

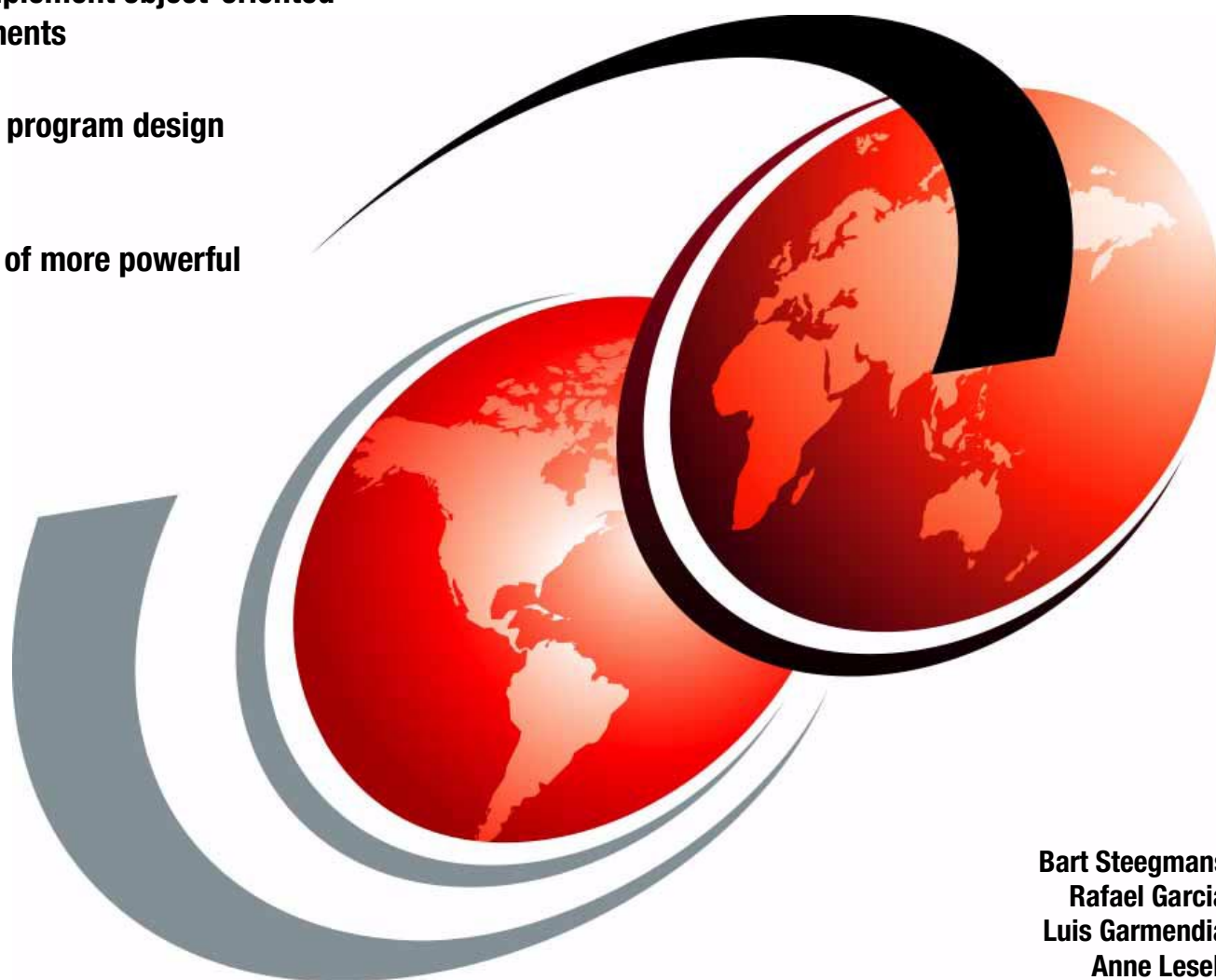
DB2 for z/OS

Application Programming Topics

How to implement object-oriented enhancements

Increased program design flexibility

Examples of more powerful SQL



Bart Steegmans
Rafael Garcia
Luis Garmendia
Anne Lesell



International Technical Support Organization

DB2 for z/OS Application Programming Topics

October 2001

Take Note! Before using this information and the product it supports, be sure to read the general information in “Special notices” on page 257.

First Edition (October 2001)

This edition applies to Version 7 of IBM DATABASE 2 Universal Database Server for z/OS and OS/390 (DB2 for z/OS and OS/390 Version 7), Program Number 5675-DB2.

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. QXXE Building 80-E2
650 Harry Road
San Jose, California 95120-6099

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

© Copyright International Business Machines Corporation 2001. All rights reserved.

Note to U.S Government Users - Documentation related to restricted rights - Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Contents

Contents	iii
Figures	ix
Tables	xi
Examples	xiii
Preface	xvii
The team that wrote this redbook	xvii
Special notice	xviii
IBM trademarks	xix
Comments welcome	xix
Chapter 1. Introduction	1
Part 1. Object-oriented enhancements	5
Chapter 2. Schemas	7
2.1 What is a schema?	8
2.2 Schema characteristics	8
2.2.1 Authorizations on schemas	8
2.2.2 Schema path and special register	9
2.2.3 How is a schema name determined?	10
2.3 The schema processor	10
Chapter 3. Triggers	13
3.1 Trigger definition	14
3.2 Why use a trigger	14
3.3 Trigger characteristics	16
3.3.1 Trigger activation time	17
3.3.2 How many times is a trigger activated?	18
3.3.3 Trigger action condition	19
3.3.4 Trigger action	19
3.3.5 Transition variables	20
3.3.6 Transition tables	21
3.4 Allowable combinations	22
3.5 Valid triggered SQL statements	22
3.6 Invoking stored procedures and UDFs	23
3.7 Setting error conditions	24
3.8 Error handling	26
3.9 Trigger cascading	28
3.10 Global trigger ordering	29
3.11 When external actions are backed out	30
3.12 Passing transition tables to SPs and UDFs	30
3.13 Trigger package	32
3.14 Rebinding a trigger package	33
3.15 Trigger package dependencies	33
3.16 DROP, GRANT, and COMMENT ON statements	34
3.17 Catalog changes	35

3.18	Trigger and constraint execution model	35
3.19	Design considerations	37
3.20	Some alternatives to a trigger	38
3.21	Useful queries	41
3.22	Trigger restrictions	42
Chapter 4. User-defined distinct types (UDT)		43
4.1	Introduction	44
4.2	Creating distinct data types	44
4.3	CAST functions	45
4.4	Privileges required to work with UDTs	46
4.5	Using CAST functions	48
4.6	Operations allowed on distinct types	49
4.6.1	Extending operations allowed in UDTs	49
4.7	Usage considerations	53
4.7.1	UDTs in host language programs	53
4.7.2	Using the LIKE comparison with UDTs	54
4.7.3	UDTs and utilities	54
4.7.4	Implementing UDTs in an existing environment	55
4.7.5	Miscellaneous considerations	56
4.8	UDTs in the catalog	56
Chapter 5. User-defined functions (UDF)		57
5.1	Terminology overview	58
5.2	Definition of a UDF	59
5.3	The need for user-defined functions	59
5.4	Implementation and maintenance of UDFs	60
5.4.1	Scalar functions	60
5.4.2	Column functions	61
5.4.3	Table functions	62
5.5	UDF design considerations	64
5.5.1	Maximizing UDF efficiency	64
5.5.2	Consider sourced functions	65
Chapter 6. Built-in functions		71
6.1	What is a built-in function?	72
6.2	Why use a built-in function	72
6.3	Built-in function characteristics	72
6.4	List of built-in functions before Version 6	72
6.5	New built-in functions in Version 6	73
6.6	New functions in Version 7	75
6.7	Built-in function restrictions	76
Part 2. Enhancements that allow a more flexible design		77
Chapter 7. Temporary tables		79
7.1	Summary of differences between types of tables	80
7.2	Created temporary tables	80
7.2.1	What is a created temporary table?	81
7.2.2	Why use created temporary tables	81
7.2.3	Created temporary tables characteristics	82
7.2.4	Created temporary tables pitfalls	86
7.2.5	Created temporary tables restrictions	86
7.3	Declared temporary tables	88

7.3.1	What is a declared temporary table?	88
7.3.2	Why use declared temporary tables	88
7.3.3	Declared temporary tables characteristics	88
7.3.4	Creating a temporary database and table space	89
7.3.5	Creating a declared temporary table.	90
7.3.6	Using declared temporary tables in a program	92
7.3.7	Creating declared temporary tables for scrollable cursors	93
7.3.8	Remote declared temporary tables	93
7.3.9	Creating indexes	94
7.3.10	Usage considerations	94
7.3.11	Converting from created temporary tables	95
7.3.12	Authorization	95
7.3.13	Declared temporary table restrictions	95
Chapter 8. Savepoints		97
8.1	What is a savepoint?	98
8.2	Why to use savepoints	98
8.3	Savepoint characteristics	99
8.4	Remote connections	101
8.5	Savepoint restrictions	102
Chapter 9. Unique column identification		103
9.1	Identity columns	104
9.1.1	What is an identity column?	104
9.1.2	When to use identity columns	104
9.1.3	Identity column characteristics	104
9.1.4	Creating a table with an identity column	105
9.1.5	How to populate an identity column	106
9.1.6	How to retrieve an identity column value	109
9.1.7	Identity columns in a data sharing environment	110
9.1.8	Trying to overcome the identity column deficiencies	110
9.1.9	Application design considerations	111
9.1.10	Identity column restrictions	112
9.2	ROWID and direct row access	112
9.2.1	What is a ROWID?	113
9.2.2	ROWID implementation and maintenance	113
9.2.3	How ROWIDs are generated	115
9.2.4	Casting to a ROWID data type	117
9.2.5	ROWIDs and partitioning keys	118
9.2.6	ROWID and direct row access	119
9.2.7	ROWID and direct row access restrictions	121
9.3	Identity column and ROWID usage and comparison	122
Part 3. More powerful SQL		123
Chapter 10. SQL CASE expressions		125
10.1	What is an SQL CASE expression?	126
10.2	Why use an SQL CASE expression	127
10.3	Alternative solutions	131
10.4	Other uses of CASE expressions	131
10.5	SQL CASE expression restrictions	133
Chapter 11. Union everywhere		135
11.1	What is a union everywhere?	136

11.2	Why union everywhere	136
11.3	Unions in nested table expressions	136
11.4	Unions in subqueries.	137
11.4.1	Unions in basic predicates	137
11.4.2	Unions in quantified predicates	137
11.4.3	Unions in EXISTS predicates	138
11.4.4	Unions in IN predicates.	139
11.4.5	Unions in selects of INSERT statements	139
11.4.6	Unions in UPDATE	140
11.5	Unions in views	140
11.6	Explain and unions	142
11.7	Technical design and new frontiers.	143
Chapter 12. Scrollable cursors.		149
12.1	What is a scrollable cursor?	150
12.2	Why use a scrollable cursor	150
12.3	Scrollable cursors characteristics	151
12.3.1	Types of cursors	151
12.3.2	Scrollable cursors in depth	152
12.4	How to choose the right type of cursor	153
12.5	Using a scrollable cursor.	154
12.5.1	Declaring a scrollable cursor.	155
12.5.2	Opening a scrollable cursor	155
12.5.3	Fetching rows	157
12.5.4	Moving the cursor	164
12.5.5	Using functions in a scrollable cursor	168
12.6	Update and delete holes	170
12.6.1	Delete hole	171
12.6.2	Update hole.	172
12.7	Maintaining updates	174
12.8	Locking and scrollable cursors	177
12.9	Stored procedures and scrollable cursors.	178
12.10	Scrollable cursors recommendations	179
Chapter 13. More SQL enhancements		181
13.1	The ON clause extensions	182
13.1.1	Classifying predicates	182
13.1.2	During join predicates	182
13.2	Row expressions.	185
13.2.1	What is a row expression?	185
13.2.2	Types of row expressions	185
13.2.3	Row expression restrictions	188
13.3	ORDER BY	188
13.3.1	ORDER BY columns no longer have to be in select list (V5)	188
13.3.2	ORDER BY expression in SELECT (V7)	189
13.3.3	ORDER BY sort avoidance (V7)	190
13.4	INSERT	191
13.4.1	Using the DEFAULT keyword in VALUES clause of an INSERT	191
13.4.2	Inserting using expressions	192
13.4.3	Inserting with self-referencing SELECT	192
13.4.4	Inserting with UNION or UNION ALL	193
13.5	Subselect UPDATE/DELETE self-referencing	193
13.6	Scalar subquery in the SET clause of an UPDATE	195

13.6.1	Conditions for usage	196
13.6.2	Self-referencing considerations.	197
13.7	FETCH FIRST n ROWS ONLY.	197
13.8	Limiting rows for SELECT INTO	198
13.9	Host variables	199
13.9.1	VALUES INTO statement	199
13.9.2	Host variables must be preceded by a colon	200
13.10	The IN predicate supports any expression	201
13.11	Partitioning key update	202
Part 4.	Utilities versus applications	203
Chapter 14.	Utilities versus application programs.	205
14.1	Online LOAD RESUME.	206
14.1.1	What is online LOAD RESUME?	206
14.1.2	Why use Online LOAD RESUME	206
14.1.3	Online LOAD RESUME versus classic LOAD	207
14.1.4	Online LOAD RESUME versus INSERT programs.	208
14.1.5	Online LOAD RESUME pitfalls	209
14.1.6	Online LOAD RESUME restrictions	209
14.2	REORG DISCARD	210
14.2.1	What is REORG DISCARD?.	210
14.2.2	When to use a REORG DISCARD	210
14.2.3	Implementation and maintenance.	210
14.2.4	REORG DISCARD restrictions	211
14.3	REORG UNLOAD EXTERNAL and UNLOAD	212
14.3.1	What are REORG UNLOAD EXTERNAL and UNLOAD?	212
14.3.2	REORG UNLOAD EXTERNAL.	212
14.3.3	UNLOAD.	213
14.3.4	UNLOAD implementation	213
14.3.5	UNLOAD restrictions.	214
14.3.6	UNLOAD highlights.	214
14.3.7	UNLOAD pitfalls	215
14.3.8	Comparing DSNTIAUL, REORG UNLOAD EXTERNAL and UNLOAD	215
14.4	Using SQL statements in the utility input stream	217
14.4.1	EXEC SQL utility control statement	217
14.4.2	Possible usage of the EXEC SQL utility statement.	218
Part 5.	Appendixes	221
Appendix A.	DDL of the DB2 objects used in the examples	223
E/R-diagram of the tables used by the examples		224
JCL for the SC246300 schema definition.		224
Creation of a database, table spaces, UDTs and UDFs		225
Creation of tables used in the examples		228
Creation of sample triggers		236
Populated tables used in the examples		238
DDL to clean up the environment.		240
Appendix B.	Sample programs	243
Returning SQLSTATE from a stored procedure to a trigger		244
Passing a transition table from a trigger to a SP		246
Appendix C.	Additional material	251

Locating the Web material	251
Using the Web material	251
How to use the Web material	251
Related publications	253
IBM Redbooks	253
Other resources	253
Referenced Web sites	254
How to get IBM Redbooks	254
IBM Redbooks collections.	255
Special notices	257
Abbreviations and acronyms	259
Index	261

Figures

3-1	Allowable trigger parameter combinations.	22
3-2	Allowed SQL statement matrix.	23
3-3	Trigger cascading	29
3-4	SQL processing order and triggers	36
4-1	Comparison operators allowed on UDTs created WITH COMPARISONS	49
8-1	Travel reservation savepoint sample itinerary	98
9-1	Identity column value assignment in a data sharing environment	110
12-1	Fetch syntax changes to support scrollable cursors	158
12-2	How to scroll within the result table	167
12-3	SQLCODEs and cursor position	168
12-4	How DB2 validates a positioned UPDATE.	176
12-5	How DB2 validates a positioned DELETE	177
12-6	Stored procedures and scrollable cursors	179
13-1	Improved sort avoidance for ORDER BY clause	190
14-1	Online LOAD RESUME	208
14-2	Generated LOAD statements.	211
14-3	Sample UNLOAD utility statement.	213
A-1	Relations of tables used in the examples	224

Tables

4-1	Catalog changes to support UDTs	56
5-1	Allowable combinations of function types	59
7-1	Distinctions between DB2 base tables and temporary tables	80
10-1	Functions equivalent to CASE expressions	131
11-1	PLAN_TABLE changes for UNION everywhere	142
12-1	Cursor type comparison	154
12-2	Sensitivity of FETCH to changes made to the base table	161
13-1	How the FETCH FIRST clause and the OPTIMIZE FOR clause interact	198
14-1	Comparing different means to unload data	216

Examples

2-1	Overriding the implicit search path	9
2-2	Schema authorization	10
3-1	Trigger to maintain summary data	14
3-2	Trigger to maintain summary data	16
3-3	Trigger to initiate an external action	16
3-4	BEFORE trigger	17
3-5	Multiple trigger actions	19
3-6	Transition variables	20
3-7	Transition table	21
3-8	Single trigger action	21
3-9	Invoking a UDF within a trigger	24
3-10	Raising error conditions	24
3-11	Signaling SQLSTATE	25
3-12	Information returned after a SIGNAL SQLSTATE	25
3-13	Sample information returned when trigger receives an SQLCODE	26
3-14	Passing SQLSTATE back to a trigger	27
3-15	Cascading error message	29
3-16	Using table locators	30
3-17	Sample COBOL program using a SP and table locator	31
3-18	Rebinding a trigger package	33
3-19	Information returned when trigger package is invalid	34
3-20	Comment on trigger	34
3-21	Check constraint is better than a trigger	39
3-22	Trigger is better than a check constraint	40
3-23	Alternative trigger	40
3-24	Identify all triggers for a table	41
3-25	Identify all triggers for a database	41
4-1	Sample DDL to create UDTs	44
4-2	Automatically generated CAST functions	45
4-3	Create table using several UDTs	46
4-4	GRANT USAGE/ EXECUTE ON DISTINCT TYPE	47
4-5	DROP and COMMENT ON for UDTs	47
4-6	Strong typing and invalid comparisons	48
4-7	Two casting methods	48
4-8	Using sourced functions	50
4-9	Defining sourced column sourced functions on UDTs	50
4-10	Strong typing and invalid comparisons	51
4-11	Comparing pesetas and euros	52
4-12	Another way to compare pesetas and euros	52
4-13	Automatic conversion of euros	53
4-14	Using LIKE on a UDT	54
4-15	Loading a table with a UDT	54
5-1	Example of an SQL scalar function	60
5-2	User-defined external scalar function	61
5-3	Sourced column function	61
5-4	User-defined table function	62
5-5	External UDF to convert from SMALLINT to VARCHAR	66
5-6	Creating a sourced UDF	66
5-7	Using CAST instead of a UDF	67
5-8	Built-in function instead of a UDF	67

5-9	CHARNSI source code	67
7-1	Created temporary table DDL	82
7-2	Created temporary table in SYSIBM.SYSTABLES	82
7-3	Use of LIKE clause with created temporary tables	83
7-4	View on a created temporary table	83
7-5	Dropping a created temporary table	84
7-6	Using a created temporary table in a program.	84
7-7	Sample DDL for a declared temporary table	88
7-8	Create a database and table spaces for declared temporary tables	89
7-9	Explicitly specify columns of declared temporary table	91
7-10	Implicit define declared temporary table and identity column	91
7-11	Define declared temporary table from a view	91
7-12	Dropping a declared temporary table.	91
7-13	Declared temporary tables in a program	92
7-14	Three-part name of declared temporary table	93
8-1	Setting up a savepoint	100
9-1	IDENTITY column for member number	105
9-2	Copying identity column attributes with the LIKE clause	106
9-3	Insert with select from another table	108
9-4	Retrieving an identity column value	109
9-5	ROWID column	113
9-6	SELECTing based on ROWIDs.	114
9-7	Copying data to a table with GENERATED ALWAYS ROWID via subselect	115
9-8	Copying data to a table with GENERATED BY DEFAULT ROWID via subselect.	116
9-9	DCLGEN output for a ROWID column.	117
9-10	Coding a ROWID host variable in Cobol	117
9-11	Why not to use a ROWID column as a partitioning key.	118
9-12	ROWID direct row access	119
9-13	Inappropriate coding for direct row access.	121
10-1	SELECT with CASE expression and simple WHEN clause.	126
10-2	Update with CASE expression and searched WHEN clause.	126
10-3	Three updates vs. one update with a CASE expression	127
10-4	One update with the CASE expression and only one pass of the data	128
10-5	Three updates vs. one update with a CASE expression	128
10-6	Same update implemented with CASE expression and only one pass of the data.	128
10-7	Same update with simplified logic	129
10-8	Avoiding division by zero	129
10-9	Avoid division by zero, second example	129
10-10	Replacing several UNION ALL clauses with one CASE expression	130
10-11	Raise an error in CASE statement.	131
10-12	Pivoting tables	132
10-13	Use CASE expression for grouping	132
11-1	Unions in nested table expressions	136
11-2	Using UNION in basic predicates	137
11-3	Using UNION with quantified predicates	137
11-4	Using UNION in the EXISTS predicate	138
11-5	Using UNION in an IN predicate	139
11-6	Using UNION in an INSERT statement	139
11-7	Using UNION in an UPDATE statement	140
11-8	Create view with UNION ALL.	141
11-9	Use view containing UNION ALL.	142
11-10	PLAN_TABLE output.	142
11-11	DDL to create split tables.	144
11-12	DDL to create UNION in view	146
11-13	Sample SELECT from view to mask the underlying tables	147

11-14	Views to use for UPDATE and DELETE	147
11-15	WITH CHECK OPTION preventing INSERT	147
12-1	Sample DECLARE for scrollable cursors.	155
12-2	Opening a scrollable cursor	156
12-3	Example of a FETCH SENSITIVE request which creates an update hole	163
12-4	Scrolling through the last five rows of a table	165
12-5	Several FETCH SENSITIVE statements	166
12-6	Using functions in a scrollable cursor	168
12-7	Using functions in an insensitive scrollable cursor.	169
12-8	Aggregate function in a SENSITIVE cursor	169
12-9	Aggregate function in an INSENSITIVE cursor	170
12-10	Scalar functions in a cursor	170
12-11	Expression in a sensitive scrollable cursor	170
12-12	Delete holes.	172
12-13	Update holes	173
13-1	Sample tables and rows.	182
13-2	Inner join and ON clause with AND	183
13-3	LEFT OUTER JOIN with ANDed predicate on WORKDEPT field in the ON clause	183
13-4	LEFT OUTER JOIN with ON clause and WHERE clause	184
13-5	Inner join and ON clause with OR in the WORKDEPT column	184
13-6	Row expressions with equal operation	185
13-7	Row expressions with = ANY operation.	186
13-8	Row expression with <> ALL operator.	186
13-9	IN row expression	187
13-10	NOT IN row expression	187
13-11	Row expression restrictions.	188
13-12	ORDER BY column not in the select list	188
13-13	ORDER BY expression in SELECT	189
13-14	Data showing improved sort avoidance for the ORDER BY clause.	190
13-15	Inserting with the DEFAULT keyword	191
13-16	Inserting using an expression	192
13-17	Inserting with a self-referencing SELECT	192
13-18	UPDATE with a self referencing non-correlated subquery	193
13-19	UPDATE with a self referencing non-correlated subquery	193
13-20	DELETE with self referencing non-correlated subquery	194
13-21	DELETE with self referencing non-correlated subquery	194
13-22	Invalid positioned update	194
13-23	Non-correlated subquery in the SET clause of an UPDATE	195
13-24	Correlated subquery in the SET clause of an UPDATE.	195
13-25	Correlated subquery in the SET clause of an UPDATE with a column function	195
13-26	Correlated subquery in the SET clause of an UPDATE using the same table	196
13-27	Row expression in the SET clause of an UPDATE	196
13-28	FETCH FIRST n ROWS ONLY	197
13-29	Limiting rows for SELECT INTO	199
13-30	Use of VALUES INTO	199
13-31	Some uses of the SET assignment statement.	199
13-32	IN predicate supports any expression	201
14-1	REORG DISCARD utility statement.	211
14-2	REORG UNLOAD EXTERNAL	212
14-3	List of dynamic SQL statements	217
14-4	Create a new table with the same layout as SYSIBM.SYSTABLES	217
A-1	Schema creation	224
A-2	DDL for the stogroup, database, table space creation.	226
A-3	DDL for UDT and UDF creation.	227
A-4	DDL for the table creation	228

A-5	DDL for triggers	236
A-6	Populated tables used in the examples	238
A-7	DDL to clean up the examples environment	240
B-1	Returning SQLSTATE to a trigger from a SP	244
B-2	Passing a transition table from a trigger to a SP	246

Preface

This IBM Redbook describes the major enhancements that affect application programming when accessing DB2 data on a S/390 or z/Series platform, including the object-oriented extensions such as triggers, user-defined functions and user-defined distinct types, the usage of temporary tables, savepoints and the numerous extensions to the SQL language to help you build powerful, reliable and scalable applications, whether it be in a traditional environment or on an e-business platform.

IBM DATABASE 2 Universal Database Server for z/OS and OS/390 Version 7 (or just DB2 V7 throughout this book) is currently at its eleventh release. Over the last couple of versions a large number of enhancements were added to the product. Many of these enhancements affect application programming and the way you access your DB2 data.

This book will help you to understand how these programming enhancements work and provide examples of how to use them. It provides considerations and recommendations for implementing these enhancements and for evaluating their applicability in your DB2 environments.

The team that wrote this redbook

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

Bart Steegmans is a DB2 Product Support Specialist in IBM Belgium who has recently joined the ITSO. He has over 12 years of experience in DB2. Before joining IBM in 1997, Bart worked as a DB2 system administrator at a banking and insurance group. His areas of expertise include DB2 performance, database administration, and backup and recovery.

Rafael Garcia has been in the IT business for 19 years and has held various positions. He was a COBOL and CICS developer, an application development manager and a DB2 applications DBA for one of the top 10 banks in the US. For the last five years he has been a Field DB2 Technical Specialist working for the IBM Silicon Valley Laboratory supporting DB2 for OS/390 customers across various industries, including migrations to data sharing. He has an associate's degree in Arts and an associate's degree in Science in Business Data Processing from Miami-Dade Community College.

Luis Garmendia is a DB2 Instructor for IBM Learning Services in Spain and has more than three years of experience teaching DB2 application and system performance, database and system administration, recovery, and data sharing. He has a doctorate in Mathematics (fuzzy logic and inference) and he also teaches Computer Sciences in the Technical University of Madrid.

Anne Lesell is a DB2 and Database Design Instructor and Curriculum Manager for IBM in Finland. Anne has been working with DB2 since 1987. Before joining IBM in 1998, she worked as a Database Administrator in a large Finnish bank where she specialized in database design and DB2 application tuning.

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

Emma Jacobs
Yvonne Lyon
Gabrielle Velez
International Technical Support Organization, San Jose Center

Sherry Guo
William Kyu
Roger Miller
San Phoenix
Kalpana Shyam
Yunfei Xie
Koko Yamaguchi
IBM Silicon Valley Laboratory

Robert Begg
IBM Toronto, Canada

Michael Parbs
Paul Tainsh
IBM Australia

Rich Conway
IBM International Technical Support Organization, Poughkeepsie Center



Peter Backlund
Martin Hubel
Gabrielle Wiorkowski
IBM Database Gold Consultants

Special notice

This publication is intended to help managers and professionals understand and evaluate some of the application related enhancements that were introduced over the last couple of years into the IBM DATABASE 2 Universal Database Server for z/OS and OS/390 products. The information in this publication is not intended as the specification of any programming interfaces that are provided by the IBM DATABASE 2 Universal Database Server for z/OS and OS/390. See the PUBLICATIONS section of the IBM Programming Announcement for the IBM DATABASE 2 Universal Database Server for z/OS and OS/390 for more information about what publications are considered to be product documentation.

IBM trademarks

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

e (logo)® 	Redbooks™
IBM®	Redbooks Logo 
AIX®	Notes®
CICS®	OS/390®
DB2®	Parallel Sysplex®
DB2 Connect™	Perform™
DB2 Universal Database™	QMF™
DFS™	RACF®
DRDA®	RETAIN®
ECKD™	S/390®
Enterprise Storage Server™	SP™
IMS™	System/390®
MORE™	TME®
MVS™	WebSphere®
MVS/ESA™	z/OS™

Comments welcome

Your comments are important to us!

We want our IBM 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 redbook form found at:
ibm.com/redbooks
- ▶ Send your comments in an Internet note to:
redbook@us.ibm.com
- ▶ Mail your comments to the address on page ii.



Introduction

New DB2 versions and releases have always been a balanced mixture between system related enhancements and improvements that benefit database administrators and the application programming community.

When we first started to lay out the content of this book we wanted to include all the application programming related enhancements since DB2 Version 4, where a lot of major programming related enhancements like OUTER JOIN and nested table expressions to name only a few, became available. Even though these are very important enhancements and very relevant to application programming and design, we soon had to give up the idea and decided to concentrate mainly on the enhancements since DB2 Version 6, and even that turned out to be over ambitious. Therefore, if a certain topic is not included in this redbook, it is most likely because it has already been covered in some of the other available redbooks. Good references are:

- ▶ *DB2 UDB Server for OS/390 and z/OS Version 7 Presentation Guide*, SG24-6121
- ▶ *DB2 UDB Server for OS/390 Version 6 Technical Update*, SG24-6108
- ▶ *DB2 Server for OS/390 Version 5 Recent Enhancements - Reference Guide*, SG24-5421

There is also another category of enhancements, that, although application related, did not make it into this redbook. Stored procedures and Java support are good examples. They have a big impact on the way you develop applications and are, or therefore will be, treated in separate redbooks like:

- ▶ *DB2 for OS/390 and z/OS Powering the World's e-business Solutions*, SG24-6257
- ▶ *DB2 Java Stored Procedures Learning by Example*, SG24-5945
- ▶ *Cross-Platform DB2 Stored Procedures: Building and Debugging*, SG24-5485

For a complete list of recent DB2 for OS/390 related redbooks, see “Related publications” on page 253, or visit the Redbooks Web site at: ibm.com/redbooks.

This redbook is based on DB2 for z/OS Version 7 (PUT0106)¹ and all the examples in this book are written using a Version 7 system. However, since a large customer basis is still running DB2 V6, we mention when a feature was introduced in V7. If the version is not specifically mentioned, it is part of Version 6. (Some features that were introduced in Version 5 are also included in the book.)

¹ PUT0106 indicates the maintenance level.

DB2 Version 6 was a very large release and had a vast number of enhancements that have an impact on application programming and application design. With V6, DB2 for z/OS stepped into the world of **object-oriented databases**, introducing features like triggers, user-defined distinct types and user-defined functions. With these feature you can turn your database management system from a passive database into an active database.

Triggers provide automatic execution of a set of SQL statements (verifying input, executing additional SQL statements, invoking external programs written in any popular language) whenever a specified event occurs. This opens the door for new possibilities in application and database design.

User-defined distinct types can help you enforce strong typing through the database management system. The distinct type reflects the use of the data that is required and/or allowed. Strong typing is especially valuable for ad-hoc access to data where users don't always understand the full semantics of the data.

The number of **built-in functions** increased considerably in last versions of DB2. There are now over 90 different functions that perform a wide range of string, date, time, and timestamp manipulations, data type conversions, and arithmetic calculations.

In some cases even this large number of built-in functions does not fit all needs. Therefore, DB2 allows you to write your own **user-defined functions** that can call an external program. This extends the functionality of SQL to whatever you can code in an application program; essentially, there are no limits. User-defined functions also act as the methods for user-defined data types by providing consistent behavior and encapsulating the types.

Another set of enhancements is more geared toward giving you **more flexibility when designing databases and applications**.

When you need a table only for the life of an application process, you don't have to create a permanent table to store this information but you can use a temporary table instead. There are two kinds of temporary tables: **created temporary tables**, also know as **global temporary tables** and **declared temporary tables**. Their implementation is different from normal tables. They have reduced or no logging and also virtually no locking. The latter is not required since the data is not shared between applications. The data that is stored in the temporary table is only visible to the application process that created it.

Another major enhancement that can make you change the way you have been designing applications in the past is the introduction of **savepoints**. Savepoints enable you to code contingency or what-if logic and can be useful for programs with sophisticated error recovery, or to undo updates made by a stored procedure or subroutine when an error is detected, and to ensure that only the work done in a stored procedure or subroutine is rolled back.

Another area which has always caused a lot of debate is how and when to assign a unique identification to a relational object. Often a natural key is available and should be used to identify the relation. However, this is not always the case and this is where a lot of discussions start. Should we use an ever-ascending or descending key? Should it be random instead? Should we put that ever-increasing number in a table and if so, how do we access it and when do we update the number? These are all questions that keep DBA's from losing their jobs.

Recently DB2 has introduced two new concepts that can guarantee unique column values without having to create an index. In addition, you can eliminate the application coding that was implemented to assign unique column values for those columns.

The first concept is the introduction of **identity columns**. Identity columns offer us a new possibility to guarantee uniqueness of a column and enables us to automatically generate the value inside the database management system.

The second concept is the usage of a new data type called **ROWID**. There three aspects to ROWIDs. Its first and primary function is to access the LOB data from LOB columns (like audio and video) stored in the auxiliary table. (This usage of ROWID is beyond the scope of this redbook). The second function is the fact that a ROWID can be a unique random number generated by the DBMS. This looks like a very appealing option to solve many design issues. The third aspect of using ROWID columns is that they can be used for a special type of access path to the data, called direct row access.

Another major enhancement is **scrollable cursors**. The ability to be able to scroll backwards as well as forwards has been a requirement of many screen-based applications. DB2 V7 introduces facilities not only to allow scrolling forwards and backwards, but also the ability to jump around and directly retrieve a row that is located at any position within the cursor result table. DB2 also can, if desired, maintain the relationship between the rows in the result set and the data in the base table. That is, the scrollable cursor function allows the changes made outside the opened cursor, to be reflected in the result set returned by the cursor.

DB2 utilities are normally not directly the terrain of the application programmer. However, new **DB2 utilities** have been introduced and existing utilities have been enhanced in such a way that they **can take over some of the work that was traditionally done in application programs**. Some ideas where these enhancements can be used and how they compare to implementing the same processes using application code will be provided, such as using online LOAD RESUME versus coding your own program to add rows to a table, using REORG DISCARD versus deleting rows via a program followed by a REORG to reclaim unused space and restore the cluster ratio, and comparing REORG UNLOAD EXTERNAL, the new UNLOAD utility to a home grown program or the DSNTIAUL sample program.

And last but not least, we will discuss and provide examples for a large list of **enhancements to the SQL language** that will boost programmer productivity, such as CASE expressions, that allow you to code IF-THEN logic inside an SQL statement, UNION everywhere, that enables you to code a full select, wherever you were allowed to code a subselect before. This also included the long-awaited union-in-view feature and will finally allow you to code a UNION clause inside a CREATE VIEW statement. With this enhancement, DB2 is delivering one of the oldest outstanding requirements which should make a lot of people want to migrate to Version 7 sooner rather than later.

With such a vast range of programming capabilities, that not only provide rich functionality but also good performance and great scalability, DB2 for z/OS Version 7 is certainly capable of competing with any other DBMS in the marketplace.

We hope you enjoy reading this Redbook as much as we did writing it.

Object-oriented enhancements

In this part we describe and discuss various object-oriented enhancements that can transform DB2 from a passive database manager into an active one by allowing you to move application logic into the database. These enhancements allow you to move application logic that may reside in various places, platforms, and environments into one place, the database itself.

These are the enhancements we discuss:

- ▶ Schemas
- ▶ Triggers
- ▶ User-defined distinct types
- ▶ User defined functions
- ▶ Built-in functions



Schemas

The concept of schemas has been around for quite a some time.

In this section, we discuss the schema concept, since it is now starting to be widely used in a DB2 for z/OS environment and is required for some features implemented in DB2 V6, like triggers, user-defined functions, user-defined distinct types and stored procedures.

2.1 What is a schema?

When tables, views, indexes, and aliases are created, they are given a *qualified name*. When the qualified name is a two-part name; the second part is the *name* of the object and the first part (an authorization id) is a *qualifier* that distinguishes the object from other objects that have the same name.

In Version 6, to be consistent with the ANSI/ISO SQL92 standard, the concept of qualified names is extended to refer to the qualifier as a *schema name*. The qualifier of the new object types introduced in Version 6 (user-defined distinct types, user-defined functions and triggers) as well as stored procedures, is a schema name.

A schema name has a maximum length of eight bytes. All objects qualified by the same schema name can be thought of as a group of related objects. The schema name SYSIBM is used for built-in data types and functions, the schema name SYSPROC is used for stored procedures migrated from Version 5 and can be used for procedures created in V6. SYSFUN is the schema name used for additional functions shipped with other members of the DB2 family. Although DB2 for z/OS does not use the SYSFUN schema, it can be useful to have SYSFUN in the CURRENT PATH special register when doing distributed processing and another DB2 family member is the server.

Restriction: CREATE statements cannot specify a schema name that begins with 'SYS' (unless it is 'SYSADM' or 'SYSPROC' for stored procedures).

2.2 Schema characteristics

This section covers the following topics:

- ▶ Authorizations on schemas
- ▶ Schema path and special registers
- ▶ How a schema name is determined

2.2.1 Authorizations on schemas

The new GRANT (schema privileges) statement is used to grant privileges on schemas. The schema privileges are:

▶ **CREATEIN**

The privilege to create user-defined distinct types, user-defined functions, triggers, and stored procedures in the designated schemas.

▶ **ALTERIN**

The privilege to alter user-defined functions and stored procedures, or specify a comment for user-defined distinct types, user-defined functions, triggers, and stored procedures in the designated schemas.

▶ **DROPIN**

The privilege to drop user-defined distinct types, user-defined functions, triggers, and stored procedures in the designated schemas.

The specified schemas do not need to exist when the grant is executed.

Note: An authorization ID 'JOHN' has the implicit CREATEIN, ALTERIN, and DROPIN privileges for the schema named 'JOHN'.

2.2.2 Schema path and special register

The new PATH bind option is applicable to BIND PLAN, BIND PACKAGE, REBIND PLAN and REBIND PACKAGE. The ordered list of schemas specified in the PATH is used to resolve unqualified references to user-defined distinct types (UDTs) and user-defined functions (UDFs) in **static** SQL statements. It is also used to resolve unqualified stored procedure names when the SQL CALL statement specifies a literal for the procedure name.

The default for the PATH bind option is "SYSIBM", "SYSFUN", "SYSPROC", QUALIFIER (plan or package). If the PATH option is not specified on a REBIND, the previous value is retained.

The new SET CURRENT PATH statement is used to change the list of schemas held in the CURRENT PATH special register. The CURRENT PATH special register is used to resolve unqualified references to user-defined distinct types and user-defined functions in *dynamic* SQL statements. It is also used to resolve unqualified stored procedure names when the SQL CALL statement specifies a host variable for the procedure name.

The CURRENT PATH special register is initialized at run time as follows:

- ▶ If the PATH bind option was specified at bind time, the CURRENT PATH special register is initialized to the path specified at bind time.
- ▶ If the PATH bind option was not specified at bind time, then the default path is used. The CURRENT PATH special register is initialized to "SYSIBM", "SYSFUN", "SYSPROC", CURRENT SQLID.
- ▶ The initialization of the CURRENT PATH special register in a nested stored procedure or UDF also depends upon whether the SET CURRENT PATH statement has been issued previously.

The schemas SYSIBM, SYSFUN and SYSPROC do not need to be included in the PATH bind option or the CURRENT PATH special register. If one is not included in the path, it is implicitly assumed as the first schema. If more than one is not included in the path, SYSIBM is put first in the path followed by SYSFUN and then SYSPROC. In Example 2-1, we see what happens when the CURRENT PATH special register is set to some values.

Example 2-1 Overriding the implicit search path

```
SET CURRENT PATH = "SCHEMA1", "SCHEMA2" ;  
sets the path as:  
SYSIBM, SYSFUN, SYSPROC, SCHEMA1, SCHEMA2
```

```
SET CURRENT PATH = "SCHEMA1", "SCHEMA2", "SYSIBM" ;  
sets the path as:  
SYSFUN, SYSPROC, SCHEMA1, SCHEMA2, SYSIBM
```

```
SET CURRENT PATH = "SCHEMA1", "SYSFUN", "SCHEMA2" ;  
sets the path as:  
SYSIBM, SYSPROC, SCHEMA1, SYSFUN, SCHEMA2
```

2.2.3 How is a schema name determined?

The schema name can be specified explicitly when the object is referenced in a CREATE, ALTER, DROP or COMMENT ON statement.

If the object is unqualified and the statement is embedded in a program, the schema name of the object is the authorization ID in the QUALIFIER bind option when the plan or package was created or last rebound. If QUALIFIER was not used, the schema name of the object is the owner of the plan or package. If the object is unqualified and the statement is dynamically prepared, the SQL authorization ID contained in the CURRENT SQLID special register is the schema name of the object.

2.3 The schema processor

A lot of installations create the objects through an SQL processor program like SPUFI or DSNTEP2.

Another way to create objects is by means of a *schema processor* (program DSNHSP) and the use of the CREATE SCHEMA statement. This statement cannot be used by 'normal' SQL processor programs like SPUFI, but is only understood by the schema processor. The schema processor allows you to create a set of related objects that belong to a single schema (corresponding with an authorization id) in one CREATE SCHEMA operation.

The ability to process schema definitions is provided for conformance to ISO/ANSI standards. The result of processing a schema definition is identical to the result of executing the SQL statements without a schema definition.

Using a schema processor has some advantages over using other means of running DDL statements. Outside of the schema processor, the order of statements is important. They must be arranged so that all referenced objects have been previously created. For example, in order to create an index on a table, the table must first be created. This restriction is relaxed when the statements are processed by the schema processor, as long as the object table is created within the same CREATE SCHEMA. The requirement that all referenced objects have been previously created is not checked by the schema processor until all of the statements have been processed. So, with the schema processor, it is sufficient that the create table statement for the table is present anywhere in the input file.

Tip: The order in which statements appear is not important when using the schema processor.

The schema processor sets the current SQLID to the value of the schema authorization ID (which is also the schema name) before executing any of the statements in the schema. In Example 2-2 we create a schema "SC246300", the schema authorization is the same as the schema name. All the objects created within this schema will have a schema name and owner of "SC246300". As you can see the statements after the CREATE SCHEMA can be in any order.

Example 2-2 Schema authorization

```
CREATE SCHEMA AUTHORIZATION SC246300

CREATE INDEX ....
CREATE DISTINCT TYPE ....
CREATE DATABASE ...
CREATE TABLE ....
```



```
CREATE TABLESPACE ....
GRANT ....
CREATE TABLE ....
CREATE INDEX ....
CREATE DISTINCT TYPE ....
CREATE FUNCTION ....
CREATE UNIQUE INDEX ....
CREATE INDEX ....
CREATE TRIGGER ....
GRANT ....
```

Important: Databases and table spaces can and should be defined through the schema processor, however they are not part of a schema (not schema objects) since database names must be unique within the DB2 system and table space names must be unique within a database.

All statements passed to the schema processor are considered one unit of work. If one or more statements fail with a negative SQLCODE, all other statements continue to be processed. However, when the end of the input is reached, all work is rolled back. You can only process one schema per job step execution.

An example of a schema processor job can be found in DDL of the DB2 objects used in the examples, Example A-1 on page 224.

Note: The CREATE SCHEMA statement cannot be embedded in a host program or executed interactively. It can only be executed in a batch job using the schema processor program DSNHSP.



Triggers

Triggers provide automatic execution of a set of SQL statements whenever a specified event occurs. This opens the door for new possibilities in application design. In this section we discuss triggers, and how and when to use them.

3.1 Trigger definition

A trigger is a schema object that defines an action or a set of actions (SQL statements) that are to be executed when a specific SQL data change operation occurs on a specified table.

These SQL statements can validate and edit database changes, read and modify the database, and invoke functions that perform operations both inside and outside the database.

In Example 3-1, we show you a simple trigger to update two summary tables when a new order comes in. The number of orders is increased by one for each region and state when a new order is processed.

Example 3-1 Trigger to maintain summary data

```
CREATE TRIGGER SC246300.TGSUMORD
AFTER INSERT ON SC246300.TBORDER
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
    UPDATE SC246300.TBREGION
        SET NUM_ORDERS = NUM_ORDERS + 1;
    UPDATE SC246300.TBSTATE
        SET NUM_ORDERS = NUM_ORDERS + 1;
END
```

Triggers provide several improvements to the development and execution of DB2 applications and can bring a lot of benefits to your organization:

- ▶ Faster application development. Because triggers are stored in the database, the actions performed by triggers do not have to be coded in each application.
- ▶ Code reusability. A trigger can be defined once and then automatically used by every application program that changes data in the table on which the trigger is defined.
- ▶ Enforce data integrity rules system wide. No matter what application performs inserts, updates, or deletes on a table, you can be certain that the associated business rules that are imbedded in the trigger are carried out. This is especially important with highly distributed applications, ad-hoc queries, and dynamic SQL.
- ▶ Easier maintenance. If a business policy changes, only a change to the corresponding triggers is needed, instead of changes to multiple application programs.
- ▶ Having trigger support in DB2 makes it easier for customers to migrate from other relational database management systems that also have triggers. Triggers can also be used during migration to make some updates to the old system transparent to the applications, when not all programs have yet converted to the new system.

3.2 Why use a trigger

Triggers enable you to encapsulate business logic into the database, and this has many advantages.

Some of the common uses for triggers are:

- ▶ Enforce transitional business rules
 - Validate input data (constraints)
- ▶ Generate or edit column values for inserted and updated rows

- Set default values based on business logic
- ▶ Cross-reference other tables (enhance RI)
- ▶ Maintain summary data
- ▶ Initiate external actions via User-Defined Functions (UDFs) and Stored Procedures (SPs) to:
 - Propagate changes to an external file
 - Send e-mail, fax, or pager notifications
 - Maintain an audit trail
 - Schedule a batch job

Enforcement of transitional business rules: Triggers can enforce data integrity rules with far more capability than is possible with declarative (static) constraints. Triggers are most useful for defining and enforcing transitional business rules. These are rules that involve different states of the data. In Example 3-11 on page 25, we show a trigger constraint that prevents a salary column from being increased by more than twenty percent. To enforce this rule, the value of the salary before and after the increase must be compared. This is something that cannot be done with a check constraint.

Generation and editing of column values: Triggers can automatically generate values for newly inserted rows, that is, you can implement user-defined default values, possibly based on other values in the row or values in other tables. Similarly, column values provided in an insert or update operation can be modified/corrected before the operation occurs.

An example of this can be found in Example 4-13 on page 53. This trigger generates a value for the EUROFEE column based on a conversion formula that calculates an amount in euro based on an amount in pesetas (PESETAFEE).

Cross-reference other tables: You can enhance the existing referential integrity rules that are supported by DB2. For example, RI allows a foreign key to reference a primary key in another table. With the new union everywhere feature you can change your design to split a logical table into multiple physical tables (as an alternative to partitioning). Now it has become impossible to use DB2 RI to implement a foreign key relationship, because you cannot refer multiple tables to check whether it has an existing primary key value. Here triggers can come to the rescue. You can implement a before trigger to check the different tables to make sure the value you are inserting has a parent row.

Note: Make sure you have the PTF for APAR PQ53030 (still open at the time of writing) applied on your system when trying this.

You can also implement so called ‘negative RI’ this way. Instead of checking whether a row exists in a parent table (normal RI), you can use a trigger to make sure a row or column value does not exist in another table. For example, when you want to add a new customer to the customer table (TBCUSTOMER), you might want to check (using a trigger) whether that customer does not already exist in the customer archive database (TBCUSTOMER_ARCH, that contains customers that have not placed any orders the past year). If it does, you restrict the insert of the new customer and make the application copy the existing customer information from the customer archive database.

Maintenance of summary data: Triggers can automatically update summary data in one or more tables when a change occurs to another table. For example, triggers can ensure that every time a new order is added, updates are made to rows in the TBREGION and TBSTATE table to reflect the change in the number of orders per region and state. Example 3-2 shows part of the solution to implement this. You also need to define an UPDATE and DELETE trigger to cover all cases if you want the number of outstanding orders to be correctly reflected at the region and state level.

Example 3-2 Trigger to maintain summary data

```
CREATE TRIGGER SC246300.TGSUMORD
AFTER INSERT ON SC246300.TBORDER
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
    UPDATE SC246300.TBREGION
        SET NUM_ORDERS = NUM_ORDERS + 1;
    UPDATE SC246300.TBSTATE
        SET NUM_ORDERS = NUM_ORDERS + 1;
END
```

Initiate external actions: In Example 3-3, we demonstrate how a user-defined function can be used within a trigger to initiate an external function. Since a user-defined function can be written in any of the popular programming languages, using a user-defined function in a trigger gives access to a vast number of possible actions you can code. A common uses may be to communicate with an e-mail package and send out an e-mail to the employee to let him know that a change was made to his payroll data.

Example 3-3 Trigger to initiate an external action

```
CREATE TRIGGER PAYROLL1
AFTER UPDATE ON PAYROLL
FOR EACH STATEMENT MODE DB2SQL
VALUES (PAYROLL_LOG (USER, 'UPDATE', CURRENT TIME, CURRENT DATE))
```

3.3 Trigger characteristics

These are some basic characteristics of triggers:

- ▶ The **trigger name** is qualified by a schema name and is limited to 8 characters.
- ▶ The **triggering table** is the table on which the trigger is defined.
- ▶ The **triggering operation** is the SQL data change operation (INSERT, UPDATE or DELETE) for which the trigger is activated. Triggers are often referred to as insert, update, or delete triggers. These are the possible triggering operations:

INSERT An insert operation can only be caused by an INSERT statement. Triggers are not activated when data is loaded using utilities that do not use INSERT, such as the LOAD utility (except online LOAD RESUME in V7).

UPDATE An update operation can be caused by an UPDATE statement or as a result of enforcing a referential constraint rule of ON DELETE SET NULL. Please note that an UPDATE statement does not actually have to change data to cause a trigger to be activated.

DELETE A delete operation can be caused by a DELETE statement or as a result of enforcing a referential constraint rule of ON DELETE CASCADE.

Triggers cannot be activated by SELECT statements. The table named as the triggering table cannot be a DB2 catalog table, view, alias, synonym, or a three-part table name.

A triggering operation can be the result of changes that occur due to referential constraint enforcement. For example, given two tables TBDEPARTMENT and TBEMPLOYEE, if deleting from TBDEPARTMENT causes propagated deletes (ON DELETE CASCADE) or updates (ON DELETE SET NULL) to TBEMPLOYEE because of a referential constraint, then delete or update triggers defined on TBEMPLOYEE will be activated. The triggers defined on TBEMPLOYEE run either before or after the referential constraint operation depending on their defined activation time (whether they are a before or after trigger).

When you define an update trigger you can specify that it should only be activated when certain columns of the triggering table are updated.

Of course, if an SQL data change operation affects a table through a view, any triggers defined on the table for that operation are activated.

- ▶ A **triggering event** is the occurrence of the triggering operation on the triggering table which causes the trigger to be activated.

3.3.1 Trigger activation time

The **trigger activation time** specifies when a trigger is activated in relation to the triggering operation. It is specified as:

- ▶ **NO CASCADE BEFORE**

- The trigger is activated before the triggering operation is performed.
- It is used to validate/edit/generate input data.
- It cannot be used to further modify the database.
- It is often referred to as a *before* trigger.

The trigger action is activated before the triggering operation is processed. The trigger action is activated for each row in the set of affected rows, as the rows are accessed, but before the triggering operation is performed on each row and before any table check or referential integrity constraints that the rows may be subject to are processed. The NO CASCADE keyword is required and serves to remind you that a before trigger cannot perform update operations and, therefore, cannot cause trigger cascading.

Before triggers are generally used as an extension to the constraint system. In Example 3-4, an error is passed back to the application when an INSERT is performed containing an invalid CITY (the CITY being inserted is not in the TBCITIES table). Please note that this could have been implemented using a CHECK CONSTRAINT, RI, or using application logic and all have different performance characteristics. More information on this can be found in sections 3.19, “Design considerations” on page 37 and 3.20, “Some alternatives to a trigger” on page 38.

Example 3-4 BEFORE trigger

```
CREATE TRIGGER SC246300.TGBEFOR7
NO CASCADE BEFORE INSERT
ON SC246300.TBCUSTOMER
REFERENCING NEW AS N
FOR EACH ROW
MODE DB2SQL
WHEN (NOT EXISTS (SELECT CITYKEY
                  FROM SC246300.TBCITIES
                  WHERE CITYKEY = N.CITYKEY))
```

```
SIGNAL SQLSTATE 'ERR10' ('NOT A VALID CITY') #
```

When inserted rows satisfy the WHEN condition you receive:

```
DSNT408I SQLCODE = -438, ERROR: APPLICATION RAISED ERROR WITH DIAGNOSTIC TEXT:  
      NOT A VALID CITY  
DSNT418I SQLSTATE = ERR10 SQLSTATE RETURN CODE
```

► **AFTER**

- The trigger is activated after the triggering operation is performed.
- The trigger can be viewed as a segment of application logic to:
 - Perform follow-on update operations
 - Perform external actions
- This type of trigger is often referred to as an *after* trigger.

The trigger action is activated after the triggering operation has been processed and after all table check constraints and referential constraints that the triggering operation may be subject to have been processed.

After triggers can be viewed as segments of application logic that run every time a specific event occurs. After triggers see the database in the state that an application would see it following the execution of the change operation. This means they can be used to perform actions that an application might otherwise have performed such as maintaining summary data or an audit log. Example 3-6 on page 20 shows an after trigger.

3.3.2 How many times is a trigger activated?

Important to the concept of **trigger granularity** is understanding which rows are affected by the triggering operation. The set of affected rows contains all rows that are deleted, inserted or updated by the triggering operation.

A trigger definition specifies the granularity of the trigger activation as follows:

► **FOR EACH ROW**

- The trigger is activated once for each row affected by the triggering operation (as many times as the number of rows in the set of affected rows). This is often referred to as a *row trigger*. A row trigger is only activated if the set of affected rows is not empty.

► **FOR EACH STATEMENT**

- The trigger is activated once for the triggering operation. This is often referred to as a *statement trigger*. The statement trigger is always fired once even if the set of affected rows is empty.

Note: For a cursor-controlled UPDATE or DELETE (with a WHERE CURRENT OF clause), a statement trigger is executed once per row because only one row is affected. For online LOAD RESUME, each row that is loaded is treated as a statement. When you have a statement trigger defined on that table, it will be invoked for every row that is added to the table.

3.3.3 Trigger action condition

The **trigger action condition** (WHEN clause) controls whether or not the set of triggered SQL statements are executed for the row or statement for which the trigger is executing. This acts as an additional filter between the triggering operation and the trigger action. If the WHEN clause is omitted, the triggered SQL statements are always executed.

In a row trigger, the condition is evaluated for each row in the set of affected rows. In a statement trigger, the condition is evaluated only once as the trigger is activated only once.

3.3.4 Trigger action

The **trigger action**, also called the **trigger body**, is the set of triggered SQL statements, performing the real actions of the trigger. If more than one triggered SQL statement is coded, the set of statements must be surrounded by the keywords BEGIN ATOMIC and END. Each statement in the set must be terminated by a semicolon.

The significance of the required ATOMIC keyword (see Example 3-5), is that the set of SQL statements is treated as an atomic unit. That is, either all of the statements are executed or none. If, for example, the second UPDATE statement shown in Example 3-5 fails, all changes made to the database as part of the triggered operation **and** the triggering operation are backed out. For more details see 3.8, "Error handling" on page 26.

Example 3-5 Multiple trigger actions

```
CREATE TRIGGER NEWPROJ
  AFTER INSERT ON PROJECT
  REFERENCING NEW AS N_ROW
  FOR EACH ROW MODE DB2SQL
  BEGIN ATOMIC
    UPDATE DEPARTMENT SET PROJ_COUNT = PROJ_COUNT + 1
      WHERE DEPTNO = N_ROW.RESP_DEPT;
    CALL INFORM_MANAGER (N_ROW.RESP_DEPT, N_ROW.PROJ_NAME);
    UPDATE EMPLOYEE SET PROJ_COUNT = PROJ_COUNT + 1
      WHERE EMPNO = N_ROW.RESP_EMP;
  END
```

Tip: SQL processor programs, such as SPUFI and DSNTEP2, might not correctly parse SQL statements in the trigger action that are ended with semicolons. These processor programs accept multiple SQL statements, each separated with a terminator character, as input. Processor programs that use a semicolon as the SQL statement terminator can truncate a CREATE TRIGGER statement with embedded semicolons and pass only a portion of it to DB2. Therefore, you might need to change the SQL terminator character for these processor programs. For information on changing the terminator character for SPUFI and DSNTEP2, see *DB2 UDB for OS/390 and z/OS Version 7 Application Programming and SQL Guide*, SC26-9933.

The trigger activation time determines which statements are allowed in the trigger body. See Figure 3-1 on page 22 for a list of the allowable combinations.

3.3.5 Transition variables

Transition variables enable row triggers to access row data as it existed both before and after the triggering operation has been performed

When a row trigger is activated it is likely for the trigger to refer to the column values of the row for which it was activated. The REFERENCING clause of the CREATE TRIGGER statement enables a row trigger to refer to the column values for a row, as they were before, and as they are after, the triggering operation has been performed. The REFERENCING clause is specified as:

- ▶ **REFERENCING OLD AS *correlation-name*** Specifies a correlation name that can be used to reference the column values in the original state of the row, that is, before the triggering operation is executed.
- ▶ **REFERENCING NEW AS *correlation-name*** Specifies a correlation name that can be used to reference the column values that were used to update the row, after the triggering operation is executed.

Example 3-6 Transition variables

```
CREATE TRIGGER BUDG_ADJ
  AFTER UPDATE OF SALARY ON SC246300.TBEMPLOYEE
  REFERENCING OLD AS OLD_EMP
              NEW AS NEW_EMP
  FOR EACH ROW MODE DB2SQL
  UPDATE SC246300.TBDEPARTMENT
    SET BUDGET = BUDGET + (NEW_EMP.SALARY - OLD_EMP.SALARY)
    WHERE DEPTNO = NEW_EMP.WORKDEPT
```

The above example shows how transition variables can be used in a row trigger to maintain summary data in another table. Assume that the department table has a column that records the budget for each department. Updates to the SALARY of any employee (for example, from \$50,000 to \$60,000) in the TBEMPLOYEE table are automatically reflected in the budget of the updated employee's department. In this case, NEW_EMP.SALARY has a value of 60000 and OLD_EMP.SALARY a value of 50000, and BUDGET is increased by 10000.

3.3.6 Transition tables

For those statements that affect more than one row (like non-positioned DELETE and UPDATE statements and INSERTs with a sub-select), **transition tables** enable after triggers to access the set of affected rows as they were before, and as they are after, the execution of the triggering operation.

In both row and statement triggers it might be necessary to refer to the whole set of affected rows. For example, triggered SQL statements might need to apply aggregations over the set of affected rows (MAX, MIN, or AVG of some column values). A trigger can refer to the set of affected rows by using transition tables. Transition tables are specified in the REFERENCES clause of the CREATE TRIGGER statement. The columns in transition tables are referred to using the column names of the triggering table. Like transition variables, there are two kinds of transition tables which are specified as:

- ▶ **REFERENCING OLD_TABLE AS *identifier***: Specifies a table identifier that captures the original state of the set of affected rows, that is, their state before the triggering operation is performed.
- ▶ **REFERENCING NEW_TABLE AS *identifier***: Specifies a table identifier that captures the after state of the set of affected rows, that is, their state after the triggering operation has been performed.

In Example 3-7, we show the use of a transition table in a statement trigger. The UDF named LARGE_ORDER_ALERT is invoked for each row in the new transition table that corresponds to an order worth more than \$10,000.

Example 3-7 Transition table

```
SET CURRENT PATH = 'SC246300' #

CREATE TRIGGER SC246300.LARG_ORD
  AFTER INSERT ON SC246300.TBORDER
  REFERENCING NEW_TABLE AS N_TABLE
  FOR EACH STATEMENT MODE DB2SQL
  SELECT LARGE_ORDER_ALERT(CUSTKEY, TOTALPRICE, ORDERDATE)
     FROM N_TABLE WHERE TOTALPRICE > 10000
```

Important: Transition tables are populated by DB2 before any after-row or after-statement trigger is activated. Transition tables are read-only.

In Example 3-8, a trigger is used to maintain the supply of parts in the PARTS table. The trigger action condition specifies that the set of triggered SQL statements should only be executed for rows in which the value of the ON_HAND column is less than ten per cent of the value of the MAX_STOCKED column. When this condition is true, the trigger action is to reorder (MAX_STOCKED - ON_HAND) items of the affected part using a UDF called ISSUE_SHIP_REQUEST.

Example 3-8 Single trigger action

```
CREATE TRIGGER REORDER
  AFTER UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
  REFERENCING NEW AS N_ROW
  FOR EACH ROW MODE DB2SQL
  WHEN (N_ROW.ON_HAND < 0.10 * N_ROW.MAX_STOCKED)
  VALUES (ISSUE_SHIP_REQUEST(N_ROW.PARTNO, N_ROW.MAX_STOCKED - N_ROW.ON_HAND))
```

3.4 Allowable combinations

Some combinations of trigger granularity, trigger activation time, transition variables, and transition tables do not make sense. For example, after-statement triggers cannot reference transition variables because transition variables refer to column values on a per row basis and after-statement triggers are executed once per execution of the triggering operation. (An after-statement trigger can access individual rows in a transition table). Also, it does not make sense to be able to define a before-statement trigger. See Figure 3-1 for a diagram of allowable syntax.

The triggering operation also affects which transition variables and transition tables can be referenced:

- INSERT** An insert trigger can only refer to new transition variables or a new transition table. Before the execution of an insert operation the affected rows do not exist in the database. That is, there is no original state of the rows that would define old values before the insert operation is applied to the database.
- UPDATE** An update trigger can refer to both old and new transition variables and tables.
- DELETE** A delete trigger can only refer to old transition variables or an old transition table. Because the rows will be deleted, there are no new values to reference.

Granularity	Activation Time	Triggering Operation	Transition Variables Allowed	Transition Tables Allowed
ROW	BEFORE	INSERT	NEW	NONE
		UPDATE	OLD, NEW	
		DELETE	OLD	
	AFTER	INSERT	NEW	NEW_TABLE
		UPDATE	OLD, NEW	OLD_TABLE, NEW_TABLE
		DELETE	OLD	OLD_TABLE
STATEMENT	BEFORE	INVALID TRIGGER		
	AFTER	INSERT	NONE	NEW_TABLE
		UPDATE		OLD_TABLE, NEW_TABLE
		DELETE		OLD_TABLE

Figure 3-1 Allowable trigger parameter combinations

3.5 Valid triggered SQL statements

The trigger activation time determines which triggered SQL statements are allowed. When DB2 executes triggered SQL statements, it does not return any query output to the trigger. Host variables and cursor operations are not allowed in the body of a trigger. For example, SELECT INTO and FETCH statements that read data into host variables are not allowed in a trigger body. See Figure 3-2, for a matrix of the allowable SQL statements for each type of trigger.

BEFORE Trigger	AFTER Trigger
SET <i>transition-variable</i>	INSERT
	Searched UPDATE (not a cursor UPDATE)
	Searched DELETE (not a cursor DELETE)
Full select and the VALUES statement (can be used to invoke User Defined Functions)	
CALL <i>procedure-name</i>	
SIGNAL SQLSTATE	

Figure 3-2 Allowed SQL statement matrix

SET and VALUES statements

The SET transition variable statement can be used in before row triggers to modify values of new transition variables:

```
SET NEWROW.MODIFIER = CURRENT USER
```

The VALUES statement can be used in BEFORE and AFTER triggers to invoke UDFs:

```
VALUES (expression, expression...)
```

An example of the usage of the VALUES statement can be found in Example 3-8 on page 21.

Note: SET *transition-variable* and VALUES are only allowed in a trigger body

3.6 Invoking stored procedures and UDFs

The limited control logic and restrictions on SQL allowed in triggers make it necessary for triggers to invoke UDFs or stored procedures for many tasks, for example:

- ▶ Conditional logic and looping
- ▶ Initiation of external actions
- ▶ Access to non-DB2 resources

Only SQL operations are allowed in a trigger body. However, the ability to invoke an external UDF or stored procedure from within a trigger greatly expands the types of trigger actions possible.

If a *before trigger* invokes an external UDF or stored procedure, that function or procedure cannot execute an INSERT, UPDATE, or DELETE statement because *before triggers* cannot update the database. Attempts to do so results in an SQLCODE -817.

Attention: If the SQLCODE is ignored by the stored procedure or the UDF and returns to the invoking trigger, the triggering action is NOT undone. More information on how to deal with error situations in triggers can be found in Section 3.8, “Error handling” on page 26.

UDFs cannot be invoked in a stand-alone manner, that is, they must appear in an SQL statement. A convenient method for invoking a UDF is to use a VALUES statement or a full select as show in Example 3-9.

Example 3-9 Invoking a UDF within a trigger

```
VALUES (UDF1(NEW.COL1), UDF2(NEW.COL2))

SELECT UDF1(COL1), UDF2(COL2)
FROM NEW_TABLE
WHERE COL1 > COL3
```

3.7 Setting error conditions

Triggers can be used for detecting and stopping invalid updates. Two methods are available:

SIGNAL SQLSTATE - a new SQL statement that halts processing and returns the requested SQLSTATE and message to an application

- ▶ Only valid as a triggered SQL statement
- ▶ Can be controlled with a WHEN clause

RAISE_ERROR - a built-in function that provides similar results

- ▶ Can appear where expressions appear
- ▶ Can be controlled with a CASE statement

In Example 3-10, we show how a trigger can be used to enforce constraints on inserts to the TBEMPLOYEE table. Please note that at least one clause of the CASE expression must result in a non-RAISE_ERROR condition. The constraints are:

HIREDATE must be date of insert or a future date

HIREDATE cannot be more than 1 year from date of insert

Example 3-10 Raising error conditions

```
CREATE TRIGGER CHK_HDAT
NO CASCADE BEFORE INSERT ON SC246300.TBEMPLOYEE
REFERENCING NEW AS NEW_EMP
FOR EACH ROW MODE DB2SQL
VALUES ( CASE
        WHEN NEW_EMP.HIREDATE < CURRENT DATE
        THEN RAISE_ERROR('75001', 'HIREDATE HAS PASSED')
        WHEN DAYS(NEW_EMP.HIREDATE) - DAYS(CURRENT DATE) > 365
        THEN RAISE_ERROR ('85002', 'HIREDATE TOO FAR IN FUTURE')
        ELSE 0
        END) #
```

In Example 3-11, we show how a trigger can be used to ensure that salary increases do not exceed 20%.

Example 3-11 Signaling SQLSTATE

```
CREATE TRIGGER CHK_SAL
  NO CASCADE BEFORE UPDATE OF SALARY
  ON SC246300.TBEMPLOYEE
  REFERENCING OLD AS OLD_EMP
              NEW AS NEW_EMP
  FOR EACH ROW MODE DB2SQL
  WHEN (NEW_EMP.SALARY > OLD_EMP.SALARY * 1.20)
  SIGNAL SQLSTATE '75001' ('INVALID SALARY INCREASE - EXCEEDS 20%')
```

The SIGNAL SQLSTATE statement is only valid in a trigger body. However, the RAISE_ERROR built-in function can also be used in SQL statements other than a trigger.

When SIGNAL SQLSTATE and RAISE_ERROR are issued, the execution of the trigger is terminated and all database changes performed as part of the triggering operation and all trigger actions are backed out. The application receives SQLCODE -438 along with the SQLSTATE (in SQLCA field SQLSTATE) and text (in SQLCA field SQLERRMC) that have been set by the trigger. For instance, if you perform an illegal update against the CHK_SAL trigger of Example 3-11, you will receive the following information in the SQLCA as shown in Example 3-12.

Example 3-12 Information returned after a SIGNAL SQLSTATE

```
After calling DSNTIAR

DSNT408I SQLCODE = -438, ERROR: APPLICATION RAISED ERROR WITH DIAGNOSTIC TEXT:
        INVALID SALARY INCREASE - EXCEEDS 20%
DSNT418I SQLSTATE = 75001 SQLSTATE RETURN CODE
DSNT415I SQLERRP = DSNXRTP SQL PROCEDURE DETECTING ERROR
DSNT416I SQLERRD = 1 0 0 -1 0 0 SQL DIAGNOSTIC INFORMATION
DSNT416I SQLERRD = X'00000001' X'00000000' X'00000000'
                  X'FFFFFFFF' X'00000000' X'00000000' SQL DIAGNOSTIC INFORMATION

'Raw' SQLCA format

*** START OF UNFORMATTED SQLCA ***
SQLCAID X(8) SQLCA
SQLCABC I 000000136
SQLCODE I -000000438
SQLERRML SI +000000037
SQLERRMC X(70) INVALID SALARY INCREASE - EXCEEDS 20%
SQLERRP X(8) DSNXRTP
SQLERRD1 I +000000001
SQLERRD2 I +000000000
SQLERRD3 I +000000000
SQLERRD4 I -000000001
SQLERRD5 I +000000000
SQLERRD6 I +000000000
SQLWARN0 X(1)
SQLWARN1 X(1)
SQLWARN2 X(1)
SQLWARN3 X(1)
SQLWARN4 X(1)
SQLWARN5 X(1)
```

```

SQLWARN6 X(1)
SQLWARN7 X(1)
SQLWARN8 X(1)
SQLWARN9 X(1)
SQLWARNA X(1)
SQLSTATE X(5) 75001
*** END OF UNFORMATTED SQLCA ***

```

3.8 Error handling

If a triggered SQL statement fails, SQLCODE -723 is returned to the application (the triggering SQL statement). The SQLCA contains the trigger name, the original SQLCODE and as much of the original message as possible from the failed triggered SQL statement. In some cases, the original error is returned. These include errors that cause an automatic rollback, that put the application process into a must-rollback state. In these cases, the SQLCODE associated with that error is reported back to the application.

Example 3-13 shows you what is returned in case the trigger somehow fails to execute. In this case the UPDATE statement (not shown) fails with an SQLCODE -723. The trigger DB28710.TR5EMP that is fired by the UPDATE statement failed because the table space DSNDB04.LOGGING, that is part of the trigger action, is stopped for all access (x'00C90081') and therefore the trigger gets an SQLCODE -904. As you can see, the original error (-904) is nicely imbedded in the SQLCODE -723 that is returned to the program.

The easiest way to format the information returned in the SQLCA is probably by calling DSNTIAR, but you can also handle the information yourself, as shown in the bottom part of the example.

Example 3-13 Sample information returned when trigger receives an SQLCODE

Formatting done by DSNTIAR

```

DSNT408I SQLCODE = -723, ERROR: AN ERROR OCCURRED IN A TRIGGERED SQL STATEMENT IN
                                TRIGGER DB28710.TR5EMP, SECTION NUMBER 2.
                                INFORMATION RETURNED: SQLCODE -904, SQLSTATE 57011,
                                AND MESSAGE TOKENS 00C90081,00000200,DSNDB04.LOGGING
DSNT418I SQLSTATE = 09000 SQLSTATE RETURN CODE
DSNT415I SQLERRP = DSNXRUID SQL PROCEDURE DETECTING ERROR
DSNT416I SQLERRD = -110 13172746 0
                                13223106 -974970879 12714050 SQL DIAGNOSTIC INFORMATION
DSNT416I SQLERRD = X'FFFFFF92' X'00C9000A' X'00000000'
                                X'00C9C4C2' X'C5E32001' X'00C20042' SQL DIAGNOSTIC INFORMATION

```

'Raw' SQLCA format

```

*** START OF UNFORMATTED SQLCA ***
SQLCAID X(8) SQLCA
SQLCABC I 000000136
SQLCODE I -000000723
SQLERRML SI +000000062
SQLERRMC X(70) DB28710.TR5EMP 2 -904 57011 00C90081,00000200,DSNDB04.LOGGING
DSNXRUID
SQLERRP X(8) DSNXRUID
SQLERRD1 I -000000110
SQLERRD2 I +013172746
SQLERRD3 I +000000000
SQLERRD4 I +013223106

```



```

SQLERRD5 I -974970879
SQLERRD6 I +012714050
SQLWARN0 X(1)
SQLWARN1 X(1)
SQLWARN2 X(1)
SQLWARN3 X(1)
SQLWARN4 X(1)
SQLWARN5 X(1)
SQLWARN6 X(1)
SQLWARN7 X(1)
SQLWARN8 X(1)
SQLWARN9 X(1)
SQLWARNA X(1)
SQLSTATE X(5) 09000
*** END OF UNFORMATTED SQLCA ***

```

If a stored procedure (SP) or UDF is invoked by a trigger, and the SP/UDF encounters an error, the SP/UDF can choose to ignore the error and continue or it can return an error to the trigger.

Attention: If the SQLCODE is ignored by the SP or the UDF and returns to the invoking trigger, the triggering action is NOT undone. To avoid data inconsistencies it is best (easiest) for the SP/UDF to issue a ROLLBACK. This will place the SP/UDF in a MUST_ROLLBACK state and will cause the triggering action to be rolled back also. Another way is to return an SQLSTATE to the trigger that will translate into an SQLCODE -723.

External UDFs or stored procedures can be written to perform exception checking and to return an error if an exception is detected. When a SP/UDF is invoked from a trigger, and that SP/UDF returns an SQLSTATE, this SQLSTATE is translated into a negative SQLCODE by DB2. The trigger execution terminates and all database changes performed as part of the triggering operation are backed out.

Remember that a stored procedure cannot return output parameters (containing for instance the SQLCODE) when invoked from a trigger. To pass back an SQLSTATE to the invoking trigger, the PARAMETER STYLE DB2SQL option has to be used on the CREATE PROCEDURE or CREATE FUNCTION statement. This indicates to DB2 that additional information can be passed back and forth between the caller and the procedure or function. This information includes the SQLSTATE and a diagnostic string. For more details on how to use PARAMETER STYLE DB2SQL, please refer to the *DB2 UDB for OS/390 and z/OS Version 7 Application Programming and SQL Guide*, SC26-9933.

Example 3-14 shows how the error is returned to the triggering statement. A full listing and additional information is available in the additional material that can be downloaded from the Internet (see Appendix C, “Additional material” on page 251) as well as “Returning SQLSTATE from a stored procedure to a trigger” on page 244. Both the SQLSTATE 38601 and diagnostic string 'SP HAD SQL ERROR' are set by the stored procedure after it detects its initial error.

Example 3-14 Passing SQLSTATE back to a trigger

Formatted by DSNTIAR

```

DSNT408I SQLCODE = -723, ERROR: AN ERROR OCCURRED IN A TRIGGERED SQL STATEMENT IN
TRIGGER SC246300.TSD0BMS3, SECTION NUMBER 2.

```

```

INFORMATION RETURNED: SQLCODE -443, SQLSTATE 38601, AND MESSAGE TOKENS
SDOBMS3,SDOBMS3,SP HAD SQL ERROR,
DSNT418I SQLSTATE = 09000 SQLSTATE RETURN CODE
DSNT415I SQLERRP = DSNXRRTN SQL PROCEDURE DETECTING ERROR
DSNT416I SQLERRD = -891 0 0 -1 0 0 SQL DIAGNOSTIC INFORMATION
DSNT416I SQLERRD = X'FFFFFFC85' X'00000000' X'00000000'
X'FFFFFFF' X'00000000' X'00000000' SQL DIAGNOSTIC INFORMATION

```

Or directly from the SQLCA

```

*** START OF UNFORMATTED SQLCA ***
SQLCAID X(8) SQLCA
SQLCABC I 000000136
SQLCODE I -000000723
SQLERRML SI +000000064
SQLERRMC X(70) SC246300.TSDOBMS3 2 -443 38601 SDOBMS3,SDOBMS3,SP HAD SQL ERROR,
SQLERRP X(8) DSNXRRTN
SQLERRD1 I -000000891
SQLERRD2 I +000000000
SQLERRD3 I +000000000
SQLERRD4 I -000000001
SQLERRD5 I +000000000
SQLERRD6 I +000000000
SQLWARN0 X(1)
SQLWARN1 X(1)
SQLWARN2 X(1)
SQLWARN3 X(1)
SQLWARN4 X(1)
SQLWARN5 X(1)
SQLWARN6 X(1)
SQLWARN7 X(1)
SQLWARN8 X(1)
SQLWARN9 X(1)
SQLWARNA X(1)
SQLSTATE X(5) 09000
*** END OF UNFORMATTED SQLCA ***

```

If a trigger invokes a stored procedure or external UDF and that procedure or function does something that puts the thread into a must-rollback state, no further SQL is allowed. SQLCODE -751 is returned to the trigger which causes the trigger to terminate. All database changes performed as part of the triggering operation are backed out and control is returned to the application. All subsequent SQL statements receive an SQLCODE -919.

3.9 Trigger cascading

The activation of a trigger can cause trigger cascading. An example of this is shown in Figure 3-3. This occurs when the activation of one trigger executes SQL statements that cause the activation of other triggers or even the same trigger again. A long chain of triggers and referential integrity delete rules can cause significant change to the database as a result

of a single delete, insert or update operation. In addition, triggers can invoke external UDFs and stored procedures, which in turn can activate other triggers, UDFs and stored procedures. The cascade path can therefore involve a combination of triggers, UDFs and stored procedures.

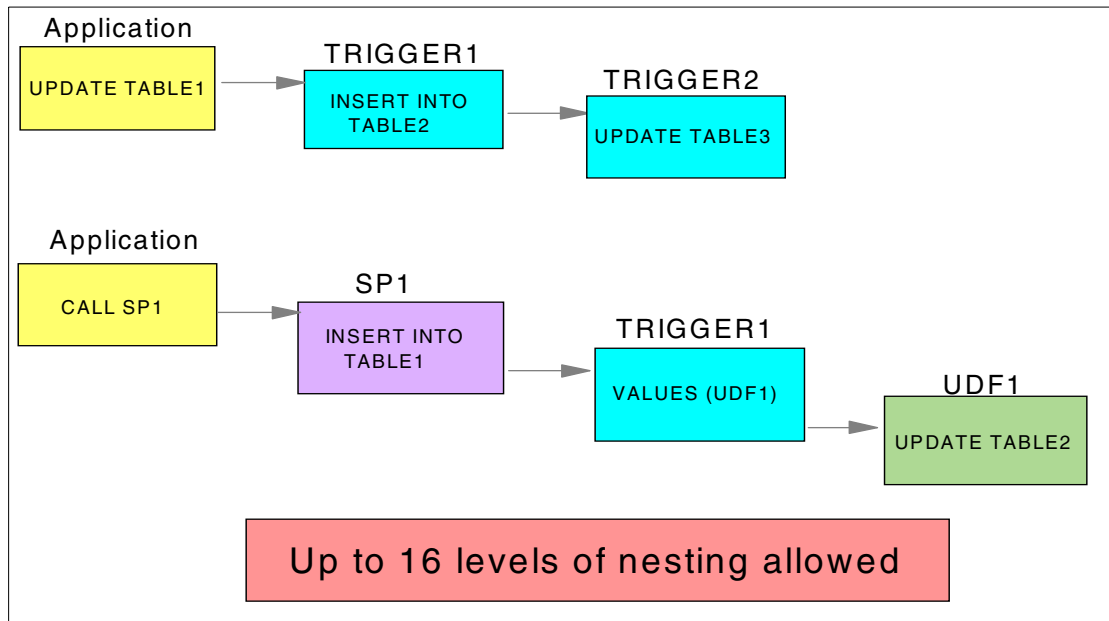


Figure 3-3 Trigger cascading

The allowed run-time depth level of a trigger, UDF or stored procedure is 16. If a trigger, UDF or stored procedure at nesting level 17 is activated, SQLCODE -724 is returned to the application. None of the database changes made as part of the triggering operation are applied to the database. This provides protection against endless loop situations that can be created with triggers.

In Example 3-15, you can see the error message you receive in your application program if you attempt to go beyond the 16 nesting levels permitted.

Example 3-15 Cascading error message

```

DSNT408I SQLCODE = -724, ERROR: THE ACTIVATION OF THE TRIGGER OBJECT objectname
WOULD EXCEED THE MAXIMUM LEVEL OF INDIRECT SQL CASCADING
    
```

Tip: Remember before triggers cannot cause trigger cascading because INSERT, UPDATE, and DELETE statements are not allowed in a before trigger.

3.10 Global trigger ordering

Multiple triggers can be created on the same triggering table and for the same triggering operation and trigger activation time. The order in which these triggers are activated is the order in which they were created. That is, the most recently created trigger will be the last trigger activated.

When triggers are defined using the CREATE TRIGGER statement, their creation time is registered by a timestamp in the DB2 catalog table SYSIBM.SYSTRIGGERS. The value of this timestamp is subsequently used to order the activation of triggers when there is more than one trigger that should be run at the same time.

Existing triggers are activated before new triggers so that new triggers can be used as incremental additions to the logic that affects the database. For example, if a triggered SQL statement of trigger T1 inserts a new row into a table T, a triggered SQL statement of trigger T2 that is run after T1 can be used to update the new row table T with specific values.

This ordering scheme means that if you drop the first trigger created and re-create it, it will become the last trigger to be activated by DB2.

Tip: If you want a trigger to remain the first one activated, you must drop all the triggers defined on the table and recreate them in the order you want them executed.

3.11 When external actions are backed out

When external actions are driven by triggers, those actions are generally not under the recovery control of DB2 in the event of a failure. There are two cases to consider:

- ▶ Case 1: The trigger and external action complete successfully, but the application later executes a rollback operation.

External actions are rolled back if the DB2 application and the affected external resource managers are under the control of the OS/390 Transaction Management and Recoverable Resource Manager Services (RRS) for commitment control. With this facility, all actions performed by resource managers under the control of RRS are treated as a single unit of recovery.

- ▶ Case 2: The external action completes successfully, but an error occurs later during the processing of the trigger or triggering operation.

In the case of a trigger or triggering operation failure after external actions have completed, the thread is put in a must-rollback state if the DB2 application and the affected external resource managers are under the control of RRS. When this occurs, DB2 does not allow any further SQL requests until a rollback operation is performed. When the rollback occurs, changes to any external resource managers under the same commit control, are rolled back.

3.12 Passing transition tables to SPs and UDFs

Transition tables can be passed from a trigger as arguments to stored procedures and user-defined functions. Using table locators, transition tables can be referenced within a stored procedure or UDF in the FROM clause of a SELECT statement or in the subselect of an INSERT statement.

Table locators can be passed as arguments in VALUES and CALL statements. A table locator cannot be used as an argument outside of a trigger action.

The trigger definition shown in Example 3-16, uses the TABLE keyword to pass new transition table NTAB as a table locator argument to a SP called SPTRTT.

Example 3-16 Using table locators

```
CREATE PROCEDURE SYSPROC.SPTRTT
```

```

      (IN TABLE LIKE SC246300.TBEMPLOYEE AS LOCATOR)
LANGUAGE COBOL
EXTERNAL NAME SPTRTT
COLLID BARTCOB
PROGRAM TYPE MAIN
NO WLM ENVIRONMENT
PARAMETER STYLE DB2SQL #

CREATE TRIGGER SC246300.TRTTTR
AFTER UPDATE OF SALARY ON SC246300.TBEMPLOYEE
REFERENCING NEW_TABLE AS NTAB
FOR EACH STATEMENT MODE DB2SQL
BEGIN ATOMIC
  CALL SPTRTT (TABLE NTAB) ;
END #

```

The SP SPTRTT is defined with a single table locator argument. The keyword LIKE followed by SC246300.TBEMPLOYEE specifies that the table represented by the table locator has the same column names and data types as table SC246300.TBEMPLOYEE.

To access a transition table in an external UDF or stored procedure, you need to:

1. Declare a parameter to receive the table locator
2. Declare a table locator host variable
3. Assign the parameter value to the table locator host variable
4. Use the table locator host variable to reference the transition table

In Example 3-17, the SQL syntax used to declare the table locator host variable TRIG-TBL-ID is transformed by the precompiler to a COBOL host language statement. The keyword LIKE followed by *table-name* specifies that the table represented by the table locator host variable TRIG-TBL-ID has the same column names and data types as table SC246300.TBEMPLOYEE. A full listing is available in “Passing a transition table from a trigger to a SP” on page 246 and in the additional material downloadable from the Internet. See Appendix C, “Additional material” on page 251 for instructions.

Using a transition table is an interesting technique. This way you call the stored procedure only once using a statement trigger, instead of using a row trigger that would call the stored procedure for every row that is updated. Passing a transition table to a UDF or SP allows you to do things that you cannot do with row triggers like calculations based on the entire set of rows that were changed by the triggering action.

Example 3-17 Sample COBOL program using a SP and table locator

```

IDENTIFICATION DIVISION.
PROGRAM-ID. "SPTRTT".
DATA DIVISION.
WORKING-STORAGE SECTION.

.....
* *****
* 2. DECLARE TABLE LOCATOR HOST VARIABLE TRIG-TBL-ID
* *****

  01 TRIG-TBL-ID SQL TYPE IS
      TABLE LIKE SC246300.TBEMPLOYEE AS LOCATOR.

.....

LINKAGE SECTION.
* *****
* 1. DECLARE TABLOC AS LARGE INTEGER PARM

```

```

* *****
01 TABLOC PIC S9(9) USAGE BINARY.
01 INDTABLOC PIC S9(4) COMP.
.....
PROCEDURE DIVISION USING TABLOC , INDTABLOC, P-SQLSTATE,
                        P-PROC, P-SPEC, P-DIAG.

* *****
* 4. DECLARE CURSOR USING THE TRANSITION TABLE
* *****
EXEC SQL
  DECLARE C1 CURSOR FOR
  SELECT EMPNO, FIRSTNME
  FROM TABLE ( :TRIG-TBL-ID LIKE SC246300.TBEMPLOYEE )
END-EXEC.
* *****
* 3. COPY TABLE LOCATOR INPUT PARM TO THE TABLE LOCATOR HOST VAR
* *****
MOVE TABLOC TO TRIG-TBL-ID.
.....
Start processing the transition table

```

3.13 Trigger package

A trigger package is created by DB2 when a CREATE TRIGGER statement is executed. Trigger packages are recorded in the DB2 catalog table SYSIBM.SYSPACKAGE.

The name of the created trigger package is the same as the name of the created trigger. The collection-id of the package is the same as the schema name of the trigger. The CREATE TRIGGER statement fails if a package already exists with the same name. The BIND PACKAGE options are set by DB2 when the trigger is created.

A trigger package is always accessible regardless of the plan or package being executed when a trigger is activated. There is no need to include trigger packages in the package list.

It is not necessary to grant privileges on a trigger package to executors. There is no execution time authorization check for triggers.

A trigger package cannot be deleted by a FREE PACKAGE subcommand or DROP PACKAGE statement. A DROP TRIGGER statement must be used to delete the trigger package. A trigger package cannot be executed by normal application programming interfaces.

Versioning is not allowed for trigger packages and the version ID column for triggers packages in the catalog reflects an empty string.

Note: Only a package is created for the trigger and no associated plan is created. Trigger packages are implicitly loaded at execution time.

Tip: We recommend that you treat trigger packages in the same way as standard packages in that you REBIND them when you REBIND other types of package, for example, when there are significant changes to the statistics. This ensures that access paths are based on accurate information.

3.14 Rebinding a trigger package

A trigger package can be explicitly rebound using the REBIND TRIGGER PACKAGE subcommand, see Example 3-18. You can use this subcommand to change a limited subset of the bind options (CURRENTDATA, ENCODING, IMMEDIATEWRITE, EXPLAIN, FLAG, ISOLATION, RELEASE) that DB2 used to create the package. You might also rebound a trigger package to re-optimize its SQL statements after you create a new index or use the RUNSTATS utility. A trigger package cannot be explicitly bound using the BIND PACKAGE subcommand and BIND PACKAGE cannot be used to create a trigger package, replace an existing trigger package, nor copy an existing trigger package.

Example 3-18 Rebinding a trigger package

```
REBIND TRIGGER PACKAGE (SG246300.ITEMNMBR)
  EXPLAIN(YES)
  CURRENTDATA(NO)
  ISOLATION(CS) ;
```

3.15 Trigger package dependencies

Triggers are dependent on the existence of all referenced database objects and on all the privileges required for the creation of the trigger. These dependencies are enforced as dependencies of the trigger package. Each dependency is recorded in the DB2 catalog table SYSIBM.SYSPACKDEP. Please note that if a Stored Procedure (SP) is called in a trigger body, the trigger package dependency only shows a dependency on the SP and not the objects the SP references.

When an object is dropped or a privilege is revoked, any trigger package containing a trigger action that references that object or requires that privilege, is invalidated. Like a regular package, an automatic rebound is performed on the invalid trigger package when the trigger is next activated. A trigger package that has been marked invalid has the VALID column set to 'N'. If a subsequent INSERT, UPDATE, or DELETE statement activates an invalid trigger, the SQL statement will fail with a resource unavailable condition (SQLCODE -904). To make the SQL statement work again, you must either drop the trigger or fix the problem that invalidated the trigger and explicitly rebound the trigger package. The information that is returned to the application, after DSNTIAR formatting, looks like Example 3-19.

(DB28710.TR5EMP.16D1060E1 is the name of the trigger package and the start of the consistency token, since not all fits in the allotted space in the SQLERRMC field of the SQLCA.)

Example 3-19 Information returned when trigger package is invalid

```
DSNT408I SQLCODE = -723, ERROR: AN ERROR OCCURRED IN A TRIGGERED SQL STATEMENT
        IN TRIGGER DB28710.TR5EMP, SECTION NUMBER 1.
        INFORMATION RETURNED: SQLCODE -904, SQLSTATE 57011, AND MESSAGE TOKENS
        00E30305,00000801,DB28710.TR5EMP.16D1060E1
DSNT418I SQLSTATE = 09000 SQLSTATE RETURN CODE
DSNT415I SQLERRP = DSNXEAL SQL PROCEDURE DETECTING ERROR
DSNT416I SQLERRD = -150 0 0 -1 0 0 SQL DIAGNOSTIC INFORMATION
DSNT416I SQLERRD = X'FFFFFF6A' X'00000000' X'00000000' X'FFFFFFF'
        X'00000000' X'00000000' SQL DIAGNOSTIC INFORMATION
```

Note: A user-defined function cannot be dropped if it is referenced within a triggered SQL statement.

Tip: If a triggering table is dropped, its triggers are also dropped. If the table is recreated, and you want the triggers back, then they must be recreated.

3.16 DROP, GRANT, and COMMENT ON statements

DROP TRIGGER also drops the package used by the trigger. The RESTRICT clause must be specified when dropping a trigger in Version 6, even if a trigger does not have dependencies that would restrict (prevent) it from being dropped. The restrict clause is no longer needed in V7.

```
DROP TRIGGER trigger-name RESTRICT (Version 6)
```

```
DROP TRIGGER trigger-name (Version 7)
```

The GRANT TRIGGER ON TABLE statement grants the privilege to use the CREATE TRIGGER statement on the specified table(s).

```
GRANT TRIGGER ON table-name TO ...
```

You can add comments for triggers.

```
COMMENT ON TRIGGER trigger-name IS string-constant
```

Example 3-20 Comment on trigger

```
COMMENT ON TRIGGER CHK_SAL
        IS 'Validate that salary increase is not more than 20%'
```

Tip: We recommend that you use COMMENT ON TRIGGER to comment the function implemented by the trigger.

3.17 Catalog changes

There is a new catalog table SYSIBM.SYSTRIGGERS in DSNDB06.SYSOBJ, which contains at least one row for each trigger and records the trigger characteristics. Some of the trigger characteristics stored in this table are: creation timestamp, validity marker, and the full CREATE TRIGGER statement text. If the CREATE TRIGGER statement text is longer than 3460 characters, then additional rows are placed in SYSIBM.SYSTRIGGERS to contain the remainder of the trigger statement text.

Some new columns have been added to existing catalog tables. The TYPE column was added to SYSIBM.SYSPACKAGE to indicate trigger packages. Type 'T' is for triggers. Also, TRIGGERAUTH was added to SYSTABAUTH to record the TRIGGER privilege.

3.18 Trigger and constraint execution model

When constructing systems with complex combinations of triggers and declarative constraints, it is important to understand the processing model to achieve the desired results. The general order of processing for an SQL statement that updates a table is shown in Figure 3-4. What follows is a description of the boxes and other items in the figure. SQL statement S1 is the DELETE, INSERT, or UPDATE statement that begins the process. S1 identifies a table (or an updatable view over some table) referred to as the target table throughout this description.

1. Determine the set of affected rows (SAR)

This step is the starting point for a process that repeats for referential constraint delete rules of CASCADE and SET NULL and for cascaded SQL statements from after triggers.

The purpose of this step is to determine the SAR for the SQL statement. The set of rows included in the SAR is based on the statement:

- For INSERT, the rows identified by the VALUES clause or the fullselect.
- For UPDATE, all rows that satisfy the search condition or the current row for a positioned UPDATE.
- For DELETE, all rows that satisfy the search condition or the current row for a positioned DELETE.

2. Process before triggers

All before triggers are processed in ascending order of creation. Each before-row trigger is activated once for each row in the SAR.

An error may occur during the processing of a trigger action in which case all changes made as a result of the original SQL statement S1 (so far) are rolled back.

If there are no before triggers or the SAR is empty, this step is skipped.

3. Apply constraints

The constraints associated with the target table are applied. This includes referential constraints, table check constraints and checks associated with the WITH CHECK OPTION on views. Referential constraints with delete rules of CASCADE or SET NULL may cause additional triggers to be activated.

A violation of any constraint or WITH CHECK OPTION results in an error and all changes made as a result of the original S1 (so far) are rolled back.

If the SAR is empty, this step is skipped.

4. Apply the SAR to the target table

The actual DELETE, INSERT, or UPDATE is applied using the SAR to the target table.

An error may occur when applying the SAR (such as attempting to insert a row with a duplicate key where a unique index exists) in which case all changes made as a result of the original SQL statement S1 (so far) are rolled back.

5. Process after triggers

All after triggers activated by S1 are processed in ascending order of creation.

After-statement triggers are activated exactly once, even if the SAR is empty. After-row triggers are activated once for each row in the SAR.

An error may occur during the processing of a trigger action in which case all changes made as a result of the original S1 (so far) are rolled back.

The trigger action of an after trigger may include triggered SQL statements that are DELETE, INSERT or UPDATE statements. Each such statement is considered a cascaded SQL statement because it starts a cascaded level of trigger processing. This can be thought of as assigning the triggered SQL statement as a new S1 and performing all of the steps described here recursively.

Once all triggered SQL statements from all after triggers activated by each S1 have been processed to completion, the processing of the original S1 is complete.

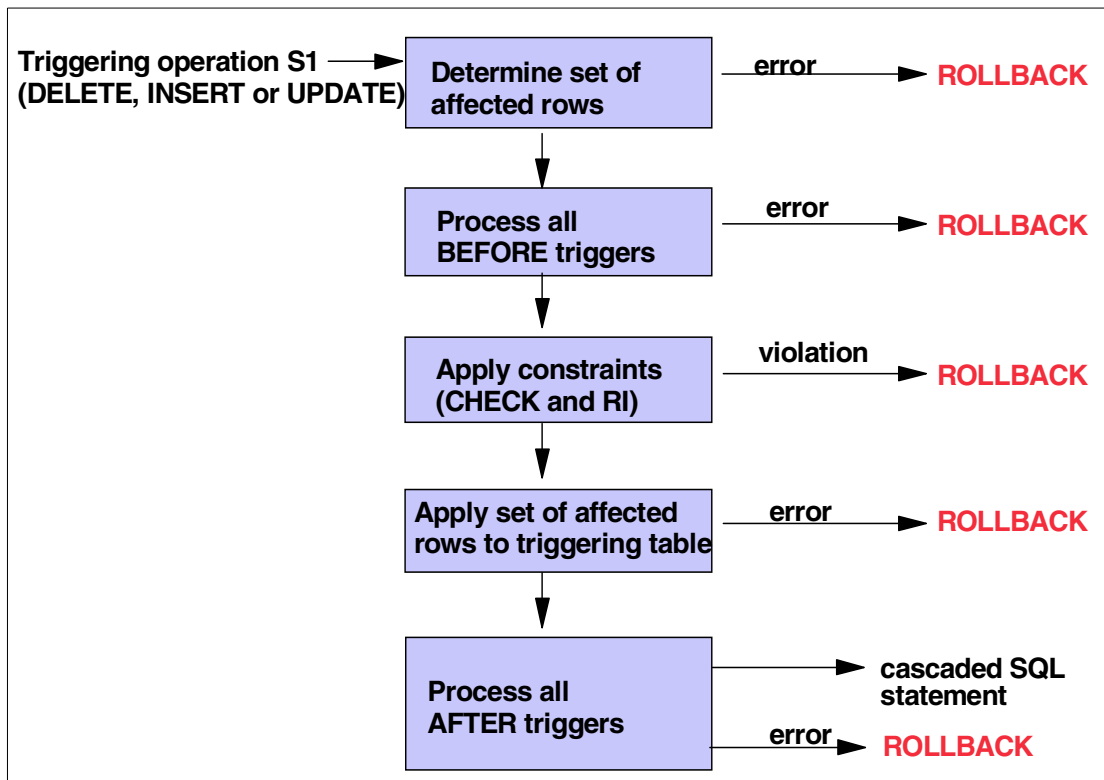


Figure 3-4 SQL processing order and triggers

3.19 Design considerations

Triggers might have a performance impact. There are a number of issues that you must be aware of:

- ▶ During the execution of an SQL statement, any activated triggers add, of course, to the resource utilization needed to execute the statement. This is somewhat offset by the fact that the application does not need to perform the trigger actions.
- ▶ When optimizing an SQL INSERT, UPDATE or DELETE statement, the DB2 optimizer does not know what triggers will be executed and cannot take this cost into account when choosing an access path. For triggers, only default values are inserted into the SQL_STATEMENT_TABLE and a COST_CATEGORY of “B” and a REASON of “TRIGGERS”.
- ▶ The predictive governor does not include the cost of trigger execution in its execution time estimate.

Now, let us review the factors that influence the cost of triggers. Understanding these factors help you evaluate the likely cost of triggers and estimate their cost relative to an application implementation of the logic, or the use of check constraints, or referential integrity.

- ▶ The base cost of a trigger (that is, all work except for the execution of the SQL in the trigger body) is about equivalent to the cost of a fetch. Where a trigger is defined but not fired, for example, if the trigger is defined as AFTER/BEFORE UPDATE OF *column1* and an UPDATE statement updates *column2*, the trigger is not activated, and so the overhead is normally negligible. However, a very complicated and badly coded WHEN clause in the trigger can still impact performance whether or not the trigger is fired.
- ▶ The trigger package has to be loaded into the EDM pool. I/O has to occur if it is not already in the EDM pool. Options to alleviate problems in this area include:
 - Monitor and increase the size of the EDM pool.
 - Consider REBIND of the trigger package with the `RELEASE (DEALLOCATE)` option. Be aware, though, that `RELEASE (DEALLOCATE)` results in more resources being held — you face the same issues as binding application packages with `RELEASE (DEALLOCATE)`.
- ▶ The object descriptor (OBD) of the trigger package is a part of the table. Therefore, be aware of the impact to the DBD size and its potential to impact the EDM pool if you already have large DBDs.
- ▶ Transition tables reside in the workfile database (usually named DSNDB07). The cost of a trigger that processes workfiles depends critically on the amount of workfile processing required (including row length and whether you have OLD and NEW transition variables). Transition tables need to be processed with a table space scan for each row trigger because there are no indexes. Trigger performance also depends on whether there is contention for the workfiles. You should anticipate the cost to be several times greater than the base cost of the trigger, because significantly more work has to be carried out.
- ▶ As we have indicated, the use of transition variables and transition tables in AFTER triggers represent a significant proportion of the cost of such a trigger. Where a trigger has to manipulate workfiles, contributions to the cost are:
 - Creation, use, and deletion of workfiles
 - Any I/O or GETPAGE request necessary to process the data
 - Base cost of trigger processing
- ▶ There is no overhead for an SQL statement that is not the triggering action. For example, if a INSERT trigger was defined on a table, it would have no overhead on update or delete statements.

Note: APAR PQ34506 provides an important performance improvement for triggers with a WHERE clause and a subselect. A where clause in the subselect can now be evaluated as a Stage 1 predicate.

We recommend that you prototype your physical design first if you are considering using triggers for tables that are heavily updated and/or fire SQL statements that process significant quantities of data. You can then evaluate their cost relative to the cost of equivalent functionality embedded in your applications. Be cautious when using triggers to create summary tables from tables that are heavily updated, you may end up creating a locking bottleneck on the summary table rows.

When you begin physical design, you may find that you need several triggers defined on a single table. To avoid the overhead of multiple triggers, you can write a stored procedure to do all the triggered processing logic. The body of the trigger could then simply consist of a CALL stored-procedure-name.

When porting applications from other RDBMS systems, don't forget that there may be syntax incompatibility with these other platforms.

Tip: Within DB2's resource limit facility, the execution of a trigger is counted as part of the triggering SQL statement.

Triggers can be very helpful in a number of areas, but can also make the design more complex. You can no longer just rely on looking at the programs to find out what is happening to the data, you also have to look inside the DBMS. Adding triggers to the data model should be implemented with great care, especially when you get into cascading situations. People doing program design should be aware of existing triggers. Otherwise you might end up doing the same update twice, once in the application program and again in the trigger.

Important: You should not use triggers just for the sake of using them. You should first see if what you are trying to implement can be done with a check constraint, if not, then try with referential integrity, if not, then try with a trigger. Do not get "trigger happy"!!!

3.20 Some alternatives to a trigger

Triggers have a number of valuable usages. However, some functions that can be implemented using triggers can be better implemented using other alternatives provided by DB2. Triggers should be defined on tables to enforce rules that involve different states of the data. For rules that do not involve more than one state of the data, table check constraints and referential constraints provide a better solution since they often have performance advantages over triggers.

Constraints are enforced when they are created but the creation of a trigger does not check existing rows and does not cause check pending to be turned on.

Triggers are not triggered by utilities (except by an online LOAD RESUME). In the following subsection, we discuss some alternatives to triggers.

Referential integrity

Triggers are implemented through the Relational Data Systems (RDS) component of DB2 through the use of packages. Referential Integrity is enforced by Data Manager which has a shorter path length and thus has a performance advantage. Referential Integrity is enforced when created and can be checked through existing utilities and triggers cannot.

User-defined defaults

If the default value can change over time, then a trigger may be a better way to implement it since all that is required is dropping and recreate the trigger with the new values. Making such a change to a column defined with a default would involve the unloading, dropping, recreating of the table with the new default value, and reloading the table (this may change in a future version of DB2). As you can see, the trigger would be much less disruptive but takes additional CPU resources to be performed. If the default value does not change, then use user-defined defaults instead of triggers to achieve better performance results.

Data replication and propagation

Although triggers can be used for simple propagation (for example, to create an audit trail), they are not intended to be used as an alternative to, or a replacement for Data Propagator.

Table check constraints

Before triggers are generally used as an extension to the constraint system. There are trade-offs between when to use a trigger versus a table check constraint. A good rule of thumb is that if the values of the constraint are static (no new values added or values not changed very often), then it is better to enforce the constraint via the use of a table check constraint. If the values of the constraint are dynamic (often changed, many new values added, and so on), then a BEFORE trigger would be a better choice to enforce the constraint. For example, (see Example 3-21), if you wanted to specify a constraint to validate a column such as sex, a check constraint would be the better choice to implement it since there are a finite number of sex codes or values and they are not changed very often. However, if you wanted to specify a constraint to validate an item number or store number (see Example 3-22), then a BEFORE trigger could be a better choice to implement the constraint since new items are constantly added and deleted and new stores may be opened or closed frequently depending on the volatility or growth of the business. In this case RI could also be used and may be a better solution.

Constraints are good for declarative comparisons, they are enforced when created and through existing utilities.

In Example 3-21, we demonstrate equivalent constraints. One is coded as a check and the other as a trigger. Lets assume that the values of L_ITEM_NUMBER are constantly changing. That is, new items are often added and old items removed. In order to make these changes to the check constraint, you would have to drop the check constraint and alter the table to re-add it. This causes the table to be in check pending status (with CURRENT RULES = 'DB2') until the check utility completes and the table will be unavailable to the application causing an outage. If high online availability is important, and you can afford the extra cost of processing a trigger, then the trigger is a better choice.

Example 3-21 Check constraint is better than a trigger

```
CREATE TRIGGER SC246300.SEXCNST
NO CASCADE BEFORE INSERT ON SC246300.TBEMPLOYEE
REFERENCING NEW AS N
FOR EACH ROW MODE DB2SQL
```

```

WHEN ( N.SEX NOT IN('M','F'))
  SIGNAL SQLSTATE 'ERRSX' ('SEX MUST BE EITHER M OR F')

CREATE TRIGGER SC246300.SEXCNST
NO CASCADE BEFORE UPDATE ON SC246300.TBEMPLOYEE
REFERENCING NEW AS N
FOR EACH ROW MODE DB2SQL
WHEN ( N.SEX NOT IN('M','F'))
  SIGNAL SQLSTATE 'ERRSX' ('SEX MUST BE EITHER M OR F')

or

ALTER TABLE SC246300.TBEMPLOYEE
  ADD CHECK (SEX IN ('M','F'))

```

Example 3-22 Trigger is better than a check constraint

```

CREATE TRIGGER SC246300.ITEMNMBR
NO CASCADE BEFORE INSERT ON SC246300.TBLINEITEM
REFERENCING NEW AS N
FOR EACH ROW MODE DB2SQL
WHEN ( N.L_ITEM_NUMBER NOT IN (1, 5, 6, ..... 9996,9998,10000) )
  SIGNAL SQLSTATE 'ERR30' ('ITEM NUMBER DOES NOT EXIST')

CREATE TRIGGER SC246300.ITEMNMBR
NO CASCADE BEFORE UPDATE ON SC246300.TBLINEITEM
REFERENCING NEW AS N
FOR EACH ROW MODE DB2SQL
WHEN ( N.L_ITEM_NUMBER NOT IN (1, 5, 6, ..... 9996,9998,10000) )
  SIGNAL SQLSTATE 'ERR30' ('ITEM NUMBER DOES NOT EXIST')

or

ALTER TABLE SC246300.TBLINEITEM
  ADD CHECK (L_ITEM_NUMBER IN (1, 5, 6, ..... 9996,9998,10000))

```

In Example 3-23, we show another way that the trigger in Example 3-22 can be coded. This example is more flexible since there is no need to update the trigger with new values but merely to insert the new values in a table called TBITEMS. This could also be accomplished more efficiently with RI because a foreign key causes less overhead and guarantees the consistency when using utilities (the trigger is only enforced in an online LOAD RESUME utility). However, a foreign key must reference a unique key and a trigger does not have that requirement. The trigger can return a customized error message.

Example 3-23 Alternative trigger

```

CREATE TRIGGER SC246300.ITEMNMB2
NO CASCADE BEFORE INSERT ON SC246300.TBLINEITEM
REFERENCING NEW AS N
FOR EACH ROW MODE DB2SQL
WHEN ( N.L_ITEM_NUMBER NOT IN
  ( SELECT ITEM_NUMBER FROM SC246300.TBITEMS
    WHERE N.L_ITEM_NUMBER = ITEM_NUMBER )
  SIGNAL SQLSTATE 'ERR30' ('ITEM NUMBER DOES NOT EXIST')

```

3.21 Useful queries

In Example 3-24, we provide a query that can be used to identify all the triggers defined for a particular table and shows the order in which the triggers are executed. The column TEXT contains the entire DDL used to create the trigger. If you are using SPUFI, you may have to modify the column width default to be able to view the entire content of the column TEXT.

Example 3-24 Identify all triggers for a table

```
SELECT SCHEMA
       ,NAME
       ,CASE WHEN TRIGTIME = 'B' THEN 'BEFORE'
             WHEN TRIGTIME = 'A' THEN 'AFTER'
             ELSE ' '
       END AS TIME
       ,CASE WHEN TRIGEVENT = 'I' THEN 'INSERT OF'
             WHEN TRIGEVENT = 'U' THEN 'UPDATE OF'
             WHEN TRIGEVENT = 'D' THEN 'DELETE OF'
             ELSE ' '
       END AS EVENT
       ,CASE WHEN GRANULARITY = 'R' THEN 'ROW'
             WHEN GRANULARITY = 'S' THEN 'STATEMENT'
             ELSE ' '
       END AS GR
       ,REPLACE(TEXT,' ','') AS TRIGGER
       --TEXT IS 3460 BYTES WIDE AND HAS A LOT OF BLANKS
FROM   SYSIBM.SYSTRIGGERS
       WHERE TBOWNER = 'SC246300'
       AND   TBNAME  = 'TBORDER'
ORDER BY TRIGTIME DESC, CREATEDTS, SEQNO;
```

In Example 3-25, we provide a query that can be used to identify all the triggers defined for all the tables in a particular database and the order in which the triggers are executed.

Example 3-25 Identify all triggers for a database

```
SELECT TBOWNER
       , TBNAME
       , SCHEMA
       , T. NAME
       , CASE WHEN TRIGTIME = 'B' THEN 'BEFORE'
             WHEN TRIGTIME = 'A' THEN 'AFTER'
             ELSE ' '
       END AS TIME
       , CASE WHEN TRIGEVENT = 'I' THEN 'INSERT OF'
             WHEN TRIGEVENT = 'U' THEN 'UPDATE OF'
             WHEN TRIGEVENT = 'D' THEN 'DELETE OF'
             ELSE ' '
       END AS EVENT
       , CASE WHEN GRANULARITY = 'R' THEN 'ROW'
             WHEN GRANULARITY = 'S' THEN 'STATEMENT'
             ELSE ' '
       END AS GR
       , REPLACE(TEXT,' ','') AS TRIGGER
       --TEXT IS 3460 BYTES WIDE AND HAS LOT OF BLANKS
```

```
FROM SYSIBM.SYSTRIGGERS T, SYSIBM.SYSDATABASE D
WHERE D.NAME = 'DB246300'
      AND T.DBID = D.DBID
ORDER BY T.BOWNER, TBNAME, TRIGTIME DESC, T.CREATEDTS, SEQNO;
```

3.22 Trigger restrictions

Triggers cannot be created on a view, temporary table, catalog table, auxiliary table, alias, or synonym.

No program code other than calls to stored procedures or user-defined functions are allowed in the body of a trigger.

Three-part names cannot be used within the body of a trigger. However, a trigger can call a stored procedure or UDF that can access or change data on a remote system.

The maximum number of triggers, stored procedures, and user-defined functions that an SQL statement can implicitly or explicitly reference is 16 nesting levels. If you attempt to go beyond the 16 nesting levels, your application program receives an `SQLCODE = -724`.

The LOAD Utility does not cause triggers to get activated. Online LOAD RESUME does activate the triggers.

Triggers can be activated by UPDATE, INSERT, and DELETE operations but not by SELECT operations.



User-defined distinct types (UDT)

User-defined distinct types (UDTs) allow you to define data types to DB2 based on the built-in data types (like CHAR, VARCHAR, DECIMAL, INTEGER and SMALLINT). By allowing the installation to define data types to DB2, code to enforce data typing is not required within the application.

User-defined distinct types are also used in object-oriented systems for strong typing, because the distinct type reflects the use of the data that is required and/or allowed. Strong typing is especially valuable for ad-hoc access to data. The user taking advantage of a query tool might attempt to compare such things as “euros” and “pesetas” without realizing his or her error. DB2 can now prevent such comparisons through the implementation of UDTs.

4.1 Introduction

A distinct type is based (*sourced*) on an existing built-in data type.

Once a distinct type is defined, column definitions can reference that type when tables are created or altered. A UDT is a schema object. If a distinct type is referenced without a schema name, the distinct type is resolved by searching the schemas in the CURRENT PATH.

Each column in a table has a specific data type which determines the column's data representation and the operations that are allowed on that column. DB2 supports a number of built-in data types, for example, INTEGER and CHARACTER. In building a database, you might decide to use one of the built-in data types in a specialized way; for example, you might use the INTEGER data type to represent ages, or the DECIMAL(8,2) data type to represent amounts of money. When you do this, you might have certain rules in mind about the kinds of operations that make sense on your data. For example, it makes sense to add or subtract two amounts of money, but it might not make sense to multiply two amounts of money, and it almost certainly makes no sense to add or compare an age to an amount of money.

UDTs provide a way for you to declare such specialized usages of data types and the rules that go with them. DB2 enforces the rules, by performing only the kinds of computations and comparisons that you have declared to be reasonable for your data. You have to define the operations that are allowed on the UDT. In other words, DB2 guarantees the type-safety of your queries.

4.2 Creating distinct data types

The way to declare a specialized use of data is to create a new data type of your own, called a *user-defined distinct type* (UDT), to supplement DB2's built-in data types. UDTs are defined with the CREATE DISTINCT TYPE DDL statement. The distinct-type-name is a two-part name which must be unique within the DB2 subsystem. The qualifier is a schema name.

The owner of the distinct type is determined in a similar way to other DDL. The owner is given the USAGE privilege on the distinct type, and the EXECUTE privilege on each of the generated CAST functions, with the GRANT option. (CAST functions are discussed in more detail in 4.3, "CAST functions" on page 45.)

The distinct type shares a common internal representation with one of the built-in data types, called its source data type. This is also known as the distinct type is *sourced* on a built-in data type. Despite this common representation, the distinct type is considered to be a separate data type, distinct from all others (hence the name).

Note: You cannot specify LONG VARCHAR or LONG VARGRAPHIC as the source type of a distinct type.

In Example 4-1, we show three user-defined distinct types: two UDTs are based on the built-in data type DECIMAL and are intended to represent monetary values in European euros and in Spanish pesetas; the other is based on the built-in data type CHARACTER and is created for customer identification.

Example 4-1 Sample DDL to create UDTs

```
CREATE DISTINCT TYPE SC246300.PESETA
AS DECIMAL(18,0)
```

```

WITH COMPARISONS ;

CREATE DISTINCT TYPE SC246300.EURO
AS DECIMAL(17,2)
WITH COMPARISONS ;

CREATE DISTINCT TYPE SC246300.CUSTOMER
AS CHAR(11)
WITH COMPARISONS ;

```

An instance of a distinct type is considered comparable only with another instance of the same distinct type. The WITH COMPARISONS clause serves as a reminder that instances of the new distinct type can be compared with each other, using any of the comparison operators. (For a list of comparison operators allowed on UDTs, see Figure 4-1 on page 49.) This clause is required if the source data type is not a large object data type. If the source data type is BLOB, CLOB, or DBCLOB, the phrase is tolerated with a warning message (SQLCODE +599, SQLSTATE 01596), even though comparisons are not supported for these source data types.

Note: Do not specify WITH COMPARISONS for sourced-data-types that are BLOBS, CLOBs, or DBCLOBs. The WITH COMPARISONS clause is required for all other source-data-types.

4.3 CAST functions

One of the characteristics of a UDT is that they enforce strong typing. This implies that columns that are defined with that distinct type can only be compared to other columns using the same distinct type. You can only compare EURO columns to other EURO columns for example. This is even true when comparing the distinct type (EURO) to its built-in type (DECIMAL(17,2)).

In order to enable you to convert between the built-in data type and a distinct type in both directions, CAST functions are used. These CAST functions are automatically generated because we specified the WITH COMPARISONS clause on the CREATE DISTINCT TYPE SQL statements.

In Example 4-2, we show the equivalent of the CAST functions that are automatically generated from the creates performed in Example 4-1. The CAST function to convert from the source data type to the distinct data type has the same name as the UDT (in this case PESETA and EURO). The CAST function to convert from the UDT to the source data type will have the same name as the source data type (in this case DECIMAL).

Example 4-2 Automatically generated CAST functions

```

-- Function PESETA(DECIMAL) returns a PESETA type
CREATE FUNCTION SC246300.PESETA(DECIMAL(18,0)) RETURNS SC246300.PESETA
SOURCE SYSIBM.DECIMAL(DECIMAL(18,0)) ;

-- Function DECIMAL(PESETA) returns a DECIMAL type
CREATE FUNCTION SC246300.DECIMAL(PESETA) RETURNS SYSIBM.DECIMAL
SOURCE SYSIBM.DECIMAL(DECIMAL(18,0)) ;

```

```

-- Function EURO(DECIMAL) returns a EURO type
CREATE FUNCTION SC246300.EURO(DECIMAL(17,2)) RETURNS SC246300.EURO
SOURCE SYSIBM.DECIMAL(DECIMAL(17,2)) ;

-- Function DECIMAL(EURO) returns a DECIMAL type
CREATE FUNCTION SC246300.DECIMAL(EURO) RETURNS SYSIBM.EURO
SOURCE SYSIBM.DECIMAL(DECIMAL(17,2)) ;

-- Function CUSTOMER(CHAR) returns a CUSTOMER type
CREATE FUNCTION SC246300.CUSTOMER(CHAR(11)) RETURNS SC246300.CUSTOMER
SOURCE SYSIBM.CHAR(CHAR(11)) ;

-- Function CHAR(CUSTOMER) returns a PESETA type
CREATE FUNCTION SC246300.CHAR(CUSTOMER) RETURNS SYSIBM.CHAR
SOURCE SYSIBM.CHAR(CHAR(11)) ;

```

CAST functions are also used to convert a data type to use a different length, precision or scale. This is explained in more detail in Section 4.5, “Using CAST functions” on page 48.

4.4 Privileges required to work with UDTs

Now that the distinct types have been defined you can start using them. In Example 4-3, we create a table TBCONTRACTS using the UDTs we created in Example 4-1. The column PESETAFEE has a distinct type of PESETA and column EUROFEE has a distinct type of EURO. The column BUYER uses the CUSTOMER UDT.

Example 4-3 Create table using several UDTs

```

SET CURRENT PATH = 'SC246300';

-- Current Path is needed to find the PESETA
-- and EURO UDTs because the UDTs were created
-- for schema SC246300.
-- You can also use full UDT name SC246300.udt

CREATE TABLE SC246300.TBCONTRACT
(
  SELLER  CHAR (6) NOT NULL ,
  BUYER   CUSTOMER NOT NULL ,
  RECNO   CHAR(15)   NOT NULL ,
  PESETAFEE SC246300.PESETA ,
  PESETACOMM PESETA ,
  EUROFEE SC246300.EURO ,
  EUROCOMM EURO ,
  CONTRDATE DATE,
  CLAUSE  VARCHAR(500) NOT NULL WITH DEFAULT,
  FOREIGN KEY (BUYER) REFERENCES SC246300.TBCUSTOMER,
  FOREIGN KEY (SELLER) REFERENCES SC246300.TBEMPLOYEE
)
IN DB246300.TS246308
WITH RESTRICT ON DROP;

```

In order to be allowed to reference a distinct type in a DDL statement, you need to have the proper authorizations.

The GRANT USAGE ON DISTINCT TYPE statement is used to grant users the privilege to:

- ▶ Use a UDT as a column data type (that is, in a CREATE or ALTER TABLE statement)
- ▶ Use a UDT as a parameter of a stored procedure or user-defined functions (that is, in a CREATE PROCEDURE or CREATE FUNCTION statement)

The GRANT EXECUTE ON statement allows users to use CAST functions on a UDT.

Both the USAGE and EXECUTE privileges can be revoked. In Example 4-4, we show a few typical grants.

Example 4-4 GRANT USAGE/ EXECUTE ON DISTINCT TYPE

```
GRANT USAGE ON DISTINCT TYPE SC246300.EURO TO PUBLIC#  
  
GRANT EXECUTE ON FUNCTION SC246300.EURO(DECIMAL) TO PUBLIC#  
  
GRANT EXECUTE ON FUNCTION SC246300.DECIMAL(EURO) TO PUBLIC#
```

In Example 4-5, we show the use of the DROP and COMMENT ON statements for distinct types. The DATA keyword can be used as a synonym for DISTINCT.

The RESTRICT clause on the DROP statement, must be specified when dropping a distinct type. This clause restricts (prevents) dropping a distinct type if one of the dependencies below exist:

- ▶ A column of a table is defined as this distinct type
- ▶ A parameter or return value of a user-defined function or stored procedure is defined as this distinct type.
- ▶ This distinct type's CAST functions are used in:
 - A view definition
 - A trigger definition
 - A check constraint on CREATE or ALTER table
 - A default clause on CREATE or ALTER table

If the distinct type can be dropped, DB2 also drop the CAST functions that were generated for the distinct type. However, if you have created additional functions to support the distinct type (second bullet above), you have to drop them first before dropping the UDT.

Use COMMENT ON to document what the distinct data type is to be used for.

Example 4-5 DROP and COMMENT ON for UDTs

```
SET CURRENT PATH = 'SC246300';  
  
COMMENT ON {DISTINCT/DATA} TYPE EURO IS string-constant;  
  
DROP {DISTINCT/DATA} TYPE EURO RESTRICT;
```

4.5 Using CAST functions

There are many occasions where a value with a given data type needs to be casted to a different data type or to the same data type with a different length, precision or scale. One example is specifying a comparison in the WHERE clause when a UDT is involved.

Suppose you want to know which contracts are more than 100000 euro. Literals are always considered to be source-type values and therefore the query in Example 4-6 will fail.

Example 4-6 Strong typing and invalid comparisons

```
SET CURRENT PATH = 'SC246300';

SELECT RECNO, BUYER, SELLER
   FROM SC246300.TBCONTRACT
   WHERE EUROFEE > 100000.00;

DSNT408I  SQLCODE = -401, ERROR: THE OPERANDS OF AN ARITHMETIC OR COMPARISON
          OPERATION ARE NOT COMPARABLE
DSNT418I  SQLSTATE = 42818 SQLSTATE RETURN CODE
```

Because you cannot compare data of type EURO (the EUROFEE column is defined as a EURO distinct type) with data of the source type of EURO (that is, DECIMAL) directly, you must use the CAST function to cast data from DECIMAL to EURO. You can also use the DECIMAL function, to cast from EURO to DECIMAL.

Either way you decide to cast, from or to the UDT, you can use:

- ▶ The function name notation, data-type(argument) or
- ▶ The cast notation, CAST(argument AS data-type)

An example of both is shown in Example 4-7.

In fact, the EURO CAST function can be invoked on any data type that can be promoted to the DECIMAL data type by the rules of data type promotion. More details on data type promotion can be found in the *DB2 UDB for OS/390 and z/OS Version 7 SQL Reference, SC26-9944*.

Example 4-7 Two casting methods

```
SET CURRENT PATH = 'SC246300';

SELECT RECNO, BUYER, SELLER
   FROM SC246300.TBCONTRACT
   WHERE EUROFEE > EURO (100000.00)
                                     -- Function name notation

SELECT RECNO, BUYER, SELLER
   FROM SC246300.TBCONTRACT
   WHERE EUROFEE > CAST (100000.00 AS EURO)
                                     -- CAST notation
```

4.6 Operations allowed on distinct types

When a distinct type is created (WITH COMPARISONS), the only operations that are automatically allowed on it are casting between the distinct type and its source type (in both directions), and comparisons between two values of the distinct type (provided the source type is not a large object data type).

The comparison operators that can be used with a distinct data type are shown in Figure 4-1.

Other operators and functions that might apply to the source type, such as arithmetic operators, are not automatically inherited by the distinct type.

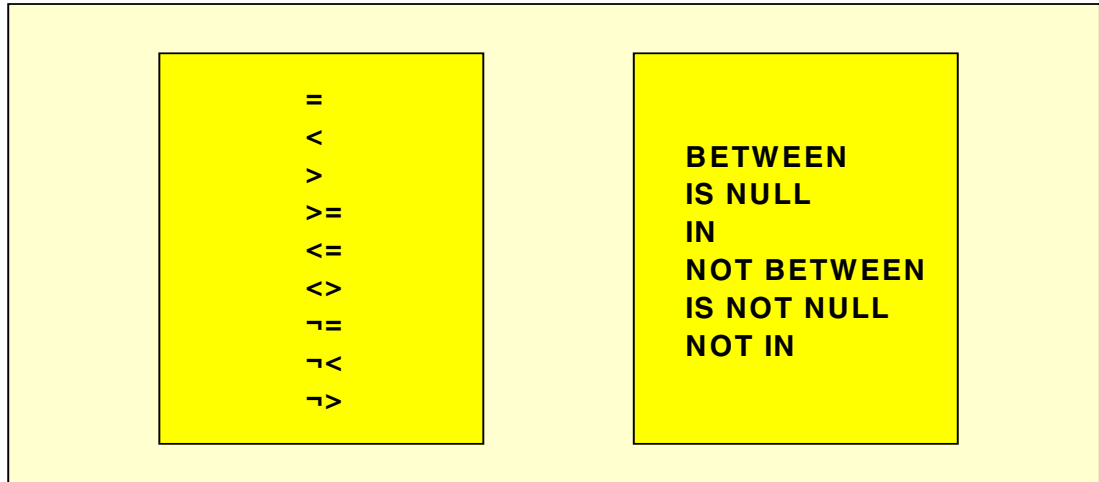


Figure 4-1 Comparison operators allowed on UDTs created WITH COMPARISONS

Note: The LIKE and NOT LIKE comparison operators are not supported for UDTs.

4.6.1 Extending operations allowed in UDTs

DB2 supports 5 built-in operators to build up expressions (CONCAT (or ||), /, *, +, and -). In order to understand the behavior of a distinct type, it is important to realize that DB2 treats these operators as functions that operate on the built-in data types. For example, the expression col1 + col2 can be seen as an invocation of the function "+" (col1,col2). Arithmetic and character operators and built-in functions that apply to the source type are not automatically inherited by the distinct type. Because of this, these operators can only be used by distinct types if they are defined as functions on the distinct types. These operators and functions need to be created explicitly.

Note: It is possible to invoke a function on a UDT instance that is allowed on its source data type even though that function has not been defined on the UDT. To do this, you must first cast the UDT instance to its source data type.

Defining sourced functions on UDTs

The built-in data types come with a collection of built-in functions that operate on them. Some of these functions implement operators such as the arithmetic operators on numeric data types and the concatenate operator on string data types. Other built-in functions include scalar functions, such as LENGTH and SUBSTR, and column functions, such as SUM and AVG.

After creating a distinct type, you can specify that the distinct type inherits some or all of the functions that operate on its source type. This is done by creating new functions, called *sourced functions*, that operate on the distinct type and duplicate the semantics of built-in functions that operate on the source type.

For example, you might specify that your distinct type WEIGHT inherits the arithmetic operators "+" and "-", and the column functions SUM and AVG, from its source type FLOAT. By selectively inheriting the semantics of the source type, you can make sure that programs do not perform operations that make no sense, such as multiplying two weights, even though the underlying source type supports multiplication.

Table SC246300.TBCONTRACT has columns EUROFEE and EUROCOMM defined with distinct type EURO, and columns PESETAFEE and PESETACOMM defined with distinct type PESETA. The first query shown in Example 4-8 is invalid because there is no "+" function defined on distinct type EURO nor PESETA. However, the "+" function can be defined for EURO and PESETA by sourcing it on the built-in "+" function for DECIMAL as shown in the CREATE statement.

Example 4-8 Using sourced functions

```

SET CURRENT PATH = 'SC246300';

SELECT BUYER, SELLER, RECNO, EUROFEE + EUROCOMM, PESETAFEE + PESETACOMM
FROM SC246300.TBCONTRACT
WHERE SELLER LIKE 'A%'
-- This is an invalid query because
-- the sourced function '+' has not
-- been defined on UDT EURO.

CREATE FUNCTION SC246300."+" (EURO,EURO)
RETURNS EURO
SOURCE SYSIBM."+" (DECIMAL(17,2),DECIMAL(17,2))

SELECT BUYER, SELLER, RECNO, EUROFEE + EUROCOMM, PESETAFEE + PESETACOMM
FROM SC246300.TBCONTRACT
WHERE SELLER LIKE 'A%'
-- This is still an invalid query because
-- the sourced function "+" has not
-- been defined on UDT PESETA

CREATE FUNCTION SC246300."+" (PESETA,PESETA)
RETURNS PESETA
SOURCE SYSIBM."+" (DECIMAL(18,0),DECIMAL(18,0));

SELECT BUYER, SELLER, RECNO, EUROFEE + EUROCOMM, PESETAFEE + PESETACOMM
FROM SC246300.TBCONTRACT
WHERE SELLER LIKE 'A%'
-- This is a valid query now that
-- functions "+" has been defined on
-- UDTs EURO and PESETA

```

Example 4-9 uses the SUM and AVG column sourced functions.

Example 4-9 Defining sourced column sourced functions on UDTs

```

SET CURRENT PATH = 'SC246300';

```



```

SELECT SELLER, SUM(PESETAFEE), AVG(PESETAFEE)
  FROM SC246300.TBCONTRACT
  GROUP BY SELLER ;
-- This is an invalid query because
-- the sourced column function SUM has not
-- been defined on UDT PESETA.

CREATE FUNCTION SC246300.SUM(PESETA)
  RETURNS PESETA
  SOURCE SYSIBM.SUM(DECIMAL(18,0)) ;

SELECT SELLER, SUM(PESETAFEE), AVG(PESETAFEE)
  FROM SC246300.TBCONTRACT
  GROUP BY SELLER ;
-- This is still an invalid query because
-- the sourced column function AVG has not
-- been defined on UDT PESETA.

CREATE FUNCTION SC246300.AVG(PESETA)
  RETURNS PESETA
  SOURCE SYSIBM.AVG(DECIMAL(18,0)) ;

SELECT SELLER, SUM(PESETAFEE), AVG(PESETAFEE)
  FROM SC246300.TBCONTRACT
  GROUP BY SELLER ;
-- This is now a valid query

```

Defining external functions on UDTs

You can also go beyond mere inheritance of source-type functions and give your distinct type semantics of its own. This is done by creating external functions, written in a host programming language, that operate on your distinct type.

The two columns in Example 4-10 cannot be directly compared and the query fails with an SQLCODE -401. This prevents you from making the mistake of directly comparing or performing arithmetic operations with columns of different currency types.

Example 4-10 Strong typing and invalid comparisons

```

SET CURRENT PATH = 'SC246300';

SELECT CUSTOMERNO, BUYER
  FROM SC246301.TBCONTRACT
  WHERE PESETAFEE > EUROFEE;

DSNT408I  SQLCODE = -401, ERROR: THE OPERANDS OF AN ARITHMETIC OR COMPARISON
         OPERATION ARE NOT COMPARABLE
DSNT418I  SQLSTATE = 42818 SQLSTATE RETURN CODE

```

In order to directly compare or perform arithmetic operations with columns of different currency types, it is necessary to first cast the column(s) to a common data type. In Example 4-11, show how the WHERE clause of the query in Example 4-10 can be re-coded in order to be able to compare pesetas to euros. The example converts EUROFEE to its

source data type (DECIMAL) using the automatically generated CAST function DECIMAL, then multiplies it by 166 (the current monetary conversion factor to convert from euros to pesetas), and finally this result is casted to pesetas with the PESETA function and compared with the column PESETAFEE which is a PESETA distinct type column.

Example 4-11 Comparing pesetas and euros

```
SET CURRENT PATH = 'SC246300';

SELECT CUSTOMERNO, BUYER
   FROM SC246301.TBCONTRACT
   WHERE PESETAFEE = PESETA((DECIMAL(EUROFEE))*166)
```

Example 4-12 accomplished the same thing than Example 4-11, but we have placed the conversion factor (multiplication by 166) into a user-defined function called EUR2PES. For more information on user-defined functions see Chapter 5, "User-defined functions (UDF)" on page 57. In the additional material you can find an external UDF (EUR22PES) using a Cobol program to implement the same functionality. See Appendix C, "Additional material" on page 251 for details.

Example 4-12 Another way to compare pesetas and euros

```
SET CURRENT PATH = 'SC246300';

CREATE FUNCTION SC246300.EUR2PES (X DECIMAL)
  RETURNS DECIMAL
  LANGUAGE SQL
  CONTAINS SQL
  NO EXTERNAL ACTION
  NOT DETERMINISTIC
  RETURN X*166 ;

-- DB2 V7 allows to create functions
-- with LANGUAGE SQL, so it is not
-- necessary to use external code.

SELECT CUSTOMERNO, BUYER
   FROM SC246301.TBCONTRACT
   WHERE PESETAFEE = PESETA(EUR2PES(DECIMAL(EUROFEE)))
```

You can also code the UDF the following way:

```
CREATE FUNCTION SC246300.EUR22PES (X EURO)
  RETURNS PESETA
  LANGUAGE SQL
  CONTAINS SQL
  NO EXTERNAL ACTION
  NOT DETERMINISTIC
  RETURN PESETA (DECIMAL(X) * 166) #
```

And refer to it as:

```
SELECT CUSTOMERNO, BUYER
   FROM SC246301.TBCONTRACT
   WHERE PESETAFEE = EUR2PES(EUROFEE)
```

Note: The EUR2PES function to convert euros to pesetas that is used here is just for illustration purposes. The actual conversion formula is somewhat more complicated and is probably best implemented through an external function using a regular programming language like C, Cobol or PL/I.

Example 4-13 shows the use of the CAST functions EURO(DECIMAL) and DECIMAL(PESETA) in a trigger. The trigger automatically supplies a value for the EUROFEE column with the amount in euros when a contract is inserted in table SC246300.TBCONTRACT with the fee in pesetas (the insert includes the PESETAFEE column).

Example 4-13 Automatic conversion of euros

```
SET CURRENT PATH = 'SC246300';

CREATE TRIGGER SC246300.TGEURMOD
NO CASCADE BEFORE
INSERT ON SC246300.TBCONTRACT
REFERENCING NEW AS N
FOR EACH ROW MODE DB2SQL
SET N.EUROFEE = EURO(DECIMAL(N.PESETAFEE)/166)
```

The use of this sort of trigger can be interesting during a conversion. Programs that have not been converted can continue to INSERT values in pesetas into the PESETAFEE column, whereas new programs can already use the new EUROFEE column. This way you don't have to change all the programs in one big operation, but have a more gradual migration.

Note: You probably need to implement an UPDATE trigger with similar functionality when there is a process that can updated the PESETAFEE column.

4.7 Usage considerations

In this section we focus on some specific areas related to using distinct data types.

4.7.1 UDTs in host language programs

Application programmers need to understand that CAST functions might be required when constants and host variables are used in SQL statements to specify instances of distinct data types. Application programmers also need to understand how to correctly define host variables such that they can be used in assignment operations that involve distinct data types. (An assignment operation is the process of giving a new value to a column in a table or to a host variable. Assignment operations can be performed during the execution of INSERT, UPDATE, FETCH, SELECT INTO and SET statements and during function invocation.)

We recommend that you put the source type of a column in the DECLARE TABLE statement instead of the distinct data type name. This enables the precompiler to check the embedded SQL, otherwise checking is deferred until bind time.

Note: DCLGEN generates a DECLARE TABLE statement that refers to the source data type and not the distinct type.

A host variable is compatible with a distinct type if the host variable type is compatible with the source type of the distinct type. You can assign a column value of a distinct type to a host variable if you can assign a column value of the distinct type's source type to the host variable. In other words you should use the same definition for your host variables when referring to a UDT than you would use when referring to its source data type.

If for example, a Cobol program needs to reference a distinct type named CUSTOMER that is based on a CHAR(15) built-in data type, you should define the host variable as PIC X(15).

4.7.2 Using the LIKE comparison with UDTs

As mentioned in 4.6, "Operations allowed on distinct types" on page 49, you cannot use the LIKE comparison operator on a distinct type. In order to be able to use the LIKE operator on a column that is defined as a UDT, you have to cast it to CHAR data type. This is shown in Example 4-14.

Example 4-14 Using LIKE on a UDT

```
SET CURRENT PATH = 'SC246300';

SELECT RECNO, BUYER, SELLER
FROM SC246300.TBCONTRACT
WHERE BUYER LIKE 'A%';

-- This results in:
--DSNT408I SQLCODE = -414, ERROR: A LIKE PREDICATE IS INVALID BECAUSE THE FIRST
--                                OPERAND IS NOT A STRING
--DSNT418I SQLSTATE = 42824 SQLSTATE RETURN CODE

-- You have to convert the BUYER column into a type that allows for LIKE comparison

SELECT RECNO, BUYER, SELLER
FROM SC246300.TBCONTRACT
WHERE CHAR(BUYER) LIKE 'A%';

-- or

SELECT RECNO, BUYER, SELLER
FROM SC246300.TBCONTRACT
WHERE CAST(BUYER AS CHAR) LIKE 'A%';
```

4.7.3 UDTs and utilities

Utilities don't understand distinct types. You should use the built-in data type (or its external representation equivalent) when referring to a UDT in a DB2 utility statement. When loading table SC246300.TBCONTRACT as shown in Example 4-15, you do not specify the UDTs. If you try, the LOAD utility would fail.

Example 4-15 Loading a table with a UDT

```
TEMPLATE U6830982
DSN(BART.&DB..&TS..UNLOAD)
```

```

        DISP(OLD,CATLG,CATLG)
LOAD DATA INDDN U6830982 LOG NO RESUME YES
EBCDIC CCSID(00037,00000,00000)
INTO TABLE "SC246300"."TBCONTRACT"
WHEN(00001:00002 = X'0041')
( "SELLER          " POSITION( 00003:00008) CHAR(006)
, "BUYER           " POSITION( 00009:00019) CHAR(011)
, "RECNO          " POSITION( 00020:00034) CHAR(015)
, "PESETAFEE      " POSITION( 00036:00045) DECIMAL
                    NULLIF(00035)=X'FF'
, "PESETACOMM     " POSITION( 00047:00056) DECIMAL
                    NULLIF(00046)=X'FF'
, "EUROFEE        " POSITION( 00058:00066) DECIMAL
                    NULLIF(00057)=X'FF'
, "EUROCOMM       " POSITION( 00068:00076) DECIMAL
                    NULLIF(00067)=X'FF'
, "CONTRDATE      " POSITION( 00078:00087) DATE EXTERNAL
                    NULLIF(00077)=X'FF'
, "CLAUSE         " POSITION( 00088)          VARCHAR
)

```

-- To enable you to compare, we show the table definition of TBCONTRACT hereafter

```

CREATE TABLE SC246300.TBCONTRACT
(
SELLER CHAR ( 6 ) NOT NULL ,
BUYER  CUSTOMER NOT NULL ,
RECNO  CHAR(15) NOT NULL ,
PESETAFEE SC246300.PESETA ,
PESETACOMM PESETA ,
EUROFEE SC246300.EURO ,
EUROCOMM EURO ,
CONTRDATE DATE,
CLAUSE VARCHAR(500) NOT NULL WITH DEFAULT,
FOREIGN KEY (BUYER) REFERENCES SC246300.TBCUSTOMER,
FOREIGN KEY (SELLER) REFERENCES SC246300.TBEMPLOYEE
                    ON DELETE CASCADE
)
IN DB246300.TS246308
WITH RESTRICT ON DROP;

```

4.7.4 Implementing UDTs in an existing environment

User distinct types can bring a lot of benefits to your organization especially the strong typing that goes hand in hand with UDTs.

When implementing distinct types in an existing environment careful planning is required. When you decide to change the column of an existing table to a user-defined distinct type, you have to make sure that all programs and ad-hoc queries that reference that column are changed accordingly (using the proper CAST functions and UDFs). You also have to make sure that if this column is part of an RI structure, that the other tables referencing the column you are trying to change to a UDT, are also changed. For more on RI and UDTs, see 4.7.5, “Miscellaneous considerations” on page 56.

4.7.5 Miscellaneous considerations

An application requestor can only issue SQL statements that reference columns defined with a distinct data type if the DRDA protocol is used.

You can define RI relationships on columns that are defined with a distinct type. However, both the parent's primary key column(s) must have the same data (distinct) type as the foreign key's. For example, a foreign key on a column (BUYER) that is defined with a data type of CUSTOMER in TBCONTRACT, can reference the primary key column (CUSTID) in the TBCUSTOMER table, because the primary key is also defined as a CUSTOMER distinct type. If they don't have a matching data type, you receive the following SQL error:

```
SQLCODE = -538, ERROR: FOREIGN KEY BUYERFK DOES NOT CONFORM TO THE DESCRIPTION OF A
PARENT KEY OF TABLE SC246300.TBCUSTOMER
SQLSTATE = 42830 SQLSTATE RETURN CODE
```

You cannot create a declared temporary table that contains a user distinct type. It is not supported and you receive the following error:

```
DSNT408I SQLCODE = -607, ERROR: OPERATION OR OPTION USER DEFINED DATA TYPE IS NOT
DEFINED FOR THIS OBJECT
DSNT418I SQLSTATE = 42832 SQLSTATE RETURN CODE
```

A field procedure can be defined on a distinct data type column. The source type of the distinct data type must be a short string column that has a null default value. When the field procedure is invoked, the value of the column is casted to its source type and then passed to the field procedure.

Also be aware that are indexable predicates:

```
WHERE BUYER = CAST ('ANNE' AS CUSTOMER) and
WHERE CHAR(BUYER) = 'ANNE'#
```

4.8 UDTs in the catalog

In Table 4-1, we see the changes made to the DB2 catalog to support UDTs.

Table 4-1 Catalog changes to support UDTs

Catalog Table	Contents
SYSIBM.SYSDATATYPES	one row for each distinct type
SYSIBM.SYSROUTINES	one row for each CAST function
SYSIBM.SYSROUTINEAUTH	records privileges held by users on CAST functions
SYSIBM.SYSRESAUTH	new value added to OBTYPE for distinct type USAGE privilege



User-defined functions (UDF)

The number of built-in functions increased considerably in Version 6 and Version 7 over previous releases. There are now over 90 different functions that perform a wide range of string, date, time, and timestamp manipulations, data type conversions, and arithmetic calculations.

In some cases even this large number of built-in functions does not fit all needs. Therefore, DB2 allows you to write your own user-defined functions (UDF) that call an external program. This extends the functionality of SQL to whatever you can code in an application program; essentially, there are no limits.

In this section, we discuss the different sorts of user-defined functions and how to use them.

5.1 Terminology overview

Before we head out to explore user-defined functions we first discuss some general characteristics and types of functions available in DB2 for z/OS.

There are different ways to categorize functions. You can classify them as:

Built-in functions	Built-in functions are so-called because they are built into DB2's system code. Examples of built-in functions are CHAR and AVG. These functions now reside in the SYSIBM schema. For more details on built-in functions see Chapter 6, "Built-in functions" on page 71.
User-defined functions	UDFs allow you to write your own functions for the usage in SQL statements. They reside in the schema you create them in.

Another way to categorize them is by the type of arguments they use as input and the number of arguments they return as output:

Scalar functions	A scalar function is an SQL operation that returns a single value from another value or set of values, and is expressed as a function name, followed by a list of arguments that are enclosed in parentheses. Each argument of a scalar function is a single value. Examples of scalar functions are CHAR, DATE, and SUBSTR.
Column functions	A column function is an SQL operation that produces a single value from the values of a single column (or a subset thereof). As with scalar functions, column functions also return a single value. However, the argument of a column function is a set of like values. Examples of column functions are: AVG, COUNT, MAX, MIN and SUM.
Table functions	A table function is a function that returns a table to the SQL statement that references it. A table function can be referenced only in the FROM clause of a SELECT statement. In general, the returned table can be referenced in exactly the same way as any other table. Table functions are useful for performing SQL operations on non-DB2 data or moving non-DB2 data into a DB2 table.
Arithmetic and string operators	These are the traditional operators that are allowed on columns (depending on their data type). They can also be thought of as functions. Arithmetic and string operators are: "+", "-", "*", "/", CONCAT and " ".

When discussing user-defined functions, you can classify those as:

External	Are based on programs written by you, and may be written in any of the programming languages supported by the target database management system. DB2 V7 even allows you to build external functions using SQL. These are SQL functions. You can only define scalar functions this way.
Internal/sourced	Are based on existing built-in functions or existing user-defined functions that are already known to DB2. Their primary purpose is to extend existing functions (for example, the AVG function or the LENGTH function) for the

source data type to a newly created user-defined distinct type.

The following Table 5-1 shows the different combinations of function types that are allowed:

Table 5-1 Allowable combinations of function types

		Scalar	Column	Table	Arithmetic
Built-in function		OK	OK	N/A	OK
UDF	External	OK	N/A	OK	N/A
	Sourced	OK	OK	N/A	OK

5.2 Definition of a UDF

User-Defined Functions (UDFs) allow you to write your own functions for the usage in SQL statements. The user-defined functions provided by you can be used in Data Manipulation Language (DML) statements or Data Definition Language (DDL) statements. When creating a UDF, you can either create your own source code (external function) or use another function as the source (sourced function).

When writing external UDFs, the functions have to follow certain conventions concerning the passing and returning of arguments, but you can do pretty much what you want. If this program contains SQL statements then there is an associated package that contains the program's bound SQL statements.

When you source a UDF on another function, that function can be either a built-in function or another UDF.

5.3 The need for user-defined functions

A user-defined function is a mechanism by which you can write your own extensions to the SQL language. The built-in functions supplied with DB2 are a useful set of functions, but they might not satisfy all of your requirements. You might need to extend the SQL language for the following reasons:

- ▶ Customization - The function specific to your application does not exist in DB2. Whether the function is a simple transformation, a trivial calculation, or a complicated analysis, you can probably use a UDF to do the job.
- ▶ Flexibility - The DB2 built-in function does not quite permit the variations that you wish to include in your application.
- ▶ Standardization - Many of the programs at your site implement the same basic set of functions, but there are minor differences in all the implementations. Thus, you are unsure about the consistency of the results you receive. If you correctly implement these functions once, in a UDF, then all these programs can use the same implementation directly in SQL and provide consistent results.
- ▶ Object-relational support - UDTs can be very useful in extending the capability and enforcing consistent use of the data in your DB2 system. UDFs act as the methods for UDTs by providing consistent behavior and encapsulating the types. More information can be found in Chapter 4, "User-defined distinct types (UDT)" on page 43.
- ▶ Migration from other DBMS systems - When you are migrating from another DBMS to DB2 it is not unlikely that the DBMS you are migrating from has some functions built into it that

DB2 does not have or have a different name. You can make up for this by creating your own UDF to implement the function from the other DBMS in your DB2 system.

- ▶ UDFs and UDTs can also be exploited by software developers who write specialized applications. The software developer can provide UDTs and UDFs as part of their software package. This approach is used in the family of DB2 Extender products.
- ▶ Simplifying SQL syntax - With a UDF you can encapsulate the logic of having to write a complex expression into a UDF. Replacing a complex expression by a UDF improves readability of the SQL statement. It can also avoid coding errors as you can easily make a mistake when repeatedly coding the same complex expressions.

5.4 Implementation and maintenance of UDFs

User-defined functions are extensions or additions to the built-in functions of the SQL language. UDFs are defined to DB2 using the CREATE FUNCTION statement. A user-defined function (external or sourced) can be either a:

- ▶ Scalar function
- ▶ Column function
- ▶ Table function

Tip: External functions can be either scalar functions or table functions, they cannot be column functions.

You can overload functions. You can define multiple functions with the same name as long as the signatures of the various functions are different. This means that the data type of at least one parameter or the number of parameters must be different. Based on the data types of the arguments passed, the database management system is capable of selecting the proper function.

A user-defined function is invoked by specifying its function name followed by parentheses enclosing the input arguments to the function; *function-name(argument-1, argument-2, ..)*.

5.4.1 Scalar functions

A scalar function is an SQL operation that produces a single value from another value or set of values and is expressed as a function name, followed by a list of arguments that are enclosed in parentheses

The number of different business applications using DB2 as a database manager is wide and varied, and the number continues to grow. This diverse usage places demands on the database to support a wide variety of scalar functions.

Example 5-1 creates an SQL scalar function (available in V7) that returns size of the surface area of a circle based on its radius.

Example 5-1 Example of an SQL scalar function

```
CREATE FUNCTION AREA_CIRCLE (X DOUBLE)
  RETURNS DOUBLE
  LANGUAGE SQL
  CONTAINS SQL
  NO EXTERNAL ACTION
  RETURN X*X* 3.1415926 #
SELECT AREA_CIRCLE (3) FROM SYSIBM.SYSDUMMY1 # -- Test
```

+0.2827433340000000E+02

Another example of an SQL scalar UDF can be found in Example 4-12 on page 52.

Example 5-2 demonstrates the definition and use of a user-defined scalar function called CALC_BONUS. The CALC_BONUS function is defined to DB2 by using the CREATE FUNCTION statement which associates the function with a user-written C program called CBONUS. The function calculates the bonus for each employee based upon the employee's salary and commission. The function takes two input parameters (both DECIMAL(9,2)) and returns a result of DECIMAL(9,2). Note that a scalar function can only return one value. The external name 'CBONUS' identifies the name of the load module containing the code for the function.

A user-defined scalar function can be referenced in the same context that built-in functions can be referenced; that is, wherever an expression can be used in a SELECT, INSERT, UPDATE or DELETE statement. User-defined scalar functions can also be referenced in CALL, VALUES and SET statements. The UPDATE statement in Example 5-2 shows how to use the CALC_BONUS function.

Example 5-2 User-defined external scalar function

```
CREATE FUNCTION CALC_BONUS (DECIMAL(9,2),DECIMAL(9,2))
    RETURNS DECIMAL(9,2)
    EXTERNAL NAME 'CBONUS'
    LANGUAGE C
```

```
Function Program pseudo code:
    cbonus (salary,comm,bonus)
        bonus=(salary+comm)*.10
    return
```

```
UPDATE SC246300.EMPLOYEE
    SET BONUS = CALC_BONUS (SALARY,COMM)
```

5.4.2 Column functions

A column function is an SQL operation that uses the values of a single column (or a subset thereof from the result set of an SQL statement) and returns a single value which generally is derived from the values of the column.

Example 5-3 shows the creation of a column function used to be able to calculate the minimum value in a PESETA UDT column.

Example 5-3 Sourced column function

```
SET CURRENT PATH = 'SC246300';

CREATE FUNCTION SC246300.MIN(PESETA)
    RETURNS PESETA
    SOURCE SYSIBM.MIN(DECIMAL(18,0)) ;
```

See also Example 4-9 on page 50.

User-defined column functions can be referenced wherever a built-in column function can be used in a SELECT, INSERT, UPDATE or DELETE statement. The ALL or DISTINCT keywords can only be specified for a built-in or user-defined column function. Transition tables cannot be passed to user-defined column functions.

User-defined column functions can only be created based on a DB2 built-in column function; you cannot create your own programs to do that.

5.4.3 Table functions

A user-defined table function is a function that returns a table to the SQL statement that references it. A user-defined table function can be referenced only in the FROM clause of a SELECT statement. In general, the returned table can be referenced in exactly the same way as any other table. Table functions are useful for performing SQL operations on non-DB2 data or moving non-DB2 data into a DB2 table.

Example 5-4 demonstrates the definition and use of a user-defined table function called EXCH_RATES. This function is written in Cobol and returns exchange rate information for various currencies. The function takes no input parameters but returns a table of 3 columns. The external name 'EXCHRATE' identifies the name of the load module that contains the code for the function. The SELECT statement in the example shows how the EXCH_RATES function is invoked. In the additional material you can find sample coding showing how to implement this table UDF. See Appendix C, "Additional material" on page 251 for details.

Table UDFs can be helpful when the actual data that is returned in the table is coming from an external resource (non-DB2 resource). In our example, currency exchange rates change constantly and our financial analyst wants to have the latest numbers when simulating his investment model. Since his model uses ad-hoc queries instead of regular programs, we cannot use a stored procedure to obtain the information, because you cannot use a CALL statement outside a program. By using a table UDF, you can provide a result table to the user. Inside the program that is actually invoked by the UDF, you code pretty much the same things that you would have when using a stored procedure. To keep our example simple, we obtain the information from a sequential file. In real life you will probably connect to an external system to obtain the exchange rate information.

Example 5-4 User-defined table function

```
CREATE FUNCTION
  SC246300.EXCH_RATES()
  RETURNS
    TABLE(
      CURRENCY_FROM CHAR(30) CCSID EBCDIC,
      CURRENCY_TO   CHAR(30) CCSID EBCDIC,
      EXCHANGE_RATE DECIMAL (12,4)
    )
  LANGUAGE COBOL
  NOT DETERMINISTIC
  NO SQL
  EXTERNAL NAME EXCHRATE
  PARAMETER STYLE DB2SQL
  CALLED ON NULL INPUT
  EXTERNAL ACTION
  SCRATCHPAD
  FINAL CALL
  DISALLOW PARALLEL
  NO COLLID
  ASUTIME LIMIT 5
```

```

STAY RESIDENT NO
PROGRAM TYPE SUB
WLM ENVIRONMENT V7PERF
SECURITY DB2
NO DBINFO ;

```

```

SELECT *
  FROM TABLE( SC246300.EXCH_RATES()) AS X
 WHERE CURRENCY_FROM = 'USD' ;

```

CURRENCY_FROM	CURRENCY_TO	EXCHANGE_RATE
USD	EURO	1.0970
USD	FF	7.1955
USD	BEF	44.6730

When considering table UDFs there are a few additional things that you have to know. Table UDFs do not use the same mechanism to pass information back and forth as triggers do when passing transition tables to for instance a stored procedure. There are no table locator variables used when dealing with table UDFs.

The program that is invoking the table UDF (SPUFI in the example above) does its normal SQL processing as with any other 'regular' table. It will OPEN the cursor, FETCH rows from it and CLOSE the cursor. (We are ignoring here the additional calls that can take place with dynamic SQL like PREPARE and DESCRIBE.) When executing OPEN, FETCH and CLOSE calls, a trip is made to the UDF program that executes in a WLM address space. On each trip to the UDF program, a 'CALLTYPE' parameter is passed to the (Cobol) program that is invoked. The program uses this information to do decide what part of the code to execute. If you are using the FINAL CALL keyword in the CREATE FUNCTION statement a couple of extra calls (with a special CALLTYPE) are executed. Following is a list of possible CALLTYPES that are used with table UDFs. For more information, see section "Passing parameter values to and from a user-defined function" in the *DB2 UDB for OS/390 and z/OS Version 7 Application Programming and SQL Guide*, SC26-9933.

- 2 This is the first call to the user-defined function for the SQL statement. This type of call occurs only if the FINAL CALL keyword is specified in the user-defined function definition.
- 1 This is the OPEN call to the user-defined function by an SQL statement.
- 0 This is a FETCH call to the user-defined function by an SQL statement. For a FETCH call, all input parameters are passed to the user-defined function. If a scratchpad is also passed, DB2 does not modify it. You will have multiple calls of this type, basically one for every row you want to pass back to the caller to end up in the result table.
- 1 This is a CLOSE call.
- 2 This is a final call. This type of final call occurs only if FINAL CALL is specified in the user-defined function definition.
- 255 This is another type of final call. This type of final call occurs when the invoking application executes a COMMIT or ROLLBACK statement, or when the invoking application abnormally terminates. When a value of 255 is passed to the user-defined function, the user-defined function

cannot execute any SQL statements, except for CLOSE CURSOR. If the user-defined function executes any close cursor statements during this type of final call, the user-defined function should tolerate SQLCODE -501 because DB2 might have already closed cursors before the final call.

When you want to signal to the calling application (SPUFI in our example) that you have finished passing rows (reached the end of the sequential file in our case), you set the SQLSTATE variable to '02000' before returning to the invoker.

When you want to preserve information between subsequent invocations of the same table UDF (for instance when processing/reading the sequential file - CALLTYEP='0') you can use a scratchpad to store that information. In our example there is no real need to do so. The CREATE FUNCTION does specify the SCRATCHPAD keyword because to debug the code, it was interesting to keep a counter to track the number of invocations of the table UDF to build the result table.

The filtering from the WHERE clause is done by DB2 not by the table UDF's program. The WHERE clause information is not passed to the program. So when the amount of information that you pass back to the invoker is large (big sequential file) and almost all of the rows are filtered out by a WHERE clause, you can end up using more resources than you might expect based on the number of rows that actually show up in the result set.

You must code a user-defined table function that accesses external resources as a subprogram. Also ensure that the definer specifies the EXTERNAL ACTION parameter in the CREATE FUNCTION or ALTER FUNCTION statement. Program variables for a subprogram persist between invocations of the user-defined function, and use of the EXTERNAL ACTION parameter ensures that the user-defined function stays in the same address space from one invocation to another.

5.5 UDF design considerations

User-defined functions are part of the SQL99 standard. Be aware that they are fairly new and not (yet) supported by all database management systems when your installation has cross-platform requirements.

Just as there are techniques to ensure efficient access paths using SQL, there are ways you can maximize the efficiency and reduce the costs of UDFs.

5.5.1 Maximizing UDF efficiency

The difference between the cost of DB2's built-in functions and a user-defined function can be understood when you know that the UDF is fenced, by definition. This means that a UDF does not execute within the DB2 address spaces, to protect the integrity of DB2 from application code errors. Your external UDF executes under Language Environment (LE) control in a WLM address space. Conversely, DB2 built-in functions are a component of the data base engine. Therefore, overhead is necessarily associated with external UDFs.

However, there are several ways you can improve the efficiency of external UDFs:

- ▶ You should try to avoid the cost of having to create a WLM address space and re-use an existing WLM address space. This may not always be possible, though, if you have a requirement to isolate different workloads and applications.

- ▶ If you can, code your load module as re-entrant. This allows you to override the default NO of the STAY RESIDENT option of the CREATE FUNCTION statement. If you specify YES:
 - The load module remains in storage after it has been loaded.
 - This single copy of the module can then be shared across multiple invocations of the UDF.

The impact of STAY RESIDENT YES is very important if multiple instances of a UDF are specified in the same SQL statement.

- ▶ There is overhead processing for each input parameter, so keep the number to the minimum required.
- ▶ Remember that, just as with built-in functions, or with any change to your application, the access path chosen by DB2 can be affected by an external UDF. A statement that is indexable without the function may become non-indexable by adding an improperly coded function. There are two obvious cases in which the statement can become non-indexable:
 - The UDF is returning a CHAR value with a length different from the one that it is compared to.
 - The UDF is returning a nullable result and the compared value is not nullable.

We strongly recommend that you use EXPLAIN to determine whether the access path is what you expect, and whether it is as efficient as it can be. If you think the UDF is preventing DB2 from choosing an efficient access path, experiment by coding the statement with and without the UDF. This helps you understand the impact of the UDF on the access path.

UDFs have been fully integrated into the SQL language. This means that the UDF call can appear anywhere in the statement. Also, a single SQL statement can often be written in different ways and still achieve the same result. Use this to:

- Ensure that the access path is efficient.
- Code the SQL statement such that the UDF processes the fewest rows. This reduces the cost of the statement.
- ▶ Exploit the fact that the LE architecture makes processing subroutines more efficient than main programs by defining the UDF program type as SUB.
- ▶ It is evident that you should make your UDF application code as efficient as possible. Two frequently overlooked opportunities to maximize efficiency are:
 - Ensure that all variable types match. This ensures that additional overhead is not incurred within LE, performing unnecessary conversion.
 - In the C programming language, ensure that pragmas are coded correctly.
- ▶ Since the cost of DB2 built-in functions is low, exploit them wherever possible.

5.5.2 Consider sourced functions

When you want to determine the most efficient way to code your function, consider whether you can source your function based on one of the built-in DB2 functions.

For example, if you need to translate a SMALLINT data type to a CHARACTER for some subsequent string-based manipulation, you have several options, depending on your precise requirements:

- ▶ Write an external UDF.

This may appear as a highly attractive option if, for example, you are converting from another database management system to DB2. The application might extensively use a function that has a different name in the other DBMS, or behaves slightly differently from DB2's version of the same function. Suppose, for example, the function used by the application to convert SMALLINT data to a string is called CHARNSI (see Example 5-9 on page 67 for sample code). There is no function in DB2 with this name. To reduce the need to alter application code, you could code your own external UDF in a host language. The application can then run without any change and invoke your UDF.

The CREATE FUNCTION SQL necessary to define it to DB2 can be found in Example 5-5.

Example 5-5 External UDF to convert from SMALLINT to VARCHAR

```
CREATE FUNCTION SC246300.CHAR_N_SI (SMALLINT )
RETURNS VARCHAR(32)
SPECIFIC CHAR_N_SI
LANGUAGE C
DETERMINISTIC
NO SQL
EXTERNAL NAME CHARNSI
PARAMETER STYLE DB2SQL
NULL CALL
NO EXTERNAL ACTION
NO SCRATCHPAD
NO FINAL CALL
ALLOW PARALLEL
NO COLLID
ASUTIME LIMIT 5
STAY RESIDENT YES
PROGRAM TYPE SUB
WLM ENVIRONMENT V7PERF
SECURITY DB2
NO DBINFO;

SET CURRENT PATH = 'SC246300';

SELECT CHAR_N_SI(SMALLINT(4899))
FROM SYSIBM.SYSDUMMY1;

4899 -- Result

-- To show it is a CHAR string now use a SUBSTR function on it
SELECT SUBSTR(CHAR_N_SI(SMALLINT(4899)),1,2)
FROM SYSIBM.SYSDUMMY1 #

48 -- Result
```

► Create a sourced UDF.

Since a sourced UDF is based on an internal DB2 built-in function, you can expect comparable performance. There is no call to LE, and the UDF does not need to execute under a WLM environment. In Example 5-6, we show how you can code the sourced UDF. It can be called CHARNSI, which would satisfy your requirement that the application could be readily converted to DB2.

Example 5-6 Creating a sourced UDF

```
CREATE FUNCTION CHARNSI(DECIMAL(6,0)) RETURNS VARCHAR(32)
SOURCE SYSIBM.SMALINT(DECIMAL(6,0));
```



```

SELECT CHARNSI(4899) FROM SYSIBM.SYSDUMMY1;

489 -- Result

SELECT SUBSTR(CHARNSI(4899),1,2) FROM SYSIBM.SYSDUMMY1;

48 -- Result

```

-
- Use the CAST function or use DB2 built-in functions.

The CAST function is illustrated in Example 5-7, the use of a DB2 built-in function is shown in Example 5-8. You can expect good and comparable performance from both. The disadvantage, if you are converting from another database management system, is that application code needs to be changed.

If you need to change application code anyway or choose to do it for other reasons, then we recommend switching to DB2 built-in functions.

Example 5-7 Using CAST instead of a UDF

```

SELECT SUBSTR(CAST(4899 AS CHAR(6)),1,2) FROM SYSIBM.SYSDUMMY1;

48 -- Result

```

Example 5-8 Built-in function instead of a UDF

```

SELECT SUBSTR(CHAR(4899),1,2) FROM SYSIBM.SYSDUMMY1;

48 -- Result

```

Example 5-9 CHARNSI source code

```

C program listing
/*****
* Module name = CHARNSI
*
* DESCRIPTIVE NAME = Convert small integer number to a string
*
*
* LICENSED MATERIALS - PROPERTY OF IBM
* 5645-DB2
* (C) COPYRIGHT 1999 IBM CORP. ALL RIGHTS RESERVED.
*
* STATUS = VERSION 6
*
*
* Example invocations:
* 1) EXEC SQL SET :String = CHARN(number) ;
* ==> converts the small integer number to a string
* Notes:
* Dependencies: Requires IBM C/C++ for OS/390 V1R3 or higher
*
* Restrictions:
*
* Module type: C program
* Processor: IBM C/C++ for OS/390 V1R3 or higher
* Module size: See linkedit output
*****/

```



```

#define NULLCHAR '\0'
/***** GREATN functions *****/
void CHARNSI /* main routine */
(
short int *p1In, /* First parameter address */
char *pOut, /* Output address */
short int *null1In, /* in: indic var for null1In */
short int *nullpOut, /* out: indic var for pOut */
char *sqlstate, /* out: SQLSTATE */
char *fnName, /* in: family name of function*/
char *specificName, /* in: specific name of func */
char *message /* out: diagnostic message */
);
/***** main routine *****/
/***** main routine *****/
void CHARNSI /* main routine */
(
short int *p1In, /* in: timestp1 */
char *pOut, /* out: timestp */
short int *null1In, /* in: indic var for null1In */
short int *nullpOut, /* out: indic var for pOut */
char *sqlstate, /* out: SQLSTATE */
char *fnName, /* in: family name of function*/
char *specificName, /* in: specific name of func */
char *message /* out: diagnostic message */
)
{
#define DEF_OUTPUT_LENGTH 32

/***** Local variables *****/
char stret??( 100 ??); /* string reciever */
/*****
* Initialize SQLSTATE to 00000 and MESSAGE to ""
*****/
message[0] = NULLCHAR;
*nullpOut = 0; /* -1 if Null value returned */
memcpy( sqlstate,"00000",6 );
memset(pOut, NULLCHAR, DEF_OUTPUT_LENGTH);

/*****
* Return NULL if at least one input parameter is NULL
*****/
if (*null1In != 0)
{
*nullpOut = -1;
return;
}

/*****
* Convert an integer to a string
*****/

sprintf( pOut, "%-d", *p1In);

return;
} /* end of CHARNSI */

```




Built-in functions

In the last couple of versions, DB2 has expanded the number of built-in functions dramatically.

In this chapter, we give an overview of the built-in functions that were added to DB2 in versions 6 and 7 and briefly describe their characteristics.

6.1 What is a built-in function?

A built-in function is a function that is supplied with DB2. The function has a name and zero, one or more arguments, enclosed in parenthesis. The result of a built-in function is a single value (there are no built-in table functions). Built-in functions are classified as column functions or scalar functions. Comparison operators as well as arithmetic and string operators can also be regarded as built-in functions.

Built-in functions are part of the SYSIBM schema.

6.2 Why use a built-in function

Built-in functions can be used as sources for user-defined functions which you need to create for *user-defined distinct types*. User-defined distinct types do not automatically inherit all functions allowed on its source data type and they have to be explicitly created.

There are a large number of DB2 built-in functions with rich functionality which perform very well. Therefore, before you start coding your own functions, evaluate what is supplied with DB2 and understand how to use it. This allows you to:

- ▶ Maximize the efficiency of your application. Consider here not just the cost of executing your external function compared to DB2's built-in functions, but also the best access path that can be achieved with a UDF as compared to a DB2 built-in function. For instance, a UDF can be stage 2 when compared to an equivalent stage 1 built-in function.
- ▶ Improve your productivity, as you do not need to develop and maintain your own code.

6.3 Built-in function characteristics

Built-in functions are classified as column functions or scalar functions depending on the type of their arguments. Scalar functions (like the CHAR function) can only have single values as arguments when column functions (like AVG for example) can have a set of like values as an argument.

6.4 List of built-in functions before Version 6

This is a list of all the functions that were available prior to DB2 UDB for OS/390 Version 6:

Column functions

AVG, COUNT, MAX, MIN and SUM.

Scalar functions

CHAR, COALESCE, DATE, DAY, DAYS, DECIMAL, DIGITS, FLOAT, HEX, HOUR, INTEGER, LENGTH, MICROSECOND, MINUTE, MONTH, NULLIF, SECOND, STRIP, SUBSTR, TIME, TIMESTAMP, VALUE, VARGRAPHIC and YEAR.

Arithmetic and string operators

+, -, *, /, || and CONCAT

6.5 New built-in functions in Version 6

Following is the list containing the new built-in functions that were introduced in DB2 Version 6. As you can see, the list is quite extensive.

Column functions

COUNT_BIG	Returns the number of rows or values in a set of rows or values. It performs the same function as COUNT, except the result can be greater than the maximum value of an integer.
STDDEV	Returns the standard deviation of a set of numbers.
VAR, VARIANCE	Returns the variance of a set of numbers.

Scalar functions

ABS, ABSVAL	Returns the absolute value of the argument.
ACOS	Returns the arccosine of an argument as an angle, expressed in radians.
ASIN	Returns the arcsine of an argument as an angle, expressed in radians.
ATAN	Returns the arctangent of an argument as an angle, expressed in radians.
ATANH	Returns the hyperbolic arctangent of an argument as an angle, expressed in radians.
ATAN2	Returns the arctangent of x and y coordinates as an angle, expressed in radians.
BLOB	Returns a BLOB representation of a string of any type or a ROWID type.
CEIL, CEILING	Returns the smallest integer value that is greater than or equal to the argument.
CLOB	Returns a CLOB representation of a character string or ROWID type.
COS	Returns the cosine of an argument that is expressed as an angle in radians.
COSH	Returns the hyperbolic cosine of an argument that is expressed as an angle in radians.
DAYOFMONTH	Identical to the DAY function.
DAYOFWEEK	Returns an integer between 1 and 7 which represents the day of the week where 1 is Sunday and 7 is Saturday.
DAYOFYEAR	Returns an integer between 1 and 366 which represents the day of the year where 1 is January 1.
DBCLOB	Returns a DBCLOB representation of a graphic string type.
DOUBLE	Returns a double precision floating-point representation of a number or character string in the form of a numeric constant.
DOUBLE_PRECISION	See description for built-in function DOUBLE.
EXP	Returns the exponential function of an argument.
FLOOR	Returns the largest integer value that is less than or equal to the argument.

GRAPHIC	Returns a GRAPHIC representation of a character or graphic string value.
IDENTITY_VAL_LOCAL	Returns the most recently assigned value for an identity column.
IFNULL	Identical to the COALESCE and VALUE functions with two arguments.
INSERT	Returns the modified contents of a string.
JULIAN_DAY	Returns an integer value representing a number of days from January 1,4712 BC (the start of the Julian date calendar) to the date specified in the argument.
LCASE	Returns a string with the characters converted to lowercase.
LOWER	Identical to LCASE.
LEFT	Returns a string that consists of the specified number of leftmost bytes of a string.
LN	Returns the natural logarithm of an argument.
LOCATE	Returns the starting position of the first occurrence of one string within another string based on a specified starting position.
LOG	Identical to LN.
LOG10	Returns the base 10 logarithm of an argument.
LTRIM	Removes blanks from the beginning of a string.
MIDNIGHT_SECONDS	Returns an integer value in the range 0 to 86400 representing the number of seconds between midnight and the time specified in the argument.
MOD	Divides the first argument by the second argument and returns the remainder.
POSSTR	Identical to LOCATE function (except that POSSTR always starts at position 1).
POWER	Returns the value of one argument raised to the power of a second argument.
QUARTER	Returns an integer between 1 and 4 which represents the quarter of the year in which the date resides.
RADIANS	Returns the number of radians for an argument that is expressed in degrees.
RAISE_ERROR	Causes the statement that includes the function to return an error with the specified SQLSTATE and diagnostic-string.
RAND	Returns a double precision floating-point random number.
REAL	Returns a single precision floating-point representation of a number or character string in the form of a numeric constant.
REPEAT	Returns a string composed of an expression repeated a specified number of times.
REPLACE	Replaces all occurrences of a string within an input string with a new string.
RIGHT	Returns a string that consists of the specified number of rightmost bytes of a string.
ROUND	Rounds a number to a specified number of decimal points.

ROWID	Casts the input argument type to the ROWID type.
RTRIM	Removes blanks from the end of a string.
SIGN	Returns an indicator of the sign of the argument.
SIN	Returns the sine of an argument that is expressed as an angle in radians.
SINH	Returns the hyperbolic sine of an argument that is expressed as an angle in radians.
SMALLINT	Returns a small integer representation of a number or character string in the form of a numeric constant.
SPACE	Returns a character string consisting of the number of SBCS blanks specified by the argument.
SQRT	Returns the square root of the argument.
TAN	Returns the tangent of an argument that is expressed as an angle in radians.
TANH	Returns the hyperbolic tangent of an argument that is expressed as an angle in radians.
TIMESTAMP_FORMAT	Returns a timestamp for a character string, using a specified format to interpret the string.
TRANSLATE	Returns a string with one or more characters translated.
TRUNCATE	Truncates a number to a specified number of decimal points.
UCASE	Returns a string with the characters converted to uppercase.
UPPER	Identical to UCASE.
VARCHAR	Returns a varying length character string representation of a character string, datetime value, integer number, decimal number, floating-point number, or ROWID value.
VARCHAR_FORMAT	Returns a varying-length character string representation of a timestamp, with the string in a specified format.
WEEK	Returns an integer between 1 and 54 which represents the week of the year. The week starts with Sunday.

Some of these functions provide different ways of obtaining the same result. For a detailed description of the syntax and how to use these functions, please refer to the *DB2 UDB for OS/390 Version 6 SQL Reference*, SC26-9014.

6.6 New functions in Version 7

Here is list containing the new built-in functions that were added in DB2 Version 7.

Column functions

STDDEV_SAMP	Returns the sample standard deviation ($\sqrt{n-1}$) of a set of numbers.
VARIANCE_SAMP	Returns the sample variance of a set of numbers.

Scalar functions

ADD_MONTHS	Returns a date that represents the date argument plus the number of months argument.
------------	--

CCSID_ENCODING	Returns the encoding scheme of a CCSID with a value of ASCII, EBCDIC, UNICODE, or UNKNOWN.
DAYOFWEEK_ISO	Returns an integer in the range of 1 to 7, where 1 represents Monday.
LAST_DAY	Returns a date that represents the last day of the month indicated by date-expression.
MAX(scalar)	Returns the maximum value in a set of values.
MIN(scalar)	Returns the minimum value in a set of values.
MULTIPLY_ALT	Returns the product of the two arguments as a decimal value, used when the sum of the argument precision exceeds 31.
NEXT_DAY	Returns a timestamp that represents the first weekday, named by the second argument, after the date argument.
ROUND_TIMESTAMP	Returns a timestamp rounded to the unit specified by timestamp format string.
TRUNC_TIMESTAMP	Returns a timestamp truncated to the unit specified by the timestamp format string.
WEEK_ISO	Returns an integer that represents the week of the year with Monday as first day of week.

For a complete list of all the functions available and for a detailed description of the syntax and how to use these functions, please refer to *DB2 UDB for OS/390 and z/OS Version 7 SQL Reference*, SC26-9944.

Tip: When using the function WEEK, make sure that you understand that the weeks are based on a starting day of SUNDAY. If you want your week to start on a Monday, then you should use WEEK_ISO instead.

6.7 Built-in function restrictions

The argument of a scalar function can be a column function only if a column function is allowed in the context in which the scalar function is used. This simply means that you may code something like CHAR(AVG(PRICE)). It makes sense to convert the result of the column function AVG into char. On the other hand, it does not make sense to code WEEK_ISO(AVG(SALARY)) since the result of the AVG function is not a date data type and you receive an SQLCODE -171.

If the argument of a scalar function is a string from a column with a field procedure, the function is applied to the decoded form of the value.

Enhancements that allow a more flexible design

In this part we discuss several enhancements that allow for more flexible application design including:

- ▶ Temporary tables
 - SQL global temporary tables (created temporary tables)
 - Declared temporary tables
- ▶ Savepoints
- ▶ Unique column identification
 - Identity columns
 - ROWID and direct row access



Temporary tables

When you need a table only for the life of an application process, you can create a temporary table. There are two kinds of temporary tables:

- ▶ Created temporary tables, which you define using a `CREATE GLOBAL TEMPORARY TABLE` statement (introduced in DB2 V5 as global temporary tables).
- ▶ Declared temporary tables, which you define in a program using a `DECLARE GLOBAL TEMPORARY TABLE` statement.

SQL statements that use temporary tables can run faster, because:

- ▶ There is no logging for created temporary tables. Only UNDO records (required for rollback processing) are logged for declared temporary tables.
- ▶ There is no locking for created temporary tables and only share level locks for declared temporary table space.

7.1 Summary of differences between types of tables

The following sections provide more details on created temporary tables and declared temporary tables. In the Table 7-1 we show the most significant differences between the three types of tables used by DB2.

Table 7-1 Distinctions between DB2 base tables and temporary tables

Base tables	Created temporary tables	Declared temporary tables
Can be pre-created or created in an application program	Can be pre-created or created in an application program	Must be created in an application program
CREATE TABLE statement places definition in SYSTABLES.	CREATE GLOBAL TEMPORARY TABLE statement places definition in SYSTABLES.	DECLARE GLOBAL TEMPORARY TABLE statement does not place a definition in SYSTABLES.
CREATE TABLE creates an empty instance of the table.	CREATE GLOBAL TEMPORARY TABLE does not create an instance of the table.	DECLARE GLOBAL TEMPORARY TABLE creates an empty instance of the table for each application process.
All references to the table from multiple applications are to a single "persistent" table.	References to the table in multiple application processes refer to the same description but a distinct instance of the table.	References to the table in multiple application processes refer to a distinct description and instance of the table.
Locking, logging, and recovery.	No locking, no logging, no recovery.	Lock on DBD and intent lock on table space, logging of UNDO records, and limited recovery.
The table can be stored in a simple, segmented, or partitioned table space in a user-defined database or in the default database DSNDB04.	Logical work files are used to store the table in the WORKFILE database, usually called DSNDB07.	The table is stored in a segmented table space in the TEMP database (a database that is defined AS TEMP).
Can have indexes.	Cannot have indexes.	Can have indexes.
Can INSERT, DELETE, and UPDATE individual rows.	Can INSERT, mass DELETE (without WHERE clause), cannot UPDATE rows.	Can INSERT, DELETE, and UPDATE individual rows.
WITH DEFAULT clause supported.	No WITH DEFAULT clause other than null.	WITH DEFAULT clause supported.
UDTs supported.	UDTs supported.	UDTs not supported.
SAVEPOINTS supported.	SAVEPOINTS not supported.	SAVEPOINTS supported.
Threads can be reused.	Threads can be reused.	Threads can be reused under certain conditions.

7.2 Created temporary tables

In this section, we discuss SQL global temporary tables (from now on called created temporary tables or CTTs).

7.2.1 What is a created temporary table?

Created temporary tables were introduced in DB2 V5 (as “global temporary tables”). These tables are created once by the administrator, and then any program, thread or connection can allocate their own instance of the table by inserting into the table. Other applications referencing the created temporary table instantiate (materialize) a different instance of the table in a work file for their connection. The rows in the table only exist for the duration of a unit of work (unless the cursor is defined WITH HOLD).

The term “global” is taken from the SQL92 Full-Level standard and, as applied to DB2 temporary tables, it means global to an application process at the current server. Thus, if an instance of a global (created) temporary table exists in an application process at the current server, that instance can be referenced by any program in the same process at the *same* server.

Note that an “application process” is a thread/connection. As long as a thread persists, any instance of a global temporary table created within that thread persists. So when a thread is reused, the table persists (the work file is still defined in DSNDB07), although a COMMIT deletes all the rows of the temporary table (when no cursors WITH HOLD are open against the table).

7.2.2 Why use created temporary tables

The main advantages of created temporary tables are:

- ▶ No log records are written when changes to a created temporary table are made (because created temporary tables are not recoverable).
- ▶ No locks need to be taken, because every program that references the created temporary table uses its own work file (instances of work files are not shared).

The properties of these tables are a subset of the properties of the global temporary tables of the SQL92 Full-Level standard.

In stored procedures, temporary tables can be very useful when there is a need to return large result sets from a stored procedure to a calling program, especially when the caller of the stored procedure is executing on a remote system, to avoid contention on the real table.

It can also be used as a means to pass data between subroutines within a program. Instead of returning data in working storage as a large array, a subroutine can build a temporary table and pass back only the name of the table to the caller that can fetch data from that table at its convenience.

Another uses for this type of temporary table is to store data that has been read from a non-DB2 source, like IMS, VSAM or flat files, for subsequent SQL processing, like joining the created temporary table with other DB2 tables.

Yet another example of the usage of created temporary tables can be in a data warehouse environment. Tables in those environments are usually very large, and some joins between several data warehouse tables might be unsuitable. Created temporary tables can be used to select some sample data from a large table and make a join with the temporary table (for example, for some data mining or pattern recognition processes). However, a created temporary table is not indexable, so it should not be used, for example, as an inner table of a nested loop join.

7.2.3 Created temporary tables characteristics

In this section we will describe the typical characteristics of created temporary tables.

No locking, no logging, no recovery

Since an instance of a temporary table exists in a work file, locking and logging do not apply.

The table work files are unique to an application, so there is no locking necessary or desirable. Because they are temporary, there is no concept of recovery. If the application rolls back, the work file data is lost and changes in the created temporary table are not undone. Therefore, there is no need for logging.

The lack of locking is consistent with SQL92 temporary tables. The lack of logging is consistent with the default for SQL92 temporary tables (ON COMMIT DELETE ROWS).

Defining a created temporary table

At least one of the following privileges is required to create created temporary tables:

- ▶ The CREATETMTAB system privilege to allow users to issue the CREATE GLOBAL TEMPORARY TABLE statement.
- ▶ The CREATETAB database privilege for any database
- ▶ DBADM, DBCTRL, DBMAINT authority for any database
- ▶ SYSADM, SYSCTRL authority

Additional privileges might be required when the data type of a column is a distinct type or the LIKE clause is specified.

In Example 7-1, the CREATE GLOBAL TEMPORARY TABLE statement creates a definition of a created temporary table called GLOBALITEM in the current server's catalog.

Example 7-1 Created temporary table DDL

```
CREATE GLOBAL TEMPORARY TABLE SC246300.GLOBALINEITEM
  (NORDERKEY  INTEGER NOT NULL
  ,ITEMNUMBER INTEGER NOT NULL
  ,QUANTITY   INTEGER NOT NULL)
```

Example 7-2 shows the information that is stored in the DB2 catalog for created temporary tables.

Example 7-2 Created temporary table in SYSIBM.SYSTABLES

```
SELECT NAME, CREATOR, TYPE, DBNAME, TSNAME
FROM SYSIBM.SYSTABLES
WHERE NAME = 'GLOBALINEITEM'
```

NAME	CREATOR	TYPE	DBNAME	TSNAME
GLOBALINEITEM	SC246300	G	DSNDB06	SYSPKAGE

Note: All created temporary tables seem to reside in the catalog table space SYSPKAGE, but in reality, they are instantiated (materialized) in the DSNDB07 work files. The TYPE = 'G' denotes a created temporary table.

In Example 7-3, the LIKE *table-name* or *view-name* specifies that the columns of the created temporary table have exactly the same name and description as the columns from the identified table or view. That is, the columns of SC246300.GLOBALIKE_ITEM have the same name and description as those of SC246300.TBLINEITEM except for attributes not allowed for created temporary tables and no default values other than NULL. The name specified after the LIKE must identify a table, view, or created temporary table that exists at the current server, and the privilege set must implicitly or explicitly include the SELECT privilege on the identified table or view.

A created temporary table GLOBALIKE_ITEM, similar to TBLINEITEM can be useful, for example, for a program that handles all the changes that a customer makes while shopping on the internet. All items from an order can be inserted in a created temporary table work file without locking and logging contention. When the customer confirms an order, the created temporary rows can be inserted in TBLINEITEM before the work file data is lost at COMMIT time. If the customer cancels the order, then entries in the GLOBALIKE_ITEM temporary table are removed by the rollback and the TBLINEITEM table is not involved in the process and contention is avoided.

However, created temporary tables are not updatable. So if the customer would like to change the quantity of an item during the ordering process, he has to start the order process from the beginning.

Example 7-3 Use of LIKE clause with created temporary tables

```
CREATE GLOBAL TEMPORARY TABLE SC246300.GLOBALIKE_ITEM
LIKE SC246300.TBLINEITEM
```

This clause is similar to the LIKE clause on CREATE TABLE with the following differences:

- ▶ If any column of the identified table or view has an attribute value that is not allowed for a column of a created temporary table (for example, UNIQUE), that attribute value is ignored and the corresponding column in the new created temporary table will have the default value for that attribute unless otherwise indicated.
- ▶ If a column of the identified table or view allows a default value other than the null value, then that default value is ignored and the corresponding column in the new created temporary table will have no default. A default value other than null value is not allowed for any column of a created temporary table.

You can also create a view on a created temporary table. Example 7-4 show a view on the created temporary table SC246300.GLOBALINEITEM. Every application sees different values returned from the view SC2463.GLOBALVIEW depending on the content of their own created temporary table SC246300.GLOBALINEITEM. The view SC246300.GLOBALVIEW is defined in the catalog as a normal view on a base table.

Example 7-4 View on a created temporary table

```
CREATE VIEW SC246300.GLOBALVIEW
AS SELECT NORDERKEY
, ITEMNUMBER
```

```
FROM SC246300.GLOBALINEITEM ;
```

Example 7-5 shows how to drop a created temporary table.

Example 7-5 Dropping a created temporary table

```
DROP TABLE SC246300.GLOBALINEITEM
```

Creating an instance of a temporary table

An empty instance of a given temporary table is instantiated (allocated or materialized) with the first implicit or explicit reference to the named temporary table in an OPEN, SELECT, INSERT, or DELETE operation executed by any program in the application process.

Note: The SQL UPDATE statement is not in the list because updates are not allowed on created temporary tables.

An instance of a created temporary table exists at the current server until one of the following actions occurs:

- ▶ The remote server connection under which the instance was created terminates.
- ▶ The unit of work under which the instance was created completes.
When you execute a COMMIT statement, DB2 deletes the instance of the created temporary table unless a cursor for accessing the created temporary table is defined WITH HOLD and is open. When you execute a ROLLBACK statement, DB2 deletes the instance of the created temporary table.
- ▶ The application process ends.

Suppose that you create a temporary table GLOBALINEITEM and then run an application like the one shown in Example 7-6:

Example 7-6 Using a created temporary table in a program

```
EXEC SQL DECLARE C1 CURSOR
      FOR SELECT * FROM SC246300.GLOBALINEITEM ;

EXEC SQL INSERT INTO SC246300.GLOBALINEITEM
      SELECT      NORDERKEY
                 ,L_ITEM_NUMBER
                 ,QUANTITY
      FROM SC246300.TBLINEITEM ;

EXEC SQL OPEN C1 ;
.
application process
.
EXEC SQL COMMIT ;
.
application process
.
EXEC SQL CLOSE C1;
```

When you execute the INSERT statement, DB2 creates an instance of GLOBALINEITEM and populates that instance with rows from table TBLINEITEM. When the COMMIT statement is executed, DB2 deletes all rows from GLOBALINEITEM.

However, if you change the declaration of cursor C1 to:

```
EXEC SQL DECLARE C1 CURSOR WITH HOLD
FOR SELECT * FROM SC246300.GLOBALINEITEM ;
```

The contents of GLOBALINEITEM is not deleted until the application ends because C1, a cursor defined WITH HOLD, is open when the COMMIT statement is executed. In either case, DB2 drops the instance of GLOBALINEITEM when the application ends.

Multiple instances of a temporary table

Created temporary table instances are implemented in DB2 using an existing facility called logical work files. Each instance occupies its own logical work file exclusively. There can be one or more logical work files per work file table space. There is only one instantiation of a temporary table per logical work file. The user cannot select which work file table space or which logical work file is used to contain a given temporary table. DB2 chooses the work file table space (among those created by the administrator).

The logical work file for a temporary table is available for that temporary table name for the life of the thread/connection which created the logical work file and is not used for any other purpose, unless the work file is deleted at COMMIT or ROLLBACK/Abort.

Note: Each application process (thread/connection) may create an instance of a created temporary table. Each instance is unique to the thread/connection that created it.

COMMIT and ROLLBACK with created temporary tables

COMMIT deletes all rows from all created temporary tables that do not have WITH HOLD cursors on them, and ROLLBACK deletes all rows from all created temporary tables, regardless of held-cursors. If the plan or package is bound RELEASE(COMMIT), then COMMIT or ROLLBACK also causes the logical work files to be deleted.

All rows and logical work files are deleted at thread termination.

The use of a created temporary table does not preclude thread reuse, and a logical work file for a temporary table name remains available until de-allocation (assuming RELEASE(DEALLOCATE)). No new logical work file is allocated for that temporary table name when the thread is reused.

A temporary table instantiated by an SQL statement using a three-part table name via DB2 private protocol, can be accessed by another SQL statement using the same three-part name in the same application process for as long as the DB2 thread/connection which established the instantiation is not terminated.

Using a temporary table to return result sets

You can use a created temporary table or declared temporary table to return result sets from a stored procedure. This capability can be used to return non-relational data to a DRDA client.

For example, you can access IMS data from a stored procedure in the following way:

- Use the IMS ODBA interface to access IMS databases.

- ▶ Insert the data into a temporary table.
- ▶ Open a cursor against the temporary table.
- ▶ End the stored procedure.
- ▶ The client can then fetch the rows from the cursor defined on the temporary table.

Considerations

Those responsible for system planning should be aware that work file and buffer pool storage might need to be increased depending on how large a created temporary table is and the amount of sorting activity required. Given that a logical work file will not be used for any other purpose for as long as that instantiation of a created temporary table is active, there may be a need to increase the size or number of physical work file table spaces, especially when there is also a significant amount of other work (like sort activity) using the work file database concurrently running on the system.

Those responsible for performance monitoring and tuning should be aware that for created temporary tables, no index scans are done, only table scans are associated with created temporary tables. In addition, a DELETE of all rows performed on a created temporary table does not immediately reclaim the space used for the table. Space reclamation does not occur until a COMMIT is done.

7.2.4 Created temporary tables pitfalls

Different applications can allocate different work files containing instances of a created temporary table, which is already defined in the catalog, so the definition of the table is common for every application but not the rows contained in it.

Created temporary tables are useful whenever some non-sharable temporary processing is needed and the data does not have to be kept when an application issues a commit. Before created temporary tables, the only way to avoid logging was to move the process out of DB2. Remember that there is no locking for created temporary tables.

Be careful, you should not blindly move all batch processing that you used to do outside DB2 in order to avoid logging, into created temporary tables. Created temporary tables are not indexable. Therefore, they may not provide the best access path if they are very large or if you have to scan them many times.

7.2.5 Created temporary tables restrictions

Examples of implementing temporary tables and information about restrictions and extensions of temporary tables can be found in:

- ▶ *DB2 UDB for OS/390 and z/OS Version 7 Application Programming and SQL Guide*, SC26-9933
- ▶ *DB2 UDB for OS/390 and z/OS Version 7 SQL Reference*, SC26-9944

For information about temporary tables and their impact on DB2 resources, see:

- ▶ “Work file data sets” in the *DB2 UDB for OS/390 and z/OS Version 7 Administration Guide*, SC26-9931

Here are some restrictions of created temporary tables:

- ▶ No indexes can be defined for created temporary tables.
- ▶ Their columns cannot have default values other than NULLS.
- ▶ They cannot have primary, foreign or unique key specifications.

- ▶ They cannot be defined as parents in a referential constraint.
- ▶ The columns cannot have LOB or ROWID data types (or a distinct type based on one).
- ▶ Cannot have a validproc, editproc, fieldproc or trigger.
- ▶ Cannot be referenced:
 - In any DB2 utility commands (message DSNU062I is issued for this error).
 - In a LOCK TABLE statement.
 - As the target of an UPDATE statement, where the target is the created temporary table or a view on the created temporary table. If you try to UPDATE a created temporary table, you receive the following message:

```

DSNT408I  SQLCODE = -526, ERROR:  THE REQUESTED OPERATION OR USAGE DOES NOT
        APPLY TO CREATED TEMPORARY TABLE tablename
DSNT418I  SQLSTATE  = 42995 SQLSTATE RETURN CODE

```

However, the created temporary table can be referenced in the WHERE clause of an UPDATE statement.

- ▶ A created temporary table can be referenced in the FROM clause of any subselect. As with all tables stored in work files, query parallelism is not considered for any query referencing a created temporary table in the FROM clause.
- ▶ DELETE FROM specifying a created temporary table is valid when the statement does not include a WHERE or WHERE CURRENT OF clause. When a view is created on a created temporary table, then the CREATE VIEW statement for the view cannot contain a WHERE clause because the DELETE FROM view fails with an SQLCODE -526. However, you can delete all rows (mass delete) from a created temporary table or a view on a created temporary table.
- ▶ If a created temporary table is referenced in the subselect of a CREATE VIEW statement, the WITH CHECK OPTION must not be specified for the WHERE clause of the subselect of the CREATE VIEW statement.
- ▶ GRANT ALL PRIVILEGES ON a created temporary table is valid, but specific privileges cannot be granted on a created temporary table. Of the ALL privileges, only the ALTER, INSERT, DELETE, and SELECT privileges can actually be used on a created temporary table.
- ▶ REVOKE ALL PRIVILEGES ON a created temporary table is valid, but specific privileges cannot be revoked from a created temporary table.
- ▶ The DROP DATABASE statement cannot be used to implicitly drop a created temporary table. You must use the DROP TABLE statement to drop a created temporary table.
- ▶ ALTER TABLE on a created temporary table is valid only when used to add a column to it and if any column being added has a default value of NULL. When the ALTER is performed, any plan or package that references the table is marked as “invalid” (that is, the SYSPLAN or SYSPACKAGE column VALID is changed to ‘N’), and the next time the plan or package is run, DB2 performs an automatic rebind (autobind) of the plan or package. The added column is then available to the SQL statements in the plan or package. On a successful autobind, the VALID column is changed to have a ‘Y’.
- ▶ Created temporary tables can be referenced in DROP TABLE, CREATE VIEW, COMMENT ON, INSERT, SELECT, LABEL ON, CREATE ALIAS, CREATE SYNONYM, CREATE TABLE LIKE, DESCRIBE TABLE, and DECLARE TABLE. There are no restrictions or additional rules other than the ones mentioned above.

7.3 Declared temporary tables

In this section, we describe the features of declared temporary tables (DTTs) and their characteristics.

7.3.1 What is a declared temporary table?

A declared temporary table is a type of table which is defined with the `DECLARE GLOBAL TEMPORARY TABLE` statement. Declared temporary tables are created in a program and exist only during the life of the application process. Declared temporary tables provide another convenient way to store temporary data.

You can populate the declared temporary table using `INSERT` statements, modify the table using searched or positioned `UPDATE` or `DELETE` statements, and query the table using `SELECT` statements.

A temporary database and temporary table spaces must be created before you can start using declared temporary tables.

7.3.2 Why use declared temporary tables

Some of the possible uses of declared temporary tables include:

- ▶ For business intelligence applications, when you want to process extracts of the data and perform further result set processing. For example, in the case of joining a big table to many others and there is not a good index for the selection criteria, you can extract the rows needed and use the smaller temporary table for the join. You can create an index on a declared temporary table to improve the performance.
- ▶ As a staging area for making IMS data accessible to ODBC. For example, a client application program can call a stored procedure to access IMS data. The IMS data can be inserted into a declared temporary table. The data in the declared temporary table may then be processed by the client using standard SQL.
- ▶ To hold result sets in stored procedures when you are worried about remote clients holding locks for too long on the actual table. Therefore, you can have the stored procedure access the actual table, extract the rows that are of interest to the client program and store them into a declared temporary table. Again, you can create an index on the declared temporary table if table scans do not provide adequate performance.

7.3.3 Declared temporary tables characteristics

The `DECLARE GLOBAL TEMPORARY TABLE` statement creates a temporary table for the current application process.

The qualifier for a declared temporary table, if specified, must be `SESSION`. The qualifier need not to be specified, it is implicitly defined to be `SESSION`. The `DECLARE` statement is successful even if a table is already defined in the DB2 catalog with the same fully-qualified name. In Example 7-7, we show you typical DDL to declared a temporary table.

Example 7-7 Sample DDL for a declared temporary table

```
DECLARE GLOBAL TEMPORARY TABLE SESSION.TEMP_EMP
    (EMPNO CHAR(6) NOT NULL
    ,SALARY DECIMAL(9,2)
    ,BONUS DECIMAL(9,2)
    ,COMM DECIMAL(9,2))
```

ON COMMIT PRESERVE ROWS;

ON COMMIT DELETE ROWS, which is the default definition, specifies that the rows of the table are to be deleted following a commit operation (if no WITH HOLD cursors are open on the table). To avoid mistakes, define the table always as ON COMMIT PRESERVE ROWS when you want to preserve the rows at COMMIT. This way there is not need to have a cursor WITH HOLD open to preserve the rows in the DTT across COMMITS.

Important: Always explicitly drop the declared temporary table when it is no longer needed. If you use the ON COMMIT PRESERVE ROWS option, the thread cannot be inactivated or reused unless the program explicitly drops the table before the commit. If you do not explicitly drop the table, it is possible to run out of usable threads.

7.3.4 Creating a temporary database and table space

Before you can define declared temporary tables, you must create a special database and table spaces for them. You do that by executing the CREATE DATABASE statement with the AS TEMP clause, and then creating several segmented table spaces in that database. A DB2 subsystem can have only one database for declared temporary tables, but that database can contain many table spaces spread across a number of volumes. In a data sharing environment, you must define a single temporary database for each member in a data sharing group. A DB2 subsystem or data sharing member can only have one database defined AS TEMP. The database is not shareable across members.

In Example 7-8, we show you how to create a database and several table spaces to be used for the creation of declared temporary tables.

Example 7-8 Create a database and table spaces for declared temporary tables

```
CREATE DATABASE DBTEMP AS TEMP ;
CREATE TABLESPACE TSTEMP01 IN DBTEMP USING STOGROUP SG246300 SEGSIZE 32
  PRIQTY 120000 SEQTY 120000 BUFFERPOOL BP10 ;
CREATE TABLESPACE TSTEMP02 IN DBTEMP USING STOGROUP SG246300 SEGSIZE 32
  PRIQTY 120000 SEQTY 120000 BUFFERPOOL BP10 ;
CREATE TABLESPACE TSTEMP03 IN DBTEMP USING STOGROUP SG246300 SEGSIZE 32
  PRIQTY 120000 SEQTY 120000 BUFFERPOOL BP10 ;
CREATE TABLESPACE TSTEMP20 IN DBTEMP USING STOGROUP SG246300 SEGSIZE 32
  PRIQTY 24000 SEQTY 24000 BUFFERPOOL BP8K0 ;
CREATE TABLESPACE TSTEMP21 IN DBTEMP USING STOGROUP SG246300 SEGSIZE 32
  PRIQTY 24000 SEQTY 24000 BUFFERPOOL BP8K0 ;
CREATE TABLESPACE TSTEMP30 IN DBTEMP USING STOGROUP SG246300 SEGSIZE 32
  PRIQTY 48000 SEQTY 48000 BUFFERPOOL BP16K0 ;
CREATE TABLESPACE TSTEMP40 IN DBTEMP USING STOGROUP SG246300 SEGSIZE 32
  PRIQTY 96000 SEQTY 96000 BUFFERPOOL BP32K ;
```

Tip: All table spaces in a TEMP database should be created with the same segment size and with the same primary space allocation values. DB2 chooses which table space to place your created temporary table.

You should create table spaces in your TEMP database for all page sizes you use in base tables. DB2 determines which table space in the TEMP database is used for a given declared temporary table. If a DECLARE GLOBAL TEMPORARY TABLE statement specifies a row size that is not supported by any of the table spaces defined in the TEMP database, the

statement fails. DB2 determines the buffer pool based on the page size that is required for the declared temporary table and assigns DTT to a table space in the TEMP database that can handle this page size. An INSERT statement fails if there is insufficient space in the table space used for the declared temporary table. Allocate enough space for all concurrently executing threads to create their declared temporary tables. You may want to have several smaller table spaces rather than a few large ones, to limit the space one declared temporary table can use, since a declared temporary table cannot span multiple physical TEMP table spaces.

Tip: You may want to isolate declared temporary tables to their own set of buffer pools.

An encoding scheme (`CCSID ASCII`, `EBCDIC` or `UNICODE`) cannot be specified for a TEMP database or for a table space defined within a TEMP database. However, an encoding scheme can be specified for a declared temporary table `USING CCSID`. This means that a table space defined within a TEMP database can hold temporary tables with different encoding schemes.

You should bear in mind that the TEMP database could become quite large if the usage of declared temporary tables is high. You may also need to increase the size of the EDM pool to account for this extra database.

START, STOP and DISPLAY DB are the only supported commands against the TEMP database. The standard command syntax should be used but please note the following:

- ▶ You cannot start a TEMP database as RO (read only).
- ▶ You cannot use the AT COMMIT option of the STOP DB command.
- ▶ You cannot stop and start any index spaces that the applications have created.

Tip: Do not specify the following clauses of the CREATE TABLESPACE statement when defining a table space in a TEMP database: `CCSID`, `LARGE`, `MEMBER CLUSTER`, `COMPRESS`, `LOB`, `NUMPARTS`, `DSSIZE`, `LOCKSIZE`, `PCTFREE`, `FREEPAGE`, `LOCKPART`, `TRACKMOD`, `GBPCACHE`, `LOG`.

7.3.5 Creating a declared temporary table

You create an instance of a declared temporary table using the SQL statement `DECLARE GLOBAL TEMPORARY TABLE`. That instance is known only to the application process in which the table is declared, so you can declare temporary tables with the same name, with the same or different columns, in different applications.

You can define a declared temporary table in any of the following three ways:

- ▶ Specify all the columns in the table. Unlike columns of created temporary tables, columns of declared temporary tables can include the `WITH DEFAULT` clause.
- ▶ Use a `LIKE` clause to copy the definition of a base table, created temporary table, or view, alias or synonym that exist at the current server. The implicit definition includes all attributes of the columns as they are described in `SYSIBM.SYSCOLUMNS`.
- ▶ Use the `AS` clause and a fullselect to choose specific columns from a base table, created temporary table or view.

If you want the declared temporary table columns to inherit the defaults of the columns from the table or view that is named in the select, specify the `INCLUDING COLUMN DEFAULTS` clause. If you want the declared temporary table columns to have default values that correspond to their data types, specify the `USING TYPE DEFAULTS` clause.

If the base table, created temporary table, or view from which you select columns has identity columns, you can specify that the corresponding columns in the declared temporary table are also identity columns. Do that by specifying the `INCLUDING IDENTITY COLUMN ATTRIBUTES` clause when you define the declared temporary table.

In Example 7-9, the statement defines a declared temporary table called `TEMPPROD` by explicitly specifying the columns.

Example 7-9 Explicitly specify columns of declared temporary table

```
DECLARE GLOBAL TEMPORARY TABLE SESSION.TEMPPROD
(SERIAL      CHAR(8)      NOT NULL WITH DEFAULT '99999999'
,DESCRIPTION VARCHAR(60) NOT NULL
,PRODCOUNT  INTEGER      GENERATED ALWAYS AS IDENTITY
,MFGCOST    DECIMAL(8,2)
,MFGDEPT    CHAR(3)
,MARKUP     SMALLINT
,SALESDEPT  CHAR(3)
,CURDATE    DATE          NOT NULL);
```

In Example 7-10, the statement defines a declared temporary table called `TEMPPROD` by copying the definition of a base table. The base table has an identity column that the declared temporary table also uses as an identity column.

Example 7-10 Implicit define declared temporary table and identity column

```
DECLARE GLOBAL TEMPORARY TABLE SESSION.TEMPPROD LIKE BASEPROD
INCLUDING IDENTITY COLUMN ATTRIBUTES;
```

In Example 7-11, the statement defines a declared temporary table called `TEMPPROD` by selecting columns from a view. The view has an identity column that the declared temporary table also uses as an identity column. The declared temporary table inherits the default defined columns from the view definition. Notice also the `DEFINITION ONLY` clause. This is to make clear that the `SELECT` is not copying data from the original table but merely its definition.

Example 7-11 Define declared temporary table from a view

```
DECLARE GLOBAL TEMPORARY TABLE SESSION.TEMPPROD
AS (SELECT * FROM PROVIEW)
DEFINITION ONLY
INCLUDING IDENTITY COLUMN ATTRIBUTES
INCLUDING COLUMN DEFAULTS;
```

Example 7-12 show how to drop the definition of a declared temporary table.

Example 7-12 Dropping a declared temporary table

```
DROP TABLE SESSION.TEMPPROD ;
```

7.3.6 Using declared temporary tables in a program

To refer to a declared temporary table in an SQL statement, you must qualify the table-name with SESSION. You can specify SESSION explicitly in the table-name reference in an SQL statement. You can also specify it using the QUALIFIER bind option to specify SESSION as the qualifier for all unqualified table references in the plan or package. This means that you either need to specify qualifier (owner) in all base tables or you specify the SESSION for all temporary tables. The later may be easier to implement, since the qualifier does not change depending on the DB2 subsystem where the program is run.

Tip: Use a fully qualified name for your declared temporary tables inside your programs

If you use SESSION as the qualifier for a table name in DML but the application process has not yet declared a temporary table with the same name, DB2 assumes that you are not referring to a declared temporary table and looks to see if the table can be found in the catalog. If you have used SESSION as the owner for non-declared temporary tables, you may have a release-to-release incompatibility problem. A called program may assume that a declared temporary table has been declared by its calling program, if a new calling program does not declare the temporary table, the called program may end up using a base table instead of the intended declared temporary table. Of course this assumes that the intended declared temporary table has the same name as an existing base table with an owner of SESSION.

When a plan or package is bound, any static SQL statement that references any table qualified by SESSION, is not completely bound. However, the bind of the plan or package succeeds if there are no errors. These static SQL statements which reference a table-name qualified by SESSION are incrementally bound at run time if they are executed by the application process. DB2 handles the SQL statements this way because the definition of a declared temporary table does not exist until the DECLARE GLOBAL TEMPORARY TABLE statement is executed and, therefore, DB2 must wait until the plan or package is run to determine if SESSION.*tablename* refers to a base table or a declared temporary table.

When a program in an application process 'P' executes a DECLARE GLOBAL TEMPORARY TABLE statement, an empty instance of the declared temporary table is created. Any program in process 'P' can reference the declared temporary table and any of these references is a reference to the same instance of the declared temporary table.

A declared temporary table is automatically dropped when the application process that declared it terminates. It is advisable, though, to explicitly drop all declared temporary tables, when they are no longer needed. The DB2 thread used by an application process that declares one or more temporary tables with the ON COMMIT PRESERVE ROWS attribute, only qualifies for reuse or to become an inactive thread, if the application process explicitly drops all such declared temporary tables before it issues its last explicit COMMIT request. This action is not required for temporary tables declared with the ON COMMIT DELETE ROWS attribute for thread reuse in a local environment (IMS or CICS) but is always required for a remote connection to be eligible for reuse (or become inactive).

Suppose you execute the statements from Example 7-13 in an application program:

Example 7-13 Declared temporary tables in a program

```
EXEC SQL
  DECLARE GLOBAL TEMPORARY TABLE SESSION.TEMPPROD
  AS (SELECT * FROM BASEPROD)
  DEFINITION ONLY
  INCLUDING IDENTITY COLUMN ATTRIBUTES
```

```

        INCLUDING COLUMN DEFAULTS
        ON COMMIT PRESERVE ROWS
END-EXEC
.
.
.
EXEC SQL INSERT INTO SESSION.TEMPPROD SELECT * FROM BASEPROD END-EXEC
.
.
.
EXEC SQL COMMIT END-EXEC
.
.
EXEC SQL DROP TABLE SESSION.TEMPPROD END-EXEC

```

When the DECLARE GLOBAL TEMPORARY TABLE statement is executed, DB2 creates an empty instance of TEMPPROD. The INSERT statement populates that instance with rows from table BASEPROD. The qualifier, SESSION, must be specified in any statement that references TEMPPROD. When the application issues the COMMIT statement, DB2 keeps all rows in TEMPPROD because TEMPPROD is defined with ON COMMIT PRESERVE ROWS. In that case you need to drop the table before the program ends to avoid problems with thread reuse and inactive threads.

7.3.7 Creating declared temporary tables for scrollable cursors

DB2 uses declared temporary tables for processing scrollable cursors. Therefore, before you can use a scrollable cursor, your database administrator needs to create a TEMP database and TEMP table spaces for those declared temporary tables. If there is more than one TEMP table space in the subsystem, DB2 chooses the table spaces to use for scrollable cursors.

The page size of the TEMP table space must be large enough to hold the longest row in the declared temporary table. See the *DB2 UDB for OS/390 and z/OS Version 7 Installation Guide*, GC26-9936 for information on calculating the page size for TEMP table spaces that are used for scrollable cursors.

7.3.8 Remote declared temporary tables

Where remote servers are involved, a reference to a declared temporary table must use the same server connection that was used to declare the temporary table, and that server connection must have never been terminated after the temporary table was declared there. When the connection to the application server at which the temporary table was declared terminates, the temporary table is dropped and its rows are destroyed.

Accessing declared temporary tables using three-part names

You can access a remote declared temporary table using a three-part name only if you use DRDA access. However, if you combine explicit CONNECT statements and three-part names in your application, a reference to a remote declared temporary table must be a forward reference. The Example 7-14 shows how you can perform the series of actions, which includes a forward reference to a declared temporary table:

Example 7-14 Three-part name of declared temporary table

```

EXEC SQL CONNECT TO CHICAGO ;           /* Connect to the remote site */

EXEC SQL

```

```

DECLARE GLOBAL TEMPORARY TABLE TEMPPROD /* Define the temporary table */
      (CHARCOL CHAR(6) NOT NULL) ;      /* at the remote site */

EXEC SQL CONNECT RESET ;                /* Connect back to local site */

EXEC SQL
      INSERT INTO CHICAGO.SESSION.TEMPPROD /* Access the temporary table */
      (VALUES 'ABCDEF') ;                /* at the remote site (forward */
                                          /* reference) */

```

7.3.9 Creating indexes

You can also create indexes on a declared temporary table. Creating an index on a declared temporary table should only be done in cases where it has a significant positive impact on performance. If you specify a qualifier on the CREATE INDEX statement for the index-name, you must specify SESSION. If you omit the qualifier, DB2 uses SESSION as the implicit qualifier. In the ON *table-name* clause you must specify SESSION as the qualifier for *table-name*. Otherwise, *table-name* does not refer to a declared temporary table.

Do not specify the following clauses of the CREATE INDEX statement when defining an index on a declared temporary table: COPY, FREEPAGE, PART (on CLUSTER), DEFER, PCTFREE, USING VCAT, GBPCACHE, DEFINE NO. Index spaces are created in the TEMP database. An index on a declared temporary table is implicitly dropped when the declared temporary table on which it was defined is explicitly or implicitly dropped.

You can specify the following clauses: Column names ASC/DESC, CLUSTER, PIECESIZE, UNIQUE with or without WHERE NOT NULL, and USING STOGROUP.

In order to assist DB2's access path selection for declared temporary tables, some basic statistics are collected by DB2 but not stored in the catalog. These statistics are maintained and used dynamically for optimizing the access path as rows are processed. If you are concerned about access path selection, you should use EXPLAIN for SQL executed against declared temporary tables, analyze the output in your PLAN_TABLE as you would for any other SQL and review your indexing strategy. You cannot run RUNSTATS against a declared temporary table or its indexes. Since there is no catalog definition for the temporary objects, there are no statistics available for the utility or you to modify.

When considering whether to create an index on a declared temporary table, bear in mind the overhead of creating it. A large number of create index statements can have an impact on system performance. You see more logging occurring. In addition, large scale creation, opening and deletion of VSAM data sets for the indexes can increase global resource serialization (GRS) activity. In some cases it may be quicker to simply scan the table, than for DB2 to create an index and use it to access the data. Remember that you probably only insert the required rows into the table, so with index access you merely avoid a sort or improve joins rather than minimize the number of rows read.

7.3.10 Usage considerations

Since rollback and savepoints are supported for declared temporary tables, you may see an increase in the amount of logging activity compared to that expected for created temporary tables. The same number of log records are written as for activity against a base table. However, the individual records are shorter as only UNDO information is recorded rather than UNDO/REDO information.

7.3.11 Converting from created temporary tables

If you are using created temporary tables as they were introduced in *DB2 Version 5* and you wish to create indexes on the tables or do positioned/searched updates and deletes, you may want to convert them to declared temporary tables. Here are some considerations:

- ▶ You can see an increase in logging. UNDO records are written to support rollback when using declared temporary tables.
- ▶ There is a difference in the way in which space is managed. A single declared temporary table is limited to the space available within the TEMP table space in which DB2 has placed it (or a max of 64GB). Created temporary tables are stored in the work files (typically DSNDB07) and can span multiple work files.
- ▶ Since declared temporary tables are stored in their own database, you have the option of preventing impact to other work using the work files. Actions might include isolation of buffer pools for TEMP table spaces, allocating them on separate I/O devices and prevention of out-of-space conditions with other sort processes.

7.3.12 Authorization

No authority is required to declare a temporary table unless you use the LIKE clause. In this case, SELECT access is required on the base table or view specified.

PUBLIC implicitly has authority to create tables in the TEMP database and USE authority on the table spaces in that database. PUBLIC also has all table privileges (declare, access, and drop) on declared temporary tables implicitly. The PUBLIC privileges are not recorded in the catalog nor are they revokable.

Despite PUBLIC authority, there is no security exposure, as the table can only be referenced by the application process that declared it.

7.3.13 Declared temporary table restrictions

Please note the following restrictions when using declared temporary tables:

- ▶ LOBs, ROWID, and user-defined data types (UDT) columns are not allowed.
- ▶ They cannot be specified in referential constraints.
- ▶ They cannot be specified in a TABLE LIKE parameter to a user defined function (UDF) or stored procedure.
- ▶ They cannot be referenced using private protocol when BIND option DBPROTOCOL(PRIVATE) is in effect.
- ▶ Multi-CEC parallelism is disabled for any query containing a declared temporary table.
- ▶ Dynamic statement caching is not supported for any statement containing a declared temporary table.
- ▶ ODBC and JDBC functions such as SQLTables and SQLColumns cannot be used, as the information required does not exist in the catalog.
- ▶ Thread reuse with declared temporary tables is allowed for CICS and IMS when the rows are implicitly deleted at commit time (ON COMMIT DELETE ROWS) or the declared temporary table is dropped before the COMMIT. Thread reuse is also possible for DDF pool threads but only if the table is explicitly dropped before committing (irrespective of the ON COMMIT PRESERVE/DELETE ROWS option).
- ▶ Triggers cannot be defined on declared temporary tables.

- ▶ Currently, declared temporary tables cannot be used within the body of a trigger. However, a trigger can call a stored procedure or UDF that refers to a declared temporary table.

The following statements are not allowed against a declared temporary table:

- ▶ CREATE VIEW
- ▶ ALTER TABLE
- ▶ ALTER INDEX
- ▶ RENAME TABLE
- ▶ LOCK TABLE
- ▶ GRANT/REVOKE table privileges
- ▶ CREATE ALIAS
- ▶ CREATE SYNONYM
- ▶ CREATE TRIGGER
- ▶ LABEL ON/COMMENT ON



Savepoints

In this chapter, we discuss how savepoints can be used to create points of consistency within a logical unit of work.

8.1 What is a savepoint?

A savepoint is a named entity that represents the state of data at a particular point in time within a logical unit of work. Savepoints can be set and released. It is possible to roll back the data to the state that the named savepoint represents. A rollback resets all savepoints which were taken after the savepoint we rolled back to.

Data (DML) and schema (DDL) changes made by the transaction after a savepoint is set can be rolled back to the savepoint, as application logic requires, without affecting the overall outcome of the transaction. There is no limit to the number of savepoints that can be set. The scope of a savepoint is the DBMS on which it was set.

8.2 Why to use savepoints

Savepoints enable the coding of contingency or what-if logic and can be useful for programs with sophisticated error recovery or to undo stored procedure updates when an error is detected and only the work done in stored procedure should be rolled back. The overhead of maintaining a savepoint is small (the measurements performed show that the cost of taking a savepoint is equivalent to a simple fetch). Release a savepoint after it is no longer feasible from an application logic perspective to roll back to that savepoint.

A good example for using savepoints is when making flight reservations. John Davenport from Australia is going on vacation in Denmark. He asks a travel agent to book the flights with maximum 4 legs (3 stops) both ways. He leaves from Alice Springs, Australia. He has to fly to Melbourne. From Melbourne he can either fly to Singapore and from there to Copenhagen, or he can fly via Kuala Lumpur and Amsterdam to Copenhagen. Figure 8-1 shows the possible flying routes from Alice Springs to Copenhagen.

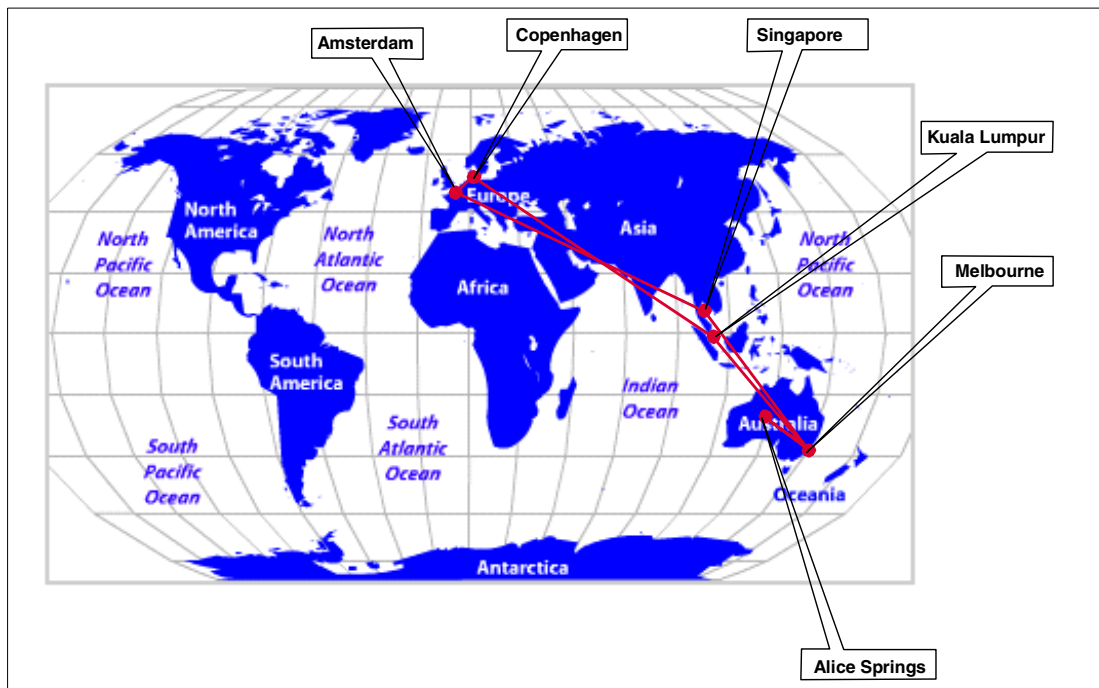


Figure 8-1 Travel reservation savepoint sample itinerary

- ▶ First we make the flight reservation to Melbourne and take a savepoint called **FIRSTSTOP**.
- ▶ Then we make the flight reservation to Singapore and take a savepoint called **SECONDSTOP**.
- ▶ Then we find out that there are no seats from Singapore to Copenhagen.
- ▶ We **ROLLBACK TO SAVEPOINT FIRSTSTOP**, since we do not want to lose the reservation to Melbourne.
- ▶ Then we make the reservation to Kuala Lumpur and take a savepoint called **SECONDSTOP**.
- ▶ And to Amsterdam with savepoint **THIRDSTOP**.
- ▶ And to Copenhagen with savepoint **DESTINATION**. Now that we are at our destination and have not used more than 3 stops, we can release all savepoints except **DESTINATION**.
- ▶ Then we make the return reservation to Singapore with a savepoint called **FIRSTSTOP**.
- ▶ There is no seats from Singapore to Melbourne and we need to **ROLLBACK TO SAVEPOINT DESTINATION** that is Copenhagen. Rolling back also releases the **FIRSTSTOP** savepoint.
- ▶ Then we make the reservation from Copenhagen to Amsterdam with a savepoint called **FIRSTSTOP**.
- ▶ And to Kuala Lumpur with savepoint **SECONDSTOP**.
- ▶ And to Melbourne with savepoint **THIRDSTOP**.
- ▶ If the reservation from Melbourne to Alice Springs fails we want to **ROLLBACK** the whole reservation, that is to the beginning of the logical unit of work. If we can find a seat to Alice Springs and the number of stops is no more than 3 (as in our case), we can **COMMIT** the UOW.

This example shows that there is sometimes a need to additional points in time to roll back to. They do not change the behavior - nor the need - to **COMMIT**.

Important: Savepoints are not a substitute for **COMMITs**.

8.3 Savepoint characteristics

The **SAVEPOINT** statement is used to set a savepoint. After executing such a statement, you should check the SQL return code to verify that the savepoint was set. It is also a good practice to choose a meaningful name (maximum length 128 bytes) for the savepoint.

The **SAVEPOINT** related statements (**SAVEPOINT**, **ROLLBACK TO SAVEPOINT** and **RELEASE SAVEPOINT**) can be issued from an application program or from a stored procedure that is defined as **MODIFIES SQL DATA**. These statements cannot be issued while executing under an external user-defined function or trigger, for example, a stored procedure that was invoked by a trigger. If you try, an **SQLCODE -20111** (cannot issue savepoint, release savepoint, rollback to savepoint from a trigger or from a user-defined function or from a global transaction) is returned to the application.

For the complete syntax of the **SAVEPOINT** statement, refer to the *DB2 UDB for OS/390 Version 7 SQL Reference*, SC26-9944. When you issue the **SAVEPOINT** statement, DB2 writes an external savepoint log record.

The **UNIQUE** clause is optional and specifies that the application program cannot activate a savepoint with the same name as an already active savepoint with the unit of recovery. If you plan to use a **UNIQUE** savepoint in a loop, and you do not release or rollback that savepoint in the loop prior to its reuse, you get an error:

```
SQLCODE -881: A SAVEPOINT WITH NAME savepoint-name ALREADY EXISTS, BUT THIS
          SAVEPOINT NAME CANNOT BE REUSED
SQLSTATE: 3B501
```

Omitting **UNIQUE** indicates that the application can reuse this savepoint name within the unit of recovery. If a savepoint with the same name already exists within the unit of recovery and the savepoint was not created with the **UNIQUE** option, the old (existing) savepoint is destroyed and a new savepoint is created. This is different than using the **RELEASE SAVEPOINT** statement, which releases the named savepoint and all subsequently established savepoints. Rollback to released savepoints is not possible.

Application logic determines whether the savepoint name needs to be or can be reused as the application progresses, or whether the savepoint name needs to denote a unique milestone. Specify the optional **UNIQUE** clause on the **SAVEPOINT** statement when you do not intend to reuse the name without first releasing the savepoint. This prevents an invoked program from accidentally reusing the name.

Tip: You can reuse a savepoint that has been specified as **UNIQUE** as long as the prior savepoint with the same name has been released (through the use of a **ROLLBACK** or a **RELEASE SAVEPOINT**) prior to attempting to reuse it.

The **ON ROLLBACK RETAIN CURSORS** clause is mandatory and specifies that any cursors that are opened after the savepoint is set are not tracked, and thus, are not closed upon rollback to the savepoint. Although these cursors remain open after rollback to the savepoint, they might not be usable. For example, if rolling back to the savepoint causes the insertion of a row upon which the cursor is positioned to be rolled back, using the cursor to update or delete the row results in an error:

```
SQLCODE -508: THE CURSOR IDENTIFIED IN THE UPDATE OR DELETE STATEMENT IS NOT
          POSITIONED ON A ROW
SQLSTATE: 24504
```

With scrollable cursors (see “Update and delete holes” on page 170 for more details), you would get a different error:

```
SQLCODE -222: AN UPDATE OR DELETE WAS ATTEMPTED AGAINST A HOLE USING cursor-name
SQLSTATE: 24510
```

The **ON ROLLBACK RETAIN LOCKS** clause specifies that any locks that are acquired after the savepoint is set are not tracked and are not released upon rollback to the savepoint. If you do not specify this clause, it is implied (this is the default and at the present time, there is no other option for locks).

In Example 8-1, we show how to set a unique savepoint named **START_OVER**.

Example 8-1 Setting up a savepoint

```
EXEC SQL SAVEPOINT START_OVER
          UNIQUE
          ON ROLLBACK RETAIN CURSORS
          ON ROLLBACK RETAIN LOCKS ;
```

The **ROLLBACK** statement *with* the **TO SAVEPOINT** clause is used to restore to a savepoint, that is, undo data and schema changes (excluding changes to created temporary tables) made after the savepoint was set. Changes made to created temporary tables are not logged and are not backed out; a warning is issued instead. The same warning is also issued when a created temporary table is changed and there is an active savepoint. The warning issued is:

```
SQLCODE +883: ROLLBACK TO SAVEPOINT OCCURED WHEN THERE WERE OPERATIONS THAT CANNOT
              BE UNDONE, OR AN OPERATION THAT CANNOT BE UNDONE OCCURRED WHEN THERE WAS
              A SAVEPOINT OUTSTANDING
SQLSTATE: 01640
```

Any updates outside the local DBMS, such as remote DB2s, VSAM, CICS, and IMS, are not backed out upon rollback to the savepoint, not even when under the control of RRS. Any cursors that are opened after the savepoint is set are not closed when a rollback to the savepoint is issued. Changes in cursor positioning are not backed out upon rollback to the savepoint. Any locks that are acquired after the savepoint is set are not released upon rollback to the savepoint. Any savepoints that are set after the one to which you roll back to, are released. The savepoint to which the rollback is performed is not released. If a savepoint name is not specified, the rollback is to the last active savepoint. If no savepoint is active, an error occurs:

```
SQLCODE -880: SAVEPOINT savepoint-name DOES NOT EXIST OR IS INVALID IN THIS CONTEXT
SQLSTATE: 3B001
or
SQLCODE -882: SAVEPOINT DOES NOT EXIST
SQLSTATE: 3B502
```

Rolling back a savepoint has no effects on created temporary tables because there is no logging for CTTs. Changes to declared temporary tables on the other hand, can be 'safeguarded' or undone by rolling back to a savepoint. For more information on declared temporary tables, see 7.3, "Declared temporary tables" on page 88.

The **ROLLBACK** statement *without* the **TO SAVEPOINT** clause (this is the normal SQL ROLLBACK statement) rolls back the entire unit of recovery. All savepoints set within the unit of recovery are released.

The **RELEASE SAVEPOINT** statement is used to release a savepoint and any subsequently established savepoints. The syntax of the **COMMIT** statement is unchanged. **COMMIT** releases all savepoints that were set within the unit of recovery.

8.4 Remote connections

While there are outstanding savepoints, access to a remote DBMS with DB2 private protocol access or with DRDA access using aliases or three-part names is not allowed. For example, if there is a savepoint set at location A, location A cannot connect to location B by either of these two ways to access data. This is due to the consideration that the programmer using three-part names or aliases is normally unaware of the remote sites involved. However, you can access a remote DBMS with DB2 private protocol access or with DRDA access using aliases or three-part names when there are no savepoints outstanding at the current location. Once at the remote site, you may set savepoints but the local site is not aware of them.

DRDA access using a **CONNECT** statement is allowed; however, the savepoints are local to their site and do not cross an explicit **CONNECT**. For example, location A can connect to location B, but the savepoint set at A is unknown to location B and does not cover any operations performed at location B. A savepoint set prior to a **CONNECT** is known only at the local site, and a savepoint set after a connect is known only at the remote site. Consequently, application-directed connections are expected to manage the processing at the alternate site.

We recommend that you code the `RELEASE SAVEPOINT svptname` statement to release savepoints that are no longer required for clarity, and to re-enable the use of three part name remote connections.

8.5 Savepoint restrictions

The following are the restriction when using savepoints:

- ▶ You cannot use savepoints in global transactions.
- ▶ You cannot use savepoints in triggers.
- ▶ You cannot use savepoints in user-defined functions.
- ▶ You cannot use savepoints in a stored procedure that is called from a trigger or UDF.
- ▶ Rolling back to a savepoint does not undo changes to created temporary tables.
- ▶ Rolling back to a savepoint does not release any locks.
- ▶ Cursor position does not change when you roll back to a savepoint.
- ▶ Savepoint names cannot be specified with host variables.



Unique column identification

Since day one, in order to guarantee a unique value for a column in DB2 for z/OS, you had to create a unique index on that column. Although the index can also be used to get a good performing query, it still was a burden on the system having to maintain it (changes to the indexed column have to be made to the index as well).

Recently DB2 has introduced two new concepts that can guarantee unique column values without having to create an index. In addition, you can eliminate the application coding that was implemented to assign unique column values for those columns.

In this chapter we introduce:

- ▶ IDENTITY columns
- ▶ ROWIDs

We evaluate their characteristics, usability and especially their ability to really replace a unique index and application generated unique numbers.

9.1 Identity columns

In this section, we describe identity columns and several ways to use them when designing databases. Identity columns offer us a new possibility to guarantee uniqueness of a column and to be able to automatically generate the value. We also discuss identity columns in the context of declared temporary tables in “Creating a declared temporary table” on page 90.

9.1.1 What is an identity column?

An identity column is a numeric column, either SMALLINT, INTEGER, or DECIMAL with a scale of zero, or a user defined distinct type based on any of these data types, which is UNIQUE and NOT NULL by definition. The support for identity columns provides a way to have DB2 automatically generate unique, sequential, and recoverable values for the column defined as the identity column for each row in the table.

A table can have only one column defined AS IDENTITY.

9.1.2 When to use identity columns

Normally, logical database design should find good unique keys for all tables during the design process. However, during physical design you may need to add tables to satisfy additional requirements (like tables to log certain actions, intermediate results tables, and so on) and resolve issues of long multi-column keys as primary keys. In some of these cases, there may not be a good natural key to be used, therefore people often resort to using an artificial key.

When you have a multi-column primary key and the table has several dependent tables, you have to ‘copy’ many columns to the dependent tables to build the foreign keys. This makes the keys very long and you have to code many predicates when joining the tables. Having too many predicates can increase the chance that you make a mistake and sometimes forget a join predicate. Having many columns in the index also makes the index grow quite large. Instead of using the long key as the primary and foreign key, you can use an artificial unique identifier for the parent table and use that generated value as a primary key and foreign key for the dependent table.

Another use for identity columns is when you just need a generated unique column for a table. If we do not need to reference the rows by this column, then there is no need to even create an index for it, and uniqueness is guaranteed by the system for generated values (when you use GENERATED ALWAYS, when you define the column as an IDENTITY column).

Important: Please note that it is possible to have duplicate values for an identity column if you specify the CYCLE option (introduced in V7) as one of the keywords when you define the identity column.

Another reason to use identity columns is so you don’t have to keep track of ever increasing numbers used as unique keys, for example, order number, employee number, and so on in your applications. DB2 takes over the responsibility of keeping track of the last number used so this complexity can be removed from your application.

9.1.3 Identity column characteristics

When you define an identity column, you can specify the start value of the identity column and the increment to specify the interval between two consecutive values. Both numbers can be negative or positive, default is one. Naturally, the increment cannot be zero.

For better performance, DB2 can keep some preallocated numbers in memory. The default is to cache 20 numbers, but can be defined in the *CACHE integer* clause. The minimum value is 2. If you do not want or need caching, specify *NO CACHE*. If DB2 fails, you lose the numbers which are cached but not used yet. So the values may have some gaps for this reason. Other reasons to end up with gaps can be found in 9.1.9, “Application design considerations” on page 111.

9.1.4 Creating a table with an identity column

In Example 9-1, we show the DDL to create the table *TBMEMBERS* using an identity column. We want DB2 to always generate the values for the member number (*MEMBNO* identity) column. The default for the starting value is +1. We chose a different number to start with. In this case we want to start with the number 1000. We can have DB2 increase or decrease the generated values from the starting point, by an increment we define. The default for the increment is +1, we have chosen to increment by +2. We just started this club and there is a lot of interest in it and at this point in time we have a lot of members joining, so to help with insert performance, we chose to pre-allocate 40 values instead of the default of 20. We specified a *MAXVALUE* of 9999 and this is the largest member number we wish to assign. We specified *MINVALUE* of 1007. *MINVALUE* is not necessarily the smallest member number you wish to assign, it is the minimum value to start using once you have cycled to the end of the allowable numbers. Cycling means that we have reached the *MAXVALUE* and we start assigning numbers again from *MINVALUE*. We also wanted the founding members of the club, the first 4 member numbers, to be the lowest member numbers and to always be unique numbers, that is why we specified a *START WITH* amount smaller than *MINVALUE*. Let’s take a look at how the *MEMBNO* values are assigned:

1. The first number assigned is 1000 (*START WITH* number), and the increment is +2 so the next number is 1002, 1004, 1006, and so on. So we assign all the even numbers.
2. *MEMBNO* is incremented by +2 until we reach our *MAXVALUE* of 9999. The last even number that is generated is 9998.

At this point we have assigned nothing but even numbered member numbers.

3. When we reached *MAXVALUE* and because *CYCLE* was also specified, the system continues assigning numbers starting from our *MINVALUE* of 1007. Now the odd numbers (1007, 1009, 1011, 1013, 1015, and so on) are assigned since our increment is still +2. Note that the odd numbers between 1000 and 1006 were not assigned.

(If we had not specified *CYCLE* and reached *MAXVALUE*, our inserts would start receiving an *SQLCODE* -359, no new numbers would be assigned, and our inserts would fail.)

4. When we reach our *MAXVALUE* of 9999 again, DB2 continues to cycle starting from *MINVALUE* of 1007 and the odd numbers are assigned again causing us to have duplicate member numbers in our database.

In this case, only after we reach *MAXVALUE* 9999 the second time, we start to duplicate the odd member numbers.

Example 9-1 IDENTITY column for member number

```
CREATE TABLE TBMEMBERS
  (MEMBNO DECIMAL(5, 0) GENERATED ALWAYS AS IDENTITY
   (START WITH 1000,
    INCREMENT BY +2,
    CACHE 40
    CYCLE,
    MAXVALUE 9999,
    MINVALUE 1007))
```

```

,NAME CHAR(30) NOT NULL WITH DEFAULT
,INCOME DECIMAL(15,2) NOT NULL WITH DEFAULT
,DONATION SMALLINT NOT NULL WITH DEFAULT ;

```

This is the order in which member numbers get assigned:

```

1000
1002
1004
1006
...
9994
9996
9998
1007 <-----
1009
1011
1013
...
9995
9997
9999 -----

```

once we reach MAXVALUE 9999, we CYCLE back to MINVALUE and begin assigning numbers from there

The fact that you can create a table with an IDENTITY column as specified in Example 9-1, does not mean that you should. More often than not, it is very important that you avoid duplicates, so you do not specify the CYCLE keyword. Also, you should take into account how large the number may get over a long period of time and provide for a much larger MAXVALUE. If you reach MAXVALUE and you don't have CYCLE specified, you are not able to insert additional rows, you have to drop the table and recreate it with a larger MAXVALUE and reload the rows.

The GENERATED ALWAYS attribute of the identity column definition is treated in more detail in section 9.1.5, "How to populate an identity column" .

When a table is being created LIKE another table that contains an identity column, a new option on the LIKE clause, INCLUDING IDENTITY COLUMN ATTRIBUTES, can be used to specify that all the identity column attributes are to be inherited by the new table. If INCLUDING IDENTITY COLUMN ATTRIBUTES is omitted, the new table only inherits the data type of the identity column and none of the other column attributes. You cannot create a table LIKE a view and specify the INCLUDING IDENTITY COLUMN ATTRIBUTES keywords.

In Example 9-2, we specify that T2 should inherit all of the identity column attributes from T1 by specifying the INCLUDING IDENTITY COLUMN ATTRIBUTES clause.

Example 9-2 Copying identity column attributes with the LIKE clause

```

CREATE TABLE T2 LIKE T1
INCLUDING IDENTITY COLUMN ATTRIBUTES

```

9.1.5 How to populate an identity column

Identity columns come in two flavours and they behave in quite differently. So before deciding which type you will use, you should thoroughly consider the implications of the two options.

Let's discuss the various ways that an identity column is populated.

When inserting or updating

The way an identity column is populated using an INSERT or UPDATE statement depends on the column definition. The column can be defined as GENERATED BY DEFAULT or GENERATED ALWAYS:

GENERATED BY DEFAULT

When inserting a row you can either provide the value for the identity column or let DB2 generate the value for you (by not specifying a value or using the DEFAULT keyword in the VALUES clause).

Values can be changed using an UPDATE statement.

DB2 does not guarantee the uniqueness of a 'GENERATED BY DEFAULT' identity column value among all the column values, but only among the set of values it previously generated. To guarantee uniqueness, you have to create a unique index on the identity column.

You must code re-try logic in your application in order to handle the possibility of duplicates.

GENERATED ALWAYS

DB2 always generates a value for the column when a row is inserted into the table, and DB2 guarantees that all column values are unique unless you use the CYCLE keyword at column definition time.

If an INSERT or UPDATE statement is issued in an attempt to provide an explicit value for the identity column, DB2 returns an error and the statement fails. However, the keyword DEFAULT can be used in the VALUES clause of the insert statement in order to have DB2 generate the value. Using the DEFAULT keyword is especially useful in dynamic SQL, since it eliminates the need to name all the columns of a table in an INSERT statement because there is a column we are not allowed to provide a value for. In statically bound programs, you should always name the columns in order to avoid a table change from impacting your program. Therefore, it is also a good idea to use the DEFAULT in the values list of static programs.

When adding an identity column to an existing table

An identity column can be added to a table using the ALTER TABLE statement. When you add an identity column to a table that is not empty, the table space that contains the table is placed in REORP (reorg pending) status. When the REORG utility is subsequently run, DB2 generates a unique identity column value for each row and then removes the REORP status.

Note: If you add an identity column to a table and run a point in time recovery for that table space to an RBA prior to the time the REORG on the table populated the column, the table space is again placed to REORP status.

When loading a table

An identity column that is defined as GENERATED ALWAYS, cannot be included in a LOAD utility control statement *field-specification* list or be implied by a LOAD FORMAT UNLOAD, a LOAD with no *field-specification* list. Such a request is rejected by the LOAD utility with an error.

However, if the (un)load file contains identity column information, you can use the DSN_IDENTITY for the identity column in the LOAD control cards and specify the IGNOREFIELDS keyword. The values in the (un)load file are ignored and a new value is generated by the LOAD utility for the identity column.

Identity columns defined as GENERATED BY DEFAULT can be loaded like any other column. That is, you can load data into the identity column. To do this, you specify the column name of the identity column in your load control cards just like the rest of the columns.

Even when the identity column is defined as GENERATED BY DEFAULT and the (un)load file contains identity column values, if you prefer to have DB2 generate (or re-generate) values for the column, then you can use the name of DSN_IDENTITY for the identity column in the load control cards and specify the IGNOREFIELDS keyword.

For more information on this, refer to *DB2 UDB for OS/390 and z/OS Utility Guide and Reference*, SC26-9945.

Important: There is no means by which you can load existing values into an identity column that is GENERATED ALWAYS. If you are reloading data to a table with such a column, you must first drop and recreate the table and make the identity column GENERATED BY DEFAULT. There is no way to convert from GENERATED ALWAYS to GENERATED BY DEFAULT or visa versa.

When inserting with a select from another table

Example 9-3 shows how to insert with a select from another table. Column C1 is defined as an identity column in both tables T1 and T2. The OVERRIDING USER VALUE clause means that we override the identity column values coming from T1 with new identity column values that DB2 generates for the T2 table. The OVERRIDING USER VALUE clause can only be specified if the target table identity column (in this case the identity column for T2) is defined as GENERATED ALWAYS. Otherwise you receive an SQLCODE -109 (OVERRIDING USER VALUE CLAUSE IS NOT PERMITTED).

Another way to accomplish the same thing, no matter how the column is defined, is to explicitly specify all the columns, except for the identity column (C1), from T1 and T2.

We recommend that you do not use the OVERRIDING USER VALUE clause because if you ever have to change the identity column definition from GENERATED ALWAYS to GENERATED BY DEFAULT, then the BIND to pick up the new table would fail and it would require application changes in order to correct the problem.

The OVERRIDING USER VALUE clause can be used in a single row INSERT statement.

Example 9-3 Insert with select from another table

```
INSERT INTO T2
  OVERRIDING USER VALUE
  SELECT * FROM T1
```

A better (safer) way to code this is:

```
INSERT INTO T2
  (C2, C3, C4, C5, C6)
  SELECT C2, C3, C4, C5, C6 FROM T1      -- In this case, C1 is the identity
                                         -- column and C2, C3, C4, C5, C6
                                         -- are all the other columns of the
                                         -- table, note that C1 is not coded
                                         -- in either column list
```

9.1.6 How to retrieve an identity column value

It is often very interesting to know what number DB2 has assigned to an identity column; for example, when the identity column is the primary key in a parent table (TBORDERS) and you want on insert rows into a child table (TBORDERITEMS) that refer, via a foreign key, back to the parent. In order to insert the child rows, you have to know the generated identity column value for the parent table. For this purpose you can use the new built-in scalar function `IDENTITY_VAL_LOCAL()`.

This function returns the most recently assigned value for an identity column at the same nesting level. (A new nesting level is initiated any time a trigger, external UDF, or stored procedure is invoked.) The data type for the result of the `IDENTITY_VAL_LOCAL()` value function is always `DECIMAL(31,0)`, regardless of the actual data type of the identity column whose value is being returned.

The function has no input parameters. It is a non-deterministic function and belongs to schema `SYSIBM`. If the identity column is defined with a user-defined distinct type, this function must be casted for the UDT. See 4.5, "Using CAST functions" on page 48 for more details.

The `IDENTITY_VAL_LOCAL` function returns a null value in two cases. The first case is when you run the function before inserting any rows with identity columns at the current nesting level. The second case is when a `COMMIT` or `ROLLBACK` has been issued since the most recent `INSERT` statement that assigned a value.

You always get the last identity column value inserted at the current nesting level. So if you are inserting to many tables with identity columns, call the `IDENTITY_VAL_LOCAL` function immediately after each insert if you want to know the value of each identity column inserted. The result of the function is not affected if you insert to other tables which do not have identity columns or process other SQL that does not insert.

In Example 9-4, we show how to set host variable `:ID-MEMBNO` to the value that was assigned to the identity column `MEMBNO` when the first row was inserted into the table `TBMEMBER` (see Example 9-1). In this case, since it is our first insert into the `TBMEMBER` table, the value returned by the function is 1000.

Example 9-4 Retrieving an identity column value

```
INSERT INTO TBMEMBER
      (NAME, INCOME, DONATION)
VALUES ('Kate', 120000.00, 500);
```

```
SET :ID-MEMBNO = IDENTITY_VAL_LOCAL()
or
VALUES IDENTITY_VAL_LOCAL() INTO :ID-MEMBNO;
```

Note: the insert could also have been coded like this:

```
INSERT INTO TBMEMBER
      (MEMBNO, NAME, INCOME, DONATION)
VALUES (DEFAULT, 'Kate', 120000.00, 500) ;
```

or like this:

```
INSERT INTO TBMEMBER
      VALUES (DEFAULT,'Kate', 120000.00, 500) ;
```

9.1.7 Identity columns in a data sharing environment

The values generated by DB2 for a given identity column are unique across a data sharing group. When *CACHE integer* is specified in a data sharing environment, each member gets its own range of consecutive values to assign.

In Figure 9-1, we assume *CACHE 20* (the default) is used for an identity column and member DB2A has cached values 1-20 and member DB2B has cached values 21-40. If the application that inserts the first row into the table runs on member DB2B, the identity column value assigned to that row is 21. If the application that inserts the second row into the table runs on member DB2A, the identity column value assigned to that row is 1.

In a data sharing environment, there is a synchronous forced log write each time the counter is updated. If you do this once every 20 times instead of every insert (when *NO CACHE* is used), the overhead is significantly reduced.

Tip: Increasing the *CACHE* value above the default of 20 probably gives negligible benefits.

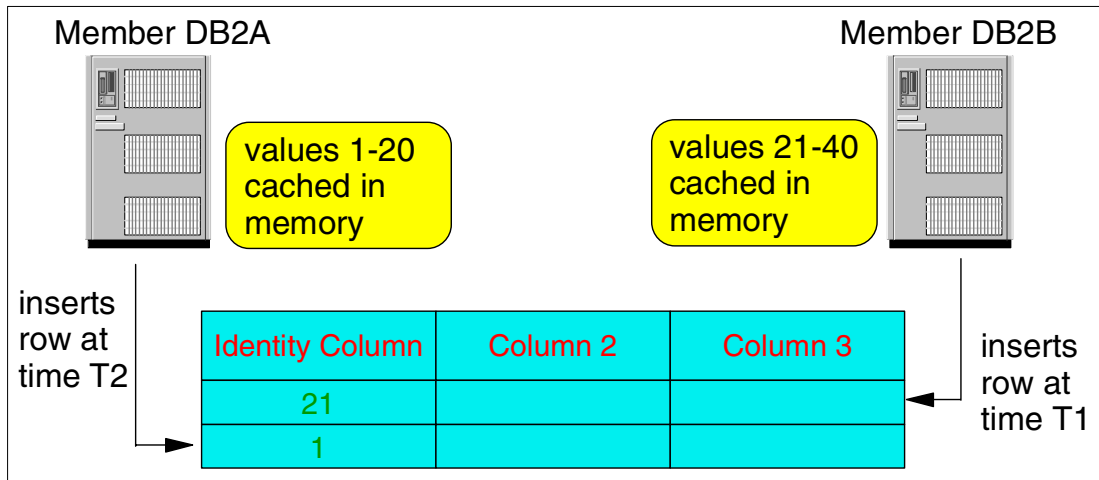


Figure 9-1 Identity column value assignment in a data sharing environment

9.1.8 Trying to overcome the identity column deficiencies

To overcome some of the problems associated with identity columns, you can try to combine the old 'highest key assigned table' technique and using identity column values generated by DB2 to provide you that unique number.

Every data table that you want to 'equip' with this technique uses two tables; A table with that contains the actual data and a table that has the identity column but no data. You use the table with the identity column to obtain the unique number and use that number to insert the real row in the actual data table. This way you separate the actual data from the (single row) table with the identity column.

This technique has the following advantages:

- ▶ The table with the actual data does not contain an identity column, so it does not suffer from all restrictions associated with them.

- ▶ If you ever have to reset the value of the identity column, you just drop the table with the identity column and recreate it. You might have to do this after loading additional data in a partition. (LOAD PART x is not allowed on a table with an identity column).
- ▶ By cutting the link between the actual 'sequence' number used in the data table and the table with the identity column that generates the 'sequence' number, you are in control of the numbers that will be assigned (at least you will be able to (re)set starting values). When both the data and the identity column are in the same table, DB2 is in control of the numbers that are assigned and this irrespective of the actual data in the table. Even if you for example, delete all rows from a table with an identity column, DB2 will not reset the MAXASSIGNEDVAL (highest value used so far).

In order to hide the complexity of using two tables instead of one, you can mask the process with a stored procedure or a UDF. The stored procedure or UDF can be implemented as follows:

1. Insert a row into the identity column table.
2. Retrieve its value back using the IDENTITY_VAL_LOCAL() function.
3. Delete that row from the identity column table since we no longer need it.
4. Insert the actual row using the retrieved value as a unique identifier.

The major advantage of this approach over the old style 'highest key assigned table' is that there is no locking problem. The row(s) in the identity column table are never updated. We only INSERT and DELETE rows. In case of a very high volume transaction workload, you might want to avoid deleting the row from the identity column table in the stored procedure or UDF because it is an extra SQL call. Instead you can have an asynchronous process running in the background that cleans up rows at a regular intervals. This does not cause any locking contention either, since INSERTS take a new page in case the candidate page is locked.

9.1.9 Application design considerations

Be aware that DB2's identity column implementation allows for gaps to be created in the assignment of identity column values. Gaps can occur due to the following reasons:

- ▶ If an inserting agent rolls back while other agents are also inserting to the table with the identity column.
- ▶ When the DB2 subsystem terminates abnormally before all the cached values have been assigned.
- ▶ When the increment amount is greater than 1 or less than -1.
- ▶ When you specify GENERATED BY DEFAULT and you specify values.

Tip: If your application cannot tolerate gaps for the unique keys, then identity columns may not be a viable solution for you.

We recommend that you use the default caching value (CACHE 20) in a data sharing environment to reduce serialization overhead. Be aware that in a data sharing environment with cached identity column values, it is possible for identity column values to be assigned out of sequence.

Tip: If you are in a data sharing environment, and your application does not tolerate for identity column values to be assigned out of sequence, then you should use the NO CACHE option. Be aware that this can have negative performance implications on your application and can reduce the throughput of insert transactions.

Be aware of running out of values. If you created an identity column as SMALLINT, the maximum value that can be inserted is 32767. When you reach this value (depending on the parameters you used to define your identity column) you may either start getting duplicate values assigned (if you specified CYCLE) or your inserts fail. If you unload, drop, recreate and (re)load the table, you may end up having different identity column values than you had before. This is very dangerous, especially if you have RI or if you saved the identity column values in other tables.

This is also very important when you have to change the table design, like eliminating a column. A lot of these design changes require the table to be dropped and recreated. You probably want the same identity column values to be used after the drop/create operation than before, for example, because of RI relationships with other tables. This forces you to define the identity column as GENERATED BY DEFAULT. In order to guarantee uniqueness of the values, which is probably why you wanted the identity column in the first place, a unique index is required.

Be aware that when you specify the keyword CYCLE, once you reach MAXVALUE, you start the numbering from the MINVALUE again. DB2 does not warn you when reaching the MAXVALUE. When using CYCLE, to be sure that the values are unique, you should create a unique index on the identity column, even when the identity column is defined as GENERATED ALWAYS.

Important: You cannot alter any of the definitions for an identity column. Be sure you consider every possible situation of how the data is or may be used before adding an identity column to a table.

When the identity column is defined as GENERATED ALWAYS and is the primary key in an RI structure, loading data into the RI structure is extremely complicated. Since the values for the identity column are not known until they are loaded (because of the GENERATED ALWAYS attribute) you have to generate the foreign keys in the dependent tables based on these values since they have to match the primary key. This can be another reason for not using GENERATED ALWAYS.

9.1.10 Identity column restrictions

In addition to not providing a gapless series of values and the fact that the identity column attributes are non-alterable, identity columns have some other restrictions.

An identity column cannot be defined with the FIELDPROC clause or the WITH DEFAULT clause. Nor can an EDITPROC be specified for a table that has an identity column.

Also you cannot load a partition (LOAD INTO PART x) when an identity column is part of the partitioning index.

9.2 ROWID and direct row access

In this section, we describe and discuss how to use ROWIDs and where not to use them. If your tables have LOBs, then DB2 uses ROWIDs to access the LOB data stored in the auxiliary table. If your table do not have LOBS, you can still use ROWIDs.

There are two aspects to ROWIDs besides its use to retrieve LOB data:

- ▶ The first one is the fact that a ROWID can be a unique random number generated by the DBMS. This can look like a very appealing option to solve many design issues.

- ▶ The second aspect of using ROWID columns is that they can be used for a special type of access path to the data called direct row access.

We try to explain that there are some advantages when using ROWIDs, but there is also a maintenance cost (REORGs cause the external representation of the ROWIDs to change) associated with using ROWIDs and it is easy to use them the wrong way in application programs.

9.2.1 What is a ROWID?

A ROWID is a new data type. A column or a host variable can have a ROWID data type. A ROWID is a value that uniquely identifies a row in the table. When a row is inserted into the table, DB2 generates a value for the ROWID column unless one is supplied. If a value is supplied, it must be a valid row ID value that was previously generated by DB2 and the column must be defined as GENERATED BY DEFAULT.

You must use have a column with a ROWID data type in a table that contains a LOB column. The ROWID column is stored in the base table and is used to lookup the actual LOB data in the LOB table space. However, a ROWID is not the same as a RID.

Although ROWIDs were designed to be used with LOBs, they can also be used in 'regular' tables that don't contain LOB columns. In that case, a ROWID column can be used in a WHERE predicate to enable queries to navigate directly to a row in the table.

Normally, DB2 generates the value (as indicated by GENERATED ALWAYS clause) for the ROWID column. ROWIDs are random and unique, therefore at first glance they look like ideal candidates to be used as a partitioning key of a partitioned table space if you need to spread rows randomly across the partitions.

9.2.2 ROWID implementation and maintenance

A ROWID column must exist in the base table before a LOB column can be defined. If you forget the ROWID column, you receive the following error:

```
SQLCODE = -770, ERROR:  TABLE TEST CANNOT HAVE A LOB COLUMN UNLESS IT
                        ALSO HAS A ROWID COLUMN
```

There is one ROWID column per base table, so, all LOB columns in a row have the same ROWID. The ROWID column is a unique identifier for each row of the table and the basis of the key for indexing the auxiliary table(s).

Example 9-5 shows a table that contains LOB columns, and therefore, a ROWID column is required. Column IDCOL is defined with the ROWID data type. Notice that in order to be able to work with the table that contains a LOB column, you also need to create the LOB table space, the auxiliary table and the auxiliary index.

Example 9-5 ROWID column

```
CREATE TABLESPACE TSLITERA IN DB246300 #
CREATE TABLE SC246300.LITERATURE
  ( TITLE          CHAR(25)
    ,IDCOL          ROWID NOT NULL GENERATED ALWAYS
    ,MOVLENGTH      INTEGER
    ,LOBMVIE        BLOB(2K)
    ,LOBBOOK        CLOB(10K) )
  IN DB246300.TSLITERA #
CREATE LOB TABLESPACE TSL0B1 IN DB246300 #
CREATE AUX TABLE SC246300.LOBMVIE_ATAB
```

```

        IN DB246300.TSLOB1
        STORES SC246300.LITERATURE
        COLUMN LOBMOVIE#
CREATE INDEX DB246300.AXLOB1
    ON SC246300.LOBMOVIE_ATAB #
CREATE LOB TABLESPACE TSLOB2 IN DB246300 #
CREATE AUX TABLE SC246300.LOBBOOK_ATAB
    IN DB246300.TSLOB2
    STORES SC246300.LITERATURE
    COLUMN LOBBOOK#
CREATE INDEX DB246300.AXLOB2
    ON SC246300.LOBBOOK_ATAB #

INSERT INTO SC246300.LITERATURE (TITLE) VALUES ('DON QUIJOTE DE LA MANCHA')#
INSERT INTO SC246300.LITERATURE (TITLE) VALUES ('MACBETH') ;

SELECT TITLE, IDCOL FROM SC246300.LITERATURE #

```

TITLE	IDCOL
MACBETH	CDC8A0A420E6D44F260401D37018010000000000203
DON QUIJOTE DE LA MANCHA	7B80A0A420E6D64F260401D37018010000000000204

As mentioned in the introduction of this section, the use of a ROWID column is not restricted to tables that have LOB columns defined. You can have a ROWID column in a table that does not have any LOB columns defined.

Selecting a ROWID results in 22 logical bytes, but it is stored physically as 17 bytes. Example 9-6 shows how to code a ROWID function in an SQL statement.

Example 9-6 SELECTing based on ROWIDs

```

SELECT TITLE, IDCOL
FROM SC246300.LITERATURE
WHERE IDCOL=ROWID(X'7B80A0A420E6D64F260401D37018010000000000204')

```

TITLE	IDCOL
DON QUIJOTE DE LA MANCHA	7B80A0A420E6D64F260401D37018010000000000204

The value generated by DB2 is essentially a random number. If data is being propagated or copied from one table to another, the ROWID value is allowed to appear in the VALUES clause of an INSERT statement. In this case, the ROWID column has to be defined as GENERATED BY DEFAULT and to ensure uniqueness, a unique index on the ROWID column must be created.

Note: Applications are not permitted to UPDATE the ROWID column.

9.2.3 How ROWIDs are generated

There are two ways to specify how to generate ROWIDs:

GENERATED ALWAYS

- ▶ Use this option unless copying data containing a ROWID column from another table.
- ▶ ROWID column values cannot appear in the VALUES clause of an INSERT statement.
- ▶ No index is required to guarantee uniqueness of the values.

GENERATED BY DEFAULT (usually you do not want this option)

- ▶ DB2 generates a ROWID value if none is specified in the VALUES clause.
- ▶ Intended only for copying data from tables containing ROWID columns. DB2 performs validity checking at INSERT time to verify whether the supplied value meets the criteria to qualify for a ROWID column.
- ▶ Requires a unique index on the ROWID column to guarantee uniqueness of the ROWID values.

Note: Users should never try to generate ROWID values.

In Example 9-7, we show how you can copy data from one table to another. If you have defined the ROWID as GENERATED ALWAYS, then DB2 generates a new ROWID for each INSERTed row (the ROWID columns cannot be copied or propagated).

Note that the LITERATURE_GA table does no longer contain the LOB columns; this shows that you can also create a table with a ROWID column without any LOB columns. Note also that in that case you no longer need the LOB table space, auxiliary table and index (they are only present when you have LOB columns defined).

Example 9-7 Copying data to a table with GENERATED ALWAYS ROWID via subselect

```
CREATE TABLE SC246300.LITERATURE_GA
  (TITLE          CHAR(30)
  ,IDCOL          ROWID NOT NULL GENERATED ALWAYS
  ,MOVLENGTH      INTEGER
  )
IN DB246300.TS246300 ;

INSERT INTO SC246300.LITERATURE_GA
  (TITLE
  ,MOVLENGTH
  )
SELECT
  TITLE
  ,MOVLENGTH
  FROM SC246300.LITERATURE;

-- The ROWID column (IDCOL) is not specified because you can not insert
-- values in a GENERATED ALWAYS column.

SELECT TITLE, IDCOL
FROM SC246300.LITERATURE
-----+-----+-----+-----+-----+-----+-----+-----+-----
TITLE                                     IDCOL
-----+-----+-----+-----+-----+-----+-----+-----
```

```

MACBETH          CDC8A0A420E6D44F260401D37018010000000000203
DON QUIJOTE DE LA MANCHA  7B80A0A420E6D64F260401D37018010000000000204

```

```

SELECT TITLE, IDCOL
FROM SC246300.LITERATURE_GA;

```

```

-----+-----+-----+-----+-----+-----+-----+-----+-----+
TITLE          IDCOL
-----+-----+-----+-----+-----+-----+-----+-----+-----+
MACBETH        88DDF53DA0E6D808260401D37010010000000000207
DON QUIJOTE DE LA MANCHA  D1DDF53DA0E6D818260401D37010010000000000208

```

Notice that when copying data to a table with a GENERATED ALWAYS ROWID column via a subselect, the ROWIDs are completely different. New values are generated when the rows are inserted into the LITERATURE_GA table. The ROWID column is *not* selected because you cannot insert values in a GENERATED ALWAYS column.

If GENERATED BY DEFAULT was specified, you can supply a ROWID value for the INSERTed rows.

Example 9-8 Copying data to a table with GENERATED BY DEFAULT ROWID via subselect

```

CREATE TABLE SC246300.LITERATURE_GDEF
(TITLE CHAR(30)
, IDCOL ROWID NOT NULL GENERATED BY DEFAULT
, MOVLENGTH INTEGER
)
IN DB246300.TS246300 ;

CREATE UNIQUE INDEX DD ON SC246300.LITERATURE_GDEF (IDCOL) ;
-- This index is required for tables with GENERATED BY DEFAULT ROWID
-- columns in order to guarantee uniqueness. You receive an
-- SQLCODE -540 if the index does not exist.

```

```

INSERT INTO SC246300.LITERATURE_GDEF
(TITLE
, IDCOL)
SELECT
TITLE
, IDCOL
FROM SC246300.LITERATURE;

```

```

SELECT TITLE, IDCOL
FROM SC246300.LITERATURE

```

```

-----+-----+-----+-----+-----+-----+-----+-----+-----+
TITLE          IDCOL
-----+-----+-----+-----+-----+-----+-----+-----+-----+
MACBETH        CDC8A0A420E6D44F260401D37018010000000000203
DON QUIJOTE DE LA MANCHA  7B80A0A420E6D64F260401D37018010000000000204

```

```

SELECT TITLE, IDCOL
FROM SC246300.LITERATURE_GDEF

```

```

-----+-----+-----+-----+-----+-----+-----+-----+-----+
TITLE          IDCOL
-----+-----+-----+-----+-----+-----+-----+-----+-----+
MACBETH        CDC8A0A420E6D44F260401D37018010000000000209
DON QUIJOTE DE LA MANCHA  7B80A0A420E6D64F260401D3701801000000000020A

```

Notice that the ROWID values in the output (external representation) are similar when copying data on a table with GENERATED BY DEFAULT ROWID via subselect. The only difference is the last bytes that include the row's physical location. Actually the ROWID values that are stored physically on DASD (internal representation-highlighted part in the example) are the same in both tables.

9.2.4 Casting to a ROWID data type

You can cast from a CHAR expression, VARCHAR expression or HEX literal string to a ROWID data type using the CAST function or the ROWID function.

The only column function that allows the ROWID data type is COUNT. The ROWID data type column is allowed in the BLOB, CAST, CHAR, CLOB, HEX, IFNULL, LENGTH, NULLIF, VALUE, and VARCHAR scalar functions.

You can create an index on a ROWID column. DECLARE CURSOR, FETCH, DECLARE TABLE, DESCRIBE, PREPARE INTO and EXECUTE also support ROWID columns.

The output that DCLGEN generates for a ROWID is shown in Example 9-9.

Example 9-9 DCLGEN output for a ROWID column

```

PL/I: DCL 1 LITERATURE
      ..
      01 IDCOL SQL TYPE IS ROWID,
      ..

COBOL: 01 LITERATURE.
      ..
      10 IDCOL USAGE SQL TYPE IS ROWID.
      ..

C/C++: struct
      {
      ..
      SQL TYPE IS ROWID IDCOL;
      ..
      } LITERATURE;

```

The precompiler will turn **'SQLTYPE IS ROWID'** into normal host language variable definitions as shown in Example 9-10 in case of a Cobol program. The variable that will contain the ROWID column is defined as a 40 byte character field and a 2 byte length field. The length field will contain the actual length of the ROWID value (normally 22 bytes when the ROWID column has not been added to the table after the table was created).

Example 9-10 Coding a ROWID host variable in Cobol

```

*01 IDCOL    USAGE SQL TYPE IS ROWID. <-- What you code in Cobol
                                     ('*' is the comment placed by precompiler)
01 IDCOL.    <-- Precompiler replacement
  49 IDCOL-LEN PIC S9(4) USAGE COMP.
  49 IDCOL-TEXT PIC X(40).

```

Note: External ROWID format is variable length (up to 40 bytes).

9.2.5 ROWIDs and partitioning keys

Since a ROWID column contains a semi-random and unique value, they may appear to be a good candidate for a partitioning key.

Although the random nature of the ROWID column eliminates insert hot-spots from the table space, you usually have to define additional non-partitioning indexes to retrieve the data afterwards, since the partitioning key, the ROWID column, is not a meaningful 'natural' key. Especially on very large table spaces, having several non-partitioning indexes can introduce some challenges for 'sequential' batch runs and utilities.

There are better techniques to spread rows to be inserted randomly across the partitions of a partitioned table space when there is no column that can serve as a good partitioning key. You can, for example, create a hashing fieldproc.

Example 9-11 shows that you need a non-partitioning index if you choose a ROWID column as partitioning key.

Example 9-11 Why not to use a ROWID column as a partitioning key

```
CREATE TABLESPACE TS246399 IN DB246300 Numparts 4 #

CREATE TABLE SC246300.LITERATURE_PART
  (TITLE      CHAR(25)
  ,IDCOL      ROWID  NOT NULL GENERATED ALWAYS
  ,SEQNO      INTEGER
  ,BOOKTEXT   LONG VARCHAR )
IN DB246300.TS246399 #

CREATE INDEX SC246300.IDCOLIX_PART
ON SC246300.LITERATURE_PART (IDCOL)
CLUSTER (PART 1 VALUES(X'3F'),
        PART 2 VALUES(X'7F'),
        PART 3 VALUES(X'BF'),
        PART 4 VALUES(X'FF')) #

-- However you still need a non-partitioning index to find the rows (for the first
-- time) so you need another index.

CREATE UNIQUE INDEX SC246300.TITLEIX
ON SC246300.LITERATURE_PART (TITLE,SEQNO) #

INSERT INTO SC246300.LITERATURE_PART
  (TITLE,SEQNO,BOOKTEXT)
VALUES ( 'MY MASTERPIECE', 1 , 'IT WAS THE BEST OF TIMES,') #

SELECT * FROM SC246300.LITERATURE_PART
WHERE TITLE = 'MY MASTERPIECE' #
```

Important: LOAD PART x is not allowed for a table whose partitioning key contains a ROWID. This is because ROWIDs are pseudo-random generated and may point to any partition of the table, whereas LOAD PART x can only insert rows into partition x.

9.2.6 ROWID and direct row access

If an application selects a row from a table that contains a ROWID column, the ROWID value implicitly contains the location of the row. If you use that ROWID value in the search condition of subsequent SELECTs, UPDATEs or DELETEs, DB2 can choose to navigate directly to that row without using an index. This access method is called **direct row access**, and is a very efficient access path. However, if you expect direct row access to be used, but for some reason (like a REORG has happened in the meantime), DB2 decides not to use direct row access at execution time, it may lead to serious performance problems.

Therefore, use direct row access with great caution.

Application programs that want to implement using direct row access have to be carefully designed. It is very easy to code it wrong, especially when using pseudo conversational transaction managers. If not used correctly, it can lead to serious performance problems when the row accessed changed location in the meantime. There is no integrity exposure, like trying to update the wrong row. The system detects that the ROWID value is no longer 'valid' and uses an alternate access path to access the data.

When explaining SQL statements that might qualify for direct row access, a new PLAN_TABLE column, PRIMARY_ACCESTYPE indicates whether (value D) or not (blank) direct row access is attempted.

From a performance point of view, using direct row access may save some index accesses (getpages and possibly I/Os as well), but, there is always the 'risk' that DB2 has to revert to an alternate access path if direct row access cannot be used for some reason at execution time. That alternate access path can mean degrading to a table space scan (see "Falling back from direct row access to another access path" on page 120).

Note: ROWIDs were originally conceived to be used with LOBs. However, direct row access has nothing to do with LOBs whatsoever. Direct row access is used to locate rows in a table that have a ROWID column defined. Accessing LOBs in the LOB table space is done using the index on the auxiliary table.

Direct row access can be used on any table that has a ROWID column. Example 9-12 show an example of direct row access.

Example 9-12 ROWID direct row access

```
SELECT IDCOL INTO :bookid FROM SC246300.LITERATURE WHERE ...

SELECT TITLE, SUBSTR(LOBBOOK, 1, 1000) INTO ...
      FROM SC246300.LITERATURE
      WHERE IDCOL = :bookid
```

Direct row access can be very fast, but should only be used if extremely high performance is needed. In order for an application to be able to benefit from direct row access the application logic must first retrieve the row with data and at a later stage update or delete it.

If the application can use a cursor that is defined as 'for update of', then it should use this type of cursor instead of direct row access. The 'update where current of cursor' also does a 'direct' update to the row as it does not have to search the row before updating it. The cursor is already positioned on the row that you are about to update. Using a cursor that is defined with 'for update of' does not have to consider a fallback access path as is the case with direct row access.

A lot of applications cannot use a 'for update of' cursor. For example, an on-line IMS transaction cannot use this type of cursor since selecting the data and displaying it on the screen, and updating the data are separate executions of a transaction. The cursor is no longer there when the transaction returns to do the update. Another reason for not using a 'for update of' cursor by many applications is the type of locking that is involved with this type of cursor. The UPDATE or DELETE operation takes place immediately after the row has been fetched because of the usage of the WHERE CURRENT OF clause (that is what makes it a positioned UPDATE or DELETE). Therefore, the X-lock on the page/row is usually held longer than is the case when using a non-cursor update, that you usually perform as close to the COMMIT as possible (to reduce the amount of time the X-lock is held). Therefore, using a positioned UPDATE or DELETE may have an impact on concurrency.

Therefore, in order to be considered as a candidate for direct row access, your application:

- ▶ Needs to read the data first and update/delete it later
- ▶ Is unable to use a 'for update of' cursor

If the application does not have to read the row, you cannot use direct row access. You must retrieve the ROWID column first and use it later on to read/update/delete the same data again.

A good use may be if for some reason you have to update the same row several times.

Even though direct row access can be a great access path (avoiding index access and scanning needs), storing ROWID values in a table is usually a bad idea, especially when storing a large number of rows for a longer period of time as is explained in the next section.

Falling back from direct row access to another access path

Although DB2 might plan to use direct row access, circumstances can cause DB2 not to use direct row access at run time. DB2 remembers the location of the row as of the time it is accessed. However, that row can change location (such as after a REORG) before it is accessed again, which means that DB2 can no longer use direct row access to find the row. Instead of using direct row access, DB2 uses the 'alternate' access path. The 'alternate' access path is shown in the ACCESSTYPE column of the PLAN_TABLE when explaining the SQL statement or binding with the EXPLAIN(YES) option.

This can have a profound impact on the performance of applications that count on direct row access. So it is important that applications handle this situation. Some options are:

- ▶ Ensure that your application does not try to remember ROWID values across reorganizations of the table space or other processes that could move the row. (A not-in-place update, either within the page or even to a different page does not invalidate the ROWID, as such.)
 - If an application variable stores a ROWID column value longer than it holds a claim on the table space, a REORG can occur and move the row, disabling direct row access. Remember that claims are released at commit if the cursor is not defined WITH HOLD. Therefore, a recommended practice is to select the ROWID just prior to using it in another SQL statement, without intermediate COMMITS, that could allow a REORG to occur. So plan your commit processing accordingly or you could have unexpected results, as it could happen if you code Example 9-13.

- If you are storing ROWID columns from another table (as VARCHARs, not as ROWIDs), you should somehow update those values after the table with the ROWID columns is reorganized. Therefore, storing ROWIDs of a table in another table is not recommended.
- ▶ Supplement the ROWID column predicate with another boolean term predicate that enables DB2 to use an existing index on the table or a good alternate access path.
- ▶ Create an index on the ROWID column, so that DB2 can use the index if direct row access is disabled.

Example 9-13 Inappropriate coding for direct row access.

```
--Do NOT attempt the following:

SELECT IDCOL INTO :bookid FROM SC246300.LITERATURE WHERE ... ;

COMMIT ;

DELETE FROM SC246300.LITERATURE           -- This results in a table space scan
WHERE IDCOL = :bookid                     -- if the ROWID is no longer valid and
                                           -- no index exists on IDCOL
```

One of the checks that is performed to see if a ROWID value is valid is verifying the EPOCH number. The EPOCH number can be found in SYSIBM.SYSTABLEPART. When a table space is created, the initial value of EPOCH is zero, and it is incremented whenever an operation resets the page set or partition (such as REORG or LOAD REPLACE).

Important: If you use non-IBM utilities that reset the page set or partition (like REORG or LOAD REPLACE) and you use direct row access, make sure that these utilities update the EPOCH column in SYSTABLEPART when they reset the page set or partition.

9.2.7 ROWID and direct row access restrictions

Some ROWID restrictions are as follows:

- ▶ A table can have only one ROWID column.
- ▶ A ROWID column cannot be updated.
- ▶ A ROWID column cannot be defined as nullable.
- ▶ LOAD PART is not allowed if the partitioning key contains a ROWID column.
- ▶ No editproc is allowed on tables containing a ROWID column.
- ▶ No fieldproc is allowed on a ROWID column.
- ▶ No check constraints can be defined on a ROWID column.
- ▶ A ROWID column cannot be used in a temporary table.
- ▶ A ROWID column cannot be part of a primary or foreign key.

If the unique index defined on a ROWID column with the GENERATE BY DEFAULT attribute is dropped, the table is marked incomplete, and access to the table is not allowed until the index is created.

Direct row access can be used to navigate directly to a row if:

- ▶ The ROWID is used in an equal (=) or IN predicate
- ▶ The predicate is a Boolean Term predicate (can only be ANDed with other predicates)
- ▶ If it is used in an IN predicate, an index on ROWID must exist.
- ▶ Direct row access and parallelism are mutually exclusive (its use disables parallelism).
- ▶ Direct row access and RID list processing are mutually exclusive.
- ▶ Direct row access is not used for join processing (this may be changed in the future).

- ▶ The DB2 predictive governor estimates the cost of the query with the assumption that the direct access is successful. It does not report the estimated cost of the alternative access path.

9.3 Identity column and ROWID usage and comparison

At first glance they both seem good alternatives to having the application code to generate unique values for certain columns.

Although both identity columns and ROWIDs guarantee a unique number the nature of the number is quite different. A ROWID value is a random hex string whereas an identity column is an ever increasing or decreasing (when cycling is not allowed) numeric number.

Both have some similar restrictions concerning the usage of GENERATED ALWAYS and GENERATED BY DEFAULT columns. GENERATED ALWAYS seems the more attractive as it does not require you to have a unique index to guarantee uniqueness. However, in the case of a GENERATED ALWAYS ROWID column it might be wise to create a unique index on the ROWID column in case DB2 cannot use direct row access and DB2 has to revert to the 'alternate' access path.

A lot of installations copy (LOAD) their production data into a development and/or quality assurance (QA) environment. This can cause problems when using GENERATED ALWAYS columns (for both identity and ROWID columns). The LOAD will replace the values in those columns by new values. This can pose a problem if tables are related to each other based on such a column because the value in the 'GENERATED ALWAYS' column will have changed but the value in the other table has not. As an alternative you might choose to define those columns as GENERATED BY DEFAULT in your development and/or quality assurance environment. This will allow the LOAD utility to preserve the values from the production environment (where the column is defined as GENERATED ALWAYS). However, you should be aware that this can cause other problems. Programs will be allowed to specify a 'GENERATED BY DEFAULT' column in an INSERT statement for example. This will work fine in development and QA, but will fail once the program is moved into production. Programmers should be made aware of this potential problem to avoid production outages.

Identity and ROWID columns pose some complications when having to add data to a partitioned table space using the LOAD utility (for example, when GENERATED ALWAYS is used for an identity column or to load a partition when a ROWID column is part of the partitioning key).

The ROWID data type has no built-in function that is comparable to IDENTITY_VAL_LOCAL() that can be used to retrieve the most recently used value for an identity column.

The use of identity columns can be a replacement for application code that generates ever increasing or decreasing values. Identity columns do not have the locking considerations that you need to deal with when implementing this with application logic. However, applications that use identity columns have to tolerate gaps in the numbers that are assigned. Another consideration is that loading data becomes more complex when using the GENERATED ALWAYS attribute. Also all considerations associated with inserting rows at the end of a table still apply when using identity columns.

So the truth of the matter is that although very promising at first glance, both identity columns and ROWID columns (outside of LOBs) need to be examined very closely before you decide to use them. Although they can serve specific needs they probably will not be that widely used.



Part 3

More powerful SQL

This part discusses the following topics:

- ▶ SQL CASE expressions
- ▶ Union everywhere
- ▶ Scrollable cursors
- ▶ Extensions to the ON clause
- ▶ Row expressions
- ▶ ORDER BY improvements
- ▶ INSERT, UPDATE and SELECT INTO enhancements



SQL CASE expressions

In this chapter, we discuss CASE expressions. They can increase programmer productivity, portability and query performance by replacing multiple SQL statements with a single CASE expression. The CASE expression allows you to write conditional logic based on the evaluation of one or more conditions.

You can write CASE expressions in:

- ▶ SELECT lists
- ▶ Predicates (WHERE clauses)
- ▶ User-defined functions
- ▶ Triggers
- ▶ SET clauses in UPDATE statements

10.1 What is an SQL CASE expression?

A case expression allows an expression to be selected based on the evaluation of one or more conditions. The value of the case expression is the value of the first result expression that evaluates to true. If no expression in the case evaluates to true, then the result is the result expression of the ELSE, or Null when the ELSE keyword is not present.

CASE expression characteristics

The keyword CASE begins a case expression

The data type of the *expression* prior to the first WHEN keyword must be comparable to the data types of each *expression* that follows the WHEN keywords.

The WHEN clause comes in two flavors:

- ▶ Simple-when-clause
- ▶ Searched-when-clause

Simple-when-clause specifies that the value of the expression prior to the first WHEN keyword is tested for equality with the value of each *expression* that follows the WHEN keywords. Specifies the result for each WHEN keyword when the expressions are equal.

```
expression WHEN expression THEN result-expression / NULL
DEPTNO WHEN 'C' THEN 'SYSTEMS'
```

In the Example 10-1 we select the employee number, last name and division from the TBEMPLOYEE table. The first character of the work department number represents the division within the organization. By using a CASE expression with a *simple-when-clause*, it is possible to translate the codes and list the full name of the division to which each employee belongs.

Example 10-1 SELECT with CASE expression and simple WHEN clause

```
SELECT
    EMPNO
    ,LASTNAME
    ,CASE SUBSTR(WORKDEPT,1,1)
        WHEN 'A' THEN 'ADMINISTRATION'
        WHEN 'B' THEN 'HUMAN RESOURCES'
        WHEN 'C' THEN 'DESIGN'
        WHEN 'D' THEN 'OPERATIONS'
        ELSE 'UNKNOWN DEPARTMENT'
    END AS DIVISION
FROM SC246300.TBEMPLOYEE ;
```

Searched-when-clause specifies a search condition to be applied to each row or group of table data presented for evaluation, and the result when that condition is true.

```
WHEN search-condition THEN result-expression / NULL
WHEN ORDERDATE > CURRENT DATE + 60 DAYS THEN '2'
```

Example 10-2 gives an example of using a CASE expression with a *searched-when-clause* to update the salary of our employees depending on their job. As shown in the example, CASE expressions are not limited to SELECT statements.

Example 10-2 Update with CASE expression and searched WHEN clause

```
UPDATE SC246300.TBEMPLOYEE
```

```

SET SALARY = CASE
    WHEN JOB IN ('MANAGER' , 'SUPRVSR') THEN SALARY * 1.10
    WHEN JOB IN ('DBA' , 'SYS PROG') THEN SALARY * 1.08
    WHEN JOB = 'PRGRMR' THEN SALARY * 1.05
    ELSE SALARY * 1.035
END ;

```

Result-expression specifies an expression that follows the THEN and ELSE keywords. It specifies the result of a *searched-when-clause* or a *simple-when-clause* that is true, or a result if no case is true.

```

    WHEN 'D' THEN 'OPERATIONS'
    ELSE 'UNKNOWN DEPARTMENT'

```

All result-expressions must be compatible. There must be at least one result-expression in the CASE expression with a defined data type. NULL cannot be specified for every data type. RAISE_ERROR cannot be specified for every CASE result-expression. For an example of using CASE expressions in a trigger see Example 3-10 on page 24.

Search-condition specifies a condition that is true, false or unknown about a row, or group of table data.

The ELSE clause is either followed by a *result-expression* or *NULL*.

The keyword END ends a case-expression

10.2 Why use an SQL CASE expression

There are many areas where CASE expressions can be used successfully. The best way to get some feeling for their capabilities is by looking at some examples.

CASE expressions can be used to replace decision logic implemented with UNION, UNION ALL or complicated OR-clauses. These methods temporarily divide result sets based on multiple returned values which may cause DB2 to repeatedly scan the same data pages. In contrast, with the CASE expression it is possible to accomplish the same with one pass of the data. For this reason, CASE expressions may significantly reduce the elapsed time of queries.

Looking at Example 10-3 and Example 10-4, you can see how to avoid coding several update statements by coding a single update with a CASE expressions. The examples show two different ways to process a salary increase. The salary raise is dependent on the job class. Certain classes get 10%, others 8%, others 5%, and yet others a 3.5% raise. You can code several update statements, one for each job class, and scan the TBEMPLOYEE table once for each update (assuming no index is present) as shown in Example 10-3, or you can use a CASE expression, shown in Example 10-4, and write one SQL statement to do all the updates with one pass through the data.

Example 10-3 Three updates vs. one update with a CASE expression

```

UPDATE SC246300.TBEMPLOYEE
  SET SALARY = SALARY * 1.10
  WHERE JOB IN ('MANAGER' , 'SUPRVSR') ;

UPDATE SC246300.TBEMPLOYEE
  SET SALARY = SALARY * 1.08
  WHERE JOB IN ('DBA' , 'SYS PROG') ;

```

```
UPDATE SC246300.TBEMPLOYEE
SET SALARY = SALARY * 1.05
WHERE JOB = 'PRGRMR' ;
```

```
UPDATE SC246300.TBEMPLOYEE
SET SALARY = SALARY * 1.035
WHERE JOB NOT IN ('MANAGER' , 'SUPRVSR', 'DBA' , 'SYS PROG', 'PRGRMR') ;
```

Example 10-4 One update with the CASE expression and only one pass of the data

```
UPDATE SC246300.TBEMPLOYEE
SET SALARY = CASE
    WHEN JOB IN ('MANAGER' , 'SUPRVSR') THEN SALARY * 1.10
    WHEN JOB IN ('DBA' , 'SYS PROG') THEN SALARY * 1.08
    WHEN JOB = 'PRGRMR' THEN SALARY * 1.05
    ELSE SALARY * 1.035
END ;
```

In the following examples we update ORDERSTATUS on table TBORDER depending on the value of ORDERDATE. We can accomplish this with three SQL statements and three passes of the data (see Example 10-5) or with one SQL statement using a CASE expression and one pass of the data (see Example 10-6). Additionally, since the CASE expression stops evaluating once the first WHEN clause evaluates true, we can also simplify the logic (see Example 10-7).

Example 10-5 Three updates vs. one update with a CASE expression

```
UPDATE SC246300.TBORDER
SET ORDERSTATUS = '0'
WHERE ORDERDATE <= CURRENT DATE + 30 DAYS ;

UPDATE SC246300.TBORDER
SET ORDERSTATUS = '1'
WHERE ORDERDATE > CURRENT DATE + 30 DAYS
AND ORDERDATE <= CURRENT DATE + 60 DAYS ;

UPDATE SC246300.TBORDER
SET ORDERSTATUS = '2'
WHERE ORDERDATE > CURRENT DATE + 60 DAYS ;
```

Example 10-6 Same update implemented with CASE expression and only one pass of the data

```
UPDATE SC246300.TBORDER
SET ORDERSTATUS = CASE
    WHEN ORDERDATE <= CURRENT DATE + 30 DAYS THEN '0'
    WHEN (ORDERDATE > CURRENT DATE + 30 DAYS AND
    ORDERDATE <= CURRENT DATE + 60 DAYS) THEN '1'
    WHEN ORDERDATE > CURRENT DATE + 60 DAYS THEN '2'
END ;
```

Example 10-7 Same update with simplified logic

```
UPDATE SC246300.TBORDER
SET ORDERSTATUS = CASE
    WHEN ORDERDATE <= CURRENT DATE + 30 DAYS THEN '0'
    WHEN ORDERDATE <= CURRENT DATE + 60 DAYS THEN '1'
    ELSE '2'
END ;
```

Example 10-8 uses a CASE expression to avoid 'division by zero' errors. The following queries show an accumulation or summing operation. In the first query, it is possible to get an error because PAYMT_PAST_DUE_CT can be zero and division by zero is not allowed. Notice that in the second part of the example, the CASE statement first checks to see if the value of PAYMT_PAST_DUE_CT is zero, if it is we return a zero, otherwise we perform the division and return the result to the SUM operation.

Example 10-8 Avoiding division by zero

```
SELECT REF_ID
      ,PAYMT_PAST_DUE_CT
      ,SUM ( BAL_AMT / PAYMT_PAST_DUE_CT )
FROM PAY_TABLE
GROUP BY REF_ID
      ,PAYMT_PAST_DUE_CT; --This statement can get a division by zero error
```

versus

```
SELECT REF_ID
      ,PAYMT_PAST_DUE_CT
      ,SUM (CASE
            WHEN PAYMT_PAST_DUE_CT = 0 THEN 0
            WHEN PAYMT_PAST_DUE_CT > 0 THEN BAL_AMT / PAYMT_PAST_DUE_CT
            END)
FROM PAY_TABLE
GROUP BY REF_ID
      ,PAYMT_PAST_DUE_CT; --This statement avoids division by zero errors
```

Following is another example (Example 10-9) that also shows how to use a CASE expression to avoid division by zero errors. From the TBEMPLOYEE table, find all employees who earn more than 25 percent of their income from commission, but who are not fully paid on commission.

Example 10-9 Avoid division by zero, second example

```
SELECT
    EMPNO
    ,WORKDEPT
    ,SALARY + COMM
FROM SC246300.TBEMPLOYEE
WHERE
    (CASE WHEN SALARY = 0 THEN 0
         ELSE COMM/(SALARY + COMM)
        END) > 0.25 ;
```

SALARY	COMM	SALARY+COMM	COMM/(SALARY+COMM)
90000	10000	100000	0.10
100000	0	100000	0.00

0	100000	100000	1.00
0	0	0	error, division by zero

Example 10-10 shows how to replace many UNION ALL clauses with one CASE expression. In this example, if the table is not clustered by the column STATUS, DB2 probably does not use an index to access the data and scan the entire table once for each SELECT. By using the CASE expression we scan the data only once. This could be a significant performance improvement depending on the size of the table and the number of rows that are accessed by the query.

Example 10-10 Replacing several UNION ALL clauses with one CASE expression

```

SELECT
    CLERK
    ,CUSTKEY
    ,'CURRENT' AS STATUS
    ,ORDERPRIORITY
FROM SC246300.TBORDER
WHERE ORDERSTATUS = '0'
UNION ALL
SELECT
    CLERK
    ,CUSTKEY
    ,'OVER 30' AS STATUS
    ,ORDERPRIORITY
FROM SC246300.TBORDER
WHERE ORDERSTATUS = '1'
UNION ALL
SELECT
    CLERK
    ,CUSTKEY
    ,'OVER 60' AS STATUS
    ,ORDERPRIORITY
FROM SC246300.TBORDER
WHERE ORDERSTATUS = '2' ;

```

versus

```

SELECT CLERK
    ,CUSTKEY
    ,CASE
        WHEN ORDERSTATUS = '0' THEN 'CURRENT'
        WHEN ORDERSTATUS = '1' THEN 'OVER 30'
        WHEN ORDERSTATUS = '2' THEN 'OVER 60'
    END AS STATUS
    ,ORDERPRIORITY
FROM SC246300.TBORDER ;
-- this statement is equivalent to the above
-- UNION ALL but requires only one pass through
-- the data.

```

CASE expressions can also be used in before triggers to edit and validate input data and raise an error if the input is not correct. This can simplify programming by taking application logic out of programs and placing it into the data base management system, reducing the number of lines of code needed since the code is only in one place. See the trigger example in Example 3-10 on page 24.

CASE expressions also provide additional DB2 Family consistency and thereby enhance application portability in a client/server environment.

10.3 Alternative solutions

There are two scalar functions, NULLIF and COALESCE, that can be used to perform a subset of the functionality provided by the CASE statement. Table 10-1 shows the equivalent expressions using CASE expressions or these functions. As you can see, the CASE expression is more self documenting and it is easier to see what operation it is performing.

Note: These CASE expressions are stage 2 if coded in a WHERE clause while the NULLIF and COALESCE are stage 1 predicates.

Table 10-1 Functions equivalent to CASE expressions

CASE expression	Equivalent expression
CASE WHEN <i>e1=e2</i> THEN NULL ELSE <i>e1</i> END	NULLIF(<i>e1,e2</i>)
CASE WHEN <i>e1</i> IS NOT NULL THEN <i>e1</i> ELSE <i>e2</i> END	COALESCE(<i>e1,e2</i>) or VALUE(<i>e1,e2</i>)
CASE WHEN <i>e1</i> IS NOT NULL THEN <i>e1</i> WHEN <i>e2</i> IS NOT NULL THEN <i>e2</i> WHEN ... ELSE <i>eN</i> END	COALESCE(<i>e1,e2,...,eN</i>) or VALUE(<i>e1,e2,...,eN</i>)

10.4 Other uses of CASE expressions

Example 10-11 lists the employee number and education level from the employee table. List the education level as 'post graduate', 'graduate' and 'diploma' instead of the integer code that is stored in the table. If an education level is greater than 20, raise an error ('70001') with a detailed error message.

Example 10-11 Raise an error in CASE statement

```

SELECT
    EMPNO
    , CASE WHEN EDLEVEL < 16 THEN 'DIPLOMA'
          WHEN EDLEVEL < 18 THEN 'GRADUATE'
          WHEN EDLEVEL < 20 THEN 'POST GRADUATE'
          ELSE
            RAISE_ERROR ('70001', 'EDUCLVEL HAS VALUE GREATER THAN 20')
          END AS EDUCLEVEL
FROM SC246300.TBEMPLOYEE ;

```

Example 10-12 shows how a CASE statement can be used to 'pivot' a table. Sometimes it is convenient to pivot the results of a query in order to simplify program logic to display the data on a screen or simplify the generation of a report.

Example 10-12 Pivoting tables

```

SELECT CUSTKEY
  , SUM(CASE
        WHEN TOTALPRICE BETWEEN 0 AND 99 THEN 1
        ELSE 0
        END) AS SMALL
  , SUM(CASE
        WHEN TOTALPRICE BETWEEN 100 AND 250 THEN 1
        ELSE 0
        END) AS MEDIUM
  , SUM(CASE
        WHEN TOTALPRICE > 250 THEN 1
        ELSE 0
        END) AS LARGE
FROM SC246300.TBORDER
GROUP BY CUSTKEY #

```

Assume that table TBORDER contains the following rows:

CUSTKEY	TOTPRICE
03	224.
05	30.
03	560.
05	150.
01	50.
03	550.
01	40.
04	40.
02	438.
03	75.

The result of the above query is:

CUSTKEY	SMALL	MEDIUM	LARGE
01	2	0	0
02	0	0	1
03	1	1	2
04	1	0	0
05	1	1	0

Example 10-13 shows how a CASE statement can be used to group the results of a query without having to re-type the expression. Using the employee table, find the maximum, minimum, and average salary. Instead of finding these values for each department, assume that we want to combine some departments into the same group. Combine departments A00 and E21, and combine departments D11 and E11.

Example 10-13 Use CASE expression for grouping

```

SELECT CASE_DEPT

```

```

,MAX(SALARY) AS MAX_SALARY
,MIN(SALARY) AS MIN_SALARY
,AVG(SALARY) AS AVG_SALARY
FROM (SELECT
      SALARY
      ,CASE WORKDEPT WHEN 'A00' THEN 'A00_E21'
                    WHEN 'E21' THEN 'A00_E21'
                    WHEN 'D11' THEN 'D11_E11'
                    WHEN 'E11' THEN 'D11_E11'
                    ELSE WORKDEPT
      END AS CASE_DEPT
      FROM SC246300.TBEMPLOYEE) AS X
GROUP BY CASE_DEPT ;

```

Assume that table TBEMPLOYEE contains the following rows:

SALARY	WORKDEPT
70000	A00
60000	A00
50000	A00
36000	B20
40000	B20
44000	B20
42000	D11
54000	D11
45000	E11
30000	E21
40000	F20
51200	F20
80100	F20
40250	J11
50800	J11

The result of the above query is:

CASE_DEPT	MAX_SALARY	MIN_SALARY	AVG_SALARY
A00_E21	70000	30000	52500
B20	44000	36000	40000
D11_E11	54000	42000	47000
F20	80100	40000	57100
J11	50800	40250	30350

10.5 SQL CASE expression restrictions

The *search-condition* of a CASE expression cannot contain a subselect. If the CASE expression is in a select list, an IN predicate, or a SET clause of an UPDATE statements, the *search-condition* cannot be a quantified (ANY, SUM, ALL) predicate, an IN predicate, or an EXISTS predicate. Otherwise you receive an SQLCODE -582.

In a CASE expression, if a column with a field procedure is used as the *result-expression* in a THEN or ELSE clause, all other columns that are used as *result-expressions* must have the same field procedure. Otherwise, no column used in a *result-expression* may name a field procedure.

CASE expressions cannot be used in CHECK CONSTRAINTS.

CASE expressions in the WHERE clause are stage 2 predicates.

Tip: If the best possible access path is not a table space scan and you have CASE expressions in the WHERE clause, make sure that you also code other indexable and filtering predicates in the WHERE clause whenever possible.

The *result-expressions* of a CASE statement (expressions following the THEN and ELSE keywords) cannot be coded so that all of them are NULL. If you attempt to code a CASE expression that always returns a NULL result, you receive an SQLCODE -580.

The data type of every result-expressions must be compatible. If the CASE condition result data types are not compatible (either all character, graphic, numeric, date, time or timestamp) you receive an SQLCODE -581.

Columns of data types VARCHAR (greater than 255 bytes), VARGRAPHIC (greater than 127 bytes), and LOBs (CLOB, DBCLOB or BLOB) cannot be used anywhere within a CASE expression. In addition, the only type of user defined function that is allowed in the expression prior to the first WHEN keyword must be a deterministic UDF and it cannot contain external actions.



Union everywhere

In this chapter we discuss UNIONS and the many new places where they can now be used.

Starting with DB2 Version 7, UNIONS may now be used in:

- ▶ Views
- ▶ Table expressions
- ▶ Predicates (subqueries)
- ▶ Inserts
- ▶ Updates

Because a UNION can now be coded everywhere that you could previously code a subselect, this feature is also called 'union everywhere'.

We also discuss some ways 'union everywhere' can be applied in the physical design of extremely large tables.

11.1 What is a union everywhere?

A *subselect* is that portion of a query containing no more than SELECT, FROM, WHERE, GROUP BY, and HAVING clauses, but not including ORDER BY clause, UPDATE clause, or UNION operators. By definition, a *fullselect* is a SELECT statement that contains a UNION or UNION ALL, but not ORDER BY.

In DB2 V7, the SQL syntax has been enhanced so that you may use a *fullselect* (a portion of a select statement that contains a UNION or UNION ALL between subselects) anywhere a *subselect* was previously allowed.

Prior to DB2 V7, UNIONS could not be used in a CREATE VIEW, in nested table expressions, in subqueries, or in INSERT and UPDATE statements. Beginning with DB2 V7, the syntax of the view definition, nested table expressions, subqueries, inserts and updates has been enhanced to allow a *fullselect* clause instead of only a *subselect*.

Note: *Subselect* is no longer specified as a component of any statement syntax since the only difference between a *subselect* and a *fullselect* was the ability to use UNION and UNION ALL, and now you can use a *fullselect* everywhere only a *subselect* was previously allowed.

11.2 Why union everywhere

Union everywhere increases SQL usability by allowing logical tables that are split into multiple physical tables to be viewed by the end users as a logical table without the user having to understand the nuances of coding a UNION.

Performance has also been recognized as being an integral part of the making of this change. DB2 avoids materializing a view with unions as much as possible by using query rewrite. Other performance benefits have been included to exploit this enhancement.

The union everywhere implementations enhances the compatibility with other members within the DB2 UDB family and complies with the SQL99 standard.

11.3 Unions in nested table expressions

This enhancement is of benefit when there is a need to use functions across similar data which is stored in multiple tables. In this example, the data that has been merged, is grouped. Prior versions of DB2 would have required a temporary table to be created with a separate SQL statement to implement this statement. Now this can be achieved with a single SQL statement.

The query in Example 11-1 finds the number of male and female customers and employees in city 1010 and their average age.

Example 11-1 Unions in nested table expressions

```
SELECT SEX
       ,AVG(AGE)
       ,COUNT(*)
       ,CITYKEY
FROM TABLE (SELECT
              SEX
              ,YEAR(DATE(DAYS(CURRENT DATE)-DAYS(BIRTHDATE))) AS AGE
```

```

        ,CITYKEY
    FROM SC246300.TBEMPLOYEE
UNION ALL
SELECT
    SEX
    ,YEAR(DATE(DAYS(CURRENT DATE)-DAYS(BIRTHDATE))) AS AGE
    ,CITYKEY
    FROM SC246300.TBCUSTOMER
) AS EMPCUST
WHERE AGE > 21
GROUP BY SEX, CITYKEY

```

11.4 Unions in subqueries

A *subquery* is a subselect or a fullselect used within a search condition of any statement.

Unions can now be coded within basic, quantified (ANY, SOME, ALL), EXISTS, and IN (subquery) predicates. This gives us the ability to merge data from multiple tables and compare to single column values within a single SQL statement.

11.4.1 Unions in basic predicates

In Example 11-2, we find the customer city for a given phone number. The customers are spread across 2 tables. The ones that have not placed an order are moved to an archive table. However, for this query, we want to look through the entire set of customers including the archive. Be careful if more than one customer exists with the same phone number. If this is the case, you would receive an SQLCODE -811, since a basic operation (=, <>, <, >, >=, <=, ≠, ≠, ≠) against a *fullselect* is only allowed if the *fullselect* returns a single row. Having a customer with the same phone number in both the active table and the archive is not a problem since the UNION eliminates those duplicates.

Example 11-2 Using UNION in basic predicates

```

SELECT CITYKEY
    ,CITYNAME
    FROM SC246300.TBCITIES
    WHERE CITYKEY =
        (SELECT CITYKEY
            FROM SC246300.TBCUSTOMER
            WHERE PHONENO = :PHONENUM
        UNION
        SELECT CITYKEY
            FROM SC246300.TBCUSTOMER_ARCH
            WHERE PHONENO = :PHONENUM
        ) ;

```

11.4.2 Unions in quantified predicates

Example 11-3 produces a list of all the cities where we have customers or employees.

Example 11-3 Using UNION with quantified predicates

```
--Prior to DB2 V7
```

```

SELECT CITYKEY
      ,CITYNAME
FROM SC246300.TBCITIES A
WHERE CITYKEY = SOME
      (SELECT CITYKEY
        FROM SC246300.TBCUSTOMER)
OR CITYKEY = SOME
      (SELECT CITYKEY
        FROM SC246300.TBEMPLOYEE) ;

--In DB2 V7
SELECT CITYKEY
      ,CITYNAME
FROM SC246300.TBCITIES A
WHERE CITYKEY = SOME
      (SELECT CITYKEY
        FROM SC246300.TBCUSTOMER
UNION
SELECT CITYKEY
        FROM SC246300.TBEMPLOYEE ) ;

```

Note: The V7 query is coded with a UNION and not a UNION ALL. In this case, because of the =SOME predicate, DB2 converts the UNION into a UNION ALL. This shows up in the explain output. There is no sorting of the result of the UNION operation.

11.4.3 Unions in EXISTS predicates

In Example 11-4 we show the use of the UNION with the EXISTS predicate. In many, but not all cases, you may be able to write these queries in a different way and achieve a better access path. In this example we are merely showing you the new syntax. This example returns the same results as Example 11-3 (select all the cities where we have customers or employees).

Example 11-4 Using UNION in the EXISTS predicate

```

--Prior to DB2 V7
SELECT CITYKEY
      ,CITYNAME
FROM SC246300.TBCITIES A
WHERE EXISTS
      (SELECT 'DUMMY'
        FROM SC246300.TBCUSTOMER
        WHERE CITYKEY=A.CITYKEY)
OR EXISTS
      (SELECT 'DUMMY'
        FROM SC246300.TBEMPLOYEE
        WHERE CITYKEY=A.CITYKEY) ;

--In DB2 V7
SELECT CITYKEY
      ,CITYNAME
FROM SC246300.TBCITIES A
WHERE EXISTS
      (SELECT 'DUMMY'
        FROM SC246300.TBCUSTOMER

```



```

WHERE CITYKEY=A.CITYKEY
UNION ALL
SELECT 'DUMMY'
FROM SC246300.TBEMPLOYEE
WHERE CITYKEY=A.CITYKEY ) ;

```

11.4.4 Unions in IN predicates

In Example 11-5 we show the usage of the UNION in an IN predicate. This query returns the same result as Example 11-3 and Example 11-4. The first part of this example shows how this query could have been written prior to DB2 V7; the second part show the query as it can be written in DB2 V7. Note that prior to DB2 V7, you were required to code two IN predicates (subqueries in this case) ORed together.

Example 11-5 Using UNION in an IN predicate

```

--Prior to DB2 V7
SELECT CITYKEY
      ,CITYNAME
FROM SC246300.TBCITIES A
WHERE CITYKEY IN (SELECT CITYKEY
                  FROM SC246300.TBCUSTOMER)
OR CITYKEY IN (SELECT CITYKEY
               FROM SC246300.TBEMPLOYEE) ;

--In DB2 V7
SELECT CITYKEY
      ,CITYNAME
FROM SC246300.TBCITIES A
WHERE CITYKEY IN
      (SELECT CITYKEY
       FROM SC246300.TBCUSTOMER
       UNION ALL
       SELECT CITYKEY
       FROM SC246300.TBEMPLOYEE ) ;

```

11.4.5 Unions in selects of INSERT statements

Example 11-6 shows how the UNION keyword can now be used in INSERT statements. In this example we build an invitation list to invite all our employees and customers who where born after 1968 to attend a customer and employee appreciation Rock-n-Roll concert.

Example 11-6 Using UNION in an INSERT statement

```

INSERT INTO SC246300.INVITATION_CARDS
(PHONENO,STATUS,SEX,BIRTHDATE,CITYKEY,FIRSTNAME,LASTNAME ,ADDRESS)
SELECT
      PHONENO
      ,'U'
      ,SEX
      ,BIRTHDATE
      ,CITYKEY
      ,FIRSTNME
      ,LASTNAME

```

```

        ,ADDRESS
FROM SC246300.TBEMPLOYEE
WHERE YEAR(BIRTHDATE) > 1968
UNION
SELECT
    PHONENO
    , 'U'
    ,SEX
    ,BIRTHDATE
    ,CITYKEY
    ,FIRSTNAME
    ,LASTNAME
    ,ADDRESS
FROM SC246300.TBCUSTOMER
WHERE YEAR(BIRTHDATE) > 1968 #

```

11.4.6 Unions in UPDATE

Example 11-7 shows how the UNION keyword can now also be used in the UPDATE statement. Here we set the STATUS column on the INVITATION table to 'L' when we find a match on employee or customer based on phone number and birth date for those customers and employees in city 22 (assuming that a match in phone number and birth date indicates it is the same person). (City 22 is where the concert will be and these people are considered local (STATUS='L') so we don't have to organize transportation for them). In case there is no row that qualifies in the subquery, STATUS is set to NULL.

Example 11-7 Using UNION in an UPDATE statement

```

UPDATE SC246300.INVITATION_CARDS X
SET STATUS = (
    SELECT 'L'
    FROM SC246300.TBEMPLOYEE Y
    WHERE
        CITYKEY = 22 AND
        Y.PHONENO = X.PHONENO AND
        Y.BIRTHDATE = X.BIRTHDATE
    UNION ALL
    SELECT 'L'
    FROM SC246300.TBCUSTOMER Z
    WHERE
        CITYKEY = 22 AND
        Z.PHONENO = X.PHONENO AND
        Z.BIRTHDATE = X.BIRTHDATE
) ;

```

11.5 Unions in views

Starting with DB2 V7, you can now use the UNION or UNION ALL keywords as part of the CREATE VIEW syntax. However, it is important to note that following the standard SQL rules for views, if the UNION or UNION ALL keywords are used in the creation of a view, the view is read-only.

Important: No updates are allowed on views containing the UNION ALL or UNION keyword.

Before UNIONS were allowed in views, there were only two options to perform the equivalent function:

- ▶ The first option was to create a physical table containing the merged data from the multiple tables. This required some work, you needed to unload the multiple tables and load into the single table periodically. This meant that there was the potential for the data to be inaccurate, since the actual tables were not directly used for the user's queries.
- ▶ The second option was for the user code UNIONS without the use of a view. However, coding the union is a bit more complex, and had to be coded again for every query wanting to work with the same data. Additionally, functions (such as AVERAGE, COUNT, and SUM) could not range across the union, further complicating the process. These functions could be accomplished through additional steps, for example, by creating a temporary table containing the output of the SELECT statement with the union clause, and then run another SELECT to perform the column function against the temporary table.

DB2 V7 provides a simple answer for this problem. Data from the base tables can be merged dynamically by creating a view using unions. Once coded, the user only has to refer to the view to have access to data across several tables and can now easily use the full suite of functions, such as COUNT, AVG, and SUM across all the data.

To get a combined list of employees and customers, we can create a view described in Example 11-8.

Example 11-8 Create view with UNION ALL

```
CREATE VIEW SC246300.CUSTOMRANDEMPLOYEE
AS
SELECT
    FIRSTNAME AS FIRSTNAME
    ,LASTNAME
    ,PHONENO
    ,BIRTHDATE
    ,SEX
    ,YEAR(DATE(DAYS(CURRENT DATE)-DAYS(BIRTHDATE))) AS AGE
    ,ADDRESS
    ,CITYKEY
FROM SC246300.TBEMPLOYEE
UNION ALL
SELECT
    FIRSTNAME
    ,LASTNAME
    ,PHONENO
    ,BIRTHDATE
    ,SEX
    ,YEAR(DATE(DAYS(CURRENT DATE)-DAYS(BIRTHDATE))) AS AGE
    ,ADDRESS
    ,CITYKEY
FROM SC246300.TBCUSTOMER
;
```

Example 11-9 shows a query to get the average age of employees and customers under 35 years old and how many of them there are.

Example 11-9 Use view containing UNION ALL

```
SELECT AVG(AGE),COUNT(*)
FROM SC246300.CUSTOMRANDEMPLOYEE
WHERE AGE < 35
```

As you can see, the addition of unions in the CREATE VIEW statement simplifies the merging of data from tables for end user queries and allows the use of the full suite of DB2 functions against the data without the need of temporary tables or complex SQL.

11.6 Explain and unions

As part of the enhancement support for UNION and UNION ALL operators, two new columns have been added to the PLAN_TABLE. The new columns are TABLE_TYPE and PARENT_QBLOCKNO. New values have been added to the existing QBLOCK_TYPE column, which shows the type of operation performed by the query block. Table 11-1 show these changes.

Table 11-1 PLAN_TABLE changes for UNION everywhere

Column	Value	Description
TABLE_TYPE	F	Table function
	Q	Table queue (not materialized). Temporary intermediate result table (used by STAR JOIN, not UNION)
	T	Table
	W	Workfile
QBLOCK_TYPE	TABLEX	Table expression
	UNION	UNION
	UNIONA	UNION ALL
PARENT_QBLOCKNO	Contains the query block numbers of the parent operation.	

In Example 11-10 we show the partial output from the PLAN_TABLE for the explain of the query in Example 11-6 on page 139. This EXPLAIN output shows that query block 3 and 4 (QBLOCKNO 3 and 4) have a parent query block 2 (PARENT_QBLOCKNO 2). If you see the the query processing as a tree, the outer select is the root (PARENT_QBLOCKNO 0). At the next level you find a union (PARENT_QBLOCKNO 1) which has 2 fullselects and each of them has PARENT_QBLOCKNO 2.

Note: Examining the QBLOCKNO and PARENT_QBLOCKNO sequence is the only means of figuring out if query rewrite has taken place. DB2 does not provide specific information on the outcome of a query rewrite.

Example 11-10 PLAN_TABLE output

```
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
QBLOCKNO  TNAME                ACESSTYPE  QBLOCK_TYPE  PARENT_QBLOCKNO  TABLE_TYPE
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
      1  INVITATION_CARDS                INSERT                0  T
```

2			UNION	1	-----
3	TBEMPLOYEE	R	NCOSUB	2	T
4	TBCUSTOMER	R	NCOSUB	2	T

11.7 Technical design and new frontiers

In some cases during the technical design it can be necessary to split up some tables. A reason for considering to split a table can be a requirement for high availability on a very large table. Even storing the table space in multiple physical partitions does not solve the problem of maintaining very large multi-data set non-partitioning indexes (NPIs). Recovering or reorganizing large NPIs may induce longer service downtime than the availability agreement allows for, or maintaining the NPIs may require too much resources and result in many sleepless nights.

Splitting a table horizontally into multiple tables is never an easy decision. Dividing is always more or less visible for applications. Using union in view, you can hide part of the change from the applications. This way the programs do not see any changes.

Impact on response times depends on the way the view is coded as well as the queries that are written against the view.

If the WHERE predicates reference the columns that were used for dividing up the tables, and these columns are compared with a constant(s) or host variables, only the table(s) having qualifying rows are accessed, and response time should be more or less the same as with a single large table. (This technique of limiting the number of tables that are accessed is called subquery pruning.)

When host variables are used for those columns, DB2, at bind time, cannot know which table(s) have to be accessed to fetch the rows from. Therefore, at bind time, the optimizer has to choose an access path that accesses every table. However, at execution time when the actual host variables are known, DB2 can do subquery pruning and accesses only the tables that can have qualifying rows. The subquery pruning for host variables always takes place and is not dependent on the use of the REOPT(VARS) bind option.

For more information about how the access path is determined and the different ways DB2 tries to optimize the execution of queries referencing views containing unions, like the distribution of predicates, joins, and aggregations, as well as the pruning of subqueries, table elimination and predicate distribution, see *DB2 for z/OS and OS/390 Performance Topics*, SG24-6129.

Note: Each individual query block that is part of the union everywhere query can run in parallel. However, multiple query blocks of the same query cannot run in parallel.

Sometimes a better access path can be obtained using CASE expressions. See Chapter 10, "SQL CASE expressions" on page 125 for more details. With CASE expressions you may save excessive access to tables with no qualifying rows.

Because views containing a UNION operator are read only, inserting, updating and deleting cannot be hidden from the programs by using a view.

However, the number of inserting, deleting and updating programs is usually limited. Writing an insert, a delete and an update (or few) modules to take care of the manipulation of data from these tables is usually sufficient. To prevent inserting rows into the wrong table, you should either define check constraint on the tables reflecting the same criteria that were used to divide the tables, or do these updates to views on each individual table using a WITH CHECK OPTION. (Views referencing a single table can be updatable.)

This way you can be sure that although DB2 cannot verify the data integrity by design, there is a way to guarantee that rows are only inserted or changed in the correct tables. Anyhow, programs still have to know in which table to insert, update or delete. Special processing is also required in case the value in the column that was used to assign rows to different tables is changed and the row now has to move to a different table. This type of manipulation requires a delete from the original table and an insert into the new table, operations similar to updating a column that is part of the partitioning key, before the introduction of the partitioning key update feature.

Because of the complexity of handling inserts, updates and deletes correctly, this type of design is probably most beneficial in a (mostly) read only environment, like data warehouses.

Loading the data is not a problem. DB2 can load many tables in one run. You only need to specify the exact rules how to select the correct table to which the data belongs. (That rule is our splitting criteria.)

Splitting tables is not relevant just because we can do it. Partitioning a table must always be the first choice for large tables. Usually partitioning the data is sufficient to solve most availability and manageability issues. DB2 offers more possibilities (more partitions, more parallelism, more flexibility) for partitioned tables in each new release. Dividing tables should be a last resort when nothing else is good enough. Note also that the 'divided tables' can themselves be partitioned as well.

Let's try to explain the concept using an example. Assume that our TBORDER table is getting to large and we decide to go for the UNION in view design. Suppose that the application logic uses some sort of randomizing formula to assign ORDERKEYs randomly taking into account that ORDERKEY is an integer column. We decide to split the TBORDER table into 3 separate tables, TBORDER_1, TBORDER_2 and TBORDER_3. Because of the randomizer, each table receive more or less the same number of rows. Sample DDL is shown in Example 11-11.

Example 11-11 DDL to create split tables

```
SET CURRENT PATH = 'SC246300' # -- to make sure we find the UDT

CREATE TABLESPACE TS246331 IN DB246300 #

CREATE TABLE SC246300.TBORDER_1
(
    ORDERKEY INTEGER NOT NULL ,
    CUSTKEY CUSTOMER NOT NULL ,
    ORDERSTATUS CHAR ( 1 ) NOT NULL WITH DEFAULT,
    TOTALPRICE FLOAT NOT NULL ,
    ORDERDATE DATE NOT NULL WITH DEFAULT,
    ORDERPRIORITY CHAR (15 ) NOT NULL WITH DEFAULT,
    CLERK CHAR ( 6 ) NOT NULL WITH DEFAULT,
    SHIPPRIORITY INTEGER NOT NULL WITH DEFAULT,
    STATE CHAR ( 2 ) NOT NULL WITH DEFAULT,
    REGION_CODE INTEGER,
    INVOICE_DATE DATE NOT NULL WITH DEFAULT,
    COMMENT VARCHAR ( 79 ),
```

```

        PRIMARY KEY (ORDERKEY),
        FOREIGN KEY (CUSTKEY) REFERENCES SC246300.TBCUSTOMER,
        FOREIGN KEY (REGION_CODE) REFERENCES SC246300.TBREGION
    )
    IN DB246300.TS246331
    WITH RESTRICT ON DROP #

-- Create unique index on primary key
CREATE UNIQUE INDEX SC246300.X1TBORDER_1
    ON SC246300.TBORDER_1(ORDERKEY ASC) #

-- Create indexes on foreign keys
CREATE INDEX SC246300.X2TBORDER_1
    ON SC246300.TBORDER_1(CUSTKEY ASC) #

CREATE INDEX SC246300.X3TBORDER_1
    ON SC246300.TBORDER_1(REGION_CODE ASC) #

CREATE TABLESPACE TS246332 IN DB246300 #

CREATE TABLE SC246300.TBORDER_2
    (
        ORDERKEY INTEGER NOT NULL ,
        CUSTKEY CUSTOMER NOT NULL ,
        ORDERSTATUS CHAR ( 1 ) NOT NULL WITH DEFAULT,
        TOTALPRICE FLOAT NOT NULL ,
        ORDERDATE DATE NOT NULL WITH DEFAULT,
        ORDERPRIORITY CHAR (15 ) NOT NULL WITH DEFAULT,
        CLERK CHAR ( 6 ) NOT NULL WITH DEFAULT,
        SHIPPRIORITY INTEGER NOT NULL WITH DEFAULT,
        STATE CHAR ( 2 ) NOT NULL WITH DEFAULT,
        REGION_CODE INTEGER,
        INVOICE_DATE DATE NOT NULL WITH DEFAULT,
        COMMENT VARCHAR ( 79 ),
        PRIMARY KEY (ORDERKEY),
        FOREIGN KEY (CUSTKEY) REFERENCES SC246300.TBCUSTOMER,
        FOREIGN KEY (REGION_CODE) REFERENCES SC246300.TBREGION
    )
    IN DB246300.TS246332
    WITH RESTRICT ON DROP#

CREATE UNIQUE INDEX SC246300.X1TBORDER_2
    ON SC246300.TBORDER_2(ORDERKEY ASC) #

CREATE INDEX SC246300.X2TBORDER_2
    ON SC246300.TBORDER_2(CUSTKEY ASC) #

CREATE INDEX SC246300.X3TBORDER_2
    ON SC246300.TBORDER_2(REGION_CODE ASC) #

CREATE TABLESPACE TS246333 IN DB246300 #

CREATE TABLE SC246300.TBORDER_3
    (
        ORDERKEY INTEGER NOT NULL ,
        CUSTKEY CUSTOMER NOT NULL ,
        ORDERSTATUS CHAR ( 1 ) NOT NULL WITH DEFAULT,
        TOTALPRICE FLOAT NOT NULL ,

```

```

ORDERDATE DATE NOT NULL WITH DEFAULT,
ORDERPRIORITY CHAR (15 ) NOT NULL WITH DEFAULT,
CLERK CHAR ( 6 ) NOT NULL WITH DEFAULT,
SHIPRIORITY INTEGER NOT NULL WITH DEFAULT,
STATE CHAR ( 2 ) NOT NULL WITH DEFAULT,
REGION_CODE INTEGER,
INVOICE_DATE DATE NOT NULL WITH DEFAULT,
COMMENT VARCHAR ( 79 ),
PRIMARY KEY (ORDERKEY),
FOREIGN KEY (CUSTKEY) REFERENCES SC246300.TBCUSTOMER,
FOREIGN KEY (REGION_CODE) REFERENCES SC246300.TBREGION
)
IN DB246300.TS246332
WITH RESTRICT ON DROP #

CREATE UNIQUE INDEX SC246300.X1TBORDER_3
ON SC246300.TBORDER_3(ORDERKEY ASC) #

CREATE INDEX SC246300.X2TBORDER_3
ON SC246300.TBORDER_3(CUSTKEY ASC) #

CREATE INDEX SC246300.X3TBORDER_3
ON SC246300.TBORDER_3(REGION_CODE ASC) #

```

Create a view, as shown in Example 11-12, that is used to retrieve rows from any of the 3 tables. This way it is transparent to the user in which physical table the data is stored. The user only uses this view to retrieve the data. the WHERE clauses are extremely important and don't serve just as documentation to show the ORDERKEY range of each table. This information is used by the optimizer to eliminate the scanning of certain partitions.

Example 11-12 DDL to create UNION in view

```

CREATE VIEW SC246300.VWORDER
AS
SELECT *
FROM SC246300.TBORDER_1
WHERE ORDERKEY BETWEEN 1 AND 700000000
UNION ALL
SELECT *
FROM SC246300.TBORDER_2
WHERE ORDERKEY BETWEEN 700000001 AND 1400000000
UNION ALL
SELECT *
FROM SC246300.TBORDER_3
WHERE ORDERKEY BETWEEN 1400000001 AND 2147483647 #

```

When the user executes the query in Example 11-13, DB2 is smart enough, based on the information in your query and the information in the view definition, to determine that the data can only come from TBORDER_1 and there is no need to look at any row in TBORDER_2 and TBORDER_3.

However, keep in mind that DB2, although the individual subqueries against each table can run in parallel, the first subquery has to complete before the next one starts. (DB2 has to finish selecting from TBORDER_1 before he starts working on TBORDER_2.)

Example 11-13 Sample SELECT from view to mask the underlying tables

```
SELECT *
FROM SC246300.VWORDER
WHERE ORDERKEY = 123456
AND CUSTKEY = CUSTOMER('000006') # -- CUSTKEY IS USING A UDT
-- called CUSTOMER. Therefore a
-- CAST function is required
```

When doing an insert, update or delete against the set of tables, you should use one of the following views shown in Example 11-14. The use of the WITH CHECK OPTION prevents you from inserting in the wrong table. An example of an insert using a wrong view is shown in Example 11-15. This is very important because the view that is used to retrieve rows, restricts the data that can be retrieved from each table. If, by accident, a row with an ORDERKEY of 555 would end up into table TBORDER_2, you would not be able to retrieve it using the VWORDER view. The WHERE clause associated with TBORDER_2 in the VWORDER view eliminates ORDERKEY from the result set.

Example 11-14 Views to use for UPDATE and DELETE

```
CREATE VIEW SC246300.VWORDER_1UPD
AS
SELECT *
FROM SC246300.TBORDER_1
WHERE ORDERKEY BETWEEN 1 AND 700000000
WITH CHECK OPTION

CREATE VIEW SC246300.VWORDER_2UPD
AS
SELECT *
FROM SC246300.TBORDER_2
WHERE ORDERKEY BETWEEN 700000001 AND 1400000000
WITH CHECK OPTION

UNION ALL

CREATE VIEW SC246300.VWORDER_3UPD
AS
SELECT *
FROM SC246300.TBORDER_3
WHERE ORDERKEY BETWEEN 1400000001 AND 2147483647
WITH CHECK OPTION
```

Example 11-15 WITH CHECK OPTION preventing INSERT

```
INSERT INTO SC246300.VWORDER_1UPD
VALUES (
    700000001 -- ORDERKEY INTEGER NOT NULL
    , '01' -- CUSTKEY CUSTOMER NOT NULL
    , 'N' -- ORDERSTATUS CHAR ( 1 ) NOT NULL WITH DEFAULT
    , 123.45 -- TOTALPRICE FLOAT NOT NULL
    , CURRENT DATE -- ORDERDATE DATE NOT NULL WITH DEFAULT
    , 'LOW' -- ORDERPRIORITY CHAR (15 ) NOT NULL WITH DEFAULT
    , 'BART' -- CLERK CHAR ( 6 ) NOT NULL WITH DEFAULT
    , 5 -- SHIPPRIORITY INTEGER NOT NULL WITH DEFAULT
    , 'NW' -- STATE CHAR ( 2 ) NOT NULL WITH DEFAULT
    , 55 -- REGION_CODE INTEGER
```

```
,CURRENT DATE -- INVOICE_DATE DATE NOT NULL WITH DEFAULT
,'MY ORDER' -- COMMENT VARCHAR ( 79 )
) #
```

```
-----
DSNT408I SQLCODE = -161, ERROR: THE INSERT OR UPDATE IS NOT ALLOWED BECAUSE A
RESULTING ROW DOES NOT SATISFY THE VIEW DEFINITION
DSNT418I SQLSTATE = 44000 SQLSTATE RETURN CODE
```

This type of view is not all that helpful when updating or deleting rows. However, you receive an SQLCODE +100 when you try to run the statement using the wrong view. This should be a signal that you might be using the wrong table, but is of course no guarantee that this is actually the case. It is also possible that the row you are trying to update or delete does not exist in the table. In trying also to close this loophole, at some extra cost, you could select the row first using the VWORDER view before trying to update or delete it using the correct VWORDER_xUPD view.

Note also that when your TBORDER tables need to be the parent table in an RI relationship, this construct cannot be used. A foreign key cannot point to a set of tables (our TBORDER_x tables).

In summary, although splitting very large tables into several smaller ones and using the union in view concept to make it transparent to the application has some attractive features, there are also several drawbacks that have to be carefully considered before implementing this design.



Scrollable cursors

The ability to be able to scroll backwards as well as forwards is a requirement of many screen-based applications. DB2 V7 introduces facilities not only to allow scrolling forwards and backwards, but also the ability to jump around and directly retrieve a row that is located at any position within the cursor result table.

DB2 also can, if desired, maintain the relationship between the rows in the result set and the data in the base table. That is, the scrollable cursor function allows the changes made outside the opened cursor, to be reflected. For example, if the currently fetched row has been updated while being processed by the user, and an update is attempted, a warning is returned by DB2 to reflect this. When another user has deleted the row currently fetched, DB2 returns an SQLCODE if an attempt is made to update the deleted row.

12.1 What is a scrollable cursor?

Scrollable cursors are cursors that allow scrolling in any direction. New keywords have been added to both the DECLARE CURSOR and FETCH statements to support scrollable cursors.

Cursors can be scrolled:

- ▶ Backward
- ▶ Forward
- ▶ To an absolute position within the result set
- ▶ To a position relative to the current cursor position
- ▶ Before/after the beginning/end of result set

12.2 Why use a scrollable cursor

Before DB2 V7, cursors could only be scrolled in a forward direction. Cursors were opened and would be positioned before the first row of the result set. To move through the cursor, a FETCH would be executed, and the cursor would move forward one row. A given row could only be fetched once. The application program had to rely on different techniques to get around this limitation.

With non-scrollable cursors, if we want to move backwards through a result set, we have a number of options. One option is to declare 2 cursors, one which uses an ascending index, and one which uses a descending index. This requires additional program logic and complexity since two cursors are needed. Also additional resources are consumed since a second index is needed.

Another option is to CLOSE and reopen the current cursor to start at the beginning, then repeat the FETCH until the desired row is reached. This can have a large negative impact on response times for large result sets. Many rows may be read unnecessarily on the way to the target row. Also, if another process inserts or deletes rows, the relative position of the row can change and may cause the wrong row to be accessed. Program logic has to be built to either ignore or somehow handle the changes in row positions.

Yet another alternative is to cache the results in working storage. The result set can be opened and an arbitrary number of the rows can be read into an array. The program may, then move backward and forward within this array. This option needs to be carefully planned, as it often wastes memory through low utilization of the space, or it restricts the number of rows returned to an arbitrary number. If other processes are changing the data, the program is insensitive to the changes.

With scrollable cursors, DB2 provides the ability to scroll in any direction and even the ability to skip around within the result table. This can greatly reduce program complexity by simplifying logic. Another benefit of scrollable cursors (compared to the case where you need to have multiple indexes or sort the result table in order to scroll backward) is that they can avoid some sorts and reduce the need for extra indexes on your tables. Also, since a scrollable cursor always materializes the result table, an insensitive scrollable cursor may be a good way to build a point in time result table that your program can process without having to lock the base table. Scrollable cursors are also ideal for remote applications that build screens and allow end users to scroll or skip around through the data being displayed.

12.3 Scrollable cursors characteristics

In this section, we discuss the characteristics of scrollable cursors.

12.3.1 Types of cursors

To understand scrollable cursors characteristics, let us compare the different kinds of cursors:

Non-scrollable cursor

- ▶ Used by an application program to retrieve a set of rows or retrieve a result set from a stored procedure.
- ▶ The rows must be processed one at a time.
- ▶ The rows are fetched sequentially.
- ▶ Result sets may be stored in a workfile.

Scrollable cursor

- ▶ Used by an application program to retrieve a set of rows or retrieve a result set from a stored procedure.
- ▶ The rows can be fetched in random order.
- ▶ The rows can be fetched forward or backward.
- ▶ The rows can be fetched relative to the current position or from the top of the result table or result set.
- ▶ The result set is fixed at OPEN CURSOR time.
- ▶ Result sets are stored in declared temporary tables.
- ▶ Result sets go away at CLOSE CURSOR time.

Insensitive scrollable cursor

- ▶ Always static.
- ▶ Fixed number of rows.
- ▶ Results stored in declared temporary tables.

Sensitive static scrollable cursor

- ▶ Always static.
- ▶ Fixed number of rows.
- ▶ Results stored in declared temporary tables.
- ▶ Sensitive to changes, but not to inserts.

Sensitive dynamic scrollable cursor (not currently supported)

- ▶ Direct table access.
- ▶ Sensitive to changes and inserts.

Note: DB2 Version 7 supports a subset of the SQL99 standard for scrollable cursors. Sensitive DYNAMIC scrollable cursors are not supported by DB2 Version 7. To allow for possible support for sensitive dynamic cursors in a future release of DB2, the keyword STATIC must be explicitly specified for a sensitive scrollable cursor.

12.3.2 Scrollable cursors in depth

A significant problem with previous methods of cursor management was in maintaining the relationship between the data being updated by the cursor and the actual data in the base table.

DB2 V7 introduces new keywords INSENSITIVE and SENSITIVE STATIC for the DECLARE CURSOR statement to control whether the data in the result set is validated against the data in the base table. DB2 now ensures that only the current values of the base table are updated and recognizes where rows have been deleted from the result set. It can, if required, refresh the rows in the result set at fetch time to ensure that the data under the cursor is current.

Basically, INSENSITIVE means that the cursor is read-only and is not interested in changes made to the base data once the cursor is opened. With SENSITIVE, the cursor is interested in changes which may be made after the cursor is opened. The levels of this awareness are dictated by the combination of SENSITIVE STATIC in the DECLARE CURSOR statement and whether INSENSITIVE or SENSITIVE is defined in the FETCH statement.

Tip: INSENSITIVE cursors are strictly read-only. SELECT statements using with FOR UPDATE OF must use SENSITIVE STATIC cursors in order to be updatable scrollable cursors.

When creating a scrolling cursor the INSENSITIVE or SENSITIVE STATIC, keywords must be used in the DECLARE CURSOR statement. This sets the default behavior of the cursor.

Insensitive

If an attempt is made to code the FOR UPDATE OF clause in a cursor defined as INSENSITIVE, then the bind returns an SQLCODE:

```
-228 FOR UPDATE CLAUSE SPECIFIED FOR READ-ONLY SCROLLABLE CURSOR USING  
cursor-name.
```

The characteristics of an insensitive scrollable cursor are:

- The cursor **cannot** be used to issue positioned updates and deletes.
- FETCH processing on the result table is *insensitive* to changes made to the base table after the result table is built (even if changes are made by the current agent outside the cursor).
- The number and content of the rows stored in the result table is fixed at OPEN CURSOR time and **does not change**.

Sensitive

Fundamentally, the SENSITIVE STATIC cursor is updatable. As such, the FOR UPDATE OF clause can be coded for a SENSITIVE cursor.

If the SELECT statement connected to a cursor declared as SENSITIVE STATIC uses any keywords that forces the cursor to be read-only, the bind rejects the cursor declaration. In this case the bind returns an SQLCODE:

-243 SENSITIVE CURSOR cursor-name CANNOT BE DEFINED FOR THE SPECIFIED SELECT STATEMENT.

Important: Use of column functions, such as MAX and AVG, and table joins forces a scrollable cursor into implicit read-only mode and therefore are not valid for a SENSITIVE cursor. In contrast with non-scrollable cursors, the usage of the ORDER BY clause in a scrollable cursor does not make it read-only.

A SENSITIVE cursor can be made explicitly read-only by including FOR FETCH ONLY in the DECLARE CURSOR statement. Even if a SENSITIVE cursor is read-only, it is still aware of all changes made to the base table data through updates and deletes.

The characteristics of a sensitive scrollable cursor are:

- The cursor can be used to issue positioned updates and deletes
- FETCH processing on the result table is sensitive (to varying degrees) to changes made to the base table after the result table has been built
- The number of rows in the result table does not change but the row content can change
- STATIC cursors are insensitive to inserts.

Sensitive but read-only

A read-only cursor is one which cannot be used to issue positioned updates and deletes.

A cursor can be made read-only in 3 ways:

- The DECLARE CURSOR statement specifies FOR FETCH ONLY or FOR READ ONLY.
- The DECLARE CURSOR statement specifies INSENSITIVE SCROLL CURSOR.
- The SELECT statement of the DECLARE CURSOR statement can implicitly make the cursor read-only. DB2 cannot make changes visible to the cursor when the cursor implicitly becomes read-only. For example, a cursor is read-only if the SELECT column-list contains a column function. The current list of conditions that result in an implicit read-only cursor can be found in the DECLARE CURSOR section of the *DB2 UDB for OS/390 and z/OS Version 7 SQL Reference*, SC26-9944.

If a cursor is implicitly made read-only, the cursor cannot be declared as a sensitive scrollable cursor. If you attempt to declare such a cursor, you get an SQLCODE -243.

An ORDER BY clause does not make a sensitive scrollable cursor read-only. However, the order of the rows in the result table remains constant after the cursor has been opened, that is, if a value of an ordering column is updated, the row in the result table is not moved to a new position. Note that this is different from a non-scrollable cursor. A non-scrollable cursor is NOT updatable when an ORDER BY clause is present.

12.4 How to choose the right type of cursor

Having the ability to scroll backward and forward also means that you must decide whether changes to previously fetched data need to be seen or not. Specifically, does your application need to detect updated and deleted rows. This decision to be sensitive to changes is significant because a result table defines a set of rows that meet a certain criteria. However, while the application is scrolling in the result table, previously fetched rows may be excluded or new rows may be included if the table is updated.

Some applications may require that a result table remains constant (STATIC) as the application scrolls through it. For example, some accounting applications require data to be constant. On the other hand, other applications, like airline reservations, may require to see the latest flight availability no matter how much they scroll through the data (DYNAMIC).

Furthermore, sensitivity can be limited to visibility of changes made by the same cursor and process that is fetching rows or the sensitivity can be extended to also see updates made outside the cursor and process by refreshing the row explicitly.

Table 12-1 can be used to help you decide which type of cursor to use. If you just want to blast through your data then choose a forward-only cursor. If you want to scroll through a constant copy of your data you may want to use an INSENSITIVE CURSOR instead of making a regular cursor materialize the result table. If you want to control the cursor's position, scroll back and forth, choose a scrollable cursor. If you don't care about the freshness of data, choose an INSENSITIVE CURSOR. If you want fresh data some times, choose a SENSITIVE on DECLARE CURSOR and SENSITIVE or INSENSITIVE on FETCH. If you want fresh data all the time, choose a SENSITIVE CURSOR and SENSITIVE on FETCH or non specific FETCH.

Table 12-1 Cursor type comparison

Cursor type	Result table	Visibility of own cursor's changes	Visibility of other cursors' changes	Updatability
Non-scrollable Materialized	Fixed, workfile	No	No	No
Non-Scrollable Not Materialized	No workfile, base table access	Yes	Yes	Yes
INSENSITIVE SCROLL	Fixed, declared temp table	No	No	No
SENSITIVE STATIC SCROLL	Fixed, declared temp table	Yes (Inserts not allowed)	Yes (Not to inserts)	Yes
<i>SENSITIVE ** DYNAMIC SCROLL</i>	<i>No declared temp tables, base table access</i>	Yes	Yes	Yes

****Note:** Sensitive Dynamic scrollable cursors are not available as of DB2 V7 and are only shown here for comparison purposes.

12.5 Using a scrollable cursor

Scrollable cursors can be used in static and dynamic SQL compiled programs, compiled stored procedures (including SQL stored procedures). Scrollable cursors cannot be used in SPUFI, QMF, REXX programs and Java programs. Client programs using DB2 Connect V7.1 with Fixpack 2 can use scrollable cursors.

Scrollable cursors can be used in CICS conversational programs and batch processing using TSO, batch, CAF, RRSF, a background CICS task, DL/1 batch, and IMS BMP. They do not apply to CICS pseudo-conversational and IMS/TM transactions because resources are freed after displaying a screen (including declared temporary tables that scrollable cursors use under the cover).

In this section we discuss how to use a scrollable cursor. We discuss the following topics:

- ▶ Declaring a scrollable cursor
- ▶ Opening a scrollable cursor
- ▶ Fetching rows from a scrollable cursor
- ▶ Moving the cursor
- ▶ Using functions in a scrollable cursor

12.5.1 Declaring a scrollable cursor

To declare a scrollable cursor, you use the **SCROLL** keyword in the **DECLARE CURSOR** statement. The **FETCH** statements for this cursor can now be used to move in any direction in the result set defined by the **OPEN CURSOR** statement.

The new keywords **INSENSITIVE** and **SENSITIVE STATIC** of the **DECLARE CURSOR** statement deal with the sensitivity of the cursor to changes made to the underlying table.

The **STATIC** keyword in the context of this clause does not refer to static and dynamic SQL, as scrollable cursors can be used in both these types of SQL. Here, **STATIC** refers to the size of the result table, once the **OPEN CURSOR** is completed, the number of rows remains constant.

In Example 12-1, we show you how to **DECLARE** two scrollable cursors, one **INSENSITIVE** and the other **SENSITIVE STATIC**

Example 12-1 Sample DECLARE for scrollable cursors

```
DECLARE C1  INSENSITIVE SCROLL CURSOR
           FOR  SELECT  .... FROM  ....
                WHERE  .... ;

DECLARE C2  SENSITIVE STATIC SCROLL CURSOR
           FOR  SELECT  .... FROM  ....
                WHERE  ....
           FOR UPDATE OF  .... ;
```

12.5.2 Opening a scrollable cursor

When a scrollable cursor is opened, that is the **DECLARE *cursor-name*...SCROLL** and **OPEN CURSOR** statements are executed (see Example 12-2), the qualifying rows are copied to a declared temporary table which is automatically created by DB2 in the DB2 TEMP database. Therefore, before scrollable cursors can be used, you must make sure the TEMP database has been defined and that adequate space has been allocated in this database. User-declared temporary tables and system-declared temporary tables (used by scrollable cursors) share the set of table spaces that are defined in the TEMP database. Example 7-8 on page 89 shows how to create the TEMP database and several tables paces.

The temporary table that is created is only accessible by the agent that created it. For each user and for each scrollable cursor a temporary table is created at **OPEN CURSOR** time. The temporary table is dropped when you close the cursor or at the completion of the program that invoked it. For more information on declared temporary tables, see “Declared temporary tables” on page 88.

The record identifier (RID) of the row is also retrieved and stored with the rows in the temporary table. If the cursor is declared as SENSITIVE STATIC, the RID's are used to maintain changes between the result set row and the base table row.

Note: DECLARE CURSOR statements that do not use the new keyword SCROLL do not create the temporary table and are only able to scroll in a forward direction.

It is important to note that, for a cursor which is declared as INSENSITIVE or SENSITIVE STATIC, the number of rows of the result set table does not change once the rows are retrieved from the base table and stored. This means that all subsequent inserts which are made by other users or by the current process into the base table and which would fit the selection criteria of the cursor's SELECT statement are not visible to the cursor. Only updates and deletes to data within the result set may be seen by the cursor.

Once the result set has been retrieved, it is only visible to the current cursor process and remains available until a CLOSE CURSOR is executed or the process itself completes. For programs, the result set is dropped on exit of the current program; for stored procedures, the cursors defined are allocated from the calling program, and the result set is dropped when the calling program concludes.

Example 12-2 Opening a scrollable cursor

```
DECLARE CUR1 SENSITIVE STATIC SCROLL CURSOR
  WITH HOLD
  FOR SELECT   ACCOUNT
              ,ACCOUNT_NAME,
              ,CREDIT_LIMIT
              ,TYPE
  INTO        :ACCOUNT
              ,:ACCOUNT_NAME,
              ,:CREDIT_LIMIT
              ,:TYPE
  FROM ACCOUNT
             WHERE CREDIT_LIMIT > 20000 ;
...
OPEN CUR1
...
```

If a LOB column is selected in the DECLARE CURSOR statement, the LOB column is represented by a LOB descriptor column in the result table. The LOB descriptor column is 120 bytes and holds information which enables DB2 to quickly retrieve the associated LOB column value from the auxiliary table when the application fetches a row from the result table. DB2 has to retrieve the LOB column value (from the auxiliary table) when it processes either a sensitive or an insensitive FETCH request.

Suppose an application issues a ROLLBACK TO SAVEPOINT S1, and savepoint S1 was set before scrollable cursor C1 was opened. Once the rollback has completed, the result table for C1 contains the same data as it did on completion of the open cursor statement for C1. In addition, the rollback does not change the position of cursor C1.

The OPEN CURSOR and ALLOCATE CURSOR statement return the following information in the SQLCA for scrollable cursors regarding the sensitivity of the cursor even though the SQLCODE and SQLSTATE are zero:

- ▶ Whether the cursor is scrollable or not

This information is provided in the SQLWARN1 field. It is set to:

- S = scrollable and
- N = non-scrollable cursor

- ▶ Effective sensitivity of the cursor (that is, insensitive versus sensitive):

This information is returned in SQLWARN4. SQLWARN4 is not set for non-scrollable cursors.

- I = insensitive
- S = sensitive

- ▶ Effective capability of the cursor (that is, whether the scrollable cursor is updatable, deletable or read-only):

This information is returned in SQLWARN5. SQLWARN5 is not set for non-scrollable cursors.

- 1 = Read-only, the result table of the query is read-only either because the content of the SELECT statement (implicitly read-only), or for READ/FETCH ONLY was explicitly specified.
- 2 = Read and Delete allowed, the result table of the query is deletable, but not updatable.
- 4 = Read, Delete and Update allowed, the result table of the query is deletable and updatable.

When SQLWARN1, SQLWARN4 and SQLWARN5 are set, then SQLWARN0 (the summary flag) is NOT set for these cases.

Note: If the size of the result table exceeds the DB2 established limit of declared temporary tables, a resource unavailable message, DSNT501I, is generated with the appropriate resource reason code at OPEN CURSOR time.

12.5.3 Fetching rows

The FETCH statement positions a cursor on a row of the result table. It can return zero or one row and assign the values of the row returned to host variables.

Figure 12-1 shows the syntax diagram for the FETCH statement. The syntax has been expanded (items in **bold**) to allow this statement to control cursor movement both forwards and backwards. It also has keywords to position the cursor in specific positions within the result set returned by the OPEN CURSOR statement.

Tip: The BEFORE and AFTER clauses are positioning orientation, which means that no data is returned and an SQLCODE = 0 is returned to the application.

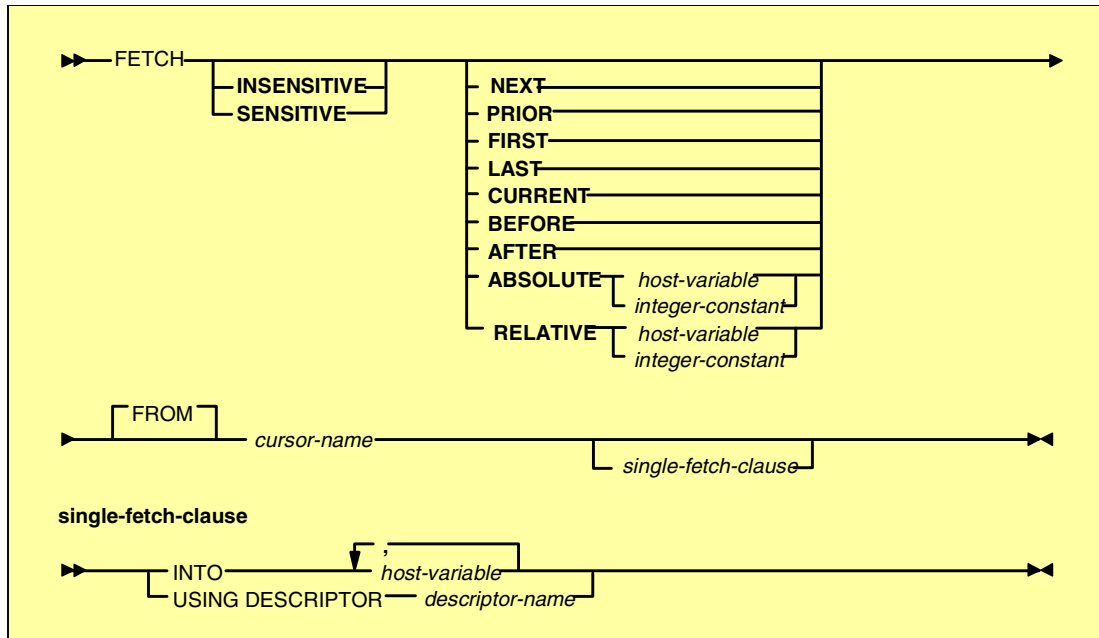


Figure 12-1 Fetch syntax changes to support scrollable cursors

Below is a complete list of the new keywords which have been added to the FETCH statement syntax for moving the cursor.

- | | |
|----------------|---|
| NEXT | Positions the cursor on the next row of the result table relative to the current cursor position and fetches the row - This is the default. |
| PRIOR | Positions the cursor on the previous row of the result table relative to the current position and fetches the row. |
| FIRST | Positions the cursor on the first row of the result table and fetches the row. |
| LAST | Positions the cursor on the last row of the result table and fetches the row. |
| CURRENT | Fetches the current row without changing position within the result table.
If CURRENT is specified and the cursor is not positioned at a valid row (for example, BEFORE the beginning of the result table) a warning SQLCODE +231, SQLSTATE 02000 is returned. |
| BEFORE | Positions the cursor before the first row of the result table.
No output host variables can be coded with this keyword as no data can be returned |
| AFTER | Positions the cursor after the last row of the result table
No output host variables can be coded with this keyword, as no data can be returned. |

ABSOLUTE Used with either a *host-variable* or *integer-constant*. This keyword evaluates the *host-variable* or *integer-constant* and fetches the data at the row number specified.

If the value of the *host-variable* or *integer-constant* is 0, then the cursor is positioned at the position before the first row and the warning SQLCODE +100, SQLSTATE 02000 is returned.

If the value of the *host-variable* or *integer-constant* is greater than the count of rows in the result table, the cursor is positioned after the last row in the result table, and the warning SQLCODE +100, SQLSTATE 02000 is returned.

RELATIVE Used with either a *host-variable* or *integer-constant*. This keyword evaluates the *host-variable* or *integer-constant* and fetches the data in the row which is that value away from the current cursor position.

If the value in the *host-variable* or *integer-constant* is equal to 0, then the current cursor position is maintained and the data fetched.

If the value in the *host-variable* or *integer-constant* is less than 0, then the cursor is positioned the number of rows specified in the *host-variable* or *integer-constant* from the cursor position towards the beginning of the result table and the row is fetched.

If the value in the *host-variable* or *integer-constant* is greater than 0, then the cursor is positioned the number of rows specified in the *host-variable* or *integer-constant* from the cursor position towards the end of the result table and the row is fetched.

If a relative position is specified that is before the first row or after the last row, a warning SQLCODE +100, SQLSTATE 02000 is returned, and the cursor is positioned either before the first row or after the last row and no data is returned.

In Figure 12-2 on page 167, we show you the results of various fetches when the cursor is currently positioned on the 10th row. In Figure 12-3 on page 168, we show you a matrix of the possible SQLCODEs returned after a FETCH from a scrollable cursor.

Sensitive and insensitive FETCH

There is also the facility within the related FETCH statements to further specify the way in which the cursor interacts with data in the base table. This is done by specifying INSENSITIVE or SENSITIVE in the FETCH statement itself. If these keywords are not used in the FETCH statement, then the sensitivity attributes of the DECLARE CURSOR statement are used. For example, suppose the DECLARE CURSOR is coded as:

```
DECLARE CUR1 SENSITIVE STATIC SCROLL CURSOR
FOR SELECT ACCOUNT
      ,ACCOUNT_NAME
FROM SC246300.TBACCOUNT
FOR UPDATE OF ACCOUNT_NAME ;
```

And, the FETCH statement is defined as:

```
FETCH CUR1 INTO :hvaccount, :hvacct_name;
```

In this case, the cursor uses the SENSITIVE characteristics.

A `FETCH INSENSITIVE` request retrieves the row data from the result table. A `FETCH SENSITIVE` request retrieves the row data from the base table.

Tip: You want to specify `FETCH SENSITIVE` when you want DB2 to check if the underlying data has changed since you last retrieved it from the base table and you intend to update or delete the row or you simply need the latest data. See “Maintaining updates” on page 174 for more details.

In `SENSITIVE STATIC` scrollable cursors, the number of rows in the result table is fixed but deletes and updates to the underlying table can create delete holes or update holes. In “Update and delete holes” on page 170 we discuss in detail how these holes are created.

Allowable combinations of `CURSOR` and `FETCH` sensitivity

The sensitivity to changes made to the base table (see Table 12-2) is specified at two levels:

- ▶ On the `DECLARE CURSOR` statement (discussed in topic “Scrollable cursors in depth” on page 152).
- ▶ On the `FETCH` statement (discussed in the previous topic).

Let’s take a closer look at the allowable combinations that can be used and the characteristics of their attributes.

`CURSOR INSENSITIVE` and `FETCH INSENSITIVE`

A DB2 temporary table is created at `OPEN CURSOR` time and filled with rows which match the selection criteria. Once the result table is built, reads never reference the rows in the base table.

Once the cursor has been opened, if an update is made to a row in the base table that would create an update or delete hole in the cursor’s result set, this cursor does not recognize it. A fetch against a row in the result set whose matching base table row has been deleted or updated still returns an `SQLCODE` of 0 and return the row values to the host variables as they were set at `OPEN CURSOR` time.

The resultant cursor is read-only. If a `FOR UPDATE OF` clause is coded in the `DECLARE CURSOR SELECT` statement, then the following SQL code is returned at bind time:

```
-228 FOR UPDATE CLAUSE SPECIFIED FOR READ-ONLY SCROLLABLE CURSOR USING  
cursor-name
```

If the cursor has been defined as `INSENSITIVE` and the `FOR UPDATE OF CURSOR` clause is coded in the `FETCH` statement, then the following SQL code is returned at bind time and the bind fails:

```
-510 THE TABLE DESIGNATED BY THE CURSOR OF THE UPDATE OR DELETE STATEMENT CANNOT  
BE MODIFIED
```

`CURSOR INSENSITIVE` and `FETCH SENSITIVE`

This is not a valid combination. If a `FETCH SENSITIVE` statement is coded following for an `INSENSITIVE` cursor, then the following SQL code is returned at runtime:

```
-224 SENSITIVITY sensitivity SPECIFIED ON THE FETCH IS NOT VALID FOR CURSOR  
cursor-name
```

`CURSOR SENSITIVE STATIC` and `FETCH INSENSITIVE`

A temporary table is created at `OPEN CURSOR` time with all rows that match the select criteria.

Updates and deletes can be made in a SENSITIVE STATIC cursor using the clause WHERE CURRENT OF in UPDATE and DELETE statements. However, when the INSENSITIVE keyword is used in the FETCH statement, we are saying that we do not want the cursor to reflect updates or deletes of the cursor's rows made by statements outside the cursor. In this case, the FETCH statement shows holes made by updates and deletes from within the current cursor, but does not show any holes created by any update and deletes outside the cursor.

If the application issues a FETCH INSENSITIVE request it sees the positioned updates and deletes it has made via the current scrollable cursor. These changes are visible to the application because DB2 updates both the base table and the result table when a positioned update or delete is issued by the application owning the cursor. Changes made by agents outside the cursor is not visible to data returned by the FETCH INSENSITIVE.

CURSOR SENSITIVE STATIC and FETCH SENSITIVE

A temporary table is created at open cursor time to hold all rows returned by the SELECT statement.

Updates and deletes can be made using the WHERE CURRENT OF CURSOR clause. If an update is to be made against the cursor, the FOR UPDATE OF must be coded in the DECLARE CURSOR statement.

The FETCH statement returns all updates or deletes made by the cursor, all updates and deletes made outside the cursor and within the current process, and all committed updates and deletes made to the rows within the cursor's result set by all other processes.

As you would expect, the application sees the uncommitted changes as well as the committed changes it has made to the base table. However, it does not see uncommitted changes made by other agents unless isolation level UR is in effect for the scrollable cursor.

Table 12-2 Sensitivity of FETCH to changes made to the base table

Specification on DECLARE CURSOR	Specification on FETCH	Comment	Sensitivity to changes made to the base table
INSENSITIVE	INSENSITIVE	Default for FETCH is INSENSITIVE	None
INSENSITIVE	SENSITIVE	Invalid combination	Not applicable
SENSITIVE	INSENSITIVE	Valid combination	Application sees the changes it has made using positioned UPDATES and DELETES
SENSITIVE	SENSITIVE	Default for FETCH is SENSITIVE	Application sees: <ol style="list-style-type: none"> Changes it has made using positioned and searched UPDATES and DELETES Changes it has made outside the cursor (using searched UPDATES and DELETES) Committed UPDATES and DELETES made by other applications

Tip: A FETCH request never sees new rows which have been INSERTed into the base table after the result table has been built.

How FETCH SENSITIVE request are processed

Each row in the result table of a sensitive scrollable cursor contains the RID value of the corresponding row in the base table. DB2 uses the RID value to retrieve the base table row when a FETCH SENSITIVE, positioned update or positioned delete request are processed.

DB2 does not allow RIDs to be re-used in the base table space when a sensitive cursor is open. If DB2 allowed RIDs to be re-used, then it would be possible for the following sequence of events to occur:

1. Another application could delete a base table row and then insert a new row at the same RID location.
2. A FETCH SENSITIVE request would retrieve the new row but this new row would not correspond to the result table row established at open cursor time.

Suppose you have defined and opened the following scrollable cursor:

```
DECLARE C1 SENSITIVE STATIC SCROLL CURSOR
FOR SELECT ....
      FROM BASE_TABLE
      WHERE ....
FOR UPDATE OF ....
```

When you issue the following fetch:

```
FETCH SENSITIVE ABSOLUTE 4
FROM C1 INTO :hv1, :hv2, :hv3, :hv4
```

DB2 attempts to retrieve the corresponding row from the base table. If the row is not found, DB2 marks the result table row as a delete hole and returns SQLCODE +222. If row is found, DB2 checks to see if the base table row still satisfies the search condition specified in the DECLARE CURSOR statement. If the row no longer satisfies the search condition, DB2 marks the result table row as an update hole and returns SQLCODE +222. If the row still satisfies the search condition, DB2 refreshes the result table row with the column values from the base table row and returns the result table row to the application.

When a FETCH request encounters a hole in the result table

Here is what happens when a FETCH request encounter a hole in the result table:

- ▶ If a FETCH INSENSITIVE or FETCH SENSITIVE request encounters a delete hole, DB2 returns SQLCODE +222.
- ▶ If a FETCH INSENSITIVE request encounters an update hole, DB2 returns SQLCODE +222. This situation can only occur if you have a sensitive cursor with and insensitive FETCH and you create an update hole with an UPDATE WHERE CURRENT OF on this scrollable cursor.
- ▶ If a FETCH SENSITIVE request encounters an update hole, DB2 checks to see if the corresponding row in the base table now satisfies the search condition that was specified in the original DECLARE CURSOR statement.
 - If it does, the row in the result table is marked as valid and DB2 refreshes the result table row with the column values from the base table row and returns the result table row to the application.
 - If it does not, the result table row remains an update hole and DB2 returns SQLCODE +222.

A delete or update hole is counted as a row by DB2 when a FETCH request is processed, that is, holes are never skipped over.

An application cannot distinguish between a delete hole and an update hole.

Once a row in the result table is marked as a delete hole, then that row remains a delete hole.

In Example 12-3, we show a FETCH SENSITIVE request which creates an update hole in the result table. The update hole is created because the base table row no longer satisfies the search condition WHERE TXNID = 'SMITH'.

Example 12-3 Example of a FETCH SENSITIVE request which creates an update hole

```

DECLARE CURSOR C1 SENSITIVE STATIC SCROLL
FOR SELECT TXNDATE
           ,AMT
FROM TBTRAN
WHERE TXNID = 'SMITH'
ORDER BY TXNDATE
    
```

Sequence of events:

1. Cursor C1 is opened and the result table is built.
2. Another application updates the column value of TXNID of the row with RID A0D to "SMYTHE".
3. A FETCH SENSITIVE is invoked that positions the cursor on the row with RID A0D.
4. DB2 checks the base table row (using its RID), since the row no longer satisfies the search condition WHERE TXNID = 'SMITH', DB2 changes the current row in the result table to an update hole, and returns an SQLCODE +222.

Before update hole is created:

BASE TABLE TBTRAN					RESULT TABLE		
	TXNID	TXNDATE	DESC	AMT	RID	TXNDATE	AMT
RID	-----+	-----+	-----+	-----	----	-----+	-----
902	Brown	080200	CR	+500	903	071200	+500
903	Smith	071200	CR	+500	A0F	080100	+200
A04	Brown	080900	DB	-20	A0C	080200	+500
A05	Doe	081000	CR	+500	A07	080300	-40
A07	Smith	080300	DB	-40	A0D	080400	+100
A08	George	080800	DB	-100	A09	081600	+800
A09	Smith	081600	CR	+800			
A0A	Black	081700	CR	+200			
A0B	White	071100	DB	-50			
A0C	Smith	080200	CR	+500			
A0D	Smith	080400	CR	+100			
A0E	Black	080500	DB	-500			
A0F	Smith	080100	CR	+200			

After update hole is created:

BASE TABLE TBTRAN					RESULT TABLE		
	TXNID	TXNDATE	DESC	AMT	RID	TXNDATE	AMT
RID	-----+	-----+	-----+	-----	----	-----+	-----
902	Brown	080200	CR	+500	903	071200	+500
903	Smith	071200	CR	+500	A0F	080100	+200
A04	Brown	080900	DB	-20	A0C	080200	+500
A05	Doe	081000	CR	+500	A07	080300	-40
A07	Smith	080300	DB	-40	A0D		
A08	George	080800	DB	-100	A09	081600	+800
A09	Smith	081600	CR	+800			

AOA	Black	081700	CR	+200
AOB	White	071100	DB	-50
AOC	Smith	080200	CR	+500
AOD	Smythe	080400	CR	+100
AOE	Black	080500	DB	-500
AOF	Smith	080100	CR	+200

12.5.4 Moving the cursor

To achieve moving the cursor both backwards and forwards within a result set, two types of commands have been provided for cursor movement.

- ▶ The first category allows for positioning on specific rows within the result set, based on the first row being row number 1. These are called absolute moves.
- ▶ The second type allows for movement relative to the current cursor position, such as moving five rows back from the current cursor position. These are known as relative moves.

Cursor movement

The RELATIVE and ABSOLUTE keywords can be followed by either an integer constant or a host variable which contains the value to be used.

The INTO clause specifies the host variable(s) into which the row data should be fetched. This clause must be specified for all FETCH requests except for a FETCH BEFORE and a FETCH AFTER request. An alternative to the INTO clause is the INTO DESCRIPTOR clause.

If the RELATIVE or ABSOLUTE keyword is followed by the name of a host variable, then the named host variable must be declared with a data type of INTEGER or DECIMAL(n,0). Data type DECIMAL(n,0) only has to be used if a number specified is beyond the range of an integer (-2147483648 to +2147483647). If the number of the row to be fetched is specified via a constant, then the constant must be an integer. For example, 7 is valid, 7.0 is not valid.

Absolute moves

An absolute move is one where the cursor position is moved to an absolute position within the result table. For example, if a program wants to retrieve the fourth row of a table, the FETCH statement would be coded as:

```
EXEC SQL
    FETCH ... ABSOLUTE +4 FROM CUR1 INTO ...
END-EXEC.

or

01 CURSOR-POSITION    PIC S9(9) USAGE BINARY.
..
MOVE 4 TO CURSOR-POSITION.
EXEC SQL
    FETCH ... ABSOLUTE :CURSOR-POSITION FROM CUR1 INTO ...
END-EXEC.
```

Here, *:CURSOR-POSITION* is a host-variable of type INTEGER.

Another form of the absolute move is through the use of keywords which represent fixed positions within the result set. For example, to move to the first row of a result set, the following FETCH statement can be coded:

```
FETCH ... FIRST FROM CUR1
```

This statement can also be coded as:

```
FETCH ... ABSOLUTE +1 FROM CUR1
```

There are also two special absolute keywords which allow for the cursor to be positioned outside the result set. The keyword **BEFORE** is used to move the cursor before the first row of the result set and **AFTER** is used to move to the position after the last row in the result set. Host variables cannot be coded with these keywords as they can never return values.

A **FETCH ABSOLUTE 0** request and a **FETCH BEFORE** request both position the cursor before the first row in the result table. DB2 returns **SQLCODE +100** for **FETCH ABSOLUTE 0** requests (that is, no data is returned) and **SQLCODE 0** for **FETCH BEFORE** requests.

A **FETCH ABSOLUTE -1** request is equivalent to a **FETCH LAST** request, that is, both requests fetch the last row in the result table. DB2 returns **SQLCODE 0** for both of these requests.

Tip: A **FETCH BEFORE** and a **FETCH AFTER** request only position the cursor, DB2 does not return any row data. The **SENSITIVE** and **INSENSITIVE** keywords cannot be used if **BEFORE** or **AFTER** are specified on the **FETCH** statement.

Relative moves

A relative move is one made with reference to the current cursor position. To code a statement which moves three rows back from the current cursor, the statement would be:

```
FETCH ... RELATIVE -3 FROM CUR1 INTO ... ;  
or  
MOVE -3 TO CURSOR-MOVE.  
FETCH ... RELATIVE :CURSOR-MOVE FROM CUR1 INTO ... ;
```

Here, **CURSOR-MOVE** is a host-variable of **INTEGER** type.

If you attempt to make a relative jump which positions you either before the first row or after the last row of the result set, an **SQLCODE** of **+100** is returned. In this case the cursor is positioned just before the first row, if the jump was backwards through the result set; or just after the last row, if the jump was forward within the result set.

The keywords **CURRENT**, **NEXT**, and **PRIOR** make fixed moves relative to the current cursor position. For example, to move to the next row, the **FETCH** statement would be coded as:

```
FETCH ... NEXT FROM CUR1 ;  
or  
FETCH ... FROM CUR1 ;
```

Please refer to the *DB2 UDB for OS/390 and z/OS Version 7 SQL Reference*, SC26-9944, for a complete list of synonymous scroll specifications for **ABSOLUTE** and **RELATIVE** moves inside a scrollable cursor.

In Example 12-4 we show you sample program logic to display the last five rows from a table.

Example 12-4 Scrolling through the last five rows of a table

```
DECLARE CURSOR CUR1 SENSITIVE STATIC SCROLL  
FOR SELECT    TXNID  
              ,TXSTATUS  
              ,TXNDATE  
              ,AMT
```

```

        FROM TBTRAN
        WHERE TXNID = 'SMITH'
        AND TXSTATUS = 'LATE'
        ORDER BY TXNDATE ;

OPEN CURSOR CUR1 ;

FETCH INSENSITIVE ABSOLUTE -6      -- Position us on the 6th row from the bottom
FROM CUR1 INTO :HV1                -- of the result table
        ,:HV2 ;

DO I = 1 TO 5
    FETCH NEXT FROM CUR1 INTO :HV1
        ,:HV2 ;

        application logic to process the rows

END

CLOSE CURSOR CUR1 ;

```

Some examples of the full syntax are shown in Example 12-5. In this example, first we fetch the 20th row from the result table. Then we position the cursor to the beginning of the result table (no data is returned). Then we scroll down to the “NEXT” row (in this case the first row in the result table). Then we scroll forward 10 rows. After that we scroll forward an additional :rownumhv rows. Then we position the cursor at the end of the result table (no data is returned). Then we scroll backwards 4 rows from the bottom of the result table, and finally we scroll forward to the next row from there.

Example 12-5 Several FETCH SENSITIVE statements

```

FETCH ABSOLUTE 20 FROM C1 INTO :hv1, :hv2 ;
FETCH BEFORE FROM C1 ;
FETCH NEXT FROM C1 INTO :hv1, :hv2 ;
FETCH SENSITIVE RELATIVE 10 FROM C1 INTO :hv1, :hv2 ;
FETCH SENSITIVE RELATIVE :rownumhv FROM C1 INTO :hv1, :hv2 ;
FETCH AFTER FROM C1 ;
FETCH INSENSITIVE PRIOR FROM C1 INTO :hv1, :hv2 ;
FETCH SENSITIVE RELATIVE -4 USING DESCRIPTOR :sqldahv ;
FETCH FROM C1 INTO :hv1, :hv2 ;

```

In Figure 12-2 we show the effects of different FETCH requests when the cursor is currently positioned on row number 10. NEXT is the default for a FETCH request.

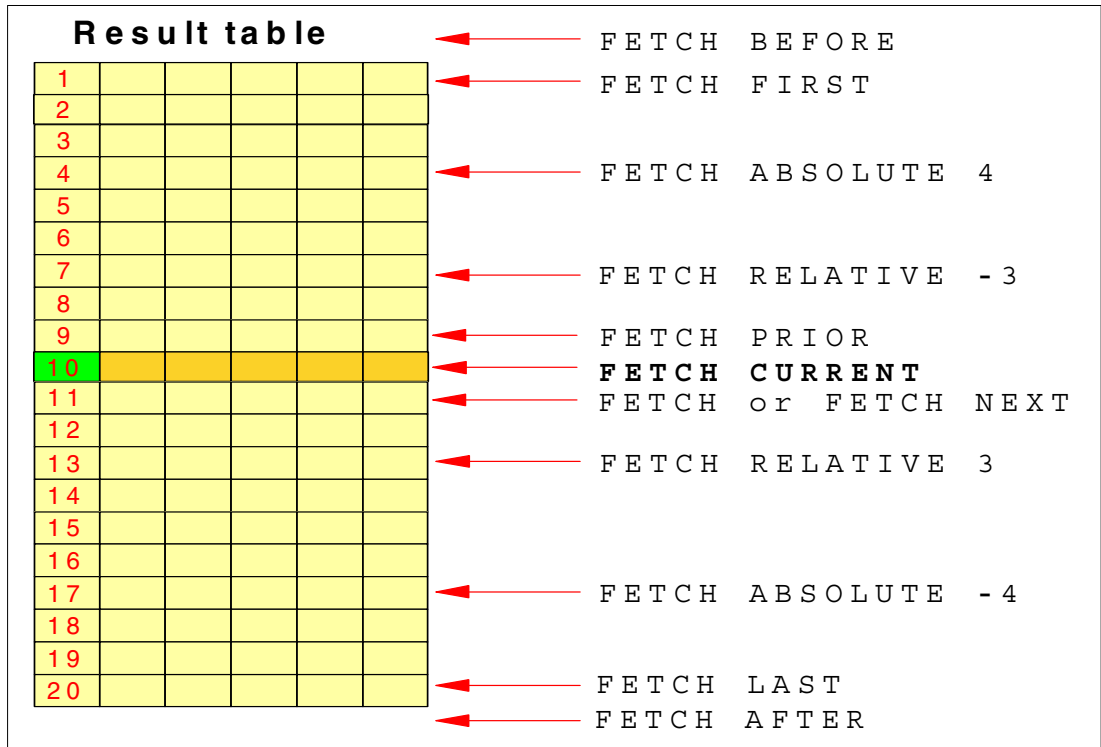


Figure 12-2 How to scroll within the result table

Refer to Figure 12-3 for a list of cursor positioning values and the possible SQLCODEs that may be returned.

	Current Position before first row	Current Position on first row	Current Position on last row	Current Position After Last row	Resulting Position on Delete Hole	Resulting Position on Update hole	Resulting Position on normal Row	Resulting Position Before First row	Resulting Position After Last Row
NEXT	OK	OK	+100	+100	+222	+222	IF OK	IF +100 FROM FIRST ROW	IF +100 FROM LAST ROW
PRIOR	+100	+100	OK	OK	+222	+222	IF OK	IF +100 FROM LAST ROW	IF +100 FROM FIRST ROW
FIRST	OK	OK	OK	OK	+222	+222	IF OK	N/A	N/A
LAST	OK	OK	OK	OK	N/A	N/A	IF OK	N/A	N/A
BEFORE, ABSOLUTE 0	OK	OK	OK	OK	N/A	N/A	N/A	IF OK	N/A
AFTER	OK	OK	OK	OK	+222	+222	N/A	N/A	IF OK
CURRENT RELATIVE 0	+231	OK	OK	+231	+222	+222	IF OK	N/A	N/A
ABSOLUTE +n	OK	OK	OK	OK	+222	+222	IF OK	N/A	IF +100 AND n OUT OF RANGE
ABSOLUTE -n	OK	OK	OK	OK	+222	+222	IF OK	IF +100 AND n OUT OF RANGE	N/A
RELATIVE +n	OK	OK	+100	+100	+222	+222	IF OK	N/A	IF +100 AND n OUT OF RANGE
RELATIVE -n	+100	+100	OK	OK	+222	+222	IF OK	IF +100 AND n OUT OF RANGE	N/A

Figure 12-3 SQLCODEs and cursor position

12.5.5 Using functions in a scrollable cursor

There are two types of built-in functions used in DB2; scalar functions like SUBSTR, or column functions like AVG. For more information on this topic, see “User-defined functions (UDF)” on page 57 and “Built-in functions” on page 71.

Column functions such as MAX and AVG causes a scrollable cursor into implicit read-only mode and are not valid for a SENSITIVE STATIC cursor.

The basic rule for using column functions in a scrollable cursor is that if the column function is part of the predicate, you can use it in an insensitive cursor, as the one shown in Example 12-6.

Example 12-6 Using functions in a scrollable cursor

```

DECLARE C1 INSENSITIVE SCROLL
CURSOR FOR
  SELECT ITEM_NUMBER, PRICE
  FROM SC246300.TBITEMS
  WHERE PRICE >
    (SELECT AVG(PRICE)
     FROM SC246300.TBITEMS )

```

Then the value is frozen at the OPEN CURSOR.

However, with a *scalar function*, DB2 can maintain the relationship between the temporary result set and the rows in the base table, and therefore allows these functions to be used in both INSENSITIVE and SENSITIVE cursors. If used in an INSENSITIVE cursor, the function is evaluated once at OPEN CURSOR time. For SENSITIVE cursors and where a FETCH SENSITIVE is used, this function is evaluated at FETCH time. For SENSITIVE cursors with an INSENSITIVE FETCH the function is evaluated at FETCH time only against the result set for the cursor, the function is not evaluated against the base table.

In Example 12-7, we can see an expression (COMM + BONUS) and a column function (AVG(SALARY)) being used in an insensitive scrollable cursor. Here, the column function and the expression are evaluated when the cursor is opened and the results are saved by DB2.

Example 12-7 Using functions in an insensitive scrollable cursor

```
EXEC SQL DECLARE C1 INSENSITIVE SCROLL
CURSOR FOR
  SELECT EMPNO
         ,FIRSTNME
         ,SALARY
         ,COMM
         ,BONUS
         ,COMM + BONUS AS ADDITIONAL_MONEY
  FROM SC246300.TBEMPLOYEE
  WHERE SALARY >
         (SELECT AVG(SALARY)
          FROM SC246300.TBEMPLOYEE)
```

Scalar functions, UDFs, and expressions are re-evaluated using the base table row when a FETCH SENSITIVE request is processed. A positioned update or delete compares the column value in the result table with the re-evaluated value for the base table row.

External UDFs are executed for each qualifying row when a scrollable cursor is opened. Therefore, if an external UDF sends an e-mail then an e-mail is sent for each qualifying row.

UDFs are not re-executed when an insensitive fetch is issued.

Resolving functions during scrolling

Lets take a look at a few examples of how functions are resolved when used in a scrollable cursor.

In Example 12-8, we show a non-deterministic column function. You cannot declare a sensitive cursor with an AVG function because the result of the column function may change if the column is updated.

Example 12-8 Aggregate function in a SENSITIVE cursor

```
DECLARE C1 SENSITIVE STATIC SCROLL CURSOR
WITH HOLD FOR
  SELECT NORDERKEY, AVG(TAX)
  FROM SC246300.TBLINEITEM
  GROUP BY NORDERKEY
```

The BIND returns
-243 SENSITIVE CURSOR C1 CANNOT BE DEFINED FOR THE SPECIFIED SELECT STATEMENT

In Example 12-9 we show how the same cursor defined in Example 12-8 is valid if it is INSENSITIVE. Since the AVG function is only processed at OPEN CURSOR time, the data does not change and thus the values for the AVG does not change.

Example 12-9 Aggregate function in an INSENSITIVE cursor

```
DECLARE C1 INSENSITIVE SCROLL CURSOR
  WITH HOLD FOR
  SELECT NORDERKEY, AVG(TAX)
  FROM SC246300.TBLINEITEM
  GROUP BY NORDERKEY
```

In Example 12-10 we show the scalar function SUBSTR. If the cursor is SENSITIVE, the function is evaluated at FETCH time. If the cursor is INSENSITIVE, the function is evaluated at OPEN CURSOR time.

Example 12-10 Scalar functions in a cursor

```
DECLARE C1 SENSITIVE STATIC SCROLL CURSOR
  WITH HOLD FOR
  SELECT CUSTKEY, LASTNAME, SUBSTR(COMMENT,1,20)
  FROM SC246300.TBCUSTOMER
```

In Example 12-11 shows that you can also use an expression in a scrollable cursor. In this example we use a sensitive scrollable cursor. Therefore, the expression will be re-evaluated against the base table at each FETCH operation to make sure the rows still qualifies.

Example 12-11 Expression in a sensitive scrollable cursor

```
DECLARE TELETEST SENSITIVE STATIC SCROLL
  CURSOR FOR
  SELECT EMPNO
  ,FIRSTNME
  ,SALARY
  ,COMM
  ,BONUS
  ,COMM + BONUS AS ADDITIONAL_MONEY
  FROM SC246300.TBEMPLOYEE
  WHERE COMM + BONUS > 200
  FOR UPDATE OF BONUS
```

12.6 Update and delete holes

Update and delete holes are only created for SENSITIVE STATIC scrollable cursors. An update hole occurs when the corresponding row of the underlying table has been updated such that the updated row no longer satisfies the search condition specified in the SELECT statement of the cursor. A delete hole occurs when the corresponding row of the underlying table has been deleted.

Note: An application program is not able to distinguish between a delete hole and an update hole, only that there is a hole.

12.6.1 Delete hole

Delete holes can be created in three ways:

- ▶ When a row has been deleted from the base table by another agent.
- ▶ When the cursor itself has deleted a row that was part of the result set returned at OPEN CURSOR time.
- ▶ When the current process deletes a row outside of the scrollable cursor, and the row is part of the cursor's result set.

An example of the occurrence of a delete hole is:

```
DECLARE C1 SENSITIVE STATIC SCROLL CURSOR
FOR SELECT ACCOUNT
      ,ACCOUNT_NAME
FROM TBACCOUNT
WHERE TYPE = 'P'
FOR UPDATE OF ACCOUNT_NAME;
```

The OPEN CURSOR is executed and the DB2 temporary table is built with two rows. See Example 12-12 for the results of the OPEN CURSOR.

Another user executes the statement:

```
DELETE FROM TBACCOUNT
WHERE TYPE = 'P'
AND ACCOUNT = 'MNP230' ;
COMMIT ;
```

The row is deleted from the base table.

The process executes its first FETCH:

```
FETCH SENSITIVE FROM C1 INTO :hv_account, hv_account_name ;
```

DB2 attempts to fetch the row from the base table but the row is not found, DB2 marks the row in the result table as a delete hole.

DB2 returns the SQLCODE +222 to highlight the fact that the current cursor position is over a hole.

```
+222: HOLE DETECTED USING cursor-name
```

At this stage, the host variables are empty; however, it is important for your application program to recognize the hole, as DB2 does not reset the host variables if a hole is encountered.

If the FETCH is executed again, the cursor is positioned on the next row, which in the example is for account 'ULP231'. The host variables now contain 'ULP231' and 'MS S FLYNN'.

It is important to note that if an INSENSITIVE fetch is used, then only update and delete holes created under the current open cursor are recognized. Updates and deletes made by other processes or outside the cursor are not recognized by the INSENSITIVE fetch.

If the above SENSITIVE fetch was replaced with an INSENSITIVE fetch, the fetch would return a zero SQLCODE, since the delete to the base row was made by another process. The column values would be set to those at the time of the OPEN CURSOR statement execution.

Example 12-12 Delete holes

Base Table

ACCOUNT	ACCOUNT_NAME	TYPE
ABC010	BIG PETROLEUM	C
BWH450	RUTH & DAUGHTERS	C
ZXY930	MIGHTY DUCKS PLC	C
MNP230	BASEL FERRARI	P
BMP291	MR R GARCIA	C
XPM673	SCREAM SAVER LTD	C
ULP231	MS S FLYNN	P
XPM961	MR CJ MUNSON	C

Result table

RID	ACCOUNT	ACCOUNT_NAME
A04	MNP230	MR BASEL FERRARI
A07	ULP231	MS S FLYNN

Base Table after DELETE

ACCOUNT	ACCOUNT_NAME	TYPE
ABC010	BIG PETROLEUM	C
BWH450	RUTH & DAUGHTERS	C
ZXY930	MIGHTY DUCKS PLC	C
		<-----deleted row from base table
BMP291	MR R GARCIA	C
XPM673	SCREAM SAVER LTD	C
ULP231	MS S FLYNN	P
XPM961	MR CJ MUNSON	C

Result table after FETCH

RID	ACCOUNT	ACCOUNT_NAME
A04		
A07	ULP231	MS S FLYNN

12.6.2 Update hole

An update hole can be created when a row that was returned in the initial result set is updated in such a way as to make it no longer qualify by the WHERE conditions of the SELECT statement of a SENSITIVE STATIC cursor. An example of the occurrence of an update hole is:

```

DECLARE C1 SENSITIVE STATIC SCROLL CURSOR
FOR SELECT ACCOUNT
      ,ACCOUNT_NAME
FROM TBACCOUNT
WHERE TYPE = 'P'
FOR UPDATE OF ACCOUNT_NAME;

```

The OPEN CURSOR is executed and the DB2 temporary table is built with two rows. See Example 12-13 for the results of the OPEN CURSOR.

Another user executes the statement:

```

UPDATE TBACCOUNT

```

```

SET TYPE = 'P
WHERE ACCOUNT = 'MNP230' ;
COMMIT ;

```

Here, it can be seen that the row for account 'MNP230' no longer qualifies the requirements of the WHERE clause of the DECLARE CURSOR statement.

The process executes its first FETCH:

```

FETCH SENSITIVE FROM C1 INTO :hv_account, hv_account_name ;

```

DB2 verifies that the row is valid by executing a SELECT with the WHERE values used in the initial open against the base table. If the row now falls outside the SELECT, DB2 returns the SQLCODE +222 to highlight the fact that the current cursor position is over an update hole.

```
+222: HOLE DETECTED USING cursor-name
```

At this stage, the host variables are empty; however, it is important for your application program to recognize the hole, as DB2 does not reset the host variables if a hole is encountered.

If the FETCH is executed again, the cursor is positioned on the next row, which in the example is for account 'ULP231'. The host variables now contain 'ULP231' and 'MS S FLYNN'.

It is important to note that if an INSENSITIVE fetch is used, then only update and delete holes created under the current open cursor are recognized. Updates and deletes made by other processes are not recognized by the INSENSITIVE fetch.

If the above SENSITIVE fetch was replaced with an INSENSITIVE fetch, the fetch would return a zero SQLCODE, as the update to the base row was made by another process. The column values would be set to those at the time of the OPEN CURSOR statement execution.

Example 12-13 Update holes

```

Base Table
ACCOUNT  ACCOUNT_NAME  TYPE
ABC010  BIG PETROLEUM  C
BWH450  RUTH & DAUGHTERS  C
ZXY930  MIGHTY DUCKS PLC  C
MNP230  BASEL FERRARI    P
BMP291  MR R GARCIA    C
XPM673  SCREAM SAVER LTD  C
ULP231  MS S FLYNN     P
XPM961  MR CJ MUNSON   C

```

```

Result table
RID  ACCOUNT  ACCOUNT_NAME
A04  MNP230  BASEL FERRARI
A07  ULP231  MS S FLYNN

```

```

Base Table after UPDATE
ACCOUNT  ACCOUNT_NAME  TYPE
ABC010  BIG PETROLEUM  C
BWH450  RUTH & DAUGHTERS  C
ZXY930  MIGHTY DUCKS PLC  C
MNP230  BASEL FERRARI    C
BMP291  MR R GARCIA    C
XPM673  SCREAM SAVER LTD  C
ULP231  MS S FLYNN     P
XPM961  MR CJ MUNSON   C

```

```
Result table after FETCH
RID  ACCOUNT  ACCOUNT_NAME
A04
A07  ULP231    MS S FLYNN
```

12.7 Maintaining updates

Prior to DB2 V7, programs had to provide a means to ensure that the values in the currently fetched row were still current when performing an update or delete of the row. Normally, this involved executing a SELECT (using the primary key of the table to be updated) and checking each column to ensure that the values had not changed since the last time the row was retrieved. Another way of performing the validation is by maintaining a “last updated” timestamp on each row and verifying that the timestamp did not change since the row was last retrieved. Yet another way was to use the FOR UPDATE OF in a cursor to maintain the lock on the row (this method could negatively impact concurrency).

DB2 now maintains a relationship between the rows returned by a SENSITIVE STATIC scrolling cursor and those in the base table. If an attempt is made to UPDATE or DELETE the currently fetched row, DB2 goes to the base table, using the RID, and verifies that the columns match by value. If columns are found to have been updated, then DB2 returns the SQL code:

```
-224: THE RESULT TABLE DOES NOT AGREE WITH THE BASE TABLE USING cursor-name.
```

When you receive this return code, you can choose to fetch the new data again by using the FETCH CURRENT to retrieve the new values. The program can then choose to reapply the changes or not.

Important: DB2 only validates the columns listed in the select clause of the cursor against the base table. Other changed columns do not cause the SQLCODE -224 to be issued.

Note: A scrollable cursor never sees rows that have been inserted to the base table nor rows that are updated and now (after open cursor time) fit the selection criteria of the DECLARE CURSOR statement.

How DB2 validates a positioned UPDATE and DELETE

With SENSITIVE STATIC scrollable cursors, DB2 always validates a positioned update before allowing it to proceed. The technique used by DB2 is referred to as *optimistic locking concur by value*.

Let's assume isolation level CS or UR is in effect for the SENSITIVE STATIC scrollable cursor. The application issues a FETCH SENSITIVE request which positions the cursor on a row. Once the FETCH has completed, DB2 releases the lock (if held) on the base table row. DB2 does this to improve concurrency. The application now decides to update the current row. However, the base table row could have been updated or deleted by another application between the time it was fetched and the time the positioned update is requested. Therefore, DB2 must validate the base table row before allowing the update to proceed.

Now let's assume isolation level RR or RS is in effect for the sensitive scrollable cursor. The application issues a FETCH SENSITIVE request which positions the cursor on a row. DB2 does not release the lock on the base table row. The application now decides to update the current row. DB2 must still validate the positioned update. This is because the same application might have updated or deleted the base table row (by issuing a searched update or delete) between the time it was fetched and the time the positioned update is requested.

Validation of a positioned UPDATE:

The Optimistic Locking Concur By Value technique is used by DB2 to validate a positioned UPDATE. Lets take a look at how this process works. Suppose that we have executed the following statements:

```
DECLARE C1 SENSITIVE STATIC SCROLL CURSOR
  FOR   SELECT ....
        FROM BASE_TABLE
        WHERE ....
        FOR UPDATE OF ....

OPEN CURSOR C1

FETCH SENSITIVE

UPDATE BASE_TABLE
  SET ... = ...,
      ... = ...
  WHERE CURRENT OF C1
```

The flow chart in Figure 12-4 shows the sequence of events that occurs.

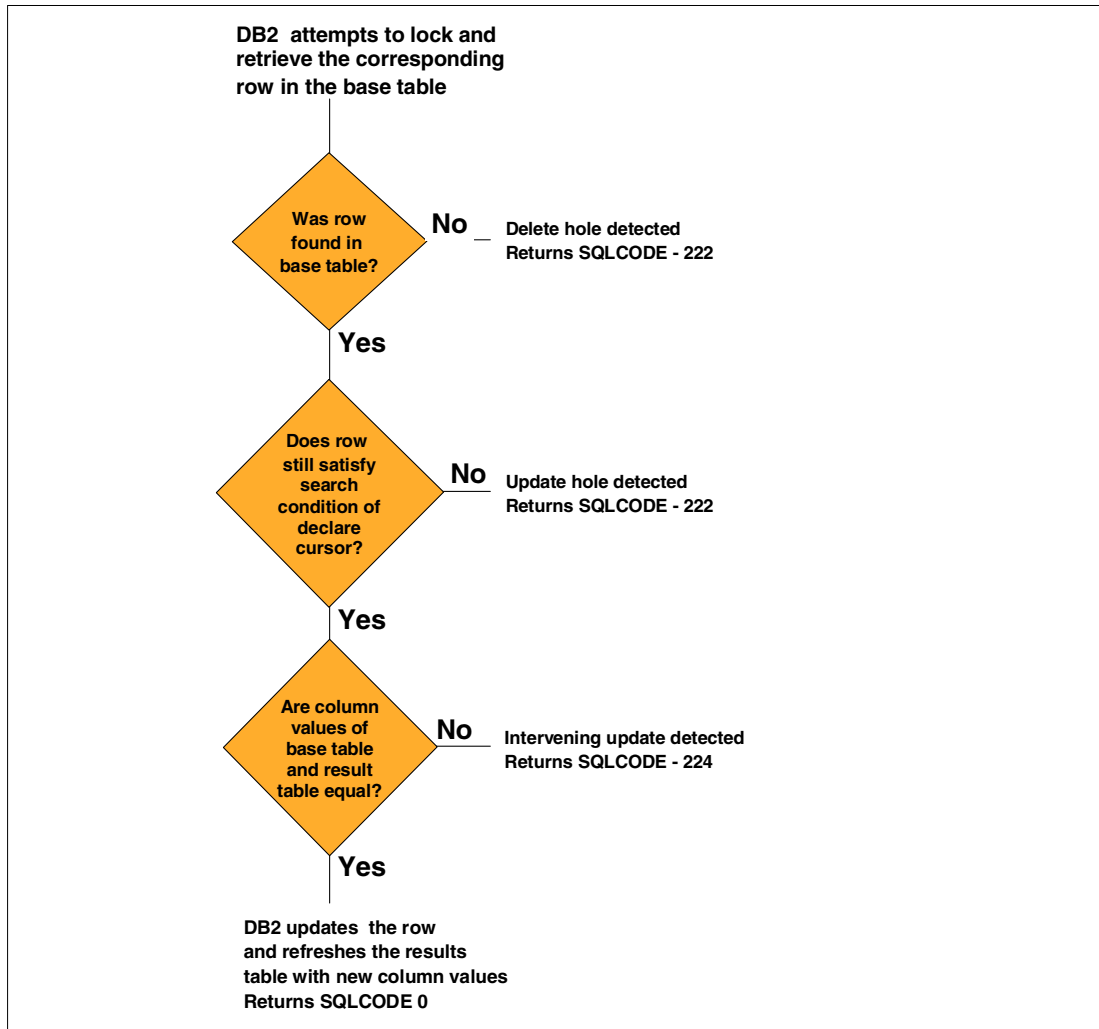


Figure 12-4 How DB2 validates a positioned UPDATE

Validation of a positioned DELETE:

The *optimistic locking concur by value* technique is also used by DB2 to validate a positioned delete. Lets take a look at how this process works. Suppose that we have executed the following statements:

```

DECLARE C1 SENSITIVE STATIC SCROLL CURSOR
FOR SELECT ....
FROM BASE_TABLE
WHERE ....

OPEN CURSOR C1

FETCH SENSITIVE

DELETE FROM BASE_TABLE
WHERE CURRENT OF C1
  
```

The flow chart in Figure 12-5 shows the sequence of events that occurs.

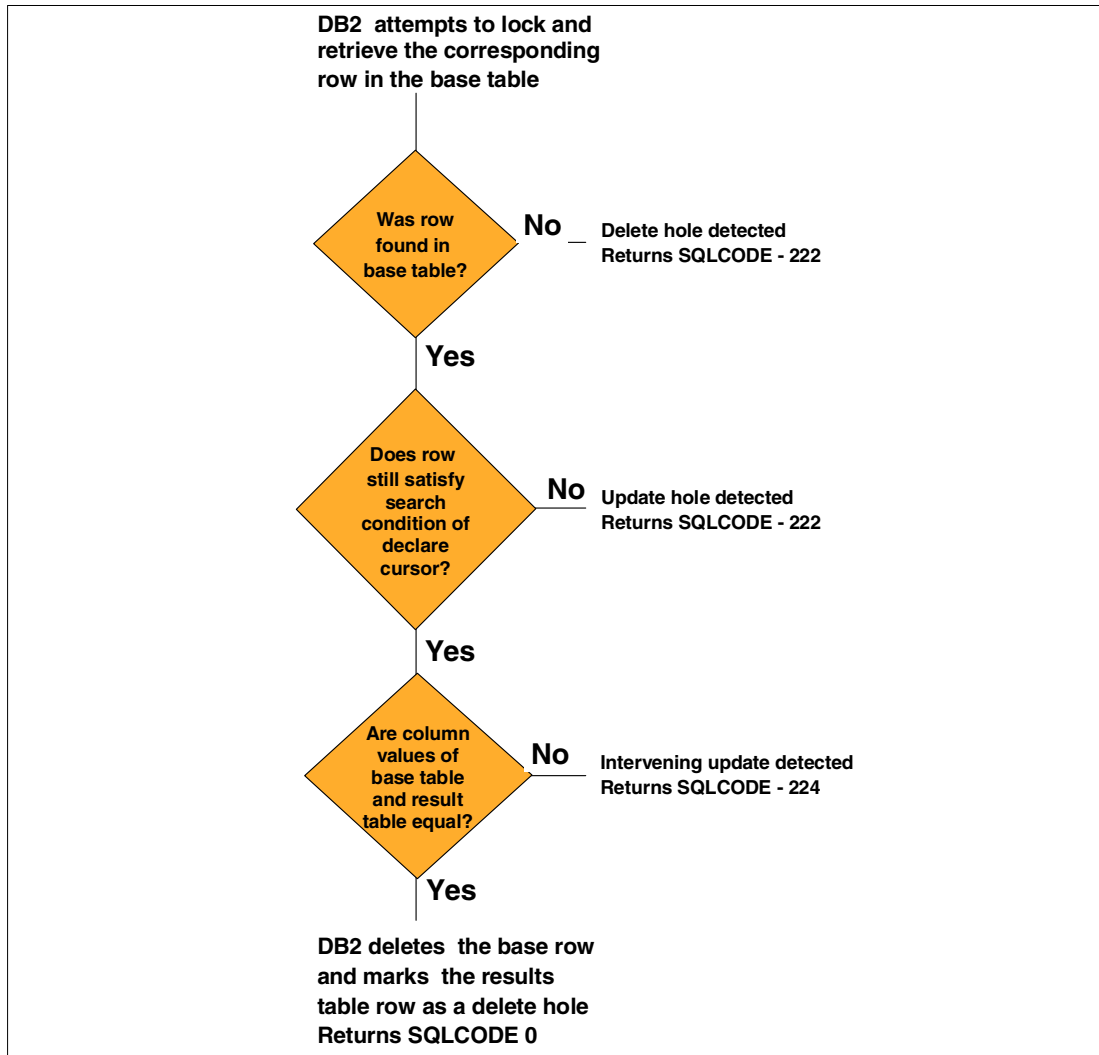


Figure 12-5 How DB2 validates a positioned DELETE

12.8 Locking and scrollable cursors

Scrollable cursors behave just like non-scrollable cursors in terms of locking. The existing locking mechanism apply, as usual, as defined by lock size and isolation level. The RR, RS, CS, UR parameters have the same implications as always.

For example, during open of a cursor that requires a temporary result table:

- ▶ A scrollable cursor bound with isolation level RR (Repeatable Read) keeps the lock on every page or row it reads whether or not the row is qualified for the select.
- ▶ A scrollable cursor bound with isolation level RS (Read Stability) keeps the lock on every page or row that is qualified based on the stage 1 predicates.
- ▶ A scrollable cursor bound with isolation level CS (Cursor Stability):
 - When reading rows during execution of the OPEN CURSOR statement with the CURRENTDATA setting of YES, takes a lock on the last page or row read.
 - If CURRENTDATA is set to NO, does not take locks except where the cursor has been declared with the FOR UPDATE OF clause or lock avoidance was not able to avoid

taking the lock. When FOR UPDATE OF is specified, a lock is taken for the last page or row read.

- Keeps locks on the last page or row read if the cursor was declared FOR UPDATE OF.
- Releases all locks on completion of the OPEN CURSOR.
- ▶ A cursor that has been bound with uncommitted read (UR) does not take any page or row locks, and does not check whether a row has been committed when selecting it for inclusion in the temporary result set.

Application programs can leverage the use of SENSITIVE STATIC scrollable cursors in combination with the Isolation level CS and the SENSITIVE option of FETCH to minimize concurrency problems and assure currency of data when required. The STATIC cursor does give the application a constant result table to scroll on, thus perhaps eliminating the need for isolation level RR and RS. The SENSITIVE option of FETCH statement provides the application a means of re-fetching any preselected row requesting the most current data when desired. For example, when the application is ready to update a row.

Duration of locks

Locks acquired for positioned updates, positioned deletes, or to provide isolation level RR or isolation level RS are held until commit. If the cursor is defined WITH HOLD, then the locks are held until the first commit after the close of the cursor.

12.9 Stored procedures and scrollable cursors

A scrollable cursor can be opened by a stored procedure and the result table can be returned as a stored procedure result set. The following rules must be followed:

- ▶ A stored procedure can open multiple scrollable cursors but it must leave each cursor positioned before the first row in the result table before returning to the calling program. This is different from non-scrollable cursors. If you don't stick to this rule you will receive a SQLCODE -20123 (call to stored procedure *mrsbms* failed because the result set returned for cursor *teste-cursor* is scrollable, but the cursor is not positioned before the first row).
- ▶ On return, the calling program then allocates the cursor and can then scroll through the result table.
- ▶ All stored procedure defined cursors are read-only from the client (caller's) side, as is the case for non-scrollable cursors. However, these cursors can still make use of both INSENSITIVE and SENSITIVE style cursors. (You can do a positioned update through the scrollable cursor in the stored procedure itself, as long as you reposition before the result set before returning to the calling program.)

Figure 12-6 shows an example of how to use a scrollable cursor in a stored procedure using result sets. A full listing is available in the additional material. See Appendix C, "Additional material" on page 251 for more details.

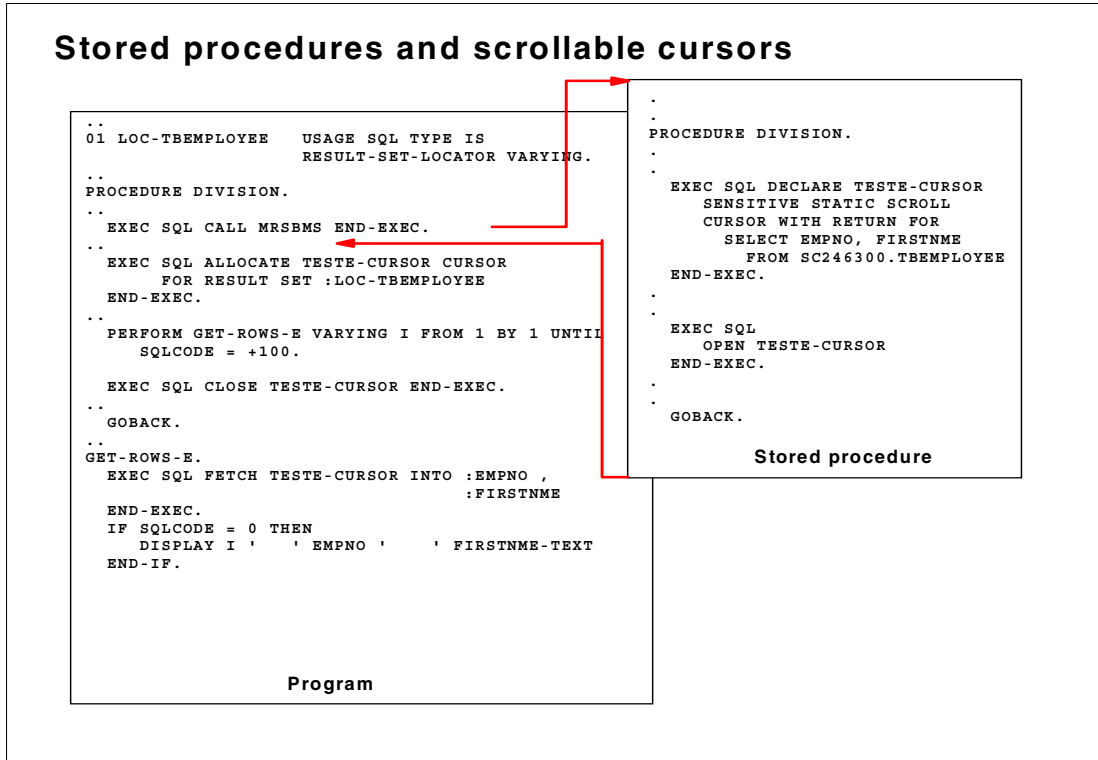


Figure 12-6 Stored procedures and scrollable cursors

Note: If a stored procedure issues FETCH statements for a scrollable cursor, then before ending, it must issue a FETCH BEFORE statement to position the cursor before the first row in the result table before returning to the calling program.

12.10 Scrollable cursors recommendations

If you don't need the cursor to be updatable, use a stored procedure to open the cursor in a distributed environment.

Choose the appropriate isolation level, CS with CURRENTDATA(NO) is usually sufficient. Remember that isolation level RR and RS greatly reduce concurrency.

Provide sufficient TEMP database table space storage to hold the result tables for all scrollable cursors that might be open concurrently and all user-declared temporary tables that might be in use concurrently.

Commit as often as is practical and specify WITH HOLD if you want the cursor to remain if the application issues a commit. At commit, a scrollable cursor is closed and the result table deleted if WITH HOLD is not specified on the DECLARE CURSOR statement.



More SQL enhancements

In this chapter, we discuss a set of various smaller SQL enhancements like:

- ▶ ON clause extensions
- ▶ Row expressions
- ▶ ORDER BY improvements
- ▶ INSERT
- ▶ UPDATE
- ▶ SELECT INTO enhancements
- ▶ FETCH FIRST n ROWS ONLY
- ▶ Changes to the usage of host variables
- ▶ Full predicate support by IN predicates
- ▶ Partitioning key update support

13.1 The ON clause extensions

The ON clause syntax has been extended for left, right and inner joins.

You can now include boolean predicates (ANDed, ORed, and NOT) in the ON clause of a join. These predicates, coded in the ON clause, are applied to the join and are called *during join predicates*. The join result is built while all the predicates in the ON clause are being applied.

13.1.1 Classifying predicates

There are different types of predicates. To better understand where the ON clause extensions fit in, we give a short overview of the different predicate types.

Table access predicates	These predicates are also called before join predicates. These predicates are applied to the table before the join, at the time rows are retrieved from the table.
During join predicates	This type of predicate was introduced in DB2 Version 6. They are coded in the ON clause and processed during the join and influence the join result. This is what we discuss in the section, During join predicates.
After join step predicates	This predicate type is also new in V6. They are coded in the WHERE clause, but are processed before processing the next join step for performance reasons.
Totally after join predicates	After all join processing is done these predicates are evaluated.

For more information on predicate classification, see *DB2 UDB for OS/390 Version 6 Performance Topics*, SG24-5351.

13.1.2 During join predicates

The ON clause syntax has been extended (in DB2 Version 6) to allow more expressions for left, right and inner joins. Now you can include any search condition just like a WHERE clause, except coding a subquery. These predicates are called *during join predicates*.

Note: The ON clause extensions are not implemented for full outer joins. You receive an SQLCODE -338.

The conditions of the ON clause are join conditions; they specify conditions for the rows to be joined. The cartesian product is performed with the rows that satisfy the enhanced ON condition predicate, which is not necessarily only about equality of different columns, but also connected with predicates on single columns and expressions. You can now include any search condition just like a WHERE clause, with one exception. Coding a subquery in the ON clause is not allowed.

To understand this concept, let's look at some examples.

Example 13-1 shows the tables and rows that is used in the examples in this section.

Example 13-1 Sample tables and rows

SC246300.TBEMPLOYEE		SC246300.TBDEPARTMENT		
FIRSTNAME	WORKDEPT	DEPTNO	DEPTNAME	...

MIRIAM	A01	A01	SALES
JUKKA	A02	A02	MARKETING
GLADYS	--	B01	DB2
ABI	B01	C01	MVS
TONI	--		
EVA	A01		

Example 13-2 shows an inner join with a predicate *E.WORKDEPT='A01'* in the ON clause. Before this enhancement these predicate could not be coded in the ON clause. Only join predicates were allowed.

In this particular case, because it is an INNER JOIN and the additional predicate in the ON clause was ANDed, the query behaves the same way as if the '*AND E.WORKDEPT = 'A01'*' predicate was coded in the WHERE clause. However, this is not always the case.

Example 13-2 Inner join and ON clause with AND

```

SELECT E.FIRSTNME,E.WORKDEPT, D.DEPTNO,D.DEPTNAME
FROM SC246300.TBEMPLOYEE E
INNER JOIN SC246300.TBDEPARTMENT D
ON E.WORKDEPT = D.DEPTNO
AND E.WORKDEPT = 'A01' #
-----+-----+-----+-----+-----+-----+-----+
FIRSTNME      WORKDEPT  DEPTNO  DEPTNAME
-----+-----+-----+-----+-----+-----+
MIRIAM        A01        A01     SALES
EVA           A01        A01     SALES

```

Example 13-3 show a left outer join with AND in the ON clause on the DEPTNO column. Now you notice the difference between ANDing the predicate *E.WORKDEPT = 'A01'* in the ON clause, and coding the sample predicate in the WHERE clause, as shown in Example 13-4.

Example 13-3 LEFT OUTER JOIN with ANDed predicate on WORKDEPT field in the ON clause

```

SELECT E.FIRSTNME,E.WORKDEPT, D.DEPTNO,D.DEPTNAME
FROM SC246300.TBEMPLOYEE E
LEFT OUTER JOIN SC246300.TBDEPARTMENT D
ON E.WORKDEPT = D.DEPTNO
AND E.WORKDEPT = 'A01' #
-----+-----+-----+-----+-----+-----+
FIRSTNME      WORKDEPT  DEPTNO  DEPTNAME
-----+-----+-----+-----+-----+-----+
EVA           A01        A01     SALES
MIRIAM        A01        A01     SALES
JUKKA         A02        -----
ABI           B01        -----
GLADYS        -----
TONI          -----

```

In Example 13-3, in order for the join condition to be satisfied, both conditions have to be true. If this is not the case, the columns from the right hand table (TBDEPARTMENT) are set to null.

Example 13-4 LEFT OUTER JOIN with ON clause and WHERE clause

```

SELECT E.FIRSTNME,E.WORKDEPT, D.DEPTNO,D.DEPTNAME
FROM SC246300.TBEMPLOYEE E
LEFT OUTER JOIN SC246300.TBDEPARTMENT D
ON E.WORKDEPT = D.DEPTNO
WHERE E.WORKDEPT = 'A01' #

```

FIRSTNME	WORKDEPT	DEPTNO	DEPTNAME
EVA	A01	A01	SALES
MIRIAM	A01	A01	SALES

This is totally different from the case where you code the *E.WORKDEPT = 'A01'* condition in the WHERE clause, as in Example 13-4. The WHERE clause is not evaluated at the time the join is performed. According to the semantics of the SQL language It must be evaluated after the result from the join is built (to eliminate all rows where *E.WORKDEPT is not equal to 'A01'*. (Actually this is not entirely true. Version 6 introduced a lot of performance enhancements related to outer join and in this case, the predicate that is coded in the WHERE clause can and is evaluated when the data is retrieved from the TBEMPLOYEE table. The system only moves the evaluation of the predicate to an earlier stage during the processing, *if it is sure the result is not influenced by this transformation*. In our example this is the case because the WHERE predicate is not on the 'null supplying' table. For more information on the outer join performance enhancements, see *DB2 UDB for OS/390 Version 6 Performance Topics*, SG24-5351.)

Example 13-5 shows another flavour of the ON clause extensions using an OR condition in the ON clause. MIRIAM and EVA (rows satisfying *WORKDEPT = 'A01'*) are joined with every row of the TBDEPARTMENT table. The predicates on the ON clause determine whether we have a match for the join. So in this example, If either *E.WORKDEPT = D.DEPTNO* or *E.WORKDEPT = 'A01'* is true, the rows of both tables are matched up.

Example 13-5 Inner join and ON clause with OR in the WORKDEPT column

```

SELECT E.FIRSTNME,E.WORKDEPT, D.DEPTNO,D.DEPTNAME
FROM SC246300.TBEMPLOYEE E
INNER JOIN SC246300.TBDEPARTMENT D
ON E.WORKDEPT = D.DEPTNO
OR E.WORKDEPT = 'A01' #

```

FIRSTNME	WORKDEPT	DEPTNO	DEPTNAME
MIRIAM	A01	B01	DB2
MIRIAM	A01	A01	SALES
MIRIAM	A01	C01	MVS
MIRIAM	A01	A02	MARKETING
JUKKA	A02	A02	MARKETING
EVA	A01	B01	DB2
EVA	A01	A01	SALES
EVA	A01	C01	MVS
EVA	A01	A02	MARKETING
ABI	B01	B01	DB2

13.2 Row expressions

In this section, we discuss row expressions. Row expressions can simplify WHERE clauses. They enable us to compare multiple columns simultaneously using equal and not equal expressions and compare multiple columns against the results of a subquery using IN and NOT IN operators.

This is primarily a usability enhancement, but row expressions can impact performance. Rewriting a query to use row expressions can change the access path used.

13.2.1 What is a row expression?

Prior to DB2 V7, the SQL syntax only allowed a single column on one side to be used in a comparison operation with equal and not equal operators. DB2 V7 extends the syntax to allow the use of multiple expressions to be specified in the IN and NOT IN subquery predicates, some quantified predicates (=SOME, =ANY, <>ALL), as well as with equal and not equal operators (=, <>).

The performance impact may be significant if the access path changes when you rewrite the query to use row expressions. The size of the result set of the outer table may not have much impact, but the size of the result set of the inner table may have impact performance. If a large number of rows qualify from the inner table, the increase of response time may be significant. When the access path does not change with and without row expression, then there is no performance impact either.

13.2.2 Types of row expressions

There are three basic types of row value expressions:

- ▶ Equal and not equal operator (=, <>)
- ▶ Quantified predicates (= ANY, = SOME, <> ALL)
- ▶ IN and NOT IN with a subquery

Equal and unequal operators '=' and '<>'

Multiple expression can be used to compare against multiple expressions in a single predicate. These multiple expressions are called *row-value-expressions*. If a subselect is used, it must return the same number of columns in the result set as the *row-value-expression* it is compared to. Matching is done by comparing expressions on one side of the operator with the expressions in same position on the other side of the operator. If the operator is =, the result is true if all the pairs of expressions evaluate to true, and false otherwise. If the operator is <>, the result is true if any pair of expressions evaluate to true, and false otherwise.

For example, prior to DB2 V7, you would code COL1 = :HV1 AND COL2 = :HV2; now you can code (COL1, COL2) = (:HV1, :HV2). Both conditions have to be fulfilled for the statement to be true. You may now code (COL1, COL2) <> (:HV1, :HV2) instead of COL1 <> :HV1 OR COL2 <> :HV2. In this case, at least one condition has to evaluate true for the predicate to be true. In these simple cases the access path is the same with and without row expression and has no impact on performance.

Example 13-6 shows how to use a *row-value-expression* containing two column and compare (=) it to a *row-value-expression* that contains a constant value and a host variable. It finds all employees in city "1" and country "34" (in this case country is a host variable *:nationkey*).

Example 13-6 Row expressions with equal operation

```
SELECT FIRSTNME
```

```

        ,JOB
        ,WORKDEPT
        ,SEX
        ,EDLEVEL
FROM SC246300.TBEMPLOYEE
WHERE (CITYKEY, NATIONKEY) = (1, :nationkey ) ;

```

```

-----+-----+-----+-----+-----+--
FIRSTNME      JOB      WORKDEPT  SEX      EDLEVEL
-----+-----+-----+-----+-----+--
MIRIAM        DBA      A01      F        4

```

Note: Row expressions with equal and not equal operators are not allowed to be compared against fullselect.

Quantified predicates (=ANY, =SOME, <>ALL)

The =ANY and =SOME operators are synonyms of each other. All value expressions on the left hand side of the comparison must match at least one row value expression on the right side of the comparison in order for the comparison to be evaluated to true.

Example 13-7 uses a row expression with an = ANY operator against a subquery. This example lists the employees that live in the same city and country as any of our customers.

Example 13-7 Row expressions with = ANY operation

```

SELECT FIRSTNME
        ,JOB
        ,WORKDEPT
        ,SEX
        ,EDLEVEL
FROM SC246300.TBEMPLOYEE
WHERE (CITYKEY,NATIONKEY) = ANY
      (SELECT CITYKEY,NATIONKEY
       FROM SC246300.TBCUSTOMER ) ;

```

```

-----+-----+-----+-----+-----+--
FIRSTNME      JOB      WORKDEPT  SEX      EDLEVEL
-----+-----+-----+-----+-----+--
EVA           SYSADM  A01      F        0

```

In order for the <>ALL comparison to evaluate to true, there must not be any row on the right hand side of the comparison that matches the values specified on the left hand side.

Example 13-8 uses a row expression with a <> ALL operator against a subquery. This example lists the employees that do not live in the same city and country as any of our customers.

Example 13-8 Row expression with <> ALL operator

```

SELECT FIRSTNME
        ,JOB
        ,WORKDEPT
        ,SEX
        ,EDLEVEL
FROM SC246300.TBEMPLOYEE

```



```

WHERE (CITYKEY,NATIONKEY) <> ALL
      (SELECT CITYKEY,NATIONKEY
       FROM SC246300.TBCUSTOMER ) ;

```

FIRSTNME	JOB	WORKDEPT	SEX	EDLEVEL
MIRIAM	DBA	A01	F	4
JUKKA	SALESMAN	A02	M	7
TONI		-----	M	0
GLADYS		-----	F	0
ABI	TEACHER	B01	F	9

Note: The use of row value expressions on the left-hand side of a predicate with = SOME or = ANY operators is the same as using the IN keyword. The <> ALL operator is the same as using the NOT IN keywords.

IN and NOT IN with a subquery

Example 13-9 is equivalent to Example 13-7 and evaluates in the same way.

Example 13-9 IN row expression

```

SELECT FIRSTNME
       ,JOB
       ,WORKDEPT
       ,SEX
       ,EDLEVEL
FROM SC246300.TBEMPLOYEE
WHERE (CITYKEY,NATIONKEY) IN
      (SELECT CITYKEY,NATIONKEY
       FROM SC246300.TBCUSTOMER ) ;

```

FIRSTNME	JOB	WORKDEPT	SEX	EDLEVEL
EVA	SYSADM	A01	F	0

Example 13-10 is equivalent to Example 13-8 and evaluates in the same way.

Example 13-10 NOT IN row expression

```

SELECT FIRSTNME
       ,JOB
       ,WORKDEPT
       ,SEX
       ,EDLEVEL
FROM SC246300.TBEMPLOYEE
WHERE (CITYKEY,NATIONKEY) NOT IN
      (SELECT CITYKEY,NATIONKEY
       FROM SC246300.TBCUSTOMER ) ;

```

FIRSTNME	JOB	WORKDEPT	SEX	EDLEVEL
MIRIAM	DBA	A01	F	4
JUKKA	SALESMAN	A02	M	7

TONI		-----	M	0
GLADYS		-----	F	0
ABI	TEACHER	B01	F	9

13.2.3 Row expression restrictions

Row expressions are not supported for 'IN lists'. The following syntax is not valid for finding all employees in city 1 or 2 in country 34.

Example 13-11 Row expression restrictions

```
SELECT FIRSTNME, JOB, WORKDEPT, SEX, EDLEVEL
FROM SC246300.TBEMPLOYEE
WHERE (CITYKEY, NATIONKEY) IN (( 1, 34 ), ( 2, 34 ) );

-- SQLCODE = -104
```

If the number of expressions and the number of columns returned do not match, then an SQL error is returned:

```
SQLCODE -216 THE NUMBER OF ELEMENTS ON EACH SIDE OF A PREDICATE OPERATOR DOES
      NOT MATCH. PREDICATE OPERATOR IS IN.
SQLSTATE: 428C4
```

13.3 ORDER BY

The order of the selected rows depends on the sort keys that you identify in the ORDER BY clause. A sort key can be a column name, an integer that represents the number of a column in the result table, or an expression. DB2 sorts the rows by the first sort key, followed by the second sort key, and so on.

You can list the rows in ascending or descending order. Null values appear last in an ascending sort and first in a descending sort. The ordering can be different for each column in the ORDER BY clause.

13.3.1 ORDER BY columns no longer have to be in select list (V5)

Before DB2 V5 the columns in the ORDER BY clause had to be in the SELECT list. Now we can specify a column in the ORDER BY clause that is NOT in the SELECT list. DB2 sorts strings in the collating sequence associated with the encoding scheme of the table, before the projection phase (selecting the columns to return) of the select.

Example 13-12 shows a SELECT that lists all employees in department 'A00' sorted by birth date in ascending order. Notice that the BIRTHDATE column is not in the select list.

Example 13-12 ORDER BY column not in the select list

```
SELECT EMPNO
      , LASTNAME
      , HIREDATE
FROM SC246300.TBEMPLOYEE
WHERE WORKDEPT = 'A00'
ORDER BY BIRTHDATE ASC;
```

Prior to this enhancement the query above would have returned an SQLCODE -208.

The following restrictions apply:

- ▶ There is no UNION or UNION ALL (SQLCODE -208).
- ▶ There is no GROUP BY clause (SQLCODE -122).
- ▶ There is no DISTINCT clause in the select list (new SQLCODE -214).

Tip: When calculating the amount of sort space required for the query, all columns, including the ones being sorted on should be included in the sort data length as well as in the sort key length.

13.3.2 ORDER BY expression in SELECT (V7)

In DB2 V7 we can include expressions in the ORDER BY clause. Prior to DB2 V7, in order to be able to sort by an expression, you must explicitly include the expression in the SELECT list and then reference the number of the column (the expression) of the result table in the ORDER BY clause.

In Example 13-13, we have a query to retrieve the employee number, total compensation (salary plus commission), salary, commission, for employees with a total compensation greater than \$40000. Prior to V7 we would have to code ORDER BY 2 to produce the same result.

Example 13-13 ORDER BY expression in SELECT

```
-- Prior to DB2 V7
SELECT EMPNO
       ,SALARY+COMM AS "TOTAL COMP"   -- Note: this is the second column in the list
       ,SALARY
       ,COMM
FROM SC246300.TBEMPLOYEE
WHERE SALARY+COMM > 40000
ORDER BY 2;

-- DB2 V7
SELECT EMPNO
       ,SALARY+COMM AS "TOTAL COMP"
       ,SALARY
       ,COMM
FROM SC246300.TBEMPLOYEE
WHERE SALARY+COMM > 40000
ORDER BY SALARY+COMM ;

-- Or even better
SELECT EMPNO
       ,SALARY+COMM AS "TOTAL COMP"
       ,SALARY
       ,COMM
FROM SC246300.TBEMPLOYEE
WHERE SALARY+COMM > 1000
ORDER BY "TOTAL COMP" ;
```

Some vendor applications generate SQL statements which contain ORDER BY expressions. In such cases, it is often not feasible to re-write the generated SQL.

13.3.3 ORDER BY sort avoidance (V7)

DB2 can now logically remove columns from an ORDER BY clause and an index key if their values are constant. This enables DB2 to recognize more situations in which a sort operation can be avoided.

This enhancement is based on the assertion that any constant value included as part of an ordering key can be logically removed from the ordering key without changing the order. An ordering key can be the columns listed in an ORDER BY clause or the columns of an index key.

In Figure 13-1, the predicates for columns C2, C4 and C5 specify a constant value. Therefore, DB2 can logically remove these columns from the ORDER BY clause without changing the order requested by the user. DB2 can also logically remove these columns from the index key without changing the ordering capability of the index.

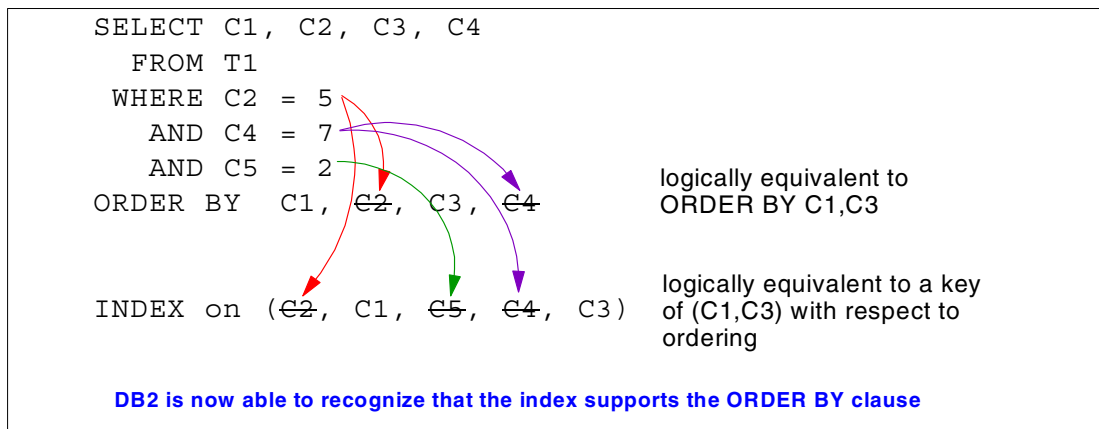


Figure 13-1 Improved sort avoidance for ORDER BY clause

In Example 13-14, we show how this works. The ORDER BY in Figure 13-1 is equivalent to an ORDER BY C1, C3. Note that index on C2, C1, C5, C4, C3 can be used and is equivalent to an index on C1, C3 and thus avoiding a sort or the need for an additional index.

Example 13-14 Data showing improved sort avoidance for the ORDER BY clause

	C2	C1	C5	C4	C3	
	1	8	7	7	3	
	1	9	5	4	8	
	2	4	1	3	2	
	2	4	2	8	5	
	3	2	6	9	7	
	3	4	3	1	3	
	3	4	4	0	6	
	4	1	2	3	9	
	4	7	4	8	0	
	5	2	2	7	8	-- This row qualifies
	5	2	3	8	1	
	5	3	2	7	5	-- This row qualifies
	5	5	3	7	3	
	5	6	2	5	8	
	5	6	2	7	7	-- This row qualifies
	5	8	2	7	4	-- This row qualifies
	5	9	3	7	3	
	5	9	4	7	1	

```

6      3      2      4      3
Qualified rows:
5      2      2      7      8
5      3      2      7      5
5      6      2      7      7
5      8      2      7      4

```

Note: Logically removing columns from an index key has no effect on the filtering capability of the index.

13.4 INSERT

In this section, we discuss enhancements made to the INSERT statement.

The new enhancements are:

- ▶ Inserting with DEFAULT keyword
- ▶ Inserting with any expression
- ▶ Inserting with self-referencing select
- ▶ Inserting with UNION

13.4.1 Using the DEFAULT keyword in VALUES clause of an INSERT

Using the DEFAULT keyword in the VALUES clause of an INSERT statement can be very practical when inserting using dynamic SQL. All the columns the end user does not provide a value for, can be inserted using DEFAULT. The default value used is the values defined on the table or the system default for each data type if one is not specified in the table definition. However, the column must be defined with WITH DEFAULT.

The last column in the definition of the table SC246300.TBITEMS is:

```
COMMENT VARCHAR (100 ) WITH DEFAULT 'NONE'.
```

In Example 13-15, we show how we can code an INSERT to take advantage of this keyword.

Example 13-15 Inserting with the DEFAULT keyword

```

INSERT INTO SC246300.TBITEMS
VALUES (440
       , 'HAMMER'
       , 50
       , 1.25
       , DEFAULT) ;

```

Inserts the following row:

```

-----+-----+-----+-----+-----+-----+-----+
ITEM_NUMBER  PRODUCT_NAME      STOCK      PRICE  COMMENT
-----+-----+-----+-----+-----+-----+
          440    HAMMER                50        1.25    NONE

```

13.4.2 Inserting using expressions

You can specify any expression in the list of values of an INSERT statement. This can be very useful with user defined functions, cast functions on user-defined distinct data types, values based on arithmetic, and new datetime functions.

Example 13-16 Inserting using an expression

```
INSERT INTO SC246300.TBITEMS
VALUES (445
      , 'TELEVISION'
      , 30
      , SC246300.PES2EUR(15000)
      , DEFAULT);
```

```
-----+-----+-----+-----+-----+-----+-----+
ITEM_NUMBER  PRODUCT_NAME      STOCK      PRICE  COMMENT
-----+-----+-----+-----+-----+-----+-----+
          445    TELEVISION              30      90.00  NONE
```

13.4.3 Inserting with self-referencing SELECT

Before DB2 V6, if you wanted to INSERT rows into a table, based on a selection of rows from that same table, you had to implement views on that table, to give DB2 the impression you were using two different tables in the statement. Now this is no longer needed. The fullselect (in V7, see 13.4.4, "Inserting with UNION or UNION ALL" on page 193) that you specify in the INSERT statement can now be a SELECT from the same table that returns more than a single row. This is a very practical way to create more rows based on the rows you already have in the table, varying values with arithmetic expressions as well as with functions.

Example 13-17 Inserting with a self-referencing SELECT

```
-- Before Version 6 this was the only self-referencing INSERT allowed
```

```
INSERT INTO SC246300.TBEMPLOYEE (EMPNO
                                ,SALARY
                                ,SEX)
SELECT 'AVG',AVG(SALARY),'F'
FROM SC246300.TBEMPLOYEE;
-- one row is inserted
```

```
-- Full self-referencing support in Version 7
```

```
SET CURRENT PATH = 'SC246300';

INSERT INTO SC246300.TBCONTRACT (BUYER
                                ,SELLER
                                ,RECNO
                                ,EUROFEE)
SELECT BUYER
      ,SELLER
      ,SUBSTR(RECNO,1,5) || 'COPY'
      , PES2EUR(DECIMAL(PES2AFEE))
FROM SC246300.TBCONTRACT
-- many rows are inserted
```

13.4.4 Inserting with UNION or UNION ALL

We can now (V7) insert rows into a table by using UNION or UNION ALL in the SELECT statement. This enhancement can be very useful, especially when populating temporary tables in a data warehouse environment.

See Example 11-6 on page 139 for more details.

13.5 Subselect UPDATE/DELETE self-referencing

This enhancement to SQL in DB2 V7 allows searched UPDATE and DELETE statements to use the target tables within the subselect in the WHERE clause. Before DB2 V7, if you attempted to do this, DB2 returned an SQLCODE -118.

Note: This feature requires the subquery to be completely evaluated before any rows are updated or deleted.

Example 13-18 shows how we can give a 10% salary increase to all employees with a salary lower than the average salary.

Example 13-18 UPDATE with a self referencing non-correlated subquery

```
UPDATE SC246300.TBEMPLOYEE
  SET SALARY = SALARY * 1.10
  WHERE SALARY < (SELECT AVG(SALARY)
                  FROM SC246300.TBEMPLOYEE )
```

A non-correlated subquery is executed only once before update and delete.

Example 13-19 shows how we can give a 10% salary increase to all employees whose salary is lower than the average salary of their department.

Example 13-19 UPDATE with a self referencing non-correlated subquery

```
UPDATE SC246300.TBEMPLOYEE X
  SET SALARY = SALARY * 1.10
  WHERE SALARY < (SELECT AVG(SALARY)
                  FROM SC246300.TBEMPLOYEE Y
                  WHERE X.WORKDEPT = Y.WORKDEPT) ;
```

DB2 processes the query in Example 13-19 as follows:

1. The correlated subquery is executed for each row in the outer (TBEMPLOYEE X) table. DB2 creates a record, in a work file, for each row that satisfies the subquery. For the UPDATE statement, DB2 stores the RID of the row and the updated SALARY column value. For the DELETE statement, DB2 stores the RID of the row.
2. Once all the qualifying rows have been determined, DB2 reads the work file, and for each record, updates or deletes the corresponding row in the TBEMPLOYEE table.

This two-step processing is not shown in the EXPLAIN output. Two-step processing is used for an UPDATE whenever a column that is being updated is also referenced in the WHERE clause of the UPDATE or is used in the correlation predicate of the subquery as shown in Example 13-19. For a DELETE statement, two-step processing is always used for a correlated subquery.

Example 13-20 shows a new way to delete the department with the lowest budget in just one SQL sentence. However, be careful when coding such a statement because even though only one value is returned by the self-referencing subselect, more than one row may have a budget equal to the minimum and thus would be deleted.

Example 13-20 DELETE with self referencing non-correlated subquery

```
DELETE FROM SC246300.TBDEPARTMENT
WHERE BUDGET = (SELECT MIN(BUDGET)
                FROM SC246300.TBDEPARTMENT)
```

Example 13-21 shows how we can delete the employees with the highest salary for each department. To perform this DELETE in our sample environment, we have to drop the self referencing foreign key of the TBEMPLOYEE table since it is defined with the delete rule ON DELETE NO ACTION.

Example 13-21 DELETE with self referencing non-correlated subquery

```
DELETE FROM SC246300.TBEMPLOYEE X
WHERE SALARY = (SELECT MAX(SALARY)
                FROM SC246300.TBEMPLOYEE Y
                WHERE X.WORKDEPT= Y.WORKDEPT)
```

The enhanced support for the UPDATE and DELETE statements also improves DB2 family compatibility.

Restrictions on usage

This enhancement does not extend to a positioned UPDATE or DELETE statement, that is, an UPDATE or DELETE statement which uses the WHERE CURRENT OF cursor-name clause.

DB2 positioned updates and deletes continue to return the SQLCODE -118 if a subquery in the WHERE clause references the table being updated or which contains rows to be deleted.

For example, the positioned update in Example 13-22 is still invalid.

Example 13-22 Invalid positioned update

```
EXEC SQL DECLARE CURSOR C1 CURSOR FOR
        SELECT T1.ACCOUNT, T1.ACCOUNT_NAME, T1.CREDIT_LIMIT
        FROM ACCOUNT T1
        WHERE T1.CREDIT_LIMIT < (SELECT AVG(T2.CREDIT_LIMIT)
                                FROM ACCOUNT T2)
        FOR UPDATE OF T1.CREDIT_LIMIT;
.
EXEC SQL OPEN C1;
...
EXEC SQL FETCH C1 INTO :hv_account, :hv_acctname, :hv_crdlmt;
...
```



```
EXEC SQL UPDATE ACCOUNT
        SET CREDIT_LIMIT = CREDIT_LIMIT * 1.1
        WHERE CURRENT OF C1;
```

An SQLCODE -118 is returned at bind time.

13.6 Scalar subquery in the SET clause of an UPDATE

Beginning with DB2 V6, a scalar subselect can now be specified on the SET clause of both a searched and positioned UPDATE statement. A scalar subselect is a subselect that returns a single row. The number of columns in the row must match the number of columns that are specified to be updated.

Example 13-23 shows how to use a subquery that returns a single row in the SET clause of an update. Before V6 you had to use two SQL statements.

Example 13-23 Non-correlated subquery in the SET clause of an UPDATE

```
UPDATE SC246300.TBEMPLOYEE
        SET DEPTSIZE = (SELECT COUNT(*)
                        FROM SC246300.TBDEPARTMENT
                        WHERE DEPTNO = 'A01')
        WHERE WORKDEPT = 'A01'
```

Note: If the subselect returns no rows, the null value is assigned to the column to be updated, if the column cannot be null, an error occurs.

The columns of the target table or view of the UPDATE can be used in the search condition of the subselect. Using correlation names to refer to these columns is only allowed in a searched UPDATE.

In Example 13-24 we update the manager of all employees with the manager assigned to their department. It is up to the user to make sure only a single row is returned in the subselect. Otherwise you receive an SQLCODE -811.

Example 13-24 Correlated subquery in the SET clause of an UPDATE

```
UPDATE SC246300.TBEMPLOYEE X
        SET MANAGER = (SELECT MGRNO
                        FROM SC246300.TBDEPARTMENT Y
                        WHERE X.WORKDEPT = Y.DEPTNO
                        ) #
```

In Example 13-25 we update the number of orders (NUMORDERS column) for all employees with a NATIONKEY of 34.

Example 13-25 Correlated subquery in the SET clause of an UPDATE with a column function

```
UPDATE SC246300.TBEMPLOYEE X
        SET NUMORDERS = (SELECT COUNT(*)
                        FROM SC246300.TBORDER
                        WHERE CLERK = X.EMPNO)
```

```
WHERE NATIONKEY = 34
```

Example 13-26 shows how the update in Example 13-23 can be changed to provide DEPTSIZE for employees of all departments. This is done by using an UPDATE with a correlated subquery in the SET clause.

Example 13-26 Correlated subquery in the SET clause of an UPDATE using the same table

```
UPDATE SC246300.TBEMPLOYEE X
SET DEPTSIZE = (SELECT COUNT(*)
                FROM SC246300.TBEMPLOYEE Y
                WHERE X.WORKDEPT = Y.WORKDEPT) ;
```

Example 13-27 shows how to move employees Mark and Ivan to department number 'A01' using a row expression in the SET clause of an UPDATE.

Example 13-27 Row expression in the SET clause of an UPDATE

```
UPDATE SC246300.TBEMPLOYEE
SET (MANAGER,WORKDEPT) = (SELECT MGRNO,DEPTNO
                          FROM SC246300.TBDEPARTMENT
                          WHERE DEPTNO = 'A01')
WHERE FIRSTNME IN ('MARK','IVAN')
```

The SET assignment statement assigns the value of one or more expressions or a NULL value to one or more host-variables or transition-variables and replaces the SET host-variable statement documented in previous releases of DB2.

The statement can be embedded in an application program or can be contained in the body of a trigger. If the statement is a triggered SQL statement, then it must be part of a trigger whose action is BEFORE and whose granularity is FOR EACH ROW. In this context, a host-variable cannot be specified.

This enhancement also improves DB2 family compatibility.

13.6.1 Conditions for usage

The subselect can reference a table, view, synonym or alias or a join of any of these. You must make sure the number of columns selected is equal to the number of columns to be updated and the data types must be compatible.

You need to ensure that the subselect does not return more than one value (row), as the statement would then fail with SQLCODE -811. Since only one row must be returned from the subselect, you cannot use the GROUP BY and HAVING clause, if you do, you get an SQLCODE -815. You should also consider whether it is possible for the statement to return no rows. In this case, the null value is assigned to the column(s) to be updated. If a column does not accept null values an SQLCODE -407 is returned. Consequently, this function is ideally suited when you want to update a column to the result of functions such as COUNT, MAX, SUM or where the subselect is accessing data by its primary key.

13.6.2 Self-referencing considerations

When using a subselect in the SET clause of an UPDATE the object of the update and the subselect must not be the same table.

However, for a searched update, you may reference a column in the table to be updated within the subselect.

13.7 FETCH FIRST n ROWS ONLY

DB2 V7 introduces the FETCH FIRST n ROWS ONLY clause, which allows you to specify a limit on the number of rows returned into the result set. This clause has been introduced to support ERP vendor products, many of which require only the first row to be returned. This enhancement is of particular value for distributed applications, but it is also applicable to local SQL.

Performance is improved with DRDA applications, since the client application can specify limits to the amount of data returned and DRDA closes the cursor implicitly when the limit is met.

Example 13-28 gives a simple example of the usage of the FETCH FIRST n ROWS ONLY clause. In this case the cursor receives an SQLCODE +100 at the sixth FETCH operations (after 5 ROWS have been returned), even though there are more rows that qualify the predicates.

Example 13-28 FETCH FIRST n ROWS ONLY

```
SELECT T1.CREATOR
       ,T1.NAME
FROM SYSIBM.SYSTABLES T1
WHERE T1.CREATOR = 'SYSIBM'
      AND T1.NAME LIKE 'SYS%'
ORDER BY T1.CREATOR
        ,T1.NAME
FETCH FIRST 5 ROWS ONLY;
```

CREATOR	NAME
SYSIBM	SYSAUXRELS
SYSIBM	SYSCHECKDEP
SYSIBM	SYSCHECKS
SYSIBM	SYSCHECKS2
SYSIBM	SYSCOLAUTH

DSNE610I NUMBER OF ROWS DISPLAYED IS 5
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 100

The FETCH FIRST clause can also be used for scrollable cursors.

The OPTIMIZE FOR clause is another that you can specify in the SELECT statement. Table 13-1 describes the effect of specifying the FETCH FIRST clause with and without the OPTIMIZE FOR clause.

Note: The behavior described in the following table only applies when the PTF for APAR PQ49458 (still open at the time of writing) is applied to your system.

Table 13-1 How the FETCH FIRST clause and the OPTIMIZE FOR clause interact

Clauses specified on the SELECT statement	OPTIMIZE value used by DB2
FETCH FIRST <i>n</i> ROWS ONLY (OPTIMIZE FOR clause not specified)	DB2 optimizes for <i>n</i> rows
FETCH FIRST <i>n</i> ROWS ONLY OPTIMIZE FOR <i>m</i> ROWS (where $m > n$)	DB2 optimizes for <i>m</i> rows
FETCH FIRST <i>n</i> ROWS ONLY OPTIMIZE FOR <i>m</i> ROWS (where $m < n$)	DB2 optimizes for <i>m</i> rows

When both options are specified, only the OPTIMIZE FOR clause is used during access path selection and determining how to do blocking in a DRDA environment.

By using the new FETCH FIRST *n* ONLY clause, you can:

- ▶ Limit the number of rows in a result table
- ▶ Use a SELECT INTO statement to retrieve the first row of a result table
- ▶ Limit the number of rows returned by a DB2 for z/OS and OS/390 DRDA server

13.8 Limiting rows for SELECT INTO

In previous versions of DB2, using the SELECT...INTO required the program to ensure that only a single row was returned. This was normally done by using the primary key or any other unique key that existed on the table. To guarantee that only a single row was returned was more complicated if the SELECT statement contained a join to another table which could have any number of rows matching the join criteria.

If the SELECT...INTO statement is coded and returns more than one row, DB2 returns an SQLCODE -811, and the statement would be rejected. You can use this statement for existence checking (to see if one or more rows qualify) when you don't actually need the data itself. However, in order to be sure there is only one row that qualifies, DB2 has to continue looking through the data and applying predicates to make sure there is no second row. This does take up unnecessary resources when you want to find out if a row exists. You don't need to know there is that there is more than one.

Because of the SQLCODE -811, there is no guarantee that the host variables are actually filled with the values returned from the query.

Therefore, if you needed to see the data itself, and if there was no way to ensure that only a single row was returned, then a cursor had to be opened and the program itself would have to read in the first row that matches the selection and join criteria and throw away all other rows by closing the cursor.

The FETCH FIRST 1 ROWS ONLY clause can now be added to the SELECT ... INTO statement to specify that only one row is returned to the program, even if multiple rows match the WHERE criteria. Wherever uniqueness is not significant, this clause can be very powerful. In case of the existence checking, DB2 does not look for a second row and not return an SQLCODE -811.

Example 13-29 shows how to use a SELECT INTO statement that could return several rows. By adding the FOR FETCH ONLY we can avoid having to open a cursor.

Example 13-29 Limiting rows for SELECT INTO

```
SELECT ACCOUNT
      ,ACCOUNT_NAME
      ,TYPE
      ,CREDIT_LIMIT
INTO   :hv_account
      ,hv_acctname
      ,:hv_type
      ,:hv_crdlmt
FROM ACCOUNT
      WHERE ACOCUNT_NAME = :hv_in_acctname
      FETCH FIRST 1 ROW ONLY ;
```

As you can choose to pick up only a row, the SELECT INTO statement support also the GROUP BY and HAVING clauses.

Note: Be cautious when using this feature, make sure that logically (within your application) it is appropriate to ignore multiple rows.

13.9 Host variables

In this section we discuss some topics related to host variables, including:

- ▶ The new VALUES INTO statement and
- ▶ The fact that host variables must now be preceded by a colon ':'

13.9.1 VALUES INTO statement

The VALUES INTO statement assigns the values of one or more expressions to one or more host variables. This statement can only be embedded in an application program as shown in Example 13-30.

Example 13-30 Use of VALUES INTO

```
EXEC SQL VALUES(CURRENT DATE)
      INTO :HV1;
```

The SET assignment statement is an alternative to assign the value of one or more expressions or a NULL value to one or more host-variables or transition-variables. In Example 13-31, we show several ways to use the SET.

The implementation of VALUES INTO is another enhancement to increase DB2 family compatibility.

Example 13-31 Some uses of the SET assignment statement

```
SET :hv = CUSTOMER('01')
This is equivalent to VALUES(CUSTOMER('01')) INTO :hv
The SET assignment statement is recommended
```

```
SET :hv = NULL
SET :hv = udf1(:hv2,1,1+2)
SET :hv = udf2(:hv2,1)
SET :lastmon = MONTH(CURRENT DATE - 30 DAYS)
```

13.9.2 Host variables must be preceded by a colon

DB2 Version 6 enforces the standard that requires all host variables to be preceded by a colon ":". All host variable references *must* have the leading colon. If you neglect to use a colon to designate a host variable, the precompiler issues a DSNH104I message or interprets the host variable as an unqualified column name, which might lead to unintended results.

Background information

The colon was an option in DB2 V1 code and documentation. It was not an option for some other products, and the optional colon did not get accepted into the SQL standard. With DB2 V2, the manuals were changed to indicate that the colon should always be specified, and a warning message, DSNH315I, was implemented. Most customers noticed the warnings and corrected their applications to comply with the standard.

DB2 V3, V4, and V5, continued to allow the optional colons with the warning messages. With DB2 V6 the level of complexity of SQL was such that it was decided not to allow the colons to remain optional: when DB2 V6 was announced in 1998, this incompatible change was included in the announcement, and in the documentation.

Impact of the DB2 V6 restriction

Applications that contain a host variable reference that do not have the leading colon fail with an error instead of the informational message DSNH315I. In order to anticipate any problem and find out if you have host variables that are not preceded with a colon, you can just precompile your programs again before migrating to DB2 V6 and check for the warning message DSNH315I.

With the DB2 V6 precompiler an error message is produced, generally DSNH104I. The only way to correct the problem is to correct the source and precompile again. This assumes that the source is available.

If you cannot change the source to put in the colons to fix the problem, then you can use a precompiler from DB2 V4 or V5.

BIND and REBIND on DB2 V6 can also fail for DBRMs that were created prior to DB2 Version 2 Release 3. For example, if you drop an index that is used in a package or plan, then the package or plan must be rebound, either by explicit command or automatic BIND. If the DBRM was produced on DB2 Version 2 Release 2 or earlier, then the BIND or REBIND fails.

DBRMs produced by the DB2 Version 2 Release 3 or later precompiler have a colon in the DBRM, even if the source code does not.

If you still have a host variable without a preceding colon and you have migrated to DB2 V6, the error that you receive depends on when the application was originally precompiled. If it was precompiled with the Version 2 Release 2 precompiler (or earlier), you get an error when you try to rebind or when an automatic rebind occurs. If you do a second bind using the DBRM from the Version 2 Release 2 precompiler, you get a failure as well. If you re-compile at V6, you receive a syntax error from the precompiler or a bind error. If the application was bound on Version 2 release 3 or later, you should have no problems with rebind, automatic rebind, or a second bind using the same DBRM.

For cases where you have the source code, the best resolution is running the precompiler and then binding the new DBRM.

The best practice is to set an application programming standard and to add the colon in all DB2 applications. If the return code from the precompiler is not zero, then the warning could cause problems in production.

If you do not have the source code, the options are very limited. APAR II12100 may provide some help. APARs PQ26922 and PQ30390 may be applicable in some cases.

Detecting the problem

You can detect the problem in source code by precompiling. If the source code is not readily available, and you only want to detect the problems which would occur on a BIND or REBIND, then the following technique may be useful.

A sample REXX procedure, which analyzes all Format 1 DBRMs (pre V2.3) to check whether all host variables are preceded with a colon, is available from the Internet. The REXX/DB2 interface is used, and therefore DB2 V5 or V6 is required. The procedure creates temporary EXEC libraries, copies the REXX EXEC, executes DSNTIAUL using PARM('SQL') to extract data from the catalog, extracts DBRM listings from the catalog, executes the REXX to analyze the output looking for missing colons preceding host variables ":hv", and produces a report.

You need to examine the exceptions identified by the REXX program. It should be obvious where you need to amend the source SQL and re-compile.

The program is a sample only and is provided without any implied warranty or support. We have not checked all eventualities, so we cannot guarantee that every invalid DBRM is found, but it can assist you with the migration.

The procedure is called *DBRM Colon Finder* and it is available from the URL:

<http://www.ibm.com/software/db2/os390/downloads.html>

13.10 The IN predicate supports any expression

The IN predicate has been enhanced to allow any expression in the list of values. Example 13-32 shows the use of a scalar cast function in an IN list.

Example 13-32 IN predicate supports any expression

```
SET CURRENT PATH = 'SC246300' ;

SELECT BUYER
       ,SELLER
       ,RECNO
       ,PESETAFEE
FROM SC246300.TBCONTRACT
WHERE BUYER IN (CUSTOMER('01'),CUSTOMER('03'))
```

DB2 now also supports multiple expressions (row expressions) on the left-hand side of the IN and NOT IN predicate when a fullselect is specified on the right-hand side. Example 13-10 on page 187 shows a row expression and a fullselect into a NOT IN list.

13.11 Partitioning key update

DB2 V6 (retrofitted into V5 with PQ16946 offers the possibility to update a partitioned key, instead of deleting and inserting the row.

Note: This feature is available only for tables created under V6 (or V5 with APAR PQ16946).

If your partitioned table is created before V6 and you want the partitioning key to be updatable, remember that when you drop and create the table, plans and packages must be rebound, authorizations on the dropped table re-granted, synonyms, views, and referential constraints re-created.

Tip: Applications that use UPDATEs on the partitioning key (instead of DELETEs and INSERTs) could be tested successfully in a test environment, but they may not run successfully in a production environment if the production table was created prior to DB2 V6.

If an update moves a row from one partition to another, partitions are drained and the application may deadlock or time out, because of the same restrictions on claim/drain processing.

DB2 administrators should not allow a partitioned key to be updatable in systems with high concurrency requirements and long running transactions. This can be controlled via a new ZPARM (PARTKEYU). This parameter gives you the option to either disable the partitioning key update feature, allow the update to move the key to a different partition, or limit the update to stay in the same partition.



Part 4

Utilities versus applications

In this part we look at some of the enhancements that were made to the DB2 utilities, that enable them to do things that you could only do in an application program before. Therefore, it might be a good time to re-evaluate some of the choices that were made in the past and look for a new balance between what to do in an application program and what functions can be done by a DB2 utility.



Utilities versus application programs

Existing DB2 utilities have been enhanced and new utilities have been introduced that can take over some of the work that was previously done by application programs.

In this chapter we describe some ideas where these enhancements can be used and compare them to implementing the same processes using application code, such as:

- ▶ Online LOAD RESUME versus program insert
- ▶ REORG DISCARD versus deleting rows via a program followed by a REORG
- ▶ Using REORG UNLOAD EXTERNAL and UNLOAD instead of a home grown program or the DSNTIAUL sample program
- ▶ Using SQL statements in the utility input stream

14.1 Online LOAD RESUME

Availability requirements differ depending on the installation. For data warehouse environments, where queries can run for many hours, availability means something different than for a 24/24 internet application. For many installations the ability to load new rows into tables without interfering with other users is crucial and they are willing to trade off utility performance for availability and easy maintenance. Now it is possible to run an online LOAD RESUME instead of coding and maintaining an INSERT program or having a service interruption to load new rows.

In this section, we discuss the LOAD RESUME SHRLEVEL CHANGE that was introduced in V7. This 'online version' of the LOAD utility allows other users access to the table(s) while the data is being loaded. With this type of LOAD utility, installations should have another look whether they still have to develop (and maintain) their own batch SQL INSERT programs.

14.1.1 What is online LOAD RESUME?

SHRLEVEL is a new keyword for the LOAD utility in V7 and it specifies the extent to which applications can concurrently access the table space or partition(s) being loaded.

SHRLEVEL NONE (default) specifies that applications have no concurrent access to the table space or partition(s) being loaded, that is, the LOAD utility operates the same as prior to version 7. This type of LOAD is also referred to as classic LOAD throughout this chapter.

The term 'online' refers to the mode of operation of the LOAD utility when SHRLEVEL CHANGE is specified. When using online LOAD RESUME, other applications can concurrently issue SQL SELECT, UPDATE, INSERT and DELETE statements against table(s) in the table space or partition(s) being loaded.

In brief, by using online LOAD RESUME, you can load data with minimal impact on SQL applications and without writing and maintaining INSERT programs.

14.1.2 Why use Online LOAD RESUME

The classic LOAD drains all access to the table space or partitions (including the associated indexes or index partitions). Therefore, the data is not available to SQL applications. If end-users require access to a table while new rows are being loaded because of availability reasons, prior to V7 a SQL batch program was needed to INSERT the new rows.

Online LOAD RESUME behaves externally like a classic LOAD, but it works internally like a mass INSERT.

Online LOAD RESUME uses a data manager INSERT process to load rows into the table space or partition(s). Online load resume behaves like a normal insert program using up free space, taking locks, logging the inserts and trying to maintain clustering order of the data. Although online LOAD RESUME takes locks like a normal program, the utility monitors the lock situation constantly and dynamically adapts the commit interval to avoid impacting other programs.

Data integrity cannot be always assured by only using foreign keys and check constraints. In some cases, triggers are additionally used to ensure the correctness of the data. The classic LOAD does not activate triggers, which may result in a data integrity exposure after loading new data. The new online LOAD RESUME utility operates like SQL INSERTs, therefore, triggers are activated, check constraints are checked as well as referential integrity relationships. Therefore, data integrity is guaranteed at all times. The cost is that, compared

to the classic LOAD, online LOAD RESUME runs longer. But on other hand, all tables are available to other users all the time. Also when you compare online LOAD RESUME to a program doing massive SQL INSERTs, online LOAD RESUME is faster and consumes less CPU. For more details, see *DB2 for z/OS and OS/390 Version 7 Performance Topics*, SG24-6129.

14.1.3 Online LOAD RESUME versus classic LOAD

These are some of the consequences of online LOAD RESUME processing.

Claiming, draining and locking

Whereas the classic LOAD drains the table space, thus inhibiting any SQL access, online LOAD RESUME acts like a normal INSERT program by using claims when accessing an object. That is, it behaves like any other SQL application and can run concurrently with other, even updating, SQL programs.

Online LOAD RESUME also takes locks the same way normal SQL applications do. In order to avoid locking out other applications, the online LOAD RESUME utility constantly monitors the lock situation and dynamically adapts the commit interval to avoid impacting other programs.

When an online LOAD RESUME utility has to wait for a lock, it behaves like a normal utility, using the utility time-out multiplier zparm (UTIMOUT) to determine the amount of time to wait for a lock.

Logging

Only LOG YES is allowed. Therefore, no COPY is required afterwards. If you are thinking about converting from classic LOAD to online LOAD RESUME for large tables, you may want to check your DB2 logging environment, like the number of active logs, their placement on the disk volumes and fast devices, consider striping and increasing DB2 log output buffer size.

RI

Referential integrity is enforced when loading a table with online LOAD RESUME.

When you use online LOAD RESUME on a self-referencing table, it forces you to sort the input in such a way that referential integrity rules are met, rather than sorting the input in clustering sequence, which you used to do for classic LOAD.

Duplicate keys

The handling of duplicate keys is somewhat different when using online LOAD RESUME compared to the classic LOAD utility. When using online LOAD RESUME and a unique index is defined, INSERTs (done by the online LOAD) are accepted as long as they provide different values for this column (or set of columns). This is different from the classic LOAD procedure, which discards all rows that you try to load having the same value for a unique key.

You may have setup a procedure (manual or automated) to handle the rows classic LOAD discards. When you move over to online LOAD RESUME, you have to change handling the discarded rows accordingly.

Clustering

Whereas the classic LOAD RESUME stores the new records (in the sequence of the input) at the end of the already existing records, online LOAD RESUME tries to insert the records into the available free space respecting the clustering order as much as possible. When you have to LOAD (insert) a lot of rows, make sure there is enough free space available. Otherwise these rows are likely to be stored out of the clustering order and you might end up having to run a REORG to restore proper clustering (which can be run online as well).

So, a REORG may be needed after the classic LOAD, as the clustering may not be preserved, but also after an online LOAD RESUME if no sufficient free space is available.

Free space

As mentioned before, the available free space, PCTFREE or FREEPAGE, is used by online LOAD RESUME, in contrast to the classic LOAD.

As a consequence, a REORG may be needed after an online LOAD RESUME to ensure sufficient free space (PCTFREE and FREEPAGE) is available for subsequent inserts.

Figure 14-1 shows an example of the online LOAD RESUME syntax and the inserts that it performs. Some differences with the classic LOAD are listed.

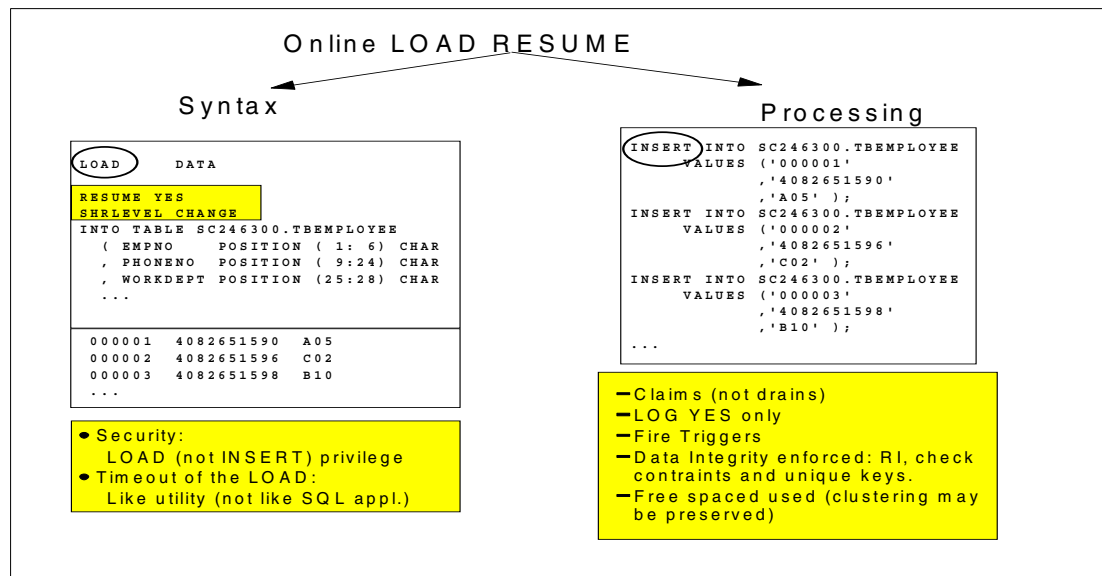


Figure 14-1 Online LOAD RESUME

14.1.4 Online LOAD RESUME versus INSERT programs

When comparing online LOAD RESUME with coding a mass INSERT program, there are a number of reasons to favour the LOAD utility over a normal application program.

Commit frequency

Online LOAD RESUME dynamically monitors the current locking situation for the table spaces or partitions being loaded. This enables online LOAD RESUME to choose the commit frequency and avoid lock contention with other SQL. This kind of commit frequency flexibility is not possible to code at batch insert programs, though the frequency can be changed while program is running.

Restart

During RELOAD, internal commit points are set, therefore, RESTART(CURRENT) is possible as with the classic LOAD. When using an INSERT program, application repositioning techniques are needed to be able to restart, which is not all that easy when sequential files are involved, especially when writing them (for example, records that could not be inserted for some reason).

Phases

Some utility phases are obviously not included in the online LOAD RESUME, as this kind of LOAD operates like SQL INSERTs. But the DISCARD and REPORT phase are still performed. Input records which fail the insert are written to the discard data set and error information is stored in the error data set. Batch INSERT programs have to take care of records which were not inserted. Application programs allow more refined handling of errors, than just finding duplicates and data type violations that the LOAD utility does. On the other hand, if that type of checking is required, you can write a checking routine to validate input before you start loading the data into the tables.

Data Manager rather than RDS INSERT

Online LOAD RESUME uses a 'cheaper' interface (Data Manager) to INSERT (load) rows into a table. Normal SQL INSERTs use the Relational Data System (RDS) interface. Both Data Manager and RDS are DB2 subcomponents.

Some reasons for still using batch INSERT program

- ▶ In some cases you need to be able to control the time at which you commit, for example, to force a commit after a logical unit of work completes. When mass inserting orders, after inserting an order (or several orders) in the TBORDER table and its dependent line items in the TBLINEITEMS table we need to force a commit point in our application. When the online LOAD RESUME utility is in control, you cannot be sure that the internal commit takes place when a complete unit of work (and order and its dependent line items) has been handled. Since online LOAD RESUME runs concurrently with other processes, another program might pick up an incomplete order. If this type of processing is required, you may want to code your own mass insert program to have this extra level of control.
- ▶ Another reason for using or continuing to use home grown applications is when the program does not only insert data, but also does other processing, for example, updating summary tables.

If the additional processing that is done is fairly limited, for example, just keeping a summary record up to date with a grand total, you might be able to implement this by defining a trigger on the table to handle this summary update.

14.1.5 Online LOAD RESUME pitfalls

Load parent rows before dependent rows to avoid referential constraint violations.

Be careful not to produce inconsistent data for other applications. Remember that other transactions and programs are running while you are loading data. This was not the case with the classic LOAD. All other access was drained. No-one wants to see orders without line items.

After running online LOAD resume, it is recommended to run RUNSTATS and depending on the result, run a REORG to restore clustering sequence and free space.

14.1.6 Online LOAD RESUME restrictions

If LOAD SHRLEVEL CHANGE is specified, then some other LOAD options cannot be used. Running LOAD SHRLEVEL CHANGE can also cause conflicts with other utilities when they run concurrently on the same object.

- ▶ RESUME NO. RESUME YES must be specified at the table space level, or for all INTO TABLE PART specifications.
- ▶ REPLACE. That is to comply with the RESUME YES keyword.

- ▶ LOG NO. Online LOAD RESUME always writes to the DB2 log.
- ▶ ENFORCE NO. Inserts done by online LOAD RESUME always enforce RI and Check constraints.
- ▶ STATISTICS. Online LOAD does not gather inline statistics.
- ▶ COPYDDN/RECOVERYDDN. Online LOAD does not create inline image copies
- ▶ PART integer REPLACE
- ▶ INCURSOR. You cannot use the cross-loader functionality with online LOAD.
- ▶ Online LOAD RESUME and REORG (including online REORG) cannot concurrently process the same target object.

For a complete list of options and other utilities that are incompatible with LOAD SHRLEVEL CHANGE, see *DB2 UDB for OS/390 and z/OS Version 7 Utility Guide and Reference*, SC26-9945.

14.2 REORG DISCARD

In this section, we describe and discuss the REORG DISCARD option that was introduced in DB2 V6 and retrofitted back to V5 with PQ19897 and PQ23219.

14.2.1 What is REORG DISCARD?

The REORG DISCARD enhancement allows you to discard unwanted rows during a normal REORG. This capability is mutually exclusive with UNLOAD EXTERNAL, but shares much of the same implementation.

14.2.2 When to use a REORG DISCARD

A REORG DISCARD can be used successfully in table spaces that require periodic mass DELETES. Very often after the mass delete, the table space is disorganized and needs to be reorganized afterwards. Now you can combine the delete operation with running REORG in a single operation. Having REORG discard rows is much cheaper than application DELETES. You can have REORG send the discarded rows to a file, because very often you don't just want to delete rows from a table, what you actually want is to archive them (for example, for legal reasons). It is possible even to LOAD the discarded rows if they are needed again in the future.

Some pages might need more free space than others. For example, the table is clustered by customer and you have some very active customers, who have a lot of transactions. You can then run the REORG first while discarding rows from less active customers and delete the rows for the active customers afterwards with an SQL DELETE operation to gain that extra space near the 'hot' pages. This is one case where you can consider to continue using a batch programs to do the deletes or rethink your free space strategy.

14.2.3 Implementation and maintenance

The output of the discarded records is similar to the output of running a REORG UNLOAD EXTERNAL against the table. That is, the data is in external format and LOAD utility statements can be generated. Similarly, the criteria for discard are specified in a WHEN clause. A sample REORG DISCARD utility statement is shown in Example 14-1.

Example 14-1 REORG DISCARD utility statement

```
REORG TABLESPACE DB246300.TS246304
DISCARD
  FROM TABLE SC246300.TBEMPLOYEE
  WHEN (EMPNO = '000001')
```

The WHEN conditions that you can specify are a simple subset of what can be coded on a WHERE clause. WHEN conditions allow AND-ing, OR-ing of selection conditions and predicates. Comparison operators are =, >, <, <>, <=, >=, IS (NOT) NULL, (NOT) LIKE, (NOT) BETWEEN and (NOT) IN. The predicates that are allowed are only comparisons between a column and a constant or a labeled-duration-expression (like CURRENT DATE + 30 DAYS).

Discarded rows are written to a SYSDISC data set or a DD-name specified with DISCARD DDN keyword.

Important: If no discard data set is specified, the discarded records are lost.

Either UNLOAD EXTERNAL or REORG DISCARD can generate the same LOAD control statements, based on the data being processed. A sample LOAD statement generated by REORG DISCARD is shown in Figure 14-2.

```
LOAD DATA LOG NO INDDN SYSREC
EBCDIC CCSID (500,0,0)
INTO TABLE "SC246300"."TBEMPLOYEE"
WHEN (00004:00005 = X'0012')
( "EMPNO      " POSITION(00007:00012) CHAR(006)
, "PHONENO    " POSITION(00013:00027) CHAR(015)
, "WORKDEPT   " POSITION(00028:00030) CHAR(003)
, ...
, ...
, "SALARY     " POSITION(00091:00095) DECIMAL    NULLIF(00090)=X'FF'
, ...
)
```

Annotations in the image:

- Arrow pointing to LOG NO INDDN SYSREC: may be ASCII unloaded in same CCSID as stored
- Arrow pointing to INTO TABLE "SC246300"."TBEMPLOYEE": this identifies the table

Figure 14-2 Generated LOAD statements

14.2.4 REORG DISCARD restrictions

Some of the restrictions that apply to REORG DISCARD are:

- ▶ No column to column comparisons are allowed
- ▶ No LIKE on columns with FIELDPROCS
- ▶ Literal values for ASCII data must be in hex
- ▶ DISCARD not allowed with SHRLEVEL CHANGE
- ▶ Difficulty correlating discards with pointers

14.3 REORG UNLOAD EXTERNAL and UNLOAD

In this section, we describe and discuss REORG UNLOAD EXTERNAL, made available in DB2 V6 and retrofitted back to V5 with PQ19897 and PQ23219, and the UNLOAD utility that was introduced in DB2 V7.

14.3.1 What are REORG UNLOAD EXTERNAL and UNLOAD?

Many installations want to be able to unload data into a user friendly format, and to do it quickly (several times faster than DSNTIAUL, for example).

REORG UNLOAD ONLY is fast, but places the data in a internal format that is distinctly not user friendly. Its only use is to be used as input for a LOAD FORMAT UNLOAD utility, and it must be loaded back into the same table it was unloaded from.

A new option, REORG UNLOAD EXTERNAL, provides the required capability to unload data in an external format. Like DSNTIAUL, this function also generates standard LOAD utility statements as part of the process. The unloaded data can be loaded into another table.

The UNLOAD utility is a new member of the DB2 Utilities Suite that was introduced in V7. The UNLOAD utility unloads the data from one or more source objects to one or more sequential data sets in external format. The source objects can be DB2 table spaces or DB2 image copy data sets.

14.3.2 REORG UNLOAD EXTERNAL

REORG UNLOAD EXTERNAL also uses the FROM TABLE ... WHEN clause (just like REORG DISCARD) to determine which rows are to be unloaded.

Example 14-2 show a REORG UNLOAD EXTERNAL utility statement on table space TS246304 to unload the employees from the table SC246300.TBEMPLOYEE working in department 'A01'.

The selection criteria that can be used to unload rows using REORG UNLOAD EXTERNAL are identical those that are available for REORG DISCARD.

As with REORG DISCARD, REORG UNLOAD EXTERNAL does not provide you with formatting options for data that is unloaded. This can be problematic for numeric data when the information needs to be transported to another platform. Numeric data is unloaded in a host based format (binary and packed decimal). Using the UNLOAD utility can provide a solution here.

Example 14-2 REORG UNLOAD EXTERNAL

```
REORG TABLESPACE DB246300.TS246304
  UNLOAD EXTERNAL
  FROM TABLE SC246300.TBEMPLOYEE
  WHEN (WORKDEPT = 'A01')
```

Important: If there are multiple tables in the table space, those not subject to the WHEN clause are unloaded in their entirety.

14.3.3 UNLOAD

The UNLOAD utility can unload data from one or more source objects to one or more BSAM sequential data sets in external format. The source objects can be DB2 table spaces or DB2 image copy data sets. The UNLOAD utility does not use indexes to access the source table(s). The utility scans the table space or partition(s) directly.

In addition to the functions that are also supported by REORG UNLOAD EXTERNAL, the UNLOAD utility also supports the ability to:

- ▶ Unload data from an image copy data set(s), including full, incremental, DSN1COPY and inline copies.
- ▶ Select columns (specifying an order of the fields in the output record).
- ▶ Sample and limit the number of rows unloaded (by table).
- ▶ Specify the start position, length and data type of output fields.
- ▶ Format output fields.
- ▶ Translate output character-type data to EBCDIC, ASCII or UNICODE.
- ▶ Specify SHRLEVEL and ISOLATION level.
- ▶ Unload table space partitions in parallel.

14.3.4 UNLOAD implementation

Figure 14-3 gives an example of an UNLOAD statement and shows some of the possibilities of the UNLOAD utility.

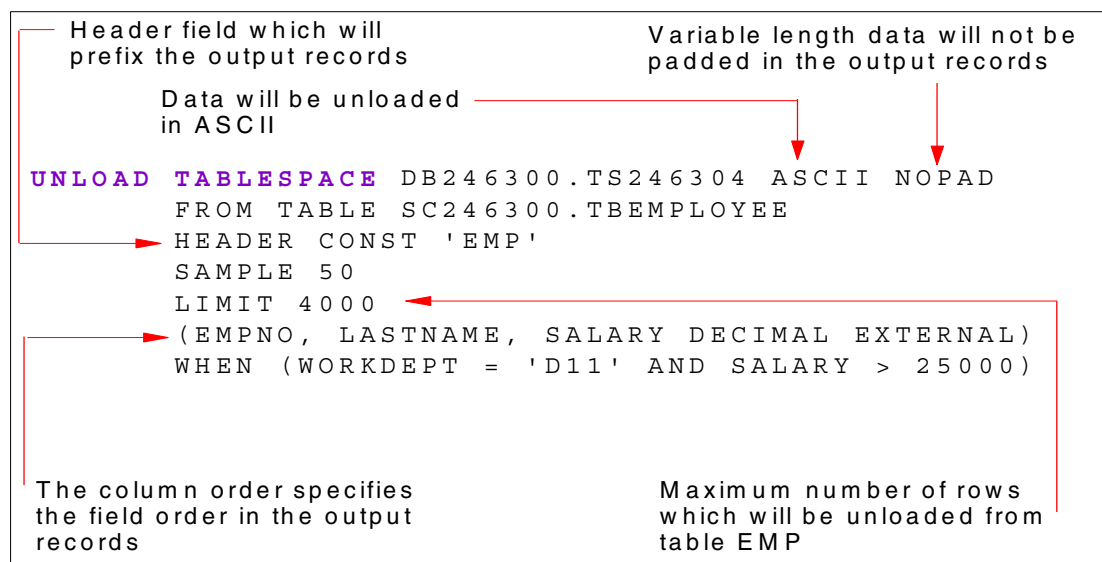


Figure 14-3 Sample UNLOAD utility statement

The figure shows some of the options that can be used by the UNLOAD utility. Here we use the HEADER keyword to give a more meaningful identifier to an output record (instead of the OBID). The data are unloaded in ASCII format and variable length data are not padded. We are not unloading the entire table, but only unloading every 50th row (that qualifies the WHEN conditions) up to a maximum of 4000 rows. We only unload the columns specified in the order specified and the SALARY column is unloaded in 'readable' format instead of 'normal' packed decimal.

For more detailed information on how to use the UNLOAD utility refer to:

- ▶ *DB2 UDB for OS/390 and z/OS Version 7 Utility Guide and Reference*, SC26-9945
- ▶ *DB2 for z/OS and OS/390 Version 7: Using the Utilities Suite*, SG24-6289

14.3.5 UNLOAD restrictions

The UNLOAD utility does not support:

- ▶ Unloading DB2 directory table spaces (DSNDB01) and DSNDB07. However, you can use the UNLOAD utility to UNLOAD your catalog tables (in DSNDB06).
- ▶ *LOB table spaces* cannot be directly unloaded. However, you can unload LOB columns by implicitly or explicitly selecting them from the *base table* instead of the LOB table space. Then the LOB data is expanded (materialized) in the output records. (However, remember that the output is a sequential file with a maximum record length of 32K).
- ▶ The UNLOAD utility does not support the selection of LOB columns from an *image copy data set*. When unloading from an image copy data set, unloading rows from a table containing LOB columns is only supported if the LOB columns are not included in the column list of the FROM TABLE specification.
- ▶ Unloading from index spaces.
- ▶ The UNLOAD utility does not support the specification of a view name in the FROM TABLE clause.
- ▶ Concurrent image copies cannot be used as input to the UNLOAD utility.
- ▶ The maximum length of an output record is limited to 32KB. This limit includes the record header field, the NULL indicator bytes, the length fields for variable length fields, the padding if applied to variable length fields, and the gaps, if any, between output fields.
- ▶ Specify an ORDER BY in the FROM TABLE specification.
- ▶ Specify a delimiter for the output fields (for example, a comma); However, you can add a delimiter to the output file by including a *dummy-field-name* CONSTANT ',' clause after every column in the selection list. But even with this 'trick' you still cannot create something like a delimited ASCII file that you can import into your spreadsheet. VARCHAR columns still have their length field and converting them to CHAR fields results in spaces that you might not want (and the STRIP keyword only works on variable length columns).
- ▶ UNLOAD does not allow you to unload a set of RI related rows. Unload from multiple tables does not choose RI related rows. For example, sampling the orders while unloading, does not unload the line items belonging to the sampled orders, even when the tables are unloaded at the same time.

14.3.6 UNLOAD highlights

Performance wise it is recommended to use the DB2 UNLOAD utility. It performs better than REORG UNLOAD EXTERNAL and several times better than DSNTIAUL.

On top, it has a lot of additional capabilities that have been described earlier in this section like field output selection, ordering of the columns in the output, position, data type, length and format specification, sampling and limiting the number of rows, unloading table space partitions in parallel, and so on.

Another major advantage of UNLOAD is the possibility to unload data from an image copy data set, avoiding access to the base table and interfering with other processes. Even though the UNLOAD utility can run with an isolation level of UR, it still access data through the buffer pool which might still cause response time degradation for other processes when pages get pushed out of the buffer pool by the full scan of the UNLOAD utility.

Note: It is possible to unload from an image copy that is no longer present in SYSIBM.SYSCOPY. However, the table space and table from which you want to unload still has to exist in the DB2 catalog.

If it is necessary to access the base table to unload the latest version of the table, you could use the SHRLEVEL and ISOLATION clauses as you do for other utilities. Otherwise unload from an image copy.

Because the DB2 UNLOAD utility uses the DB2 buffer pools to retrieve the data, no QUIESCE WRITE YES is required before starting the unload process.

14.3.7 UNLOAD pitfalls

If you unload from an inline COPY, it is possible that you get duplicate rows from the duplicate pages which may be present in the inline COPY data set. However, you should receive a warning message that multiple pages with the same page number have been read.

If there are multiple tables in the table space, those not subject to the WHEN clause are unloaded in their entirety.

14.3.8 Comparing DSNTIAUL, REORG UNLOAD EXTERNAL and UNLOAD

Since the DSNTIAUL sample unload program has been around for ages, we are not going into the details of its functionality. If you want to read more about this sample program, see Appendix D in the *DB2 UDB for OS/390 and z/OS Version 7 Utility Guide and Reference*, SC26-9945.

The following table, Table 14-1, gives an overview of the major functions and differences between the three alternatives. Since DSNTIAUL is a normal application program using normal SQL requests, most of the comparison criteria also apply to home grown application programs that might have been developed in your installation.

Table 14-1 Comparing different means to unload data

	DSNTIAUL	REORG UNLOAD EXTERNAL	UNLOAD
Unloading rows through full SQL SELECTs (including joins from multiple tables)	Yes	No	No
Unloading rows in a specific order	Yes	No	No
Performance	Slow	Fast	Even faster
Unloading from image copies	No	No	Yes
Unavailability of data while running (1)	UR	Complete	UR
Formatting possibilities (2)	Limited	None	More
Parallelism (3)	Optimizer	No	Part level
Can connect to remote systems to unload data	Yes	No	No
Support for TEMPLATE and LISTDEF	No	Yes	Yes
Unload against the catalog (4)	Yes	No	Yes
Restart capabilities	No	Yes	Yes
Unloading RI related data	No	No	No
<p>Notes:</p> <p>(1) You can change DSNTIAUL to run with ISOLATION level UR or add the WITH UR clause to your unload SQL statement. REORG UNLOAD EXTERNAL can only use SHRLEVEL NONE. UNLOAD can use ISOLATION CS or UR (SHRLEVEL REFERENCE) or SHRLEVEL CHANGE.</p> <p>(2) Although DSNTIAUL has limited formatting capabilities (only what you can do in an SQL statement), if you write your own program you have of course full control. Even though the UNLOAD utility has a lot more formatting options than REORG UNLOAD EXTERNAL, there is some room for improvement, like standard support for common PC formats.</p> <p>(3) When executing plain SQL statements, the optimizer decides whether or not to use parallelism for a certain query (if the plan is bound with DEGREE(ANY)).</p> <p>(4) Although you can unload data from the catalog, some catalog tables that contain LOB columns might pose a problem, mostly because of the 32K record length restriction on output files.</p>			

If you are looking for a fast way to unload data either directly from the table space or from an image copy and your requirements for the format in which the data needs to be unloaded are not too stringent, the UNLOAD utility is an excellent choice.

Once you migrated to Version 7 there is probably no reason to continue using REORG UNLOAD EXTERNAL. The UNLOAD utility is a complete functional replacement (even adding a lot of extra capabilities) and its performance is slightly better.

Home grown unload applications can still be useful in some cases, for instance when you have very specific output format requirements or when you need to unload data that is somehow related (like a parent record with all its dependent rows) in a single unload operation.

Normal application programs that unload data can also perform quite well depending on the circumstances. When unloading an entire table space, DB2 utilities easily outperform home grown programs. However, when you are unloading a selective number of rows and a good index can be used to retrieve the data, normal SQL applications can outperform the UNLOAD utility that always scans the entire table space or partition.

14.4 Using SQL statements in the utility input stream

DB2 V7 gives you the ability to execute dynamic SQL statements within the utility input stream. This is done via the new EXEC SQL utility control statement.

14.4.1 EXEC SQL utility control statement

EXEC SQL is a new utility statement placed anywhere in the utility input stream. It can be used for two purposes:

- ▶ Executing a non-select dynamic SQL statement before, between or after the actual utility statements.
- ▶ Declaring a cursor with a SQL select statement for use with the LOAD utility (Cross Loader). The declare cursor produces a result table.

The EXEC SQL statement requires no extra privileges other than the ones required by the SQL statements itself.

Executing a non-select dynamic SQL statement

Executing a non-select dynamic SQL statement is done by putting it between the EXEC SQL and ENDEXEC keywords in the utility input stream:

```
EXEC SQL non-select dynamic SQL statement ENDEXEC
```

You can only put one SQL statement between the EXEC SQL and ENDEXEC keywords. The SQL statement can be any dynamic SQL statement that can be used as input for the EXECUTE IMMEDIATE statement, as listed in Example 14-3.

Example 14-3 List of dynamic SQL statements

```
CREATE,ALTER,DROP a DB2 object  
RENAME a DB2 table  
COMMENT ON,LABEL ON a DB2 table, view, or column  
GRANT,REVOKE a DB2 authority
```

```
DELETE,INSERT,UPDATE SQL operation  
LOCK TABLE operation
```

```
EXPLAIN a SQL statement
```

```
SET CURRENT register
```

```
COMMIT,ROLLBACK operation
```

In Example 14-4 we create a new table in the default database DSNDB04 with the same layout as SYSIBM.SYSTABLES.

Example 14-4 Create a new table with the same layout as SYSIBM.SYSTABLES

```
EXEC SQL  
CREATE TABLE PAOLR3.SYSTABLES LIKE SYSIBM.SYSTABLES
```

In the same way, we are able to create indexes on this table, create views on it, and so on. All this is done in the utility input stream.

14.4.2 Possible usage of the EXEC SQL utility statement

The primary use of the EXEC SQL utility statement is meant for declaring cursors for use with the LOAD utility (Cross Loader). But it can also be used to execute any non-select dynamic SQL statement before, between or after regular utility statements. Examples are:

- ▶ DDL creation of the target table for the LOAD utility (and its related objects like database, table space, indexes, views)
- ▶ DDL creation of the mapping table and index before a REORG table space SHRLEVEL CHANGE
- ▶ Dropping of the mapping table after a successful REORG table space SHRLEVEL CHANGE
- ▶ DDL alter of space related values like PRIQTY, SECQTY, FREEPAGE and PCTFREE values before a REORG or LOAD utility
- ▶ DDL alter of INDEX partitions before REORG (for partitioning key changes)
- ▶ GRANT statements (for example: grant select authorities after successful LOAD)
- ▶ SQL delete of “old” records before LOAD RESUME YES
- ▶ SQL update of an application related control table or SQL insert into an application related control table after successful LOAD (with the current timestamp)

EXEC SQL eliminates additional job steps in a job to execute dynamic SQL statements before, between or after the regular utility steps. It can simplify the JCL coding by eliminating these dynamic SQL applications, like DSNTIAD or DSNTPE2 from the JCL stream, and it enables to merge different utility steps, separated by dynamic SQL applications, into one single utility step. But be aware of the restrictions imposed by the EXEC SQL statement.

Benefits of EXEC SQL:

- ▶ You can execute any non-select dynamically preparable SQL statement within the utility input stream.
- ▶ You can declare cursors for use with the LOAD utility, including joins, unions, conversions, aggregations, and remote DRDA access.
- ▶ Successfully executed SQL statements are skipped during restart of the utility.
- ▶ In many cases, the need for extra dynamic SQL programs in the utility job stream is eliminated.
- ▶ Considerable simplification of JCL is possible.

Restrictions of EXEC SQL:

- ▶ There are no select statements.
- ▶ There is no control after error: the whole utility step stops after the first SQL error.
- ▶ There is no concept of unit-of-work consisting of multiple SQL statements.
- ▶ There are no comments possible between SQL statements.

For a more detailed description, see *DB2 for z/OS and OS/390 Version 7: Using the Utilities Suite*, SG24-6289.



Part 5

Appendixes



DDL of the DB2 objects used in the examples

This appendix provides the DDL to create the following DB2 objects used on the examples. This information is also provided on the Internet in downloadable format. For instructions, see “Additional material” on page 251.

- ▶ E/R-diagram of the tables used at examples
- ▶ JCL for the SC246300 schema definition
- ▶ Creation of a database, table spaces, UDTs and UDFs
- ▶ Creation of tables used in the examples
- ▶ Creation of triggers
- ▶ Sample table content
- ▶ DDL to clean up the examples environment

E/R-diagram of the tables used by the examples

Figure A-1 shows graphically the relations of most of the tables that are used in the examples.

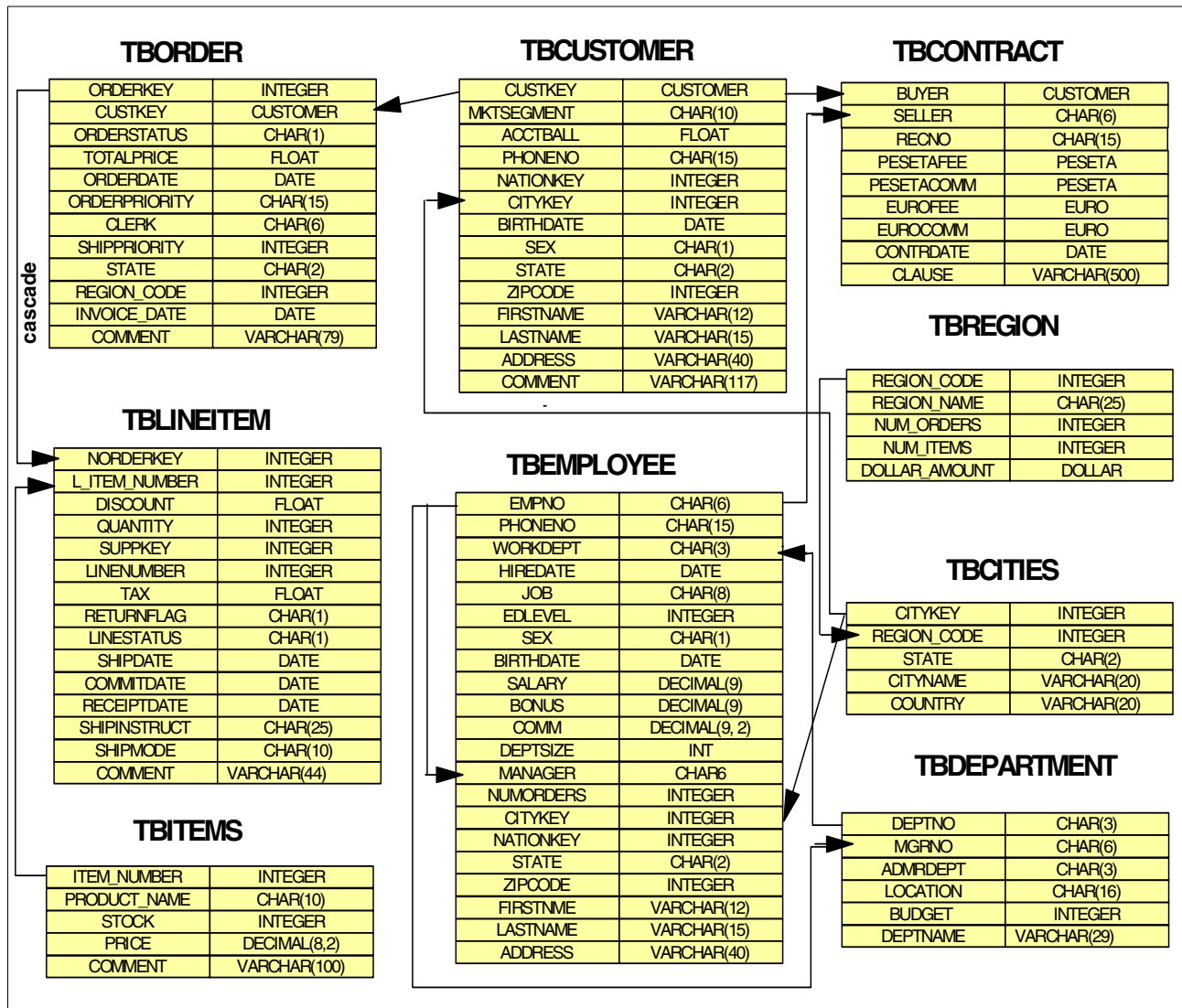


Figure A-1 Relations of tables used in the examples

JCL for the SC246300 schema definition

Example A-1 shows an example of a JCL to create the schema SC246300 using the schema processor (DSNHSP). A sample JCL to create schema is provided in member DSNTJ1S of the SDSNSAMP library.

Example: A-1 Schema creation

```
//BART2A JOB (999,P0K),REGION=5M,MSGCLASS=X,CLASS=A,
//      MSGLEVEL=(1,1),NOTIFY=&SYSUID
//*****
/** NAME = DSNTJ1S *
/** *
/** DESCRIPTIVE NAME = DB2 SAMPLE APPLICATION *
```

```

/**
/**      STATUS = VERSION 7
/**
/**      FUNCTION = THIS JCL BINDS AND RUNS THE SCHEMA PROCESSOR.
/**
/*******
//JOB LIB DD DSN=DB2G7.SDSNLOAD,DISP=SHR
/**
/**
/**      STEP 1 : BIND AND RUN PROGRAM DSNHSP
//PH01SS01 EXEC PGM=IKJEFT01,DYNAMNBR=20
//DBRMLIB DD DSN=DB2G7.SDSNDBRM,
//          DISP=SHR
//SYSTSPRT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSTSIN DD *
DSN SYSTEM(DB2G)
BIND PLAN(DSNHSP71) -
      MEMBER(DSNHSPMN) ACTION(REP) ISO(CS) ENCODING(EBCDIC)
RUN PROGRAM(DSNHSP) PLAN(DSNHSP71) -
      LIB('DB2G7.SDSNLOAD')
END
/**
//SYSIN DD *

CREATE SCHEMA AUTHORIZATION SC246300

CREATE INDEX ....
CREATE DISTINCT TYPE ....
CREATE TABLE ....
CREATE TABLESPACE ....
GRANT ....
CREATE TABLE ....
CREATE INDEX ....
CREATE DISTINCT TYPE ....
GRANT ....
CREATE FUNCTION ....
CREATE DATABASE ...
CREATE UNIQUE INDEX ....
CREATE INDEX ....
CREATE TRIGGER ....

GRANT ALTERIN SC236300 TO PUBLIC
GRANT CREATEIN SC236300 TO PUBLIC
GRANT DROPIN SC236300 TO PUBLIC
/**

```

Creation of a database, table spaces, UDTs and UDFs

Example A-2 and Example A-3 show the DDL to create the stogroup, database, table spaces, UDTs and UDFs that are used in the examples throughout the book.

Example: A-2 DDL for the stogroup, database, table space creation

```
-----  
-- DDL TO CREATE THE DATABASE AND TS (OPTIONALLY STOGROUP)  
-- OBJECTS ARE VERY SMALL (SO WE CAN AFFORD TO USE DEFAULT SPACE )  
-----  
-- CREATE STOGROUP SG246300 VOLUMES(SBOX24) VCAT DB2V710G#  
--  
CREATE DATABASE DB246300  
    BUFFERPOOL BP1  
    INDEXBP BP2  
    STOGROUP SG246300  
    CCSID EBCDIC #  
--  
CREATE TABLESPACE TS246300  
    IN DB246300#  
--  
CREATE TABLESPACE TS246301  
    IN DB246300#  
--  
CREATE TABLESPACE TS246302  
    IN DB246300#  
--  
CREATE TABLESPACE TS246303  
    IN DB246300#  
--  
CREATE TABLESPACE TS246304  
    IN DB246300#  
--  
CREATE TABLESPACE TS246305  
    IN DB246300#  
--  
CREATE TABLESPACE TS246306  
    IN DB246300#  
--  
CREATE TABLESPACE TS246307  
    IN DB246300#  
--  
CREATE TABLESPACE TS246308  
    IN DB246300#  
--  
CREATE TABLESPACE TS246309  
    IN DB246300#  
--  
CREATE TABLESPACE TS246310  
    IN DB246300#  
--  
CREATE TABLESPACE TS246311  
    IN DB246300#  
--  
CREATE TABLESPACE TS246331  
    IN DB246300#  
--  
CREATE TABLESPACE TS246332  
    IN DB246300#  
--  
CREATE TABLESPACE TS246333  
    IN DB246300#  
--  
CREATE TABLESPACE TSLITERA
```



```

                IN DB246300#
--
CREATE LOB TABLESPACE TSLOB1
                IN DB246300#
--
CREATE LOB TABLESPACE TSLOB2
                IN DB246300#
--
CREATE TABLESPACE TS246399
                IN DB246300
                NUMPARTS 4 #

```

Example: A-3 DDL for UDT and UDF creation

```

--
CREATE DISTINCT TYPE SC246300.DOLLAR
AS DECIMAL(17,2)
WITH COMPARISONS#
--
CREATE DISTINCT TYPE SC246300.PESETA
AS DECIMAL(18,0)
WITH COMPARISONS #
--
CREATE DISTINCT TYPE SC246300.EURO
AS DECIMAL(17,2)
WITH COMPARISONS#
--
CREATE DISTINCT TYPE SC246300.CUSTOMER
AS CHAR(11)
WITH COMPARISONS#
--
GRANT USAGE ON DISTINCT TYPE SC246300.EURO TO PUBLIC#

GRANT EXECUTE ON FUNCTION SC246300.EURO(DECIMAL) TO PUBLIC#

GRANT EXECUTE ON FUNCTION SC246300.DECIMAL(EURO) TO PUBLIC#
--
CREATE FUNCTION SC246300."+" (EURO,EURO)
    RETURNS EURO
    SOURCE SYSIBM."+" (DECIMAL(17,2),DECIMAL(17,2)) #

GRANT EXECUTE ON FUNCTION SC246300."+"(EURO,EURO) TO PUBLIC#
--
CREATE FUNCTION SC246300."+" (PESETA,PESETA)
    RETURNS PESETA
    SOURCE SYSIBM."+" (DECIMAL(18,0),DECIMAL(18,0)) #

GRANT EXECUTE ON FUNCTION SC246300."+"(PESETA,PESETA) TO PUBLIC#
--
SET CURRENT PATH = 'SC246300' #
--
CREATE FUNCTION SC246300.PES2EUR (X DECIMAL)
    RETURNS DECIMAL
    LANGUAGE SQL
    CONTAINS SQL
    NO EXTERNAL ACTION
    NOT DETERMINISTIC

```

```

        RETURN X/166      #
--
CREATE FUNCTION SC246300.EUR2PES (X DECIMAL)
  RETURNS DECIMAL
  LANGUAGE SQL
  CONTAINS SQL
  NO EXTERNAL ACTION
  NOT DETERMINISTIC
  RETURN X*166      #
--
CREATE FUNCTION SC246300.SUM(PESETA)
  RETURNS PESETA
  SOURCE SYSIBM.SUM(DECIMAL(18,0)) #
--
CREATE FUNCTION SC246300.SUM(EURO)
  RETURNS EURO
  SOURCE SYSIBM.SUM(DECIMAL(17,2)) #
--
CREATE FUNCTION SC246300.AVG(EURO)
  RETURNS SC246300.EURO
  SOURCE SYSIBM.SUM(DECIMAL(17,2)) #
--
CREATE FUNCTION SC246300.AVG(PESETA)
  RETURNS SC246300.PESETA
  SOURCE SYSIBM.AVG(DECIMAL(18,0)) #
--
-- JUST DUMMY DEFINITION TO MAKE SAMPLES WORK
--
CREATE FUNCTION SC246300.LARGE_ORDER_ALERT
  (CUSTKEY CUSTOMER, TOTALPRICE FLOAT, ORDERDATE DATE)
  RETURNS CHAR(2)
  LANGUAGE SQL
  CONTAINS SQL
  NO EXTERNAL ACTION
  NOT DETERMINISTIC
  RETURN 'OK'      #

```

Creation of tables used in the examples

Example A-4 shows the DDL to create the tables used in the examples.

Example: A-4 DDL for the table creation

```

SET CURRENT PATH = 'SC246300' #
--CURRENT PATH IS NEEDED TO FIND THE USER DEFINED DATA TYPES
--
CREATE TABLE SC246300.TBITEMS
(
  ITEM_NUMBER INTEGER NOT NULL,
  PRODUCT_NAME CHAR( 10 ) NOT NULL WITH DEFAULT,
  STOCK INTEGER NOT NULL WITH DEFAULT,
  PRICE DECIMAL ( 8, 2 ),
  COMMENT VARCHAR ( 100 ) WITH DEFAULT 'NONE',
  PRIMARY KEY (ITEM_NUMBER)
)
IN DB246300.TS246300

```

```

WITH RESTRICT ON DROP#
--
CREATE UNIQUE INDEX SC246300.ITEMIX
ON SC246300.TBITEMS (ITEM_NUMBER)#
--
CREATE TABLE SC246300.TBCITIES
(
CITYKEY INTEGER NOT NULL ,
REGION_CODE INTEGER ,
STATE CHAR ( 2 ) NOT NULL WITH DEFAULT ,
CITYNAME VARCHAR( 20 ) NOT NULL WITH DEFAULT,
COUNTRY VARCHAR( 20 ) NOT NULL WITH DEFAULT,
PRIMARY KEY (CITYKEY)
)
IN DB246300.TS246309
WITH RESTRICT ON DROP#
--
CREATE UNIQUE INDEX SC246300.CITYIX
ON SC246300.TBCITIES (CITYKEY)#
--
CREATE TABLE SC246300.TBCUSTOMER
(
CUSTKEY CUSTOMER NOT NULL,
MKTSEGMENT CHAR( 10 ) NOT NULL WITH DEFAULT,
ACCTBAL FLOAT,
PHONENO CHAR( 15 ) NOT NULL WITH DEFAULT,
NATIONKEY INTEGER NOT NULL WITH DEFAULT,
CITYKEY INTEGER ,
BIRTHDATE DATE,
SEX CHAR( 1 ) NOT NULL,
STATE CHAR ( 2 ) NOT NULL WITH DEFAULT ,
ZIPCODE INTEGER NOT NULL WITH DEFAULT ,
FIRSTNAME VARCHAR( 12 ) NOT NULL WITH DEFAULT,
LASTNAME VARCHAR( 15 ) NOT NULL WITH DEFAULT,
ADDRESS VARCHAR ( 40 ) NOT NULL WITH DEFAULT,
COMMENT VARCHAR( 117 ),
PRIMARY KEY (CUSTKEY),
CONSTRAINT SEXCUST CHECK (SEX IN ('M','F')),
FOREIGN KEY (CITYKEY) REFERENCES SC246300.TBCITIES
)
IN DB246300.TS246301
WITH RESTRICT ON DROP#
--
CREATE UNIQUE INDEX SC246300.CUSTIX
ON SC246300.TBCUSTOMER (CUSTKEY)#
--
CREATE TABLE SC246300.TBCUSTOMER_ARCH
(
CUSTKEY CUSTOMER NOT NULL,
MKTSEGMENT CHAR( 10 ) NOT NULL WITH DEFAULT,
ACCTBAL FLOAT,
PHONENO CHAR( 15 ) NOT NULL WITH DEFAULT,
NATIONKEY INTEGER NOT NULL WITH DEFAULT,
CITYKEY INTEGER ,
BIRTHDATE DATE,
SEX CHAR( 1 ) NOT NULL,
STATE CHAR ( 2 ) NOT NULL WITH DEFAULT ,
ZIPCODE INTEGER NOT NULL WITH DEFAULT ,
FIRSTNAME VARCHAR( 12 ) NOT NULL WITH DEFAULT,
LASTNAME VARCHAR( 15 ) NOT NULL WITH DEFAULT,

```

```

ADDRESS VARCHAR ( 40 ) NOT NULL WITH DEFAULT,
COMMENT VARCHAR( 117 ),
PRIMARY KEY (CUSTKEY),
CONSTRAINT SEXCUST CHECK (SEX IN ('M','F'))
)
IN DB246300.TS246311
WITH RESTRICT ON DROP#
--
CREATE UNIQUE INDEX SC246300.CUST_ARCHIX
ON SC246300.TBCUSTOMER_ARCH (CUSTKEY)#
--
CREATE TABLE SC246300.TBREGION
(
REGION_CODE INTEGER NOT NULL,
REGION_NAME CHAR ( 25 ) NOT NULL ,
NUM_ORDERS INTEGER ,
NUM_ITEMS INTEGER ,
DOLLAR_AMOUNT DOLLAR NOT NULL WITH DEFAULT,
PRIMARY KEY (REGION_CODE)
)
IN DB246300.TS246306
WITH RESTRICT ON DROP#
--
CREATE UNIQUE INDEX SC246300.REGX
ON SC246300.TBREGION (REGION_CODE)#
--
CREATE TABLE SC246300.TBORDER
(
ORDERKEY INTEGER NOT NULL ,
CUSTKEY CUSTOMER NOT NULL ,
ORDERSTATUS CHAR ( 1 ) NOT NULL WITH DEFAULT,
TOTALPRICE FLOAT NOT NULL ,
ORDERDATE DATE NOT NULL WITH DEFAULT,
ORDERPRIORITY CHAR (15 ) NOT NULL WITH DEFAULT,
CLERK CHAR ( 6 ) NOT NULL WITH DEFAULT,
SHIPPRIORITY INTEGER NOT NULL WITH DEFAULT,
STATE CHAR ( 2 ) NOT NULL WITH DEFAULT,
REGION_CODE INTEGER,
INVOICE_DATE DATE NOT NULL WITH DEFAULT,
COMMENT VARCHAR ( 79 ),
PRIMARY KEY (ORDERKEY),
FOREIGN KEY (CUSTKEY) REFERENCES SC246300.TBCUSTOMER,
FOREIGN KEY (REGION_CODE) REFERENCES SC246300.TBREGION
)
IN DB246300.TS246303
WITH RESTRICT ON DROP#
--
CREATE UNIQUE INDEX SC246300.ORDERIX
ON SC246300.TBORDER(ORDERKEY)#
--
CREATE TABLE SC246300.TBLINEITEM
(
NORDERKEY INTEGER NOT NULL ,
L_ITEM_NUMBER INTEGER NOT NULL ,
DISCOUNT FLOAT NOT NULL WITH DEFAULT,
QUANTITY INTEGER NOT NULL ,
SUPPKEY INTEGER NOT NULL WITH DEFAULT,
LINENUMBER INTEGER NOT NULL ,
TAX FLOAT NOT NULL WITH DEFAULT,
RETURNFLAG CHAR ( 1 ) NOT NULL WITH DEFAULT,

```

```

LINESTATUS CHAR( 1 ) NOT NULL WITH DEFAULT,
SHIPDATE DATE NOT NULL WITH DEFAULT,
COMMITDATE DATE NOT NULL WITH DEFAULT,
RECEIPTDATE DATE NOT NULL WITH DEFAULT,
SHIPINSTRUCT CHAR ( 25 ) NOT NULL WITH DEFAULT,
SHIPMODE CHAR ( 10 ) NOT NULL WITH DEFAULT,
COMMENT VARCHAR ( 44 ),
PRIMARY KEY (NORDERKEY,LINENUMBER),
FOREIGN KEY (NORDERKEY) REFERENCES SC246300.TBORDER
        ON DELETE CASCADE,
FOREIGN KEY (L_ITEM_NUMBER) REFERENCES SC246300.TBITEMS
)
IN DB246300.TS246302
WITH RESTRICT ON DROP#
--
CREATE UNIQUE INDEX SC246300.LINEITEMIX
ON SC246300.TBLINEITEM (NORDERKEY,LINENUMBER)#
--
CREATE TABLE SC246300.TBDEPARTMENT
(
DEPTNO CHAR ( 3 ) NOT NULL ,
MGRNO CHAR ( 6 ) ,
ADMRDEPT CHAR ( 3 ) NOT NULL WITH DEFAULT,
LOCATION CHAR ( 16 ) NOT NULL WITH DEFAULT,
BUDGET INTEGER ,
DEPTNAME VARCHAR ( 29 ) NOT NULL WITH DEFAULT,
PRIMARY KEY (DEPTNO)
)
IN DB246300.TS246305
WITH RESTRICT ON DROP#
--
CREATE UNIQUE INDEX SC246300.DEPTIX
ON SC246300.TBDEPARTMENT (DEPTNO)#
--
CREATE TABLE SC246300.TBEMPLOYEE
(
EMPNO CHAR ( 6 ) NOT NULL ,
PHONENO CHAR ( 15 ) NOT NULL WITH DEFAULT,
WORKDEPT CHAR ( 3 ) ,
HIREDATE DATE NOT NULL WITH DEFAULT,
JOB CHAR ( 8 ) NOT NULL WITH DEFAULT,
EDLEVEL INTEGER NOT NULL WITH DEFAULT,
SEX CHAR ( 1 ) NOT NULL ,
BIRTHDATE DATE ,
SALARY DECIMAL ( 9 ) ,
BONUS DECIMAL ( 9 ) ,
COMM DECIMAL ( 9, 2 ) ,
DEPTSIZE INT NOT NULL WITH DEFAULT,
MANAGER CHAR(6),
NUMORDERS INTEGER ,
CITYKEY INTEGER NOT NULL WITH DEFAULT,
NATIONKEY INTEGER NOT NULL WITH DEFAULT,
STATE CHAR ( 2 ) NOT NULL WITH DEFAULT,
ZIPCODE INTEGER NOT NULL WITH DEFAULT,
FIRSTNAME VARCHAR ( 12 ) NOT NULL WITH DEFAULT,
LASTNAME VARCHAR ( 15 ) NOT NULL WITH DEFAULT,
ADDRESS VARCHAR( 40 ),
CONSTRAINT SEXEMPL CHECK (SEX IN ('M','F')),
PRIMARY KEY (EMPNO),
FOREIGN KEY (WORKDEPT) REFERENCES SC246300.TBDEPARTMENT

```

```

)
IN DB246300.TS246304
WITH RESTRICT ON DROP#
--
CREATE UNIQUE INDEX SC246300.EMPLIX
ON SC246300.TBEMPLOYEE (EMPNO)#
--
CREATE INDEX SC246300.EMPNMEIX
ON SC246300.TBEMPLOYEE (LASTNAME,FIRSTNME)#
--
ALTER TABLE SC246300.TBEMPLOYEE
FOREIGN KEY (MANAGER) REFERENCES SC246300.TBEMPLOYEE
ON DELETE NO ACTION #
--
ALTER TABLE SC246300.TBDEPARTMENT
FOREIGN KEY (MGRNO) REFERENCES SC246300.TBEMPLOYEE
ON DELETE NO ACTION #
--
CREATE TABLE SC246300.TBCONTRACT
(
SELLER CHAR ( 6 ) NOT NULL ,
BUYER CUSTOMER NOT NULL ,
RECNO CHAR(15) NOT NULL ,
PESETAFEE SC246300.PESETA ,
PESETACOMM PESETA ,
EUROFEE SC246300.EURO ,
EUROCOMM EURO ,
CONTRDATE DATE,
CLAUSE VARCHAR(500) NOT NULL WITH DEFAULT,
FOREIGN KEY (BUYER) REFERENCES SC246300.TBCUSTOMER,
FOREIGN KEY (SELLER) REFERENCES SC246300.TBEMPLOYEE
ON DELETE CASCADE
)
IN DB246300.TS246308
WITH RESTRICT ON DROP#
--
CREATE TABLE SC246300.TBSTATE
(
STATE CHAR ( 2 ) NOT NULL ,
NUM_ORDERS INTEGER ,
NUM_ITEMS INTEGER ,
DOLLAR_AMOUNT DOLLAR
)
IN DB246300.TS246307
WITH RESTRICT ON DROP#
--
CREATE TABLE SC246300.INVITATION_CARDS
(
PHONENO CHAR( 15 ) NOT NULL WITH DEFAULT,
STATUS CHAR( 1 ) ,
SEX CHAR( 1 ) NOT NULL WITH DEFAULT,
BIRTHDATE DATE,
CITYKEY INTEGER NOT NULL WITH DEFAULT,
FIRSTNAME VARCHAR( 12 ) NOT NULL WITH DEFAULT,
LASTNAME VARCHAR( 15 ) NOT NULL WITH DEFAULT,
ADDRESS VARCHAR ( 40 )
)
IN DB246300.TS246310 #
--

```

```

CREATE TABLE SC246300.TBORDER_1
(
  ORDERKEY INTEGER NOT NULL ,
  CUSTKEY CUSTOMER NOT NULL ,
  ORDERSTATUS CHAR ( 1 ) NOT NULL WITH DEFAULT,
  TOTALPRICE FLOAT NOT NULL ,
  ORDERDATE DATE NOT NULL WITH DEFAULT,
  ORDERPRIORITY CHAR (15 ) NOT NULL WITH DEFAULT,
  CLERK CHAR ( 6 ) NOT NULL WITH DEFAULT,
  SHIPPRIORITY INTEGER NOT NULL WITH DEFAULT,
  STATE CHAR ( 2 ) NOT NULL WITH DEFAULT,
  REGION_CODE INTEGER,
  INVOICE_DATE DATE NOT NULL WITH DEFAULT,
  COMMENT VARCHAR ( 79 ),
  PRIMARY KEY (ORDERKEY),
  FOREIGN KEY (CUSTKEY) REFERENCES SC246300.TBCUSTOMER,
  FOREIGN KEY (REGION_CODE) REFERENCES SC246300.TBREGION
)
IN DB246300.TS246331
WITH RESTRICT ON DROP #

-- Create unique index on primary key
CREATE UNIQUE INDEX SC246300.X1TBORDER_1
ON SC246300.TBORDER_1(ORDERKEY ASC) #

-- Create indexes on foreign keys
CREATE INDEX SC246300.X2TBORDER_1
ON SC246300.TBORDER_1(CUSTKEY ASC) #

CREATE INDEX SC246300.X3TBORDER_1
ON SC246300.TBORDER_1(REGION_CODE ASC) #

CREATE TABLE SC246300.TBORDER_2
(
  ORDERKEY INTEGER NOT NULL ,
  CUSTKEY CUSTOMER NOT NULL ,
  ORDERSTATUS CHAR ( 1 ) NOT NULL WITH DEFAULT,
  TOTALPRICE FLOAT NOT NULL ,
  ORDERDATE DATE NOT NULL WITH DEFAULT,
  ORDERPRIORITY CHAR (15 ) NOT NULL WITH DEFAULT,
  CLERK CHAR ( 6 ) NOT NULL WITH DEFAULT,
  SHIPPRIORITY INTEGER NOT NULL WITH DEFAULT,
  STATE CHAR ( 2 ) NOT NULL WITH DEFAULT,
  REGION_CODE INTEGER,
  INVOICE_DATE DATE NOT NULL WITH DEFAULT,
  COMMENT VARCHAR ( 79 ),
  PRIMARY KEY (ORDERKEY),
  FOREIGN KEY (CUSTKEY) REFERENCES SC246300.TBCUSTOMER,
  FOREIGN KEY (REGION_CODE) REFERENCES SC246300.TBREGION
)
IN DB246300.TS246332
WITH RESTRICT ON DROP#

CREATE UNIQUE INDEX SC246300.X1TBORDER_2
ON SC246300.TBORDER_2(ORDERKEY ASC) #

CREATE INDEX SC246300.X2TBORDER_2
ON SC246300.TBORDER_2(CUSTKEY ASC) #

CREATE INDEX SC246300.X3TBORDER_2

```

```

ON SC246300.TBORDER_2(REGION_CODE ASC) #

CREATE TABLE SC246300.TBORDER_3
(
  ORDERKEY INTEGER NOT NULL ,
  CUSTKEY CUSTOMER NOT NULL ,
  ORDERSTATUS CHAR ( 1 ) NOT NULL WITH DEFAULT,
  TOTALPRICE FLOAT NOT NULL ,
  ORDERDATE DATE NOT NULL WITH DEFAULT,
  ORDERPRIORITY CHAR (15 ) NOT NULL WITH DEFAULT,
  CLERK CHAR ( 6 ) NOT NULL WITH DEFAULT,
  SHIPPRIORITY INTEGER NOT NULL WITH DEFAULT,
  STATE CHAR ( 2 ) NOT NULL WITH DEFAULT,
  REGION_CODE INTEGER,
  INVOICE_DATE DATE NOT NULL WITH DEFAULT,
  COMMENT VARCHAR ( 79 ),
  PRIMARY KEY (ORDERKEY),
  FOREIGN KEY (CUSTKEY) REFERENCES SC246300.TBCUSTOMER,
  FOREIGN KEY (REGION_CODE) REFERENCES SC246300.TBREGION
)
IN DB246300.TS246332
WITH RESTRICT ON DROP #

```

```

CREATE UNIQUE INDEX SC246300.X1TBORDER_3
ON SC246300.TBORDER_3(ORDERKEY ASC) #

```

```

CREATE INDEX SC246300.X2TBORDER_3
ON SC246300.TBORDER_3(CUSTKEY ASC) #

```

```

CREATE INDEX SC246300.X3TBORDER_3
ON SC246300.TBORDER_3(REGION_CODE ASC) #

```

```

CREATE VIEW SC246300.VWORDER
AS
SELECT *
  FROM SC246300.TBORDER_1
  WHERE ORDERKEY BETWEEN 1 AND 700000000
UNION ALL
SELECT *
  FROM SC246300.TBORDER_2
  WHERE ORDERKEY BETWEEN 700000001 AND 1400000000
UNION ALL
SELECT *
  FROM SC246300.TBORDER_3
  WHERE ORDERKEY BETWEEN 1400000001 AND 2147483647 #

```

```

CREATE VIEW SC246300.VWORDER_1UPD
AS
SELECT *
  FROM SC246300.TBORDER_1
  WHERE ORDERKEY BETWEEN 1 AND 700000000
WITH CHECK OPTION #

```

```

CREATE VIEW SC246300.VWORDER_2UPD
AS
SELECT *
  FROM SC246300.TBORDER_2
  WHERE ORDERKEY BETWEEN 700000001 AND 1400000000
WITH CHECK OPTION #

```



```

CREATE VIEW SC246300.VWORDER_3UPD
AS
  SELECT *
  FROM SC246300.TBORDER_3
  WHERE ORDERKEY BETWEEN 1400000001 AND 2147483647
  WITH CHECK OPTION #

-- ROWID USAGE

CREATE TABLE SC246300.LITERATURE
  ( TITLE      CHAR(25)
  ,IDCOL      ROWID NOT NULL GENERATED ALWAYS
  ,MOVLENGTH  INTEGER
  ,LOBMOVIE   BLOB(2K)
  ,LOBBOOK    CLOB(10K) )
  IN DB246300.TSLITERA #

CREATE AUX TABLE SC246300.LOBMOVIE_ATAB
  IN DB246300.TSLOB1
  STORES SC246300.LITERATURE
  COLUMN LOBMOVIE#

CREATE INDEX DB246300.AXLOB1
  ON SC246300.LOBMOVIE_ATAB #

CREATE AUX TABLE SC246300.LOBBOOK_ATAB
  IN DB246300.TSLOB2
  STORES SC246300.LITERATURE
  COLUMN LOBBOOK#

CREATE INDEX DB246300.AXLOB2
  ON SC246300.LOBBOOK_ATAB #

CREATE TABLE SC246300.LITERATURE_GA
  (TITLE      CHAR(30)
  ,IDCOL      ROWID NOT NULL GENERATED ALWAYS
  ,MOVLENGTH  INTEGER
  )
  IN DB246300.TS246300 #

CREATE TABLE SC246300.LITERATURE_GDEF
  (TITLE      CHAR(30)
  ,IDCOL      ROWID NOT NULL GENERATED BY DEFAULT
  ,MOVLENGTH  INTEGER
  )
  IN DB246300.TS246300 #

CREATE UNIQUE INDEX DD ON SC246300.LITERATURE_GDEF (IDCOL) #
-- This index is required for tables with GENERATED BY DEFAULT ROWID
-- columns in order to guarantee uniqueness. You receive an
-- SQLCODE -540 if the index does not exist.

CREATE TABLE SC246300.LITERATURE_PART
  (TITLE      CHAR(25)
  ,IDCOL      ROWID NOT NULL GENERATED ALWAYS
  ,SEQNO      INTEGER
  ,BOOKTEXT   LONG VARCHAR )
  IN DB246300.TS246399 #

CREATE INDEX SC246300.IDCOLIX_PART

```

```

ON SC246300.LITERATURE_PART (IDCOL)
  CLUSTER (PART 1 VALUES(X'3F'),
    PART 2 VALUES(X'7F'),
    PART 3 VALUES(X'BF'),
    PART 4 VALUES(X'FF')) #

CREATE UNIQUE INDEX SC246300.TITLEIX
ON SC246300.LITERATURE_PART (TITLE,SEQNO) #

CREATE VIEW SC246300.CUSTOMRANDEMPLOYEE
AS
  SELECT
    FIRSTNME AS FIRSTNAME
    ,LASTNAME
    ,PHONENO
    ,BIRTHDATE
    ,SEX
    ,YEAR(Date(Days(Current Date)-Days(BirthDate))) AS AGE
    ,ADDRESS
    ,CITYKEY
  FROM SC246300.TBEMPLOYEE
  UNION ALL
  SELECT
    FIRSTNAME
    ,LASTNAME
    ,PHONENO
    ,BIRTHDATE
    ,SEX
    ,YEAR(Date(Days(Current Date)-Days(BirthDate))) AS AGE
    ,ADDRESS
    ,CITYKEY
  FROM SC246300.TBCUSTOMER #

```

Creation of sample triggers

Example A-5 shows the DDL used for the creation of the triggers used in the examples.

Example: A-5 DDL for triggers

```

----- REMEMBER TO CHANGE THE SQL TERMINATOR TO '#'
--
  SET CURRENT PATH = 'SC246300' #
--
CREATE TRIGGER SC246300.TGSUMORD
  AFTER
  INSERT
  ON SC246300.TBORDER
  FOR EACH ROW
  MODE DB2SQL
  BEGIN ATOMIC
    UPDATE SC246300.TBREGION
      SET NUM_ORDERS = NUM_ORDERS + 1;
    UPDATE SC246300.TBSTATE
      SET NUM_ORDERS = NUM_ORDERS + 1
  END #
--
CREATE TRIGGER SC246300.TGBEFOR7

```

```

NO CASCADE BEFORE INSERT
ON SC246300.TBCUSTOMER
REFERENCING NEW AS N
FOR EACH ROW
MODE DB2SQL
    WHEN (NOT EXISTS (SELECT CITYKEY
                        FROM SC246300.TBCITIES
                        WHERE CITYKEY = N.CITYKEY))
SIGNAL SQLSTATE 'ERR10' ('NOT A VALID CITY')      #
--
CREATE TRIGGER SC246300.BUDG_ADJ
AFTER UPDATE OF SALARY ON SC246300.TBEMPLOYEE
REFERENCING OLD AS OLD_EMP
            NEW AS NEW_EMP
FOR EACH ROW MODE DB2SQL
UPDATE SC246300.TBDEPARTMENT
    SET BUDGET = BUDGET + (NEW_EMP.SALARY - OLD_EMP.SALARY)
    WHERE DEPTNO = NEW_EMP.WORKDEPT                #
--
CREATE TRIGGER SC246300.CHK_SAL
NO CASCADE BEFORE UPDATE OF SALARY ON SC246300.TBEMPLOYEE
REFERENCING OLD AS OLD_EMP
            NEW AS NEW_EMP
FOR EACH ROW MODE DB2SQL
WHEN (NEW_EMP.SALARY > OLD_EMP.SALARY * 1.20)
SIGNAL SQLSTATE '75001' ('INVALID SALARY INCREASE - EXCEEDS 20%')#
--
CREATE TRIGGER SC246300.CHK_HDAT
NO CASCADE BEFORE INSERT ON SC246300.TBEMPLOYEE
REFERENCING NEW AS NEW_EMP
FOR EACH ROW MODE DB2SQL
VALUES ( CASE
        WHEN NEW_EMP.HIREDATE < CURRENT DATE
        THEN RAISE_ERROR('75001','HIREDATE HAS PASSED')
        WHEN NEW_EMP.HIREDATE - CURRENT DATE > 365
        THEN RAISE_ERROR ('85002','HIREDATE TOO FAR IN FUTURE')
        ELSE 0
        END)      #
--
-- SECURITY:  TO PREVENT UPDATING SALARY ACCIDENTALLY
--           WILL PREVENT SAMPLE SQL TO WORK SO NOT IMPLEMENTED HERE
--
-- CREATE TRIGGER SC246300.UPDSALAR
--     NO CASCADE BEFORE
--     UPDATE OF SALARY
--     ON SC246300.TBEMPLOYEE
--     FOR EACH STATEMENT  MODE DB2SQL
--     VALUES (
--     RAISE_ERROR('90001','YOU MUST DROP THE TRIGGER UPDSALAR'))#
--
CREATE TRIGGER SC246300.SEXCNST
NO CASCADE BEFORE
INSERT
ON SC246300.TBEMPLOYEE
REFERENCING NEW AS N
FOR EACH ROW
MODE DB2SQL
WHEN ( N.SEX NOT IN('M','F'))
SIGNAL SQLSTATE 'ERRSX'
    ('SEX MUST BE EITHER M OR F')      #

```

```

--
-- STATIC VERSION -- VALUES HARD CODED; THEREFORE NOT IMPLEMENTED
--
-- CREATE TRIGGER SC246300.ITEMNMBR
--     NO CASCADE BEFORE
--     INSERT
--     ON SC246300.TBLINEITEM
--     REFERENCING NEW AS N
--     FOR EACH ROW
--     MODE DB2SQL
--     WHEN ( N.L_ITEM_NUMBER NOT IN
--           (1, 5, 6,
--            9996,9998,10000) )
--     SIGNAL SQLSTATE 'ERR30'
--     ('ITEM NUMBER DOES NOT EXIST') #
--
-- MORE DYNAMIC VERSION OF PREVIOUS TRIGGER
--
CREATE TRIGGER SC246300.ITEMNMB2
    NO CASCADE
    BEFORE INSERT
    ON SC246300.TBLINEITEM
    REFERENCING NEW AS N
    FOR EACH ROW MODE DB2SQL
    WHEN ( N.L_ITEM_NUMBER NOT IN
          ( SELECT ITEM_NUMBER
            FROM SC246300.TBITEMS
            WHERE N.L_ITEM_NUMBER = ITEM_NUMBER )
          )
    SIGNAL SQLSTATE 'ERR30' ('ITEM NUMBER DOES NOT EXIST') #
--
CREATE TRIGGER SC246300.TGEURMOD
    NO CASCADE BEFORE
    INSERT ON SC246300.TBCONTRACT
    REFERENCING NEW AS N
    FOR EACH ROW MODE DB2SQL
    SET N.EUROFEE = EURO(DECIMAL(N.PESETAFFEE)/166) #
--
CREATE TRIGGER SC246300.LARG_ORD
    AFTER INSERT ON SC246300.TBORDER
    REFERENCING NEW TABLE AS N_TABLE
    FOR EACH STATEMENT MODE DB2SQL
    SELECT LARGE_ORDER_ALERT(CUSTKEY, TOTALPRICE, ORDERDATE)
    FROM N_TABLE WHERE TOTALPRICE > 10000 #

```

Populated tables used in the examples

Example A-6 shows some of the columns of the populated tables we use in the examples.

Example: A-6 Populated tables used in the examples

```

SC246300.TBEMPLOYEE
-----+-----+-----+-----+-----+-----+-----+
FIRSTME      BIRTHDATE  WORKDEPT  SEX  BIRTHDATE  JOB          EDLEVEL
-----+-----+-----+-----+-----+-----+-----+

```

MIRIAM	1968-12-22	A01	F	1968-12-22	DBA	4
JUKKA	1964-06-23	A02	M	1964-06-23	SALESMAN	7
TONI	-----	-----	M	-----		0
EVA	-----	A01	F	-----	SYSADM	0
GLADYS	-----	-----	F	-----		0
ABI	1971-06-12	B01	F	1971-06-12	TEACHER	9

SC246300.TBDEPARTMENT

DEPTNO	MGRNO	ADMDEPT	LOCATION	BUDGET	DEPTNAME
B01	-----	KHI	SAN JOSE	100000	DB2
A01	000001	EKV	MADRID	40000	SALES
C01	-----	DAH	FLORIDA	35000	MVS
A02	000001	ERG	SAN FRANCISCO	32000	MARKETING

SC246300.TBCUSTOMER

CUSTKEY	FIRSTNAME	LASTNAME	SEX	CITYKEY
01	ADELA	SALVADOR	F	1
02	MIRIAM	ANTOLIN	F	2
03	MARK	SMITH	M	3
04	SILVIA	YOUNG	F	3
05	IVAN	KENT	M	3

SC246300.TBCITIES

CITYKEY	REGION_CODE	STATE	CITYNAME	COUNTRY
1	28010		MADRID	SPAIN
2	15		AMSTERDAM	HOLLAND
3	55		SAN FRANCISCO	USA
4	42		NOKIA	FINLAND
5	33076		CORAL SPRINGS	USA
6	97		TOKIO	JAPAN

SC246300.TBCONTRACT

SELLER	BUYER	RECNO	PESETAFEE
000006	02	333	90000.
000001	03	222	10000.
000003	01	111	50000.

SC246300.TBREGION

REGION_CODE	REGION_NAME
28010	PROVINCE OF MADRID
55	SILICON VALLEY
15	THE NETHERLANDS

SC246300.TBORDER

ORDERKEY	CUSTKEY	ORDERSTATUS	ORDERPRIORITY	SHIPRIORITY	REGION_CODE
10	03	0	URGENT	1	5
1	05	M	NORMAL	3	2801
2	03	M	VERY URGENT	0	1

3	05	0	NORMAL	3	1
4	01	M	NORMAL	3	5
5	03	0	URGENT	1	5
6	01	0	NORMAL	2	2801
7	04	0	NORMAL	2	2801
8	02	M	NORMAL	2	5
9	03	0	VERY URGENT	0	1

SC246300.TBLINEITEM

NORDERKEY	LINENUMBER	L_ITEM_NUMBER	QUANTITY	TAX
1	1	100	3	5
1	2	120	1	5
2	1	440	2	10
3	1	660	3	5
4	1	505	4	10
4	2	440	1	10
4	3	660	1	5
5	1	133	1	5
6	1	100	8	5
7	1	120	1	5
8	1	440	1	10
9	1	660	1	5
10	1	440	1	10
10	2	100	1	5

SC246300.TBITEMS

ITEM_NUMBER	PRODUCT_NAME	STOCK	PRICE	COMMENT
100	WIDGET	50	1.25	INCOMPATIBLE WITH HAMMER
120	NUT	50	1.25	FOR TYPE 20 NUT ONLY
133	WASHER	50	1.25	BRONZE
440	HAMMER	50	1.25	NONE
505	NAIL	50	1.25	GALVANIZED
660	SCREW	50	1.25	WOOD

DDL to clean up the environment

To clean up the environment, you can use the DDL in Example A-7.

Example: A-7 DDL to clean up the examples environment

```
-- DROP STOGROUP SG246300;
-- SET CURRENT SQLID = 'SYS1';
--
-- SET CURRENT PATH = 'SC246300'#
--
ALTER TABLE SC246300.TBITEMS DROP RESTRICT ON DROP#
ALTER TABLE SC246300.TBORDER DROP RESTRICT ON DROP#
ALTER TABLE SC246300.TBLINEITEM DROP RESTRICT ON DROP#
ALTER TABLE SC246300.TBCUSTOMER DROP RESTRICT ON DROP#
ALTER TABLE SC246300.TBEMPLOYEE DROP RESTRICT ON DROP#
ALTER TABLE SC246300.TBCONTRACT DROP RESTRICT ON DROP#
ALTER TABLE SC246300.TBREGION DROP RESTRICT ON DROP#
ALTER TABLE SC246300.TBCITIES DROP RESTRICT ON DROP#
```

```
ALTER TABLE SC246300.TBDEPARTMENT DROP RESTRICT ON DROP#
ALTER TABLE SC246300.TBSTATE DROP RESTRICT ON DROP#
ALTER TABLE SC246300.TBORDER_1 DROP RESTRICT ON DROP#
ALTER TABLE SC246300.TBORDER_2 DROP RESTRICT ON DROP#
ALTER TABLE SC246300.TBORDER_3 DROP RESTRICT ON DROP#
--
DROP DATABASE DB246300#
--

DROP FUNCTION SC246300.PES2EUR RESTRICT#
DROP FUNCTION SC246300.EUR2PES RESTRICT#
DROP FUNCTION SC246300.EUR22PES RESTRICT#
DROP FUNCTION SC246300.SUM(PESETA) RESTRICT#
DROP FUNCTION SC246300.SUM(EURO) RESTRICT#
DROP FUNCTION SC246300.LARGE_ORDER_ALERT RESTRICT #
DROP DISTINCT TYPE SC246300.CUSTOMER RESTRICT #
DROP DISTINCT TYPE SC246300.PESETA RESTRICT #
DROP DISTINCT TYPE SC246300.EURO RESTRICT #
DROP DISTINCT TYPE SC246300.DOLLAR RESTRICT #

-- DROP STOGROUP SG246300;
```



B

Sample programs

A selected set of sample programs is shown in this appendix. The rest are available from the additional material on the Internet.

- ▶ Returning SQLSTATE from a stored procedure to a trigger
- ▶ Passing a transition table from a trigger to a SP

Returning SQLSTATE from a stored procedure to a trigger

The full program including the JCL and related DDL is provided in the additional material. See Appendix C, "Additional material" on page 251 for details on how to download the files.

Example: B-1 Returning SQLSTATE to a trigger from a SP

```
IDENTIFICATION DIVISION.
PROGRAM-ID. "SDOBMS3".

*****
* This stored procedure shows how SQLSTATE and      *
* a DIAG string can be returned to a trigger      *
* so it can undo all changes done so far by the  *
* trigger                                          *
*****

DATA DIVISION.
WORKING-STORAGE SECTION.
    EXEC SQL INCLUDE SQLCA END-EXEC.

01 VAR1 PIC X(20).

01 ERROR-MESSAGE.
    02 ERROR-LEN PIC S9(4) COMP VALUE +960.
    02 ERROR-TEXT PIC X(120) OCCURS 8 TIMES
        INDEXED BY ERROR-INDEX.
77 ERROR-TEXT-LEN PIC S9(8) COMP VALUE +120.

77 ERR-CODE PIC 9(8) VALUE 0.
77 ERR-MINUS PIC X VALUE SPACE.
77 LINE-EXEC PIC X(20) VALUE SPACE.

* VARIABLE TO GET FIELDS IN SQLCA FORMATTED PROPERLY
77 XSQLCABC PIC 9(9).
77 XSQLCODE PIC S9(9) SIGN IS LEADING , SEPARATE.
77 XSQLERRML PIC S9(9) SIGN IS LEADING , SEPARATE.
77 XSQLERRD PIC S9(9) SIGN IS LEADING , SEPARATE.

LINKAGE SECTION.
* INPUT PARM PASSED BY STORED PROC
01 PARM1 PIC X(20).
01 INDPARM1 PIC S9(4) COMP.
* DECLARE THE SQLSTATE THAT CAN BE SET BY STORED PROC
01 P-SQLSTATE PIC X(5).
* DECLARE THE QUALIFIED PROCEDURE NAME
01 P-PROC.
    49 P-PROC-LEN PIC 9(4) USAGE BINARY.
    49 P-PROC-TEXT PIC X(27).
* DECLARE THE SPECIFIC PROCEDURE NAME
01 P-SPEC.
    49 P-SPEC-LEN PIC 9(4) USAGE BINARY.
    49 P-SPEC-TEXT PIC X(18).
* DECLARE SQL DIAGNOSTIC MESSAGE TOKEN
01 P-DIAG.
    49 P-DIAG-LEN PIC 9(4) USAGE BINARY.
    49 P-DIAG-TEXT PIC X(70).

PROCEDURE DIVISION USING PARM1, INDPARM1, P-SQLSTATE,
```

P-PROC, P-SPEC, P-DIAG.

```
MOVE PARM1 TO VARI.
* DISPLAY "VARI " VARI.
MOVE "UPD ONE" TO LINE-EXEC.
* This operation will fail because the trigger that invokes
* the SP is a before trigger. Therefore it CANNOT update.
EXEC SQL
    UPDATE SC246300.TBEMPLOYEE
        SET PHONENO=:VARI WHERE EMPNO ='000006'
END-EXEC.
* DISPLAY "AFTER UPDATE ONE" SQLCODE .
IF SQLCODE NOT EQUAL 0 THEN
    PERFORM DBERROR
END-IF.

ERROR-EXIT.
GOBACK.
```

DBERROR.

```
DISPLAY "*** SQLERR FROM SDOBMS3 ***".
MOVE SQLCODE TO ERR-CODE .
IF SQLCODE < 0 THEN MOVE '-' TO ERR-MINUS.
DISPLAY "SQLCODE = " ERR-MINUS ERR-CODE "LINE " LINE-EXEC.
CALL 'DSNTIAR' USING SQLCA ERROR-MESSAGE ERROR-TEXT-LEN.
IF RETURN-CODE = ZERO
    PERFORM ERROR-PRINT VARYING ERROR-INDEX
        FROM 1 BY 1 UNTIL ERROR-INDEX GREATER THAN 8
* TO SHOW WHERE EVERYTHING GOES IN SQLCA
DISPLAY "*** START OF UNFORMATTED SQLCA ***"
DISPLAY "SQLCAID X(8) " SQLCAID
MOVE SQLCABC TO XSQLCABC
DISPLAY "SQLCABC I " XSQLCABC
MOVE SQLCODE TO XSQLCODE
DISPLAY "SQLCODE I " XSQLCODE
MOVE SQLERRML TO XSQLERRML
DISPLAY "SQLERRML SI " XSQLERRML
DISPLAY "SQLERRMC X(70) " SQLERRMC
DISPLAY "SQLERRP X(8) " SQLERRP
MOVE SQLERRD(1) TO XSQLERRD
DISPLAY "SQLERRD1 I " XSQLERRD
MOVE SQLERRD(2) TO XSQLERRD
DISPLAY "SQLERRD2 I " XSQLERRD
MOVE SQLERRD(3) TO XSQLERRD
DISPLAY "SQLERRD3 I " XSQLERRD
MOVE SQLERRD(4) TO XSQLERRD
DISPLAY "SQLERRD4 I " XSQLERRD
MOVE SQLERRD(5) TO XSQLERRD
DISPLAY "SQLERRD5 I " XSQLERRD
MOVE SQLERRD(6) TO XSQLERRD
DISPLAY "SQLERRD6 I " XSQLERRD
DISPLAY "SQLWARN0 X(1) " SQLWARN0
DISPLAY "SQLWARN1 X(1) " SQLWARN1
DISPLAY "SQLWARN2 X(1) " SQLWARN2
DISPLAY "SQLWARN3 X(1) " SQLWARN3
DISPLAY "SQLWARN4 X(1) " SQLWARN4
DISPLAY "SQLWARN5 X(1) " SQLWARN5
DISPLAY "SQLWARN6 X(1) " SQLWARN6
DISPLAY "SQLWARN7 X(1) " SQLWARN7
DISPLAY "SQLWARN8 X(1) " SQLWARN8
```

```

        DISPLAY "SQLWARN9 X(1) " SQLWARN9
        DISPLAY "SQLWARNA X(1) " SQLWARNA
        DISPLAY "SQLSTATE X(5) " SQLSTATE
        DISPLAY "**** END OF UNFORMATTED SQLCA ****"
    ELSE
        DISPLAY RETURN-CODE.
        MOVE '38601' TO P-SQLSTATE .
        MOVE 16      TO P-DIAG-LEN.
        MOVE 'SP HAD SQL ERROR' TO P-DIAG-TEXT.
    ERROR-PRINT.
        DISPLAY ERROR-TEXT (ERROR-INDEX).

```

Passing a transition table from a trigger to a SP

The full program including the JCL and related DDL is provided in the additional material. See Appendix C, "Additional material" on page 251 for details on how to download the files.

Example: B-2 Passing a transition table from a trigger to a SP

IDENTIFICATION DIVISION.	00290001
PROGRAM-ID. "SPTRTT".	00300000
	00310000
*****	00320000
* THIS PROGRAM SHOWS HOW A TRANSITION TABLE	* 00330000
* FROM A TRIGGER CAN BE ACCESSED IN A STORED	* 00340000
* PROCEDURE	* 00350000
* (ALSO USING SQLSTATE TO PASS BACK INFO TO	* 00360000
* THE TRIGGER IN CASE OF PROBLEMS)	* 00361000
*****	00370000
	00380000
DATA DIVISION.	00390000
WORKING-STORAGE SECTION.	00400000
EXEC SQL INCLUDE SQLCA END-EXEC.	00410000
	00420000
* *****	00430000
* 2. DECLARE TABLE LOCATOR HOST VARIABLE TRIG-TBL-ID	00440000
* *****	00450000
01 TRIG-TBL-ID SQL TYPE IS	00460000
TABLE LIKE SC246300.TBEMPLOYEE AS LOCATOR.	00470000
	00480000
* PARMS FETCH ROWS INTO FROM TRANSITION TABLE	00490000
01 EMPNO PIC X(6).	00500000
01 FIRSTNME.	00510000
49 FIRSTNME-LEN PIC S9(4) USAGE COMP.	00520000
49 FIRSTNME-TEXT PIC X(15).	00530000
	00550000
01 ERROR-MESSAGE.	00560000
02 ERROR-LEN PIC S9(4) COMP VALUE +960.	00570000
02 ERROR-TEXT PIC X(120) OCCURS 8 TIMES	00580000
INDEXED BY ERROR-INDEX.	00590000
77 ERROR-TEXT-LEN PIC S9(8) COMP VALUE +120.	00600000
	00610000
77 ERR-CODE PIC 9(8) VALUE 0.	00620000
77 ERR-MINUS PIC X VALUE SPACE.	00630000
77 LINE-EXEC PIC X(20) VALUE SPACE.	00640000
	00650000
* VARIABLE TO GET FIELDS IN SQLCA FORMATTED PROPERLY	00660000

```

77 XSQLCABC      PIC 9(9).                00670000
77 XSQLCODE     PIC S9(9) SIGN IS LEADING , SEPARATE. 00680000
77 XSQLERRML   PIC S9(9) SIGN IS LEADING , SEPARATE. 00690000
77 XSQLERRD    PIC S9(9) SIGN IS LEADING , SEPARATE. 00700000
                                           00710000
77 I           PIC 9(5)   VALUE 0.        00720000
                                           00740000
LINKAGE SECTION.                          00750000
* *****                                00760000
* 1. DECLARE TABLOC AS LARGE INTEGER PARM  00770000
* *****                                00780000
    01 TABLOC PIC S9(9) USAGE BINARY.     00790000
    01 INDTABLOC PIC S9(4) COMP.          00800000
* DECLARE THE SQLSTATE THAT CAN BE SET BY STORED PROC 00810000
    01 P-SQLSTATE PIC X(5).               00820000
* DECLARE THE QUALIFIED PROCEDURE NAME    00830000
    01 P-PROC.                            00840000
        49 P-PROC-LEN PIC 9(4) USAGE BINARY. 00850000
        49 P-PROC-TEXT PIC X(27).          00860000
* DECLARE THE SPECIFIC PROCEDURE NAME    00870000
    01 P-SPEC.                            00880000
        49 P-SPEC-LEN PIC 9(4) USAGE BINARY. 00890000
        49 P-SPEC-TEXT PIC X(18).         00900000
* DECLARE SQL DIAGNOSTIC MESSAGE TOKEN   00910000
    01 P-DIAG.                            00920000
        49 P-DIAG-LEN PIC 9(4) USAGE BINARY. 00930000
        49 P-DIAG-TEXT PIC X(70).        00940000
                                           00950000
                                           00954000
PROCEDURE DIVISION USING TABLOC , INDTABLOC, P-SQLSTATE, 00960000
                    P-PROC, P-SPEC, P-DIAG. 00970000
                                           00971000
* THE INDTABLOC INDICATOR VARIABLE IS IMPORTANT 00980000
* OTHERWISE YOU WON'T BE ABLE TO PASS BACK INFO THROUGH 00990000
* P-SQLSTATE 01000000
                                           01020000
* *****                                01030000
* 4. DECLARE CURSOR USING THE TRANSITION TABLE 01040000
* *****                                01050000
    EXEC SQL                               01060000
        DECLARE C1 CURSOR FOR              01070000
            SELECT EMPNO, FIRSTNME        01080000
            FROM TABLE ( :TRIG-TBL-ID LIKE SC246300.TBEMPLOYEE ) 01090000
        END-EXEC.                         01100000
* *****                                01110000
* 3. COPY TABLE LOCATOR INPUT PARM TO THE TABLE LOCATOR HOST VAR 01120000
* *****                                01130000
    MOVE TABLOC TO TRIG-TBL-ID.          01140000
                                           01150000
    MOVE "OPEN CUR" TO LINE-EXEC.        01160000
    EXEC SQL OPEN C1 END-EXEC.           01170000
* MOVE SQLCODE TO XSQLCODE.             01180000
* DISPLAY " AFTER OPEN SQLCODE " XSQLCODE. 01190000
    IF SQLCODE < 0 THEN                 01200000
        PERFORM DBERROR                  01210000
        PERFORM ERROR-EXIT.             01220000
* *****                                01230000
* 5. PROCESS DATA FROM TRANSITION TABLE 01240000
*                                         01241000
* HERE WE ONLY DISPLAY THE INFO IN THE OUTPUT OF THE SP 01242000

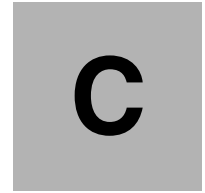
```

```

* ADDRESS SPACE, BUT HERE IS WHERE YOU WOULD DO THE 01243000
* REAL WORK ON THE TRANSITION TABLE 01244000
* 01245000
* ***** 01250000
  DISPLAY " ". 01260000
  DISPLAY " DATA FROM TRANSITION TABLE " 01270000
  DISPLAY " PEOPLE WITH SALARY INCREASE" 01280000
  DISPLAY " ". 01290000
  DISPLAY " ROW EMPNO FIRSTNAME" 01300000
  PERFORM GET-ROWS-E 01310000
  VARYING I FROM 1 BY 1 01320000
  UNTIL 01330000
    SQLCODE = 100. 01340000
  01350000
  MOVE "CLOS CUR" TO LINE-EXEC. 01360000
  EXEC SQL CLOSE C1 END-EXEC. 01370000
  01380000
ERROR-EXIT. 01420000
  GOBACK. 01430000
  01440000
GET-ROWS-E. 01450000
* 01460000
* FETCH ROWS FROM THE TRANSITION TABLE INTO HOST VARIABLES. 01470000
* 01480000
  MOVE "FETCH E " TO LINE-EXEC. 01490000
  EXEC SQL FETCH C1 INTO :EMPNO , 01500000
  :FIRSTNME 01510000
  END-EXEC. 01520000
* DISPLAY INFO TO DEBUG 01530000
* MOVE SQLCODE TO XSQLCODE. 01540000
* DISPLAY " AFTER FETCH SQLCODE " XSQLCODE. 01550000
  IF SQLCODE = 0 THEN 01560000
    01611000
    DISPLAY I ' ' EMPNO ' ' FIRSTNME-TEXT 01620000
  ELSE IF SQLCODE < 0 THEN 01630000
    PERFORM DBERROR 01640000
  END-IF. 01650000
  ADD 1 TO J. 01660000
  01670000
DBERROR. 01680000
  DISPLAY "*** SQLERR FROM SPTRTT ***". 01690000
  MOVE SQLCODE TO ERR-CODE . 01700000
  IF SQLCODE < 0 THEN MOVE '-' TO ERR-MINUS. 01710000
  DISPLAY "SQLCODE = " ERR-MINUS ERR-CODE "LINE " LINE-EXEC. 01720000
  CALL 'DSNTIAR' USING SQLCA ERROR-MESSAGE ERROR-TEXT-LEN. 01730000
  IF RETURN-CODE = ZERO 01740000
    PERFORM ERROR-PRINT VARYING ERROR-INDEX 01750000
    FROM 1 BY 1 UNTIL ERROR-INDEX GREATER THAN 8 01760000
* TO SHOW WHERE EVERYTHING GOES IN SQLCA 01770000
  DISPLAY "*** START OF UNFORMATTED SQLCA ***" 01780000
  DISPLAY "SQLCAID X(8) " SQLCAID 01790000
  MOVE SQLCABC TO XSQLCABC 01800000
  DISPLAY "SQLCABC I " XSQLCABC 01810000
  MOVE SQLCODE TO XSQLCODE 01820000
  DISPLAY "SQLCODE I " XSQLCODE 01830000
  MOVE SQLERRML TO XSQLERRML 01840000
  DISPLAY "SQLERRML SI " XSQLERRML 01850000
  DISPLAY "SQLERRMC X(70) " SQLERRMC 01860000
  DISPLAY "SQLERRP X(8) " SQLERRP 01870000
  MOVE SQLERRD(1) TO XSQLERRD 01880000

```

DISPLAY "SQLERRD1 I " XSQLERRD	01890000
MOVE SQLERRD(2) TO XSQLERRD	01900000
DISPLAY "SQLERRD2 I " XSQLERRD	01910000
MOVE SQLERRD(3) TO XSQLERRD	01920000
DISPLAY "SQLERRD3 I " XSQLERRD	01930000
MOVE SQLERRD(4) TO XSQLERRD	01940000
DISPLAY "SQLERRD4 I " XSQLERRD	01950000
MOVE SQLERRD(5) TO XSQLERRD	01960000
DISPLAY "SQLERRD5 I " XSQLERRD	01970000
MOVE SQLERRD(6) TO XSQLERRD	01980000
DISPLAY "SQLERRD6 I " XSQLERRD	01990000
DISPLAY "SQLWARN0 X(1) " SQLWARN0	02000000
DISPLAY "SQLWARN1 X(1) " SQLWARN1	02010000
DISPLAY "SQLWARN2 X(1) " SQLWARN2	02020000
DISPLAY "SQLWARN3 X(1) " SQLWARN3	02030000
DISPLAY "SQLWARN4 X(1) " SQLWARN4	02040000
DISPLAY "SQLWARN5 X(1) " SQLWARN5	02050000
DISPLAY "SQLWARN6 X(1) " SQLWARN6	02060000
DISPLAY "SQLWARN7 X(1) " SQLWARN7	02070000
DISPLAY "SQLWARN8 X(1) " SQLWARN8	02080000
DISPLAY "SQLWARN9 X(1) " SQLWARN9	02090000
DISPLAY "SQLWARNA X(1) " SQLWARNA	02100000
DISPLAY "SQLSTATE X(5) " SQLSTATE	02110000
DISPLAY "**** END OF UNFORMATTED SQLCA ****"	02120000
ELSE	02130000
DISPLAY RETURN-CODE.	02140000
MOVE '38601' TO P-SQLSTATE .	02150000
MOVE 16 TO P-DIAG-LEN.	02160000
MOVE 'SP HAD SQL ERROR' TO P-DIAG-TEXT.	02170000
ERROR-PRINT.	02180000
DISPLAY ERROR-TEXT (ERROR-INDEX).	02190000



Additional material

This 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 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/SG246300>

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 redbook form number, SG246300.

Using the Web material

The additional Web material that accompanies this redbook includes the following files:

<i>File name</i>	<i>Description</i>
SG246300.zip	Zipped Code Samples, DDL, DML statements

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.

There are two files with the *.bin suffix:

File name after unzipping -	Original data set name on OS/390:
sg246300-JCL.BIN	- SG246300.JCL
sg246300-SQL.BIN	- SG246300.SQL

These two sequential files were created from partitioned data sets using the **TSO TRANSMIT OUTDA()** command. To recreate the partitioned data sets on OS/390 from the downloaded file, you need to:

1. Transfer the files from PC to MVS as binary, with the following attributes for the output data set:

```
DSORG=PS
RECFM=FB
LRECL=80
BLKSIZE=3200
```

2. Use the **TSO RECEIVE INDA()** command to create the partitioned data sets (PDSs) from the sequential data sets you just transferred. You can use the **TSO HELP RECEIVE** command to find out about the optional parameters for the RECEIVE command.

Both PDS data sets have an **\$INDEX** member that explain the content of the individual members and how to use them.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

IBM Redbooks

For information on ordering these publications, see “How to get IBM Redbooks” on page 254.

- ▶ *DB2 for z/OS and OS/390 Version 7: Using the Utilities Suite*, SG24-6289
- ▶ *DB2 for OS/390 and z/OS Powering the World's e-business Solutions*, SG24-6257
- ▶ *DB2 for z/OS and OS/390 Version 7 Performance Topics*, SG24-6129
- ▶ *DB2 UDB Server for OS/390 and z/OS Version 7 Presentation Guide*, SG24-6121
- ▶ *DB2 UDB Server for OS/390 Version 6 Technical Update*, SG24-6108
- ▶ *DB2 Java Stored Procedures Learning by Example*, SG24-5945
- ▶ *DB2 UDB for OS/390 Version 6 Performance Topics*, SG24-5351
- ▶ *DB2 for OS/390 Version 5 Performance Topics*, SG24-2213
- ▶ *DB2 for MVS/ESA Version 4 Non-Data-Sharing Performance Topics*, SG24-4562
- ▶ *DB2 UDB for OS/390 Version 6 Management Tools Package*, SG24-5759
- ▶ *DB2 Server for OS/390 Version 5 Recent Enhancements - Reference Guide*, SG24-5421
- ▶ *DB2 for OS/390 Capacity Planning*, SG24-2244
- ▶ *Cross-Platform DB2 Stored Procedures: Building and Debugging*, SG24-5485
- ▶ *DB2 for OS/390 and Continuous Availability*, SG24-5486
- ▶ *Connecting WebSphere to DB2 UDB Server*, SG24-6219
- ▶ *Parallel Sysplex Configuration: Cookbook*, SG24-2076
- ▶ *DB2 for OS/390 Application Design Guidelines for High Performance*, SG24-2233
- ▶ *Using RVA and SnapShot for BI with OS/390 and DB2*, SG24-5333
- ▶ *IBM Enterprise Storage Server Performance Monitoring and Tuning Guide*, SG24-5656
- ▶ *DFSMS Release 10 Technical Update*, SG24-6120
- ▶ *Storage Management with DB2 for OS/390*, SG24-5462
- ▶ *Implementing ESS Copy Services on S/390*, SG24-5680

Other resources

These publications are also relevant as further information sources:

- ▶ *DB2 UDB for OS/390 and z/OS Version 7 What's New*, GC26-9946
- ▶ *DB2 UDB for OS/390 and z/OS Version 7 Installation Guide*, GC26-9936
- ▶ *DB2 UDB for OS/390 and z/OS Version 7 Command Reference*, SC26-9934
- ▶ *DB2 UDB for OS/390 and z/OS Version 7 Messages and Codes*, GC26-9940
- ▶ *DB2 UDB for OS/390 and z/OS Version 7 Utility Guide and Reference*, SC26-9945

- ▶ *DB2 UDB for OS/390 and z/OS Version 7 Application Programming Guide and Reference for Java*, SC26-9932
- ▶ *DB2 UDB for OS/390 and z/OS Version 7 Administration Guide*, SC26-9931
- ▶ *DB2 UDB for OS/390 and z/OS Version 7 Application Programming and SQL Guide*, SC26-9933
- ▶ *DB2 UDB for OS/390 and z/OS Version 7 Release Planning Guide*, SC26-9943
- ▶ *DB2 UDB for OS/390 and z/OS Version 7 SQL Reference*, SC26-9944
- ▶ *DB2 UDB for OS/390 and z/OS Version 7 Text Extender Administration and Programming*, SC26-9948
- ▶ *DB2 UDB for OS/390 and z/OS Version 7 Data Sharing: Planning and Administration*, SC26-9935
- ▶ *DB2 UDB for OS/390 and z/OS Version 7 Image, Audio, and Video Extenders Administration and Programming*, SC26-9947
- ▶ *DB2 UDB for OS/390 and z/OS Version 7 ODBC Guide and Reference*, SC26-9941
- ▶ *DB2 UDB for OS/390 and z/OS Version 7 XML Extender Administration and Programming*, SC26-9949
- ▶ *OS/390 V2R10.0 DFSMS Using Data Sets*, SC26-7339
- ▶ *DB2 UDB for OS/390 Version 6 SQL Reference*, SC26-9014

Referenced Web sites

These Web sites are also relevant as further information sources:

- ▶ <http://www.ibm.com/software/data/db2/os390/>
DB2 for OS/390
- ▶ <http://www.ibm.com/software/data/db2/os390/estimate/>
DB2 Estimator
- ▶ <http://www.ibm.com/storage/hardsoft/diskdr1s/technology.htm>
Technology for disk storage systems
- ▶ <http://www.ibm.com/software/data/db2/os390/support.html>
DB2 support and services
- ▶ <http://www.ibm.com/software/db2/os390/downloads.html>
DB2 for OS/390 downloads. See DBRM Colon Finder.

How to get IBM Redbooks

Search for additional Redbooks or Redpieces, view, download, or order hardcopy from the Redbooks Web site:

[ibm.com/redbooks](http://www.ibm.com/redbooks)

Also download additional materials (code samples or diskette/CD-ROM images) from this Redbooks site.

Redpieces are Redbooks in progress; not all Redbooks become Redpieces and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

IBM Redbooks collections

Redbooks are also available on CD-ROMs. Click the CD-ROMs button on the Redbooks Web site for information about all the CD-ROMs offered, as well as updates and formats.

Special notices

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Any pointers in this publication to external Web sites are provided for convenience only and do not in any manner serve as an endorsement of these Web sites.

The following terms are trademarks of other companies:

Tivoli, Manage. Anything. Anywhere., The Power To Manage., Anything. Anywhere., TME, NetView, Cross-Site, Tivoli Ready, Tivoli Certified, Planet Tivoli, and Tivoli Enterprise are trademarks or registered trademarks of Tivoli Systems Inc., an IBM company, in the United States, other countries, or both. In Denmark, Tivoli is a trademark licensed from Kjøbenhavns Sommer - Tivoli A/S.

C-bus is a trademark of Corollary, Inc. in the United States and/or other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States and/or other countries.

PC Direct is a trademark of Ziff Communications Company in the United States and/or other

countries and is used by IBM Corporation under license.

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States and/or other countries.

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

SET, SET Secure Electronic Transaction, and the SET Logo are trademarks owned by SET Secure Electronic Transaction LLC.

Other company, product, and service names may be trademarks or service marks of others.

Abbreviations and acronyms

AIX	Advanced Interactive eXecutive from IBM	EBCDIC	extended binary coded decimal interchange code
APAR	authorized program analysis report	ECS	enhanced catalog sharing
ARM	automatic restart manager	ECSA	extended common storage area
ASCII	American National Standard Code for Information Interchange	EDM	environment descriptor management
BLOB	binary large objects	ERP	enterprise resource planning
CCSID	coded character set identifier	ESA	Enterprise Systems Architecture
CCA	client configuration assistant	ETR	external throughput rate, an elapsed time measure, focuses on system capacity
CFCC	coupling facility control code	FDT	functional track directory
CTT	created temporary table	FTP	File Transfer Program
CEC	central electronics complex	GB	gigabyte (1,073,741,824 bytes)
CD	compact disk	GBP	group buffer pool
CF	coupling facility	GRS	global resource serialization
CFRM	coupling facility resource management	GUI	graphical user interface
CLI	call level interface	HPJ	high performance Java
CLP	command line processor	IBM	International Business Machines Corporation
CPU	central processing unit	ICF	integrated catalog facility
CSA	common storage area	ICF	integrated coupling facility
DASD	direct access storage device	ICMF	internal coupling migration facility
DB2 PM	DB2 performance monitor	IFCID	instrumentation facility component identifier
DBAT	database access thread	IFI	instrumentation facility interface
DBD	database descriptor	IPLA	IBM Program Licence Agreement
DBID	database identifier	IRLM	internal resource lock manager
DBRM	database request module	ISPF	interactive system productivity facility
DSC	dynamic statement cache, local or global	ISV	independent software vendor
DCL	data control language	I/O	input/output
DDCS	distributed database connection services	ITR	internal throughput rate, a processor time measure, focuses on processor capacity
DDF	distributed data facility	ITSO	International Technical Support Organization
DDL	data definition language	IVP	installation verification process
DLL	dynamic load library manipulation language	JDBC	Java Database Connectivity
DML	data manipulation language	JFS	journaled file systems
DNS	domain name server	JVM	Java Virtual Machine
DRDA	distributed relational database architecture	KB	kilobyte (1,024 bytes)
DTT	declared temporary tables	LOB	large object
EA	extended addressability		

LPL	logical page list
LPAR	logically partitioned mode
LRECL	logical record length
LRSN	log record sequence number
LUW	logical unit of work
LVM	logical volume manager
MB	megabyte (1,048,576 bytes)
NPI	non-partitioning index
ODB	object descriptor in DBD
ODBC	Open Data Base Connectivity
OS/390	Operating System/390
PAV	parallel access volume
PDS	partitioned data set
PIB	parallel index build
PSID	pageset identifier
PSP	preventive service planning
PTF	program temporary fix
PUNC	possibly uncommitted
QMF	Query Management Facility
QA	Quality Assurance
RACF	Resource Access Control Facility
RBA	relative byte address
RECFM	record format
RID	record identifier
RRS	resource recovery services
RRSAF	resource recovery services attach facility
RS	read stability
RR	repeatable read
SDK	software developers kit
SMIT	System Management Interface Tool
UOW	unit of work

Index

Symbols

=ANY 186
=SOME 186

A

ABSOLUTE 164
absolute moves 164
ACCESSTYPE 120
adding an identity column 107
after join step predicates 182
after trigger 18, 35, 36
alias 8, 42
ALL 186
arithmetic operator 49
AS TEMP 89
ASCII 90, 213
authorization 8
automatic rebind 33
auxiliary table 42

B

base table 89, 92
before trigger 35
BIND 200
bind 92
BLOB 45
boolean term 121
buffer pool 86, 90
built-in data type 43, 44, 45, 49
built-in function exploitation 72
built-in functions 25, 49, 57, 58, 59, 60, 64, 65, 72
 arithmetic and string operators 72
 before version 6 72
 column functions 62, 72, 73, 75
 in version 6 73
 in version 7 75
 restrictions 76
 scalar functions 72, 73, 75
built-in operator 49

C

CALL 38
CALLTYPE 63
cartesian product 182
cascade path 29
cascaded level 36
CASE 125
CASE expression 125, 143
 alternatives 131
 characteristics 126
 division by zero 129
ELSE 127

END 127
 grouping 132
 other uses 131
 pivot 132
 restrictions 133
 trigger 130
UNION 130
WHEN 126
 why 127
CAST function 44, 45, 47, 48, 53, 67
casting 49
catalog table 42
CCSID 90
check constraint 38, 39, 47, 121
check pending 38
CICS 95
CLOB 45
COALESCE 131
column function 58, 60, 72, 168
COMMIT 84, 85, 86
comparison operator 49
CONNECT 93
connection 81, 84, 85, 93
constraint 35
cost of trigger 37
COST_CATEGORY 37
CREATE DISTINCT TYPE 44
CREATE FUNCTION 63
CREATE GLOBAL TEMPORARY TABLE 82
CREATE SCHEMA 10, 11
CREATE TABLE 83
created temporary tables 79, 80, 81, 83, 84, 85, 86
 characteristics 82
 COMMIT 85
 considerations 86
 creating an instance 84
 multiple instances 85
 pitfalls 86
 RELEASE(COMMIT) 85
 RELEASE(DEALLOCATE) 85
 restrictions 86
 result sets 85
 ROLLBACK 85
CREATETAB 82
CREATETMTAB 82
Cross Loader 217
CURRENT 165
CURRENT PATH 8, 9, 44
current server 83, 84
CURRENTDATA 177, 179
cursor movement 164
cursor stability 177
cursor types 151

D

- Data Propagator 39
- data sharing 89
- data sharing group 89
- data warehouse 81
- DB2 Connect 154
- DB2 Extender 60
- DB2 family compatibility 194
- DB2 Utilities Suite 212
- DBADM 82
- DBCLOB 45
- DBCTRL 82
- DBD 37
- DBMAINT 82
- DBRM 200, 201
- DBRM Colon Finder 201
- DDF 95
- DECLARE CURSOR 150, 155, 156
- DECLARE GLOBAL TEMPORARY TABLE 88
- declared temporary tables 79, 85, 88, 89, 90, 92, 93
 - AS TEMP 89
 - characteristics 88
 - considerations 94
 - converting created temporary tables 95
 - CREATE LIKE 95
 - index 94
 - ON COMMIT DELETE ROWS 95
 - ON COMMIT PRESERVE ROWS 95
 - PUBLIC 95
 - remote 93
 - scrollable cursors 93
 - three-part name 93
 - USE 95
- DEFAULT 191
- default clause 47
- default value 83
- DELETE
 - self-referencing 193
- DELETE ALL 86
- delete hole 162, 170
- delete rule 35
- delete trigger 22
- differences between table types 80
- direct row access 119
- drain 207
- DRDA 85, 93, 197
- DROP TABLE 87
- DROP TRIGGER 34
- DSN1COPY 213
- DSNDB07 37
- DSNH315I 200
- DSNHSP 11, 224
- DSNTIAUL 205, 212
 - comparing 215
- During join predicates 182

E

- E/R-diagram 224
- EBCDIC 90, 213

- editproc 87, 121
- EDM pool 90
- encoding scheme 90
- ENDEXEC 217
- EPOCH 121
- EXEC SQL 217, 219
- EXECUTE IMMEDIATE 217
- EXPLAIN 142
- expression 192
- external 58
 - action 30
 - function 51, 59
 - program 57
- external resource 62
- extract 88

F

- falling back 120
- fence 64
- FETCH 154, 157
- FETCH ABSOLUTE 0 165
- FETCH AFTER 164
- FETCH BEFORE 164, 165
- FETCH FIRST n ROWS 197
- FETCH FIRST n ROWS ONLY
 - SELECT INTO 198
- FETCH LAST 165
- FETCH SENSITIVE 159, 162
- fieldproc 87, 121
- FINAL CALL 63
- FOR FETCH ONLY 199
- FOR UPDATE OF 161, 174
- foreign key 40
- fullselect 136

G

- GENERATED ALWAYS 113, 122
- GENERATED BY DEFAULT 113, 122
- global temporary table 79, 80
- GRANT 8, 87
- GRANT USAGE 47
- GROUP BY 196

H

- HAVING 196
- HEADER 214
- hole 162, 170
- host variables 53, 199
 - preceded by a colon 200, 201

I

- identity column 104
 - add 107
 - CACHE 105, 110, 111
 - characteristics 104
 - comparison 122
 - creating 105
 - CYCLE 105

- data sharing 110
- deficiencies 110
- design considerations 111
- DSN_IDENTITY 107
- GENERATED ALWAYS 104, 106, 107
- GENERATED BY DEFAULT 107, 111
- IDENTITY_VAL_LOCAL 109
- IGNOREFIELDS 107
- INCLUDING IDENTITY COLUMN ATTRIBUTES 106
- LOAD 107
- MAXVALUE 105
- MINVALUE 105
- OVERRIDING USER VALUE 108
- populating 106
- restrictions 112
- START WITH 105
- when 104
- IDENTITY_VAL_LOCAL 109, 122
- IMS 85, 88
- IN 187
 - supports any expression 201
- index 8, 86, 88
- indexable 81
- INNER JOIN 183
- INSENSITIVE 152, 155, 156
- INSENSITIVE FETCH 159
- INSERT 191, 206
 - DEFAULT keyword 191
 - self-referencing SELECT 192
 - UNION 193
 - using expressions 192
- INSERT program 208
- insert trigger 22
- instance workfile 81
- internal 58
- isolation level
 - UR 215

J
Java 154

L
large object 45
LEFT OUTER JOIN 184
LIKE 83
LISTDEF 216
LOAD 16, 42

- FORMAT UNLOAD 212
- SHRLEVEL CHANGE 206
- SHRLEVEL NONE 206

 LOAD PART 119
LOAD RESUME 38, 42, 205
LOB 87, 113, 156
lock 81, 82, 83, 86
LOCK TABLE 87
locking 86, 207
log 81, 82, 83, 86
logical work file 85
LONG VARCHAR 44

LONG VARCHAR 44

M
materialize 154
missing colons

- sample program 201

N
nested loop join 81
nested table expression 136
nesting level 29, 42
NEXT 165
NO CASCADE BEFORE 17
non-correlated subquery 193
non-DB2 resource 62
non-partitioning index 143
non-relational data 85
NOT IN 187
null value 83
NULLIF 131

O
OBD 37
object-oriented 43
object-relational 59
ODBC 88
ON clause extensions 182
ON COMMIT DELETE ROWS 89
ON COMMIT PRESERVE ROWS 89, 92
ON condition 182
ON DELETE CASCADE 17
ON DELETE SET NULL 16
online LOAD RESUME 16, 206

- clustering 207
- commit frequency 208
- duplicate keys 207
- free space 208
- logging 207
- pitfalls 209
- restart 208
- restrictions 209
- RI 207

 OPEN CURSOR 155
Optimistic Locking Concur By Value 174, 175
optimize 37
OPTIMIZE FOR 197
ORDER BY 188

- expression 189
- select list 188
- sort avoidance 190

 overload 60

P
package 32, 34, 39, 59, 92
page size 89, 90, 93
parallelism 216
PARAMETER STYLE DB2SQL 27
PARENT_QBLOCKNO 142

- partitioning 144
- partitioning key update 202
- PARTKEYU 202
- PATH 9
- plan 92
- PLAN_TABLE 142
- populate 85, 88
- positioned DELETE 176
- positioned UPDATE 175
- powerful SQL 123
- PQ16946 202
- PQ19897 210, 212
- PQ23219 210, 212
- PQ34506 38
- PQ53030 15
- PRIMARY_ACCESSTYPE 119
- PRIOR 165
- privilege 82, 83
- propagation 39

Q

- QBLOCK_TYPE 142
- QMF 154
- QUALIFIER 9, 10
- qualifier 8
- quantified predicates 185

R

- RAISE_ERROR 25, 127
- read stability 177
- read-only cursor 153
- REBIND 37, 200
- rebind 33
- REBIND TRIGGER PACKAGE 33
- recovery 82
- Redbooks Web site 254
 - Contact us xix
- referential constraint 35, 38, 87
- referential integrity 28
- RELATIVE 164
- relative moves 165
- RELEASE(DEALLOCATE) 37
- remote server 84, 93
- REORG
 - DISCARD 205, 210
 - DISCARD restrictions 211
 - UNLOAD EXTERNAL 205, 211, 212, 213
 - comparing 215
 - UNLOAD ONLY 212
- repeatable read 177
- restart 216
- RESTRICT 47
- result set 85, 88
- result table 154, 162, 166
- REXX 154
- REXX procedure 201
- RI 39
- RID 113
- RID list processing 121

- ROLLBACK 84, 85
- ROLLBACK TO SAVEPOINT 156
- row expressions 185
 - quantified predicates 186
 - restrictions 188
 - types 185
- row size 89
- row trigger 18
- ROWID 87, 112, 113
 - casting 117
 - comparison 122
 - DCLGEN 117
 - direct row access 119
 - EPOCH 121
 - GENERATED BY DEFAULT 115
 - implementation 113
 - IMS 120
 - LOB 113
 - partitioning 118
 - restrictions 121
 - storing 120
 - UPDATE 114
 - USAGE SQL TYPE IS 117
- row-value-expression 185

S

- SAR 35, 36
- savepoint 98
 - characteristics 99
 - CONNECT 101
 - ON ROLLBACK RETAIN CURSORS 100
 - ON ROLLBACK RETAIN LOCKS 100
 - RELEASE SAVEPOINT 99
 - remote connections 101
 - restrictions 102
 - ROLLBACK TO SAVEPOINT 99
 - UNIQUE 100
 - why 98
- scalar function 58, 60, 61, 72, 168
- scalar subquery 195
- schema 7, 8, 10, 72
 - authorization 8
 - authorization ID 10
 - characteristics 8
 - name 16, 32, 44
 - object 14
 - processor 10
- scratchpad 64
- SCROLL 155, 156
- scrollable cursor 93, 149
 - absolute moves 164
 - allowable combinations 160
 - characteristics 151
 - choose the right type 153
 - CLOSE CURSOR 156
 - cursor movement 164
 - declaring 155
 - delete hole 170
 - FETCH 154, 157
 - FETCH ABSOLUTE 159

- FETCH AFTER 158
- FETCH BEFORE 158
- FETCH CURRENT 158
- FETCH FIRST 158
- FETCH LAST 158
- FETCH NEXT 158
- FETCH PRIOR 158
- FETCH RELATIVE 159
- fetching 157
- in depth 152
- INSENSITIVE 154
- insensitive 151, 152
- LOB 156
- locking 177
- OPEN CURSOR 155
- opening 155
- read-only 153
- recommendations 179
- relative moves 165
- SENSITIVE 154
- sensitive 152
- sensitive dynamic 151
- sensitive static 151
- stored procedures 178
- TEMP database 155
- update hole 170
- using 154
- using functions 168
- why 150
- searched-when-clause 127
- SELECT INTO 198
- self-referencing
 - DELETE 193
 - INSERT 192
 - restrictions 194
 - restrictions on usage 194
 - UPDATE 193
- SENSITIVE STATIC 152, 155, 156, 168
- SESSION 88, 92
- SET 23, 196
- SET CURRENT PATH 9
- set of affected rows (SAR) 35
- SIGNAL SQLSTATE 25
- simple-when-clause 127
- source data type 44
- sourced 44
- sourced function 50, 59
- special register 8, 9, 10
- splitting a table 143
- SPUFI 154
- SQL enhancements 181
- SQL statement terminator 20
- SQL_STATEMNT_TABLE 37
- SQLSTATE 64
- SQLWARN 157
- SQLWARN0 157
- SQLWARN1 157
- SQLWARN4 157
- statement trigger 18
- STATIC 155

- stored 28
- stored procedure 7, 8, 9, 15, 28, 29, 33, 38, 42, 81, 85, 88
- string operator 58
- strong typing 45
- subquery 182
- subselect 136, 193
- synonym 42
- SYSADM 8, 82
- SYSCTRL 82
- SYSFUN 8, 9
- SYSIBM 8, 9, 72
- SYSOBJ 35
- SYSPACKAGE 35
- SYSPROC 8, 9
- SYSTABAUTH 35
- SYSTRIGGERS 30, 35

T

- table 83
- table access predicates 182
- table function 58, 60
- table space 89, 90, 93
- TABLE_TYPE 142
- TEMP database 90, 93, 155
- TEMP table spaces 93
- TEMPLATE 216
- temporary database 88, 89
- temporary table 42, 79, 92
- temporary table space 88
- thread 81, 85, 89, 90, 92
- thread reuse 85
- thread termination 85
- three-part name 42, 85, 93
- three-part table name 85
- totally after join predicates 182
- transition table 22, 37, 62
- transition variable 22, 23, 37
- transitional business rules 14, 15
- trigger 7, 8, 11, 21, 22, 23, 25, 26, 28, 29, 30, 35, 37, 38, 39, 42, 47, 53, 87, 130
 - action 19, 21, 30, 33, 36, 37
 - activation time 17, 22
 - allowable combinations 22
 - ATOMIC 19
 - body 19, 23
 - cascading 28
 - definition 14
 - error handling 26
 - external actions backout 30
 - FOR EACH ROW 18
 - FOR EACH STATEMENT 18
 - granularity 18, 22
 - invoking SP and UDF 23
 - name 16
 - ordering 29
 - package 32, 33, 37
 - passing transition tables 30
 - processing 37
 - RAISE_ERROR 24
 - restrictions 42

- SIGNAL SQLSTATE 24
- table locator 31
- transition tables 21
- transition variables 20
- useful queries 41
- valid statements 22
- VALUES 23
- WHEN 19
- trigger action condition 19
- trigger characteristics 16
- trigger happy 38
- trigger package 33
 - dependencies 33
- TRIGGERAUTH 35
- triggered operation 19
- triggering event 17
- triggering operation 16, 19, 22, 29
- triggering table 16
- two-part name 8, 44

U

- UDF 9, 15, 21, 23, 24, 27, 28, 29, 30, 42, 57, 58, 59, 64
 - column functions 61
 - definition 59
 - design considerations 64
 - efficiency 64
 - implementation and maintenance 60
 - scalar functions 60
 - sourced 66
 - sourced function 65
 - table functions 62
- UDT 9, 43, 44, 45, 47, 48, 54
 - CAST 44
 - catalog tables 56
 - COMMENT ON 47
 - DROP 47
 - EXECUTE 44
 - GRANT EXECUTE ON 47
 - privileges 46
 - USAGE 44
- UNDO record 79
- UNICODE 90, 213
- UNION 127, 136, 142
 - UPDATE statement 139
- UNION ALL 136
- union everywhere 136
 - basic predicates 137
 - EXISTS predicates 138
 - explain 142
 - IN predicates 139
 - INSERT statements 139
 - nested table expressions 136
 - quantified predicates 137
 - subqueries 137
 - UPDATE statements 140
 - views 140
- UNIQUE 83
- unit of work 11, 84
- UNLOAD 205, 212, 213
 - comparing 215

- LOB table spaces 214
- pitfalls 215
- restriction 214
- unqualified table reference 92
- UPDATE 195
 - scalar subquery 195
 - self-referencing 193
- update hole 162, 170, 172
- update trigger 22
- update with subselect
 - conditions 196
 - self referencing 197
- user-defined 58
- user-defined column function 62
- user-defined distinct type 7, 9, 44, 72
- user-defined function 7, 9, 15, 42, 58, 59, 60, 64, 72
- user-defined scalar function 61
- user-defined table function 62
- USING CCSID 90

V

- validproc 87
- VALUES INTO 199
- view 35, 42, 47, 83
- views 8

W

- WHEN 126
- WHERE CURRENT OF 87, 161
- WITH CHECK OPTIO 144
- WITH CHECK OPTION 35, 36, 87
- WITH COMPARISONS 45
- WITH HOLD 84, 85, 89
- WITH HOLD, 85
- workfile 81, 82, 83, 86

Z

- ZPARM 202



Redbooks

DB2 for z/OS Application Programming Topics

(0.5" spine)
0.475" x 0.873"
250 x 459 pages



DB2 for z/OS

Application Programming Topics



How to implement object-oriented enhancements

Increased program design flexibility

Examples of more powerful SQL

This IBM Redbook describes the major enhancements that affect application programming when accessing DB2 data on a S/390 or z/Series platform, including the object-oriented extensions such as triggers, user-defined function and user-defined distinct types, the usage of temporary tables, savepoints and the numerous extensions to the SQL language, to help you build powerful, reliable and scalable applications, whether it be in a traditional environment or on an e-business platform.

IBM DATABASE 2 Universal Database Server for z/OS and OS/390 Version 7 is currently at its eleventh release. Over the last couple of versions a large number of enhancements were added to the product. Many of these affect application programming and the way you access your DB2 data.

This book will help you to understand how these programming enhancements work and will provide examples of how to use them. It provides considerations and recommendations for implementing these enhancements and for evaluating their applicability in your DB2 environments.

**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**

SG24-6300-00

ISBN 073842353X