DEFINE YES CLUSTER BUFFERPOOL BP2
CLOSE YES DEFER NO COPY YES PIECESIZE 2097152 K;
COMMIT
ALTER TABLE "TSTLOBDDL" ADD PRIMARY KEY ( F1 );
COMMIT;
```

Note that SAP tends to compress most data, but in the statement in Example 5-9 on page 146, COMPRESS YES only works for the base table space, because as of DB2 9, LOB tables cannot be compressed. The table space names and database names have to be calculated and checked for possible namespace collisions by the dictionary interface. In this case, this is performed automatically as part of the database interface layer, but non-SAP applications need to consider their naming conventions to similarly ensure no overlapping in the name spaces of objects.

With DB2 9, this DDL simplifies considerably, as shown in Example 5-10.

```
CREATE TABLE "TSTLOBDDL"
("F1" VARGRAPHIC(10) DEFAULT ' 1' NOT NULL,
"F2" BLOB (1G) ,
"F3" DBCLOB (500M) )
CCSID UNICODE;
```



```
COMMIT;  
ALTER TABLE "TSTLOBDDL" ADD PRIMARY KEY ( F1 );  
COMMIT;
```

The new DB2 9 feature of implicitly created DB objects is a tremendous help when porting applications. This feature is described in 3.1.1, “Example of automatic creation of objects” on page 24. This feature will be used by SAP and will remove the need for special code in the database interface that previously handled this task. Some applications simply deliver SQL scripts, which contain the necessary DDL and which run on other DBMSs, such as DB2 for Linux, UNIX and Windows. With this new feature, these scripts run on DB2 for z/OS without modification. Differently from DB2 V8, tables can be assigned by default to databases other than DBSNDB04 and have attributes such as row locking instead of page locking.

What's missing

What is still missing is just the feature to enable ALTER to hide ROWID, that is, the capability to change a non-transparent ROWID into a transparent one. This would allow tables within an SAP system, with LOBs created prior to the transparent ROWID, to be easily *converted* to take advantage of the new feature.

5.7 Unicode

There are some differences with respect to LOBs for Unicode and non-Unicode systems.

Because SAP uses UTF16 as Unicode code page, character fields are stored in VARGRAPHIC instead VARCHAR, and CLOB data is stored in DBCLOB fields. As a consequence of this, the Bind for DB2 Connect CLI must use a different collection ID for non-Unicode and Unicode systems. See Example 5-11.

Example 5-11 Bind command for UNIX system (c) SAP AG; 2006

```
db2 bind <Instance directory>/sqllib/bnd/@ddcmvs.lst ACTION REPLACE  
KEEPDYNAMIC YES GENERIC \"DEFER PREPARE\" REOPT ONCE COLLECTION  
SAP<FIXPAK><U> ISOLATION UR BLOCKING UNAMBIG RELEASE COMMIT  
SQLERROR CONTINUE DYNAMICRULES RUN <ENCODING UNICODE>
```

All locator statements are bound regardless of the Encoding Bind Option with DB2 Connect 9.1 SP1. The same Collection ID can be used for Unicode and non-Unicode systems.

Another important point is that long character fields that are naturally mapped to VARCHAR in a non-Unicode environment cannot be mapped to VARGRAPHIC in a Unicode environment if their character length exceeds the threshold of 16 KB (since their byte length then exceeds the 32 KB boundary). DB2 does not support buffer pools greater than 32 KB; therefore, they have to be mapped to DBCLOB fields. As a consequence, there are a greater number of LOB fields in a Unicode environment.

Compression

SAP recommends to use compression for Unicode systems. The potential of database compression for a Unicode system is considerably higher than for a non-Unicode system. The reason is that data stored in UTF16 still has a high percentage of data mapped to ASCII; this means that the first two bytes of the UTF16 character contain mostly x'00'. Currently, compression is not available for LOB columns.

(c) SAP AG; 2006

If the content of DBCLOB columns keeps increasing, the requirement for compression for LOB fields becomes more urgent.

5.8 Some points of SAP LOB usage with CCMS

With the SAP Computing Center Management System (CCMS), some special considerations have to be made for LOB table spaces. Here, we distinguish between DB2 V8 and DB2 9.

With DB2 V8, an online REORG is not available for LOB table spaces; therefore in V8, REORG SHRLEVEL NONE must be defined for LOBs.

Jobs for LOB-related REORG should be uploaded to z/OS and only submitted during a maintenance window, when the SAP system is down. Otherwise, this would result in a severe performance impact on the running SAP system.

For this reason, suggested periodic REORG jobs of table spaces do not include LOB table spaces.

Additionally, a V8 REORG of a LOB table space did not reclaim disk space in the underlying VSAM data set, and as a result, the number of extents remained unchanged. SAP environments have typically a large number of data sets and it is common to use the number of extents as a trigger for the REORG of table spaces. Primarily, this trigger is to minimize the risk of exhausting the maximum number of extents unexpectedly. The availability of sliding secondary extent allocation in DB2 V8 removed most of the potential for problems, and this is the default for implicitly created table spaces in V9. Changes in architecture over a period of time have rendered the performance impact of a growing number of extents to be marginal, or non-existent. A growing number of extents might, however, be an indication of performance ramifications of potential internal disorganization.

LOB table spaces, therefore, introduced a complication in the sense that a REORG did not remove the trigger of a large number of extents, while it did restore the order of the records to optimal. Solutions to this problem include scheduling time to make the table space unavailable and performing an offline RECOVER as a more brutal method of resizing the data set. Another approach is to ignore the REORG trigger condition, provided the object is not at risk of reaching maximum extents, and other statistics such as ORGRATIO do not indicate that internal disorganization accompanies the high number of extents. Another circumvention is the *sliding allocation technique* introduced by MGEXTSZ (with a global scope), option 7 on install panel DSNTIPZ in DB2 V8.

With DB2 9 for z/OS, REORG SHRLEVEL REFERENCE is available and the restriction is removed. The underlying data sets are recreated as a shadow, reflecting any ALTERs to primary and secondary quantities for the object. While SHRLEVEL REFERENCE is not completely disruptive, with the nature of the data presently stored in tables with LOBs in the SAP system, it should be possible to non-disruptively schedule maintenance activities in a normal environment.

Also, the REORG of the auxiliary index can be performed online. For critical objects contained in LOBs in the SAP system, this is a great improvement. For more information, see 6.9, “REORG” on page 185.

5.9 Portability aspects

If an application or program is to run against different DBMSs, as SAP does, portability is a major concern. As an example, some applications might just provide a script to create tables and indexes and know nothing about the fact that DB2 for z/OS requires table spaces, databases, and auxiliary objects for LOBs. We note that this issue is solved with DB2 9 and automatic creation of DB objects.

A typical problem is that a simple SQL statement “SELECT *” on a table with a ROWID in DB2 for z/OS Version 7 also retrieves the ROWID. Because other database vendors might not support the concept of a ROWID or might not require LOB tables to contain a ROWID column, this poses a challenge for the application.

Even with the transparent ROWID of DB2 V8, the SAP catalog reader tools have to hide the ROWID field, because it is visible with Catalog Selects but is unknown to the SAP Dictionary.

Another aspect is that SAP also uses many small LOBs. Because DB2 stores the LOB data in auxiliary tables, which is attractive for huge LOBs, it can yield a performance penalty if small LOBs are massively accessed.

Table 5-6 shows a comparison of different databases in relation to LOBs.

Table 5-6 Comparison of different databases for aspects of LOBs (c) SAP AG; 2006

	DB2 for z/OS	DB2 for LUW	DB2 for iSeries™	Others
ROWID column required?	Yes	No	No	No
ROWID column supported?	Yes	No	No	Mostly no
Auxiliary table spaces for LOBs?	Must	Can	No	Some can
Read piece-wise supported?	Yes	Yes	Yes	Yes
Read in one piece supported?	Yes	Yes	Yes	Most
Validity of locator?	Until commit	Until commit	Until commit	Some until close cursor
Maximum number of LOB fields per table?	> 750	No restriction	No restriction	Some only few
Search function on LOBs supported?	Yes	Yes	Yes	Some
Concatenation of LOBs supported?	Yes	Yes	Yes	Mostly no
JDBC driver supports materialization of LOBs on client side?	Yes	Yes	Yes	Mostly yes
Can materialization be switched on and off?	Yes	Yes	Yes	Some yes

5.10 Monitoring and tracing

There are a variety of traces built into the SAP system to enable the user to trace the system activity. Traces include the ST05 trace, the dbsl trace, the CLI trace, and the IFI DB trace. All of these are available within transaction DB2. These traces can, of course, also be used to trace the statement flow on LOB tables.

DBSL trace

With dbsl trace level 3, all displayed data is truncated at 256 bytes. With dbsl trace 4, all LOB data is displayed. Go to transaction **DB2** → **Traces/Logs** → **DBSL** to start the trace. See Figure 5-10 on page 151.

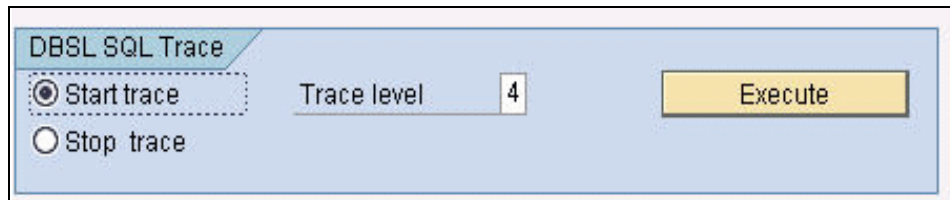


Figure 5-10 Starting the dbsl trace (c) SAP AG; 2006

All data is traced to the corresponding developer traces. This feature is particularly helpful if corrupted LOB data is suspected.

Example 5-12 shows how a BLOB looks.

Example 5-12 DBSL trace displaying all LOB data (c) SAP AG; 2006

```

C DB2TRC: 0000000072 00 0015 000000 CLI_EXTENDED_PREPARE (18) SELECT "UNAM" , "UDAT" ,
"UTIME" , "L_DATALG" , "Q_DATALG" , "SDAT" , "STIME" , "MINOR_VERS" , "MAJOR_VERS" FROM
"REPOLOAD" WHERE "PROGNAME" = ? AND "R3STATE" = ? AND "MACH" = ? FOR FETCH ONLY WITH UR
C DB2TRC: 0000000007 00 0015 000000 CLI_EXTENDED_BIND IN 65552
C DB2TRC: 0000000004 00 0015 000000 CLI_EXTENDED_BIND OUT 65552
C DB2TRC: 0000008182 00 0015 000000 CLI_EXECUTE 65552 cursor_hold = off
C DB2TRC:
CLIDA-IN BEGIN
C DB2TRC:
CLIDA-IN PARAMETER 3
C DB2TRC:
CLIDA-IN [0] (SQL_VARCHAR,8,40):SAPMSSY6
C DB2TRC:
CLIDA-IN [1] (SQL_VARCHAR,1,1):A
C DB2TRC:
CLIDA-IN [2] (SQL_SMALLINT,2,2):324
C DB2TRC:
CLIDA-IN END
C DB2TRC: 0000000009 00 0015 000000 CLI_FETCH 1/1 65552
C DB2TRC:
CLIDA-OUT BEGIN
C DB2TRC:
CLIDA-OUT PARAMETER 9
C DB2TRC:
CLIDA-OUT [0] (SQL_VARCHAR,1,12):
C DB2TRC:
CLIDA-OUT [1] (SQL_VARCHAR,8,8):20060728
C DB2TRC:
CLIDA-OUT [2] (SQL_VARCHAR,6,6):064459
C DB2TRC:
CLIDA-OUT [3] (SQL_INTEGER,4,4):16124
C DB2TRC:
CLIDA-OUT [4] (SQL_INTEGER,4,4):6397
C DB2TRC:
CLIDA-OUT [5] (SQL_VARCHAR,8,8):20060417
C DB2TRC:
CLIDA-OUT [6] (SQL_VARCHAR,6,6):062251
C DB2TRC:
CLIDA-OUT [7] (SQL_SMALLINT,2,2):1
C DB2TRC:
CLIDA-OUT [8] (SQL_INTEGER,4,4):1623
C DB2TRC:
CLIDA-OUT END
C DB2TRC: 0000000006 00 0015 000000 CLI_CLOSE_CURSOR 65552
C DB2TRC: 0000000008 00 0016 000000 CLI_ALLOC_STMT 65553
C DB2TRC: 0000000004 00 0016 000000 CLI_SET_ROWS_FETCHED_VAR 65553
C DB2TRC: 00 0016 000000 SQLESETI 65553
C DB2TRC: (
)
C DB2TRC: 0000000050 00 0016 000000 CLI_EXTENDED_PREPARE (18) SELECT length("LDATA")
,"LDATA" , "LDATA" FROM "REPOLOAD" WHERE "PROGNAME" = ? AND "R3STATE" = ? AND "MACH" = ?
FOR FETCH ONLY WITH CS
C DB2TRC: 0000000005 00 0016 000000 CLI_EXTENDED_BIND IN 65553
C DB2TRC: 0000000003 00 0016 000000 CLI_EXTENDED_BIND OUT 65553
C DB2TRC: 0000008069 00 0016 000000 CLI_EXECUTE 65553 cursor_hold = off
C DB2TRC:
CLIDA-IN BEGIN
C DB2TRC:
CLIDA-IN PARAMETER 3
C DB2TRC:
CLIDA-IN [0] (SQL_VARCHAR,8,40):SAPMSSY6
C DB2TRC:
CLIDA-IN [1] (SQL_VARCHAR,1,1):A
C DB2TRC:
CLIDA-IN [2] (SQL_SMALLINT,2,2):324
C DB2TRC:
CLIDA-IN END
C DB2TRC: 0000003566 00 0016 000000 CLI_FETCH 1/1 65553

```

(c) SAP AG; 2006

```

C DB2TRC: CLIDA-OUT BEGIN
C DB2TRC: CLIDA-OUT PARAMETER 3
C DB2TRC: CLIDA-OUT [0] (SQL_INTEGER,4,4):16124
C DB2TRC: CLIDA-OUT [1] (SQL_BLOB_LOCATOR,4,4):AF1B059B
C DB2TRC: CLIDA-OUT
[2] (SQL_BLOB,16124,64000):FF0AD10000121F9D02B4EF25005215D7DAA76EDFEE59617A60403671D851417A8
64540C0E999E9611A7A669AEE1E04151B04141790006E7149AB68E29A7125BB9868824B12624C4C8C896D342E31
2F92C5C42426E16912CD62C4EC6A0CFF3955A7EEAD7BA7866062DECBFBD3B7097F37DB59E3A75AAEE9DDDB49800
71ED847
C DB2TRC:
A77C32DB95CFAF9CD3F877A43991989398D57474624E73F3EC261000B1E3E0C821E96557C7D7DEF5A3D641CE5F3
61E07F0AD6AD0E2342F41BD0520DA88DA7C80481660549F2287DE82A749788C02A89B8A57CE278A74C2B2E7C2C0
B2409EE3300DA2F011BC136213E6111E2F8CF301449C1302BE0E70FA4E6CF86A8077E371
C DB2TRC:
171EBB7603DC99C563B57184F53E907205E21734025C82D85D253CCA009FDF0DF0A9B8E2DBF6E27D1FC071C8ADC
4EB86B2BADEC9C770E45B116B2BABB474505ADB713BF8F9CC83DA4CF585F1293341DA5AC9FEFDF4E4A6540F3A4
58CCF7F302C508D1E4A512CAD286B5451F44D45D8A4EBB1CA0C1AA5F5852E0AEE11B9B61
C DB2TRC:
080C455DE0E17C46C1887FFB34C41BFAE3575F89F8B0FEB87B2AE2C3FBE3DB06217E487FFCF91F213EA23F5EF73
8E223FBE1B1CFDC0187C1A18C8F256C2180BB505DBD033DF62F6BF15A8A4004AF4EB91AE2559568906DE01E0F64
1944FEA6CABC2302434B682A898B727589CDB608A04638220251E14A1D6D26A265CFA852
C DB2TRC:
8F95D9ACAACE680394C0151B8A50861DF087577780A8B8876C3D4A54EE8843F41ECA3D1AB94B60CC8E4B41ECC04
11F0EF04351856539307A0ED225072AF03E3A9BEE232272295E5D2A7B0A968DD3EC4ED46354CE0858307B3F7C5B
548B9A28B94B23A4A231BC4E85EEE860BC1E05BD77623A2CB35AD4C2EB220A115D26636F
C DB2TRC:
603D3BB08C2DE50A88C96B254477C01F9FC4760FC2F656C8F68E2DD7CAEB61E541F2DA581E4CED46FBEC108343F
6A9C39258A41E2F47037A7DD9736CA3DFE76CC0123E80DD2C41C9D5FC07A80FA6DEE2B8E5C1E5B8CF3B58A2AFA7
22CE27CAD172CCD3B166D2CB754A8F4461C61CF2857A3104EB241B407948C9A5FB89E07C
C DB2TRC:
BC3CB41C17D1FDBF4FC575094AC856E50631147B397607CCAB47DB34C8316D2C0F431B450618CBA1689F941816B
2CF7071882E57EA23C4C8803E4A7B1F008E078DD19BAED8FFCD68AC44069F2ADB1929BB2206B11BA01C990F6577
7E2D46F28A7215565211C084183DBF46715595983F018DD43E4CB95B08EEB35316D8D686
C DB2TRC:
B7A9AD7FC5CC11BE7F5DD7297D1A7DCEB333D51905ED837FC5BE44F45C5569A5FF57E07C2DA14F4F459F2EA14FC
FE572DFB8CF57D3AE28083AA8B9C23C68808DE0FD7BEAD399AFF620CCCC0FB3A5D67884BE07D25736F321761AE
E95D15C69C5B0023643B6A44B59C7329689473AE1BDB49FEDA0B47D19CA3F694015EDB01
C DB2TRC:
67ED467F3954C70079A5F91FF29BEB95196183538E7E85EF6D71ED206392A375E93773D86F9C7284FDC6994F7EA
2FCC665BFF131DF6F5CB67954DB57CE19B7E460EDF25E54971C87EF118F203EEC207D696C40B7F8921E1BE9B358
12FAAC5BE196E45603D598C2A7D4515B143EA5AED2455F19030BFEABC60BC724628E4F1F
C DB2TRC:
4015AE4BECE25209D421220B4B65984CF607E9D783EE70E1E812A65D81ED298168C40D938C9198B73A98B7846B5
CC931EAEAE70F14FCBA264D68B79CAC252EFE03BA2309FAC8CD7634A406B29D62FA89E40DD9CBFE4E01119A89F
35980F17D11546BDDC66CAFBAFEBAFAE57F6DBE9D76FC752FFE8B7D06F47F79BF28B7118
.....
.....
C DB2TRC: CLIDA-OUT END
C DB2TRC: 0000000812 00 0016 000000 CLI_CLOSE_CURSOR 65553

```

Examples for the DB2 Connect CLI traces are given in Example 5-1 on page 135 and in Figure 5-6 on page 133 for the ST05 trace.

We now provide an example of the IFI DB Trace using the SAP Performance Optimizer. See Figure 5-11 on page 153. You can download the SAP Performance Optimizer tool from SAP note 908770.

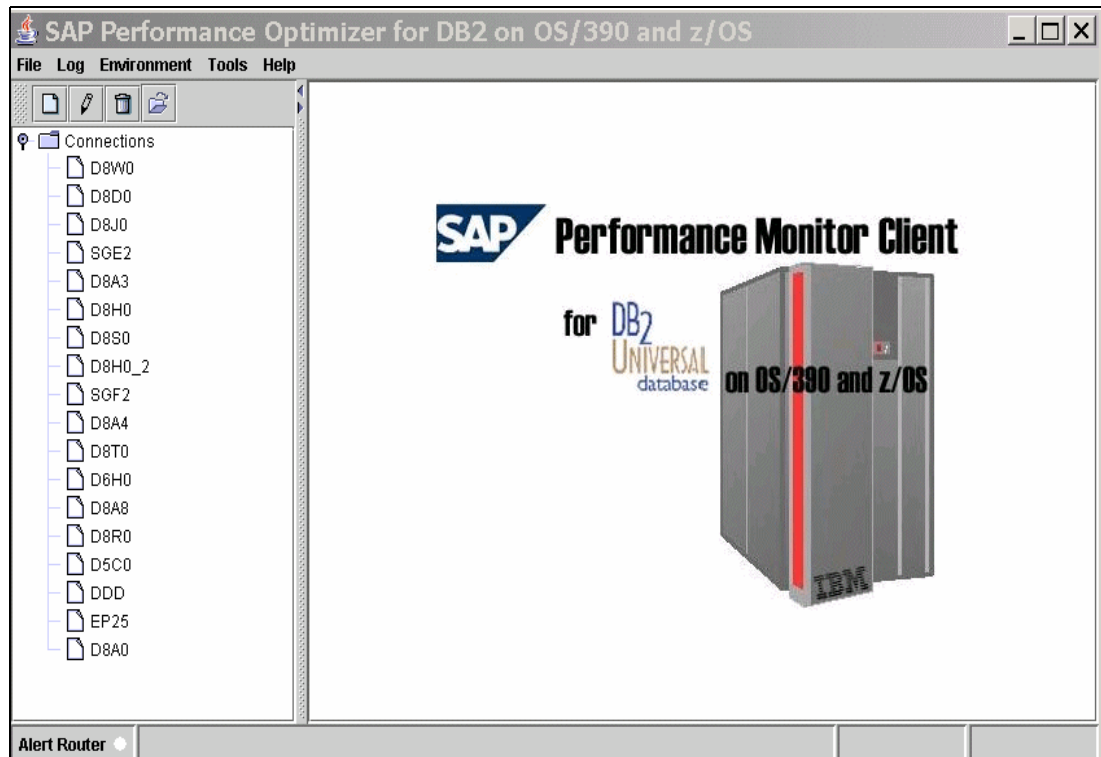


Figure 5-11 SAP Performance Optimizer Tool (c) SAP AG; 2006

To take traces, go to **Monitor Views** → **DB Trace**, as shown in Figure 5-12.

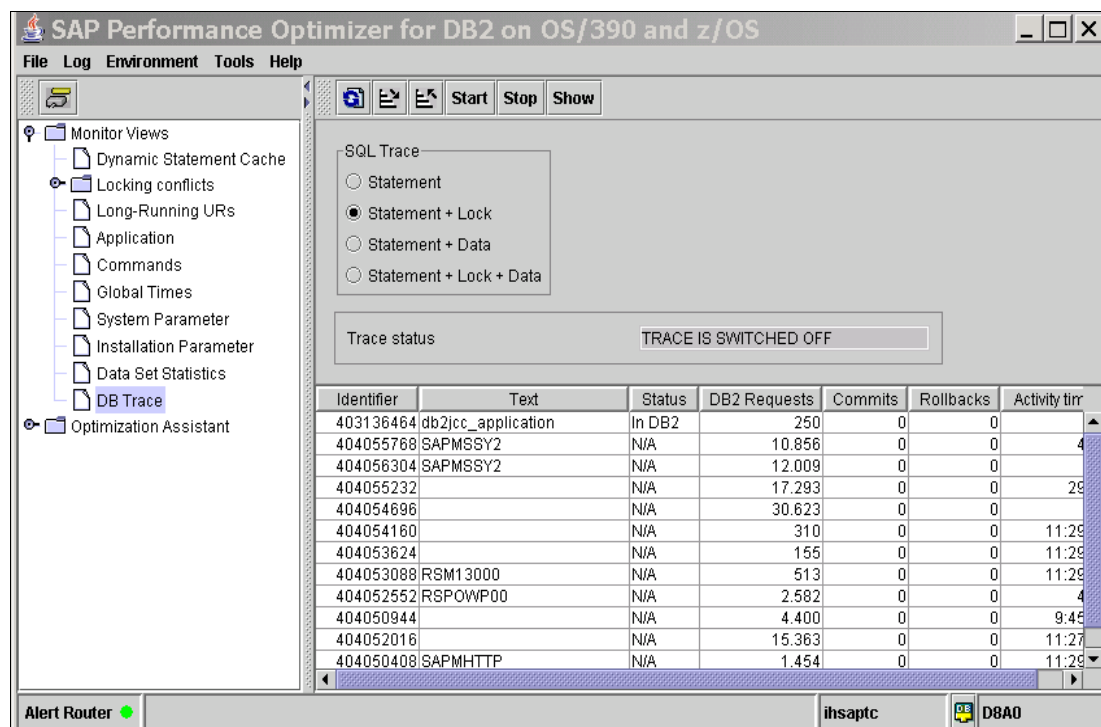


Figure 5-12 SAP Performance Optimizer Tool DB Trace (c) SAP AG; 2006

In Example 5-13 on page 154, we show the locking behavior for DB2 V8. (c) SAP AG; 2006

Example 5-13 Locks taken by simple *SELECT* on LOB table V8 (c) SAP AG; 2006

023921:646261	3.594	00000	00016448	PREPARE	CURSOR	SQL_CURLH200C2		
023921:646361		00004	00000000	CHANGE L	ANY	MANUAL+1 Data Page		
023921:646734				BIND		select FCLOB, FBLOB from TSTLOBTAB where FCHAR4 = ? and FINT = ?		
023921:647685		00000	7DC5DAE0	LOCK	S	MANUAL Data Page	ID=9	DSNDB06
023921:647815		00004	7DC5DAE0	LOCK	S	MANUAL Data Page	ID=9	DSNDB06
023921:647953		00004	7DC5DAE0	LOCK	S	MANUAL Data Page	ID=9	DSNDB06
023921:648520		00004	7DC5DAE0	LOCK	S	MANUAL Data Page	ID=9	DSNDB06
023921:649644		00004	00000000	UNLOCK	ANY	MANUAL Data Page		
023921:649820	35	00000	00016448	OPEN	CURSOR	SQL_CURLH200C2	CS	
023921:875842	382	00000	00016448	FETCH		SQL_CURLH200C2		
023921:876030		00004	7DC55210	LOCK	S	MANUAL Data Page	ID=117	DSNDB04
023921:876101		00004	7F3686B0	LOCK	S	COMMIT+1 LOB value	L9KBRBOM	DSNDB04
023921:876184		00004	7F3686B0	LOCK	S	COMMIT+1 Data Page		
023922:079864		00000	7DC5DAE0	LOCK	S	COMMIT SKPT		
023922:080889	2.685	00000	00016448	PREPARE	CURSOR	SQL_CURLN200C1		
023922:081011				BIND		SELECT LENGTH (CAST(? as DBCLOB)) FROM SYSIBM.SYSDUMMYU		

Example 5-14 shows the locking behavior for DB2 9. It demonstrates that LOB locks are no longer taken.

Example 5-14 Locks taken by simple *SELECT* on LOB table with DB2 9 (c) SAP AG; 2006

024656:730414				BIND		select FCLOB, FBLOB from TSTLOBTAB where FCHAR4 = ? and FINT = ?		
024656:731023		00004	7F5BFF80	LOCK	S	MANUAL Data Page	ID=9	DSNDB06
024656:731114		00004	7F5BFF80	LOCK	S	MANUAL Data Page	ID=9	DSNDB06
024656:731222		00004	7F5BFF80	LOCK	S	MANUAL Data Page	ID=9	DSNDB06
024656:731726		00004	7F5BFF80	LOCK	S	MANUAL Data Page	ID=9	DSNDB06
024656:732879		00004	00000000	UNLOCK	ANY	MANUAL Data Page		
024656:733073	50	00000	00016448	OPEN	CURSOR	SQL_CURLH200C2	CS	
024656:960803	277	00000	00016448	FETCH		SQL_CURLH200C2		
C0 LOCK	S	MANUAL	Data Row			TMPROSQL	DSN02348	
024657:165879		00004	00000000	CHANGE L	ANY	MANUAL+1 Data Page		
024657:165949		00000	7F5BDB80	LOCK	IS	MANUAL+1 Page Set	SYSPKAGE	DSNDB06
024657:165975		00000	7F5BF340	LOCK	IS	MANUAL+1 Table	SYSPACKAUTH	
024657:166163		00000	7F5C3190	LOCK	S	MANUAL Data Page	ID=143	DSNDB06
024657:166205		00000	7F5C3190	UNLOCK	S	COMMIT+1 Data Page		
024657:166243		00004	00000000	UNLOCK	ANY	MANUAL Data Page		
024657:166257		00000	00000000	UNLOCK	IS	MANUAL+1 Table	SYSPACKAUTH	
024657:166268		00000	00000000	UNLOCK	IS	MANUAL+1 Page Set	SYSPKAGE	DSNDB06
024657:166337		00000	7F5BDB80	LOCK	IS	COMMIT Page Set	ID=127	ID=1
024657:166351		00000	7F5BF340	LOCK	IS	COMMIT Table	ID=129	
024657:191520		00000	7F5C3190	LOCK	S	MANUAL Data Page	ID=129	ID=1
024657:202313		00000	7F5B9EB0	LOCK	S	COMMIT SKPT		
0								
024657:257137		00000	7F5C3190	UNLOCK	S	MANUAL+1 Data Page		
024657:257152		00000	7F5C3190	LOCK	S	MANUAL Data Page	ID=129	ID=1
024657:257698		00000	7F5C3190	UNLOCK	S	COMMIT+1 Data Page		
024657:258542			00016448	PREPARE	CURSOR	SQL_CURLN200C1		
024657:258673				BIND		SELECT LENGTH (CAST(? as DBCLOB)) FROM SYSIBM.SYSDUMMYU		

5.11 Database interface layer profile parameters

In this section, we describe `dbsl_lib` profile parameters, which refer to LOBs. A summary of the parameters is provided in Table 5-7 on page 155.

Table 5-7 *dbsl_lib* profile parameters for LOB handling (c) SAP AG; 2006

Parameter	Recommended value	Description
db2/db2/lob_buf_size db2_db2_lob_buf_size	64,000 bytes	Size of the local LOB buffer. Minimum value is 32,000 bytes, maximum 2,000,000 bytes.
db2/db2/use_eda db2_db2_use_eda	1	Possible values are [0,1]. Use 0 to turn off the extended da feature.
db2/db2/use_drda_lob_handling db2_db2_use_drda_lob_handling	0	Possible values are [0,1]. Use 1 to turn on the DB2 Connect CLI LOB streaming. Not supported with DB2 V8.
db2/db2/chaining db2_db2_chaining	20	Possible values [0,MAX_INT]. Use 0 to turn off chaining. For values greater than 0, this number gives the minimum value of rows which have to fit into the db2_lib array buffer before chaining is used.
db2/db2/lob_free_buffer db2_db2_lob_free_buffer	1,000	Possible values [0,MAX_INT]. Number of locators to fit into free locator buffer. Set to 0 to turn off the free locator buffer.
db2/db2/max_lob_free_length db2_db2_max_lob_free_length	1 GB	Possible values [0,2048]. Threshold for storage which locators in locator free buffer can hold before freed.
db2/db2/sql_trace db2_db2_sql_trace	0	Possible values [0,4]. 0 corresponds to db2 trace off. 1 corresponds to db2 statement trace. 2 is unused. 3 corresponds to db2 statement and data trace, data is truncated. 4 corresponds to db2 statement and data trace, data is not truncated.
db2/db2/cli_trace_value db2_db2_cli_trace_value	0	Possible values [0,1]. Set to 1 to turn on CLI trace.
db2/db2/cli_trace_dir db2_db2_cli_trace_dir	/usr/sap/<SAPSID>/<INSTANCE>/work	Directory to which CLI traces should be written. If not set, no CLI traces are written.

5.12 Performance measurements

Some measurement results are included here from different sources.

5.12.1 Locks and SELECT

The IBM/SAP performance team used the SAP program R3load (a utility often used to export tables in SAP platform independent format) as a means to measure repeated workloads against LOBs in DB2 V8 and DB2 9 subsystems. The tool is used to export records from a table with a LOB column into a file and is executed on a remote network-connected client. The purpose here is to evaluate the impact of the new locking mechanisms for LOBs in DB2 9 showing improvements in the context of a real workload.

The measurement data shows the improvements with reduced locking on LOB objects in DB2 9.

(c) SAP AG; 2006

This improvement shows through in two distinct ways: first, in the CPU resource within DB2 processing, and secondly, the number of round-trips required between the remote client and DB2 was reduced. Both of these factors lead to reduced run time of the application task, and these improvements require no change to the application execution. It should be noted that this example uses an application and underlying table specifically chosen because it contains data predominantly in LOB columns, but nonetheless, it proves significant gains with DB2 9.

The first measurement shown in Figure 5-13 shows the reduction in elapsed time of the test job, from 5,193 seconds to 4,459 seconds, a total 14% reduction.

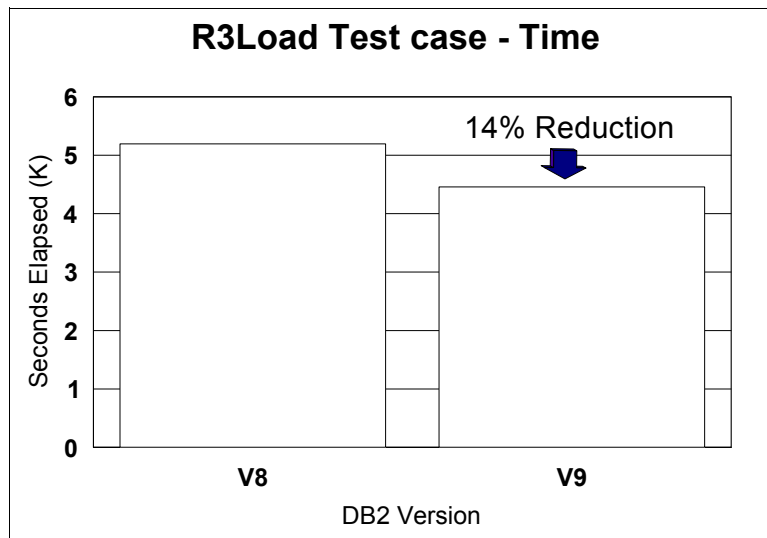


Figure 5-13 SAP R3load test case elapsed time (c) SAP AG; 2006

Figure 5-14 shows the dramatic reduction in lock requests. The reduction is comprised entirely of the elimination of LOB-lock requests for lock and unlock of the LOB row.

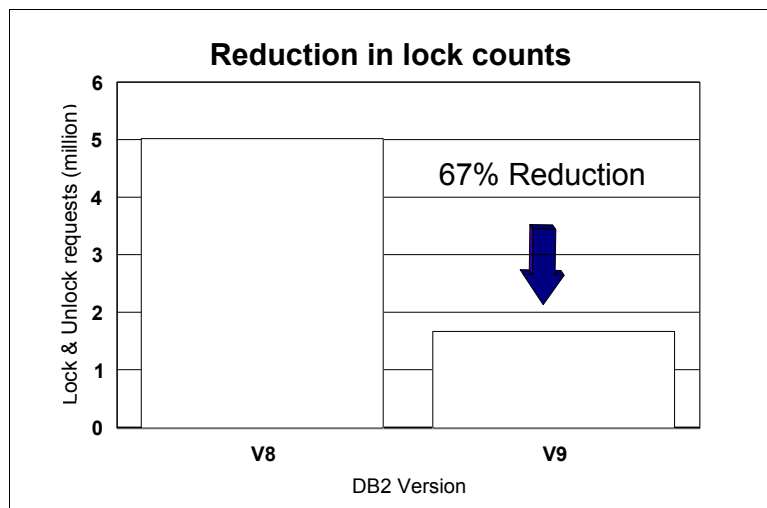


Figure 5-14 R3load test case reduced locks (c) SAP AG; 2006

What remains is the lock on the base table row as expected, so for each row processed, two of the three lock requests are eliminated. This demonstrates the significant improvement that this change makes, particularly in a DB2 data sharing environment. (c) SAP AG; 2006

The final measurement shown in Figure 5-15 demonstrates the reduction in network traffic between client and host running the R3load process.

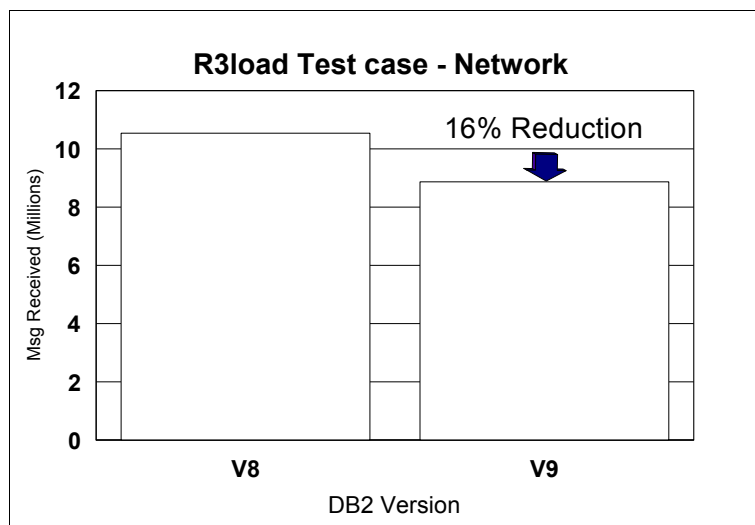


Figure 5-15 R3load test case reduced network round-trips (c) SAP AG; 2006

While the overall number of requests is still high, the 16% reduction in network round-trips actually contributes a significant portion of the elapsed run-time reduction. Again, this gain is entirely transparent to the user, and the gain comes entirely from the change to DB2 9.

5.12.2 Locks and INSERT

In case of LOB INSERT jobs, the changes in lock management with DB2 9 are meant to increase availability and reduce contention. In general, they are not expected to introduce significant changes in elapsed or CPU time in LOB insert in V9 over V8. DB2 9 is now executing Lock and Unlock for each INSERT, instead of Locks at INSERT and Unlocks only at commit in V8. V8 Unlock unlocks multiple resources while V9 Unlock unlocks one LOB at a time. There is some CPU increase in V9 as a result of issuing additional Unlock requests for the same number of resources unlocked, which is unchanged between V8 and V9. The advantage is that each resource is freed quickly, thereby, avoiding contention and LOB lock escalations.

5.12.3 UPDATE improvement

Figure 5-16 on page 158 shows data from another performance test case implemented by SAP development with APARs PK22887 and PK25241 applied to DB2 V8. These changes are being integrated in DB2 9 code and are mainly related to improve space management for mass update. Keep in mind that updates of LOBs are done using DELETE and INSERT. The data compares the time (in milliseconds) to update a single row when updating the reported groups of rows in thousands. Before maintenance, the elapsed time increased at a very high rate. After maintenance, it increases at a much slower rate.

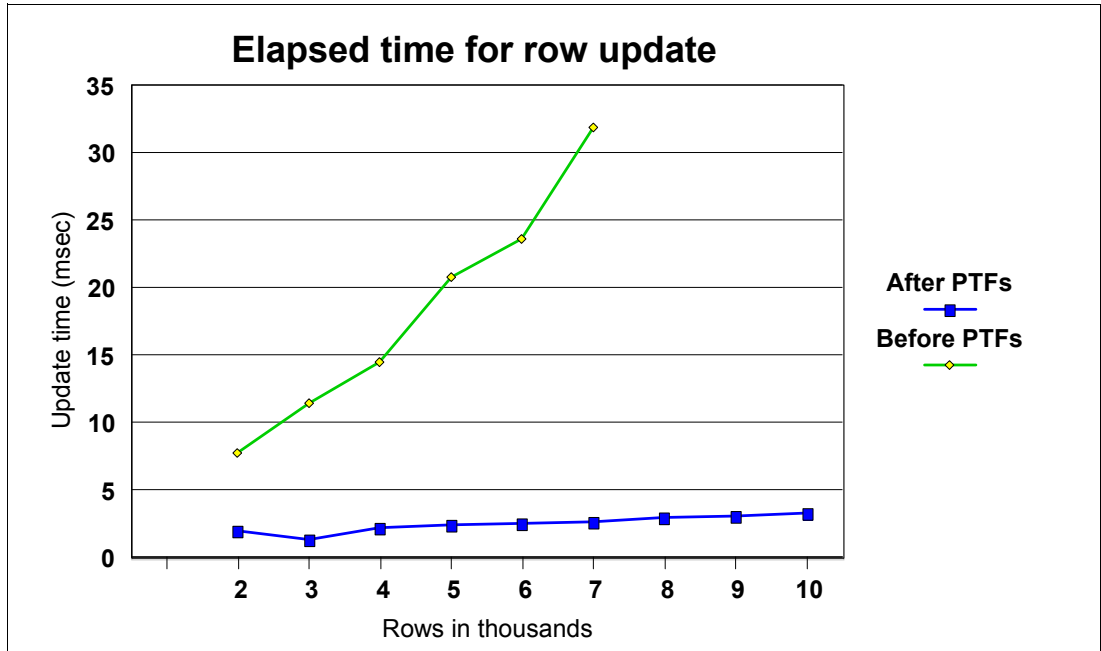


Figure 5-16 Time to update a row when increasing the numbers of updated rows (c) SAP AG; 2006



Utilities with LOBs

In this chapter, we discuss the way DB2 utilities have been enhanced to support table spaces containing tables with LOB columns. We go through the various DB2 utilities and document the specifics for LOB objects. DSN1IAUL is not a utility but is often used in administering DB2 data, so it has been added here for completeness.

The chapter contains information about the following utilities:

- ▶ UNLOAD
- ▶ DSN1IAUL
- ▶ LOAD
- ▶ COPY
- ▶ COPYTOCOPY
- ▶ QUIESCE
- ▶ REPORT
- ▶ RUNSTATS
- ▶ REORG
- ▶ RECOVER and REBUILD
- ▶ CHECK DATA
- ▶ CHECK LOB
- ▶ CHECK INDEX
- ▶ REPAIR
- ▶ DSN1COPY and DSN1PRNT

We only discuss those features that are directly related to LOBs.

6.1 UNLOAD

There are basically two ways to use the unload utility with LOB data:

- ▶ Unload LOB columns as normal data columns to the unload data set of the base table.
- ▶ Use file reference variables to unload each LOB to a separate file.

Unload LOB data as normal data columns

This method can be used if the sum of the lengths of all fields to be unloaded does not exceed 32 KB. Because the maximum record length of a sequential file in z/OS is 32 KB, and the Unload utility does not support an unload record to be spanned over multiple output records in the output data set, this method can only be used if the total record size of the data to be unloaded does not exceed 32 KB. The LOB fields are unloaded together with the other selected data fields to the output file with a maximum record length of 32 KB.

In most cases, this method is only used if the LOBs in the table are small LOBs (SLOBs) with a defined maximum length smaller than 32 KB or if the user deliberately decides to truncate the LOB data using the TRUNCATE keyword.

We demonstrate this with some examples based on table ##T.NORMEN00. The table is created with the DDL listed in Example 6-1.

Table ##T.NORMEN00 contains a BLOB column IMAGE. The table is used for storing scanned documents in formats such as TIFF, GIF, BMP, and PDF. In this example, we have explicitly specified the ROWID column. We use the DB2 9 syntax of the LOG keyword. The base table space is defined as LOGGED, and the LOB table space is defined as NOT LOGGED. All indexes are defined as COPY YES.

This is the same table we have used for illustrating the entries in the DB2 catalog (see Example 7-1 on page 212), but it is reported here again for convenience.

Example 6-1 DDL for table ##T.NORMEN00

```
CREATE DATABASE NORMEN00
  CCSID EBCDIC ;
CREATE TABLESPACE NORMEN00 IN NORMEN00
  USING STOGROUP PAOLOGS
    PRIQTY 100
    SECQTY 28
    ERASE NO
    LOGGED
    GBPCACHE CHANGED
    COMPRESS NO
    BUFFERPOOL BP1
    LOCKSIZE PAGE
    LOCKMAX 0
    CLOSE YES
    SEGSIZE 4
    CCSID EBCDIC
    MAXROWS 255 ;
CREATE TABLE ##T.NORMEN00
  (DOC_ID          VARCHAR(30)          FOR SBCS DATA NOT NULL
   ,PAGE_NUMBER    SMALLINT              NOT NULL
   ,IMPORTER       CHAR(8)              FOR SBCS DATA NOT NULL
   ,IMPORT_TIME    TIMESTAMP              NOT NULL
   ,IMAGE          BLOB)
```

```

        ,FORMAT          WITH DEFAULT
        ,ROW_ID          CHAR(8)          FOR SBCS DATA NOT NULL
        ,                ROWID          NOT NULL
        ,                GENERATED ALWAYS
        ,IMAGE           BLOB(2097152)
        ,                WITH DEFAULT NULL)
    IN NORMEN00.NORMEN00 ;
CREATE UNIQUE INDEX ##T.I_NORMEN00_1
    ON ##T.NORMEN00
    (DOC_ID          ASC,
     PAGE_NUMBER     ASC,
     FORMAT          ASC)
    USING STOGROUP PAOLOGS
    PRIQTY 12
    SECQTY 12
    ERASE NO
    GBPCACHE CHANGED
    CLUSTER
    BUFFERPOOL BP2
    CLOSE YES
    COPY YES
    PIECESIZE 2 G ;
CREATE LOB TABLESPACE NORMLOB IN NORMEN00
    USING STOGROUP PAOLOGS
    PRIQTY 20000
    SECQTY 5000
    ERASE NO
    GBPCACHE SYSTEM
    NOT LOGGED
    DSSIZE 4G
    BUFFERPOOL BP1
    LOCKSIZE LOB
    LOCKMAX 0
    CLOSE YES ;
CREATE UNIQUE INDEX ##T.I_NORMEN00_AUX
    ON ##T.NORMEN00_AUX
    USING STOGROUP PAOLOGS
    PRIQTY 52
    SECQTY 20
    ERASE NO
    GBPCACHE CHANGED
    BUFFERPOOL BP2
    CLOSE YES
    COPY YES
    PIECESIZE 2 G
    DEFINE YES
;

```

In the first case, we try to unload the complete table into a sequential output file using the utility statement shown in Example 6-2.

Example 6-2 Unload LOB data as normal data columns with UNLOAD TABLESPACE

```

TEMPLATE TSYSPPUN
    DSN(' PAOLOR2.&SS..&DB..&SN..UNLOAD.PUNCH1' )

```

```

        DISP(MOD,CATLG,CATLG)
TEMPLATE TSYSREC
        DSN('PAOLOR2.&SS..&DB..&SN..UNLOAD.SYSRC1')
        DISP(MOD,CATLG,CATLG)
UNLOAD TABLESPACE NORMEN00.NORMEN00
        UNLDDN(TSYSREC) PUNCHDDN(TSYSPUN)

```

As expected, the utility ends with RC=8 as shown in Example 6-3, because the total record length of the table ##T.NORMEN00 exceeds 32 KB (message DSNU1218I).

Example 6-3 UNLOAD exceeding a 32 KB row size

```

DSNU000I   201 17:48:30.53 DSNUGUTC - OUTPUT START FOR UTILITY, UTILID = UNLOAD.NORMEN00
DSNU1044I   201 17:48:30.67 DSNUGTIS - PROCESSING SYSIN AS EBCDIC
DSNU050I   201 17:48:30.68 DSNUGUTC - TEMPLATE TSYSREC DSN('PAOLOR2.&SS..&DB..&SN..UNLOAD.PUNCH1') DISP(MOD,
CATLG, CATLG)
DSNU1035I   201 17:48:30.68 DSNUJTDR - TEMPLATE STATEMENT PROCESSED SUCCESSFULLY
DSNU050I   201 17:48:30.68 DSNUGUTC - TEMPLATE TSYSREC DSN('PAOLOR2.&SS..&DB..&SN..UNLOAD.SYSRC1') DISP(MOD,
CATLG, CATLG)
DSNU1035I   201 17:48:30.69 DSNUJTDR - TEMPLATE STATEMENT PROCESSED SUCCESSFULLY
DSNU050I   201 17:48:30.69 DSNUGUTC - UNLOAD TABLESPACE NORMEN00.NORMEN00 UNLDDN(TSYSREC) PUNCHDDN(TSYSPUN)
DSNU1218I  -DB9B 201 17:48:30.72 DSNUULIA - LOGICAL RECORD LENGTH OF OUTPUT RECORD EXCEEDED THE LIMIT FOR TABLE
##T.NORMEN00
DSNU012I   201 17:48:30.74 DSNUGBAC - UTILITY EXECUTION TERMINATED, HIGHEST RETURN CODE=8

```

We get similar results if we change the UNLOAD command to Example 6-4.

Example 6-4 Unload LOB data as normal data columns with UNLOAD TABLE

```

TEMPLATE TSYSREC
        DSN('PAOLOR2.&SS..&DB..&SN..UNLOAD.PUNCH2')
        DISP(MOD,CATLG,CATLG)
TEMPLATE TSYSREC
        DSN('PAOLOR2.&SS..&DB..&SN..UNLOAD.SYSRC2')
        DISP(MOD,CATLG,CATLG)
UNLOAD DATA FROM TABLE ##T.NORMEN00
        UNLDDN(TSYSREC) PUNCHDDN(TSYSPUN)

```

We now show what happens if we limit the fields we are interested in and use the DELIMITED keyword. See Example 6-5.

Example 6-5 Unload LOB data as normal data columns in DELIMITED format

```

TEMPLATE TSYSREC
        DSN('PAOLOR2.&SS..&DB..&SN..UNLOAD.PUNCH4')
        DISP(MOD,CATLG,CATLG)
TEMPLATE TSYSREC
        DSN('PAOLOR2.&SS..&DB..&SN..UNLOAD.SYSRC4')
        DISP(MOD,CATLG,CATLG)
UNLOAD DATA FROM TABLE ##T.NORMEN00
        (DOC_ID,FORMAT,IMAGE)
        DELIMITED
        UNLDDN(TSYSREC) PUNCHDDN(TSYSPUN)

```

We get a different result but the UNLOAD still fails with message DSNU1233I as shown in the job output in Example 6-6 on page 163.

Example 6-6 Unload LOB data as normal data columns in DELIMITED format

```
.....
DSNU050I   201 19:03:57.57 DSNUGUTC - UNLOAD DATA
DSNU650I   -DB9B 201 19:03:57.59 DSNUUGMS - FROM TABLE ##T.NORMEN00
DSNU650I   -DB9B 201 19:03:57.59 DSNUUGMS - (DOC_ID,
DSNU650I   -DB9B 201 19:03:57.59 DSNUUGMS - FORMAT,
DSNU650I   -DB9B 201 19:03:57.59 DSNUUGMS - IMAGE) DELIMITED UNLDDN(TSYSREC) PUNCHDDN(TSYSPUN)
DSNU1038I  201 19:03:57.64 DSNUGDYN - DATASET ALLOCATED. TEMPLATE=TSYSREC
                DDNAME=SYS00001
                DSN=PAOL0R2.DB9B.NORMEN00.NORMEN00.UNLOAD.SYSRC4
DSNU1233I  -DB9B 201 19:03:57.65 DSNUULVA - DATA IS TOO LONG FOR FIELD IMAGE, TABLE ##T.NORMEN00
DSNU1219I  -DB9B 201 19:03:57.65 DSNUULVA - THE NUMBER OF RECORDS IN ERROR REACHED THE LIMIT 1
DSNU253I   201 19:03:57.69 DSNUUNLD - UNLOAD PHASE STATISTICS - NUMBER OF RECORDS UNLOADED=0 FOR TABLE ##T.NORMEN00
DSNU252I   201 19:03:57.69 DSNUUNLD - UNLOAD PHASE STATISTICS - NUMBER OF RECORDS UNLOADED=0 FOR TABLESPACE
NORMEN00.NORMEN00
DSNU250I   201 19:03:57.69 DSNUUNLD - UNLOAD PHASE COMPLETE, ELAPSED TIME=00:00:00
DSNU568I   -DB9B 201 19:03:57.70 DSNUGSRX - INDEX ##T.I_NORMEN00_AUX IS IN INFORMATIONAL COPY PENDING STATE
DSNU012I   201 19:03:57.71 DSNUGBAC - UTILITY EXECUTION TERMINATED, HIGHEST RETURN CODE=8
```

The only way that we are able to UNLOAD some LOB data is by truncating the LOB data so that the output is less than 32 KB. This is illustrated in Example 6-7 where we use the TRUNCATE keyword to truncate the LOB data to 30,000 bytes.

Example 6-7 Unload LOB data as normal data columns in truncated format

```
TEMPLATE TSYPUN
        DSN(' PAOL0R2.&SS..&DB..&SN..UNLOAD.PUNCH5 ')
        DISP(MOD,CATLG,CATLG)
TEMPLATE TSYSREC
        DSN(' PAOL0R2.&SS..&DB..&SN..UNLOAD.SYSRC5 ')
        DISP(MOD,CATLG,CATLG)
UNLOAD DATA FROM TABLE ##T.NORMEN00
        (DOC_ID,FORMAT,IMAGE BLOB(30000) TRUNCATE)
        DELIMITED
        UNLDDN(TSYSREC) PUNCHDDN(TSYSPUN)
```

The output of the resulting job is shown in Example 6-8.

Example 6-8 Unload LOB data as normal data columns in truncated format

```
.....
DSNU050I   209 12:02:26.97 DSNUGUTC - UNLOAD DATA
DSNU650I   -DB9B 209 12:02:26.97 DSNUUGMS - FROM TABLE ##T.NORMEN00
DSNU650I   -DB9B 209 12:02:26.97 DSNUUGMS - (DOC_ID,
DSNU650I   -DB9B 209 12:02:26.97 DSNUUGMS - FORMAT,
DSNU650I   -DB9B 209 12:02:26.97 DSNUUGMS - IMAGE BLOB(30000) TRUNCATE) DELIMITED UNLDDN(TSYSREC) PUNCHDDN(
TSYSPUN)
DSNU1038I  209 12:02:27.06 DSNUGDYN - DATASET ALLOCATED. TEMPLATE=TSYSREC
                DDNAME=SYS00001
                DSN=PAOL0R2.DB9B.NORMEN00.NORMEN00.UNLOAD.SYSRC5
DSNU1038I  209 12:02:51.02 DSNUGDYN - DATASET ALLOCATED. TEMPLATE=TSYSPUN
                DDNAME=SYS00002
                DSN=PAOL0R2.DB9B.NORMEN00.NORMEN00.UNLOAD.PUNCH5
DSNU253I   209 12:02:51.14 DSNUUNLD - UNLOAD PHASE STATISTICS - NUMBER OF RECORDS UNLOADED=5883 FOR TABLE ##T.NORMEN00
DSNU252I   209 12:02:51.14 DSNUUNLD - UNLOAD PHASE STATISTICS - NUMBER OF RECORDS UNLOADED=5883 FOR TABLESPACE
NORMEN00.NORMEN00
DSNU250I   209 12:02:51.14 DSNUUNLD - UNLOAD PHASE COMPLETE, ELAPSED TIME=00:00:24
DSNU010I   209 12:02:51.15 DSNUGBAC - UTILITY EXECUTION COMPLETE, HIGHEST RETURN CODE=0
```

UNLOAD LOB data using file reference variables

This is the method you probably use most when unloading tables containing LOB columns. With this method, the LOB values are unloaded to a different file than the normal unload file. DB2 creates or uses a different output file for each LOB value to be unloaded. The output file can be one of the following types:

- ▶ Member of a partitioned data set (PDS) or partitioned data set extended (PDSE)
- ▶ Hierarchical File System (HFS) file on a HFS directory

It cannot be a simple sequential data set (this was not implemented because of performance reasons: allocate, open, write, close, and deallocate a sequential file for each LOB value would have been a very costly operation).

The LOB unload file contains the entire LOB value and the name of this file is stored in the normal unload output file as a CHAR or VARCHAR field. So, instead of containing the whole LOB value, the normal unload file now only contains a file name, which in most cases no longer causes the sequential unload file to hit the 32 KB limit.

Additional keywords have been added to the CHAR and VARCHAR field-specifications of the UNLOAD utility to support a file name as an output file to store the actual LOB value:

- ▶ BLOBF: The output field contains the name of a file to store a BLOB field.
- ▶ CLOBF: The output field contains the name of a file to store a CLOB field.
- ▶ DBCLOBF: The output field contains the name of a file to store a DBCLOB field.

In the case of CLOBF or DBCLOBF, the required CCSID conversions are done when the EBCDIC, ASCII, UNICODE, or CCSID keywords have been specified in the UNLOAD command. If none of these keywords is specified, the encoding scheme of the source data is preserved. In the case of BLOBF, no conversions are done.

For UNLOAD, the actual files to be created or used for storing the LOB values are generated from a TEMPLATE definition with a name to be specified along with the BLOBF, CLOBF, or DBCLOBF keywords. The TEMPLATE definition is used to specify the characteristics of the LOB unload files:

- ▶ For a PDS:
 - The name of the PDS is generated from the DSN specification of the template.
 - Specify DSNTYPE PDS.
 - Specify DIR to specify the number of directory blocks for a new PDS.
 - The member names are automatically generated.
- ▶ For a PDSE:
 - The name of the PDSE is generated from the DSN specification of the template.
 - Specify DSNTYPE LIBRARY.
 - The member names are automatically generated.
- ▶ For a HFS file:
 - The name of the HFS directory is generated from the DSN specification of the template.
 - Specify DSNTYPE HFS.
 - The HFS directory must exist and be mounted.
 - The data set names in the directory are automatically generated.

For a PDS, the default value of DIR is the estimated number of records divided by 20, which is sufficient to have as many members as the estimated number of records. If you do not specify a DSNTYPE, a PDS is created by default (DSORG = PS).

Note: Run RUNSTATS on the base table space before UNLOAD to allow DB2 to read a realistic value of the AVGWLEN in the SYSIBM.SYSTABLESPACE used to determine the size of the base table. The size needed for LOB table space is calculated by DB2 from the high used relative page count.

The member names that are generated for a PDS or PDSE or the data set names for a HFS directory have a length of 8 and are uniquely generated from the system clock. They have the same look as when using the &UNIQ. or &UQ. variable in a TEMPLATE definition.

Important: Make sure that PTF UK13720 (DB2 V7) or UK13721 (DB2 V8) is installed in your system to use the file reference support for UNLOAD in DB2 V7 or DB2 V8.

Examples

To illustrate, we look at some examples based on the table ##T.NORMEN00 as described in Figure 6-1 on page 187.

In Example 6-9, we try to unload the same data fields from table ##T.NORMEN00 as in Example 6-5 on page 162 or Example 6-7 on page 163, but this time we want to unload the whole LOB data of column IMAGE into a PDS. We specified a TEMPLATE TSYSLOB with DSNTYPE PDS and specified for field IMAGE a field specification as VARCHAR(54) BLOBF TSYSLOB (the maximum possible length for a PDS + membername being 54).

Example 6-9 Unload LOB data to a PDS

```
TEMPLATE TSYPUN
    DSN('PAOLOR2.&SS..&DB..&SN..UNLOAD.PUNCH6')
    DISP(MOD,CATLG,CATLG)
TEMPLATE TSYSREC
    DSN('PAOLOR2.&SS..&DB..&SN..UNLOAD.SYSRC6')
    DISP(MOD,CATLG,CATLG)
TEMPLATE TSYSLOB
    DSN('PAOLOR2.&SS..&DB..&SN..UNLOAD.PDS6')
    DISP(MOD,CATLG,CATLG)
    DSNTYPE(PDS)
UNLOAD DATA FROM TABLE ##T.NORMEN00
    (DOC_ID,FORMAT,IMAGE VARCHAR(54) BLOBF TSYSLOB)
    UNLDDN(TSYSREC) PUNCHDDN(TSYPUN)
```

As a result, three files were created, as shown in the job output in Example 6-10. A punchfile containing the equivalent statements for the LOAD utility, a SYSREC file containing the non-LOB data values and the output file names of the LOB values, and a PDS containing the actual LOB values.

Example 6-10 Unload LOB data to a PDS

```
DSNU000I 205 18:00:05.58 DSNUGUTC - OUTPUT START FOR UTILITY, UTILID = UNLOAD.NORMEN00
DSNU1044I 205 18:00:05.64 DSNUGTIS - PROCESSING SYSIN AS EBCDIC
DSNU050I 205 18:00:05.65 DSNUGUTC - TEMPLATE TSYPUN DSN('PAOLOR2.&SS..&DB..&SN..UNLOAD.PUNCH6') DISP(MOD,
CATLG, CATLG)
DSNU1035I 205 18:00:05.65 DSNUJTDR - TEMPLATE STATEMENT PROCESSED SUCCESSFULLY
DSNU050I 205 18:00:05.65 DSNUGUTC - TEMPLATE TSYSREC DSN('PAOLOR2.&SS..&DB..&SN..UNLOAD.SYSRC6') DISP(MOD,
CATLG, CATLG)
DSNU1035I 205 18:00:05.65 DSNUJTDR - TEMPLATE STATEMENT PROCESSED SUCCESSFULLY
DSNU050I 205 18:00:05.66 DSNUGUTC - TEMPLATE TSYSLOB DSN('PAOLOR2.&SS..&DB..&SN..UNLOAD.PDS6') DISP(MOD, CATLG,
CATLG) DSNTYPE(PDS)
DSNU1035I 205 18:00:05.66 DSNUJTDR - TEMPLATE STATEMENT PROCESSED SUCCESSFULLY
```

```

DSNU050I   205 18:00:05.66 DSNUGUTC - UNLOAD DATA
DSNU650I   -DB9B 205 18:00:05.66 DSNUUGMS - FROM TABLE ##T.NORMEN00
DSNU650I   -DB9B 205 18:00:05.66 DSNUUGMS - (DOC_ID,
DSNU650I   -DB9B 205 18:00:05.66 DSNUUGMS - FORMAT,
DSNU650I   -DB9B 205 18:00:05.66 DSNUUGMS - IMAGE VARCHAR(54) BLOF TSYSLOB) UNLDDN(TSYSREC) PUNCHDDN(TSYSPUN)
DSNU1038I  205 18:00:05.75 DSNUGDYN - DATASET ALLOCATED. TEMPLATE=TSYSREC
          DDNAME=SYS00001
          DSN=PAOLOR2.DB9B.NORMEN00.NORMEN00.UNLOAD.SYSRC6
DSNU1038I  205 18:00:05.96 DSNUGDYN - DATASET ALLOCATED. TEMPLATE=TSYSLOB
          DDNAME=SYS00002
          DSN=PAOLOR2.DB9B.NORMEN00.NORMLOB.UNLOAD.PDS6
DSNU1038I  205 18:02:20.07 DSNUGDYN - DATASET ALLOCATED. TEMPLATE=TSYSPUN
          DDNAME=SYS00003
          DSN=PAOLOR2.DB9B.NORMEN00.NORMEN00.UNLOAD.PUNCH6
DSNU253I   205 18:02:20.16 DSNUUNLD - UNLOAD PHASE STATISTICS - NUMBER OF RECORDS UNLOADED=5883 FOR TABLE ##T.NORMEN00
DSNU252I   205 18:02:20.16 DSNUUNLD - UNLOAD PHASE STATISTICS - NUMBER OF RECORDS UNLOADED=5883 FOR TABLESPACE
NORMEN00.NORMEN00
DSNU250I   205 18:02:20.16 DSNUUNLD - UNLOAD PHASE COMPLETE, ELAPSED TIME=00:02:14
DSNU568I   -DB9B 205 18:02:20.18 DSNUGSRX - INDEX ##T.I_NORMEN00_AUX IS IN INFORMATIONAL COPY PENDING STATE
DSNU010I   205 18:02:20.18 DSNUGBAC - UTILITY EXECUTION COMPLETE, HIGHEST RETURN CODE=0

```

If we take a closer look to the SYSREC file, it contains records similar to Example 6-11. The file name of the LOB value is stored in column 44 as a VARCHAR(54) starting with a 2-byte length field. On column 43, we have a null indicator field, which is x'FF' if the file name is NULL. The actual PDS with DSNAME = 'PAOLOR2.DB9B.NORMEN00.NORMLOB.UNLOAD.PDS6' is allocated as a PDS (DSORG=PO) with 1,287 directory blocks and 5,883 members. The DCB looks like RECFM=VB,LRECL=0,BLKSIZE=27998. Actually, only 281 directory blocks are used.

Example 6-11 Contents of the SYSREC file

...F0577	PDF	...PAOLOR2.DB9B.NORMEN00.NORMLOB.UNLOAD.PDS6(AOHFYAD1)
...F0578	PDF	...PAOLOR2.DB9B.NORMEN00.NORMLOB.UNLOAD.PDS6(AOHFYAEJ)
...F0579	PDF	...PAOLOR2.DB9B.NORMEN00.NORMLOB.UNLOAD.PDS6(AOHFYAE2)
...F0580	PDF	...PAOLOR2.DB9B.NORMEN00.NORMLOB.UNLOAD.PDS6(AOHFYAFN)
...F0581	PDF	...PAOLOR2.DB9B.NORMEN00.NORMLOB.UNLOAD.PDS6(AOHFYAF6)
...F0582	PDF	...PAOLOR2.DB9B.NORMEN00.NORMLOB.UNLOAD.PDS6(AOHFYAGR)
...F0583	PDF	...PAOLOR2.DB9B.NORMEN00.NORMLOB.UNLOAD.PDS6(AOHFYAHA)
...F0584	PDF	...PAOLOR2.DB9B.NORMEN00.NORMLOB.UNLOAD.PDS6(AOHFYAHS)
...F0585	PDF	...PAOLOR2.DB9B.NORMEN00.NORMLOB.UNLOAD.PDS6(AOHFYAIC)
...F0586	PDF	...PAOLOR2.DB9B.NORMEN00.NORMLOB.UNLOAD.PDS6(AOHFYAIW)

Note: APAR PK27029 changes the LRECL=0 to LRECL=27994, which makes more sense for a BLKSIZE=27998 data set. Furthermore, by default DB2 always creates the PDS or PDSE with a BLKSIZE of 27,998 bytes regardless of the device type. Because 27,998 bytes are the optimal BLKSIZE for 3390 devices, we recommend that you preallocate the PDS yourself if you are using device types different than 3390 (for example, the optimal BLKSIZE for 3380 devices is 23,040 bytes).

To unload the LOB data to a PDSE, we can use a utility input file as shown in Example 6-12.

Example 6-12 Unload LOB data to a PDSE

```

TEMPLATE TSYSPUN
          DSN('PAOLOR2.&SS..&DB..&SN..UNLOAD.PUNCH')
          DISP(MOD,CATLG,CATLG)
TEMPLATE TSYSREC

```

```

        DSN('PAOLOR2.&SS.&DB.&SN..UNLOAD.SYSRC7')
        DISP(MOD,CATLG,CATLG)
TEMPLATE TSYSLOB
        DSN('PAOLOR2.&SS.&DB.&SN..UNLOAD.PDSE7')
        DISP(MOD,CATLG,CATLG)
        DSNTYPE(LIBRARY)
UNLOAD DATA FROM TABLE ##T.NORMEN00
        (DOC_ID,FORMAT,IMAGE VARCHAR(54) BLOBF TSYSLOB)
        UNLDDN(TSYSREC) PUNCHDDN(TSYSPUN)

```

The contents of the SYSREC file look as shown in Example 6-13. The member names are different because the job was run at a later time than in Example 6-11 on page 166.

Example 6-13 Contents of the SYSREC file

....F0677	PDF	...PAOLOR2.DB9B.NORMEN00.NORMLOB.UNLOAD.PDSE7(A0IQCZXN)
....F0678	PDF	...PAOLOR2.DB9B.NORMEN00.NORMLOB.UNLOAD.PDSE7(A0IQCZYL)
....F0679	PDF	...PAOLOR2.DB9B.NORMEN00.NORMLOB.UNLOAD.PDSE7(A0IQCZZO)
....F0680	PDF	...PAOLOR2.DB9B.NORMEN00.NORMLOB.UNLOAD.PDSE7(A0IQCZ00)
....F0681	PDF	...PAOLOR2.DB9B.NORMEN00.NORMLOB.UNLOAD.PDSE7(A0IQCZ1N)
....F0682	PDF	...PAOLOR2.DB9B.NORMEN00.NORMLOB.UNLOAD.PDSE7(A0IQCZ20)
....F0683	PDF	...PAOLOR2.DB9B.NORMEN00.NORMLOB.UNLOAD.PDSE7(A0IQCZ30)
....F0684	PDF	...PAOLOR2.DB9B.NORMEN00.NORMLOB.UNLOAD.PDSE7(A0IQCZ40)
....F0685	PDF	...PAOLOR2.DB9B.NORMEN00.NORMLOB.UNLOAD.PDSE7(A0IQCZ5Q)
....F0686	PDF	...PAOLOR2.DB9B.NORMEN00.NORMLOB.UNLOAD.PDSE7(A0IQCZ6S)

To unload the LOB data to a HFS directory, we can use a utility input file as shown in Example 6-14. Be sure that the userid associated with the utility job has the rights to add new files to the specified HFS directory. In this example, the HFS directory /u/DB9B was created and mounted before.

Example 6-14 Unload LOB data to a HFS directory

```

TEMPLATE TSYPUN
        DSN('PAOLOR2.&SS.&DB.&SN..UNLOAD.PUNCH8')
        DISP(MOD,CATLG,CATLG)
TEMPLATE TSYSREC
        DSN('PAOLOR2.&SS.&DB.&SN..UNLOAD.SYSRC8')
        DISP(MOD,CATLG,CATLG)
TEMPLATE TSYSLOB
        DSN('/u/&SS.')
        DSNTYPE(HFS)
UNLOAD DATA FROM TABLE ##T.NORMEN00
        (DOC_ID,FORMAT,IMAGE VARCHAR(20) BLOBF TSYSLOB)
        UNLDDN(TSYSREC) PUNCHDDN(TSYSPUN)

```

After the job is run, the HFS directory can be listed with a UNIX command **cd /u/DB9B** followed by **ls -l** as shown in Example 6-15 (we only show the last lines here).

Example 6-15 Contents of the HFS directory /u/DB9B

-rwxrwx---	1	HAIMO	SYS1	29350	Ju1	25	15:10	A0IUJYZ3
-rwxrwx---	1	HAIMO	SYS1	29390	Ju1	25	15:10	A0IUJYZ7
-rwxrwx---	1	HAIMO	SYS1	29357	Ju1	25	15:10	A0IUJYZC
-rwxrwx---	1	HAIMO	SYS1	29380	Ju1	25	15:10	A0IUJYZG
-rwxrwx---	1	HAIMO	SYS1	28939	Ju1	25	15:10	A0IUJYZL
-rwxrwx---	1	HAIMO	SYS1	28930	Ju1	25	15:10	A0IUJYZP
-rwxrwx---	1	HAIMO	SYS1	28987	Ju1	25	15:10	A0IUJYZU
-rwxrwx---	1	HAIMO	SYS1	29122	Ju1	25	15:10	A0IUJYZY

PAOLOR2 @ SC63:/u/DB9B>
===>

The contents of the SYSREC file are shown in Example 6-16.

Example 6-16 Contents of the SYSREC file

....F0677	PDF	.../u/DB9B/A0IUIOMQ
....F0678	PDF	.../u/DB9B/A0IUIOMU
....F0679	PDF	.../u/DB9B/A0IUIOMZ
....F0680	PDF	.../u/DB9B/A0IUIOM3
....F0681	PDF	.../u/DB9B/A0IUIOM8
....F0682	PDF	.../u/DB9B/A0IUIOND
....F0683	PDF	.../u/DB9B/A0IUIONH
....F0684	PDF	.../u/DB9B/A0IUIONM
....F0685	PDF	.../u/DB9B/A0IUI2HI
....F0686	PDF	.../u/DB9B/A0IUI2HM

Tip: Keep in mind that HFS directories are case sensitive and that the outcome of TEMPLATE variables is always in uppercase. So, create your HFS directories partly in uppercase if you use template variables to map your HFS directories.

Some additional remarks

When using the UNLOAD utility for unloading LOB data with file reference variables, beware of the following:

- ▶ Unloading to sequential files (DSORG=PS) is not supported as explained in “UNLOAD LOB data using file reference variables” on page 164.
- ▶ Unloading LOB data from image copies into files using file reference variables is not supported.
- ▶ When using HFS, the HFS directory must exist and the TEMPLATE must specify the full path name (not the relative path name).
- ▶ You cannot specify (&UNIQ.) or (&UQ.) as the member name in the TEMPLATE for a PDS or PDSE, but it is always automatically appended by DB2 to the name expression of the TEMPLATE for a DSNTYPE PDS or LIBRARY.
- ▶ You cannot specify /&UNIQ. or /&UQ. as the data set name in the TEMPLATE for a HFS, but it is always automatically appended by DB2 to the name expression of the TEMPLATE for a DSNTYPE HFS.
- ▶ Each time that you run the UNLOAD utility, the generated member names or HFS data set names are different.
- ▶ A NULL LOB is represented by a NULL output file name (null indicator field preceding the CHAR or VARCHAR field is hex FF).
- ▶ When you specify &TS., &SN., or &IS. in the template for a PDS, PDSE, or HFS, it is always replaced by the LOB table space name and not by the base table space name as shown in Example 6-10 on page 165. This is to ensure that the generated data set name is unique in case the base table contains multiple LOB columns or is partitioned.

6.2 DSNTIAUL

Like the UNLOAD utility, the DSNTIAUL sample program also provides two ways to handle LOB data:

- ▶ Unload LOB columns as normal data columns to the SYSRECxx files (with truncation).
- ▶ Unload LOB columns to a separate file (DB2 9 only).

Unload LOB data as normal data columns (SQL parameter)

As the maximum record length of a sequential file in z/OS is 32 KB, this method can only be used if the total record size of the data to be unloaded does not exceed 32 KB. The LOB fields are unloaded together with the other selected data fields to the output file with a maximum record length of 32 KB. If the sum of the lengths of all columns exceeds 32 KB, DSNTIAUL truncates all LONG VARCHAR, LONG VARGRAPHIC, and LOB columns to obtain a total output length of 32 KB.

In most cases, this method is only used if the LOBs in the table are small LOBs (SLOBs) with an defined maximum length smaller than 32 KB or if the user deliberately decides to truncate the LOB data.

We demonstrate this with the table ##T.NORMEN00 defined in Example 6-1 on page 160.

In the first case, we execute the JCL as shown in Example 6-17.

Example 6-17 DSNTIAUL with SQL parameter

```
//DSNTIAUL EXEC PGM=IKJEFT01,DYNAMNBR=20,COND=(4,LT)
//SYSTSPT DD SYSOUT=*
//SYSTSIN DD *
DSN SYSTEM(DB9B)
RUN PROGRAM(DSNTIAUL) PLAN(DSNTIB91) PARMS('SQL') -
LIB('DB9BU.RUNLIB.LOAD')
//SYSPRINT DD SYSOUT=*
//SYSREC00 DD DSN=PAOLOR2.DB9B.DSN8UNLD.SQL.SYSREC00,
//          DISP=(,CATLG),UNIT=3390,
//          SPACE=(CYL,(100,100))
//SYSPUNCH DD DSN=PAOLOR2.DB9B.DSN8UNLD.SQL.SYSPUNCH,
//          DISP=(,CATLG),UNIT=3390,
//          SPACE=(CYL,(1,1))
//SYSIN DD *
        SELECT DOC_ID,FORMAT,IMAGE FROM ##T.NORMEN00 ;
```

As a result, we get an FB sequential data set SYSREC00 with a LRECL and BLKSIZE of 32,753 bytes with DOC_ID beginning in position 1, FORMAT beginning in position 33, and the IMAGE data starting in position 41 and truncated to 32,708 bytes as shown in Example 6-18.

Example 6-18 DSNTIAUL output with SQL parameter

```
DSNT490I SAMPLE DATA UNLOAD PROGRAM
DSNT505I DSNTIAUL OPTIONS USED: SQLTOLWAR
DSNT503I UNLOAD DATA SET SYSPUNCH RECORD LENGTH SET TO      80
DSNT504I UNLOAD DATA SET SYSPUNCH BLOCK SIZE SET TO 27920
DSNT506I INPUT STATEMENT WAS NOT A FULL SELECT ON A SINGLE TABLE. LOAD STATEMENT WILL NEED
MODIFICATION.
WARNING: DATA FROM COLUMN      3 WAS TRUNCATED TO      32755 BYTES FROM 2097152.
WARNING: DATA FROM COLUMN      3 WAS TRUNCATED TO      32708 BYTES FROM      32755.
DSNT503I UNLOAD DATA SET SYSREC00 RECORD LENGTH SET TO 32753
DSNT504I UNLOAD DATA SET SYSREC00 BLOCK SIZE SET TO 32753
```

With the SQL statement `SELECT * FROM ##T.NORMEN00`, we would get all the normal data columns (except for the ROWID column that is omitted when it is type GENERATED ALWAYS) and the IMAGE column truncated to 32,632 bytes.

Unload LOB data to a separate file (LOBFIL parameter)

This is the recommended method introduced with DB2 9. With this method, the LOB values are unloaded to a different file than the normal SYSRECxx unload files. DSNTIAUL dynamically creates a sequential data set for each LOB value to be unloaded. The name of the separate file is stored in the normal SYSRECxx unload files together with the other normal data fields (except the ROWID column when it is type GENERATED ALWAYS).

DSNTIAUL does this by using the new LOB file reference variables introduced in DB2 9.

Each LOB file has a name of the form `<prefix>.Q<i>.C<j>.R<k>`, where:

- ▶ `<prefix>` is a user-specified data set name prefix. `<prefix>` must conform to the rules for an z/OS physical sequential data set name and cannot exceed 17 characters.
- ▶ `Q<i>` is the (`<i>-1`)th query processed by the current DSNTIAUL session. `<i>` ranges from 0,000,000 to 0,000,099, which corresponds to the limit on the number of queries that can be processed by a single DSNTIAUL session.
- ▶ `C<j>` is the (`<j>-1`)th column in the current SELECT statement. `<j>` ranges from 0,000,000 to 0,000,999 (no more than 750 columns are permitted in a table or view).
- ▶ `R<k>` is the (`<k>-1`)th row FETCHed for the current SELECT statement. `<k>` ranges from 0,000,000 to 9,999,999.

The data set prefix `<prefix>` is specified by means of a new DSNTIAUL run-time parameter called LOBFIL.

We demonstrate this with the JCL as shown in Example 6-19.

Example 6-19 DSNTIAUL with LOBFIL parameter

```
//DSNTIAUL EXEC PGM=IKJEFT01,DYNAMBR=20,COND=(4,LT)
//SYSTSPRT DD SYSOUT=*
//SYSTSIN DD *
DSN SYSTEM(DB9B)
RUN PROGRAM(DSNTIAUL) PLAN(DSNTIB91) PARM('LOBFIL(PAOLOR2)') -
LIB('DB9BU.RUNLIB.LOAD')
//SYSPRINT DD SYSOUT=*
//SYSRECOO DD DSN=PAOLOR2.DB9B.DSN8UNLD.LOBF.SYSRECOO,
//          DISP=(,CATLG),UNIT=3390,
//          SPACE=(CYL,(10,10))
//SYSPUNCH DD DSN=PAOLOR2.DB9B.DSN8UNLD.LOBF.SYSPUNCH,
//          DISP=(,CATLG),UNIT=3390,
//          SPACE=(CYL,(1,1))
//SYSIN DD *
      ##T.NORMEN00
```

In this case, the SYSIN input file contains the name of the base table. You can also specify SQL as first PARM:

```
RUN PROGRAM(DSNTIAUL) PLAN(DSNTIB91) PARM('SQL,LOBFIL(PAOLOR2)') -
```


And you can use an SQL statement as input to SYSIN:

```
SELECT * FROM ##T.NORMEN00;
```

In both cases, as a result we get a sequential data set SYSREC00 containing the normal data fields (except the ROWID column) and the name of the LOB output files and 5,883 new LOB files. See Example 6-20. All of the LOB output files have a name of the form PAOLOR2.Q0000000.C0000006.R000xxxx with xxxx from 0000 to 5882. They are dynamically allocated as sequential files with RECFM=VB,LRECL=27994,BLKSIZE=27998, which is the optimal BLKSIZE for 3390 devices.

Example 6-20 DSNTIAUL output with LOBFILE parameter

```
DSNT490I SAMPLE DATA UNLOAD PROGRAM
DSNT503I UNLOAD DATA SET SYSPUNCH RECORD LENGTH SET TO      80
DSNT504I UNLOAD DATA SET SYSPUNCH BLOCK SIZE SET TO 27920
DSNT503I UNLOAD DATA SET SYSREC00 RECORD LENGTH SET TO    162
DSNT504I UNLOAD DATA SET SYSREC00 BLOCK SIZE SET TO 27864
DSNT495I SUCCESSFUL UNLOAD          5883 ROWS OF TABLE ##T.NORMEN00
```

Afterwards, the SYSPUNCH file, the SYSREC00 file, and the 5883 LOB files can be used as input for the LOAD utility.

Important: When the DSNTIAUL job is run again, and the LOB files from the previous run still exist, they are REPLACED by the new run with the same name.

6.3 LOAD

Depending on the way the LOB data is provided, there are three ways the load utility can be used to load LOB data:

- ▶ Loading LOB data as normal data fields from the LOAD input file
- ▶ Using file reference variables when each LOB value is stored in a separate input file
- ▶ Using the cross loader

6.3.1 Loading LOB data as normal data columns

This method can be used when the LOB values are stored in the normal LOAD input file together with the other data fields of the base table. Because the maximum record length of a sequential file in z/OS is 32 KB, this method can only be used to LOAD LOB columns of 32 KB or less. The sum of the length of all normal data fields and LOB fields in the input file cannot exceed 32 KB.

In most cases, this method is only used if the LOBs in the table are small LOBs (SLOBs) with an actual length smaller than 32 KB. The defined length of the LOB column on the CREATE TABLE statement can exceed 32 KB.

LOB fields are varying length and require a valid 4-byte binary length field preceding the data; no intervening gaps are allowed between them and the LOB fields that follow. Specify CLOB, BLOB, or DBCLOB in the field specification portion of the LOAD statement. These options indicate that the field in the input data set is a LOB value.

Normally the provided input file is the result of:

- ▶ UNLOAD utility
- ▶ DSNTIAUL utility
- ▶ Generated by another application

6.3.2 Loading LOB data using file reference variables

This method is used when the LOB values are stored in separate input files. The normal input file contains the data for the non-LOB columns of the base table and the names of the LOB files. The sum of the length of all normal data fields and the LOB file names cannot exceed 32 KB.

The LOB input files can be any of these types:

- ▶ A sequential file
- ▶ A member of a PDS or PDSE
- ▶ A HFS file on a HFS directory

The LOB input file contains the entire LOB value and the name of this file is stored in the normal load input file as a CHAR or VARCHAR field. So, instead of containing the whole LOB value, the normal input file now only contains a file name, which in most cases no longer causes the sequential file to hit the 32 KB limit.

Additional keywords have been added to the CHAR and VARCHAR field specifications of the LOAD utility to support a file name as the input for the actual LOB value:

- ▶ BLOBF: The input field contains the name of a file with a BLOB value.
- ▶ CLOBF: The input field contains the name of a file with a CLOB value.
- ▶ DBCLOBF: The input field contains the name of a file with a DBCLOB value.

In case of CLOBF and DBCLOBF, CCSID conversions are done when the CCSID of the input data is different than the CCSID of the table space. (EBCDIC, ASCII, UNICODE, or CCSID keywords might have been specified for the input data; the default is EBCDIC input data). In case of BLOBF, no conversions are done.

When the input field of a BLOBF, CLOBF, or DBCLOBF is NULL, the resulting LOB value is NULL (null indicator field for the CHAR or VARCHAR field specified in the NULLIF keyword is hex FF).

Important: Make sure that PTF UK13720 (DB2 V7) or UK13721 (DB2 V8) is installed in your system to use the file reference support for LOAD in DB2 V7 or DB2 V8.

Furthermore, always specify a NULLIF keyword to tell DB2 when the LOB is supposed to be NULL.

Normally, the provided input files are the result of:

- ▶ UNLOAD utility (PDS members, PDSE members, or HFS files)
- ▶ DSNTIAUL utility (sequential files)
- ▶ Generated by another application

6.3.3 Using the cross loader

With DB2 9 for z/OS, the cross loader function of the LOAD utility can be used to load LOB data that resides in another table on the same or another location connected through DRDA. When you use the cross loader function for LOBs greater than 32 KB, DB2 uses a separate

buffer for LOB data and only stores 8 bytes per LOB column in the cursor result set buffer. The sum of the lengths of the non-LOB columns plus the sum of 8 bytes per LOB column cannot exceed 32 KB. The separate LOB buffer resides in storage above the 16 MB line and is only limited by the available memory above the 16 MB line.

The message DSNU1178I is issued when:

- The sum of the lengths of all non-LOB columns + 8 bytes per LOB column exceeds 32 KB.
- The sum of the lengths of all LOB columns exceeds half of the above-the-line available memory.

If a user gets message DSNU1178I, increasing the region size likely results in successful execution of the LOAD utility.

See Example 6-26 on page 176 for a cross loader job execution.

Important: Apply PTF UQ03226 (DB2 V7) or UQ03227 (DB2 V8) to use the cross loader support for LOBs in DB2 V7 or DB2 V8.

6.3.4 Impact of logging

LOAD allows you to specify LOG YES or NO during the execution. A LOB table space can also be defined with LOG YES or LOG NO in DB2 V8. In DB2 9 also, the base table space can be defined as LOGGED or NOT LOGGED.

Table 6-1 shows the effect on logging output and LOB table space in case the base table space is LOGGED.

Table 6-1 Impact of logging if base table space is LOGGED

Impact of logging on LOAD			
LOAD LOG keyword	LOB table space LOG attribute	What is logged	LOB table space status after utility completes
LOG YES	LOG YES	Control information and LOB data	No pending status
LOG YES	LOG NO	Control information	No pending status
LOG NO	LOG YES	Nothing	COPY Pending
LOG NO	LOG NO	Nothing	COPY Pending

If the base table space is defined as NOT LOGGED and the LOB table spaces are defined as LOGGED, the LOB table spaces with the LOGGED logging attribute are changed to NOT LOGGED as well (however, this is recorded in the DB2 catalog, so that if the base table space is altered to LOGGED, the LOB table spaces are also changed back to LOGGED). If both base table space and LOB table spaces are NOT LOGGED, the LOG YES attribute of the LOAD utility also is changed to LOG NO during execution. Nothing is logged and the LOB table spaces are in the no pending state afterwards.

When the LOB table space ends up being in COPY status, you should take a full image copy afterwards to establish recoverability and allow update activity to the LOBs. Adding a COPYDDN statement to the LOAD utility (with REPLACE) does not help because this only takes an image copy of the base table space.

Examples

We demonstrate this with some examples.

In Example 6-21, we use the output of DSNTIAUL from Example 6-19 on page 170 to load a newly created table ##T.NORMEN02. This table is created with the same DDL as the table ##T.NORMEN00 defined in Example 6-1 on page 160 with NORMEN00 replaced everywhere by NORMEN02.

Example 6-21 LOAD LOB data with input from DSNTIAUL

```
//LOAD EXEC PGM=DSNUTILB,PARM='DB9B,LOAD.NORMEN02',COND=(4,LT)
//SYSTEMPL DD DUMMY
//UTPRINT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSREC00 DD DSN=PAOL0R2.DB9B.DSN8UNLD.LOBF.SYSREC00,DISP=SHR
//SYSIN DD *
TEMPLATE TSORTOUT
    DSN('DB2RE.&SS..&DB..&SN..S&JU(3,5)..#&TI.')
    DISP(MOD,DELETE,CATLG)
TEMPLATE TSYSUT1
    DSN('DB2RE.&SS..&DB..&SN..U&JU(3,5)..#&TI.')
    DISP(MOD,DELETE,CATLG)
LOAD DATA LOG NO INDDN SYSREC00 INTO TABLE ##T.NORMEN02
(DOC_ID      POSITION(1)      VARCHAR,
 PAGE_NUMBER POSITION(33)     SMALLINT,
 IMPORTER     POSITION(35)     CHAR(8),
 IMPORT_TIME  POSITION(43)     TIMESTAMP EXTERNAL(26),
 FORMAT       POSITION(69)     CHAR(8),
 IMAGE        POSITION(119)    CHAR(34) BLOBF )
SORTNUM 8 SORTDEVT 3390
WORKDDN(TSYSUT1,TSORTOUT)
```

The result of this job is shown in Example 6-22.

Example 6-22 LOAD LOB data with input from DSNTIAUL

```
DSNU000I 207 20:07:27.58 DSNUGUTC - OUTPUT START FOR UTILITY, UTILID = LOAD.NORMEN02
DSNU1044I 207 20:07:27.63 DSNUGTIS - PROCESSING SYSIN AS EBCDIC
DSNU050I 207 20:07:27.64 DSNUGUTC - TEMPLATE TSORTOUT DSN('DB2RE.&SS..&DB..&SN..S&JU(3,5)..#&TI.') DISP(MOD,
DELETE, CATLG)
DSNU1035I 207 20:07:27.65 DSNUJTDR - TEMPLATE STATEMENT PROCESSED SUCCESSFULLY
DSNU050I 207 20:07:27.65 DSNUGUTC - TEMPLATE TSYSUT1 DSN('DB2RE.&SS..&DB..&SN..U&JU(3,5)..#&TI.') DISP(MOD,
DELETE, CATLG)
DSNU1035I 207 20:07:27.65 DSNUJTDR - TEMPLATE STATEMENT PROCESSED SUCCESSFULLY
DSNU050I 207 20:07:27.65 DSNUGUTC - LOAD DATA LOG NO INDDN SYSREC00
DSNU650I -DB9B 207 20:07:27.65 DSNURWI - INTO TABLE ##T.NORMEN02
DSNU650I -DB9B 207 20:07:27.65 DSNURWI - (DOC_ID POSITION(1) VARCHAR,
DSNU650I -DB9B 207 20:07:27.65 DSNURWI - PAGE_NUMBER POSITION(33) SMALLINT,
DSNU650I -DB9B 207 20:07:27.65 DSNURWI - IMPORTER POSITION(35) CHAR(8),
DSNU650I -DB9B 207 20:07:27.65 DSNURWI - IMPORT_TIME POSITION(43) TIMESTAMP EXTERNAL(26),
DSNU650I -DB9B 207 20:07:27.65 DSNURWI - FORMAT POSITION(69) CHAR(8),
DSNU650I -DB9B 207 20:07:27.65 DSNURWI - IMAGE POSITION(119) CHAR(34) BLOBF) SORTNUM 8 SORTDEVT 3390 WORKDDN(
TSYSUT1, TSORTOUT)
DSNU1038I 207 20:07:28.01 DSNUGDYN - DATASET ALLOCATED. TEMPLATE=TSYSUT1
DDNAME=SYS00001
DSN=DB2RE.DB9B.NORMEN02.NORMEN02.U06208.#000727
DSNU1038I 207 20:07:28.04 DSNUGDYN - DATASET ALLOCATED. TEMPLATE=TSORTOUT
DDNAME=SYS00002
DSN=DB2RE.DB9B.NORMEN02.NORMEN02.S06208.#000727
DSNU080I 207 20:07:28.04 DSNUGPRS - NO UTILITY STATEMENTS FOUND IN SYSTEMPL
```

```

DSNU304I  -DB9B 207 20:09:23.10 DSNURWT - (RE)LOAD PHASE STATISTICS - NUMBER OF RECORDS=5883 FOR TABLE ##T.NORMEN02
DSNU1147I -DB9B 207 20:09:23.10 DSNURWT - (RE)LOAD PHASE STATISTICS - TOTAL NUMBER OF RECORDS LOADED=5883 FOR
TABLESPACE NORMEN02.NORMEN02
DSNU302I   207 20:09:23.10 DSNURILD - (RE)LOAD PHASE STATISTICS - NUMBER OF INPUT RECORDS PROCESSED=5883
DSNU300I   207 20:09:23.10 DSNURILD - (RE)LOAD PHASE COMPLETE, ELAPSED TIME=00:01:55
DSNU042I   207 20:09:23.64 DSNUGSOR - SORT PHASE STATISTICS -
NUMBER OF RECORDS=5883
ELAPSED TIME=00:00:00
DSNU349I -DB9B 207 20:09:23.94 DSNURBXA - BUILD PHASE STATISTICS - NUMBER OF KEYS=5883 FOR INDEX ##T.I_NORMEN02_1
DSNU258I   207 20:09:23.95 DSNURBXD - BUILD PHASE STATISTICS - NUMBER OF INDEXES=1
DSNU259I   207 20:09:23.95 DSNURBXD - BUILD PHASE COMPLETE, ELAPSED TIME=00:00:00
DSNU381I -DB9B 207 20:09:24.01 DSNUGSRX - TABLESPACE NORMEN02.NORMEN02 IS IN COPY PENDING
DSNU381I -DB9B 207 20:09:24.01 DSNUGSRX - TABLESPACE NORMEN02.NORMLOB IS IN COPY PENDING
DSNU568I -DB9B 207 20:09:24.01 DSNUGSRX - INDEX ##T.I_NORMEN02_AUX IS IN INFORMATIONAL COPY PENDING STATE
DSNU568I -DB9B 207 20:09:24.01 DSNUGSRX - INDEX ##T.I_NORMEN02_1 IS IN INFORMATIONAL COPY PENDING STATE
DSNU010I   207 20:09:24.03 DSNUGBAC - UTILITY EXECUTION COMPLETE, HIGHEST RETURN CODE=4

```

In this case, nothing is logged (LOB table space defined as LOG NO, LOAD with LOG NO and both the base table space and LOB table space are put in COPY Pending).

Similar tests were done if the input LOB files are members of a PDS, members of a PDSE, or HFS files on a HFS directory (created by the UNLOAD utility).

When we do the same test again, but now, we create the base table space as NOT LOGGED and the LOB table space as LOGGED, the resulting entries in SYSIBM.SYSTABLESPACE look as shown in Example 6-23.

Example 6-23 Entries in SYSIBM.SYSTABLESPACE

NAME	DBNAME	TYPE	LOG
NORMEN02	NORMEN02		N
NORMLOB	NORMEN02	0	X

The value X in the LOG column means that the LOB table space currently has the value NOT LOGGED, because the base table space has the NOT LOGGED attribute. If we now do a LOAD with LOG YES we get message DSNU1153I as shown in Example 6-24, and nothing is logged. The base table space is put in INFORMATIONAL COPY PENDING STATE and the LOB table space has no pending states after the LOAD.

Example 6-24 LOAD with LOG YES on NOT LOGGED table spaces

```

DSNU050I   208 12:13:59.66 DSNUGUTC - LOAD DATA LOG YES INDDN SYSREC00
DSNU1153I -DB9B 208 12:13:59.66 DSNURWI - LOG YES SPECIFIED FOR THE NOT LOGGED TABLESPACE NORMEN02.NORMEN02 WILL BE
IGNORED
DSNU650I -DB9B 208 12:13:59.66 DSNURWI - INTO TABLE ##T.NORMEN02
DSNU650I -DB9B 208 12:13:59.66 DSNURWI - (DOC_ID POSITION(1) VARCHAR,
DSNU650I -DB9B 208 12:13:59.66 DSNURWI - PAGE_NUMBER POSITION(33) SMALLINT,
DSNU650I -DB9B 208 12:13:59.66 DSNURWI - IMPORTER POSITION(35) CHAR(8),
DSNU650I -DB9B 208 12:13:59.66 DSNURWI - IMPORT_TIME POSITION(43) TIMESTAMP EXTERNAL(26),
DSNU650I -DB9B 208 12:13:59.66 DSNURWI - FORMAT POSITION(69) CHAR(8),
DSNU650I -DB9B 208 12:13:59.66 DSNURWI - IMAGE POSITION(119) CHAR(34) BLOBF) SORTNUM 8 SORTDEVT 3390 WORKDDN(
TSYSUT1, TSORTOUT)
DSNU1038I  208 12:14:00.11 DSNUGDYN - DATASET ALLOCATED. TEMPLATE=TSYSUT1
DDNAME=SYS00001
DSN=DB2RE.DB9B.NORMEN02.NORMEN02.U06208.#161359
DSNU1038I  208 12:14:00.20 DSNUGDYN - DATASET ALLOCATED. TEMPLATE=TSORTOUT
DDNAME=SYS00002
DSN=DB2RE.DB9B.NORMEN02.NORMEN02.S06208.#161359
DSNU080I   208 12:14:00.20 DSNUGPRS - NO UTILITY STATEMENTS FOUND IN SYSTEMPL
DSNU304I  -DB9B 208 12:15:55.01 DSNURWT - (RE)LOAD PHASE STATISTICS - NUMBER OF RECORDS=5883 FOR TABLE ##T.NORMEN02
DSNU1147I -DB9B 208 12:15:55.01 DSNURWT - (RE)LOAD PHASE STATISTICS - TOTAL NUMBER OF RECORDS LOADED=5883 FOR

```

```

TABLESPACE NORMEN02.NORMEN02
DSNU302I   208 12:15:55.02 DSNURILD - (RE)LOAD PHASE STATISTICS - NUMBER OF INPUT RECORDS PROCESSED=5883
DSNU300I   208 12:15:55.02 DSNURILD - (RE)LOAD PHASE COMPLETE, ELAPSED TIME=00:01:54
DSNU042I   208 12:15:55.58 DSNUGSOR - SORT PHASE STATISTICS -
          NUMBER OF RECORDS=5883
          ELAPSED TIME=00:00:00
DSNU349I -DB9B 208 12:15:55.92 DSNURBXA - BUILD PHASE STATISTICS - NUMBER OF KEYS=5883 FOR INDEX ##T.I_NORMEN02_1
DSNU258I   208 12:15:55.93 DSNURBXD - BUILD PHASE STATISTICS - NUMBER OF INDEXES=1
DSNU259I   208 12:15:55.93 DSNURBXD - BUILD PHASE COMPLETE, ELAPSED TIME=00:00:00
DSNU568I -DB9B 208 12:15:55.98 DSNUGSRX - TABLESPACE NORMEN02.NORMEN02 IS IN INFORMATIONAL COPY PENDING STATE
DSNU568I -DB9B 208 12:15:55.98 DSNUGSRX - INDEX ##T.I_NORMEN02_AUX IS IN INFORMATIONAL COPY PENDING STATE
DSNU568I -DB9B 208 12:15:55.98 DSNUGSRX - INDEX ##T.I_NORMEN02_1 IS IN INFORMATIONAL COPY PENDING STATE
DSNU010I   208 12:15:55.99 DSNUGBAC - UTILITY EXECUTION COMPLETE, HIGHEST RETURN CODE=0

```

If DB2 cannot find one or more of the LOB input files, a new SQLCODE-452 is issued as shown in Example 6-25.

Example 6-25 LOAD LOB input file not found

```

DSNU350I -DB9B 208 14:23:08.29 DSNURRST - EXISTING RECORDS DELETED FROM TABLESPACE
DSNU283I -DB9B 208 14:23:08.43 DSNURWBF - LOB ERROR      SQLCODE =
-452
      SQLERRM = PA0L0R2.Q0000000.C0000006.R0000000 3
      SQLSTATE= 428A1
      SQLERRP = DSNOLFRV
      SQLERRD = 0000000A 00000000 00000000 FFFFFFFF 00000000 00000000
DSNU017I   208 14:23:12.08 DSNUGBAC - UTILITY DATA BASE SERVICES MEMORY EXECUTION ABENDED, REASON=X'00E40350'
CAUSE=X'00C90072

```

An example with the cross loader is shown in Example 6-26 where we LOAD ##T.NORMEN02 from ##T.NORMEN00 using a cursor.

Example 6-26 LOAD LOB data with cross loader

```

TEMPLATE TSORTOUT
      DSN('DB2RE.&SS..&DB..&SN..S&JU(3,5)..#&TI.')
      DISP(MOD,DELETE,CATLG)
      SPACE(10,10) CYL
TEMPLATE TSYSUT1
      DSN('DB2RE.&SS..&DB..&SN..U&JU(3,5)..#&TI.')
      DISP(MOD,DELETE,CATLG)
      SPACE(10,10) CYL
EXEC SQL
      DECLARE C1 CURSOR FOR SELECT
          DOC_ID
        ,PAGE_NUMBER
        ,IMPORTER
        ,IMPORT_TIME
        ,FORMAT
        ,IMAGE
      FROM ##T.NORMEN00
ENDEXEC
LOAD DATA INCURSOR C1 REPLACE
      WORKDDN(TSYSUT1,TSORTOUT)
      SORTDEVT 3390 SORTNUM 8
      INTO TABLE ##T.NORMEN02

```

Because the ROWID column of table ##T.NORMEN00 is explicitly defined and the ROWID column of ##T.NORMEN02 is explicitly defined as GENERATED ALWAYS, we cannot use

SELECT * here. We could have used SELECT * if both ROWIDs were implicitly created or when the ROWID of ##T.NORMEN02 was created a GENERATED BY DEFAULT (and with a unique index defined on it).

Some additional remarks

When using the LOAD utility to load LOB data, beware of:

- ▶ Indexes on the auxiliary tables are not built during the BUILD phase. Instead, LOB values are inserted (not loaded) into auxiliary tables during the RELOAD phase as each row is loaded into the base table, and each index on the auxiliary table is updated as part of the INSERT operation. Because the LOAD utility inserts keys into an auxiliary index, free space within the index might be consumed and index page splits might occur. Consider reorganizing an index on the auxiliary table after LOAD completes to introduce free space into the index for future INSERTs and LOADs.

- ▶ Using COPYDDN when loading a table with LOB columns does not create a copy of any LOB table space or auxiliary index. You get the message:

```
DSNU068I - DSNURWI - KEYWORD COPYDDN IS NOT SUPPORTED FOR LOB OR XML TABLE SPACES
```

You must perform these tasks separately.

- ▶ Using STATISTICS when loading a table with LOB columns only gathers statistics for the base table space but not for the LOB table spaces. You must perform these tasks separately.
- ▶ Since DB2 V8, the default value for SORTKEYS is SORTKEYS 0. If you plan to load a table that has LOB columns using LOAD RESUME YES SHRLEVEL NONE, and you might need to restart the LOAD job with RESTART(CURRENT), then you must specify SORTKEYS NO; otherwise, the utility cannot be restarted.
- ▶ If you use RESTART PHASE to restart a LOAD job which specified RESUME NO, the LOB table spaces and indexes on auxiliary tables are reset.
- ▶ If the SELECT statement in the cursor definition contains the ROWID column of the source table, the table to be loaded must have a ROWID column with the same name and defined as GENERATED BY DEFAULT (+ unique index defined on the ROWID column). If the ROWID column of the source table was implicitly created, SELECT * does not contain the ROWID of the table.

6.4 COPY

Both full and incremental image copies are supported for a LOB table space, as well as SHRLEVEL REFERENCE, SHRLEVEL CHANGE, and the CONCURRENT options. COPY without the CONCURRENT option does not copy empty or unformatted data pages of a LOB table space.

You can also COPY the auxiliary indexes. You can copy both the base table space and the LOB table spaces at the same time to establish a common recoverable point of consistency. To create a common recoverable point of consistency, use SHRLEVEL REFERENCE.

Note: A common recoverable point of consistency is a point where both base table data and LOB data are consistent and to which a point in time recovery can be done.

If you copy a LOB table space that has a base table space with the NOT LOGGED attribute (new DB2 9), copy the base table space and the LOB table space ALWAYS together so that a RECOVER TOLASTCOPY of the entire set results in consistent data across the base table

space and all of the associated LOB table spaces. You cannot copy the base table space or LOB table space with SHRLEVEL CHANGE if the base table space is defined as NOT LOGGED.

If you take an inline image copy of a table with LOB columns (LOAD, REORG), DB2 makes a copy of the base table space, but does not copy the LOB table spaces.

SHRLEVEL options

In general, when you use SHRLEVEL CHANGE for image copies of your LOB table space, somebody can interfere with your job and manipulate the content of the table you are currently backing up. Consider the following example, an image copy job backs up your LOB table space and it takes a couple of minutes, depending on the number and size of your LOBs stored in the auxiliary table. When somebody inserts a LOB value, some pages can be placed in the already backed up area of your table space while other pages can be stored in an area of your LOB table space that your job has not backed up yet. When you have used LOG YES for the definition of the LOB table space, you can acquire a quiesce point after the image copy is taken to create a recoverable point in time. But when you have specified LOG NO for your LOB table space, you can be in trouble if you have to recover to an image copy taken with SHRLEVEL CHANGE.

This means that you are not protected against an inconsistent status of your image copy when using SHRLEVEL CHANGE. We recommend using SHRLEVEL REFERENCE for copying the LOB table space, regardless of whether you take incremental or full image copies.

You cannot copy the base table space or LOB table space with SHRLEVEL CHANGE if the base table space is defined as NOT LOGGED. Only SHRLEVEL REFERENCE is allowed to be able to create a recoverable point. If you try SHRLEVEL CHANGE, you get the message:

```
DSNU447I - COPY SHRLEVEL CHANGE OF TABLESPACE ..... IS NOT ALLOWED BECAUSE IT  
HAS A LOGGING ATTRIBUTE OF NOT LOGGED
```

LISTDEF and TEMPLATES

The easiest way to take a copy of the base table space and all of the LOB table spaces and eventually the indexes of the base table and auxiliary indexes is to use TEMPLATES and LISTDEF.

Because a non-partitioned or partitioned base table can contain many auxiliary objects (one LOB table space and one auxiliary index per LOB column and per partition), we recommend that you use a LISTDEF to generate a list of all these objects.

Use the LOB indicator keywords BASE, LOB, or ALL if you want to include the base objects only, the auxiliary objects only, or both in your list. Use the INDEXSPACES COPY YES keyword if you do not want to include the base table indexes for which the COPY YES attribute is disabled. Use the PARALLEL keyword to take the copies in parallel.

If you specify a LISTDEF with SHRLEVEL REFERENCE, all copy data sets have the same START_RBA value in the DB2 catalog table SYSIBM.SYSCOPY, resulting in a common recoverable point of consistency. Use OPTIONS EVENT(ITEMERROR,SKIP) with SHRLEVEL CHANGE if you want to minimize the time an object is put in the utility read/write (UTRW) state (read claim).

CONCURRENT copy

You might be able to gain improved availability by using the concurrent copy function of the DFSMSdss™ component of the Data Facility Storage Management Subsystem (DFSMS). You can subsequently run the DB2 RECOVER utility to restore those image copies and apply

the necessary log records to them to complete recovery. The CONCURRENT option of COPY invokes the DFSMSdss concurrent copy.

The COPY utility records the resulting DFSMSdss concurrent copies in the catalog table SYSIBM.SYSCOPY with ICTYPE=F and STYPE=C or STYPE=J. STYPE=C indicates that the concurrent copy was taken of the 'I' instance of the table space. STYPE=J indicates that the concurrent copy was taken of the 'J' instance of the table space.

To obtain a consistent offline backup copy outside of DB2:

1. Start the DB2 objects that are being backed up for read-only access by issuing the command:

```
-START DATABASE(database-name) SPACENAM(table space name) ACCESS(RO)
```

Allowing read-only access is necessary to ensure that no updates to data occur during this procedure.

2. Run QUIESCE with the WRITE(YES) option to quiesce all DB2 objects that are being backed up.
3. COPY CONCURRENT the DB2 objects.
4. Issue the following command to allow transactions to access the data:

```
-START DATABASE(database-name) SPACENAM(table space name)
```

If you specify COPY SHRLEVEL REFERENCE with the CONCURRENT option, and if you want to copy all of the data sets for a list of table spaces to the same dump data set, specify FILTERDDN in your COPY statement. This improves the table space availability, because if you do not specify FILTERDDN, COPY might force DFSMSdss to process the list of table spaces sequentially, which might limit the availability of some of the table spaces that are being copied.

Examples

In Example 6-27, we take a consistent backup for all objects of the table ##T.NORMEN00 as defined in Example 6-1 on page 160. We use the PARALLEL option to speed up the process. The SHRLEVEL REFERENCE results in a common recoverable point of consistency.

Example 6-27 COPY LOB data

```

TEMPLATE TSYSCOPY
    DSN('DB2IM.&SS..&DB..&SN..&IC.&JU(3,5)..#&TI.')
    DISP(MOD,CATLG,CATLG)
LISTDEF MYLIST INCLUDE TABLESPACES TABLE ##T.NORMEN00 ALL
           INCLUDE INDEXSPACES COPY YES TABLE ##T.NORMEN00 ALL
COPY LIST MYLIST FULL YES SHRLEVEL REFERENCE
      PARALLEL COPYDDN(TSYSCOPY)
  
```

As a result, four image copies are taken with the same START_RBA in SYSIBM.SYSCOPY as shown in Example 6-28: One copy for each table space and one copy for each index space.

Example 6-28 Common START_RBA in SYSIBM.SYSCOPY

DBNAME	TSNAME	ICTYPE	START RBA HEX	DSNAME	SHRLEVEL	OTYPE	ICBACKUP
NORMEN00	NORMEN00	F	00005A542C27	DB2IM.DB9B.NORMEN00.NORMEN00.F06208.#230717	R	T	
NORMEN00	NORMLOB	F	00005A542C27	DB2IM.DB9B.NORMEN00.NORMLOB.F06208.#230717	R	T	
NORMEN00	IRNORMEN	F	00005A542C27	DB2IM.DB9B.NORMEN00.IRNORMEN.F06208.#230717	R	I	
NORMEN00	IRN0189G	F	00005A542C27	DB2IM.DB9B.NORMEN00.IRN0189G.F06208.#230717	R	I	

In Example 6-29, we use the CONCURRENT copy option to invoke the DFSMSdss concurrent copy function and create a common recoverable point of consistency.

Example 6-29 CONCURRENT COPY of LOB data

```

TEMPLATE TSYSCOPY
    DSN('DB2IM.&SS..&DB..&SN..&IC.&JU(3,5)..#&TI.')
    DISP(MOD,CATLG,CATLG)
LISTDEF MYLIST INCLUDE TABLESPACES TABLE ##T.NORMEN00 ALL
        INCLUDE INDEXSPACES COPY YES TABLE ##T.NORMEN00 ALL
COPY LIST MYLIST CONCURRENT SHRLEVEL REFERENCE
    COPYDDN(TSYSCOPY)

```

The resulting entries in SYSIBM.SYSCOPY are shown in Example 6-30. They all have the same START_RBA.

Example 6-30 CONCURRENT COPY entries in SYSIBM.SYSCOPY

DBNAME	TSNAME	ICTYPE	START RBA HEX	DSNAME	SHRLEVEL	STYPE	ICBACKUP
NORMEN00	NORMEN00	F	00005A575C9D	DB2IM.DB9B.NORMEN00.NORMEN00.F06208.#234445	R	C	
NORMEN00	NORMLOB	F	00005A575C9D	DB2IM.DB9B.NORMEN00.NORMLOB.F06208.#234445	R	C	
NORMEN00	IRNORMEN	F	00005A575C9D	DB2IM.DB9B.NORMEN00.IRNORMEN.F06208.#234445	R	C	
NORMEN00	IRN0189G	F	00005A575C9D	DB2IM.DB9B.NORMEN00.IRN0189G.F06208.#234445	R	C	

In Example 6-31, we use the FILTERDDN option to copy all of the objects to the same dump data set. We use the &LI. template variable, because &SN. makes little sense here.

Example 6-31 CONCURRENT COPY of LOB data to one dump data set

```

TEMPLATE TSYSCOPY
    DSN('DB2IM.&SS..&DB..&LI..&IC.&JU(3,5)..#&TI.')
    DISP(MOD,CATLG,CATLG)
TEMPLATE TFILTER
    DSN('DB2IM.&SS..&DB..&LI..Z&JU(3,5)..#&TI.')
    DISP(MOD,CATLG,CATLG)
LISTDEF MYLIST INCLUDE TABLESPACES TABLE ##T.NORMEN00 ALL
        INCLUDE INDEXSPACES COPY YES TABLE ##T.NORMEN00 ALL
COPY LIST MYLIST CONCURRENT SHRLEVEL REFERENCE
    COPYDDN(TSYSCOPY) FILTERDDN(TFILTER)

```

If you specify FILTERDDN, the SYSCOPY records for all objects in the list have the same data set name and START_RBA as shown in Example 6-32.

Example 6-32 CONCURRENT COPY entries in SYSIBM.SYSCOPY with FILTERDDN

DBNAME	TSNAME	ICTYPE	START RBA HEX	DSNAME	SHRLEVEL	STYPE	ICBACKUP
NORMEN00	NORMEN00	F	00005A95B112	DB2IM.DB9B.NORMEN00.MYLIST.F06209.#172934	R	C	
NORMEN00	NORMLOB	F	00005A95B112	DB2IM.DB9B.NORMEN00.MYLIST.F06209.#172934	R	C	
NORMEN00	IRNORMEN	F	00005A95B112	DB2IM.DB9B.NORMEN00.MYLIST.F06209.#172934	R	C	
NORMEN00	IRN0189G	F	00005A95B112	DB2IM.DB9B.NORMEN00.MYLIST.F06209.#172934	R	C	

6.5 COPYTOCOPY

COPYTOCOPY can also be run on image copies of LOB table spaces.

The COPYTOCOPY utility makes image copies from an image copy that was taken by the COPY utility. This includes inline copies that the REORG or LOAD utilities make. Starting with either the local primary or recovery-site primary copy, COPYTOCOPY can make up to three copies of one or more of the following types of copies:

- ▶ Local primary
- ▶ Local backup
- ▶ Recovery site primary
- ▶ Recovery site backup

However, you cannot run COPYTOCOPY on concurrent copies.

In Example 6-33, you see how to take a set of recovery site primary image copies starting from the last local primary image copies (because you cannot COPYTOCOPY from concurrent copies, we first had to take a new set of FULL image copies as in Example 6-27 on page 179).

Example 6-33 COPYTOCOPY

```

TEMPLATE TSYSCOPY
      DSN('DB2IM.&SS..&DB..&SN..&IC.&JU(3,5)..#&TI.')
      DISP(MOD,CATLG,CATLG)
LISTDEF MYLIST INCLUDE TABLESPACES TABLE ##T.NORMEN00 ALL
      INCLUDE INDEXSPACES COPY YES TABLE ##T.NORMEN00 ALL
COPYTOCOPY LIST MYLIST FROMLASTCOPY
      RECOVERYDDN(TSYSCOPY)

```

Both sets of primary and recovery site image copies have the same START_RBA as shown in Example 6-34.

Example 6-34 Primary and COPYTOCOPY entries in SYSIBM.SYSCOPY

DBNAME	TSNAME	ICTYPE	START RBA HEX	DSNAME	SHRLEVEL	STYPE	ICBACKUP
-----	-----	-----	-----	-----	-----	-----	-----
NORMEN00	NORMEN00	F	00005A9E3DE6	DB2IM.DB9B.NORMEN00.NORMEN00.F06209.#174842	R		
NORMEN00	NORMLOB	F	00005A9E3DE6	DB2IM.DB9B.NORMEN00.NORMLOB.F06209.#174842	R		
NORMEN00	IRNORMEN	F	00005A9E3DE6	DB2IM.DB9B.NORMEN00.IRNORMEN.F06209.#174842	R		
NORMEN00	IRN0189G	F	00005A9E3DE6	DB2IM.DB9B.NORMEN00.IRN0189G.F06209.#174842	R		
NORMEN00	NORMEN00	F	00005A9E3DE6	DB2IM.DB9B.NORMEN00.NORMEN00.F06209.#174904	R		RP
NORMEN00	NORMLOB	F	00005A9E3DE6	DB2IM.DB9B.NORMEN00.NORMLOB.F06209.#174904	R		RP
NORMEN00	IRNORMEN	F	00005A9E3DE6	DB2IM.DB9B.NORMEN00.IRNORMEN.F06209.#174904	R		RP
NORMEN00	IRN0189G	F	00005A9E3DE6	DB2IM.DB9B.NORMEN00.IRN0189G.F06209.#174904	R		RP

6.6 QUIESCE

The QUIESCE utility establishes a quiesce point for a table space, partition, table space set, or list of table spaces and table space sets. A quiesce point is the current log RBA or log record sequence number (LRSN). QUIESCE then records the quiesce point in the SYSIBM.SYSCOPY catalog table. A successful QUIESCE improves the probability of a successful RECOVER or COPY.

if you want to establish a quiesce point for the point in time recovery of a base table and all of its related LOB objects, you have to look at these objects as a table space set (like tables related with referential integrity).

The Quiesce utility drains all writers on the table space objects. The default is to write the changed pages from the table spaces and index spaces to disk (WRITE YES).

Because a non-partitioned or partitioned base table can contain many auxiliary table space objects (one LOB table space per LOB column and per partition), we recommend that you use a LISTDEF to generate a list of all the table space objects with the ALL LOB indicator keyword, as shown in Example 6-35. This creates a common quiesce point.

Example 6-35 QUIESCE a base table space and all LOB table spaces

```
LISTDEF MYLIST INCLUDE TABLESPACES TABLE ##T.NORMEN00 ALL
QUIESCE LIST MYLIST
```

The resulting SYSIBM.SYSCOPY entries are shown in Example 6-36. STYPE=W means that the QUIESCE was done with WRITE(YES). All table spaces have the same START_RBA resulting in a common quiesce point.

Example 6-36 QUIESCE entries in SYSIBM.SYSCOPY

DBNAME	TSNAME	ICTYPE	START RBA HEX	STYPE	OTYPE
-----	-----	-----	-----	-----	-----
NORMEN00	NORMEN00	Q	00005AABD468	W	T
NORMEN00	NORMLOB	Q	00005AABD468	W	T
NORMEN00	IRNORMEN	Q	00005AABD468	W	I
NORMEN00	IRN0189G	Q	00005AABD468	W	I

A valid alternative to create a common quiesce point is to use the TABLESPACESET keyword with the base table space or one of the involved LOB table spaces as shown in Example 6-37.

Example 6-37 QUIESCE a table space set

```
QUIESCE TABLESPACESET NORMEN00.NORMEN00
```

A common quiesce point is a common recoverable point of consistency when both the base table space and LOB table spaces were created as LOG YES (V7 and V8) or LOGGED (DB2 9). Be aware that to recover to a common quiesce point is not being able to recover the LOBS if the LOB table is defined as LOG NO (V7 and V8) or NOT LOGGED (DB2 9), because of the missing log records. Also, if the base table space is defined as NOT LOGGED, the quiesce point is not a common recoverable point of consistency. See “Recovering to a prior point in time” on page 196 for more details.

6.7 REPORT

The REPORT utility provides information about table spaces, tables, and indexes. Use REPORT TABLESPACESET to find the names of all the table spaces and tables in a referential structure, including LOB table spaces. The REPORT utility also provides information about the LOB table spaces that are associated with a base table space. Use REPORT RECOVERY to find information that is necessary for recovering a base table space and its indexes, and LOB table space and its indexes.

The REPORT TABLESPACESET is also useful when the base and LOB table spaces have been created by using the automatic creation of objects (DB2 9 only).

In Example 6-38 on page 183, we show how to use the REPORT TABLESPACESET on the underlying table spaces of the table ##T.NORMEN00 defined in Example 6-1 on page 160. Because the REPORT TABLESPACESET command does not support the use of a LISTDEF, the name of the base table space must first be retrieved from SYSIBM.SYSTABLES.

Example 6-38 REPORT TABLESPACESET on automatic created objects

```
REPORT TABLESPACESET TABLESPACE DSN00030.NORMEN01
```

The resulting report is shown in Example 6-39.

Example 6-39 REPORT TABLESPACESET report

```
DSNU000I   209 15:06:45.13 DSNUGUTC - OUTPUT START FOR UTILITY, UTILID = REPORT.NORMEN00
DSNU1044I   209 15:06:45.19 DSNUGTIS - PROCESSING SYSIN AS EBCDIC
DSNU050I   209 15:06:45.19 DSNUGUTC - REPORT TABLESPACESET TABLESPACE DSN00030.NORMEN01
DSNU587I   -DB9B 209 15:06:45.20 DSNUPSET - REPORT TABLESPACE SET WITH TABLESPACE DSN00030.NORMEN01
```

TABLESPACE SET REPORT:

```
TABLESPACE      : DSN00030.NORMEN01
TABLE           : ##T.NORMEN01
```

LOB TABLESPACE SET REPORT:

```
TABLESPACE      : DSN00030.NORMEN01

BASE TABLE     : ##T.NORMEN01
PART: 0001      COLUMN : IMAGE
LOB TABLESPACE : DSN00030.L96UX60E
AUX TABLE      : ##T.NORMEIMAGE96UXYNM9
AUX INDEXSPACE  : DSN00030.INORMEIM
AUX INDEX       : ##T.INORMEIMAGE96UXYYX
```

```
DSNU580I   209 15:06:45.20 DSNUPORT - REPORT UTILITY COMPLETE - ELAPSED TIME=00:00:00
DSNU010I   209 15:06:45.20 DSNUGBAC - UTILITY EXECUTION COMPLETE, HIGHEST RETURN CODE=0
```

A report on the recovery information about all automatically created objects related to table ##T.NORMEN01 can be obtained as shown in Example 6-40.

Example 6-40 REPORT RECOVERY

```
LISTDEF MYLIST INCLUDE TABLESPACES TABLE ##T.NORMEN01 ALL
REPORT RECOVERY TABLESPACE LIST MYLIST INDEX ALL
```

6.8 RUNSTATS

When dealing with LOBs, you can run the RUNSTATS utility against the base table space, the LOB table space, or the index on the auxiliary table. We do not discuss the HISTORY catalog tables in this paragraph, but they contain similar statistics when RUNSTATS is run to collect historical data. You can run RUNSTATS on the auxiliary objects, but the statistics do not have any effect on access paths and there are no column statistics. The statistics can be used to monitor space usage (such as number of extents) and ORGRATIO to schedule a REORG of the LOB table space. ORGRATIO is used for LOBs to indicate the percentage of LOBs that are properly chunked (that is, allocated in contiguous sets of 16 pages). This is explained in more detail at “Monitoring ORGRATIO” on page 191.

RUNSTATS on the base table space

In SYSIBM.SYSCOLUMNS, the cardinality of a LOB column, COLCARDF, is determined by counting only non-null and non-zero length columns. HIGH2KEY and LOW2KEY are *not* applicable for the LOB column and contain blanks.

In SYSIBM.SYSCOLSTATS, the cardinality of a LOB column, COLCARDF, is determined by counting only non-null and non-zero length columns. HIGHKEY, LOWKEY, HIGH2KEY, and LOW2KEY are not applicable for the LOB column and contain blanks.

RUNSTATS on the index of the auxiliary table

For an index on the auxiliary table, the CARDF column of SYSIBM.SYSINDEXPART indicates the number of keys in the index which refer to LOBs. Also LEAFNEAR, LEAFFAR, and PSEUDO_DEL_ENTRIES contain meaningful information. The statistics CLUSTERED, CLUSTERRATIO, NEAROFFPOSF, FAROFFPOSF, and LEAFDIST in SYSIBM.SYSINDEXPART are not applicable for the index on the auxiliary table.

RUNSTATS on the LOB table space

You can run RUNSTATS on a LOB table space to collect space statistics so that you can determine when the LOB table space should be reorganized. The statistics on the LOB table do not affect access path selection and are not taken into account by the optimizer.

Statistics in SYSIBM.SYSTABLES are not updated except for CARDF and SPACEF, which contain the number of LOBs in the auxiliary table and the number of KB. The statistics in SYSIBM.SYSTABLEPART that are updated are CARDF, SPACEF, PQTY, SQTY, DSNUM, and EXTENTS. PERCACTIVE contains -2 to indicate that this field is not updated for auxiliary tables. Statistics in SYSIBM.SYSCOLUMNS are not updated. HIGH2KEY and LOW2KEY contain blanks for the columns of the auxiliary table, and COLCARDF contains -2 to indicate that this field is not updated for auxiliary tables. The statistics for a LOB table space are stored in a new catalog table called SYSIBM.SYSLOBSTATS, which is explained in “SYSIBM.SYSLOBSTATS” on page 214.

If the table space that is specified by the TABLESPACE keyword is a LOB table space, you can specify only the following additional keywords: SHRLEVEL, REPORT, UPDATE, and HISTORY. You cannot specify the TABLE option for a LOB table space, nor options such as SAMPLE, COLUMN(ALL), COLGROUP, or the INDEX options: KEYCARD and FREQVAL.

As with the other utilities, because a non-partitioned or partitioned base table can contain many auxiliary table space objects (one LOB table space per LOB column and per partition), we recommend that you use a LISTDEF to generate a list of all the objects. Because you cannot specify all keywords for a LOB table space, you probably use two LISTDEFs, one for the base objects and one for the auxiliary objects as shown in Example 6-41.

Example 6-41 RUNSTATS on LOB data

```

LISTDEF BASETS INCLUDE TABLESPACES TABLE ##T.NORMEN00 BASE
RUNSTATS TABLESPACE LIST BASETS TABLE (ALL)
      INDEX(ALL KEYCARD FREQVAL NUMCOLS 1 COUNT 10
            FREQVAL NUMCOLS 2 COUNT 10
            FREQVAL NUMCOLS 3 COUNT 10
            FREQVAL NUMCOLS 4 COUNT 10
            FREQVAL NUMCOLS 5 COUNT 10)
      SHRLEVEL CHANGE REPORT YES HISTORY ALL
LISTDEF LOBTS INCLUDE TABLESPACES TABLE ##T.NORMEN00 LOB
RUNSTATS TABLESPACE LIST LOBTS INDEX (ALL)
      SHRLEVEL CHANGE REPORT YES HISTORY ALL

```

When using REPORT YES, you can actually see which statistics are gathered and what their values are.

6.9 REORG

In this section, we discuss the REORG of LOB objects.

As with normal table spaces and indexes, the REORG utility can be used to:

- ▶ Reorg a LOB table space to reclaim fragmented space and improve access performance
- ▶ Reorg an auxiliary index to reclaim fragmented space and improve access performance

For REORG TABLESPACE on a LOB table space, there are two kinds of REORG possible:

- ▶ REORG TABLESPACE SHRLEVEL NONE (this is the only one possible in DB2 V7 and DB2 V8)
- ▶ REORG TABLESPACE SHRLEVEL REFERENCE (new and recommended in DB2 9)

For REORG INDEX on an auxiliary index, three REORG methods are allowed:

- ▶ REORG INDEX SHRLEVEL NONE
- ▶ REORG INDEX SHRLEVEL REFERENCE
- ▶ REORG INDEX SHRLEVEL CHANGE (recommended)

Tip: Remember that it is possible to combine REORG and RUNSTATS in one run using the STATISTICS options in the REORG command.

REORG TABLESPACE SHRLEVEL NONE

This REORG method was introduced with DB2 V6. It is quite different from a normal REORG and has some drawbacks:

- ▶ No access to LOB data is allowed while the REORG executes.
- ▶ It is an inline REORG, which means that LOBs are moved within the existing LOB table space without unload and reload and without delete and define of the underlying VSAM cluster; there is no means of reclaiming physical space from the LOB data set by resizing it. ALTER of PRIQTY and SECQTY values of the LOB table space are not taken into account.
- ▶ Its aim is to store all pages belonging to an individual LOB as contiguous pages (chunking); this results in a trade-off between optimal reorganization and physical space consumption, because it does not always remove all holes between LOBs.
- ▶ It is always LOG YES, resulting in additional logging, particularly for LOB table spaces defined as LOG YES (V7 and V8) or LOGGED (DB2 9). See also “Impact of logging” on page 173.
- ▶ Inline image copy using the COPYDDN keyword is not allowed.
- ▶ The utility is not restartable during REORGLOB phase and the LOB is left in RECP status if a failure happens; a recovery is always needed afterwards in case of a failure.

The main goal of this REORG method is to improve the prefetch performance of LOB data:

- ▶ Make LOB pages as contiguous as possible.
- ▶ Properly chunk all LOBs to reach ORGRATIO = 100.

There is no delete and define done of the primary data set A001, but a delete of the secondary data sets A002 and A003 is done when they are no longer needed after the REORG. With versions prior to DB2 9, you always need to use the RECOVER utility to resize the LOB table space after changing the PRIQTY and SECQTY allocated values and the number of extents.

This REORG method has three utility phases: UTILINIT, REORGLOB, and UTILTERM. During the REORGLOB phase, the LOB table space is rebuilt in place without unloading and reloading the LOB data to and from an external data set.

The utility is not restartable during the REORGLOB phase, and it is left in RECP status if a failure happens. You have to run RECOVER to recover the LOB table space. If the LOB table space is defined with LOG NO (V7 and V8) or NOT LOGGED (DB2 9), then the LOB table space is left in a COPYP status after a successful REORG, and you should take a full image copy to assure recoverability.

No records are inserted in SYSIBM.SYSCOPY for REORG SHRLEVEL NONE.

Tip: With REORG SHRLEVEL NONE, we always recommend that you take a full image COPY SHRLEVEL REFERENCE before and after the REORG to assure recoverability.

As with the other utilities, because a non-partitioned or partitioned base table can contain many auxiliary table space objects (one LOB table space per LOB column and per partition), you can use a LISTDEF to generate a list of all the LOB table spaces. However in this case, this implies a REORG of all the LOB table spaces, and this might not be what you want.

An example of REORG SHRLEVEL NONE is shown in Example 6-42.

Example 6-42 REORG SHRLEVEL NONE of one LOB table space

```
REORG TABLESPACE NORMEN00.NORMLOB LOG YES
SHRLEVEL NONE
```

An example of REORG with a LISTDEF is shown in Example 6-43.

Example 6-43 REORG SHRLEVEL NONE of all LOB objects

```
LISTDEF LOBTS INCLUDE TABLESPACES TABLE ##T.NORMEN00 LOB
REORG TABLESPACE LIST LOBTS LOG YES
SHRLEVEL NONE
```

REORG TABLESPACE SHRLEVEL REFERENCE

This new method for REORG of LOB table spaces was introduced in DB2 9. The original LOB table space is drained of writers; that is, no update access is allowed during the REORG. All LOBs are then extracted from the original data set and inserted into a shadow data set. A new auxiliary index is also built in a shadow data set. Once this is complete, all access to the LOB table space is stopped for a short period while the original data sets are switched with the shadows. At this point, full access to the new data set is allowed. An inline copy is taken to ensure recoverability.

The allocation of the shadow and deallocation of the old original data set follow the same rules that exist today for REORG SHRLEVEL REFERENCE and CHANGE of normal table spaces.

The new method has the following benefits:

- ▶ REORG SHRLEVEL REFERENCE allows full read access to LOB data for the duration of the REORG with the exception of the switch phase during which readers are also drained. Once the switch has occurred, full access is restored.
- ▶ LOBs are reorganized implicitly during LOB allocation to the shadow data set.

- Physical space is reclaimed, because after REORG, the original data set is deleted and the shadow data set is governed by normal space allocation rules. Shadow data sets are allocated according to normal space allocation rules for the LOB table space. Changes made to PRIQTY and SECQTY with ALTER before the REORG are honored. RECOVER is therefore no longer needed to resize a LOB table space but could still be useful as described in “Use of RECOVERY for reallocating LOB table spaces” on page 199.
- As with the pre-DB2 9 REORG of LOB table spaces, the new solution has no dependency on the base table or base table indexes. Therefore, REORG of a LOB table space continues to have no impact on access to base table data.
- LOG YES is not valid for REORG SHRLEVEL REFERENCE of a LOB table space. This results in reduced logging requirements for LOB table spaces with no loss of recoverability.

An example of the differences between REORG SHRLEVEL NONE in DB2 V8 (and prior releases) is shown in Figure 6-1, Figure 6-2 on page 188, and Figure 6-3 on page 188. We assume we have an auxiliary table space for a LOB column and a heavy update workload has taken place since the initial allocation. Note, we are focusing entirely on the auxiliary table space and not the accompanying auxiliary index for this example.

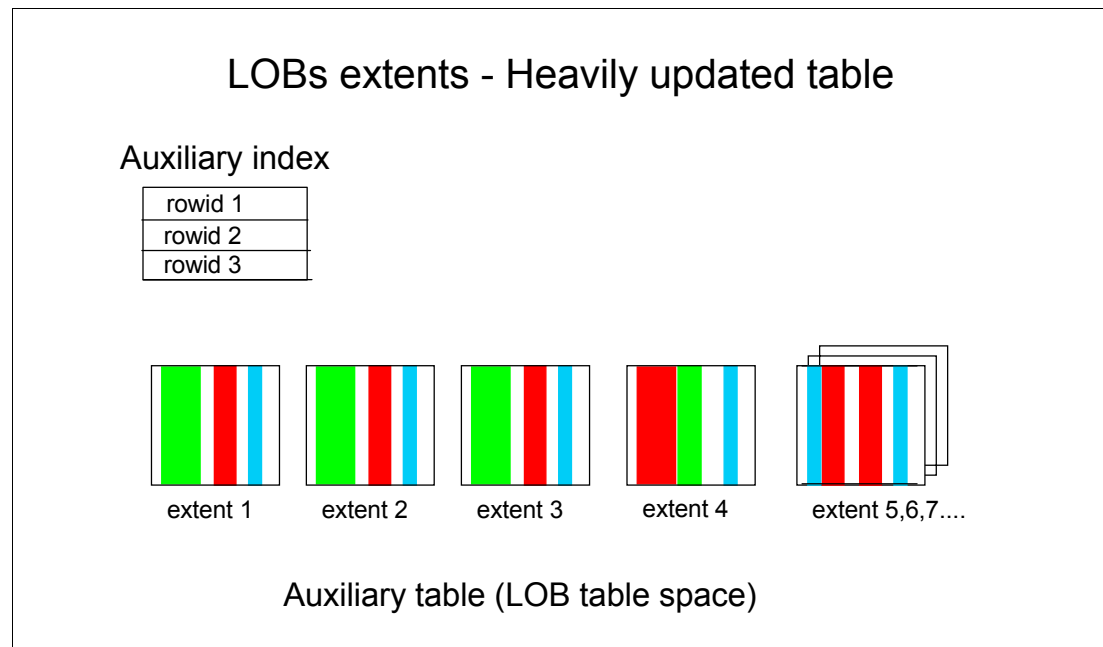


Figure 6-1 Heavily updated LOB table space

This shows a LOB auxiliary table space that has undergone significant change activity, leading both to disorganization of pages, as well as allocation of many extents for the underlying data sets.

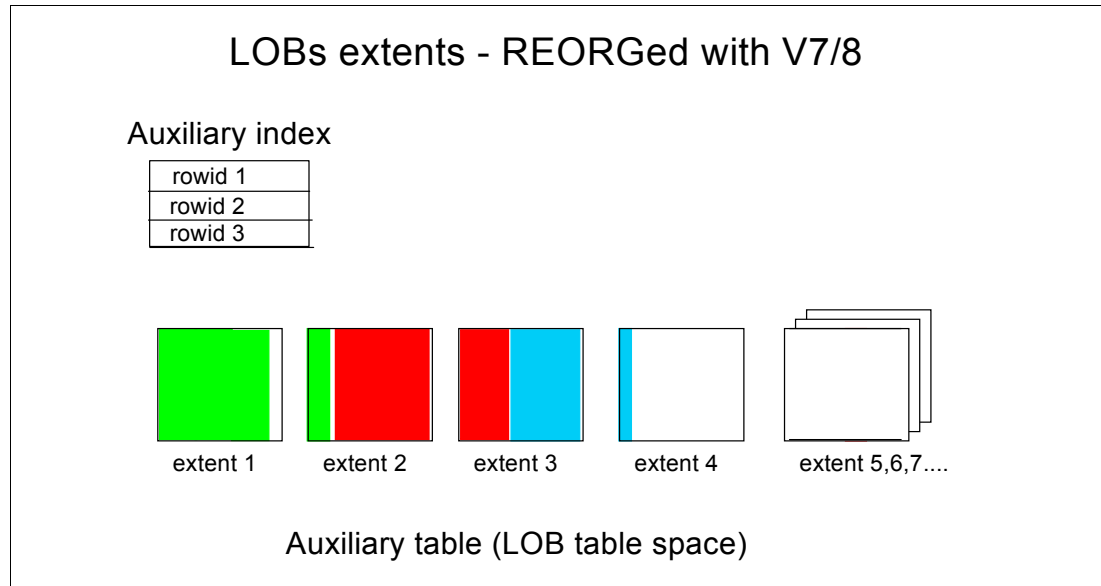


Figure 6-2 LOB Table space REORG with DB2 V8 and prior

Figure 6-2 shows that while the data within the table space has been reordered, and “holes” between objects removed, the physical layout of the data set has not been altered. If there is significant free space, this effectively means a lot of disk space remains unusable, because it is still allocated. The space can be used for subsequent expansion only with data of rows within the table in the auxiliary table space. There is also the residual inefficiency of the data spanning many extents.

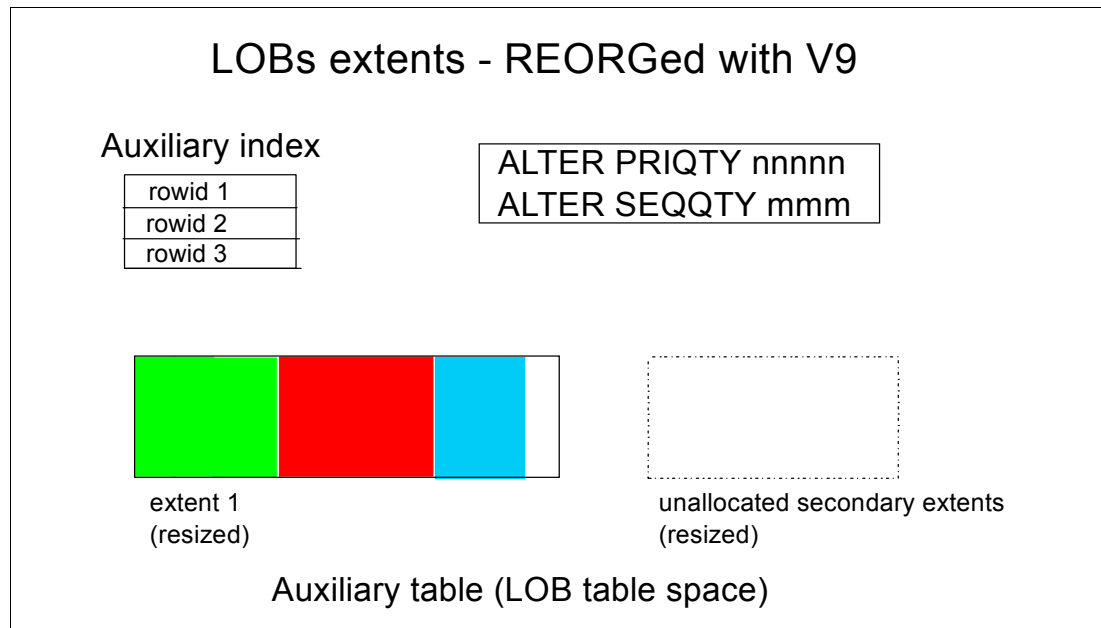


Figure 6-3 LOB Table space REORG with DB2 9

Figure 6-3 shows a table space where REORG SHRLEVEL REFERENCE has been performed. It is assumed that an ALTER of the primary quantity of the table space has been issued before the REORG reflecting the correct allocation for the table space size required for the present data volume, allowing for anticipated growth if desired, and a suitable secondary quantity also determined and ALTERed to take effect.

An inline copy is always taken to ensure recoverability. This is required and the COPYDDN parameter is needed unless a SYSCOPY DD card is specified in the utility JCL. As with a normal REORG SHRLEVEL REFERENCE, the REORG of a LOB table space now inserts two SYSCOPY records. The first SYSCOPY record represents the REORG itself, which is a nonrecoverable event, and the second SYSCOPY record represents the inline image copy.

The switch from original to shadow data sets is a nonrecoverable event requiring an inline copy to be taken during the REORG. During recovery, DB2 cannot apply log records across the switch from the original data set to the shadows. Recovery to a point after the REORG must rely upon the inline copy taken during the REORG process. In the event of a failure during the REORG, the shadow copy is discarded. The utility cannot be restarted.

The implications of using shadow data sets are:

- ▶ Before the new REORG, LOB table spaces and the associated auxiliary indexes could only have an instance of "I".
- ▶ The new REORG causes LOB table space instances and auxiliary index instances to alternate between "I" and "J".
- ▶ Any tools or applications, which assume that LOB table spaces are "I" data sets, have to change to also tolerate "J" data sets.

This REORG method has four utility phases: UTILINIT, REORGLOB, SWITCH, and UTILTERM. During the REORGLOB phase, the LOB table space is unloaded to the shadow data set and any error during this phase leaves the original data set intact. The utility is not restartable during the REORGLOB phase, but the original data set is never put in RECP status if a failure happens. You no longer have to run RECOVER to recover the LOB table space afterwards. The inline image copy assures recoverability.

As with the other utilities, because a non-partitioned or partitioned base table can contain many auxiliary table space objects (one LOB table space per LOB column and per partition), you can use a LISTDEF to generate a list of all the LOB table spaces. However in this case, you do a REORG of all the LOB table spaces, which might not be the intention. An example of REORG SHRLEVEL REFERENCE is shown in Example 6-44.

Example 6-44 REORG SHRLEVEL REFERENCE

```

TEMPLATE TSYSYCOPY
      DSN('DB2IM.&SS..&DB..&SN..&IC.&JU(3,5)..#&TI.')
```

```

      DISP(MOD,CATLG,CATLG)
REORG   TABLESPACE NORMEN00.NORMLOB LOG NO
      COPYDDN(TSYSYCOPY)
      SHRLEVEL REFERENCE
```

The resulting job output is shown in Example 6-45.

Example 6-45 REORG SHRLEVEL REFERENCE output

```

DSNU000I 212 19:44:43.16 DSNUGUTC - OUTPUT START FOR UTILITY, UTILID = REORG.NORMEN00
DSNU1044I 212 19:44:43.22 DSNUGTIS - PROCESSING SYSIN AS EBCDIC
DSNU050I 212 19:44:43.22 DSNUGUTC - TEMPLATE TSYSYCOPY DSN('DB2IM.&SS..&DB..&SN..&IC.&JU(3,5)..#&TI.')
```

```

      DISP(MOD,
      CATLG, CATLG)
DSNU1035I 212 19:44:43.22 DSNUJTDR - TEMPLATE STATEMENT PROCESSED SUCCESSFULLY
DSNU050I 212 19:44:43.23 DSNUGUTC - REORG TABLESPACE NORMEN00.NORMLOB LOG NO COPYDDN(TSYSYCOPY) SHRLEVEL
REFERENCE
DSNU1038I 212 19:44:45.01 DSNUGDYN - DATASET ALLOCATED.  TEMPLATE=TSYSYCOPY
      DDNAME=SYS00001
      DSN=DB2IM.DB9B.NORMEN00.NORMLOB.F06212.#234443
DSNU1151I -DB9B 212 19:45:16.29 DSNURLOB - REORGLOB PHASE COMPLETE - NUMBER OF RECORDS PROCESSED=5883
DSNU387I 212 19:45:16.67 DSNURSWT - SWITCH PHASE COMPLETE, ELAPSED TIME = 00:00:00
DSNU428I 212 19:45:16.68 DSNURSWT - DB2 IMAGE COPY SUCCESSFUL FOR TABLESPACE NORMEN00.NORMLOB
```

```

DSNU400I    212 19:45:16.69 DSNURBID - COPY PROCESSED FOR TABLESPACE NORMEN00.NORMLOB
           NUMBER OF PAGES=50057
           AVERAGE PERCENT FREE SPACE PER PAGE = 0.00
           PERCENT OF CHANGED PAGES =100.00
           ELAPSED TIME=00:00:31
DSNU406I -DB9B 212 19:45:16.58 DSNURWT - FULL IMAGE COPY SHOULD BE TAKEN FOR BOTH LOCAL
           SITE AND RECOVERY SITE FOR TABLESPACE NORMEN00.NORMLOB
DSNU568I -DB9B 212 19:45:17.10 DSNUGSRX - INDEX ##T.I_NORMEN00_AUX IS IN INFORMATIONAL COPY PENDING STATE
DSNU010I    212 19:45:17.11 DSNUGBAC - UTILITY EXECUTION COMPLETE, HIGHEST RETURN CODE=4

```

Message DSNU406I is issued, because previously we took image copies for a recovery site.

An example of REORG with a LISTDEF is shown in Example 6-46.

Example 6-46 REORG SHRLEVEL REFERENCE with LISTDEF

```

LISTDEF LOBTS INCLUDE TABLESPACES TABLE ##T.NORMEN00 LOB
REORG    TABLESPACE LIST LOBTS LOG NO
SHRLEVEL REFERENCE

```

REORG INDEX of an auxiliary index

The REORG INDEX utility statement for an auxiliary index is identical to a REORG INDEX statement of a normal index. See Example 6-47.

Example 6-47 REORG INDEX auxiliary index

```

TEMPLATE TSYSUT1
      DSN('DB2RE.&SS..&DB..&SN..U&JU(3,5)..#&TI.')
      DISP(MOD,DELETE,CATLG)
REORG INDEX ##T.I_NORMEN00_AUX SHRLEVEL CHANGE
      MAXRO 20 DRAIN ALL
      DRAIN_WAIT 20  RETRY 120 RETRY_DELAY 60 TIMEOUT TERM
      SORTDEVT 3390  SORTNUM 6
      WORKDDN(TSYSUT1)

```

Be aware that REORG TABLESPACE of a LOB table space also rebuilds the auxiliary index, and in this case, there is no need to run a separate REORG INDEX utility on the auxiliary index.

Impact of logging

REORG forces you to specify LOG YES for REORG SHRLEVEL NONE and LOG NO with COPYDDN for SHRLEVEL REFERENCE. A LOB table space can also be defined with LOG YES or LOG NO. In DB2 9 also, the base table space can be defined as LOGGED or NON LOGGED.

Table 6-2 on page 191 shows the effect on logging output and LOB table space in case the base table space is LOGGED.

Table 6-2 Impact of logging if base table space is LOGGED

Impact of logging on REORG			
REORG LOG keyword	LOB table space LOG attribute	What is logged	LOB table space status after utility completes
LOG YES	LOG YES	Control information and LOB data	No pending status
LOG YES	LOG NO	Control information	No pending status
LOG NO	LOG YES	Nothing	No pending status because of inline copy
LOG NO	LOG NO	Nothing	No pending status because of inline copy

If the base table space is defined as NOT LOGGED and the LOB table spaces are defined as LOGGED, the LOB table spaces with the LOGGED logging attribute are changed to NOT LOGGED as well. However, this is recorded in the DB2 catalog so that if the base table space is altered to LOGGED, the LOB table spaces are also changed back to LOGGED. If both base table space and LOB table spaces are NOT LOGGED, the LOG YES attribute of the REORG SHRLEVEL NONE utility is also changed to LOG NO during execution. Nothing is logged.

When to REORG a LOB table space or auxiliary index

There are two reasons for REORGing a LOB table space or auxiliary index:

- ▶ To improve access performance
- ▶ To resize the object and reclaim physical space

The main indicators to monitor are:

- ▶ ORGRATIO and FREESPACE in SYSIBM.SYSLOBSTATS
- ▶ DSNUM and EXTENTS in SYSIBM.SYSTABLEPART
- ▶ DSNUM, EXTENTS, LEAFNEAR, LEAFFAR, and PSEUDO_DEL_ENTRIES in SYSIBM.SYSINDEXPART
- ▶ REORGINSERTS, REORGDELETES, REORGUPDATES, REORGDISORGLob, REORGMASDELETE, and EXTENTS in SYSIBM.SYSTABLESPACESTATS (SYSIBM.TABLESPACESTATS in V7 and V8)

Monitoring ORGRATIO

ORGRATIO is the percent of organization of the LOB table space. A value of 100 indicates perfect organization. A value of 1 indicates that the LOB table space is disorganized. A value of 0 means that the LOB table space is totally disorganized. A LOB table space is considered to be perfectly organized if all the LOB groups of 16 pages (called *chunks*) belonging to a single LOB are stored in contiguous pages.

A LOB column always has an auxiliary index, which locates the LOB within the LOB table space. Access path is not an issue, because LOB access is always done through an index probe using the auxiliary index. However, performance can be affected if LOBs are scattered into more physical pieces than necessary, therefore, involving more I/O to scan and materialize.

Figure 6-4 on page 192 shows that there are four LOBs accessed via the auxiliary index. These LOBs are stored in chunks of pages. A chunk consists of 16 contiguous pages of LOB

data. If the size of a LOB is smaller than a chunk (smaller than 16 pages), then the LOB is expected to fit in one chunk. If the size of the LOB is greater than a chunk (greater than 16 pages), then it is optimized to fit into the minimum number of chunks (LOB size divided by chunk size). If, because of fragmentation within chunks, LOBs are split up to store in more chunks than optimal, the ORGRATIO decreases. In our example, the first part of LOB 1 (accessed by ROWID 1) is stored in chunk 1, the remaining bytes are placed in data pages of chunk number 2.

Because of its size, LOB number 1 could fit into one chunk, but two chunks are used to store its value. This means that one extra chunk is used to store LOB number 1. Because of its size, LOB number 2 needs at least two chunks to be stored, and this is what is used by DB2 to store LOB number 2. So, there are no extra chunks for LOB number 2. The value of LOB number 3 should fit into one chunk and is also stored in one chunk, which means that there are no extra chunks used for its storage. Things are different for LOB number 4. The value of LOB number 4 is expected to fit in two chunks, but DB2 uses three chunks. So there is another extra chunk used for LOB number 4.

In this case, ORGRATIO would be 50, since two out of four LOBs are properly chunked.

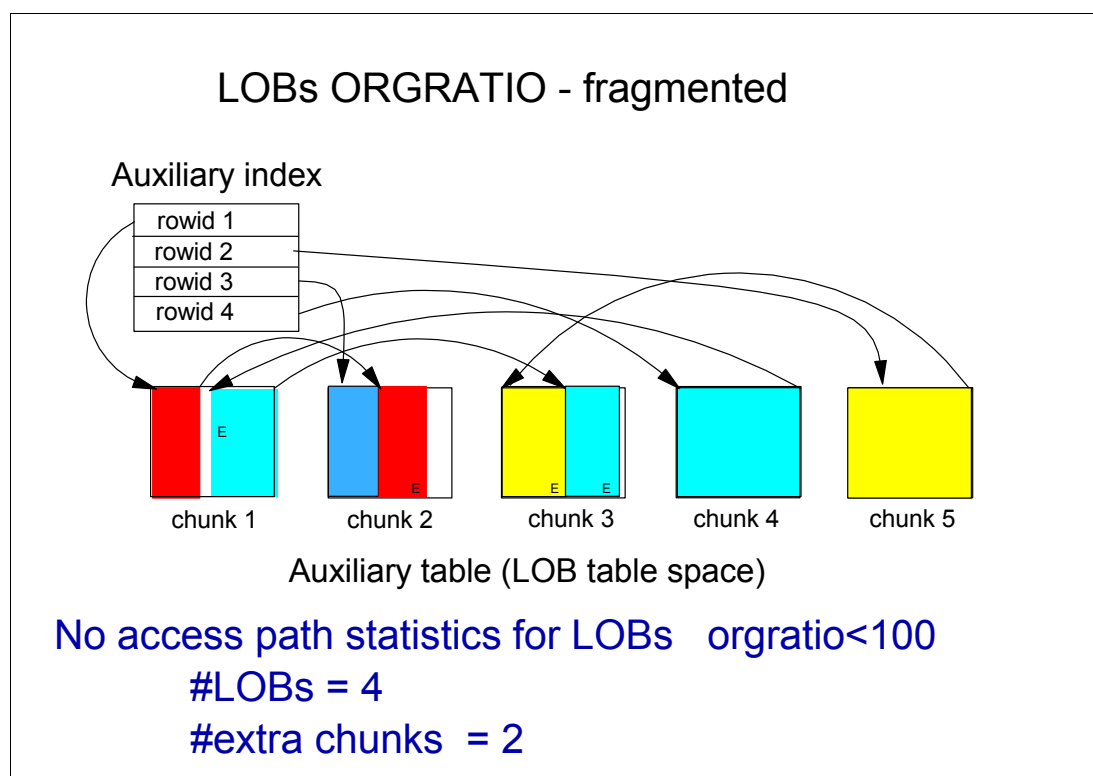


Figure 6-4 Fragmented LOB table space

As shown in Figure 6-5 on page 193, after reorganization of the LOB space, the LOBs are placed in the optimal number of chunks. Since there are no extra chunks allocated, the ORGRATIO is 100.

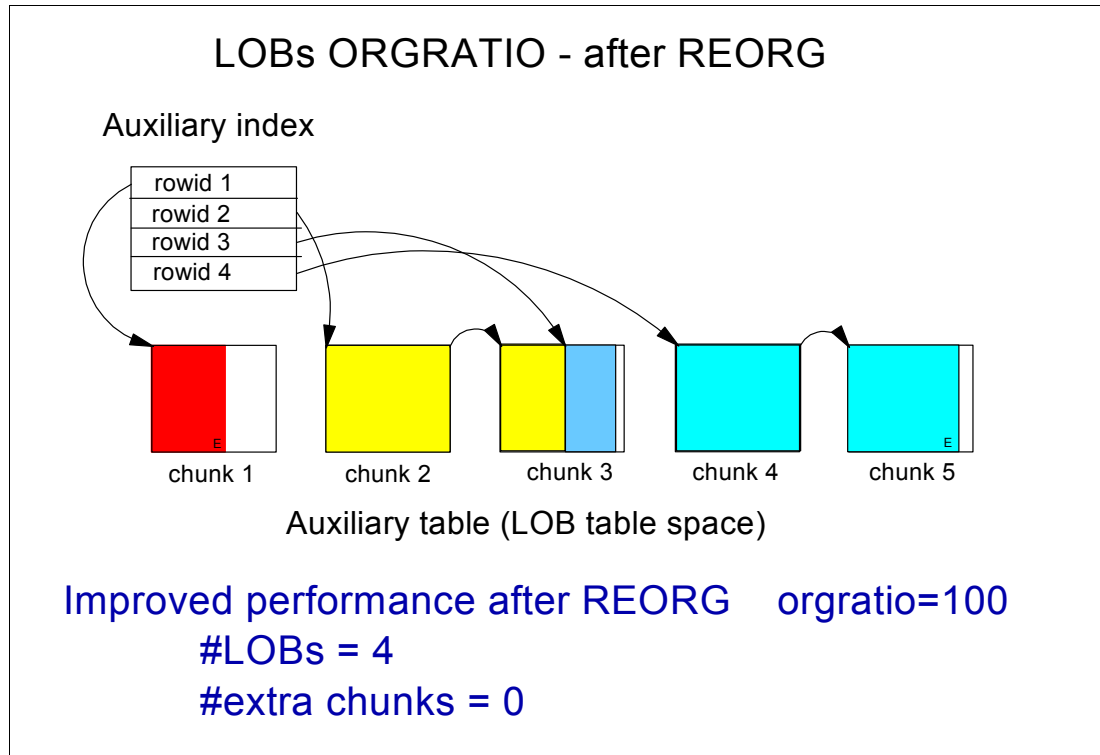


Figure 6-5 Non-fragmented LOB table space

Based on chunking considerations, you could decide to REORG a LOB table space when either:

- ▶ ORGRATIO > 10
- ▶ REORGLDISLOB (number of LOBs inserted since the last REORG that are not perfectly chunked)/TOTALROWS > 10%

The threshold value is set pretty low, because the impact of low ORGRATIO is less than one of the other factors not related to chunking. In the case of a large LOB being read, DB2 uses list prefetch, which always reads pages in ascending page number sequence after sorting. DB2 V8 has removed the 180 CI limit in list prefetch. For small LOBs, which can fit in a page, they are considered perfectly chunked and give ORGRATIO 100.

Note: APAR PK29750 corrects ORGRATIO calculations in DB2 V8.

Monitoring FREESPACE

As explained before, FREESPACE gives the number of kilobytes of free space in extents with respect to high used relative byte address (RBA) or (HURBA). The FREESPACE gives you an indication of how much allocated space is available for more LOBs.

For LOB table spaces, an updated LOB is written without immediately reclaiming the old version of the LOB. The old version of the LOB becomes free space at commit time and when no more readers claim these old versions. The same is true for deleted LOBs.

When FREESPACE approaches zero for a LOB table space, it might be a good idea to resize the LOB table space using REORG SHRLEVEL REFERENCE. FREESPACE can also be used to reduce the size of the LOB table space if a much too large PQTY was used to create the LOB table space or to reclaim physical space after many DELETES. ALTER PQTY followed by REORG SHRLEVEL REFERENCE resizes the LOB table space.

Monitoring Real Time Statistics (RTS)

You could decide to REORG a LOB table space when:

- ▶ REORGDISORGLob (number of LOBs inserted since the last REORG that are not perfectly chunked)/TOTALROWS > 10%
- ▶ REORGMASDELETE > 0

Or use the stored procedures DSNACCOR (DB2 V7 and DB2 V8) or DSNACCOX (planned for DB2 9) to automatically decide when to REORG. The stored procedures used for RTS are described in Appendix B of the *DB2 Version 9.1 for z/OS Utility Guide and Reference*, SC18-9855.

The new formula that is planned for DSNACCOX to execute when deciding to REORG a table space) is listed in Example 6-48.

Example 6-48 RTS DSNACCOX query for REORG TABLESPACE

```
((QueryType='REORG' OR QueryType='ALL') AND
(ObjectType='TS' OR ObjectType='ALL')) AND
(REORGLASTTIME IS NULL AND LOADRLASTTIME IS NULL) OR
(NACTIVE IS NULL OR NACTIVE > 5) AND
(((REORGINSERTS*100)/TOTALROWS>RRTInsertPct) AND
REORGINSERTS>RRTInsertAbs) OR
(((REORGDELETE*100)/TOTALROWS>RRTDeletePct) AND
REORGDELETE>RRTDeleteAbs) OR
(REORGUNCLUSTINS*100)/TOTALROWS>RRTUnclustInsPct OR
(REORGDISORGLob*100)/TOTALROWS>RRTDisorgLOBPct OR
((REORGNearIndRef+REORGFARIndRef)*100)/TOTALROWS>RRTIndRefLimit OR
REORGMASDELETE>RRTMassDelLimit OR
EXTENTS>ExtentLimit)) OR
((QueryType='RESTRICT' OR QueryType='ALL') AND
(ObjectType='TS' OR ObjectType='ALL') AND
The table space is in advisory or informational reorg pending status) OR
((QueryType='RESTRICT' OR QueryType='ALL') AND
(ObjectType='IX' OR ObjectType='ALL')) AND
An index on the tablespace is in advisory-REBUILD-pending stats (ARBBDP)))
```

The new formula that is planned for DSNACCOX to execute when deciding to REORG an index is listed in Example 6-49.

Example 6-49 RTS DSNACCOX query for REORG INDEX

```
((QueryType='REORG' OR QueryType='ALL') AND
(ObjectType='IX' OR ObjectType='ALL')) AND
(REORGLASTTIME IS NULL AND REBUILDLASTTIME IS NULL) OR
(NACTIVE IS NULL OR NACTIVE > 5) AND
(((REORGINSERTS*100)/TOTALENTRIES>RRIInsertPct) AND
REORGINSERTS>RRIInsertAbs) OR
(((REORGDELETE*100)/TOTALENTRIES>RRIDeletePct) AND
REORGDELETE>RRIDeleteAbs) OR
(REORGAPPENDINSERT*100)/TOTALENTRIES>RRIAppendInsertPct OR
```

```
(REORGPSEUDODELETES*100)/TOTALENTRIES>RRIPseudoDeletePct OR
REORGMASDELETE>RRIMassDeleteLimit OR
(REORGLEAFFAR*100)/NACTIVE>RRILeafLimit OR
REORGNUMLEVELS>RRINumLevelsLimit OR
EXTENTS>ExtentLimit)) OR
((QueryType='RESTRICT' OR QueryType='ALL') AND
(ObjectType='IX' OR ObjectType='ALL') AND
An index is in advisory-REBUILD-pending stats (ARBDP)))
```

Monitoring the auxiliary index

You could decide to REORG an auxiliary index when:

- ▶ LEAFFAR (Number of leaf pages located physically far away from previous leaf pages) / NLEAF > 10%
- ▶ PSEUDO_DEL_ENTRIES (entries that are logically deleted but still physically present in the index) / NLEAF > 10%

Or use the stored procedures DSNACCOR (DB2 V7 and DB2 V8) or DSNACCOX (DB2 9) to automatically decide when to REORG as shown in Example 6-48 on page 194 and Example 6-49 on page 194.

When loading LOB data, the auxiliary indexes are not built during the BUILD phase. Instead, LOB values are inserted (not loaded) into auxiliary tables during the RELOAD phase as each row is loaded into the base table, and each index on the auxiliary table is updated as part of the INSERT operation. Because the LOAD utility inserts keys into an auxiliary index, free space within the index might be consumed and index page splits might occur. Consider reorganizing an index on the auxiliary table after the LOAD completes to introduce free space into the index for future INSERTs and LOADs.

Some additional remarks

When using the REORG utility on LOB table spaces, be aware that:

- ▶ You cannot specify the following parameters when REORGing a LOB table space: PART, REBALANCE, SHRLEVEL CHANGE, OFFPOSLIMIT, INDREFLIMIT, UNLOAD, STATISTICS, and SAMPLE.
- ▶ If the partition-by-growth table space contains LOB columns, the REORG TABLESPACE utility minimizes partitions by eliminating existing holes, but does not move the data from one partition to another.
- ▶ When you specify the UNLOAD ONLY option on the base table space, REORG unloads only the data that physically resides in the base table space; LOB columns are not unloaded and you do not get any warning message.
- ▶ When you specify the UNLOAD EXTERNAL option on the base table space, REORG unloads both base columns and LOB columns, but only if the total record length does not exceed 32 KB. Otherwise, you get an error message:
“DSNU297I - COMPOSITE RECORD SIZE TOO LARGE FOR”

6.10 RECOVER and REBUILD

The RECOVER utility recovers data to the current state or to a previous point in time by restoring a copy and then applying log records if available.

In some ways, planning for LOB recovery is similar to that for user-defined referential integrity, because you have to remember that there is a *relationship* between a table with a LOB column and the associated LOB table space. It is true that tables involved in referential integrity relationships must be considered as part of the same table space set for recovery purposes. Similarly, a LOB table space and its associated base table space are part of a table space set. That set, too, is reported using REPORT table space.

For better understanding, it is important to know that each LOB has two invalid flags to mark the LOB as invalid: One in the base table space and one in the LOB table space. The one in the base table space (in the LOB indicator) can be set by the CHECK DATA utility, and the one in the LOB table space (in the LOB space map pages) by RECOVER.

Recovering to a prior point in time

You can recover your LOB table space to a prior point in time, using the TOLOGPOINT, TORBA, TOCOPY, TOLASTCOPY, or TOLASTFULLCOPY option to recover data to a point in time. Ideally, you should recover your base table space and related LOB table spaces together to a common point of consistency:

- ▶ A common set of SHRLEVEL REFERENCE image copies (with the same START_RBA in SYSIBM.SYSCOPY)
- ▶ A common QUIESCE point (with the same START_RBA in SYSIBM.SYSCOPY)

A *common point of consistency* is a common recoverable point of consistency when the RECOVER utility is able to recover all objects using image copies and eventually log records. A common point of consistency is not a common recoverable point of consistency when the RECOVER utility is unable to recover all objects because of missing log records.

See 6.4, “COPY” on page 177 and 6.6, “QUIESCE” on page 181 for how to create a common point of consistency (recoverable or nonrecoverable) for the base table space and LOB table spaces.

If you do not recover the base table space and LOB table spaces together to a common point of consistency, the base table space is marked as *auxiliary check pending* (ACHKP). Furthermore, if the LOB table space is defined as LOG NO (V7 and V8) or NOT LOGGED (DB2 9), all LOBs with missing log records between the last image copy and the recover-to-point are marked *invalid* in the LOB table space (but not in the base table space) and the LOB table space is put in the *auxiliary warning* state (AUXW). Indexes are put in *rebuild pending* (RBDP) or *check pending* state (CHKP) when the indexes are not recovered together with their table spaces to a point in time.

You should use the CHECK LOB utility on the LOB table space to find the invalid LOBs in the LOB table space. You can correct them using the SQL update of the LOB values or SQL delete of the entire rows.

You should use the CHECK DATA utility on the base table space to find and invalidate LOBs in the base table that are not synchronized with the LOB table space. You can correct them using SQL update of the LOB values or SQL delete of the entire rows.

If the base table space is defined as NOT LOGGED (DB2 9), you can only do a point in time recovery to a recoverable point with a RBA or a log record sequence number (LRSN) recorded in SYSIBM.SYSCOPY (for example, the START_RBA of a full SHRLEVEL REFERENCE image copy).

If you specify a TOLOGPOINT or LOGRBA, which is not recorded as a recoverable point in SYSIBM.SYSCOPY, you get this message:

```
DSNU1504I DSNUCASA - RECOVERY OF NOT LOGGED TABLESPACE .....CANNOT PROCEED  
BECAUSE THE TOLOGPOINT OR TORBA SPECIFIED IS NOT A RECOVERABLE POINT.
```

Recovering to the current point in time

Recovering to the current point in time for a base table containing a LOB column is no different than it was in the past. DB2 applies the appropriate image copy to the base table space and reads the log since copy time until present, redoing all the changes. This is also true for the LOB table space.

However, if the LOB table space is defined with LOG NO and log records must be applied to the LOB, the LOB is marked *invalid* and DB2 sets the auxiliary warning state (AUXW) for that LOB table space (The invalid flag is set in the LOB table space but not in the base table space).

You should use the CHECK LOB utility on the LOB table space to find the invalid LOBs in the LOB table space. You should use the CHECK DATA utility on the base table space to find and invalidate LOBs in the base table that are not synchronized with the LOB table space. Afterwards, you can correct them using SQL update of the LOB values or SQL delete of the entire rows.

If the base table space is defined as NOT LOGGED (DB2 9) and you try to recover to the current point in time, the RECOVER utility only recovers to the last recoverable point with the RBA or LRSN recorded in SYSIBM.SYSCOPY (for example, the START_RBA of the last full SHRLEVEL REFERENCE image copy). If changes have been made to the data after this recoverable point, the recover utility is unable to apply them and issues this warning message:

```
DSNU1505I - DSNUCATM - RECOVERY OF NOT LOGGED TABLESPACE ... WAS TO THE LAST  
RECOVERABLE POINT: RBA/LRSN X'.....'. THE OBJECT HAS BEEN CHANGED SINCE  
THAT POINT.
```

And, the base table space is put in the *auxiliary check pending* status (ACHKP).

Recovering LOB pages on the Logical Page List

If there are logical page list (LPL) entries for a LOB table space, then the same procedure has to take place as for regular table spaces (START DATABASE command with the SPACENAM option). But, if the LOB table space is defined with LOG NO or NOT LOGGED and that log data is needed, then check pending (CHKP) is turned on for that table space. Therefore, you have to run the CHECK LOB utility to identify which LOBs are invalid.

New to V9 is that auxiliary index spaces can be put into the LPL and marked RECP or RBDP if the base table space is not logged, and rollback and restart are required.

Use of LISTDEFS for recovery

As with the other utilities, because a non-partitioned or partitioned base table can contain many auxiliary table space objects (one LOB table space per LOB column and per partition), and because each table space can have one or more indexes, you should use LISTDEFS to generate a list of all of the involved objects when trying to RECOVER to keep everything in sync and to avoid having lots of objects in pending states such as CHKP, RECP, RBDP, and so forth, because you did not include all of the objects involved.

For the recovery of the index, you should use RECOVER INDEX or REBUILD INDEX. Use RECOVER INDEX when you regularly take full image copies of all your indexes. Use

REBUILD INDEX if you do not have full image copies of all your indexes or when you take them so infrequently that RECOVER INDEX would take a long time to finish when scanning the log (Remember that DB2 does not support incremental copies of indexes). You can also try to mix RECOVER and REBUILD index, but this can become very confusing at the end.

If you have regular full image copies of all your normal and auxiliary indexes, a typical RECOVERY job could look as shown in Example 6-50.

Example 6-50 Recover table spaces and indexes

```
recovery to the current point in time:
LISTDEF MYLIST INCLUDE TABLESPACES TABLE ##T.NORMEN00 ALL
                INCLUDE INDEXSPACES TABLE ##T.NORMEN00 ALL
RECOVER LIST MYLIST PARALLEL

point in time recovery to previous RBA:
LISTDEF MYLIST INCLUDE TABLESPACES TABLE ##T.NORMEN00 ALL
                INCLUDE INDEXSPACES TABLE ##T.NORMEN00 ALL
RECOVER LIST MYLIST TORBA X'0000729D9B16' PARALLEL (non-data sharing)
or
RECOVER LIST MYLIST TOLOGPOINT X'0000729D9B16' PARALLEL (data sharing)
```

You cannot use the keywords TOLASTCOPY or TOLASTFULLCOPY with LISTDEF. They must be seen as extensions of the TOCOPY *data set name* keyword.

If you do not have regular full image copies of all your normal and auxiliary indexes, you should use the REBUILD INDEX command as shown in Example 6-51.

Example 6-51 Recover table spaces and rebuild indexes

```
recovery to the current point in time:
LISTDEF MYTSLIST INCLUDE TABLESPACES TABLE ##T.NORMEN00 ALL
LISTDEF MYIXLIST INCLUDE INDEXSPACES TABLE ##T.NORMEN00 ALL
RECOVER LIST MYTSLIST PARALLEL
REBUILD INDEX LIST MYIXLIST SORTDEVT 3390 SORTNUM 6

point in time recovery to previous RBA:
LISTDEF MYTSLIST INCLUDE TABLESPACES TABLE ##T.NORMEN00 ALL
LISTDEF MYIXLIST INCLUDE INDEXSPACES TABLE ##T.NORMEN00 ALL
RECOVER LIST MYTSLIST TORBA X'0000729D9B16' PARALLEL
REBUILD INDEX LIST MYIXLIST SORTDEVT 3390 SORTNUM 6
```

REBUILD INDEX SHRLEVEL CHANGE

Starting with DB2 9, you can also use the SHRLEVEL CHANGE option when REBUILDING indexes to keep the table space data available for scanning during the rebuilding of the indexes. This new feature is only allowed when the table spaces are defined as LOGGED, because the logs are applied.

Also since DB2 9, as with the online REORG utility, you can now specify drain options such as DRAIN_WAIT, RETRY, and RETRY_DELAY to have the utility retry the drain operation if it fails (both SHRLEVEL REFERENCE and CHANGE). You can also specify MAXRO, LONGLOG, and DELAY options with SHRLEVEL CHANGE.

Although this feature is not very useful for auxiliary indexes, because the LOB data is always accessed through the auxiliary index, you can still specify it to use the same REBUILD INDEX

syntax for all the indexes (LISTDEF), provided that the LOB table space has the LOGGED attribute.

An example is shown in Example 6-52.

Example 6-52 REBUILD SHRLEVEL CHANGE of the indexes

```
LISTDEF MYIXLIST INCLUDE INDEXSPACES TABLE ##T.NORMEN00 ALL
REBUILD INDEX LIST MYIXLIST
SHRLEVEL CHANGE
SORTNUM 8 SORTDEVT 3390
DRAIN_WAIT 20 RETRY 120 RETRY_DELAY 60
MAXRO 20
```

Use of RECOVERY for reallocating LOB table spaces

The recommended method for resizing LOB table spaces after ALTERing PRIQTY and SECQTY values in DB2 9 is using the REORG SHRLEVEL REFERENCE utility as explained in “REORG TABLESPACE SHRLEVEL REFERENCE” on page 186. We recommend this because you do not have to take image copies before and because the LOB table space is available for read during most of the operation. However in DB2 V7 and V8, you must still use the RECOVERY utility for resizing LOB table spaces.

In practice, a LOB table space consists of many VSAM clusters, A001, A002, A003, and so forth, when the total size is bigger than the DSSIZE. For example, a 400 GB LOB table space with a DSSIZE of 4 GB has 100 VSAM clusters: a001 up to A100.

If for some reason, you only want to reallocate one of the VSAM data sets, for example, to lower the number of extents or to move it to another disk, it might still be a good idea to use RECOVER instead of REORG. If you want to reallocate one VSAM data set only, without reorganizing internally, use RECOVER with the DSNUM clause instead of reorganizing the whole LOB table space to save elapsed time and needed disk space. You could use a utility command as shown in Example 6-53 for reallocating the A003 cluster. During this operation, the whole LOB table space is unavailable.

Example 6-53 Using RECOVER to reallocate a single VSAM cluster of a LOB table space

```
TEMPLATE TSYSCPP1
  DSN('DB2IM.&SS..&DB..&SN..&IC.&JU(3,5)..#&PA.')
  DISP(MOD,CATLG,CATLG)
COPY TABLESPACE NORMEN00.NORMLOB DSNUM 3 COPYDDN(TSYSCPP1)
  SHRLEVEL REFERENCE FULL YES
RECOVER TABLESPACE NORMEN00.NORMLOB DSNUM 3 TOLASTCOPY
REPAIR OBJECT SET INDEX (##T.I_NORMEN00_AUX) NORBDPEND
REPAIR OBJECT SET TABLESPACE NORMEN00.NORMEN00 NOAUXCHKP
```

The REPAIR statements are necessary to remove the pending states without running additional utilities.

6.11 CHECK DATA

The CHECK DATA utility checks for consistency between a base table space and the corresponding LOB table spaces. Run CHECK DATA after a conditional restart or a point in time recovery where base tables and auxiliary tables might not be synchronized or when the

base table space is in the *auxiliary check pending state* (ACHKP) or *auxiliary warning state* (AUXW).

The CHECK DATA utility can only be run on a base table space, not on a LOB table space.

The CHECK DATA utility accesses the base table space and all related auxiliary indexes of the LOB table spaces. CHECK DATA relies on information in the LOB table space and the auxiliary indexes being correct. It might be necessary to first run CHECK LOB on the LOB table space and CHECK INDEX and REBUILD INDEX on the auxiliary indexes to ensure the validity of the LOB table spaces and auxiliary indexes.

CHECK DATA can be run in two modes:

- ▶ SHRLEVEL REFERENCE: Applications can read but cannot write.
- ▶ SHRLEVEL CHANGE: Applications can read and write (This is new with DB2 9).

CHECK DATA reports the following errors:

▶ Orphan LOBs

An orphan LOB column is a LOB entry found in the auxiliary index but not referenced by the base table space. An orphan can result if you recover the base table space and auxiliary index to a point in time prior to the insertion of the base table row or prior to the definition of the LOB column. An orphan can also result if you recover the LOB table space and auxiliary index to a point in time prior to the deletion of a base table row.

▶ Missing LOBs

A missing LOB column is a LOB referenced by the base table space, but the LOB entry is not in the auxiliary index. A missing LOB can result if you recover the LOB table space and auxiliary index to a point in time when the LOB column is not in the LOB table space. This could be a point in time prior to the first insertion of the LOB into the base table, or when the LOB column is null or has a zero length.

▶ Out-of-sync LOBs

An out-of-sync LOB error occurs when DB2 detects a LOB that is found in both the base table and the auxiliary index, but the LOB entry in the auxiliary index is at a different version. A LOB column is also out-of-sync if the base table LOB column is null or has a zero length, but the LOB is found through the auxiliary index. An out-of-sync LOB can occur anytime you recover the LOB table space and auxiliary index or the base table space to a prior point in time.

▶ Invalid LOBs in the base table space

An invalid LOB is an uncorrected LOB column error found by a previous execution of CHECK DATA AUXERROR INVALIDATE. An invalid LOB in the base table space has the invalid flag set in the base table space.

Note: CHECK DATA does not report invalid LOBs in the LOB table space. CHECK DATA does not invalidate LOBs in the base table space that are marked invalid in the LOB table space. CHECK DATA only inspects the auxiliary index and does not access the LOB table space itself. CHECK LOB must be run on the LOB table space to find invalid LOBs in the LOB table space.

CHECK DATA SHRLEVEL REFERENCE removes the ACHKP and AUXW status on a base table space if no errors are found, and it sets ACHKP or AUXW status on the base table space if there was no pending state before and errors are found. CHECK DATA SHRLEVEL CHANGE never sets or resets the ACHKP or AUXW states.

CHECK DATA fails if the LOB table space is in check pending (CHKP) status or recovery pending (RECP) status or when the auxiliary index is in the rebuild pending (RBDP) status.

You can use the SCOPE AUXONLY to limit the checks for LOBs only; otherwise, the CHECK DATA also checks the referential integrity constraints and table constraints.

With DB2 9, the CHECK DATA keywords AUXERROR and AUXONLY cover both LOB data and XML data.

With DB2 9, as for the online REORG utility, you can specify drain options such as DRAIN_WAIT, RETRY, and RETRY_DELAY to have the utility retry the drain operation if it fails (both SHRLEVEL REFERENCE and CHANGE).

CHECK DATA SHRLEVEL REFERENCE

Depending on the REPORT or INVALIDATE clause specified, the following actions are performed when using CHECK DATA SHRLEVEL REFERENCE:

► With AUXERROR REPORT

If the base table space is not yet in auxiliary check pending (ACHKP) status, DB2 drains all SQL writers and sets the base table space and auxiliary table space in UTRO. The data is still available for SQL readers. If errors are found, then the base table space is set to the auxiliary check pending (ACHKP) status. The whole table, or partition, if the table space is partitioned, is now unavailable for applications. The applications get SQLCODE - 904 resource unavailable reason code 00C900C5. If CHECK DATA encounters no errors and the auxiliary check pending (ACHKP) status was already set, it is reset.

If CHECK DATA encounters only invalid LOB columns, the base table space is set to the auxiliary warning status (AUXW), and the table is available for applications. Orphan LOBs in the LOB table space do not cause the AUXW status to be set. You can use SQL to populate invalid LOBs using update or to delete the entire row; however, any other attempt to access the LOB column results in a -904 SQL return code.

The ACHKP pending state on the base table space can be reset by:

- “Fix” reported LOBs and rerun CHECK DATA.
- Run CHECK DATA with AUXERROR INVALIDATE to invalidate them and make them available for SQL (table space set in AUXW).
- REPAIR SET NOAUXCHKP or START ACCESS(FORCE) of the base table space.

“Fixing” the reported LOBS can be done by using the REPAIR utility or the RECOVER utility to recover the data to another point in time, depending on which is the most appropriate.

► With AUXERROR INVALIDATE

If the base table space is not yet in auxiliary check pending (ACHKP) status, DB2 drains all SQL readers and writers, sets the base table space and auxiliary table space in UTUT, and the data is no longer available for SQL. DB2 invalidates the LOB columns that are in error (it sets the invalid flag in the base table space for the LOB value). DB2 resets the invalid status of LOB columns that have been corrected.

If CHECK DATA encounters no more invalid LOBs, then the table space is reset in no pending state. If invalid LOB columns remain, CHECK DATA sets the base table space to the auxiliary warning (AUXW) status and the table is available for applications. Orphan LOBs in the LOB table space do not cause the AUXW status to be set. You can use SQL to populate invalid LOBs using update or delete the entire row; however, any other attempt to access the column results in a -904 SQL return code. But, the base table and all other non-invalid LOBs are available for applications.

The AUXW pending state on the base table space can be reset by:

- “Fix” invalid LOBs and rerun CHECK DATA.
- REPAIR SET NOAUXWARN or START ACCESS(FORCE).

“Fixing” the reported LOBS can be done by using SQL, the REPAIR utility, or the RECOVER utility to recover the data to another point in time, depending on which is the most appropriate.

SHRLEVEL REFERENCE AND DELETE YES

If you use exception tables for checking referential integrity or table constraint problems, the exception table for the base table must have a similar LOB column and a LOB table space for each LOB column. If an exception is found, DB2 moves the base table row with its LOB column to the exception table, and moves the LOB column into the exception table's LOB table space. If you specify DELETE YES, DB2 deletes the base table row and the LOB column. Exception tables can only be used to remove rows with referential integrity or table constraint errors, not for rows with “bad” LOBs.

CHECK DATA SHRLEVEL CHANGE (DB2 9)

The CHECK DATA SHRLEVEL CHANGE utility leaves the base table space available for applications during its processing by running on a shadow copy of the base table space taken by a flashcopy snapshot. The shadow copy is deleted at the end of the utility (UTILTERM).

Note: For user-managed table spaces, it is your responsibility to create the shadow data sets and delete them afterwards. See the *DB2 Version 9.1 for z/OS Utility Guide and Reference*, SC18-9855, for how to allocate them.

Once CHECK DATA has created the shadow data sets, it drains all writers until the snapshot is complete. All table spaces are in UTRO during that time. You can specify drain options such as DRAIN_WAIT, RETRY, and RETRY_DELAY to have the utility retry the drain operation if it fails. After the snapshot is complete, the table spaces are put in UTRW.

Important: If your DASD hardware does not support the FlashCopy® Version 2 capabilities needed for SHRLEVEL CHANGE, DFSMSDSS uses the standard REPRO utility to copy the shadow data sets and the utility does not fail. However, the UTRO phase, in which SQL writers are not allowed to update the data, is significantly longer.

CHECK DATA SHRLEVEL CHANGE does not set nor reset any table space state. Depending on the REPORT or INVALIDATE clause specified, the following actions are performed when using CHECK DATA SHRLEVEL CHANGE after the snapshot:

► With AUXERROR REPORT

CHECK DATA runs on the snapshot data sets, and during its execution, the base table space remains available for applications if it was not yet in the ACHKP status. If errors are found, the shadow data sets are deleted, but the base table space is not set to the auxiliary check pending (ACHKP) status at the end of the utility. The whole table stays available for applications. If the base table space was already in ACHKP and no errors were found anymore, the ACHKP status is not reset at the end.

► With AUXERROR INVALIDATE

CHECK DATA runs on the snapshot data sets, and during its execution, the base table space remains available for applications if it was not yet in the ACHKP status. DB2 does not invalidate the LOB columns that are in error in the base table space (it does not set the invalid flag in the base table). DB2 does not reset the invalid status of LOB columns that have been corrected in the base table space. However, CHECK DATA generates REPAIR

statements to a PUNCHDDN data set to invalidate the LOBs in the base table space that you can execute later on.

If CHECK DATA does not encounter any more invalid LOBs, the base table space is not reset in no pending state if it was in ACHKP and the table remains unavailable for applications. If invalid LOB columns remain, CHECK DATA does not set the base table space to the auxiliary warning (AUXW) status.

SHRLEVEL CHANGE AND DELETE YES

The CHECK DATA SHRLEVEL CHANGE does not set the base table space in check pending (CHKP) status when referential integrity or table constraint errors are found, because this might disrupt executing applications. Nor can it reset the CHKP status when no more such errors exist. It does not delete rows when DELETE YES is specified but generates REPAIR LOCATE DELETE statements to a PUNCHDDN data set.

Examples

CHECK DATA does not allow the use of LISTDEFS and the name of the base table space must be explicitly specified. If it was automatically created, the name of the base table space must first be retrieved from SYSIBM.SYSTABLES. Examples for SHRLEVEL REFERENCE (V8 syntax) and SHRLEVEL CHANGE (DB2 9 syntax) are shown in Example 6-54.

Example 6-54 Examples of CHECK DATA

```
TEMPLATE TSORTOUT
    DSN('DB2RE.&SS..&DB..&SN..S&JU(3,5)..#&TI.')
    DISP(MOD,DELETE,CATLG)
TEMPLATE TSYSUT1
    DSN('DB2RE.&SS..&DB..&SN..U&JU(3,5)..#&TI.')
    DISP(MOD,DELETE,CATLG)
TEMPLATE TSYSERR
    DSN('DB2RE.&SS..&DB..&SN..E&JU(3,5)..#&TI.')
    DISP(MOD,DELETE,CATLG)
```

```
-- check data shrlevel reference
CHECK DATA TABLESPACE NORMEN00.NORMEN00
SCOPE AUXONLY
AUXERROR    REPORT or INVALIDATE
SORTNUM 8 SORTDEVT 3390
WORKDDN(TSYSUT1,TSORTOUT) ERRDDN(TSYSERR)
```

```
-- check data shrlevel change
TEMPLATE TSYSPUN
    DSN('DB2RE.&SS..&DB..&SN..E&JU(3,5)..#&TI.')
    DISP(MOD,CATLG,CATLG)
CHECK DATA TABLESPACE NORMEN00.NORMEN00
SHRLEVEL CHANGE
SCOPE AUXONLY
AUXERROR    REPORT or INVALIDATE
SORTNUM 8 SORTDEVT 3390
WORKDDN(TSYSUT1,TSORTOUT) ERRDDN(TSYSERR)
DRAIN_WAIT 20  RETRY 120  RETRY_DELAY 60
PUNCHDDN(TSYSPUN)
```

Important: If you want to run CHECK DATA when you suspect LOB errors, and you do not want your base table space to become unavailable for applications when errors are found, run CHECK DATA with SHRLEVEL CHANGE.

6.12 CHECK LOB

The CHECK LOB online utility can be run against a LOB table space to identify any structural defects in the LOB table space and signal any invalid LOB values. Run CHECK LOB when you suspect there might be structural defects or when the LOB table space is in the check pending state (CHKP) or auxiliary warning state (AUXW).

The CHECK LOB utility can only be run on a LOB table space, not on a base table space.

The CHECK LOB utility only accesses the LOB table space. It does not access the base table space or the auxiliary index.

CHECK LOB can be run in two modes:

- ▶ SHRLEVEL REFERENCE: Applications can read but cannot write.
- ▶ SHRLEVEL CHANGE: Applications can read and write (new with DB2 9).

If you plan to run CHECK DATA on a base table space containing at least one LOB column, you might consider the following steps prior to running CHECK DATA to ensure the validity of the LOB table spaces and auxiliary indexes. CHECK DATA relies on information in the LOB table space and the auxiliary indexes being correct. The steps are:

1. Run CHECK LOB on the LOB table space.
2. Run CHECK INDEX on the auxiliary index.
3. Run CHECK INDEX on the base table space indexes.

If the LOB table space is in either the check pending (CHKP) status or recover pending (RECP) status, or if the index on the auxiliary table is in rebuild pending (RBPD) status, CHECK DATA issues an error message and fails.

CHECK LOB reports the following errors:

- ▶ Invalid LOBs

An invalid LOB is the status of a LOB as set by RECOVER when an uncorrected LOB column error is found. CHECK LOB examines the invalid flag in the LOB table space, but it never sets it.

Important: CHECK LOB does not access the base table space. So, missing LOBs are not detected during CHECK LOB.

- ▶ Defective LOBs

A defective LOB is a logically inconsistent LOB with a structural defect (most probably as the result of an operational problem that left the LOB table space in an inconsistent state or as the result of a software defect).

When the LOB table space is in CHKP status or AUXW status, and no errors are found anymore, CHECK LOB SHRLEVEL REFERENCE resets the check pending and auxiliary warning states. CHECK LOB SHRLEVEL CHANGE does NOT reset these pending states.

When the LOB table space is in no pending state and errors are found, the CHECK LOB SHRLEVEL REFERENCE sets the CHKP or AUXW pending states, but CHECK LOB SHRLEVEL CHANGE does not.

CHECK LOB fails if the LOB table space is in recovery pending (RECP) status. You must first run the RECOVER utility on the LOB table space.

Since DB2 9, as with the online REORG utility and CHECK DATA utility, you can now specify drain options such as DRAIN_WAIT, RETRY, and RETRY_DELAY to have the utility retry the drain operation if it fails (both SHRLEVEL REFERENCE and CHANGE).

CHECK LOB SHRLEVEL REFERENCE

The following actions are performed when using CHECK LOB SHRLEVEL REFERENCE:

- ▶ DB2 drains all SQL readers and writers and set the LOB table space in check pending (CHKP) status, which makes it unavailable for all applications. The LOB table space is put in UTRO status, and the applications get SQLCODE- 904 resource unavailable with reason code 00C900A3.
- ▶ CHECK LOB issues message DSNU743I whenever it finds a LOB value that is invalid in the LOB table space. The violation is identified by the ROWID and version number of the LOB, a reason code for the error, and the page number where the error was found. Other messages are issued when it detects a defective LOB.
- ▶ If CHECK LOB encounters no more invalid LOBs and no other errors, the LOB table space is reset in no pending state. If invalid LOB columns remain, CHECK LOB sets the LOB table space to the auxiliary warning (AUXW) status and the table is available for applications.
- ▶ You can use SQL to populate an invalid LOB by updating or deleting the entire row; however, any other attempt to access the column results in a -904 SQL return code (reason code 00C900D0). But the base table and all other non-invalid LOBs are available for applications. If CHECK LOB encountered defective LOBs, the LOB table space is left in CHKP status and is unavailable for applications.

The AUXW or CHKP pending states on the LOB table space can be reset if you:

- ▶ “Fix” invalid LOBs and rerun CHECK LOB.
- ▶ REPAIR SET NOAUXCHKP or START ACCESS(FORCE).
- ▶ REPAIR SET NOCHECKPENDING or START ACCESS(FORCE).

“Fixing” the reported LOBS can be done by using SQL, the REPAIR utility, or the RECOVER utility to recover the data to another point in time, depending on which is the most appropriate.

CHECK LOB SHRLEVEL CHANGE (DB2 9)

The CHECK LOB SHRLEVEL CHANGE utility leaves the LOB table space available for applications during its processing by running on a shadow copy of the LOB table space taken by a flashcopy snapshot. These shadow copies are deleted at the end of the utility (UTILTERM).

Note: For user-managed table spaces, it is your responsibility to create the shadow data sets and delete them afterwards. See *DB2 UDB for z/OS Version 8 Utility Guide and Reference*, SC18-7427, for how to allocate them.

Once CHECK LOB has created the shadow data set, it drains all writers until the snapshot is complete. The LOB table space is in UTRO during that time. You can specify drain options

such as DRAIN_WAIT, RETRY, and RETRY_DELAY to have the utility retry the drain operation if it fails. After the snapshot is complete, the LOB table space is put in UTRW.

Important: If your DASD hardware does not support the FlashCopy Version 2 capabilities needed for SHRLEVEL CHANGE, DFSMSDSS uses the standard REPRO utility to copy the shadow data set and the utility does not fail. However, the UTRO phase, in which SQL writers are not allowed to update the data, is significantly longer.

The following actions are performed when using CHECK LOB SHRLEVEL CHANGE after the snapshot:

- ▶ CHECK LOB runs on the snapshot data set, and during its execution, the LOB table space remains available for applications if it was not yet in the CHKP status. CHECK LOB issues message DSNU743I whenever it finds a LOB value with the invalid flag set in the LOB table space. The violation is identified by the ROWID and version number of the LOB, a reason code for the error, and the page number where the error was found.
- ▶ If you provide a SYSPUNCH data set, the CHECK LOB utility generates REPAIR DELETE statements to delete the bad LOBs afterwards. You can also use SQL to update or delete these LOB columns. CHECK LOB SHRLEVEL CHANGE never sets or resets CHKP or AUXW states on the LOB table space.

Examples

CHECK LOB does not allow the use of LISTDEFS and the name of the LOB table space must be explicitly specified. If it was automatically created, the name of the LOB table space must first be retrieved from the DB2 catalog. It resides in the same database as the base table space. Remember that each LOB column and each partition have their own LOB table spaces. Examples for SHRLEVEL REFERENCE (V8 syntax) and SHRLEVEL CHANGE (DB2 9 syntax) are shown in Example 6-55.

Example 6-55 Examples of CHECK LOB

```
-- check LOB shrlevel reference
CHECK LOB TABLESPACE NORMEN00.NORMLOB
SORTNUM 8 SORTDEVT 3390

-- check LOB shrlevel change
TEMPLATE TSYSPPUN
      DSN('DB2RE.&SS..&DB..&SN..E&JU(3,5)..#&TI.')
      DISP(MOD,CATLG,CATLG)
CHECK LOB TABLESPACE NORMEN00.NORMLOB
SHRLEVEL CHANGE
SORTNUM 8 SORTDEVT 3390
DRAIN_WAIT 20 RETRY 120 RETRY_DELAY 60
PUNCHDDN(TSYSPPUN)
```

Important: If you want to run CHECK LOB when you suspect LOB errors and you do not want your LOB table space to become unavailable for applications when errors are found, run CHECK LOB with SHRLEVEL CHANGE.

6.13 CHECK INDEX

The CHECK INDEX online utility can be run against an auxiliary index to verify that each LOB is represented by an index entry. Run CHECK INDEX when you suspect there might be inconsistencies between the LOB table space and the auxiliary index. The CHECK INDEX issues warning messages when inconsistencies are found.

The CHECK INDEX utility accesses the auxiliary index and the LOB table space. It does not access the base table space.

CHECK INDEX can be run in two modes:

- ▶ SHRLEVEL REFERENCE: Applications can read but cannot write LOBs.
- ▶ SHRLEVEL CHANGE: Applications can read and write LOBs.

If you plan to run CHECK DATA on a base table space containing at least one LOB column, you might consider performing the following steps prior to running CHECK DATA to ensure the validity of the LOB table spaces and auxiliary indexes. CHECK DATA relies on information in the LOB table space and the auxiliary indexes being correct. The steps are:

1. Run CHECK LOB on the LOB table space.
2. Run CHECK INDEX on the auxiliary index.
3. Run CHECK INDEX on the base table space indexes.

If the LOB table space is in recover pending (RECP) status, or if the auxiliary index is in rebuild pending (RBPd) status, CHECK INDEX fails.

CHECK INDEX reports the following errors:

- ▶ Missing LOBs in the LOB table space compared to the auxiliary index
- ▶ Missing entries in the auxiliary index compared to the LOB table space

CHECK INDEX does not set pending states, nor does it correct inconsistencies.

Like with the other online utilities, you can specify drain options such as DRAIN_WAIT, RETRY, and RETRY_DELAY to have the utility retry the drain operation if it fails (both SHRLEVEL REFERENCE and CHANGE).

CHECK INDEX SHRLEVEL REFERENCE

The following actions are performed when using CHECK INDEX SHRLEVEL REFERENCE:

- ▶ DB2 drains all SQL writers and set the LOB table space and auxiliary index in UTRO. It then unloads all the entries from the LOB table space and compares them with the entries in the auxiliary index.
- ▶ Invalid entries in the LOB table space can be deleted by using REPAIR LOCATE DELETE specifying the ROWID and VERSION of the LOB or by doing a point in time recovery of the LOB table space to bring it in sync again with the auxiliary index. Invalid entries in the auxiliary index can be removed by doing a REBUILD INDEX of the auxiliary index.

CHECK INDEX SHRLEVEL CHANGE (DB2 9)

This support is also provided in V8 through PQ96956.

The CHECK INDEX SHRLEVEL CHANGE utility leaves the LOB table space and auxiliary index available for SQL writers during its processing by running on a shadow copy of the LOB table space and auxiliary index taken by a flashcopy snapshot. These shadow copies are deleted at the end of the utility (UTILTERM).

Note: For user-managed table spaces, it is your responsibility to create the shadow data sets and delete them afterwards. See *DB2 Version 9.1 for z/OS Utility Guide and Reference*, SC18-9855, for how to allocate them.

Once CHECK INDEX has created the shadow data sets, it drains all writers until the snapshot is complete. The LOB table space and auxiliary index are in UTRW during that time. You can specify drain options such as DRAIN_WAIT, RETRY, and RETRY_DELAY to have the utility retry the drain operation if it fails. After the snapshot is complete, the LOB table space is put in UTRW.

Important: If your DASD hardware does not support the FlashCopy Version 2 capabilities needed for SHRLEVEL CHANGE, DFSMSDSS uses the standard REPRO utility to copy the shadow data sets, and the utility does not fail. However, the UTRW phase, in which SQL writers are not allowed to update the data, is significantly longer.

The following actions are performed when using CHECK INDEX SHRLEVEL CHANGE after the snapshot:

- ▶ CHECK INDEX runs on the snapshot data sets, and during its execution, the LOB table space and auxiliary index remain available for SQL writers. It then unloads all the entries from the LOB table space shadow data set and compares them with the entries in the auxiliary index shadow data set.
- ▶ Invalid entries in the LOB table space can be deleted by using REPAIR LOCATE DELETE specifying the ROWID and VERSION of the LOB or by doing a point in time recovery of the LOB table space to bring it in sync again with the auxiliary index. Invalid entries in the auxiliary index can be removed by doing a REBUILD INDEX.

Examples

CHECK INDEX does allow the use of a LISTDEF if you do not explicitly want to name the auxiliary index. This is very convenient if the auxiliary index was automatically created, because the name must be retrieved from the DB2 catalog. See Example 6-56.

Example 6-56 Example of CHECK INDEX using a LISTDEF

```
LISTDEF MYLIST INCLUDE INDEXSPACES TABLE ##T.NORMEN00 LOB
CHECK INDEX LIST MYLIST
SHRLEVEL      REFERENCE or CHANGE
SORTNUM 8 SORTDEVT 3390
DRAIN_WAIT 20  RETRY 120 RETRY_DELAY 60
```

6.14 REPAIR

The REPAIR utility can be used to:

- ▶ Reset pending states on a base or LOB table space
- ▶ DUMP or DELETE LOBs from the LOB table space
- ▶ Verify and replace the contents of data areas in LOB table spaces and auxiliary indexes (including the invalidation of LOBs by setting the invalid flag)
- ▶ Rebuild object descriptors (OBDs) for a LOB table space

We describe the first two options briefly.

Resetting pending states of a table space

As described before, certain pending states can be set on a base table space or LOB table space. The REPAIR utility can be used to reset these pending states as shown in Table 6-3.

Table 6-3 Resetting pending states using REPAIR

Pending state	Base table space	LOB table space	REPAIR command to reset
ACHKP	Yes	No	SET NOAUXCHKP
AUXW	Yes	Yes	SET NOAUXWARN
CHKP	(Yes) ^a	Yes	SET NOCHECKPEND

a. A base table space can be set in check pending state only for non-LOB related reasons.

Here, we only showed the LOB-related pending states.

LOCATE LOBs to DELETE or DUMP

Use the LOCATE statement with the ROWID and VERSION keywords to locate a LOB in a LOB table space.

When you specify LOCATE ROWID and VERSION for a LOB table space, with the DUMP option, the entire LOB specified is dumped in hexadecimal format. You can use the MAP and DATA keywords to only dump LOB map pages or LOB data pages.

When you specify LOCATE ROWID and VERSION for a LOB table space, with the DELETE option, the entire LOB specified is deleted with its index entry. All pages occupied by the LOB are converted to free space.

Important: The DELETE statement does not remove any reference to the deleted LOB in the base table space.

One LOCATE statement is required for each unit of data to be repaired. Several LOCATE statements can appear after each REPAIR statement.

Typically, the ROWID and VERSION values are displayed in warning or error messages issued by the CHECK DATA or CHECK LOB utilities when reporting orphaned or out-of-sync LOBs.

See Example 6-57 for the syntax of how to DUMP or DELETE a LOB from a LOB table space.

Example 6-57 DUMP or DELETE an entire LOB value

```
REPAIR OBJECT
LOCATE TABLESPACE NORMEN00.NORMLOB
      ROWID X'6B8F05C204CFDE392104015C5630010000000000201'
      VERSION X'0001'
DUMP or DELETE
```

6.15 DSN1COPY and DSN1PRNT

The stand-alone utilities DSN1COPY and DSN1PRNT work on LOB table spaces just as with other types, with the exception that, when working with LOB table spaces, the PARM field LOB must be specified.

LOB specifies that the SYSUT1 data set is a LOB table space. You cannot specify the SEGMENT and INLCOPY options with the LOB parameter.



Data administration with LOBs

In this chapter, we discuss administration information related to table spaces containing tables with LOB columns. We describe the metadata related to LOB objects as contained in the DB2 catalog. We discuss backup and recovery scenarios as well as techniques for altering tables where online schema changes do not apply.

The chapter is structured as follows:

- ▶ LOBs in the DB2 catalog
 - Catalog definitions for LOBs
 - LOBs defined in DB2 catalog
 - Real Time Statistics
- ▶ Recovery strategies and considerations
 - Recovery of LOGGED base table space with LOGGED LOB table space
 - Recovery of LOGGED base table space with NOT LOGGED LOB table space
 - Recovery of NOT LOGGED base table space with NOT LOGGED LOB table space
 - LOBs and SYSTEM RECOVERY
 - Conclusions of recovery of LOB data
- ▶ Altering tables with LOB columns

7.1 LOBs in the DB2 catalog

In this section, you find a description of the major columns that have been provided by DB2 to the system catalog in order to support LOBs. This might help you navigating through the DB2 system catalog searching for LOB-related information. We also briefly describe the LOB tables that are created within the DB2 system catalog and the support for LOB table spaces and auxiliary indexes in the Real Time Statistics database.

7.1.1 Catalog definitions for LOBs

In this section, we discuss the most important catalog tables that contain entries related to LOBs. As an example, we take a look at the entries for table `##T.NORMEN00`, which contains a BLOB column `IMAGE`. The table is used for storing scanned documents in formats such as TIFF, GIF, BMP, PDF, and so forth. The table is created with the DDL listed in Example 7-1. In this example, we have explicitly specified the `ROWID` column. We use the DB2 9 syntax of the `LOG` keyword. The base table space is defined as `LOGGED`, the LOB table space is defined as `NOT LOGGED`. All indexes are defined as `COPY YES`.

Example 7-1 DDL for table `##T.NORMEN00`

```
CREATE DATABASE NORMEN00
  CCSID EBCDIC ;
CREATE TABLESPACE NORMEN00 IN NORMEN00
  USING STOGROUP PAOLOGS
    PRIQTY 100
    SECQTY 28
    ERASE NO
  LOGGED
  GBPCACHE CHANGED
  COMPRESS NO
  BUFFERPOOL BP1
  LOCKSIZE PAGE
  LOCKMAX 0
  CLOSE YES
  SEGSIZE 4
  CCSID EBCDIC
  MAXROWS 255 ;
CREATE TABLE ##T.NORMEN00
  (DOC_ID          VARCHAR(30)          FOR SBCS DATA NOT NULL
   ,PAGE_NUMBER    SMALLINT              NOT NULL
   ,IMPORTER       CHAR(8)              FOR SBCS DATA NOT NULL
   ,IMPORT_TIME    TIMESTAMP              NOT NULL
   ,FORMAT         CHAR(8)              FOR SBCS DATA NOT NULL
   ,ROW_ID         ROWID                  NOT NULL
   ,IMAGE          BLOB(2097152)
   WITH DEFAULT NULL)
  IN NORMEN00.NORMEN00 ;
CREATE UNIQUE INDEX ##T.I_NORMEN00_1
  ON ##T.NORMEN00
  (DOC_ID          ASC,
   PAGE_NUMBER    ASC,
   FORMAT         ASC)
```

```

        USING STOGROUP PAOLOGS
        PRIQTY 12
        SECQTY 12
        ERASE NO
        GBPCACHE CHANGED
        CLUSTER
        BUFFERPOOL BP2
        CLOSE YES
        COPY YES
        PIECESIZE 2 G ;
CREATE LOB TABLESPACE NORMLOB IN NORMEN00
        USING STOGROUP PAOLOGS
        PRIQTY 20000
        SECQTY 5000
        ERASE NO
        GBPCACHE CHANGED
        NOT LOGGED
        DSSIZE 4G
        BUFFERPOOL BP1
        LOCKSIZE LOB
        LOCKMAX 0
        CLOSE YES ;
CREATE UNIQUE INDEX ##T.I_NORMEN00_AUX
        ON ##T.NORMEN00_AUX
        USING STOGROUP PAOLOGS
        PRIQTY 52
        SECQTY 20
        ERASE NO
        GBPCACHE CHANGED
        BUFFERPOOL BP2
        CLOSE YES
        COPY YES
        PIECESIZE 2 G
        DEFINE YES ;

```

SYSIBM.SYSAUXRELS

The table SYSIBM.SYSAUXRELS contains one row per auxiliary table and shows the relationship with the corresponding base table. When you have a partitioned base table space, the partition number is also indicated. The columns of interest are:

- ▶ TBOWNER, authorization ID of the owner of the base table
- ▶ TBNAME, name of the base table
- ▶ COLNAME, name of the LOB column in the base table
- ▶ PARTITION, partition number when the base table space is partitioned, otherwise it is 0
- ▶ AUXTBOWNER, authorization ID of the owner of the auxiliary table
- ▶ AUXTBNAME, name of the auxiliary table
- ▶ RELCREATED, the release of DB2 that was used to create the object (new DB2 9)

Example 7-2 shows what we can find for table ##T.NORMEN00.

Example 7-2 Select from SYSIBM.SYSAUXRELS

TBOWNER	TBNAME	COLNAME	PARTITION	AUXTBOWNER	AUXTBNAME	RELCREATED
-----	-----	-----	-----	-----	-----	-----
##T	NORMEN00	IMAGE	0	##T	NORMEN00_AUX	M

The release indicator is M when the object is created in DB2 9.

SYSIBM.SYSCOLUMNS

The LOB columns defined within a base table can be found in the catalog table SYSIBM.SYSCOLUMNS as listed in Example 7-3.

Example 7-3 Select from SYSIBM.SYSCOLUMNS

NAME	TBNAME	COLTYPE	LENGTH	LENGTH2	HIDDEN
-----	-----	-----	-----	-----	-----
DOC_ID	NORMEN00	VARCHAR	30	0	N
PAGE_NUMBER	NORMEN00	SMALLINT	2	0	N
IMPORTER	NORMEN00	CHAR	8	0	N
IMPORT_TIME	NORMEN00	TIMESTAMP	10	0	N
FORMAT	NORMEN00	CHAR	8	0	N
ROW_ID	NORMEN00	ROWID	17	40	N
IMAGE	NORMEN00	BLOB	4	2097152	N
AUXID	NORMEN00_AUX	VARCHAR	17	0	N
AUXVER	NORMEN00_AUX	SMALLINT	2	0	N
AUXVALUE	NORMEN00_AUX	BLOB	4	2097152	N

The columns of interest are:

- ▶ The TBNAME specified within the SYSIBM.SYSCOLUMNS row indicates the name of the base table, because the LOB column is logically part of the base table.
- ▶ The field COLTYPE in the table is set to indicate the data types of BLOB, CLOB, and DBCLOB.
- ▶ The LENGTH catalog column also is set to a value of four for all LOB columns, because four bytes of internally defined information is stored within each row of the base table for every defined LOB.
- ▶ The LENGTH2 column is set to the actual maximum length of the LOB column.
- ▶ The HIDDEN column contains a value P (partially hidden) when the ROWID column is implicitly generated.

The catalog table SYSIBM.SYSCOLUMNS always contains three column entries: AUXID, AUXVER, and AUXVALUE for an auxiliary table.

SYSIBM.SYSCOLUMNS_HIST

The catalog table SYSIBM.SYSCOLUMNS_HIST contains similar rows for LOB columns as SYSIBM.SYSCOLUMNS when RUNSTATS has been run to collect historical data.

SYSIBM.SYSLOBSTATS

The catalog table SYSIBM.SYSLOBSTATS contains one row for each LOB table space. It holds statistics to manage the space of the LOB table space. It is populated by running RUNSTATS on the LOB table space. The columns of interest are:

- ▶ FREESPACE, kilobytes of free space in extents with respect to high used RBA (HURBA)
- ▶ AVGSIZE, average size of a LOB in bytes
- ▶ ORGRATIO, the percent of organization of the LOB table space. A value of 100 indicates perfect organization. A value of 1 indicates that the LOB table space is disorganized. A value of 0 means that the LOB table space is totally disorganized.

Note: An empty LOB table space has a value of 100 after running RUNSTATS.

After creation and the initial load of the table ##T.NORMEN00, there are no entries in SYSIBM.SYSLOBSTATS. Only after running RUNSTATS on LOB table space NORMEN00.NORMLOB can we find the entries shown in Example 7-4.

Example 7-4 Select from SYSIBM.SYSLOBSTATS

DBNAME	NAME	FREESPACE	AVGSIZE	ORGRATIO
-----	-----	-----	-----	-----
NORMEN00	NORMLOB	68	29920	60.81

SYSIBM.SYSLOBSTATS_HIST

The catalog table SYSIBM.SYSLOBSTATS_HIST contains similar rows as SYSIBM.SYSLOBSTATS for LOB table spaces when RUNSTATS has been run with the option HISTORY SPACE or HISTORY ALL to collect historical data.

SYSIBM.SYSTABLEPART

There are some more fields from SYSIBM.SYSTABLEPART, already used for regular table spaces, that can also be used to manage your LOB table spaces:

- ▶ CARDF, the number of LOBs in the LOB table space
- ▶ SPACEF, KB of space allocated
- ▶ DSNUM, number of linear data sets for the LOB table space
- ▶ EXTENTS, number of extents of the last DSNUM of the LOB table space

The results after running RUNSTATS on all table spaces of database NORMEN00 are shown in Example 7-5.

Example 7-5 Select from SYSIBM.SYSTABLEPART

DBNAME	TSNAME	CARDF	SPACEF	DSNUM	EXTENTS
-----	-----	-----	-----	-----	-----
NORMEN00	NORMEN00	5.883E+03	1.584E+03	1	2
NORMEN00	NORMLOB	5.883E+03	1.915E+05	1	20

SYSIBM.SYSTABLEPART_HIST

The catalog table SYSIBM.SYSTABLEPART_HIST contains similar rows as SYSIBM.SYSTABLEPART for LOB table spaces when RUNSTATS has been run with the option HISTORY SPACE or HISTORY ALL to collect historical data.

SYSIBM.SYSTABLES

The rows contained in SYSIBM.SYSTABLES describe the base table and the auxiliary table as listed in Example 7-6.

Example 7-6 Select from SYSIBM.SYSTABLES

NAME	CREATOR	TYPE	RECLength	CARDF	NPAGESF	SPACEF
-----	-----	---	-----	-----	-----	-----
NORMEN00	##T	T	93	5.88E+03	1.00E+02	4.00E+02
NORMEN00	##T	X	0	5.88E+03	-1.00E+00	1.92E+05

The fields from SYSIBM.SYSTABLES used to manage your LOB table spaces are:

- ▶ NAME, the name of the base table or auxiliary table
- ▶ CREATOR, the schema of the base table or auxiliary table
- ▶ TYPE, contains T for a base table, X for an auxiliary table
- ▶ RECLENGTH is set to 0 for an auxiliary table. For a base table containing LOB columns, it includes the actual four bytes of internally defined information that are stored within the base table for each defined LOB column.
- ▶ CARDF, total number of rows in the base table or number of LOBs in the auxiliary table
- ▶ NPAGESF, number of pages used by the table (always -1.00E+00 for auxiliary table!)
- ▶ SPACEF, number of KB of disk storage

If some of the auxiliary objects have not been defined for a base table, this is reflected in:

- ▶ STATUS, contains 'I' if the base table definition is complete. The reason the table is incomplete is defined in the TABLESTATUS column.
- ▶ TABLESTATUS, contains 'L' if the base table definition is incomplete, because an auxiliary table or auxiliary index has not been defined for a LOB column.

SYSIBM.SYSTABLES_HIST

The catalog table SYSIBM.SYSTABLES_HIST contains similar rows as SYSIBM.SYSTABLES for base tables and auxiliary tables when RUNSTATS has been run to collect historical data.

SYSIBM.SYSTABLESPACE

The catalog table SYSIBM.SYSTABLESPACE contains one row for each LOB table space. The columns of interest are:

- ▶ NAME, the name of the LOB table space
- ▶ DBNAME, the database containing the base table and auxiliary objects
- ▶ BPOOL, buffer pool used for the LOB table space
- ▶ LOCKRULE, locksize of the LOB table space (can be A, L, S for ANY, LOB, or TABLESPACE)
- ▶ IMPLICIT, whether the LOB table space was created implicitly (can be N or Y)
- ▶ SPACEF, number of KB of disk storage (populated only by STOSPACE utility)
- ▶ LOCKMAX, maximum number of LOB locks per user before escalation to TABLESPACE lock (0 means no escalation, -1 means LOCKMAX SYSTEM)
- ▶ TYPE, always 0 for LOB table space
- ▶ LOG, whether the changes to the LOB table space are logged:
 - Y when LOB table space is defined with LOGGED (DB2 9) or LOG YES (V8)
 - N when LOB table space is defined with NOT LOGGED (DB2 9) or LOG NO (V8)
 - X when base table space is defined with NOT LOGGED and LOB table space is defined with LOGGED (in this case, the changes to the LOB table space are NOT LOGGED because of the base table space having NOT LOGGED - DB2 9 only)

Example 7-7 on page 217 shows the entries we get after running the RUNSTATS STOSPACE utility.

Example 7-7 Select from SYSIBM.SYSTABLESPACE

NAME	DBNAME	BPOOL	LOCKRULE	IMPLICIT	SPACEF	LOCKMAX	TYPE	LOG
-----	-----	-----	-----	-----	-----	-----	----	---
NORMEN00	NORMEN00	BP1	P	N	1.58E+03	0		Y
NORMLOB	NORMEN00	BP1	L	N	1.92E+05	0	0	N

Here it is also interesting to see what happens if we let DB2 create all underlying objects automatically by not specifying a table space for the base table. See Example 7-8 (DB2 9 only).

Example 7-8 DB2 9, automatic creation of objects

```
CREATE TABLE ##T.NORMEN01
  (DOC_ID          VARCHAR(30)      FOR SBCS DATA NOT NULL
  ,PAGE_NUMBER     SMALLINT         NOT NULL
  ,IMPORTER        CHAR(8)          FOR SBCS DATA NOT NULL
  ,IMPORT_TIME     TIMESTAMP        NOT NULL
  ,FORMAT          CHAR(8)          FOR SBCS DATA NOT NULL
  ,IMAGE           BLOB(2097152)    WITH DEFAULT NULL) ;
```

In this case, DB2 creates a base table space and LOB table space in a new database as shown in SYSIBM.SYSTABLESPACE in Example 7-9.

Example 7-9 Select from SYSIBM.SYSTABLESPACE

NAME	DBNAME	BPOOL	LOCKRULE	IMPLICIT	SPACEF	LOCKMAX	TYPE	LOG
-----	-----	-----	-----	-----	-----	-----	----	---
NORMEN01	DSN00030	BP0	R	Y	7.20E+02	0	G	Y
L96UX60E	DSN00030	BP0	A	Y	1.89E+05	-1	0	Y

Both base and LOB table spaces are created with default settings for buffer pool, locksize, lockmax, and logging (and others not shown here).

7.1.2 LOBs defined in DB2 catalog

Even if you have no application using DB2 large objects in your environment, you already have LOBs in the DB2 catalog in DBSNDB06. If you run a SELECT on SYSIBM.SYSAUXRELS using WHERE clause AXTBOWNER = 'SYSIBM', you see the auxiliary tables listed in Example 7-10.

Example 7-10 DB2 catalog query

```
SELECT TBOWNER,TBNAME,COLNAME,AUXTBOWNER,AUXTBNAME,RELCREATED
FROM SYSIBM.SYSAUXRELS WHERE TBOWNER = 'SYSIBM' ;
```

TBOWNER	TBNAME	COLNAME	AUX TBOWNER	AUXTBNAME	REL CREATED
-----	-----	-----	-----	-----	-----
SYSIBM	SYSJARCONTENTS	CLASS_SOURCE	SYSIBM	SYSJARCLASS_SOURCE	
SYSIBM	SYSJAROBJECTS	JAR_DATA	SYSIBM	SYSJARDATA	
SYSIBM	SYSROUTINES	TEXT	SYSIBM	SYSROUTINESTEXT	
SYSIBM	XSROBJECTS	GRAMMAR	SYSIBM	XSROBJECTGRAMMAR	M
SYSIBM	XSROBJECTS	PROPERTIES	SYSIBM	XSROBJECTPROPERTY	M

SYSIBM	XSR	OBJECTCOMPONENTS	COMPONENT	SYSIBM	XSR	COMPONENT	M
SYSIBM	XSR	OBJECTCOMPONENTS	PROPERTIES	SYSIBM	XSR	PROPERTY	M

The first two tables are related to the JAVA stored procedures implementation within DB2:

- ▶ **SYSIBM.SYSJARCLASS_SOURCE** has a 10 MB CLOB for the contents of the class in JAR files and is the auxiliary table for column **CLASS_SOURCE** of catalog table **SYSIBM.SYSJARCONTENTS**.
- ▶ **SYSIBM.SYSJARDATA** has a 100 MB BLOB column for the contents of JAR files and is the auxiliary table for column **JAR_DATA** of table **SYSIBM.SYSJAROBJECTS**.

SYSIBM.SYSROUTINESTEXT contains a 2 MB CLOB to contain the source text of the CREATE statement or ALTER statement with the body for the routine and is the auxiliary table for column **TEXT** of table **SYSIBM.SYSROUTINES** (new with DB2 9).

The **SYSIBM.XSR** tables are the new catalog tables to support storing XML documents in DB2 9.

7.1.3 Real Time Statistics

In this section, we discuss the entries related to LOBs in the Real Time Statistics tables (RTS). The RTS tables can be used to decide when a certain utility such as REORG, RUNSTATS, or COPY should be run on the table space or index space based on certain thresholds set by your installation. For a complete discussion about the use of RTS, we refer to the DB2 manuals.

SYSIBM.SYSTABLESPACESTATS

The RTS table **SYSIBM.SYSTABLESPACESTATS** contains real-time statistics for base and LOB table spaces. For LOB table spaces, you might be interested in columns such as:

- ▶ **REORGINSERTS**, the number of LOBs that have been inserted since the last REORG or LOAD REPLACE
- ▶ **REORGDELETES**, the number of LOBs that have been deleted since the last REORG or LOAD REPLACE
- ▶ **REORGUPDATES**, this value does not include LOB updates, because LOB updates are really deletions followed by insertions
- ▶ **REORGDISORGL**, the number of imperfectly chunked LOBs that were inserted since the last REORG or LOAD REPLACE (a LOB is perfectly chunked if the allocated pages are in the minimum number of chunks)
- ▶ **REORGMASDELETE**, the number of mass deletes on the base table since the last REORG or LOAD REPLACE
- ▶ **STATSINSERTS**, the number of LOBs that have been inserted since the last RUNSTATS
- ▶ **STATSDELETES**, the number of LOBs that have been deleted since the last RUNSTATS
- ▶ **STATSUPDATES**, this value does not include LOB updates, because LOB updates are really deletions followed by insertions
- ▶ **STATSMASDELETE**, the number of mass deletes on the base table since the last RUNSTATS
- ▶ **TOTALROWS**, the total number of LOBs in the LOB table space

Other columns, such as **NACTIVE**, **EXTENTS**, and **DATASIZE** can also be examined to monitor the space statistics of the LOB table space or columns such as **COPYLASTTIME** and

COPYUPDATEDPAGES, to decide when to take a new image copy. See the *DB2 Version 9.1 for z/OS SQL Reference*, SC18-9854, Appendix D, for a complete description of the columns of SYSIBM.SYSTABLESPACESTATS.

SYSIBM.SYSINDEXSPACESTATS

The RTS table SYSIBM.SYSINDEXSPACESTATS contains real-time statistics for normal indexes and auxiliary indexes. There are no special columns dedicated to auxiliary indexes. See the *DB2 Version 9.1 for z/OS SQL Reference*, SC18-9854, Appendix D for a complete description of the columns of SYSIBM.SYSINDEXSPACESTATS.

Important: Before DB2 9, the RTS tables were called SYSIBM.TABLESPACESTATS and SYSIBM.INDEXSPACESTATS. The RTS tables resided in database DSNRTSDB, which had to be installed and activated optionally. Starting with DB2 9, the RTS tables are part of the DB2 catalog in a new table space DSNDB06.SYSRTSTS, and RTS is always enabled.

7.2 Recovery strategies and considerations

In this section, we give some examples of recovery scenarios. We provide special attention to the consequences of using the LOG NO (V7 and V8) or NOT LOGGED (DB2 9) parameter for LOB table spaces and the NOT LOGGED parameter for base table spaces (DB2 9 only). We use a newly created table ##T.NORMEN03. This table is created with the same DDL as the table ##T.NORMEN00 defined in Example 7-1 on page 212 with NORMEN00 replaced everywhere by NORMEN03 but with varying values for the LOGGED parameter of base and LOB table space.

7.2.1 LOGGED base table space with LOGGED LOB table space

In the first scenario, we create both the base and LOB table space as LOGGED and LOAD the data with the same contents as table ##T.NORMEN00 (5,883 rows). We can use UNLOAD+RELOAD, DSNTIAUL+RELOAD, or the cross loader as demonstrated in 6.3, “LOAD” on page 171. We then create a common recoverable point of consistency using the COPY utility with LISTDEF and SHRLEVEL REFERENCE as shown in Example 7-11.

Example 7-11 Creating a common recoverable point of consistency using COPY

```

TEMPLATE TSYSCOPY
    DSN('DB2IM.&SS..&DB..&SN..&IC.&JU(3,5)..#&TI.')
    DISP(MOD,CATLG,CATLG) VOLUMES(SBOX61)
LISTDEF MYLIST INCLUDE TABLESPACES TABLE ##T.NORMEN03 ALL
                INCLUDE INDEXSPACES TABLE ##T.NORMEN03 ALL
COPY LIST MYLIST FULL YES SHRLEVEL REFERENCE
    PARALLEL COPYDDN(TSYSCOPY)

```

We can verify that all objects have the same START_RBA = X'000072988DE6' by looking at SYSIBM.SYSCOPY as shown in Example 7-12.

Example 7-12 Common START_RBA in SYSIBM.SYSCOPY

DBNAME	TSNAME	ICTYPE	START RBA HEX	DSNAME
NORMEN03	NORMEN03	C	0000677E731A	NORMEN03.NORMEN03
NORMEN03	NORMLOB	C	0000677F173F	NORMEN03.NORMLOB
NORMEN03	NORMEN03	Z	0000677FE4CA	NORMEN03.NORMEN03
NORMEN03	NORMEN03	F	000072988DE6	DB2IM.DB9B.NORMEN03.NORMEN03.F06214.#220426
NORMEN03	IRNORMEN	F	000072988DE6	DB2IM.DB9B.NORMEN03.IRNORMEN.F06214.#220426

NORMEN03	NORML0B	F	000072988DE6	DB2IM.DB9B.NORMEN03.NORML0B.F06214.#220426
NORMEN03	IRN010S7	F	000072988DE6	DB2IM.DB9B.NORMEN03.IRN010S7.F06214.#220426

Afterwards, we delete some LOBs using SPUFI as shown in Example 7-13 to create some log activity.

Example 7-13 SPUFI delete

```
DELETE FROM ##T.NORMEN03 WHERE DOC_ID LIKE '%E%';
-----+-----+-----+-----+-----+-----+-----+-----+-----+
DSNE615I NUMBER OF ROWS AFFECTED IS 6
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
-----+-----+-----+-----+-----+-----+-----+-----+-----+
-----+-----+-----+-----+-----+-----+-----+-----+-----+
DSNE617I COMMIT PERFORMED, SQLCODE IS 0
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
-----+-----+-----+-----+-----+-----+-----+-----+-----+
DSNE601I SQL STATEMENTS ASSUMED TO BE BETWEEN COLUMNS 1 AND 72
DSNE620I NUMBER OF SQL STATEMENTS PROCESSED IS 1
DSNE621I NUMBER OF INPUT RECORDS READ IS 1
DSNE622I NUMBER OF OUTPUT RECORDS WRITTEN IS 14
```

We then stopped and started all spaces of database NORMEN03 and deleted the VSAM clusters using ISPF 3.4 as shown in Example 7-14:

Example 7-14 Delete VSAM clusters

delete	DB9BU.DSNDBC.NORMEN03.IRNORMEN.I0001.A001	*VSAM*
=	DB9BU.DSNDBC.NORMEN03.IRN010S7.I0001.A001	*VSAM*
=	DB9BU.DSNDBC.NORMEN03.NORMEN03.I0001.A001	*VSAM*
=	DB9BU.DSNDBC.NORMEN03.NORML0B.I0001.A001	*VSAM*
	DB9BU.DSNDBD.NORMEN03.IRNORMEN.I0001.A001	SB0X49
	DB9BU.DSNDBD.NORMEN03.IRN010S7.I0001.A001	SB0X49
	DB9BU.DSNDBD.NORMEN03.NORMEN03.I0001.A001	SB0X49
	DB9BU.DSNDBD.NORMEN03.NORML0B.I0001.A001	SB0X49

Then, we try to recover the VSAMs back to the current point in time as shown in Example 7-15.

Example 7-15 RECOVER to current point in time

```
LISTDEF MYLIST INCLUDE TABLESPACES TABLE ##T.NORMEN03 ALL
          INCLUDE INDEXSPACES TABLE ##T.NORMEN03 ALL
RECOVER LIST MYLIST PARALLEL
```

The result of the RECOVER is shown in Example 7-16.

Example 7-16 RECOVER to current point in time

```
DSNU000I 214 18:34:30.94 DSNUGTC - OUTPUT START FOR UTILITY, UTILID = RECOV.NORMEN03
DSNU1044I 214 18:34:31.00 DSNUGTIS - PROCESSING SYSIN AS EBCDIC
DSNU050I 214 18:34:31.01 DSNUGUTC - LISTDEF MYLIST INCLUDE TABLESPACES TABLE ##T.NORMEN03 ALL INCLUDE
INDEXSPACES TABLE ##T.NORMEN03 ALL
DSNU1035I 214 18:34:31.01 DSNUILDR - LISTDEF STATEMENT PROCESSED SUCCESSFULLY
DSNU050I 214 18:34:31.01 DSNUGUTC - RECOVER LIST MYLIST PARALLEL
DSNU1033I 214 18:34:31.02 DSNUGULM - PROCESSING LIST ITEM: TABLESPACE NORMEN03.NORMEN03
DSNU1033I 214 18:34:31.02 DSNUGULM - PROCESSING LIST ITEM: TABLESPACE NORMEN03.NORML0B
DSNU1033I 214 18:34:31.02 DSNUGULM - PROCESSING LIST ITEM: INDEXSPACE NORMEN03.IRNORMEN
DSNU1033I 214 18:34:31.02 DSNUGULM - PROCESSING LIST ITEM: INDEXSPACE NORMEN03.IRN010S7
DSNU427I 214 18:34:31.04 DSNUCBMT - OBJECTS WILL BE PROCESSED IN PARALLEL,
          NUMBER OF OBJECTS = 4
```

```

DSNU532I   214 18:34:31.04 DSNUCBMT - RECOVER INDEXSPACE NORMEN03.IRNORMEN  START
DSNU515I   214 18:34:31.04 DSNUCBAL - THE IMAGE COPY DATA SET DB2IM.DB9B.NORMEN03.IRNORMEN.F06214.#220426 WITH
DATE=20060802 AND TIME=180427
IS PARTICIPATING IN RECOVERY OF INDEXSPACE NORMEN03.IRNORMEN
DSNU532I   214 18:34:31.48 DSNUCBMT - RECOVER INDEXSPACE NORMEN03.IRN010S7  START
DSNU515I   214 18:34:31.48 DSNUCBAL - THE IMAGE COPY DATA SET DB2IM.DB9B.NORMEN03.IRN010S7.F06214.#220426 WITH
DATE=20060802 AND TIME=180427
IS PARTICIPATING IN RECOVERY OF INDEXSPACE NORMEN03.IRN010S7
DSNU504I   214 18:34:31.62 DSNUCBRT - MERGE STATISTICS FOR INDEXSPACE NORMEN03.IRNORMEN  -
NUMBER OF COPIES=1
NUMBER OF PAGES MERGED=43
ELAPSED TIME=00:00:00
DSNU532I   214 18:34:31.78 DSNUCBMT - RECOVER TABLESPACE NORMEN03.NORMEN03  START
DSNU515I   214 18:34:31.78 DSNUCBAL - THE IMAGE COPY DATA SET DB2IM.DB9B.NORMEN03.NORMEN03.F06214.#220426 WITH
DATE=20060802 AND TIME=180427
IS PARTICIPATING IN RECOVERY OF TABLESPACE NORMEN03.NORMEN03
DSNU504I   214 18:34:31.98 DSNUCBRT - MERGE STATISTICS FOR INDEXSPACE NORMEN03.IRN010S7  -
NUMBER OF COPIES=1
NUMBER OF PAGES MERGED=65
ELAPSED TIME=00:00:00
DSNU532I   214 18:34:32.17 DSNUCBMT - RECOVER TABLESPACE NORMEN03.NORMLOB  START
DSNU515I   214 18:34:32.17 DSNUCBAL - THE IMAGE COPY DATA SET DB2IM.DB9B.NORMEN03.NORMLOB.F06214.#220426 WITH
DATE=20060802 AND TIME=180440
IS PARTICIPATING IN RECOVERY OF TABLESPACE NORMEN03.NORMLOB
DSNU504I   214 18:34:32.36 DSNUCBRT - MERGE STATISTICS FOR TABLESPACE NORMEN03.NORMEN03  -
NUMBER OF COPIES=1
NUMBER OF PAGES MERGED=102
ELAPSED TIME=00:00:00
DSNU504I   214 18:34:58.85 DSNUCBRT - MERGE STATISTICS FOR TABLESPACE NORMEN03.NORMLOB  -
NUMBER OF COPIES=1
NUMBER OF PAGES MERGED=46308
ELAPSED TIME=00:00:26
DSNU513I   -DB9B 214 18:34:58.88 DSNUCALA - RECOVER UTILITY LOG APPLY RANGE IS RBA 00007299C000 LRSN 00007299C000 TO
RBA 00007299DF72 LRSN 00007299DF72
DSNU1510I  214 18:34:58.90 DSNUCBLA - LOG APPLY PHASE COMPLETE, ELAPSED TIME = 00:00:00
DSNU500I   214 18:34:59.03 DSNUCBDR - RECOVERY COMPLETE, ELAPSED TIME=00:00:28
DSNU010I   214 18:34:59.04 DSNUGBAC - UTILITY EXECUTION COMPLETE, HIGHEST RETURN CODE=0

```

As a result, all of the objects have been recovered and the data is in perfect shape again (no pending states) as shown in the output of the -DISPLAY DATABASE(NORMEN03) SPACENAM(*) command in Example 7-17.

Example 7-17 Display database command

```

DSNT360I   -DB9B *****
DSNT361I   -DB9B *   DISPLAY DATABASE SUMMARY
              *   GLOBAL
DSNT360I   -DB9B *****
DSNT362I   -DB9B   DATABASE = NORMEN03  STATUS = RW
              DBD LENGTH = 4028
DSNT397I   -DB9B
NAME      TYPE PART  STATUS              PHYERRLO PHYERRHI CATALOG  PIECE
-----
NORMEN03 TS          RW
NORMLOB  LS          RW
IRN010S7 IX          RW
IRNORMEN IX          RW
***** DISPLAY OF DATABASE NORMEN03 ENDED *****
DSN9022I   -DB9B DSNTDDIS 'DISPLAY DATABASE' NORMAL COMPLETION
***

```

We can now establish a quiesce point using the commands shown in Example 7-18 to create a new common recoverable point of consistency.

Example 7-18 Creating a common recoverable point of consistency using QUIESCE

```
LISTDEF MYLIST INCLUDE TABLESPACES TABLE ##T.NORMEN03 ALL
QUIESCE LIST MYLIST WRITE YES
```

We then delete again some LOBs as shown in Example 7-19. In the QUIESCE job, we get the message:

```
DSNU474I  -DB9B 214 19:00:38.24 DSNUQUIA - QUIESCE AT RBA 0000729D9B16 AND AT
LRSN 0000729D9B16
```

Example 7-19 SPUFI delete

```
DELETE FROM ##T.NORMEN03 WHERE DOC_ID LIKE '%D%' ;
-----+-----+-----+-----+-----+-----+-----+
DSNE615I NUMBER OF ROWS AFFECTED IS 8
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
-----+-----+-----+-----+-----+-----+-----+
-----+-----+-----+-----+-----+-----+-----+
DSNE617I COMMIT PERFORMED, SQLCODE IS 0
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
```

Because both the base table space and LOB table space are created with LOGGED, we have now established two common recoverable points of consistency (one with COPY at LOGRBA = X'00007298DE6' and one with QUIESCE at LOGRBA = X'0000729D9B16') to which we can do a point in time recovery using statements as shown in Example 7-20.

Example 7-20 Point in time recovery to a common recoverable point of consistency

```
LISTDEF MYLIST INCLUDE TABLESPACES TABLE ##T.NORMEN03 ALL
                        INCLUDE INDEXSPACES TABLE ##T.NORMEN03 ALL
RECOVER LIST MYLIST TORBA X'00007298DE6' PARALLEL

LISTDEF MYLIST INCLUDE TABLESPACES TABLE ##T.NORMEN03 ALL
                        INCLUDE INDEXSPACES TABLE ##T.NORMEN03 ALL
RECOVER LIST MYLIST TORBA X'0000729D9B16' PARALLEL
```

An example job output is shown in Example 7-21.

Example 7-21 Point in time recovery to a recoverable quiesce point

```
DSNU000I 214 19:40:30.12 DSNUGUTC - OUTPUT START FOR UTILITY, UTILID = RECOV.NORMEN03
DSNU1044I 214 19:40:30.18 DSNUGTIS - PROCESSING SYSIN AS EBCDIC
DSNU050I 214 19:40:30.18 DSNUGUTC - LISTDEF MYLIST INCLUDE TABLESPACES TABLE ##T.NORMEN03 ALL INCLUDE
INDEXSPACES TABLE ##T.NORMEN03 ALL
DSNU1035I 214 19:40:30.19 DSNUILDR - LISTDEF STATEMENT PROCESSED SUCCESSFULLY
DSNU050I 214 19:40:30.19 DSNUGUTC - RECOVER LIST MYLIST TORBA X'0000729D9B16' PARALLEL
DSNU1033I 214 19:40:30.19 DSNUGULM - PROCESSING LIST ITEM: TABLESPACE NORMEN03.NORMEN03
DSNU1033I 214 19:40:30.19 DSNUGULM - PROCESSING LIST ITEM: TABLESPACE NORMEN03.NORMLOB
DSNU1033I 214 19:40:30.19 DSNUGULM - PROCESSING LIST ITEM: INDEXSPACE NORMEN03.IRNORMEN
DSNU1033I 214 19:40:30.19 DSNUGULM - PROCESSING LIST ITEM: INDEXSPACE NORMEN03.IRN010S7
DSNU427I 214 19:40:30.21 DSNUCBMT - OBJECTS WILL BE PROCESSED IN PARALLEL,
NUMBER OF OBJECTS = 4
DSNU532I 214 19:40:30.21 DSNUCBMT - RECOVER INDEXSPACE NORMEN03.IRNORMEN START
DSNU515I 214 19:40:30.21 DSNUCBAL - THE IMAGE COPY DATA SET DB2IM.DB9B.NORMEN03.IRNORMEN.F06214.#220426 WITH
DATE=20060802 AND TIME=180427
IS PARTICIPATING IN RECOVERY OF INDEXSPACE NORMEN03.IRNORMEN
```

```

DSNU532I  214 19:40:30.81 DSNUCBMT - RECOVER INDEXSPACE NORMEN03.IRNO10S7  START
DSNU515I  214 19:40:30.81 DSNUCBAL - THE IMAGE COPY DATA SET DB2IM.DB9B.NORMEN03.IRNO10S7.F06214.#220426 WITH
DATE=20060802 AND TIME=180427
                IS PARTICIPATING IN RECOVERY OF INDEXSPACE NORMEN03.IRNO10S7
DSNU504I  214 19:40:30.97 DSNUCBRT - MERGE STATISTICS FOR INDEXSPACE NORMEN03.IRNORMEN  -
                NUMBER OF COPIES=1
                NUMBER OF PAGES MERGED=43
                ELAPSED TIME=00:00:00
DSNU532I  214 19:40:31.37 DSNUCBMT - RECOVER TABLESPACE NORMEN03.NORMEN03  START
DSNU515I  214 19:40:31.37 DSNUCBAL - THE IMAGE COPY DATA SET DB2IM.DB9B.NORMEN03.NORMEN03.F06214.#220426 WITH
DATE=20060802 AND TIME=180427
                IS PARTICIPATING IN RECOVERY OF TABLESPACE NORMEN03.NORMEN03
DSNU504I  214 19:40:31.55 DSNUCBRT - MERGE STATISTICS FOR INDEXSPACE NORMEN03.IRNO10S7  -
                NUMBER OF COPIES=1
                NUMBER OF PAGES MERGED=65
                ELAPSED TIME=00:00:00
DSNU532I  214 19:40:31.92 DSNUCBMT - RECOVER TABLESPACE NORMEN03.NORMLOB  START
DSNU515I  214 19:40:31.92 DSNUCBAL - THE IMAGE COPY DATA SET DB2IM.DB9B.NORMEN03.NORMLOB.F06214.#220426 WITH
DATE=20060802 AND TIME=180440
                IS PARTICIPATING IN RECOVERY OF TABLESPACE NORMEN03.NORMLOB
DSNU504I  214 19:40:32.13 DSNUCBRT - MERGE STATISTICS FOR TABLESPACE NORMEN03.NORMEN03  -
                NUMBER OF COPIES=1
                NUMBER OF PAGES MERGED=102
                ELAPSED TIME=00:00:00
DSNU504I  214 19:40:58.89 DSNUCBRT - MERGE STATISTICS FOR TABLESPACE NORMEN03.NORMLOB  -
                NUMBER OF COPIES=1
                NUMBER OF PAGES MERGED=46308
                ELAPSED TIME=00:00:26
DSNU513I  -DB9B 214 19:40:58.93 DSNUCALA - RECOVER UTILITY LOG APPLY RANGE IS RBA 00007299C000 LRSN 00007299C000 TO
                RBA 00007299DF72 LRSN 00007299DF72
DSNU1510I 214 19:40:59.02 DSNUCBLA - LOG APPLY PHASE COMPLETE, ELAPSED TIME = 00:00:00
DSNU535I  -DB9B 214 19:40:59.03 DSNUCATM - FOLLOWING TABLESPACES RECOVERED TO A CONSISTENT POINT
NORMEN03.NORMLOB
DSNU599I  -DB9B 214 19:40:59.03 DSNUCATM - INDEXSPACE NORMEN03.IRNORMEN  HAS BEEN RECOVERED TO A CONSISTENT
                POINT IN TIME WITH TABLESPACE NORMEN03.NORMEN03
DSNU599I  -DB9B 214 19:40:59.03 DSNUCATM - INDEXSPACE NORMEN03.IRNO10S7  HAS BEEN RECOVERED TO A CONSISTENT
                POINT IN TIME WITH TABLESPACE NORMEN03.NORMLOB
DSNU500I  214 19:40:59.15 DSNUCBDR - RECOVERY COMPLETE, ELAPSED TIME=00:00:28
DSNU010I  214 19:40:59.16 DSNUGBAC - UTILITY EXECUTION COMPLETE, HIGHEST RETURN CODE=0

```

We get message DSNU599I, and no objects are put in a pending state. This is the ideal situation, because no additional CHECK utilities must be run.

Tip: The easiest point in time RECOVERY of LOB data is to recover all objects together to a recoverable point of consistency.

In the next step, we try to recover to an in-between log point, which is not a common recoverable point of consistency, specifying TORBA X'00007299D000' in the RECOVER job. This is an RBA in the middle of the delete statement from Example 7-13 on page 220. Here we can see the effect of the new DB2 9 “RECOVER point time with consistency” feature, where DB2 detects that the RBA is in the middle of an active UR and backs out all of the updates of this active UR so that the data is at least consistent from the transaction point of view after the RECOVER.

During the new LOGSCR phase, DB2 reads the log forward from the last checkpoint prior to the recovery point and identifies the URs that were both active (INFLIGHT, INABORT, INDOUBT, or POSTPONED ABORT) during the recovery point and also changed the objects being recovered. During the new LOGUNDO phase, the RECOVER utility backs out the changes made on the recovered objects by the active URs. No objects are put in a pending state. See the new messages DSNU1550I up to DSNU1557I in the job output in Example 7-22 on page 224.

Example 7-22 Point in time recovery with consistency in DB2 9

```

DSNU000I 215 12:59:28.26 DSNUGUTC - OUTPUT START FOR UTILITY, UTILID = RECOV.NORMEN03
DSNU1044I 215 12:59:28.32 DSNUGTIS - PROCESSING SYSIN AS EBCDIC
DSNU050I 215 12:59:28.33 DSNUGUTC - LISTDEF MYLIST INCLUDE TABLESPACES TABLE ##T.NORMEN03 ALL INCLUDE
INDEXSPACES TABLE ##T.NORMEN03 ALL
DSNU1035I 215 12:59:28.33 DSNUIHDR - LISTDEF STATEMENT PROCESSED SUCCESSFULLY
DSNU050I 215 12:59:28.33 DSNUGUTC - RECOVER LIST MYLIST TORBA X'00007299D000' PARALLEL
DSNU1033I 215 12:59:28.34 DSNUGULM - PROCESSING LIST ITEM: TABLESPACE NORMEN03.NORMEN03
DSNU1033I 215 12:59:28.34 DSNUGULM - PROCESSING LIST ITEM: TABLESPACE NORMEN03.NORMLOB
DSNU1033I 215 12:59:28.34 DSNUGULM - PROCESSING LIST ITEM: INDEXSPACE NORMEN03.IRNORMEN
DSNU1033I 215 12:59:28.34 DSNUGULM - PROCESSING LIST ITEM: INDEXSPACE NORMEN03.IRNO10S7
DSNU427I 215 12:59:28.35 DSNUCBMT - OBJECTS WILL BE PROCESSED IN PARALLEL,
        NUMBER OF OBJECTS = 4

DSNU532I 215 12:59:28.35 DSNUCBMT - RECOVER INDEXSPACE NORMEN03.IRNORMEN START
DSNU515I 215 12:59:28.35 DSNUCBAL - THE IMAGE COPY DATA SET DB2IM.DB9B.NORMEN03.IRNORMEN.F06214.#220426 WITH
DATE=20060802 AND TIME=180427
        IS PARTICIPATING IN RECOVERY OF INDEXSPACE NORMEN03.IRNORMEN
DSNU532I 215 12:59:28.94 DSNUCBMT - RECOVER INDEXSPACE NORMEN03.IRNO10S7 START
DSNU515I 215 12:59:28.94 DSNUCBAL - THE IMAGE COPY DATA SET DB2IM.DB9B.NORMEN03.IRNO10S7.F06214.#220426 WITH
DATE=20060802 AND TIME=180427
        IS PARTICIPATING IN RECOVERY OF INDEXSPACE NORMEN03.IRNO10S7
DSNU504I 215 12:59:29.11 DSNUCBRT - MERGE STATISTICS FOR INDEXSPACE NORMEN03.IRNORMEN -
        NUMBER OF COPIES=1
        NUMBER OF PAGES MERGED=43
        ELAPSED TIME=00:00:00
DSNU532I 215 12:59:29.33 DSNUCBMT - RECOVER TABLESPACE NORMEN03.NORMEN03 START
DSNU515I 215 12:59:29.33 DSNUCBAL - THE IMAGE COPY DATA SET DB2IM.DB9B.NORMEN03.NORMEN03.F06214.#220426 WITH
DATE=20060802 AND TIME=180427
        IS PARTICIPATING IN RECOVERY OF TABLESPACE NORMEN03.NORMEN03
DSNU504I 215 12:59:29.53 DSNUCBRT - MERGE STATISTICS FOR INDEXSPACE NORMEN03.IRNO10S7 -
        NUMBER OF COPIES=1
        NUMBER OF PAGES MERGED=65
        ELAPSED TIME=00:00:00
DSNU532I 215 12:59:29.87 DSNUCBMT - RECOVER TABLESPACE NORMEN03.NORMLOB START
DSNU515I 215 12:59:29.87 DSNUCBAL - THE IMAGE COPY DATA SET DB2IM.DB9B.NORMEN03.NORMLOB.F06214.#220426 WITH
DATE=20060802 AND TIME=180440
        IS PARTICIPATING IN RECOVERY OF TABLESPACE NORMEN03.NORMLOB
DSNU504I 215 12:59:30.09 DSNUCBRT - MERGE STATISTICS FOR TABLESPACE NORMEN03.NORMEN03 -
        NUMBER OF COPIES=1
        NUMBER OF PAGES MERGED=102
        ELAPSED TIME=00:00:00
DSNU504I 215 12:59:56.84 DSNUCBRT - MERGE STATISTICS FOR TABLESPACE NORMEN03.NORMLOB -
        NUMBER OF COPIES=1
        NUMBER OF PAGES MERGED=46308
        ELAPSED TIME=00:00:26
DSNU513I -DB9B 215 12:59:56.88 DSNUCALA - RECOVER UTILITY LOG APPLY RANGE IS RBA 00007299C000 LRSN 00007299C000 TO
        RBA 00007299D000 LRSN 00007299D000
DSNU1510I 215 12:59:57.17 DSNUCBLA - LOG APPLY PHASE COMPLETE, ELAPSED TIME = 00:00:00
DSNU1550I -DB9B 215 12:59:57.17 DSNUCALC - LOGCSR IS STARTED FOR MEMBER , PRIOR CHECKPOINT RBA =
X'000072439CFE'
DSNU1551I -DB9B 215 12:59:57.24 DSNUCALC - LOGCSR IS FINISHED FOR MEMBER , ELAPSED TIME = 00:00:00
DSNU1552I -DB9B 215 12:59:57.24 DSNUCALC - LOGCSR PHASE COMPLETE, ELAPSED TIME = 00:00:00
DSNU1553I -DB9B 215 12:59:57.24 DSNUCALC - RECOVER DETECTS THE FOLLOWING ACTIVE URS:
        INFLIGHT = 1, INABORT = 0, INDOUBT = 0, POSTPONED ABORT = 0
        MEM T CONNID CORRID AUTHID PLAN S URID DATE TIME
        B TSO PAOLR2 PAOLR2 DSNESPCS F 00007299C4BC 2006-08-02 22.16.02
        DBNAME SPACENAME DBID/PSID PART RBA
        NORMEN03 NORMLOB 014D/0007 0000 00007299CD90
        NORMEN03 IRNORMEN 014D/0005 0000 00007299C57B
        NORMEN03 NORMEN03 014D/0002 0000 00007299C9A1
DSNU1554I -DB9B 215 12:59:57.52 DSNUCALU - LOGUNDO IS STARTED FOR MEMBER
DSNU1556I -DB9B 215 12:59:57.55 DSNUCALU - LOGUNDO IS FINISHED FOR MEMBER , ELAPSED TIME = 00:00:00
DSNU1557I -DB9B 215 12:59:57.55 DSNUCALU - LOGUNDO PHASE COMPLETE, ELAPSED TIME = 00:00:00
DSNU535I -DB9B 215 12:59:57.56 DSNUCATM - FOLLOWING TABLESPACES RECOVERED TO A CONSISTENT POINT
        NORMEN03.NORMEN03

```

```

NORMEN03.NORMLOB
DSNU599I -DB9B 215 12:59:57.56 DSNUCATM - INDEXSPACE NORMEN03.IRNORMEN HAS BEEN RECOVERED TO A CONSISTENT
POINT IN TIME WITH TABLESPACE NORMEN03.NORMEN03
DSNU599I -DB9B 215 12:59:57.56 DSNUCATM - INDEXSPACE NORMEN03.IRNO10S7 HAS BEEN RECOVERED TO A CONSISTENT
POINT IN TIME WITH TABLESPACE NORMEN03.NORMLOB
DSNU500I 215 12:59:57.67 DSNUCBDR - RECOVERY COMPLETE, ELAPSED TIME=00:00:29
DSNU010I 215 12:59:57.68 DSNUGBAC - UTILITY EXECUTION COMPLETE, HIGHEST RETURN CODE=0

```

After this RECOVERY, the LOB table space contains 5,883 LOBs, which proves that the delete statement has been backed out and is in its initial state after the LOAD utility.

In the next step, we first redo the deletes as in Example 7-13 on page 220 and Example 7-19 on page 222 as shown in Example 7-23. As expected, 14 rows are now deleted.

Example 7-23 SPUFI delete

```

DELETE FROM ##T.NORMEN03 WHERE DOC_ID LIKE '%D%'
OR DOC_ID LIKE '%E%' ;
-----+-----+-----+-----+-----+-----+-----+
DSNE615I NUMBER OF ROWS AFFECTED IS 14
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
-----+-----+-----+-----+-----+-----+-----+
-----+-----+-----+-----+-----+-----+-----+
DSNE617I COMMIT PERFORMED, SQLCODE IS 0
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0

```

We now do a point in time recovery of the base table only to the initial state as shown in Example 7-24, which results in 14 missing LOBs in the LOB table space. This is a situation which often occurs in client sites when the DBA wants to back out the changes of an invalid transaction on the base data but forgets about the LOB table space.

Example 7-24 Point in time recovery of base table only

```

LISTDEF MYLIST INCLUDE TABLESPACES TABLE ##T.NORMEN03 BASE
INCLUDE INDEXSPACES TABLE ##T.NORMEN03 BASE
RECOVER LIST MYLIST TORBA X'000072988DE6' PARALLEL

```

As a result of the point in time recovery, the base table space is put in auxiliary check pending (ACHKP) status as shown in Example 7-25. The base table is unavailable for applications (resource unavailable with reason code 00C900C5).

Example 7-25 Base table space in ACHKP

```

DSNT360I -DB9B *****
DSNT361I -DB9B * DISPLAY DATABASE SUMMARY
* GLOBAL
DSNT360I -DB9B *****
DSNT362I -DB9B DATABASE = NORMEN03 STATUS = RW
DBD LENGTH = 4028
DSNT397I -DB9B
NAME TYPE PART STATUS PHYERRLO PHYERRHI CATALOG PIECE
-----
NORMEN03 TS RW,ACHKP
NORMLOB LS RW
IRNO10S7 IX RW
IRNORMEN IX RW
***** DISPLAY OF DATABASE NORMEN03 ENDED *****
DSN9022I -DB9B DSNTDDIS 'DISPLAY DATABASE' NORMAL COMPLETION

```

To find the bad LOBs, we must now run a CHECK DATA. It makes little sense to run CHECK LOB on the LOB table space here because the LOB table space is not in a pending state, and missing LOBs are not found by the CHECK LOB utility. We first run CHECK DATA with the AUXERROR REPORT as shown in Example 7-26 just to report the bad LOBs. Because the table space is already in ACHKP, we use SHRLEVEL REFERENCE because the table space is already unavailable for applications.

Example 7-26 CHECK DATA SHRLEVEL REFERENCE AUXERROR REPORT

```
TEMPLATE TSORTOUT
  DSN('DB2RE.&SS..&DB..&SN..S&JU(3,5)..#&TI.')
  DISP(MOD,DELETE,CATLG)
TEMPLATE TSYUT1
  DSN('DB2RE.&SS..&DB..&SN..U&JU(3,5)..#&TI.')
  DISP(MOD,DELETE,CATLG)
TEMPLATE TSYERR
  DSN('DB2RE.&SS..&DB..&SN..E&JU(3,5)..#&TI.')
  DISP(MOD,DELETE,CATLG)
CHECK DATA TABLESPACE NORMEN03.NORMEN03
SHRLEVEL REFERENCE
SCOPE AUXONLY
AUXERROR REPORT
SORTNUM 8 SORTDEVT 3390
WORKDDN(TSYUT1,TSORTOUT) ERRDDN(TSYERR)
```

As a result, we get 14 missing LOBs as indicated by message DSNU809I as shown in Example 7-27. The base table space remains in ACHKP and is still unavailable for applications.

Example 7-27 Result of CHECK DATA AUXERROR REPORT

```
.....
DSNU809I    216 14:35:32.93 DSNUKERR - TABLE=##T.NORMEN03 COLUMN=IMAGE IS MISSING IN INDEX ##T.I_NORMEN03_AUX
          ROWID=X'6B8F05C204CFDE392104015C5630010000000000201'
          VERSION=X'0001'
DSNU809I    216 14:35:32.93 DSNUKERR - TABLE=##T.NORMEN03 COLUMN=IMAGE IS MISSING IN INDEX ##T.I_NORMEN03_AUX
          ROWID=X'A9DB85C204CFD7052104015C5630010000000000202'
          VERSION=X'0001'
.....
```

We can also do a REPAIR OBJECT SET TABLESPACE NORMEN03.NORMEN03 NOAUXCHKP to remove the ACHKP state on the base table to make it available immediately for the applications, and run a CHECK DATA SHRLEVEL CHANGE afterwards (DB2 9 only) as shown in Example 7-28.

Example 7-28 CHECK DATA SHRLEVEL CHANGE AUXERROR REPORT

```
.....
CHECK DATA TABLESPACE NORMEN03.NORMEN03
SHRLEVEL CHANGE
SCOPE AUXONLY
AUXERROR REPORT
SORTNUM 8 SORTDEVT 3390
WORKDDN(TSYUT1,TSORTOUT) ERRDDN(TSYERR)
DRAIN_WAIT 20  RETRY 120  RETRY_DELAY 60
```


The CHECK DATA is now run on a snapshot copy of the base table space without disturbing the applications. The same 14 LOB errors are reported, and the base table space is set in ACHKP at the end of the utility.

To make the table space available for applications when in ACHKP status, run CHECK DATA with AUXERROR INVALIDATE and with SHRLEVEL REFERENCE or CHANGE (DB2 9). As a result, the base table space is set in the auxiliary warning state (AUXW), which makes the data available again for the applications. The 14 bad LOBS are invalidated in the base table space as shown in Example 7-29.

Example 7-29 CHECK DATA AUXERROR INVALIDATE

```

.....
DSNU806I  -DB9B 219 12:43:04.95 DSNUKRDN - TABLE=##T.NORMEN03 COLUMN=IMAGE WAS SET INVALID
          ROWID=X'2D1165C204CFD30A2104015C563001000000000020A'
          VERSION=X'0001'
DSNU806I  -DB9B 219 12:43:04.95 DSNUKRDN - TABLE=##T.NORMEN03 COLUMN=IMAGE WAS SET INVALID
          ROWID=X'50C8D45204CFD00D2104015C5630010000000005105'
          VERSION=X'0001'
DSNU816I  -DB9B 219 12:43:05.81 DSNUGSRX - TABLESPACE NORMEN03.NORMEN03 IS IN AUX WARNING STATE
DSNU749I   219 12:43:05.81 DSNUK001 - CHECK DATA COMPLETE, ELAPSED TIME=00:00:29
DSNU010I   219 12:43:05.89 DSNUGBAC - UTILITY EXECUTION COMPLETE, HIGHEST RETURN CODE=4

```

We can now manually populate the missing LOBs using SQL update statements. However, in this test case, because no applications have touched the table spaces since the base table has been point in time recovered, we can also do a point in time recovery of the LOB table space to the same RBA as the base table space as shown in Example 7-30.

Example 7-30 Point in time recovery of the LOB table space

```

LISTDEF MYLIST INCLUDE TABLESPACES TABLE ##T.NORMEN03 LOB
          INCLUDE INDEXSPACES TABLE ##T.NORMEN03 LOB
RECOVER LIST MYLIST TORBA X'000072988DE6' PARALLEL

```

As a result, base and LOB table space are in sync again. However, the base table space is now set in the ACHKP and AUXW states as shown in Example 7-31.

Example 7-31 State of the table spaces afterwards

```

DSNT360I  -DB9B *****
DSNT361I  -DB9B *   DISPLAY DATABASE SUMMARY
          *   GLOBAL
DSNT360I  -DB9B *****
DSNT362I  -DB9B   DATABASE = NORMEN03   STATUS = RW
          DBD LENGTH = 4028
DSNT397I  -DB9B
NAME      TYPE PART   STATUS          PHYERRLO PHYERRHI CATALOG  PIECE
-----
NORMEN03 TS          RW,ACHKP,AUXW
NORMLOB  LS          RW
IRN010S7 IX          RW
IRNORMEN IX          RW,ICOPY
***** DISPLAY OF DATABASE NORMEN03 ENDED *****
DSN9022I  -DB9B DSNTDDIS 'DISPLAY DATABASE' NORMAL COMPLETION

```

The only thing left now to remove the pending states is run CHECK DATA again with AUXERROR REPORT and SHRLEVEL REFERENCE. Because no more inconsistencies are found, the pending states are reset as shown in Example 7-32.

Example 7-32 State of the table spaces after CHECK DATA

```
DSNT360I  -DB9B *****
DSNT361I  -DB9B *   DISPLAY DATABASE SUMMARY
              *   GLOBAL
DSNT360I  -DB9B *****
DSNT362I  -DB9B      DATABASE = NORMEN03  STATUS = RW
              DBD LENGTH = 4028
DSNT397I  -DB9B
NAME      TYPE PART  STATUS              PHYERRLO PHYERRHI CATALOG  PIECE
-----
NORMEN03 TS          RW
NORMLOB  LS          RW
IRN010S7 IX          RW
IRNORMEN IX          RW,ICOPY
*****  DISPLAY OF DATABASE NORMEN03 ENDED  *****
DSN9022I  -DB9B DSNTDDIS 'DISPLAY DATABASE' NORMAL COMPLETION
***
```

This is a good time to create a new common recoverable point of consistency with the same job as in Example 7-11 on page 219. The common START_RBA in SYSIBM.SYSCOPY is now x'0000877165D0'.

We now demonstrate the use of the REPAIR utility to delete orphan rows in the LOB table space. We first delete again six rows as shown in Example 7-33.

Example 7-33 SPUFI delete

```
-----+-----+-----+-----+-----+-----+-----+
DELETE FROM ##T.NORMEN03 WHERE DOC_ID LIKE '%E%' ;
-----+-----+-----+-----+-----+-----+-----+
DSNE615I NUMBER OF ROWS AFFECTED IS 6
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
-----+-----+-----+-----+-----+-----+-----+
-----+-----+-----+-----+-----+-----+-----+
DSNE617I COMMIT PERFORMED, SQLCODE IS 0
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
-----+-----+-----+-----+-----+-----+-----+
DSNE601I SQL STATEMENTS ASSUMED TO BE BETWEEN COLUMNS 1 AND 72
DSNE620I NUMBER OF SQL STATEMENTS PROCESSED IS 1
DSNE621I NUMBER OF INPUT RECORDS READ IS 1
DSNE622I NUMBER OF OUTPUT RECORDS WRITTEN IS 14
```

We then perform a point in time recovery of the LOB table space to our most recent consistency point as shown in Example 7-34 on page 229.

Example 7-34 Point in time recovery of LOB table space only

```
LISTDEF MYLIST INCLUDE TABLESPACES TABLE ##T.NORMEN03 LOB
                INCLUDE INDEXSPACES TABLE ##T.NORMEN03 LOB
RECOVER LIST MYLIST TORBA X'0000877165D0' PARALLEL
```

As a result, the base table space NORMEN03.NORMEN03 is again put in the ACHKP restrictive pending state making it unavailable for applications. The LOB table space now contains six orphan LOBs. We can identify these orphans by running CHECK DATA as in Example 7-35, which resets the ACHKP pending state. Because the base table does not contain invalid LOBs, it is not set in the AUXW status and is available again for the applications.

Example 7-35 CHECK DATA AUXERROR INVALIDATE SHRLEVEL REFERENCE

```
.....
CHECK DATA TABLESPACE NORMEN03.NORMEN03
SCOPE AUXONLY
AUXERROR INVALIDATE
SORTNUM 8 SORTDEVT 3390
WORKDDN(TSYSUT1,TSORTOUT) ERRDDN(TSYSERR)
```

The CHECK data gives us the ROWID and VERSION of the orphan LOBs as shown in Example 7-36.

Example 7-36 Identification and Invalidation of orphan LOBs

```
.....
DSNU730I    221 14:55:23.80 DSNUKDST - CHECKING TABLE ##T.NORMEN03
DSNU042I    221 14:55:24.41 DSNUGSOR - SORT PHASE STATISTICS -
                NUMBER OF RECORDS=5877
                ELAPSED TIME=00:00:00
DSNU042I    221 14:55:25.08 DSNUGSOR - SORT PHASE STATISTICS -
                NUMBER OF RECORDS=6
                ELAPSED TIME=00:00:00
DSNU813I    221 14:55:25.08 DSNUKERE - LOB IN TABLE SPACE NORMEN03.NORMLOB WITH
                ROWID=X'05A455C204CFD1062104015C5630'
                AND VERSION=X'0001' HAS NO BASE TABLE ROW
DSNU813I    221 14:55:25.08 DSNUKERE - LOB IN TABLE SPACE NORMEN03.NORMLOB WITH
                ROWID=X'2D1165C204CFD30A2104015C5630'
                AND VERSION=X'0001' HAS NO BASE TABLE ROW
DSNU813I    221 14:55:25.08 DSNUKERE - LOB IN TABLE SPACE NORMEN03.NORMLOB WITH
                ROWID=X'3E8FB7D204CFDA0F2104015C5630'
                AND VERSION=X'0001' HAS NO BASE TABLE ROW
DSNU813I    221 14:55:25.08 DSNUKERE - LOB IN TABLE SPACE NORMEN03.NORMLOB WITH
                ROWID=X'50C8D45204CFD00D2104015C5630'
                AND VERSION=X'0001' HAS NO BASE TABLE ROW
DSNU813I    221 14:55:25.08 DSNUKERE - LOB IN TABLE SPACE NORMEN03.NORMLOB WITH
                ROWID=X'6B8F05C204CFDE392104015C5630'
                AND VERSION=X'0001' HAS NO BASE TABLE ROW
DSNU813I    221 14:55:25.08 DSNUKERE - LOB IN TABLE SPACE NORMEN03.NORMLOB WITH
                ROWID=X'9B4177D204CFD50B2104015C5630'
                AND VERSION=X'0001' HAS NO BASE TABLE ROW
DSNU739I    221 14:55:25.08 DSNUKDST - CHECK TABLE ##T.NORMEN03 COMPLETE, ELAPSED TIME=00:00:00
DSNU749I    221 14:55:25.10 DSNUK001 - CHECK DATA COMPLETE,ELAPSED TIME=00:00:01
DSNU010I    221 14:55:25.18 DSNUGBAC - UTILITY EXECUTION COMPLETE, HIGHEST RETURN CODE=4
```

We can now use the REPAIR utility to delete the six orphan rows as shown in Example 7-37 on page 230 followed by the same CHECK DATA as in Example 7-35 to check if the orphan LOBs are gone.

Example 7-37 Use of REPAIR to delete orphan LOBs

```
REPAIR OBJECT
  LOCATE TABLESPACE NORMEN03.NORMLOB
  ROWID X'05A455C204CFD1062104015C5630' VERSION X'0001' DELETE
REPAIR OBJECT
  LOCATE TABLESPACE NORMEN03.NORMLOB
  ROWID X'2D1165C204CFD30A2104015C5630' VERSION X'0001' DELETE
REPAIR OBJECT
  LOCATE TABLESPACE NORMEN03.NORMLOB
  ROWID X'3E8FB7D204CFDA0F2104015C5630' VERSION X'0001' DELETE
REPAIR OBJECT
  LOCATE TABLESPACE NORMEN03.NORMLOB
  ROWID X'50C8D45204CFD00D2104015C5630' VERSION X'0001' DELETE
REPAIR OBJECT
  LOCATE TABLESPACE NORMEN03.NORMLOB
  ROWID X'6B8F05C204CFDE392104015C5630' VERSION X'0001' DELETE
REPAIR OBJECT
  LOCATE TABLESPACE NORMEN03.NORMLOB
  ROWID X'9B4177D204CFD50B2104015C5630' VERSION X'0001' DELETE
```

The REPAIR deletes the six orphan LOBs, and the base table space and the LOB table space are in sync again.

7.2.2 LOGGED base table space with NOT LOGGED LOB table space

In the second scenario, we create the base table space as LOGGED and the LOB table space as NOT LOGGED. We then redo the following steps as in the first scenario:

1. LOAD the data.
2. Create a common recoverable point of consistency using the COPY utility with LISTDEF and SHRLEVEL REFERENCE as shown in Example 7-11 on page 219. In SYSIBM.SYSCOPY, we now get a common START_RBA = X'00008927CDE6'.
3. Delete six LOBs using SPUFI as shown in Example 7-13 on page 220 to create some update activity. As a result, the LOB table space is put in the informational copy (ICOPY) status, because the changes to the LOB table spaces have not been logged.
4. Stop and start all spaces of database NORMEN03 and delete the VSAM clusters using ISPF 3.4 as shown in Example 7-14 on page 220.
5. Recover the VSAM clusters back to the current point with the same statements as shown in Example 7-15 on page 220. Because of the logged system pages, the recovery of the LOB table space succeeds without problems. DB2 is able to delete the six LOBS again from the full image copy of the LOB table space, even when the LOB table space is created as NOT LOGGED.
6. We then reinsert the six rows with SPUFI as shown in Example 7-38.

Example 7-38 SPUFI Insert

```
INSERT INTO ##T.NORMEN03 (DOC_ID,PAGE_NUMBER,FORMAT,IMAGE)
SELECT DOC_ID,PAGE_NUMBER,FORMAT,IMAGE FROM ##T.NORMEN00
  WHERE DOC_ID LIKE '%E%' ;
-----+-----+-----+-----+-----+-----+-----+
DSNE615I NUMBER OF ROWS AFFECTED IS 6
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
-----+-----+-----+-----+-----+-----+-----+-----+
```

```

-----+-----+-----+-----+-----+-----+-----+
DSNE617I COMMIT PERFORMED, SQLCODE IS 0
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0

```

- Now it is interesting to see what happens if we stop, start, and delete all VSAM clusters again and try to recover them back to the current point with the same statements as in Example 7-15 on page 220. Because the inserts of the LOBs were not logged, the LOB table space is put in AUXW and the six LOBs are invalidated in the LOB table space by the RECOVER utility as shown in Example 7-39.

Example 7-39 LOB table space in AUXW

```

DSNT360I  -DB9B *****
DSNT361I  -DB9B *   DISPLAY DATABASE SUMMARY
              *   GLOBAL
DSNT360I  -DB9B *****
DSNT362I  -DB9B      DATABASE = NORMEN03  STATUS = RW
              DBD LENGTH = 4028
DSNT397I  -DB9B
NAME      TYPE PART  STATUS              PHYERRLO PHYERRHI CATALOG  PIECE
-----
NORMEN03 TS          RW
NORMLOB  LS          RW,ICOPY,AUXW
IRN01BNY IX          RW
IRNORMEN IX          RW
*****  DISPLAY OF DATABASE NORMEN03 ENDED  *****
DSN9022I  -DB9B DSNTDDIS 'DISPLAY DATABASE' NORMAL COMPLETION
***

```

The invalid LOBS can be shown by running a CHECK LOB utility on the LOB table space. The job output is shown in Example 7-40.

Example 7-40 CHECK LOB output

```

.....
DSNU743I  222 17:03:36.39 DSNUKLB - LOB IS INVALID.
          ROWID X'00E0C2EC03CFD4062104015C5630' VERSION X'0001'
DSNU743I  222 17:03:36.40 DSNUKLB - LOB IS INVALID.
          ROWID X'554542EC03CFD1372104015C5630' VERSION X'0001'
DSNU743I  222 17:03:36.41 DSNUKLB - LOB IS INVALID.
          ROWID X'94EF82EC03CFD5092104015C5630' VERSION X'0001'
DSNU743I  222 17:03:36.41 DSNUKLB - LOB IS INVALID.
          ROWID X'D3FD82EC03CFD4272104015C5630' VERSION X'0001'
DSNU743I  222 17:03:36.41 DSNUKLB - LOB IS INVALID.
          ROWID X'D578C2EC03CFD4222104015C5630' VERSION X'0001'
DSNU743I  222 17:03:36.41 DSNUKLB - LOB IS INVALID.
          ROWID X'F20AC2EC03CFD41F2104015C5630' VERSION X'0001'
DSNU796I  222 17:03:36.42 DSNUKLB - REPRLOB PHASE COMPLETE, ELAPSED TIME=00:00:00
DSNU568I  -DB9B 222 17:03:36.62 DSNUGSRX - TABLESPACE NORMEN03.NORMLOB IS IN INFORMATIONAL COPY PENDING STATE
DSNU816I  -DB9B 222 17:03:36.62 DSNUGSRX - TABLESPACE NORMEN03.NORMLOB IS IN AUX WARNING STATE
DSNU010I  222 17:03:36.63 DSNUBAC - UTILITY EXECUTION COMPLETE, HIGHEST RETURN CODE=4

```

These LOBs can now be populated using SQL update. CHECK DATA on the base table space does not report or invalidate these LOBs in the base table space. We now use SPUIF to update these invalid LOBs as shown in Example 7-41 on page 232.

Example 7-41 SPUFI update invalid LOB

```
UPDATE ##T.NORMEN03
  SET IMAGE = BLOB('INVALID LOB')
  WHERE DOC_ID LIKE '%E%'
;
-----+-----+-----+-----+-----+-----+
DSNE615I NUMBER OF ROWS AFFECTED IS 6
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
-----+-----+-----+-----+-----+-----+
-----+-----+-----+-----+-----+-----+
DSNE617I COMMIT PERFORMED, SQLCODE IS 0
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
```

Afterwards, CHECK LOB SHRLEVEL REFERENCE does not find invalid LOBs anymore and the AUXW pending status on the LOB table space is reset.

More complicated scenarios could involve point in time recovery, but here the same techniques apply as in the first scenario. The basic principles remain:

- ▶ LOBs with missing log records between the used image copy and the recovery point are marked *invalid* by RECOVERY and the LOB table space is put in the *auxiliary warning state* (AUXW). CHECK LOB is needed to identify the invalid LOBs, and SQL can be used to populate them again using update or by deleting the entire row.
- ▶ If you do not recover the base table space and LOB table spaces together to a common point of consistency, the base table space is marked as *auxiliary check pending* (ACHKP). CHECK DATA is needed to identify and invalidate the LOBs, which are no longer synchronized between the base and LOB table space, and SQL can be used to populate them again using update or by deleting the entire row.

7.2.3 NOT LOGGED base table space with NOT LOGGED LOB table space

In the third scenario, we create both the base table space as NOT LOGGED and the LOB table space as NOT LOGGED. We then redo the following steps:

1. LOAD the data.
2. Create a common recoverable point of consistency using the COPY utility with LISTDEF and SHRLEVEL REFERENCE as shown in Example 7-11 on page 219. In SYSIBM.SYSCOPY, we now get a common START_RBA = X'0002BB16C862'.
3. Delete six LOBs using SPUFI as shown in Example 7-13 on page 220 to create some update activity. As a result, both the base table space and the LOB table space and all underlying indexes are put in the informational copy (ICOPY) status, because no changes have been logged at all.
4. Stop and start all spaces of database NORMEN03 and delete the VSAM clusters using ISPF 3.4 as shown in Example 7-14 on page 220.
5. Recover the VSAM clusters back to the current point with the same statements as shown in Example 7-15 on page 220. The job output is shown in Example 7-42.

Example 7-42 RECOVER of NOT LOGGED objects

```
DSNU000I 223 18:37:16.58 DSNUGUTC - OUTPUT START FOR UTILITY, UTILID = RECOV.NORMEN03
DSNU1044I 223 18:37:16.64 DSNUGTIS - PROCESSING SYSIN AS EBCDIC
DSNU050I 223 18:37:16.65 DSNUGUTC - LISTDEF MYLIST INCLUDE TABLESPACES TABLE ##T.NORMEN03 ALL INCLUDE
INDEXSPACES TABLE ##T.NORMEN03 ALL
DSNU1035I 223 18:37:16.65 DSNUILDR - LISTDEF STATEMENT PROCESSED SUCCESSFULLY
DSNU050I 223 18:37:16.65 DSNUGUTC - RECOVER LIST MYLIST PARALLEL
DSNU1033I 223 18:37:16.67 DSNUGULM - PROCESSING LIST ITEM: TABLESPACE NORMEN03.NORMEN03
```

```

DSNU1033I 223 18:37:16.67 DSNUGULM - PROCESSING LIST ITEM: TABLESPACE NORMEN03.NORMLOB
DSNU1033I 223 18:37:16.67 DSNUGULM - PROCESSING LIST ITEM: INDEXSPACE NORMEN03.IRNORMEN
DSNU1033I 223 18:37:16.67 DSNUGULM - PROCESSING LIST ITEM: INDEXSPACE NORMEN03.IRNO1RLT
DSNU427I 223 18:37:16.68 DSNUCBMT - OBJECTS WILL BE PROCESSED IN PARALLEL,
        NUMBER OF OBJECTS = 4
DSNU532I 223 18:37:16.68 DSNUCBMT - RECOVER INDEXSPACE NORMEN03.IRNORMEN START
DSNU515I 223 18:37:16.68 DSNUCBAL - THE IMAGE COPY DATA SET DB2IM.DB9B.NORMEN03.IRNORMEN.F06223.#222751 WITH
DATE=20060811 AND TIME=182751
        IS PARTICIPATING IN RECOVERY OF INDEXSPACE NORMEN03.IRNORMEN
DSNU532I 223 18:37:17.18 DSNUCBMT - RECOVER INDEXSPACE NORMEN03.IRNO1RLT START
DSNU515I 223 18:37:17.18 DSNUCBAL - THE IMAGE COPY DATA SET DB2IM.DB9B.NORMEN03.IRNO1RLT.F06223.#222751 WITH
DATE=20060811 AND TIME=182751
        IS PARTICIPATING IN RECOVERY OF INDEXSPACE NORMEN03.IRNO1RLT
DSNU504I 223 18:37:17.28 DSNUCBRT - MERGE STATISTICS FOR INDEXSPACE NORMEN03.IRNORMEN -
        NUMBER OF COPIES=1
        NUMBER OF PAGES MERGED=43
        ELAPSED TIME=00:00:00
DSNU532I 223 18:37:17.62 DSNUCBMT - RECOVER TABLESPACE NORMEN03.NORMEN03 START
DSNU515I 223 18:37:17.62 DSNUCBAL - THE IMAGE COPY DATA SET DB2IM.DB9B.NORMEN03.NORMEN03.F06223.#222751 WITH
DATE=20060811 AND TIME=182751
        IS PARTICIPATING IN RECOVERY OF TABLESPACE NORMEN03.NORMEN03
DSNU504I 223 18:37:17.72 DSNUCBRT - MERGE STATISTICS FOR INDEXSPACE NORMEN03.IRNO1RLT -
        NUMBER OF COPIES=1
        NUMBER OF PAGES MERGED=63
        ELAPSED TIME=00:00:00
DSNU532I 223 18:37:18.07 DSNUCBMT - RECOVER TABLESPACE NORMEN03.NORMLOB START
DSNU515I 223 18:37:18.07 DSNUCBAL - THE IMAGE COPY DATA SET DB2IM.DB9B.NORMEN03.NORMLOB.F06223.#222751 WITH
DATE=20060811 AND TIME=182756
        IS PARTICIPATING IN RECOVERY OF TABLESPACE NORMEN03.NORMLOB
DSNU504I 223 18:37:18.17 DSNUCBRT - MERGE STATISTICS FOR TABLESPACE NORMEN03.NORMEN03 -
        NUMBER OF COPIES=1
        NUMBER OF PAGES MERGED=102
        ELAPSED TIME=00:00:00
DSNU504I 223 18:37:24.06 DSNUCBRT - MERGE STATISTICS FOR TABLESPACE NORMEN03.NORMLOB -
        NUMBER OF COPIES=1
        NUMBER OF PAGES MERGED=46308
        ELAPSED TIME=00:00:05
DSNU513I -DB9B 223 18:37:24.10 DSNUCALA - RECOVER UTILITY LOG APPLY RANGE IS RBA 0002BBB98311 LRSN 0002BBB98311 TO
        RBA 0002BBB98DC5 LRSN 0002BBB98DC5
DSNU1510I 223 18:37:24.11 DSNUCBLA - LOG APPLY PHASE COMPLETE, ELAPSED TIME = 00:00:00
DSNU1505I -DB9B 223 18:37:24.12 DSNUCATM - RECOVERY OF NOT LOGGED INDEXSPACE NORMEN03.IRNORMEN WAS TO
        THE LAST RECOVERABLE POINT: RBA/LRSN X'0002BB16C862'.
        THE OBJECT HAS BEEN CHANGED SINCE THAT POINT
DSNU815I -DB9B 223 18:37:24.18 DSNUGSRX - TABLE SPACE NORMEN03.NORMEN03 IS IN AUX CHECK PENDING STATE
DSNU568I -DB9B 223 18:37:24.18 DSNUGSRX - TABLESPACE NORMEN03.NORMLOB IS IN INFORMATIONAL COPY PENDING STATE
DSNU500I 223 18:37:24.18 DSNUCBDR - RECOVERY COMPLETE, ELAPSED TIME=00:00:07
DSNU010I 223 18:37:24.19 DSNUGBAC - UTILITY EXECUTION COMPLETE, HIGHEST RETURN CODE=4

```

DB2 is only able to recover the data to the last recoverable point 0002BB16C862 and issues message DSN1505I. No effort is made to recover the lost data and the base table space is put in the ACHKP status, making it unavailable for applications. We can now run CHECK DATA SHRLEVEL REFERENCE as in Example 7-26 on page 226 to remove the pending state and be sure that the LOB data is in good shape again. We are back now in the initial position with the six deleted LOBS being back in the data.

7.2.4 LOBs and SYSTEM RECOVERY

The online BACKUP SYSTEM utility invokes z/OS DFSMSHsm™ (Version 1 Release 7 or above) to copy the volumes on which the DB2 data and log information resides for either a DB2 subsystem or data sharing group. You can use BACKUP SYSTEM to copy all data for a single application (for example, when DB2 is the database server for a resource planning solution). All data sets that you want to copy must be SMS-managed data sets. You can subsequently run the RESTORE SYSTEM utility to recover the data.

The RESTORE SYSTEM utility invokes z/OS DFSMSHsm (Version 1 Release 5 or above) to recover a DB2 subsystem or a data sharing group to a previous point in time. To perform the recovery, the utility uses data that is copied by the BACKUP SYSTEM utility. All data sets that you want to recover must be SMS-managed data sets. You must use the change log inventory utility DSNJU003 to record the log point to which you want to recover the system:

- ▶ CRESTART CREATE,SYSPITR=log-truncation-point: Specifies the log RBA (non-data sharing system) or the log LRSN (data sharing system) that represents the log truncation point for the point in time for system recovery.
- ▶ CRESTART CREATE,SYSPITRT=log-truncation-timestamp: Specifies the timestamp value that represents the point in time log truncation point for system recovery. Log-truncation-timestamp specifies a timestamp value that is to be used as the log truncation point. A valid log truncation point is any GMT timestamp for which there exists a log record with a timestamp that is greater than or equal to the specified timestamp value. Any log information in the bootstrap data set, the active logs, and the archive logs with a timestamp greater than SYSPITRT is discarded. If you omit SYSPITRT, DB2 determined the end of the log range. The SYSPITRT keyword is new in DB2 9.

The RESTORE SYSTEM utility uses the most recent system-level backup of the database copy pool that DB2 took prior to the SYSPITR or SYSPITRTlog truncation point.

Complete the following steps prior to running RESTORE SYSTEM:

1. Stop DB2.
2. Run DSNJU003 (Change Log Inventory) with the CRESTART SYSPITR or SYSPITRT option. For SYSPITR or SYSPITRT, specify the log truncation point that corresponds to the previous point in time to which the system is to be recovered.
3. Start DB2. When the restart that is specified by CRESTART SYSPITR or SYSPITRT completes, DB2 enters system RECOVER-pending and access maintenance mode. During system RECOVER-pending mode, you can run only the RESTORE SYSTEM utility.
4. Ensure that the ICF catalog volumes for DB2 data are not active. The ICF catalog for the data must be on a separate volume that the ICF catalog for the logs.

Important: If RESTORE SYSTEM determines that a NOT LOGGED base table space or NOT LOGGED LOB table space was updated after the point at which the system level copy was taken, the table space is marked RECOVER-pending. Use the RECOVER utility and/or the REBUILD INDEX utility afterwards to recover all objects in RECOVER-pending (RECP) or REBUILD-pending (RBDP) status.

There are no other special considerations when your DB2 data contains LOB table spaces in regard to the BACKUP SYSTEM and RESTORE SYSTEM utilities.

7.2.5 Conclusions on recovery of LOB data

Planning for LOB recovery is similar to planning for user-defined referential integrity, because you have to remember that there is a *relationship* between a table with a LOB column and the associated LOB table space. It is true that tables involved in referential integrity relationships must be considered as part of the same table space set for recovery purposes. Similarly, a base table space and related LOB table spaces are part of a table space set.

Recovery of LOB data is quite complex and can involve:

- ▶ RECOVER table space and index and REBUILD INDEX utilities
- ▶ CHECK LOB, CHECK DATA, and CHECK INDEX utilities

- ▶ REPAIR utility
- ▶ SQL update and delete of LOB columns

To be prepared for recovery to the current point in time:

- ▶ Take image copies of all base and LOB table spaces using LISTDEFS.
- ▶ Eventually take image copies of the underlying indexes.
- ▶ Use LOGGED or LOG YES if possible to avoid missing log records when recovering.

To be prepared for point in time recovery:

- ▶ Take image copies of all base and LOB table spaces using LISTDEFS.
- ▶ Take quiesce point for all base and LOB table spaces using LISTDEFS to establish meaningful recovery points from the application point of view.
- ▶ Use LOGGED or LOG YES if possible to avoid missing log records when recovering.

To make your life easy when doing point in time recovery:

- ▶ Create common recoverable points of consistency by taking full image copies with SHRLEVEL REFERENCE and using LISTDEFS to include all objects.
- ▶ Create common recoverable quiesce points.
- ▶ Always recover all your objects together to a common recoverable consistency point.
- ▶ All this avoids the use of CHECK DATA, CHECK LOB, REPAIR, and so forth.

If LOGGED or LOG YES is not acceptable:

- ▶ Take image copies of all base and LOB table spaces using LISTDEFS before and after updating the LOB data.
- ▶ Keep new LOB data, that is not yet included in a full image copy, aside to be able to correct LOBs using SQL update when needed.

Take only image copies with SHRLEVEL CHANGE when you have no windows available for taking image copies with SHRLEVEL REFERENCE. Image copies with SHRLEVEL CHANGE do not create common recoverable points of consistency. As a consequence during point in time recovery, the base table space can be set in the auxiliary check pending (ACHKP) state, which makes it unavailable for applications as explained before.

7.3 Altering tables containing LOB columns

Prior to the introduction of the full UNLOAD and LOAD support for LOBs introduced by APAR PK22910 for DB2 V7 and V8, changing a table with LOB columns required a drop and recreate of the table:

- ▶ Dropping the base table requires a recreate of the auxiliary tables and indexes as well.
- ▶ How to UNLOAD and LOAD the LOB data if LOB columns > 32 KB?
 - DB2 UNLOAD/LOAD was not applicable.
 - No or limited support for LOBs in most commercial tools.
 - Write your own unload and load programs with static or dynamic SQL.
 - Copy to another DB2 table.

The same was true for a change of the base or LOB table spaces that required a drop and recreate of the table space.

So there are several ways to perform this task:

- Write your own program.

This requires a DB2 user or administrator to write a custom program for unloading and loading LOBs. The programs would usually read the LOB columns of the tables and write them to flat files. When the change is completed, the program inserts these LOB values back into the tables. You could speed up the process by altering the LOB table spaces to LOG NO and LOCKSIZE TABLESPACE to consume less of your system resources. Afterwards, a recoverable point of consistency should be taken in the form of an Image Copy.

This process has some disadvantages, because you have to code your programs and set up the recoverable step. This also had a tremendous impact on logs, because massive LOB writes were recorded in them, and you could prevent it only by changing LOG attributes and again by setting up proper controls. And of course, the data was inaccessible for a long period of time during this maintenance.

- Use shadow tables.

The recommended way was creating a shadow set of the objects, identical to the original ones. Use DSN1COPY with OBIDXLAT to copy the original objects into the shadow ones. Recreate the original table with new definitions and then use SQL INSERT INTO *original-table* SELECT FROM *shadow-table*... to copy the data from the shadow table back into the original one. In the end of the process, a recoverable point of consistency could be taken in the form of a DB2 COPY.

You could speed up the process by altering the LOB table spaces to LOG NO and LOCKSIZE TABLESPACE to consume less of your system resources. Also be careful not to copy the ROWID values when the ROWID in the target table is defined as GENERATED ALWAYS.

This process is usually faster than the one described before. And of course, there was a great impact in the log I/O, so this process should have been made during the quiet hours of the system. A great improvement to this method was using DB2 cross loader to load the data back into the original changed table. For more information about cross loader, refer to 6.3, “LOAD” on page 171.

- Today clients using DB2 V7 and DB2 V8 are encouraged to use the UNLOAD and LOAD utilities to perform these changes that disrupt their business, because the new maintenance allows UNLOAD and LOAD LOB to deal with values larger than 32 KB together with the other columns of the base table. For more information, refer to the following sections:
 - 6.1, “UNLOAD” on page 160
 - 6.2, “DSNTIAUL” on page 169
 - 6.3, “LOAD” on page 171



Performance with LOBs

In this chapter, we summarize general performance considerations, provide information about trace fields for LOBs, and mention preliminary performance measurements.

We discuss the following:

- ▶ LOB materialization
- ▶ Virtual storage management for LOBs
- ▶ Buffer pools and group buffer pools
- ▶ Logging with LOBs
- ▶ Accessing LOBs
- ▶ Comparing SQL accounting profiles
- ▶ Important I/O aspects
- ▶ IFCID enhancements for LOBs
- ▶ DRDA LOB flow optimization performance
- ▶ LOB recommendations for performance

8.1 LOB materialization

It is essential to underline at this stage that buffer pools are part of a standard mechanism for DB2 to move data from disks to applications. All of the data that is read or written passes through buffer pools. Thus, when materialization of LOBs is discussed, we talk about the storage that LOB manager allocates in storage areas that are different from buffer pools. Since data spaces are no longer used for LOB materialization starting with DB2 V8, the materialization area is now allocated in the DBM1 address space above the 2 GB bar.

LOB materialization is the function DB2 uses to place the whole LOB value into contiguous storage in virtual storage. Materialization happens also on disk when the LOB is written into table space, but we are especially interested in the cases where the LOB materializes in DBM1 address space above the 2 GB bar or the user (allied) address space.

In several cases, you cannot avoid the materialization of LOBs in storage. Examples of when the materialization is necessary are:

- ▶ A LOB needs to be converted from one CCSID to another.
- ▶ A LOB is an argument of a user-defined function.
- ▶ A LOB is moved into or out of a stored procedure.
- ▶ A LOB host variable is assigned to a LOB locator host variable (you are bringing the whole LOB into your application local storage).
- ▶ A FETCH CONTINUE statement is used.

With or without the use of a locator, retrieving a LOB value always implies the passage of all LOB pages through the buffer pool associated with the LOB table space. Then, if a LOB needs to be materialized in storage, all the pages of the LOB value are placed in the designated area above the 2 GB bar.

The amount of storage used in the DBM1 address space above the 2 GB bar for LOB materialization depends upon a number of factors. They include the size of the LOB, the number of LOBs in a statement that needs to be materialized, and the use of cursors to hold the position in an application. Figure 8-1 on page 239 provides an overview of the materialization process.

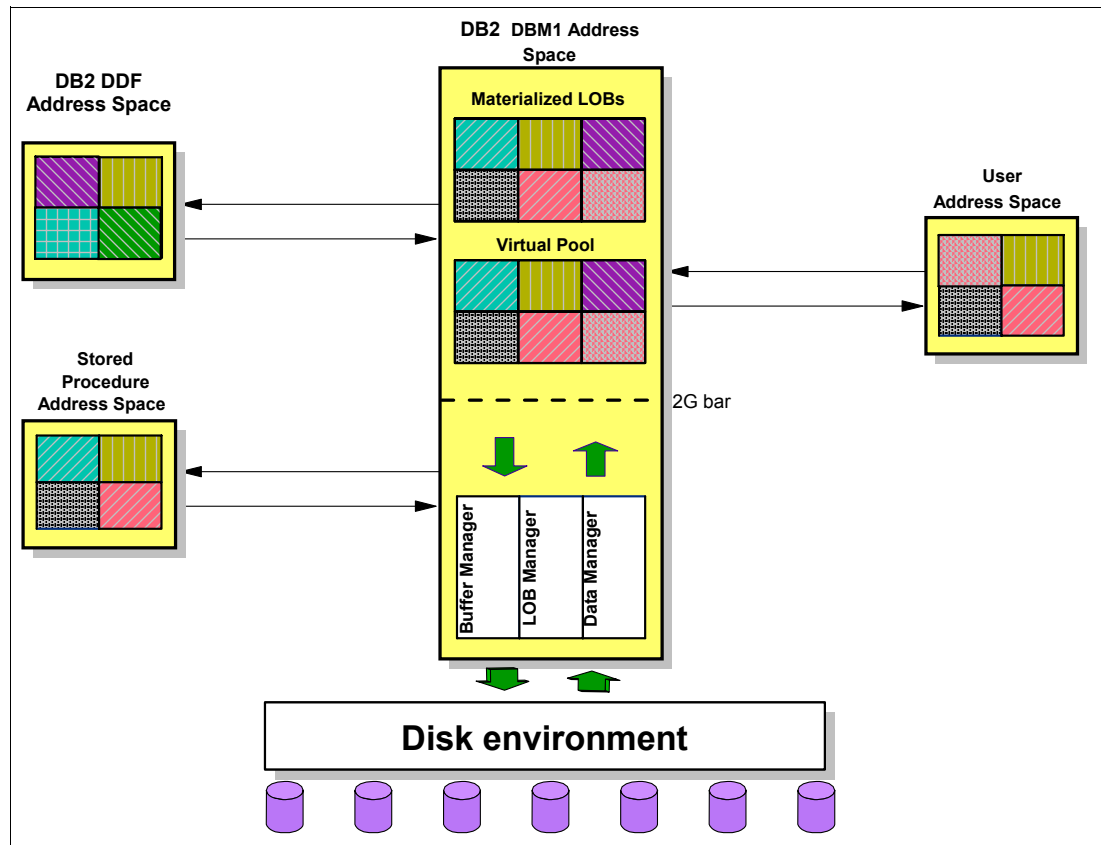


Figure 8-1 DB2 materialization overview illustration

8.1.1 The different cases of materialization

In order to give you a better picture of LOB materialization and when it occurs, we describe different scenarios.

In general, you can UPDATE, INSERT, DELETE, or SELECT a LOB value. DB2 9 introduces new and exciting ways of manipulating LOBs with LOB file reference variables. The FETCH CONTINUE clause is introduced by DB2 9 to make the programmer's life easier and the code less complicated.

With versions prior to DB2 9, there are several good reasons for using LOCATORS when processing LOB values. A LOCATOR is recommended if you care about storage allocation within your users' address spaces, overall performance, and useful handling of your LOB data.

The V9 LOB file reference variables are very helpful when all you need to do is transfer data between DB2 and a file that is external to DB2.

Materialization within your DB2 DBM1 virtual storage depends on the way the LOB value is accessed.

SELECT

We assume we have three applications that select a LOB for further processing. This could be printing a LOB that contains a book or storing it into a non-DB2 data set. Let us say we have application A using a LOCATOR, application B using neither a LOCATOR nor file reference variables, but rather a host variable, and application C using LOB file reference

variables. From the point of view of DB2 LOB materialization, application A, application B, and application C do *not* materialize the LOB in DB2 virtual storage when they only select a LOB value. See Figure 8-2.

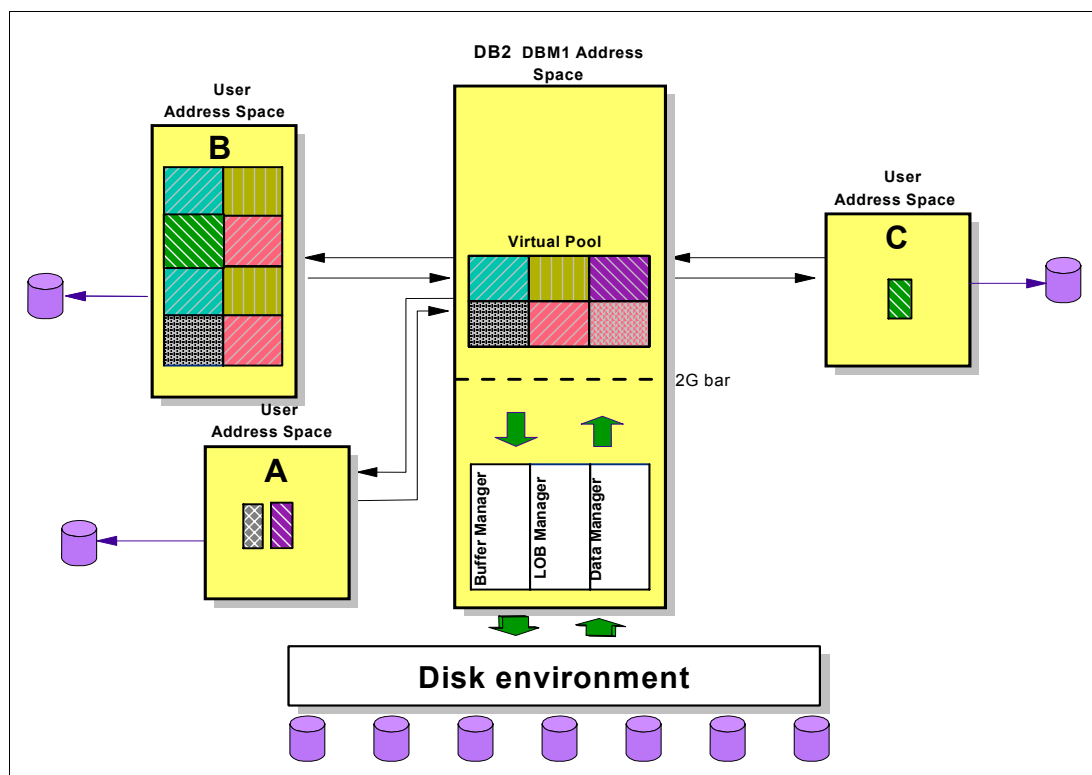


Figure 8-2 LOB Materialization In user address space in case of data retrieval

Application A uses a locator and might just use the space it needs for the chunks of data pointed to through the defined locator. Application B might require the complete LOB data. Therefore, the private user address space might require, based on the size of the LOB, a huge amount of storage. Application C uses LOB file reference variables, eliminating any need to allocate any kind of variable for storing chunks of the LOB in its local address space.

If you have applications in a distributed environment selecting LOB values through the network, they involve the DB2 DDF address space. You can minimize the server's storage consumption by utilizing the DRDA flow optimization functionality. See 4.3, "DRDA LOB flow optimization" on page 79.

If you use stored procedures, the selected LOB data is not materialized within a DBM1 address space. Stored procedures use LOCATORS and move the data in small chunks from disk through the stored procedure address space.

INSERT

Assume a scenario where your application A and application B INSERT into your LOB table space. From the point of view of materialization, the picture is different. Now there are space allocations in virtual storage involved. The materialization in DB2 virtual storage occurs if you are using LOCATORS, and it probably does not occur if you use other techniques. However, not using LOCATORS, like in application B, causes you to require more storage in your private user address space, and depending on the size of your LOB, this could consume a large amount of storage. See Figure 8-3 on page 241.

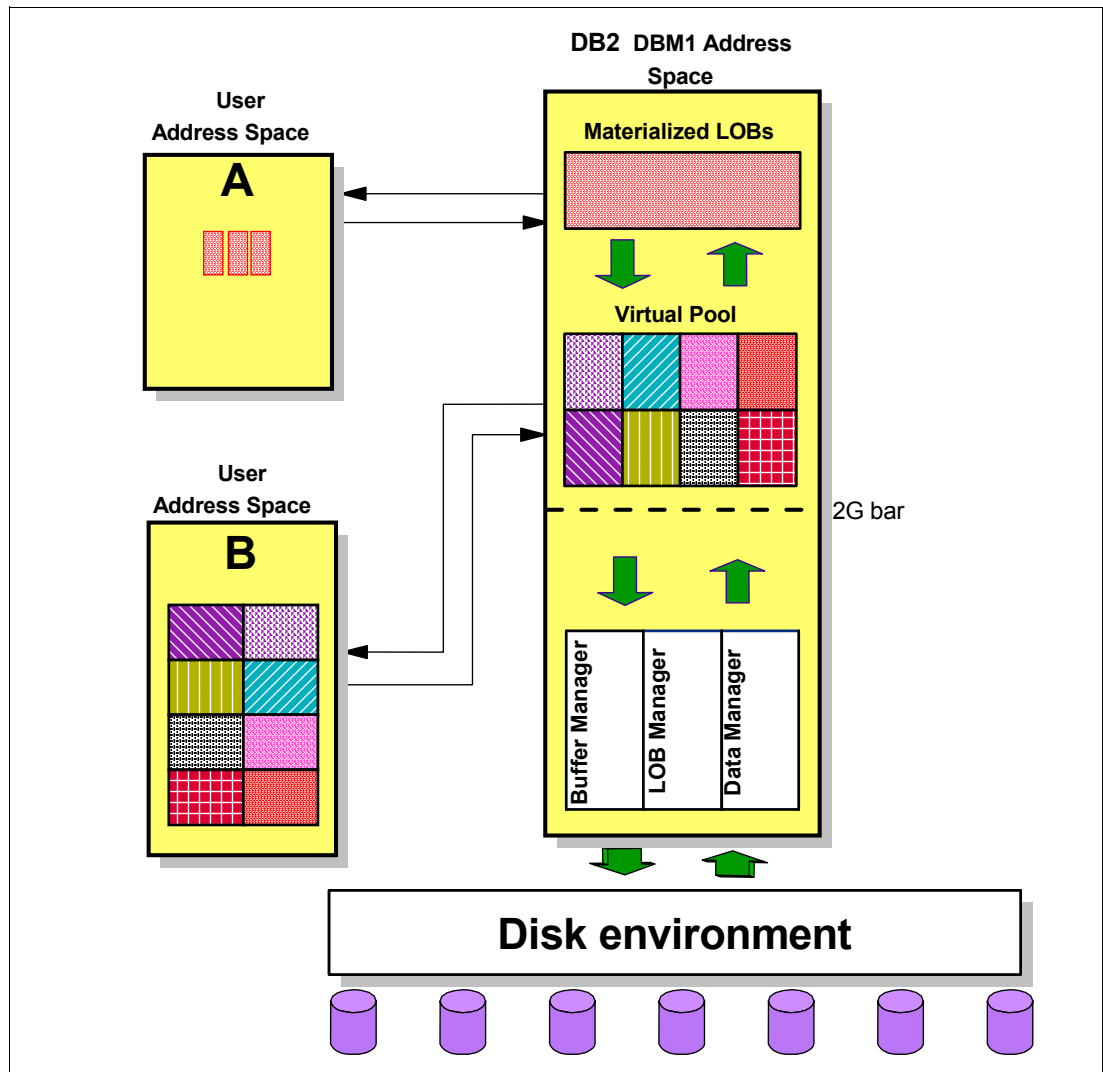


Figure 8-3 LOB materialization with INSERT

If LOB file reference variables are used, as in application C, the materialization is avoided. On the other hand, if an application INSERTs LOB values using the network (this implicates the use of the DB2 DDF address space), or it uses a stored procedure, or it uses some kind of conversion (for example from one CCSID to another), LOB materialization occurs in the DBM1 address space.

8.1.2 Materialization avoidance techniques

We recommend the following techniques to avoid LOB materialization when programming your applications.

LOB file reference variables

A LOB file reference variable manages the movement of LOBs from the database server to an application-specified data set or from an application-specified data set to the database server without going through the memory of the application. The other advantage that the file reference variable gives is that it bypasses the host language limitation on the maximum size allowed for dynamic storage to contain a LOB. So, you can INSERT a LOB from a file into a DB2 table or SELECT a LOB from a DB2 table to put into a file without having to acquire any

application storage. LOB file reference variables bypass host language limitations on the maximum allowed size for LOB values located in working storage as well. This technique is recommended when no manipulation on the LOB has to be performed. See 2.1, “Introduction to LOB data types” on page 10 and 4.2, “LOB locators” on page 73 for syntax examples and explanations.

LOB locators

When the whole LOB is not needed in the program, you can use locators to operate on the LOBs. Explanation about locators and the programming techniques is at 2.3, “LOB locators” on page 16 and 4.2, “LOB locators” on page 73.

Note: When partial continuous retrieval of a LOB is required, you can use FETCH CONTINUE instead of locators. It does not prevent LOBs from materializing. The data still needs to be materialized in the server, but this new DB2 9 functionality enables writing applications in a less complicated way. The technique is used to “stream” the data into the application. After the original FETCH is performed and there is more of the LOB to retrieve, it uses as many subsequent FETCH CONTINUE statements as necessary to finish retrieving the data, using the same buffer area. This assumes that the data in the buffer is processed after each FETCH or FETCH CONTINUE operation. The function is explained in 4.6.2, “Using FETCH CONTINUE” on page 113.

8.2 Virtual storage management for LOBs

Since the hardware and the operation system have evolved to 64-bit real and virtual addressability, the need for data spaces and hyperspaces has disappeared. All the space needed to materialize and operate LOBs in DB2 is allocated above the 2 GB bar, first in buffer pools and then possibly in variable storage.

The amount of storage used for LOB materialization depends on a number of factors including:

- ▶ The size of the LOBs
- ▶ The number of LOBs in a statement that needs to be materialized

Field QXSTLOBV in statistics or accounting traces for IFCID 0002 or IFCID 0003 contains the maximum amount of storage used for LOB materialization.

You can monitor the memory consumption above the bar by running the statistics report of *IBM Tivoli® Omegamon XE for DB2 Performance Expert* as shown on Example 8-1. Space allocated by LOBs is counted under VARIABLE STORAGE.

Example 8-1 Omegamon XE for DB2 Performance Expert output

DBM1 STORAGE ABOVE 2 GB		QUANTITY
-----		-----
FIXED STORAGE	(MB)	0.01
GETMAINED STORAGE	(MB)	79.00
COMPRESSION DICTIONARY	(MB)	0.00
CACHED DYNAMIC SQL STATEMENTS (MAX)	(MB)	14.46
DBD CACHE (MAX)	(MB)	14.46
VARIABLE STORAGE	(MB)	50.05
VIRTUAL BUFFER POOLS	(MB)	179.69
VIRTUAL POOL CONTROL BLOCKS	(MB)	0.07
CASTOUT BUFFERS	(MB)	0.00

8.2.1 DB2 subsystem parameters for LOBs

DB2 allocates storage as needed. LOB Manager keeps track of the amount used and manages the space within the DBM1 address space above the 2 GB bar. You can set the virtual storage limits in DSNZPARM by specifying the LOBVALA and LOBVALS parameters in the installation panel DSNTIP7 as shown on Figure 8-4.

Note: In DB2 9, the parameter definitions for LOBVALA and LOBVALS system parameters have moved from installation panel DSNTIP7 to DSNTIPD.

```

DSNTIPD          INSTALL DB2 - SIZES
====>

Check numbers and reenter to change:

1  DATABASES          ===> 200      In this subsystem
2  TABLES            ===> 20      Per database (average)
3  COLUMNS           ===> 10      Per table (average)
4  VIEWS              ===> 3       Per table (average)
5  TABLE SPACES      ===> 20      Per database (average)
6  PLANS               ===> 200     In this subsystem
7  PLAN STATEMENTS    ===> 30      SQL statements per plan (average)
8  PACKAGES           ===> 300     In this subsystem
9  PACKAGE STATEMENTS ===> 10      SQL statements per package (average)
10 PACKAGE LISTS      ===> 2       Package lists per plan (average)
11 EXECUTED STMTS     ===> 15      SQL statements executed (average)
12 TABLES IN STMT    ===> 2       Tables per SQL statement (average)

13 USER LOB VALUE STG ===> 10240    Max KB storage per user for LOB values
14 SYSTEM LOB VAL STG ===> 2048     Max MB storage per system for LOB values
15 MAXIMUM LE TOKENS  ===> 20      Maximum tokens at any time. 0-50

PRESS:  ENTER to continue  RETURN to exit  HELP for more information
. . . . .

```

Figure 8-4 DSNTIPD installation panel

Both DSNZPARMs can be changed without stopping DB2. For details, refer to -SET SYSPARM in the *DB2 UDB for z/OS Version 8 Command Reference*, SC18-7416, or *DB2 Version 9.1 for z/OS Installation Guide*, GC18-9846.

Table 8-1 LOB linked subsystem parameters

Parameter	Macro	Panel	Values range	Default
LOBVALA	DSN6SYSP	DSNTIPD	1 - 2,097,152 KB	10,240 KB
LOBVALS	DSN6SYSP	DSNTIPD	1 - 51,200 MB	2,048 MB

The *value* LOBVALA establishes an upper limit for the amount of variable storage that each user can have for storing LOB values. The specified value indicates the numbers of *kilobytes*.

The value LOBVALS establishes an upper limit for the amount of variable storage per system that can be used for storing LOB values. The specified value indicates the numbers of *megabytes*.

Because these definitions can have an impact on your DB2 subsystem (for example, if you exceed the virtual storage backed up by real storage, the DB2 subsystem abends) and operating system, specifically on paging and auxiliary storage, and these definitions depend on how heavy the use of LOBs is within your DB2 subsystem, the values of these settings represent a safeguard for your other workload. Set these values after consulting with your z/OS system programmer.

If you run into a resource unavailable situation (SQLCODE -904) with virtual storage allocations (resource 00000907), the reason codes you can expect are 00C900Dx (x = 1, 2, and 3). The reason codes mean respectively:

- ▶ User limit exceeded
- ▶ System limit exceeded
- ▶ Out of space condition in the virtual storage above the bar (this one would be hard to reach!)

Note: Some messages in DB2 V8 and DB2 9 might still use the term “data space”, but they do not really mean data space. We include this just to remind you that since V8, data spaces are no longer in use.

8.3 Buffer pools and group buffer pools

This section gives you an overview of buffer pool and group buffer pool considerations for when you are using large LOBs in your production or non-production environments.

8.3.1 Virtual buffer pools

Independently of how you have planned to use LOBs in your shop, you want to put LOB data in separate buffer pools, in order not to interfere with data used by online transactions or other work. If the LOBs have their own buffer pools, you have the capability of setting different thresholds than with your other buffer pools, without affecting other work. The main parameters you want to set uniquely for buffer pools containing LOBs are VPSEQT, DWQT, and VDWQT.

Tip: You should set the virtual pool sequential steal threshold (VPSEQT) to 99%, taking full advantage of the sequential access, knowing that this buffer pool is only used for LOB pages. Note that this is deliberately not set to 100%, to allow some non-sequential pages in the buffer pool. They are the LOB space map pages, and they are fetched individually and not marked as sequentially fetched.

Also, minimizing the buffer pool-wide deferred write threshold (DWQT) almost to 0 forces DB2 to asynchronously write buffers sooner out of the buffer pool. At the data set level, DB2 has the vertical deferred write threshold (VDWQT). It is expressed as a percentage of the virtual pool size for a single data set. This value should always be less than DWQT.

The effect of low values for both thresholds is to force DB2 to begin writing dirty pages to disk early in the phase of updating LOB values. This avoids the complications that might arise if the buffer pool is subjected to an intensive period of changes, such as multiple threads

updating large LOBs. In this case, if the write thresholds are high, the buffer pool might become overwhelmed, and the number of dirty pages might reach levels where critical data manager thresholds are triggered. The result of this would be for I/Os to become synchronous. This could have a dramatic impact on the performance of the updates. Note that this recommendation is based on the data length of the LOB values being quite large, that is, you are using real LOBs.

One major factor affecting the design and setup of buffer pools for LOB objects relates to whether you expect the LOB objects to be re-referenced, and the hit ratio for pages in the buffer pool becomes significant to increase performance. The opposite scenario is that you expect the re-reference of objects to be very low. See Figure 8-5.

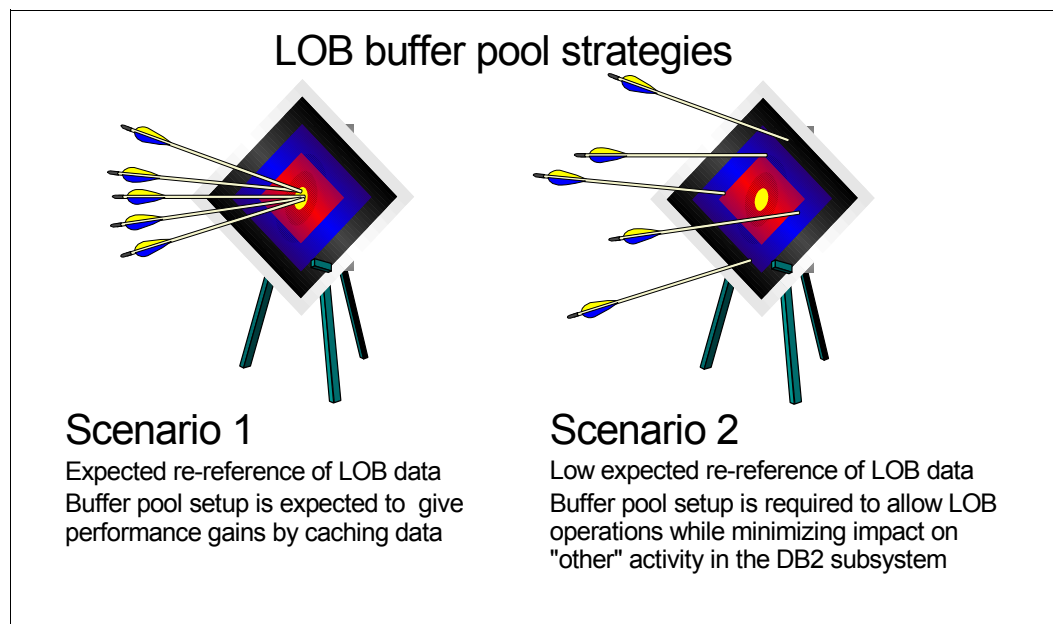


Figure 8-5 Buffer pool strategies

With the advancement of 64-bit addressing, it is possible to provide large buffer pools for LOB objects. But the availability of physical memory to back the allocated buffer pools must still be carefully considered, so as not to cause paging problems with overallocation of real storage.

If re-reference of objects is not expected, it follows that storage resources can probably be better utilized by other buffer pools to avoid I/O. The considerations for the buffer pool for these LOBs is to allow prefetching and determine a minimum size to avoid effects such as different threads stealing pages of LOB objects prior to their materialization.

If re-reference of objects is expected, the buffer pool for the LOBs is probably increased to greater than the minimum. What then follows is the trade-off of what resources to allocate to the respective buffer pools, within the available physical memory constraints. This is a topic beyond the scope of this IBM Redbook as it is obviously a major DB2-wide issue. What is needed with respect to LOBs is to determine the importance performance plays for the LOB tables involved relative to the entire DB2 subsystem content.

Sizing the LOB buffer pools is an important issue. Due to LOBs being prefetched into virtual pools for further processing, you would prefer having the buffer pool big enough to satisfy the needs of all of the parallel executing threads, and avoid prefetched pages being paged out in order to free enough space for some other LOB currently being read in. When DB2 9 is prefetching LOBs, it is done using list prefetch in blocks of up to 64 pages of 4 KB. To prevent

active threads stealing pages from each other, you would prefer to have at least three prefetched blocks to stay in the virtual buffer pool for processing.

Example 8-2 shows an algorithm to estimate the allocation for LOB buffer pool.

Example 8-2 LOB buffer pool allocation size

IOB - I/O read block for prefetch	$3 * 32 \text{ pages} * 4 \text{ KB} = 384 \text{ KB}$
PT - Maximum number of active parallel threads reading\writing LOBs	
TOTAL SIZE = PT * IOB	

Example for LOB buffer pool:

PT = 100
TOTAL SIZE = 100 * 384 KB= 38400 KB

Figure 8-6 shows the separation of LOB buffer pools for better performance and monitoring. In the figure, we have omitted BP7, associated to work files, and BP8K0, BP16K0, BP32K needed for DB2 catalog access besides BP0.

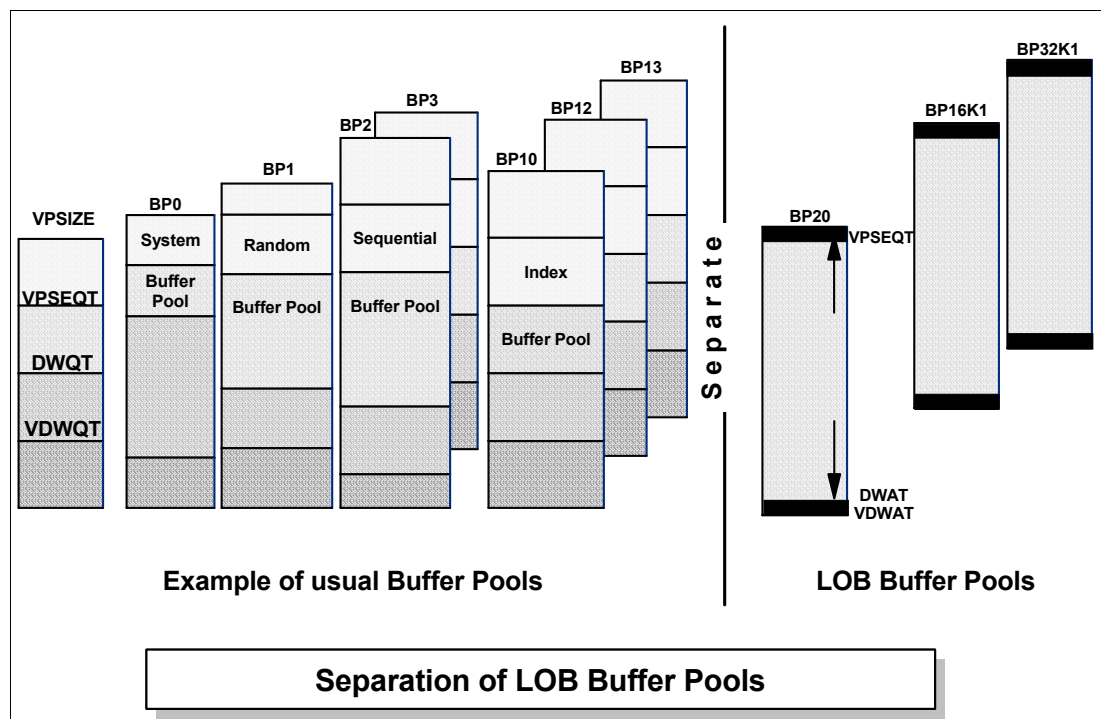


Figure 8-6 Separation of LOB buffer pools

8.3.2 Considerations for a data sharing environment

Several settings for the GBPCACHE parameter are allowed when defining and altering table spaces. If you choose GBPCACHE NONE or GBPCACHE SYSTEM, no user data pages are actually stored in the group buffer pool. However, with GBPCACHE SYSTEM, which is the default for LOB table spaces, space map pages for LOBs are cached in the coupling facility. All other data pages are written directly to disk, similar to GBPCACHE NONE page sets. See Figure 8-7 on page 247.

By choosing GBPCACHE ALL, you prevent multiple members from reading the same page in from disk. For this reason, LOB table spaces containing small LOBs that typically have a high

degree of inter-DB2 read interest are good candidates for GBPCACHE ALL. Keep in mind that allowing GBPCACHE ALL demands increased coupling facility resources: processing power, storage, and channel utilization.

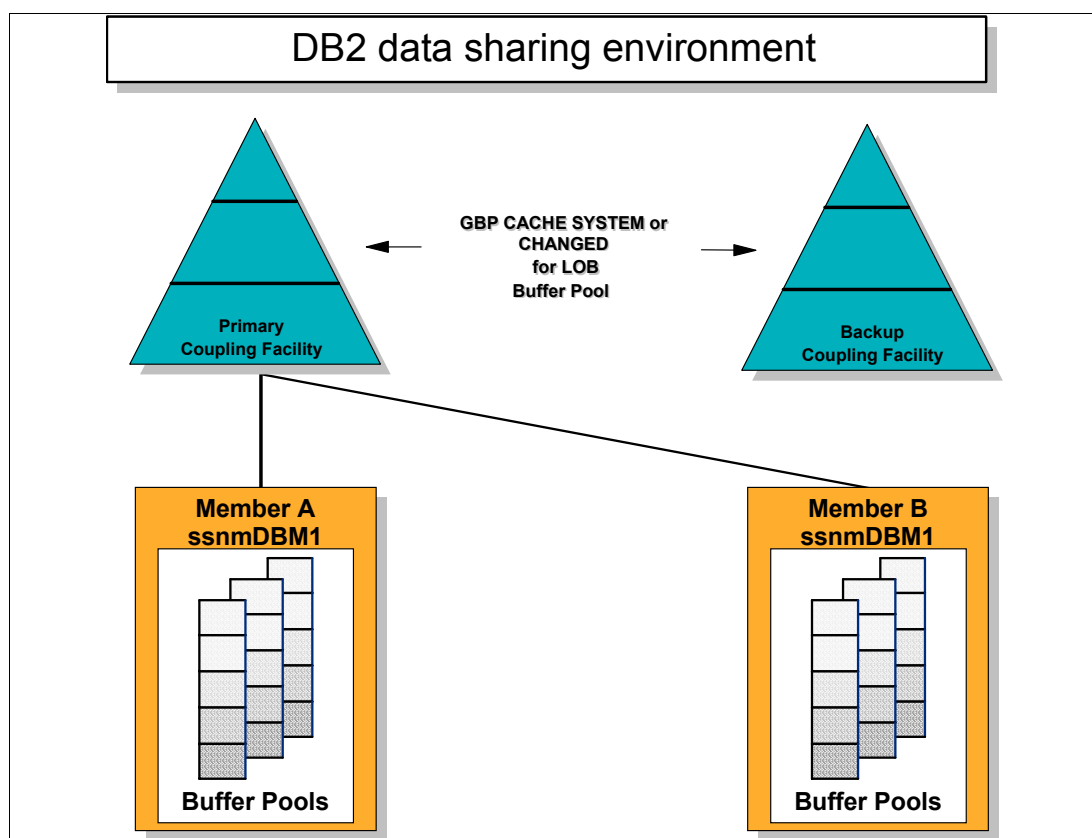


Figure 8-7 LOB group buffer pool

When the LOB table space is NOT LOGGED, we recommend using GBPCACHE CHANGED for LOB table spaces. In this case, if the coupling facility fails, the LOB table space is placed in GRECP. When group buffer pool recovery occurs, all LOB values that were in the coupling facility at the time of the failure are marked invalid, because the log records that are necessary to perform the recovery for those values are missing due to the NOT LOGGED attribute.

In DB2 9, consider specifying GBPCACHE CHANGED to flush changed pages to the CBP rather than flushing changed pages to disk and delaying the release of LOB locks.

When extremely large objects are used, you might consider using GBCACHE SYSTEM to avoid having large LOB values overwhelm the group buffer pool.

Castout threshold for group buffer pools

The *castout threshold* determines the total number of changed pages that can exist in the group buffer pool before castout occurs. DB2 casts out enough class castout queues to bring the number of changed pages below the threshold. DB2 periodically determines whether the threshold is exceeded.

We recommend setting the group buffer pool castout threshold to zero, or a low value, to reduce the need to have a large group buffer pool for LOBs.

For more information about DB2 Data Sharing, refer to *DB2 Version 9.1 for z/OS Data Sharing: Planning and Administration*, SC18-9845.

8.4 Logging with LOBs

The two major changes introduced by DB2 9 in new function mode are described at 3.3, “LOBs and LOG activity” on page 44:

- ▶ LOGGED and NOT LOGGED attributes
- ▶ Logging for all LOB sizes

While logging LOBs has performance implications for large LOBs and heavy parallel log activity, the usability and operational considerations are always important, whatever the size of LOBs and the workload. Therefore, the usability and operational considerations should be taken into account at LOB creation time.

If your logging is becoming a bottleneck and it is I/O bound (that is you do not have a CPU or latching issue with the log activity), you should consider striping to increase the parallelism of I/O and reduce the synchronous suspension time.

8.5 Accessing LOBs

Because accessing large LOBs usually involves low CPU time compared to I/O time, LOB applications tend to benefit greatly from fast devices and fast channels. LOB table spaces are also good candidates for striping. This is valid for both read and write I/O.

8.5.1 Reading LOBs

The major considerations when selecting LOBs are:

- ▶ LOB processing is generally more efficient for larger LOB columns.
- ▶ You get more consistent responses and CPU times reading LOB data using locator variables regardless of the size of the LOB column.
- ▶ The efficiency by which large LOBs can be processed is increased enormously by using LOB file reference variables.
- ▶ Prefetch in DB2 9 allocates twice as much storage per I/O if the buffer pool size is 200 MB or greater, 400 MB in the case of utilities. This might reduce the elapsed time to read a large LOB by 10% or more, and the benefit of such a “large” buffer pool is even higher if combined with striping.

8.5.2 Writing LOBs

We examine the three cases of INSERT, UPDATE, and DELETE.

LOB INSERT

As with read processing, the greater the LOB length, the greater the efficiency of insert processing. To insert 100 times the data does not require 100 times the resources. The relative improvement of insert efficiency with respect to LOB size results, however, are less dramatic than for SELECT processing. The reason is that the cost of preformatting disk space to accommodate the newly inserted LOB can increase with the size of the LOB, and it accounts for most of the elapsed time. With V9, depending on the allocation quantity, DB2

triggers the preformatting earlier and preformats 16 rather than 2 cylinders. This reduces the impact of preformatting.

There is relatively little difference in insert performance if a LOB locator is used.

When the LOB file reference variables are used, the I/O operation of writing the data to disk can be done at the device speed. Application logic needed to use the method is minimal.

LOB update

A LOB update is equivalent to a LOB insert and a LOB delete. Most of the cost of a LOB update can, therefore, be attributed to the cost of the LOB insert.

LOB delete

A LOB delete is a logical delete, and therefore, it is relatively inexpensive and gives consistently good performance. The CPU cost is somewhat independent of the size of the LOB column. The larger the LOB column, the greater the number of space map pages that have to be updated to reflect the logically deleted row, but the relationship between elapsed time and LOB size for deletes is not linear. The larger the LOB column, the greater the efficiency of the delete operation. DB2 can delete 833 KB per second if the LOB is 20 KB in length. With a 2 MB LOB, DB2 can delete 45,511 KB per second, which is more than 50 times faster.

8.6 Comparing SQL accounting profiles

There are good reasons for the use of LOBs, but there are also good reasons to consider using VARCHAR or VARBINARY (DB2 9 only) instead, if possible. Even when dealing with the wide variety of additional data types, such as audio, video, or mixed text, if they are below 32 KB in size, you should consider the possibility of storing them into a VARCHAR column. The main reasons for still wanting to use VARCHAR, VARGRAPHIC, or VARBINARY, for your small (<32 KB) multimedia objects, are that there is added complexity with LOBs, and that there might be more SQL calls executed to process a LOB than when processing a VARCHAR column. As an example, we have set up an environment:

- ▶ Simple base table and its associated LOB. The LOB is 5 KB in size. The table contains 1,200 rows.
- ▶ Simple base table and its associated LOB. The LOB is 30 KB in size. The table contains 200 rows.
- ▶ Normal table that contains the same structure as the base table, except that it has a VARCHAR(5120) column. The table contains 1,200 rows.

For our comparison, we now use three different batch application programs, all of them using host variables (not locators and not file reference variables). Two of them select LOBs and the third selects the VARCHAR column into a host variable.

We now look at the accounting trace output provided by DB2 PM and reported in Figure 8-8 on page 250.

Selecting 30K LOBS		
AVERAGE	APPL(CL.1)	DB2 (CL.2)
-----	-----	-----
ELAPSED TIME	0.032034	0.012463
CPU TIME	0.015927	0.010114
Selecting 5K LOBS		
AVERAGE	APPL(CL.1)	DB2 (CL.2)
-----	-----	-----
ELAPSED TIME	0.076652	0.041317
CPU TIME	0.055328	0.039350
Selecting VARCHARs		
AVERAGE	APPL(CL.1)	DB2 (CL.2)
-----	-----	-----
ELAPSED TIME	0.045266	0.013610
CPU TIME	0.027320	0.012744

Figure 8-8 Accounting trace report for LOB and VARCHAR applications

The values shown for DB2 Class 2 elapsed time clearly indicates that when dealing with small LOBs (<32 KB), it would be better to implement the VARCHAR, VARBINARY, and VARGRAPHIC strategy. If you decide to span your LOB over several non-LOB columns (in the example, we simulate dividing a 30 KB LOB into six 5 KB VARCHAR fields), you might want to reconsider the spanning and use LOBs instead to avoid the whole process of merging VARCHARs into one whole object in your application and save some application CPU time and DB2 CPU time.

8.7 Important I/O aspects

As part of our tests, we tried to compare LOB writing between *striped* and regular LOB table spaces. Both LOB table spaces were defined as not logged to avoid LOG write contention. They were sized large enough to hold 100 5 MB LOBs in one extent. Both table spaces were assigned to a very small 32 KB buffer pool to force synchronous I/O. The striped LOB table space had 4 stripes (divided among 4 different DASD volumes).

A batch COBOL program inserts 100 5 MB LOBs into each table. Figure 8-9 on page 251 shows the output of accounting traces from the program's run.

STRIPED LOBs - 100 5Mb inserts		
AVERAGE	APPL (CL.1)	DB2 (CL.2)
-----	-----	-----
ELAPSED TIME	16.485975	16.462227
CPU TIME	1.842678	1.834109
SUSPEND TIME	0.000000	13.665551
NON STRIPED LOBs - 100 5Mb inserts		
AVERAGE	APPL (CL.1)	DB2 (CL.2)
-----	-----	-----
ELAPSED TIME	50.755741	50.731716
CPU TIME	2.036847	2.027694
SUSPEND TIME	0.000000	46.740564

Figure 8-9 Striped versus non-striped LOB table spaces

You can see that striping is extremely beneficial to performance. Though you do not save much in terms of CPU, the elapsed time is considerably shorter. In this example, the reduction of time is almost the same as the number of stripes (four), indicating that the parallel access available to the different stripes was almost entirely concurrent. However, the largest improvement is with 2 stripes. When increasing the stripes (in pairs), diminishing returns apply.

This example is an isolated test case, so in an environment with a greater workload, you would expect a smaller reduction. The value of striping is very sensitive to the speed of the device and channel. Path utilization can be the limiting factor.

Striping is especially recommended in case you are running on older disk controller technology as a means to obtain relief from performance constraints. The greatest performance gain can be seen when massive sequential reads and writes are performed, which should reflect the I/O associated with LOBs providing you are using the LOB data type for truly “large” objects.

Tip: Consider using striping for your active logs as well to speed up log writes, especially when heavy sequential data updating is involved (large batch or busy OLTP environment).

With introduction of z9 and new I/O controllers, MIDAW technology was introduced to optimize the I/O stream and maximize channel throughput. It does not increase the bandwidth of your FICON channel, but it increases the efficiency of the channels (*efficiency* in this case is the ratio of throughput to channel utilization). More information is available in the redpaper *How does the MIDAW facility improve the performance of FICON channels using DB2 and other workloads?*, REDP-4201.

8.8 IFCID enhancements for LOBs

If you have defined LOBs within your system, the need to track their performance and monitor resources used by applications retrieving LOB data can become very important. Inappropriate usage and techniques can have a dramatic effect on the system due to the sheer volume of data potentially involved. The use of LOBs can increase in the near future and getting to know the main counters within the IFCID environment can help you. Because the IFCIDs are part of your DB2 traces, you might have to start them unless you already have them active in your DB2 system.

A collection of the current IFCIDs providing information regarding LOBs is listed in Table 8-2.

Table 8-2 Current IFCIDs providing information regarding LOBs

IFCID	Fields	Description
0002	DB2 statistical data on the database services address space	
	QXSTLOBV	Maximum storage used for LOB values
	QXCRATB	Number of CREATE AUXILIARY TABLE statements
	QXHLDLOC	Number of HOLD LOCATOR statements
	QXFRELOC	Number of FREE LOCATOR statements
0003	DB2 accounting record	
	QXSTLOBV	Maximum storage used for LOB values
	QWACLRN	Number of log records written
	QWACLRAB	Total number of bytes of log records written
0018	Ends sequential scan, index scan, or insert. The additional pages scanned in a LOB table space and for a count of the number of LOBs updated. Other fields in IFCID 18 are only applicable to the base table.	
	QW0018PL	Additional pages scanned in a LOB table space
	QW0018UL	Count of LOB data pages updated (either by an SQL INSERT or an SQL UPDATE)
0020	Summary of page, row, and LOB locks held and lock escalation	
	QW0020TP	Maximum number of page, row, and LOB locks held
	QW0020PL	Maximum number of either page, row, or LOB locks held for the thread
	QW0020F5	LOB table space
	QW0020R3	LOB lock
0021	Detail lock trace LOB lock type	
	QW0021ML	LOB lock type (value '30 'x)
	QW0021KX	ID of resource for LOB locks
	QW0021K6	ROWID
	QW0021K7	Version number
0023	Record utility start information	
	QW0023PH	CHECKLOB is a new phase for the CHECK LOB utility.
0024	Record utility object or phase change record	
	QW0024PH	CHECKLOB is a new phase for the CHECK LOB utility.
0025	Record utility start information	
	QW0025PH	CHECKLOB is a new phase for the CHECK LOB utility.

IFCID	Fields	Description
0044	Records lock suspension. LOB lock type record utility start information	
	QW0044ML	LOB lock type (value '30 'x)
	QW0044KX	ID of resource for LOB locks
	QW0044K6	ROWID
	QW0044K7	Version number
0058	End of SQL statement execution. Add fields for the additional pages scanned in a LOB table space and for a count of the number of LOBs updated. Other fields in IFCID 58 are only applicable to the base table.	
	QW0058PL	Additional pages scanned in a LOB table space
	QW0058UL	Count of LOB data pages updated (either by an SQL INSERT or an SQL UPDATE)
0059	DB2 performance record. Records the start of the execution of a “fired” SQL statement.	
	QW0059CTU	Continue clause x'0000' - CONTINUE not specified x'0001' - WITH CONTINUE specified x'0002' - CURRENT CONTINUE specified
0062	DDL (and other) execution statement start	
	QW0062CX	Create auxiliary table. Value x'F2'
	QW0062HL	Hold locator. Value x'CE'.
	QW0062FL	Free locator. Value x'CF'.
0105	Maps the DBID and OBID to the database and table space name. Also maps the DBID and OBID of the LOB table space and index on the auxiliary table to the LOB table space name and name of the index on the auxiliary table. DDL (and other) execution statement start.	
0106	System initialization parameters	
	QWP1LVA	Bytes for LOB values - per user
	QWP1LVS	Bytes for LOB values - per system
	QWP1MOFR	Maximum number of concurrently open data sets for processing LOB file references. This value corresponds to "max open file refs" on installation panel DSNTIPE, or DSNZPARM name: MAXOFILR in DSN6SYSP.
0107	Open and close information. Also records open and close for LOB table spaces and indexes on auxiliary tables.	
0141	Records grants and revokes	
	QW0140BS	For a LOB table space
	QW0140BT	For an auxiliary table
0148	DB2 monitor trace record	

IFCID	Fields	Description
0150	Lock information for a given agent	
	QW0150ML	LOB lock type (value '30 'x)
	QW0150KX	ID of resource for LOB locks
	QW0150K6	ROWID
	QW0150K7	Version number
0172	Deadlock trace	
	QW0172MO	LOB lock type (value '30 'x)
	QW0172KX	ID of resource for LOB locks
	QW0172K6	ROWID
	QW0172K7	Version number
0185	Data capture information deadlock trace	
	QW0185ST	For a LOB column, this is the data type of the LOB.
	QW0185LE	For a LOB column, this is the length of the indicator column.
0196	Time-out trace. Lock types are not explicitly specified for this trace.	
	QW0196FR	This value represents a LOB lock '30 'X.
	QW0196KX	ID of resource for LOB locks
	QW0196K6	ROWID
	QW0196K7	Version number
0225	Historical storage usage. Used to determine which areas are responsible for the REAL frames buildup.	
	QW0225VA	TOTAL VARIABLE STORAGE ABOVE THE 31-BIT BAR
0306	Records log records. Note that when NOT LOGGED has been specified for a LOB table space, log records are not written for the value of the LOB.	
0321	Trace record to trace the beginning of a force-at-commit	
0322	Trace record to trace the end of a force-at-commit	
	QW0322NP	Number of pages written
0337	Lock escalation occurrences	

Traces to use with LOBs

- IFCID 58
 - QW0058PL - pages scanned in LOB table space
 - QW0058UL - pages updated in LOB table spaces
- IFCID 198, class 3
 - Getpage activity

- ▶ IFCID 321 and 322
 - Respectively begin and end force-at-commit
 - Belong to the same classes as other I/O wait events:
 - Accounting classes 3 and 8
 - Monitor classes 3 and 8
 - Performance class 4
- ▶ QXSTLOBV - Maximum storage used for LOB values
 - Value is in KB for accounting
 - Value is in MB for statistics

Tip: You can find the IFCID field descriptions in the member of the installation library *HLQ.SDSNIVPD(DSNWMSGGS)*.

8.9 DRDA LOB flow optimization performance

We have described the flow optimization for LOBs and XML data at 4.3, “DRDA LOB flow optimization” on page 79. Some indication of the performance advantages can be seen from test cases where the LOB length is varied around the `streamBufferSize` threshold, which effectively compares the old non-progressive mechanism with the new progressive streaming, using varying sized LOB workloads.

Figure 8-10 shows the effect of varying LOB data sizes with differing flow mechanisms. For small LOB sizes, the longest run time comes from using LOB locators. It is somewhat more efficient in terms of elapsed time to materialize the LOB instead of using a locator, because a degree of overhead is removed from DB2, and a reduction occurs in the number of data flows. Using progressive streaming further improves the performance by eliminating overhead involved with invoking LOB specific flow mechanisms. However, when the LOB size is increased to 80 KB, which is above our specified value of 70 KB, the same performance is seen, because the mechanisms used are the same.

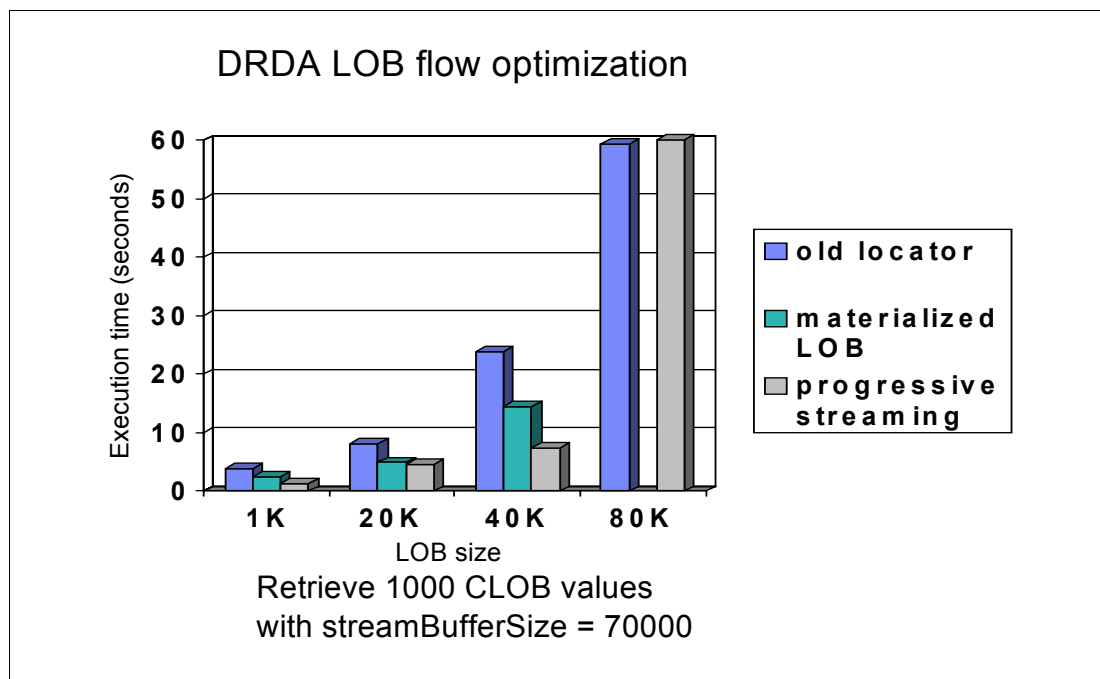


Figure 8-10 Performance of LOB Progressive Streaming

Note: The highest performance improvement is seen when selecting a large number of rows in one cursor. For singleton selects, only minimal performance improvement is seen.

8.10 LOB recommendations for performance

In this section, we summarize performance recommendations:

- There are LOBs and small LOBs (<32 KB).

With DB2 V8 in general, the larger the LOB, the greater the efficiency. LOB insert (and therefore update) processing is relatively expensive for small length LOB columns.

You must take into account that there is only one LOB per page; if you have specified a pagesize of 32 KB, and a row of 100 bytes, the overhead in dealing with each row in terms of I/O and CPU cannot be negligible.

When trying to avoid to use LOBs, keep in mind that until DB2 9, you had only CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC variables. However, DB2 9 has introduced XML and the new types BINARY and VARBINARY to maximize the flexibility of data formats.

DB2 9 also has improved the handling of small LOBs. When fetching LOBs, progressive streaming (as described in 4.3, “DRDA LOB flow optimization” on page 79) can provide improved system performance for small LOBs.

- Use the correct page size.

For really large LOBs, page size should be 32 KB. Because DB2 places only one LOB per page, space can be wasted for small LOBs (less than the page size). Use a 4 KB page for LOB table space to minimize wasted disk space and to achieve potentially much higher I/O time if both small and large LOBs exist, because this is found to be a very common situation.

Note: When estimating page size, base the decision on average LOB size and not the maximum LOB size.

- Use LOB locators to manipulate LOBs for V8 and prior versions.

For versions prior to DB2 9, we still recommend using LOB locators. They allow you to process LOB data consistently and more efficiently than using host variables and to avoid application buffering problems. When building a new application, consider coding LOB handling processes in separate code parts to enable easy conversion to file reference variables when possible.

- Use LOB file reference variables to manipulate LOBs for DB2 9.

When coding for DB2 9 and *NO* special processing is needed for LOBs (just simple retrieval and update), use file reference variables. They dramatically increase processing speed. Whether using SQL or one of the utilities, LOB file reference variables are simpler to code than other techniques for managing external LOB files and achieve better performance. Using file reference variables, the LOB does not use any storage in the application; the file I/O can be overlapped with DB2 deferred writes or with network traffic in the case of remote client applications. Also, the CPU time is less than other techniques.

- Mind the physical design.

- Do not place *LARGE* seldom updated LOBs with frequently updated data.

To avoid complicated recovery, do not put rarely, massively updated LOBs with frequently updated non-LOB data (in the same base table). For example, in case we have online *logged* updates of employee personal details, and a weekly *not logged* batch updated picture, you might want to separate this data to avoid complicated point in time recovery scenarios.

- Do not use LOBS as a means of normalizing data.

We do not recommend that you use LOBs as a technique to obtain performance improvements that normalization and proper physical design can achieve. Note that:

- Any improvements are entirely application dependent.
- At the physical design stage, you should have been able to identify the columns that are infrequently accessed and the relationship across the columns. Depending on the application, there might be a net benefit in placing some columns into its own base table, that is keyed on application unique identifier and holding the 20 KB VARCHAR and possibly other infrequently accessed columns.

The additional cost, particularly when processing small LOBs, means you are likely to create a different bottleneck from the perceived one of logging that you were trying to solve. Use LOBs only for their intended purpose.

- ▶ Use LOAD and UNLOAD for massive LOB update and retrieval.

Notice that DB2 9 introduced a lot of changes to the LOAD and UNLOAD utilities. These changes were retrofitted to earlier versions as well, making massive LOB updating and retrieval faster and more efficient.

- ▶ Tune VSAM and GRS with data sharing.

In data sharing environments, we observed some contention during drop LOB table space and unusually high Coupling Facility (CF) CPU activity. We could attribute this to VSAM making use of GRS. When we specified GRS=STAR, these problems were eliminated. We recommend that you investigate the GRS setting in data sharing environments if you are observing high CF CPU activity.

- ▶ Use REORG SHRLEVEL REFERENCE.

Starting from DB2 9, you can keep your LOBs readable even during maintenance while the objects are being reorganized.

- ▶ Use CHECK DATA and CHECK LOB with SHRLEVEL CHANGE.

CHECK DATA and CHECK LOB utilities are available with SHRLEVEL CHANGE in DB2 9.



A

Additional material

This IBM Redbook refers to additional material that can be downloaded from the Internet as described below.

Locating the Web material

The Web material associated with this IBM Redbook is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

<ftp://www.redbooks.ibm.com/redbooks/SG247270>

Alternatively, you can go to the IBM Redbooks Web site at:

ibm.com/redbooks

Select the **Additional materials** and open the directory that corresponds with the IBM Redbook form number, SG24-7270.

Using the Web material

The additional Web material that accompanies this IBM Redbook includes the following files:

CLOB examples

File name	Description
SGDBC	DDL for database and storage group
BASETSC	DDL for base table space
BASETC	DDL for base table
BASEIXC	DDL for base table unique index
AUXTSC	DDL for auxiliary table space
AUXTC	DDL for auxiliary table
AUXIXC	DDL for auxiliary index
BINDC	BIND member for BLOB samples
SAMPnC	JCL to execute SAMPLEnC, (<i>n</i> between 1 and 9, B)

SAMPLE1C	Insert a CLOB using a large host variable
SAMPLE2C	Insert a CLOB using a large host variable and one locator chain
SAMPLE3C	Insert a CLOB using a large host variable and two locator chains
SAMPLE4C	Unload a CLOB using a large host variable
SAMPLE5C	Unload a CLOB using a large host variable and one locator chain
SAMPLE6C	Delete parts of a CLOB using locators
SAMPLE7C	Update parts of a CLOB using locators
SAMPLE8C	Load a CLOB using a file-reference-variable
SAMPLE9C	Unload a CLOB using a file-reference-variable
SAMPLEAC	Update a CLOB using a file-reference-variable
SAMPLEBC	FETCH WITH/CURRENT CONTINUE example

All the above files are contained in:
CLOBSAMPLES.zip Zipped Code Samples

BLOB examples

File name	Description
SGDBB	DDL for database and storage group
BASETSB	DDL for base table space
BASETB	DDL for base table
BASEIXB	DDL for base table unique index
AUXTSB	DDL for auxiliary table space
AUXTB	DDL for auxiliary table
AUXIXB	DDL for auxiliary index
BINDB	BIND member for CLOB samples
SAMPnB	JCL to execute SAMPLEnB, (<i>n</i> between 1 and 9, B)
SAMPLE1B	Insert a BLOB using a large host variable
SAMPLE2B	Insert a BLOB using a large host variable and one locator chain
SAMPLE3B	Insert a BLOB using a large host variable and two locator chains
SAMPLE4B	Unload a BLOB using a large host variable
SAMPLE5B	Unload a BLOB using a large host variable and one locator chain
SAMPLE6B	Delete parts of a BLOB using locators
SAMPLE7B	Update parts of a BLOB using locators
SAMPLE8B	Load a BLOB using a file-reference-variable
SAMPLE9B	Unload a BLOB using a file-reference-variable
SAMPLEAB	Update a BLOB using a file-reference-variable
SAMPLEAB	FETCH WITH/CURRENT CONTINUE example

All the above files are contained in:
BLOBSAMPLES.zip Zipped Code Samples

Java CLOB example

File name	Description
JavaClobSample.zip	Sample Java program with CLOB

System requirements for downloading the Web material

The following system configuration is recommended:

Hard disk space:	8 MB minimum
Operating System:	Windows 95 or Windows NT or Windows 2000
Processor:	Intel® 386 or higher
Memory:	16 MB

How to use the Web material

Create a subdirectory (folder) on your workstation and unzip the contents of the Web material zip file into this folder. Modify JCL to comply with your library definitions.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this IBM Redbook.

IBM Redbooks

For information about ordering these publications, see “How to get IBM Redbooks” on page 265. Note that some of the documents referenced here may be available in softcopy only.

- ▶ *Data Integrity with DB2 for z/OS*, SG24-7111
- ▶ *DB2 UDB for z/OS: Design Guidelines for High Performance and Availability*, SG24-7134
- ▶ *DB2 UDB for z/OS Version 8 Performance Topics*, SG24-6465
- ▶ *Disk storage access with DB2 for z/OS*, REDP-4187
- ▶ *DB2 UDB for z/OS Version 8: Everything You Ever Wanted to Know, ... and More*, SG24-6079
- ▶ *DB2 for z/OS and OS/390 Version 7 Performance Topics*, SG24-6129
- ▶ *DB2 for z/OS Application Programming Topics*, SG24-6300
- ▶ *DB2 for z/OS Stored Procedures: Through the CALL and Beyond*, SG24-7083
- ▶ *WebSphere Information Integrator Q Replication: Fast Track Implementation Scenarios*, SG24-6487
- ▶ *SAP on DB2 Universal Database for OS/390 and z/OS: Multiple Components in One Database (MCOD)*, SG24-6914
- ▶ *How does the MIDAW facility improve the performance of FICON channels using DB2 and other workloads?*, REDP-4201

Other publications

These publications are also relevant as further information sources:

DB2 Version 8

- ▶ *Enterprise COBOL for z/OS Programming Guide*, SC27-1412-04
- ▶ *DB2 UDB for z/OS Version 8 Data Sharing: Planning and Administration*, SC18-7417-03
- ▶ *DB2 UDB for z/OS Version 8 Administration Guide*, SC18-7413-03
- ▶ *DB2 UDB for z/OS Version 8 Application Programming and SQL Guide*, SC18-7415-03
- ▶ *DB2 UDB for z/OS Version 8 Application Programming Guide and Reference for Java*, SC18-7414-02
- ▶ *DB2 UDB for z/OS Version 8 Command Reference*, SC18-7416-03
- ▶ *DB2 UDB for z/OS Version 8 Installation Guide*, GC18-7418-04
- ▶ *DB2 UDB for z/OS Version 8 Codes*, GC18-9603-01
- ▶ *DB2 UDB for z/OS Version 8 Messages*, GC18-9602-01

- ▶ *DB2 UDB for z/OS Version 8 SQL Reference*, SC18-7426-03
- ▶ *DB2 UDB for z/OS Version 8 Utility Guide and Reference*, SC18-7427-03
- ▶ *DB2 UDB for z/OS Version 8 Internationalization Guide*, available from:
<http://www.ibm.com/software/data/db2/zos/v8books.html>

DB2 Version 9.1

- ▶ *DB2 Version 9.1 for z/OS Administration Guide*, SC18-9840-00
- ▶ *DB2 Version 9.1 for z/OS Application Programming and SQL Guide*, SC18-9841-00
- ▶ *DB2 Version 9.1 for z/OS Application Programming Guide and Reference for JAVA*, SC18-9842-00
- ▶ *DB2 Version 9.1 for z/OS Codes*, GC18-9843-00
- ▶ *DB2 Version 9.1 for z/OS Command Reference*, SC18-9844-00
- ▶ *DB2 Version 9.1 for z/OS Data Sharing: Planning and Administration*, SC18-9845-00
- ▶ *DB2 Version 9.1 for z/OS Diagnosis Guide and Reference*, LY37-3218-00
- ▶ *DB2 Version 9.1 for z/OS Diagnostic Quick Reference*, LY37-3219-00
- ▶ *DB2 Version 9.1 for z/OS Installation Guide*, GC18-9846-00
- ▶ *DB2 Version 9.1 for z/OS Introduction to DB2*, SC18-9847-00
- ▶ *DB2 Version 9.1 for z/OS Licensed Program Specifications*, GC18-9848-00
- ▶ *DB2 Version 9.1 for z/OS Messages*, GC18-9849-00
- ▶ *DB2 Version 9.1 for z/OS ODBC Guide and Reference*, SC18-9850-00
- ▶ *DB2 Version 9.1 for z/OS Performance Monitoring and Tuning Guide*, SC18-9851-00
- ▶ *DB2 Version 9.1 for z/OS RACF Access Control Module Guide*, SC18-9852-00
- ▶ *DB2 Version 9.1 for z/OS Reference for Remote DRDA Requesters and Servers*, SC18-9853-00
- ▶ *DB2 Version 9.1 for z/OS Reference Summary*, SX26-3854-00
- ▶ *DB2 Version 9.1 for z/OS SQL Reference*, SC18-9854-00
- ▶ *DB2 Version 9.1 for z/OS Utility Guide and Reference*, SC18-9855-00
- ▶ *DB2 Version 9.1 for z/OS What's New?*, GC18-9856-00
- ▶ *DB2 Version 9.1 for z/OS XML Extender Administration and Programming*, SC18-9857-00
- ▶ *DB2 Version 9.1 for z/OS XML Guide*, SC18-9858-00
- ▶ *DB2 Version 9.1 for z/OS Internationalization Guide*, available from:
<http://www.ibm.com/software/data/db2/zos/v9books.html>

Online resources

These Web sites are also relevant as further information sources:

- ▶ Unicode site
<http://www.unicode.org/>
- ▶ The Unicode character code charts
<http://www.unicode.org/charts/>

How to get IBM Redbooks

You can search for, view, or download Redbooks, Redpapers, Hints and Tips, draft publications and Additional materials, as well as order hardcopy Redbooks or CD-ROMs, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Abbreviations and acronyms

AC	autonomic computing	CRD	collect report data
ACS	automatic class selection	CRUD	create, retrieve, update or delete
AIX	Advanced Interactive eXecutive from IBM	CSA	common storage area
APAR	authorized program analysis report	CSF	Integrated Cryptographic Service Facility
API	application programming interface	CTE	common table expression
AR	application requester	CTT	created temporary table
ARM	automatic restart manager	CUoD	Capacity Upgrade on Demand
AS	application server	DAC	discretionary access control
ASCII	American National Standard Code for Information Interchange	DASD	direct access storage device
B2B	business-to-business	DB	database
BCDS	DFSMSHsm backup control data set	DB2	Database 2™
BCRS	business continuity recovery services	DB2 PE	DB2 Performance Expert
BI	Business Intelligence	DBA	database administrator
BLOB	binary large objects	DBAT	database access thread
BPA	buffer pool analysis	DBCLOB	double-byte character large object
BSDS	boot strap data set	DBCS	double-byte character set
CBU	Capacity BackUp	DBD	database descriptor
CCA	channel connection address	DBID	database identifier
CCA	client configuration assistant	DBM1	database master address space
CCP	collect CPU parallel	DBRM	database request module
CCSID	coded character set identifier	DCL	data control language
CD	compact disk	DDCS	distributed database connection services
CDW	central data warehouse	DDF	distributed data facility
CEC	central electronics complex	DDL	data definition language
CF	coupling facility	DDL	data definition language
CFCC	coupling facility control code	DES	Data Encryption Standard
CFRM	coupling facility resource management	DLL	dynamic load library manipulation language
CICS®	Customer Information Control System	DML	data manipulation language
CLI	call level interface	DNS	domain name server
CLOB	character large object	DPSI	data partitioning secondary index
CLP	command line processor	DRDA	Distributed Relational Data Architecture
CMOS	complementary metal oxide semiconductor	DSC	dynamic statement cache, local or global
CP	central processor	DSNZPARMs	DB2's system configuration parameters
CPU	central processing unit	DSS	decision support systems
CRCR	conditional restart control record	DTT	declared temporary tables

DWDM	dense wavelength division multiplexer	ICSF	Integrated Cryptographic Service Facility
DWT	deferred write threshold	IDE	integrated development environments
EA	extended addressability	IFCID	instrumentation facility component identifier
EAI	enterprise application integration	IFI	Instrumentation Facility Interface
EAS	Enterprise Application Solution	IFL	Integrated Facility for Linux
EBCDIC	extended binary coded decimal interchange code	IGS	IBM Global Services
ECS	enhanced catalog sharing	IMS	Information Management System
ECSA	extended common storage area	IORP	I/O Request Priority
EDM	environmental descriptor manager	IPLA	IBM Program Licence Agreement
EJB™	Enterprise JavaBean	IRD	Intelligent Resource Director
ELB	extended long busy	IRLM	internal resource lock manager
ENFM	enable-new-function mode	IRWW	IBM Relational Warehouse Workload
ERP	enterprise resource planning	ISPF	interactive system productivity facility
ERP	error recovery procedure	ISV	independent software vendor
ESA	Enterprise Systems Architecture	IT	information technology
ESP	Enterprise Solution Package	ITR	internal throughput rate, a processor time measure, focuses on processor capacity
ESS	Enterprise Storage Server®	ITSO	International Technical Support Organization
ETR	external throughput rate, an elapsed time measure, focuses on system capacity	IVP	installation verification process
EWLC	Entry Workload License Charges	J2EE	Java 2 Enterprise Edition
EWLM	Enterprise Workload Manager	JDBC	Java Database Connectivity
FIFO	first in first out	JFS	journaled file systems
FLA	fast log apply	JNDI	Java Naming and Directory Interface
FTD	functional track directory	JTA	Java Transaction API
FTP	File Transfer Program	JTS	Java Transaction Service
GB	gigabyte (1,073,741,824 bytes)	JVM™	Java Virtual Machine
GBP	group buffer pool	KB	kilobyte (1,024 bytes)
GDPS®	Geographically Dispersed Parallel Sysplex™	LCU	Logical Control Unit
GLBA	Gramm-Leach-Bliley Act of 1999	LDAP	Lightweight Directory Access Protocol
GRS	global resource serialization	LOB	large object
GUI	graphical user interface	LPAR	logical partition
HALDB	High Availability Large Databases	LPL	logical page list
HPJ	high performance Java	LRECL	logical record length
HTTP	Hypertext Transfer Protocol	LRSN	log record sequence number
HW	hardware	LRU	least recently used
I/O	input/output	LSS	logical subsystem
IBM	International Business Machines Corporation	LUW	logical unit of work
ICF	internal coupling facility		
ICF	integrated catalog facility		
ICMF	integrated coupling migration facility		

LVM	logical volume manager
MAC	mandatory access control
MB	megabyte (1,048,576 bytes)
MBps	megabytes per second
MLS	multi-level security
MQT	materialized query table
MTBF	mean time between failures
MVS	Multiple Virtual Storage
NALC	New Application License Charge
NFM	new-function mode
NFS	Network File System
NPI	non-partitioning index
NPSI	nonpartitioned secondary index
NVS	non volatile storage
ODB	object descriptor in DBD
ODBC	Open Database Connectivity
ODS	Operational Data Store
OLE	Object Link Embedded
OLTP	online transaction processing
OP	Online performance
OS/390	Operating System/390®
OSC	optimizer service center
PAV	parallel access volume
PCICA	Peripheral Component Interface Cryptographic Accelerator
PCICC	PCI Cryptographic Coprocessor
PDS	partitioned data set
PIB	parallel index build
PPRC	Peer-to-Peer Remote Copy
PR/SM™	Processor Resource/System Manager
PSID	pageset identifier
PSP	preventive service planning
PTF	program temporary fix
PUNC	possibly uncommitted
PWH	Performance Warehouse
QA	Quality Assurance
QMF	Query Management Facility
QoS	Quality of Service
QPP	Quality Partnership Program
RACF®	Resource Access Control Facility
RAS	reliability, availability and serviceability
RBA	relative byte address
RBLP	recovery base log point

RDBMS	relational database management system
RDS	relational data system
RECFM	record format
RI	Referential Integrity
RID	record identifier
ROI	return on investment
RPO	recovery point objective
RR	repeatable read
RRS	resource recovery services
RRSAF	resource recovery services attach facility
RS	read stability
RTO	recovery time objective
RTS	real-time statistics
SAN	storage area networks
SBCS	store single byte character set
SCUBA	self contained underwater breathing apparatus
SDM	System Data Mover
SDP	Software Development Platform
SLA	service-level agreement
SMIT	System Management Interface Tool
SOA	service-oriented architecture
SOAP	Simple Object Access Protocol
SPL	selective partition locking
SQL	Structured Query Language
SQLJ	Structured Query Language for Java
SRM	Service Request Manager
SSL	Secure Sockets Layer
SU	Service Unit
TCO	total cost of ownership
TPF	Transaction Processing Facility
UA	Unit Addresses
UCB	Unit Control Block
UDB	Universal Database
UDF	user-defined functions
UDT	user-defined data types
UOW	unit of work
UR	unit of recovery
USS	UNIX System Services
vCF	virtual coupling facility
VIPA	Virtual IP Addressing
VLDB	very large database

VM	virtual machine
VSE	Virtual Storage Extended
VSIP	Visual Studio® Integrator Program
VWLC	Variable Workload License Charges
wizards	Web-based assistants
WLC	Workload License Charges
WLM	Workload Manager
WSDL	Web Services Definition Language
WTO	write to operator
XA	Extended Architecture
XML	Extensible Markup Language
XRC	eXtended Remote Copy
z800	zSeries 800
z890	zSeries 890
z990	zSeries 990
zAAP	zSeries Application Assist Processor
zELC	zSeries Entry License Charge

Index

Numerics

00C900A3 205
00C900D0 205
00C900Dx 244
-423 138
-857 32
-904 201

A

ABAP 126–129
ABAP reports 131–132
ACHKP 196, 200, 229
adding a LOB column 37–38
adding a ROWID 42
Advanced Business Application Programming 126, 129
Altering table 211
APIs 20
application programming xvii, 2, 107, 126
ASCII 33, 54–55, 84, 147, 164, 172
Automatic creation of objects xvii, 6, 24–26, 29, 182, 217
autorel duration 102
aux table 183
AUX WARNING state 50, 227, 231
AUXERROR 201
AUXERROR INVALDATE 201–202
AUXERROR REPORT 201–202
auxiliary check pending 196–197
auxiliary check pending state 200
auxiliary column 13, 39
auxiliary index 14, 24–26, 28–31, 34, 36, 39, 103, 148, 177–178, 185–186, 195, 216, 259–260
DDL 36
auxiliary LOB table 39
auxiliary table 12–15, 24, 39, 61, 69, 71–72, 83, 87, 90, 96, 105, 143, 150, 213–216, 218, 252–253, 259–260
auxiliary warning 196
auxiliary warning state 200
AUXONLY 201
AUXTBNAME 213
AUXTBOWNER 213, 217
AUXW 47, 50, 196, 200
AVGSIZE 214–215

B

backup and recovery 211
base table 12–15, 24, 31, 39, 44, 61, 69, 71–72, 83, 87, 90, 95–96, 143, 156, 160, 212–216, 259–260
base table locks 95
Binary Large Objects 10
BIT 11, 33, 41, 54–55, 119, 131, 242, 254
BLOB 2, 10–11, 17–20, 24–25, 31–34, 37, 61, 66, 71, 79, 107–108, 114, 117, 119, 121, 126, 128–129, 142, 145–146, 160–161, 163, 212, 214, 217–218, 259–260

BLOB_FILE 18
BLOBs 11–12, 33, 54, 88, 108–110, 120, 126
BP40. 144
BPOOL 216
buffer pools 24, 35, 56–57, 123, 144, 147, 238, 242, 245

C

Call Level Interface 79
CARDF 215–216
CAST 11, 71–72
CCSID 11, 19, 21, 27, 33, 54–55, 84, 146, 160, 164, 172, 212, 238, 241
Character Large Objects 10
check constraints 35, 40, 61
CHECK DATA xvii, 7, 123, 196, 199, 257
CHECK DATA SHRLEVEL CHANGE 202
CHECK INDEX 207
CHECK INDEX SHRLEVEL CHANGE 207
CHECK INDEX SHRLEVEL REFERENCE 207
CHECK LOB 204
CHECK LOB SHRLEVEL CHANGE 205
CHECK LOB SHRLEVEL REFERENCE 205
CHECK LOB sort 6
check pending 196
CHECK-pending 204, 207
CHKP 196
choosing a page size 57
chunk 62–64, 78, 112, 117, 133, 191
chunking 142
CLI 79
CLOB 10–11, 17–19, 24–25, 31, 33–34, 55, 66, 71–72, 84, 87–88, 107, 109, 114, 120–121, 126, 129, 136, 141, 145, 164, 171–172, 214, 218, 259–260
CLOB_FILE 18
codepage 147
COLNAME 213
COLTYPE 32, 214
compression 27, 35, 61, 147, 242
CONCAT 70, 73, 78, 89, 91, 121–122
CONCURRENT copy 178–180
COPY 47, 49, 51, 100, 103, 139, 146, 160–161, 163, 166, 212–213, 218–219
COPY NO 30
COPYDDN 173, 177, 179–180, 185, 189
COPYLASTTIME 218
COPYTOCOPY 180–181
COPYUPDATEDPAGES 219
creating a LOB table space 34
creating a new table with ROWID 41
creating auxiliary table 35
creating LOBs 36
DDL statement 31
CREATOR 216
Cross Loader xvii, 5, 83, 85, 94, 171–172, 176

- cross-loader xvii
- CS 103, 111, 135, 151
- CURRENT RULES 24, 29–31, 60, 112, 146
 - impact on CREATE and DROP 29
- CURRENT RULES = 'STD' 29
- CURRENT RULES DB2 29, 60
- CURRENT RULES STD 30, 60, 112
- CURRENTAPPENSCHM 21
- cursor stability 111

D

- data conversion 52, 54
- Data length 19
- data propagation 62
- data sharing 60, 102–103, 123, 156, 246, 248, 257
- database management system xvii, 93
- DATASIZE 218
- DB2 Connect 83, 93, 127, 135–136, 141
- DB2 Extender 5
- DB2-generated file reference variable construct 19
- DBCLOB 10–11, 17–19, 55, 66, 72, 84, 107, 114, 117, 119, 126, 128–129, 141, 146–147, 164, 171–172, 214
- DBCLOB_FILE 18
- DBMS xvii, 126, 144, 147, 149
- DBNAME 216
- dbsl 151
- dbsl profile 139
- DBSL trace 150
- DBSNDB06 217
- DDL 24–25, 126, 139, 145–147, 160, 174, 212, 253, 259–260
- Defective LOBs 204
- deferred write threshold 244
- delete 34, 44, 50, 56, 75, 97–99, 111, 120–121, 138, 174, 176, 185, 239, 260
- deleting a LOB 49, 56, 98, 103
- deleting part of a LOB 120
- DFSMS 59, 178
- DFSMSDSS 206
- displaying LOB objects 37
- Distributed 127
- Distributed Data Facility 127
- Double Byte Character Large Objects 10
- DRDA flow optimization 240
- DRDA LOB flow optimization 6
- Dropping implicitly created objec 26
- DSN1505I 233
- DSN1COPY 210
- DSN1PRNT 210
- DSNACCOR 195
- DSNDB06 219
- DSNJU003 234
- DSNT408I 32
- DSNTEJ7 7
- DSNTEP2 113
- DSNTIAUL 7, 83–84, 169–170
- DSNTIJMS 141
- DSNTIP7 243
- DSNTIPE 21
- DSNU1178I 173

- DSNU1218I 162
- DSNU1504I 197
- DSNU1505I 197
- DSNU297I 195
- DSNU406I 190
- DSNU599I 223
- DSNU743I 205
- DSNUM 215
- DSNZPARM 27–29, 243
- DSSIZE 27–28, 30, 34, 58–59, 161, 199, 213
- DWQT 58, 244
- Dynpro 126, 129, 137
- Dynpros 129–130

E

- efficiency 251
- encoding systems 53
- Enterprise Resource Planning 125
- Exclusive LOB lock 61
- exclusive LOB lock 95
- extenders 2–3, 83
- EXTENTS 218
- EXTENTS, 215

F

- FETCH xvii, 60, 102, 112, 114, 135–137, 238–239, 242
- fetch 60, 66, 112–114, 135, 137, 151, 242
- FETCH CONTINUE 6, 113–114, 242
- File name length 19
- file option variable 20
- File reference variables xvii, 5, 7, 10, 18–19, 85, 108, 123, 160, 164, 168, 239, 241, 256
- file reference variables 107
- FOR BIT DATA 11
- FREE LOCATOR 66, 74, 89, 92, 110, 121, 137, 155, 252–253
- FREESPACE 34, 191, 193, 214–215
- full Imag Copy 49

G

- GBPCACHE 28, 30, 34, 59, 123, 143, 146, 160–161, 212–213, 246
- GBPCACHE SYSTEM 60, 123, 146, 161, 213, 246
- GENERATED ALWAYS 38, 40–42, 161, 170, 176, 212
- GENERATED BY DEFAULT 24, 34, 38, 40–41, 94, 177
- GRS 257

H

- HIDDEN 214
- Hierarchical File System 21
- HISTORY SPACE 215
- host variable 16–19, 42, 60, 66, 71, 73, 85–87, 96, 107, 109–110, 122, 238–239, 260
- host variables 10, 18, 66–67, 70, 87, 91, 107, 110, 112, 117, 138, 256
- HURBA 193, 214

I

IAV Extenders 3
ICLI 127
ICTYPE 48
II13767 123
Image Copy 50, 173, 178, 181, 219
IMPLICIT 216
INCURSOR 176
INSERT 32–33, 44, 50–51, 75, 86–89, 92, 122–123, 134, 136, 139, 230, 241, 260
insert 5, 19, 32–34, 44, 50, 67, 79, 87–89, 97, 99, 107, 110, 122–123, 133, 136, 138–140, 177, 189, 195, 240–241, 252–253, 260
inserting LOBs
 using a host-variable 86
 using locators 88
intent exclusive 97
intent-share lock 96
internal resource lock manager 99
Invalid LOBs 200, 204
IRLM 61, 99–100
IS xvii, 2, 52, 54–55, 61, 63, 83, 87–88, 96, 107–109, 120, 126–127, 159–160, 212, 238–239, 259, 261
ISO-10646 54
ISOLATION (UR) 56, 97
IX 25, 37, 97–99, 221, 225

J

Java 5, 79, 81, 83, 126–128, 133
Java enhanced LOB 6
Java stored procedures 218
JCC driver 140–141
JDBC driver 113, 127, 140–142
JDBC functions 5

L

large objects xvii, 1–4, 9–10, 12, 15–16, 18, 52–53, 57, 60–61, 70–71, 78, 83, 95, 107, 123, 131, 136, 217
LENGTH 5, 11, 20, 25, 30, 32–33, 37, 39, 41, 47, 55, 57, 61, 70, 72–73, 80, 86–88, 100, 108–110, 114, 120, 127, 131, 160, 162, 165–166, 169, 214, 254
LENGTH2 41, 78, 214
LIKE 2, 11, 33, 38, 53, 56, 61, 63, 70, 72–73, 126, 160, 166, 169, 212, 220, 240
list 70–71, 94, 112, 119, 122, 144, 178–181
list prefetch 112
LISTDEF 178–181, 197
LISTDEFS 235
LOAD xvii, 5, 7, 21, 23, 40, 45, 83, 85, 93, 123, 128–129, 131, 165, 169–171, 215, 218, 257, 260
loading a LOB column 83
LOB column 10, 12–13, 15, 20, 24, 26, 29–30, 39, 61, 66, 70–71, 83, 85, 87, 99, 107, 113, 116, 119, 126, 130, 132, 143, 155, 171, 173, 178, 213–214, 216, 254
LOB data 7, 15–16, 19, 31, 39, 44, 66–67, 74, 83, 87, 107, 109–110, 136, 144, 149–151, 160, 162, 239–240, 244
LOB database 7, 34
LOB indicator 14, 33–34, 178, 182, 196

LOB locators 10, 16–18, 20, 66, 71, 74, 88, 90, 107, 112–114, 134, 141, 239, 256
LOB lock 56, 74–75, 95–97, 111–112, 252–254
LOB lock serialization 97
LOB locks 95
LOB locks avoidance 6
LOB Manager 72, 112, 238, 243
LOB manipulation 120
LOB map page 63–64
LOB materialization 18, 55, 89, 238–242
LOB table 13, 15, 24, 39, 95–100, 123, 143–144, 146, 148, 154, 212, 214–215, 247
LOB table space 6, 12–13, 15, 24, 26, 39, 61–63, 97–100, 102, 123, 143, 148, 160, 212, 214–216, 238, 240, 247
 DDL statement 34
LOBFILE 170–171
LOBs xvii, 1–2, 4, 6, 10–11, 15, 17–18, 20, 23, 39, 54–55, 65–66, 70, 83, 107, 109, 112–113, 121, 125–127, 159, 211–212, 215, 237–238, 241–242
 allocation 49, 58, 186
 delete 121, 196–197, 201
 insert 45, 88, 102, 121, 139, 252
 processing 79, 87, 102, 109, 245, 247
 recommendations 256
 update 45, 56, 61, 102, 173, 256
LOBs creation 24
LOBVALA 5, 86, 144, 243
LOBVALS 5, 86, 144, 243–244
LOCATE 209
locator 74
locators 16–17, 66, 71, 73, 78, 80, 85, 88, 107, 109–110, 112, 120–121, 133–134, 137, 239–240, 260
 COBOL syntax 17, 20
 concatenating 78
 concatenation 91
 materialization 77
 multiple units of work 76
 precompiler conversion 17, 20
 types 17–18
Locking
 DB2 9 101
 DB2 V8 95
locking 56, 61, 94–95, 97, 100, 111, 131, 147, 153
LOCKMAX 143, 146, 160–161, 212–213, 216
LOCKRULE 216
locks xvii, 56, 61, 74, 76, 88, 96–97, 111, 154, 156, 216, 252–254
locks with DELETE 98, 105
locks with INSERT 97, 105
locks with UPDATE 99, 105
LOCKSIZE LOB 56, 143, 146, 161, 213
LOCKSIZE TABLESPACE 56, 95
LOG 216
LOGGED 211
 combinations of attributes 45
logging 44–45, 122–123, 147, 173, 178, 185, 187, 190, 217, 248
Logical Page List 197
Longer SQL statement 6

LPL 48, 197

M

maintenance xvii, 4, 123, 126–127, 148
map page 50–52, 63–64, 103–104
mass delete 97–99
mass delete statements 97
materialization 16, 18, 55, 73, 77, 89, 110, 150, 238–240
 INSERT 239–240
 SELECT 239
MAXOFLR 21
MCOD 128–129
MGEXTSZ 148
Missing LOBs 200
Multiple Components in One Database 128

N

NACTIVE 218
NAME, 216
NetWeaver 126
NOT LOGGED 34, 44–45, 49, 160–161, 211–213, 247
NOT LOGGED LOB table space 6, 46, 230, 232
NOT PADDED 146
NPAGESF 216
NULL 20, 25, 31, 33, 38, 40–41, 71, 84, 100, 116, 135, 146, 160, 166, 212, 217

O

object-relational 2
occurrence of a string 71, 119
Online CHECK DATA 7
Online CHECK LOB 6
optimization xvii, 125, 137–138
ORGRATIO 148, 183, 185, 191–192, 214–215
Orphan LOBs 200
Out-of-synch LOBs 200

P

PADDED 146
page size 15, 27–28, 39, 57–58, 63, 256
pageset structure 63, 111
PARTITION 213
Partition Data Set 21
Partition Data Set Extended 21
partition-by-growth table spaces 29
partitioned base table 13, 35, 59, 178, 182, 184, 186, 189, 197, 213
partitioning 35, 40, 44
partitions 13, 15–16, 40, 44, 59, 61, 195
performance xvii, 4, 15, 17, 55–56, 66, 70, 82, 91–92, 101, 113, 125–126, 128, 164, 185, 239, 242, 255
PK10278 5, 21
PK13287 22
PK22887 5, 157
PK22910 21
PK25241 5, 157
PK27029 166
PK29281 22

PK29750 193
POSSTR 11, 70–71, 73, 114, 119–122
PQ90263 5, 85
PQ96956 207
PRIQTY 34, 36, 58–59, 143, 160–161, 185, 187, 199, 212–213
progressive references 136, 141–142
Progressive Streaming 136, 141–142, 255

Q

QUIESCE 178–179, 181–182

R

RBDP 196
read performance 248
Real 212
Real Time Statistics 194, 211–212, 218
REBUILD INDEX 197
rebuild pending 196
REBUILD-pending 204, 207
RECLENGTH 215–216
record identifier 32
RECOVER-pending 204, 207
recovery pending 205
RECP 48, 205
Redbooks Web site 259, 265
 Contact us xx
RELCREATED 213
Reordered Row Format 41
REORG xvii, 6, 123, 148, 218, 257
REORGDELETES 218
REORGDISORGLob 218
REORGINSERTS 218
REORGMASDELETE 218
REORGUPDATES 218
REPAIR 208
REPORT 36, 182–184, 242
REPORT TABLESPACESET 182
REPORT utility 182
resource 00000907 244
RID 32
row locking 131
ROWID 5, 11, 13–15, 24–26, 30–31, 33, 39–42, 61, 94, 144, 147, 149, 160–161, 170–171, 176–177, 192, 212, 214
 adding the column 38
 assigning 32
 GENERATED BY DEFAULT 38, 40, 177
ROWIDs 10
RTS 194
RUNSTATS 183, 214–216

S

Samples for LOBs 6
SAP
 ABAP 129
 CLI array input chaining 138
 CLI streaming interface 134

- Computing Center Management System 148
- dbsl_lib profile parameter 154
- DSNZPARMs 144
- Integrated Call Level Interface 127
- LOB INSERT performance 157
- LOB UPDATE performance 157
- LOB usage 126
- locator access 133
- measurement results 155
- monitoring and tracing 150
- performance measurements 150
- portability 149
- programming techniques for ABAP 133
- programming techniques with JDBC 140
- query rewrite 136
- REORG 148
- ROWID 144
- SGEN 138
- Unicode 147
- SAP Cluster 128
- SAP Data Dictionary 144
- SAP Enterprise 128
- SAP NetWeaver 2004s 128
- SAP Web Application Server 126, 131
- SECQTY 34, 36, 58–59, 143, 160–161, 185, 187, 199, 212–213
- SELECT SUBSTR 111
- sequential files 21
- Shared LOB lock 61
- SHRLEVEL 123, 148, 177–180, 257
- SHRLEVEL options 178
- side 79
- single LOB DELETE 98
- singleton 44, 256
- singleton delete 98
- S-LOB 56, 74, 95, 97–101, 103, 111–112
- S-LOB lock 56, 61, 74, 96–97, 99–100, 102, 106, 111–112
- space map pages 44, 49, 63, 105, 196, 246
- SPACEF 215–216
- SPACEF, 216
- spanning pages 15
- SPUFI 113
- SQL accounting 249
- SQL stored procedures 6, 73
- SQLCODE - 904 201
- SQLCODE -104 21
- SQLCODE -171 18
- SQLCODE -747 34
- SQLCODE -763 45
- SQLCODE -766 69
- SQLCODE -767 36
- SQLCODE -803 38
- SQLCODE -904 50, 59, 244
- SQLCODE-452 176
- SQLSTATE 560A1 45
- STAR 243, 257
- STATSDELETES 218
- STATSINSERTS 218
- STATSMASSDELETE 218

- STATSUPDATES 218
- STATUS 25, 37, 47–48, 72, 173, 178, 185–186, 189, 191, 216, 221
 - values X and I 216
- STORES 10, 24, 35, 57, 85, 103, 173
- STOSPACE 216
- STYPE 48
- SUBSTR 11, 70–71, 73, 80, 110–111, 120–122
- SYSIBM.SYSAUXRELS 213, 217
- SYSIBM.SYSCOLUMNS 183–184, 214
- SYSIBM.SYSCOPY 48
- SYSIBM.SYSINDEXSPACESTATS 219
- SYSIBM.SYSJARCLASS_SOURCE 218
- SYSIBM.SYSJARDATA 218
- SYSIBM.SYSLGRNX 48
- SYSIBM.SYSLOBSTATS 184, 191, 214–215
- SYSIBM.SYSLOBSTATS_HIST 215
- SYSIBM.SYSROUTINESTEXT 218
- SYSIBM.SYSSTRINGS 54–55
- SYSIBM.SYSTABLEPART 215–216
- SYSIBM.SYSTABLEPART_HIST 215
- SYSIBM.SYSTABLES 34, 182, 184, 203, 206, 215–216
- SYSIBM.SYSTABLES_HIST 216
- SYSIBM.SYSTABLESPACE 46, 216
- SYSIBM.SYSTABLESPACESTATS 218
- SYSIBM.XSR 218
- SYSTABLEPART 184, 191, 215–216

T

- table space
 - partition sizes 58
- table space scans 15
- TABLESTATUS 34, 216
 - values L and P 216
- TBNAME 36, 213–214, 217
- TBOWNER 213
- TEMPLATES 178
- Templates 5
- Text Extender 3
- The 216
- TOTALROWS 218
- traces 150–152, 242, 251, 254
- transparent ROWID 144, 147, 149
- triggers 1–3, 61
- TRUNCATE 160, 163, 169
- TS 25, 37, 47, 59, 168, 221, 225
- TYPE 216

U

- UCS-2 54
- UCS-4 54
- UDFs 1–2, 70, 73
- UDTs 1–2
- UK03226 5
- UK03227 5
- UK13720 5, 165, 172
- UK13721 5, 165, 172
- UK15036 5
- UK15037 5

- undo log records 49
- UNICODE 54–55, 78, 84, 146–147, 164, 172
- Unicode 54–55, 72, 78, 126, 134, 144, 147
- UNICODE support 55
- UNIQUE 24, 26, 54, 146, 161, 168, 212–213, 257, 259–260
- UNLOAD xvii, 5, 21, 83–84, 107–109, 123, 160–162, 164, 257, 260
- unload parts of LOB using locators 112
- unloading a LOB 107
- unloading a LOB using a host variable 109
- unloading an entire LOB using locators 109
- UPDATE 5, 19, 34, 40, 42, 44, 47, 50–52, 61, 90, 95, 99, 102–103, 105, 120–122, 134, 186, 196–197, 201, 239, 252–253, 260
- user defined functions 1–2
- user defined types 2
- UTF-16 54–55, 72
- UTF-8 11, 54–55, 72, 78
- UTRW 206, 208

V

- VALUE 11, 14–16, 18, 27, 30, 32–33, 61–62, 64, 70–71, 76, 85–87, 107–109, 120–121, 164, 214, 218, 238–239, 243–244
- VARCHAR 2, 10, 12, 32–33, 36, 41–43, 55, 72, 79–80, 84, 131, 136, 145, 147, 160, 164–165, 212, 214, 217, 256
- VARGRAPHIC 2, 55, 72, 146–147, 169, 256
- VDWQT 58, 244
- version number 36, 61, 205–206, 252–254
- vertical deferred write threshold 244
- VPSEQT 244

W

- write performance 248

X

- X-LOB 56, 97, 102, 105
- X-LOB lock 56, 61, 95, 99, 103, 105–106
- XML 3, 72, 81–82, 113–114, 142, 145, 177, 201, 218
- XML support 3
- XML2CLOB 5, 72

Z

- z/OS Conversion Services 55



LOBs with DB2 for z/OS: Stronger and Faster

(0.5" spine)

0.475" <-> 0.873"

250 <-> 459 pages



Redbooks

LOBs with DB2 for z/OS: Stronger and Faster

Define LOBs, see how they work, and see how to store them

Manage LOBs in operational environments

Use LOBs in your applications and with SAP solutions

The requirements for a database management system (DBMS) have included support for very large and complex data objects.

DB2 UDB for OS/390 Version 6 introduced the support for large objects (LOBs): they can contain text documents, images, or movies, and can be stored directly in the DBMS with sizes up to 2 gigabytes per object and 65,536 TB for a single LOB column in a 4,096 partition table. The introduction of these new data types has implied some changes in the administration processes and programming techniques. The Redbook Large Objects with DB2 for z/OS and OS/390, SG24-6571, introduced and described the usage of LOBs with DB2 for z/OS at Version 7 level.

Major enhancements for LOB manipulation have been introduced with DB2 UDB for z/OS Version 8 and DB2 Version 9.1 for z/OS (DB2 9 in this IBM Redbook). These enhancements include performance functions such as the avoidance of LOB locks and DRDA LOB flow optimization, usability functions such as file reference variables, FETCH CONTINUE, and the automatic creation of objects. DB2 utilities provide integrated support with LOAD and UNLOAD, cross-loader, REORG, CHECK DATA, and CHECK LOB.

In this IBM Redbook, we provide a totally revised description of the DB2 functions for LOB support as well as useful information about how to design and implement them. We also offer examples of their use, programming considerations, and the enhanced processes used for their administration and maintenance. We also detail how SAP solutions use LOBs. This IBM Redbook replaces the previous IBM Redbook Large Objects with DB2 for z/OS and OS/390, SG24-6571, for DB2 Version 8 and Version 9.1.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks

SG24-7270-00

ISBN 0738496936