



# IBM solidDB **SQL Guide**

**Version 6.1** | June 2008

---

# IBM solidDB SQL Guide

Copyright © Solid Information Technology Ltd. 1993, 2008

Document number: SQLG61

Product version: 06.10.0014

Date: 2008-06-13

All rights reserved. No portion of this product may be used in any way except as expressly authorized in writing by Solid Information Technology Ltd. or International Business Machines Corporation.

"IBM", the IBM logo, "DB2", "Informix", "Solid" and "solidDB" are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

All other products, services, companies and publications are trademarks or registered trademarks of their respective owners.

This product is protected by U.S. patents 6144941, 7136912, 6970876, 7139775, 6978396, and 7266702.

This product contains lexical analyzer Flex. Copyright (c) 1990 The Regents of the University of California. All rights reserved.

This code is derived from software contributed to Berkeley by Vern Paxson. The United States Government has rights in this work pursuant to contract no. DE-AC03-76SF00098 between the United States Department of Energy and the University of California. Redistribution and use in source and binary forms are permitted provided that: (1) source distributions retain this entire copyright notice and comment, and (2) distributions including binaries display the following acknowledgement: "This product includes software developed by the University of California, Berkeley and its contributors" in the documentation or other materials provided with the distribution and in all advertising materials mentioning features or use of this software. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission. THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

This product contains zlib general purpose compression library version 1.1.4, March 11th, 2002. Copyright (C) 1995-2002 Jean-loup Gailly and Mark Adler.

This software is provided "as-is", without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software. Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions: 1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required. 2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software. 3. This notice may not be removed or altered from any source distribution.

This product contains the Qsort routine in the external sorter, Copyright (c) 1980, 1983, 1990 The Regents of the University of California. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
  2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
-

- 
3. All advertising materials mentioning features or use of this software must display the following acknowledgment: This product includes software developed by the University of California, Berkeley and its contributors.
  4. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

This product contains the DES cipher (in ECB mode), parts of this code are Copyright (C) 1996 Geoffrey Keating. All rights reserved.

Its use is FREE FOR COMMERCIAL AND NON-COMMERCIAL USE as long as the following conditions are adhered to.

Copyright remains Geoffrey Keating's, and as such any Copyright notices in the code are not to be removed. If this code is used in a product, Geoffrey Keating should be given attribution as the author of the parts used. This can be in the form of a textual message at program startup or in documentation (online or textual) provided with the package.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software must display the following acknowledgment: This product includes software developed by Eric Young (eay@mincom.oz.au)

THIS SOFTWARE IS PROVIDED BY GEOFFREY KEATING ``AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Parts of this code (in particular, the string representing SPtrans below) are Copyright (C) 1995 Eric Young (eay@mincom.oz.au). All rights reserved.

Its use is FREE FOR COMMERCIAL AND NON-COMMERCIAL USE as long as the following conditions are adhered to.

---

---

Copyright remains Eric Young's, and as such any Copyright notices in the code are not to be removed. If this code is used in a product, Eric Young should be given attribution as the author of the parts used. This can be in the form of a textual message at program startup or in documentation (online or textual) provided with the package.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software must display the following acknowledgment: This product includes software developed by Eric Young (eay@mincom.oz.au)

THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

This product is assigned the U.S. Export Control Classification Number ECCN=5D992b.

---

---

# Table of Contents

1 Welcome .....	1
1.1 About This Guide .....	1
1.1.1 Organization .....	1
1.1.2 Audience .....	2
1.2 Conventions .....	2
1.2.1 Typographic Conventions .....	2
1.2.2 Syntax Notation .....	3
1.3 IBM solidDB Documentation .....	3
2 Database Concepts .....	5
2.1 Relational Databases .....	5
2.1.1 Tables, Rows, and Columns .....	5
2.1.2 Relating Data In Different Tables .....	6
2.2 Client-Server Architecture .....	9
2.3 Multi-User Capability .....	9
2.4 Transactions .....	10
2.5 Transaction Logging and Recovery .....	10
2.5.1 Background .....	10
2.6 Summary .....	11
3 Getting Started with SQL .....	13
3.1 Tables, Rows, and Columns .....	13
3.2 SQL .....	13
3.3 The Mathematical Origins of SQL .....	16
3.4 Creating Tables with Related Data .....	16
3.4.1 Table Aliases .....	19
3.4.2 Subqueries .....	20
3.5 Which Formats Are Used for Each Data Type? .....	21
3.5.1 BLOBs (or Binary Data Types) .....	22
3.5.2 NULL IS NOT NULL (or "How to say 'None of the above' in SQL") .....	23
3.5.3 NOT NULL .....	26
3.5.4 Expressions and Casts .....	26
3.5.5 Row Value Constructors .....	30
3.6 More about Transactions .....	32
3.7 Summary .....	33
3.8 Where to Find Additional Information about SQL .....	33
4 Stored Procedures, Events, Triggers, and Sequences .....	35
4.1 Stored Procedures .....	35
4.1.1 Basic Procedure Structure .....	35
4.1.2 Naming Procedures .....	36
4.1.3 Parameter Section .....	36

4.1.4 Declare Section .....	41
4.1.5 Procedure Body .....	42
4.1.6 Assignments .....	42
4.1.7 Expressions .....	45
4.1.8 Control Structures .....	47
4.2 Remote Stored Procedures .....	56
4.2.1 ACCESS RIGHTS .....	58
4.3 Using SQL in a Stored Procedure .....	60
4.3.1 EXECDIRECT .....	60
4.3.2 Using a Cursor .....	61
4.3.3 Error Handling .....	64
4.3.4 Parameter Markers in Cursors .....	67
4.4 Calling other Procedures .....	70
4.4.1 Positioned Updates and Deletes .....	70
4.4.2 Transactions .....	72
4.4.3 Default Cursor Management .....	72
4.4.4 Notes on SQL .....	73
4.4.5 Functions for Procedure Stack Viewing .....	74
4.5 Procedure Privileges .....	74
4.6 Using Triggers .....	75
4.6.1 How Triggers Work .....	75
4.6.2 Creating Triggers .....	76
4.6.3 Keywords and Clauses .....	77
4.6.4 Triggers Comments and Restrictions .....	81
4.7 Triggers and Procedures .....	82
4.7.1 Setting Default or Derived Columns .....	83
4.7.2 Using Parameters and Variables .....	83
4.8 Triggers and Transactions .....	86
4.8.1 Recursion and Concurrency Conflict Errors .....	87
4.8.2 Trigger Privileges and Security .....	95
4.8.3 Raising Errors from Inside Triggers .....	96
4.8.4 Trigger Example .....	96
4.8.5 Dropping Triggers .....	101
4.8.6 Altering Trigger Attributes .....	103
4.8.7 Obtaining Trigger Information .....	103
4.8.8 Trigger Functions .....	103
4.8.9 SYS_TRIGGERS System Table .....	104
4.8.10 Trigger Parameter Settings .....	105
4.9 Deferred Procedure Calls .....	105
4.9.1 Sync Pull Notify ("Push Synchronization") Example .....	117
4.9.2 Tracing the Execution of Background Jobs .....	120
4.9.3 Controlling Background Tasks .....	122

4.10 Using Sequences .....	122
4.11 Using Events .....	123
5 Using IBM solidDB SQL for Database Administration .....	137
5.1 Using IBM solidDB SQL Syntax .....	137
5.1.1 IBM solidDB SQL Data Types .....	137
5.1.2 IBM solidDB ADMIN COMMAND .....	137
5.1.3 Using Functions .....	138
5.2 Managing User Privileges and Roles .....	138
5.2.1 User Privileges .....	139
5.2.2 User Roles .....	139
5.2.3 Examples of SQL Statements .....	140
5.3 Managing Tables .....	143
5.3.1 Accessing System Tables .....	144
5.3.2 Examples of SQL Statements .....	145
5.4 Managing Indexes .....	147
5.4.1 Examples of SQL Statements .....	148
5.4.2 Primary Key Indexes .....	149
5.4.3 Secondary Key Indexes .....	149
5.4.4 Protection Against Duplicate Indexes .....	150
5.5 Referential Integrity .....	151
5.5.1 Primary Keys and Candidate Keys .....	152
5.5.2 Foreign Keys .....	152
5.5.3 Referential Actions .....	155
5.5.4 Dynamic Constraint Management .....	156
5.6 Managing Database Objects .....	158
5.6.1 Introduction .....	158
5.6.2 Catalogs .....	159
5.6.3 Schemas .....	159
5.6.4 Uniquely Identifying Objects within Catalogs and Schemas .....	159
5.6.5 Examples of SQL Statements .....	161
6 Managing Transactions .....	165
6.1 Managing Transactions .....	165
6.1.1 Defining Read-Only and Read-Write Transactions .....	165
6.1.2 Setting Concurrency Control .....	166
6.2 Concurrency Control and Locking .....	167
6.2.1 The Purpose of Concurrency Control .....	168
6.2.2 EXCLUSIVE vs. SHARED LOCKS .....	169
6.2.3 PESSIMISTIC vs. OPTIMISTIC Concurrency Control .....	169
6.2.4 Table Locks .....	175
6.2.5 Lock Duration .....	177
6.2.6 TRANSACTION ISOLATION Levels .....	178
6.2.7 Miscellaneous Lock Information .....	179

6.2.8 Summary of Locking Information .....	180
6.3 Choosing the Transaction Durability .....	180
6.3.1 Setting the Transaction Durability Level .....	180
7 Diagnostics and Troubleshooting .....	183
7.1 Observing Performance .....	183
7.1.1 SQL Info Facility .....	183
7.1.2 The EXPLAIN PLAN FOR Statement .....	184
7.2 Problem Reporting .....	190
7.3 Problem Categories .....	191
7.3.1 IBM solidDB ODBC API Problems .....	191
7.3.2 IBM solidDB ODBC Driver Problems .....	192
7.3.3 IBM solidDB JDBC Driver Problems .....	192
7.3.4 UNIFACE Driver for IBM solidDB Problems .....	193
7.3.5 Communication between a Client and Server .....	193
7.4 Tracing Facilities for Stored Procedures and Triggers .....	193
7.4.1 User-Definable Trace Output from Procedure Code .....	193
7.4.2 Procedure Execution Trace .....	194
7.5 Measuring and Improving Performance of START AFTER COMMIT Statements .....	196
7.5.1 Tuning Performance of START AFTER COMMIT Statements .....	196
7.5.2 Analyzing Failures in START AFTER COMMIT Statements .....	196
8 Performance Tuning .....	197
8.1 Tuning SQL Statements and Applications .....	197
8.1.1 Evaluating Application Performance .....	198
8.1.2 Using Stored Procedure Language .....	198
8.2 Optimizing Single-Table SQL Queries .....	199
8.3 Using Indexes to Improve Query Performance .....	199
8.3.1 Full Table Scan .....	201
8.3.2 Concatenated Indexes .....	201
8.4 Waiting on Events .....	202
8.5 Optimizing Batch Inserts and Updates .....	203
8.5.1 Increasing Speed of Batch Inserts and Updates .....	203
8.6 Using Optimizer Hints .....	204
8.7 Diagnosing Poor Performance .....	206
A Data Types .....	209
A.1 Supported Data Types .....	209
A.1.1 Character Data Types .....	209
A.1.2 Numeric Data Types .....	210
A.1.3 Binary Data Types .....	213
A.1.4 Date Data Type .....	214
A.1.5 Time Data Type .....	214
A.1.6 Timestamp Data Type .....	214
A.1.7 The Smallest Possible Non-Zero Numbers .....	215



A.1.8 BLOBs and CLOBs .....	215
<b>B IBM solidDB SQL Syntax .....</b>	<b>219</b>
<b>B.1 ADMIN COMMAND .....</b>	<b>219</b>
B.1.1 Supported in .....	219
B.1.2 Usage .....	219
<b>B.2 ADMIN EVENT .....</b>	<b>238</b>
B.2.1 Usage .....	238
B.2.2 Examples .....	239
<b>B.3 ALTER TABLE .....</b>	<b>239</b>
B.3.1 Usage .....	240
B.3.2 Example .....	241
<b>B.4 ALTER TABLE ... SET HISTORY COLUMNS .....</b>	<b>241</b>
B.4.1 Usage .....	241
B.4.2 Usage in Master .....	242
B.4.3 Usage in Replica .....	242
B.4.4 Example .....	242
B.4.5 Return Values .....	242
B.4.6 See Also .....	242
<b>B.5 ALTER TABLE ... SET SYNCHISTORY .....</b>	<b>243</b>
B.5.1 Usage .....	243
B.5.2 Usage in Master .....	244
B.5.3 Usage in Replica .....	244
B.5.4 Example .....	244
B.5.5 Return Values .....	244
B.5.6 See Also .....	245
<b>B.6 ALTER TRIGGER .....</b>	<b>245</b>
B.6.1 Usage .....	245
B.6.2 Example .....	246
<b>B.7 ALTER USER .....</b>	<b>246</b>
B.7.1 Usage .....	246
B.7.2 Example .....	246
<b>B.8 ALTER USER .....</b>	<b>246</b>
B.8.1 Usage .....	246
B.8.2 Usage in Master .....	247
B.8.3 Usage in Replica .....	247
B.8.4 Example .....	247
B.8.5 Return Values .....	248
<b>B.9 CALL .....</b>	<b>248</b>
B.9.1 Supported in .....	248
B.9.2 Usage .....	248
B.9.3 Transactions .....	249
B.9.4 Return Values from the Remote Procedure .....	250

B.9.5 Access Rights for Remote Stored Procedure Calls .....	250
B.9.6 Durability .....	251
B.9.7 Example .....	252
B.10 COMMIT WORK .....	252
B.10.1 Usage .....	252
B.10.2 Example .....	252
B.10.3 See Also .....	252
B.11 CREATE CATALOG .....	252
B.11.1 Usage .....	252
B.11.2 Examples .....	255
B.12 CREATE EVENT .....	256
B.12.1 Usage .....	256
B.12.2 Example .....	258
B.12.3 See Also .....	258
B.13 CREATE INDEX .....	258
B.13.1 Usage .....	258
B.13.2 Example .....	259
B.13.3 See Also .....	259
B.14 CREATE PROCEDURE .....	259
B.14.1 Usage .....	261
B.14.2 Preparing SQL Statements .....	266
B.14.3 Executing Prepared SQL Statements .....	266
B.14.4 Fetching Results .....	267
B.14.5 Closing and Dropping Cursors .....	267
B.14.6 Checking for Errors .....	267
B.14.7 Using Transactions .....	268
B.14.8 Using Sequencer Objects and Event Alerts .....	268
B.14.9 Writetrace .....	268
B.14.10 Procedure Stack Functions .....	268
B.14.11 Dynamic Cursor Names .....	269
B.14.12 EXECDIRECT .....	270
B.14.13 CREATE PROCEDURE .....	271
B.14.14 Using the Explicit RETURN Statement .....	271
B.14.15 Using EXECDIRECT .....	272
B.14.16 Using CURSORNAME .....	273
B.14.17 Using GET_UNIQUE_STRING and CURSORNAME .....	273
B.14.18 Example 6 .....	274
B.14.19 Creating a Unique Name for a Synchronization Message .....	275
B.14.20 Using GET_UNIQUE_STRING .....	275
B.15 CREATE [OR REPLACE] PUBLICATION .....	277
B.15.1 Usage .....	277
B.15.2 Usage in Master .....	279

B.15.3 Usage in Replica .....	279
B.15.4 Example .....	279
B.15.5 Return Values .....	281
B.16 CREATE ROLE .....	281
B.16.1 Usage .....	281
B.16.2 Example .....	282
B.17 CREATE SCHEMA .....	282
B.17.1 Usage .....	282
B.17.2 Examples .....	284
B.18 CREATE SEQUENCE .....	284
B.18.1 Usage .....	284
B.18.2 Examples .....	286
B.19 CREATE SYNC BOOKMARK .....	286
B.19.1 Supported in .....	286
B.19.2 Usage .....	287
B.19.3 Usage in Master .....	287
B.19.4 Usage in Replica .....	287
B.19.5 Example .....	287
B.19.6 Return Values .....	287
B.20 CREATE TABLE .....	288
B.20.1 Usage .....	289
B.20.2 Example .....	291
B.21 CREATE TRIGGER .....	292
B.21.1 Usage .....	292
B.21.2 Trigger Name .....	293
B.21.3 BEFORE   AFTER Clause .....	294
B.21.4 INSERT   UPDATE   DELETE Clause .....	296
B.21.5 Table_name .....	297
B.21.6 Trigger_body .....	297
B.21.7 REFERENCING Clause .....	297
B.21.8 {OLD   NEW} column_name AS col_identifier .....	298
B.21.9 Triggers Comments and Restrictions .....	299
B.22 CREATE USER .....	303
B.22.1 Usage .....	303
B.22.2 Example .....	303
B.23 CREATE VIEW .....	303
B.23.1 Usage .....	303
B.23.2 Example .....	303
B.24 DELETE .....	304
B.24.1 Usage .....	304
B.24.2 Example .....	304
B.25 DELETE (positioned) .....	304

B.25.1 Usage .....	304
B.25.2 Example .....	304
B.26 DROP CATALOG .....	304
B.26.1 Usage .....	304
B.26.2 Example .....	305
B.27 DROP EVENT .....	305
B.27.1 Usage .....	305
B.27.2 Example .....	305
B.28 DROP INDEX .....	306
B.28.1 Usage .....	306
B.28.2 Example .....	306
B.29 DROP MASTER .....	306
B.29.1 Usage .....	306
B.29.2 Usage in Master .....	307
B.29.3 Usage in Replica .....	307
B.29.4 Examples .....	307
B.29.5 Return Values .....	307
B.30 DROP PROCEDURE .....	307
B.30.1 Usage .....	307
B.30.2 Example .....	308
B.31 DROP PUBLICATION .....	308
B.31.1 Usage .....	308
B.31.2 Usage in Master .....	308
B.31.3 Usage in Replica .....	308
B.31.4 Example .....	308
B.31.5 Return Values .....	308
B.32 DROP PUBLICATION REGISTRATION .....	309
B.32.1 Supported in .....	309
B.32.2 Usage .....	309
B.32.3 Usage in Master .....	309
B.32.4 Usage in Replica .....	309
B.32.5 Example .....	309
B.32.6 Return Values .....	309
B.33 DROP REPLICA .....	310
B.33.1 Supported in .....	310
B.33.2 Usage .....	310
B.33.3 Usage in Master .....	310
B.33.4 Usage in Replica .....	311
B.33.5 Example .....	311
B.33.6 Return Values .....	311
B.34 DROP ROLE .....	311
B.34.1 Usage .....	311

B.34.2 Example .....	312
B.35 DROP SCHEMA .....	312
B.35.1 Usage .....	312
B.35.2 Examples .....	312
B.36 DROP SEQUENCE .....	312
B.36.1 Usage .....	312
B.36.2 Examples .....	313
B.37 DROP SUBSCRIPTION .....	313
B.37.1 Supported in .....	313
B.37.2 Usage .....	313
B.37.3 Usage in Master .....	314
B.37.4 Usage in Replica .....	314
B.37.5 Example .....	314
B.38 DROP SYNC BOOKMARK .....	315
B.38.1 Supported in .....	315
B.38.2 Usage .....	315
B.38.3 Usage in Master .....	316
B.38.4 Usage in Replica .....	316
B.38.5 Example .....	316
B.38.6 Return Values .....	316
B.39 DROP TABLE .....	317
B.39.1 Usage .....	317
B.39.2 Examples .....	317
B.40 DROP TRIGGER .....	318
B.40.1 Usage .....	318
B.40.2 Examples .....	318
B.41 DROP USER .....	318
B.41.1 Usage .....	318
B.41.2 Example .....	319
B.42 DROP VIEW .....	319
B.42.1 Usage .....	319
B.42.2 Examples .....	319
B.43 EXPLAIN PLAN FOR .....	319
B.43.1 Usage .....	319
B.43.2 Example .....	319
B.44 EXPORT SUBSCRIPTION .....	319
B.44.1 Supported in .....	320
B.44.2 Usage .....	320
B.44.3 Usage in Master .....	322
B.44.4 Usage in Replica .....	322
B.44.5 Example .....	322
B.44.6 Return Values .....	322

B.45 EXPORT SUBSCRIPTION TO REPLICA .....	323
B.45.1 Supported in .....	323
B.45.2 Usage .....	323
B.45.3 Usage in Master .....	324
B.45.4 Usage in Replica .....	324
B.45.5 Example .....	324
B.45.6 Return Values .....	324
B.46 GET_PARAM() .....	325
B.46.1 Supported in .....	325
B.46.2 Usage .....	325
B.46.3 Usage in Master .....	325
B.46.4 Usage in Replica .....	325
B.46.5 IBM solidDB System Parameters .....	325
B.46.6 Example .....	326
B.46.7 Return Values .....	326
B.46.8 See Also .....	326
B.47 GRANT .....	327
B.47.1 Usage .....	327
B.47.2 Example .....	328
B.47.3 See Also .....	328
B.48 GRANT REFRESH .....	328
B.48.1 Supported in .....	328
B.48.2 Usage .....	328
B.48.3 Usage in Master .....	329
B.48.4 Usage in Replica .....	329
B.48.5 Example .....	329
B.48.6 Return Values .....	329
B.49 HINT .....	329
B.49.1 Pseudo Comment Identifier .....	330
B.49.2 Example 1 .....	331
B.49.3 Example 2 .....	331
B.49.4 Usage .....	334
B.49.5 Example .....	335
B.50 IMPORT .....	336
B.50.1 Usage .....	336
B.50.2 Usage in Master .....	337
B.50.3 Usage in Replica .....	337
B.50.4 Example .....	337
B.50.5 Return Values .....	337
B.51 INSERT .....	338
B.51.1 Usage .....	339
B.51.2 Example .....	339

B.52 LOCK TABLE .....	340
B.52.1 Usage .....	341
B.52.2 Examples .....	342
B.52.3 Return Values .....	343
B.52.4 See Also .....	343
B.53 MESSAGE APPEND .....	343
B.53.1 Supported in .....	344
B.53.2 Usage .....	344
B.53.3 Usage in Master .....	345
B.53.4 Usage in Replica .....	345
B.53.5 Example .....	346
B.53.6 Return Values .....	346
B.54 MESSAGE BEGIN .....	347
B.54.1 Supported in .....	347
B.54.2 Usage .....	347
B.54.3 Usage in Master .....	347
B.54.4 Usage in Replica .....	347
B.54.5 Example .....	348
B.54.6 Return Values from Master .....	348
B.55 MESSAGE DELETE .....	349
B.55.1 Supported in .....	349
B.55.2 Usage .....	349
B.55.3 Usage in Master .....	349
B.55.4 Usage in Replica .....	349
B.55.5 Example .....	350
B.56 MESSAGE DELETE CURRENT TRANSACTION .....	351
B.56.1 Supported in .....	351
B.56.2 Usage .....	351
B.56.3 Usage in Master .....	352
B.56.4 Usage in Replica .....	352
B.56.5 Example .....	352
B.56.6 Return Values .....	352
B.57 MESSAGE END .....	353
B.57.1 Supported in .....	353
B.57.2 Usage .....	353
B.57.3 Usage in Master .....	353
B.57.4 Usage in Replica .....	353
B.57.5 Return Values from Replica .....	354
B.57.6 Return Values from Master .....	354
B.58 MESSAGE EXECUTE .....	355
B.58.1 Supported in .....	355
B.58.2 Usage .....	355

B.58.3 Usage in Master .....	355
B.58.4 Usage in Replica .....	355
B.58.5 Result Set .....	356
B.58.6 Example .....	356
B.58.7 Return Values .....	356
B.59 MESSAGE FORWARD .....	357
B.59.1 Supported in .....	357
B.59.2 Usage .....	357
B.59.3 Example .....	358
B.59.4 Return Values from Replica .....	359
B.59.5 Return Values from Master .....	361
B.60 MESSAGE FROM REPLICA DELETE .....	361
B.61 MESSAGE FROM REPLICA EXECUTE .....	361
B.61.1 Supported in .....	362
B.61.2 Usage .....	362
B.61.3 Usage in Master .....	362
B.61.4 Usage in Replica .....	362
B.61.5 Example .....	362
B.61.6 Return Values .....	362
B.62 MESSAGE FROM REPLICA RESTART .....	363
B.63 MESSAGE GET REPLY .....	363
B.63.1 Supported in .....	363
B.63.2 Usage .....	364
B.63.3 Usage in Master .....	364
B.63.4 Usage in Replica .....	365
B.63.5 Example .....	365
B.63.6 Return Values from Replica .....	365
B.63.7 Return Values from Master .....	366
B.63.8 Result Set .....	367
B.64 POST EVENT .....	368
B.65 PUT_PARAM() .....	368
B.65.1 Supported in .....	368
B.65.2 Usage .....	368
B.65.3 Usage in Master .....	369
B.65.4 Usage in Replica .....	369
B.65.5 Differences between "PUT_PARAM( )" and "SAVE PROPERTY property_name VALUE property_value;" .....	369
B.65.6 Example .....	370
B.65.7 Return Values .....	370
B.65.8 See Also .....	370
B.66 REFRESH .....	370
B.66.1 Usage .....	370



B.66.2 Example .....	371
B.66.3 Return Values .....	371
B.67 REGISTER EVENT .....	374
B.68 REVOKE (Role from User) .....	374
B.68.1 Usage .....	374
B.68.2 Example .....	375
B.69 REVOKE (Privilege from Role or User) .....	375
B.69.1 Usage .....	375
B.69.2 Example .....	376
B.69.3 See Also .....	376
B.70 REVOKE REFRESH .....	376
B.70.1 Supported in .....	376
B.70.2 Usage .....	376
B.70.3 Usage in Master .....	376
B.70.4 Usage in Replica .....	376
B.70.5 Example .....	377
B.70.6 Return Values .....	377
B.71 ROLLBACK WORK .....	377
B.71.1 Usage .....	377
B.71.2 Example .....	377
B.72 SAVE .....	377
B.72.1 Supported in .....	377
B.72.2 Usage .....	378
B.72.3 Usage in Master .....	379
B.72.4 Usage in Replica .....	379
B.72.5 Example .....	379
B.72.6 Return Values .....	379
B.73 SAVE PROPERTY .....	379
B.73.1 Supported in .....	380
B.73.2 Usage .....	380
B.73.3 Usage in Master .....	381
B.73.4 Usage in Replica .....	381
B.73.5 Differences between "PUT_PARAM( )" and "SAVE PROPERTY property_name VALUE property_value;" .....	381
B.73.6 Example .....	381
B.73.7 Return Values .....	381
B.73.8 Result set .....	381
B.74 SELECT .....	382
B.74.1 Usage .....	382
B.74.2 Examples .....	383
B.74.3 START WITH Example .....	383
B.74.4 LEVEL and ORDER SIBLINGS BY Example .....	384

B.75 SET .....	384
B.75.1 Usage .....	384
B.75.2 Differences between SET and SET TRANSACTION .....	385
B.75.3 SET (Read/Write Level) .....	386
B.75.4 SET CATALOG .....	386
B.75.5 SET DURABILITY .....	386
B.75.6 SET ISOLATION LEVEL .....	387
B.75.7 SET SAFENESS .....	387
B.75.8 SET SCHEMA .....	388
B.75.9 SET SQL .....	389
B.75.10 SET STATEMENT MAXTIME .....	390
B.75.11 SET SYNC .....	390
B.75.12 SET TIMEOUT .....	402
B.75.13 SET TRANSACTION .....	402
B.76 START AFTER COMMIT .....	408
B.76.1 Usage .....	408
B.76.2 Transactions .....	409
B.76.3 Context of the Background Statements .....	409
B.76.4 Durability .....	409
B.76.5 Rollback .....	410
B.76.6 Order of Execution .....	410
B.76.7 Examples .....	410
B.77 TRUNCATE TABLE .....	411
B.77.1 Usage .....	411
B.78 UNLOCK TABLE .....	411
B.78.1 Usage .....	411
B.78.2 Examples of Using LOCK and UNLOCK .....	412
B.78.3 Return Values .....	413
B.78.4 See Also .....	413
B.79 UNREGISTER EVENT .....	413
B.80 UPDATE (Positioned) .....	413
B.80.1 Usage .....	413
B.80.2 Example .....	414
B.81 UPDATE (Searched) .....	414
B.81.1 Usage .....	414
B.81.2 Example .....	414
B.82 WAIT EVENT .....	414
B.83 Table_reference .....	414
B.84 Query_specification .....	416
B.85 Search_condition .....	417
B.86 Check_condition .....	419
B.87 Expression .....	420

B.88 String Functions .....	423
B.89 Numeric Functions .....	424
B.90 Date Time Functions .....	425
B.91 System Functions .....	427
B.92 Miscellaneous Functions .....	428
B.93 Data_type .....	428
B.94 Date and Time Literals .....	429
B.95 Pseudo Columns .....	430
B.96 Wildcard Characters .....	430
B.96.1 Using SQL Wildcards .....	431
B.96.2 Wildcard Characters as Literals .....	432
C Reserved Words .....	433
D Database System Tables and System Views .....	447
D.1 System Tables .....	447
D.1.1 SQL_LANGUAGES .....	447
D.1.2 SYS_ATTAUTH .....	447
D.1.3 SYS_BACKGROUNDJOB_INFO .....	448
D.1.4 SYS_BLOBS .....	448
D.1.5 SYS_CARDINAL .....	449
D.1.6 SYS_CATALOGS .....	449
D.1.7 SYS_CHECKSTRINGS .....	450
D.1.8 SYS_COLUMNS .....	450
D.1.9 SYS_COLUMNS_AUX .....	451
D.1.10 SYS_DL_REPLICA_CONFIG .....	452
D.1.11 SYS_DL_REPLICA_DEFAULT .....	452
D.1.12 SYS_EVENTS .....	453
D.1.13 SYS_FORKEYPARTS .....	453
D.1.14 SYS_FORKEYS .....	454
D.1.15 SYS_HOTSTANDBY .....	454
D.1.16 SYS_INFO .....	454
D.1.17 SYS_KEYPARTS .....	455
D.1.18 SYS_KEYS .....	455
D.1.19 SYS_PROCEDURES .....	456
D.1.20 SYS_PROCEDURE_COLUMNS .....	457
D.1.21 SYS_PROPERTIES .....	457
D.1.22 SYS_RELAUTH .....	458
D.1.23 SYS_SCHEMAS .....	458
D.1.24 SYS_SEQUENCES .....	459
D.1.25 SYS_SYNC_REPLICA_PROPERTIES .....	459
D.1.26 SYS_SYNONYM .....	459
D.1.27 SYS_TABLEMODES .....	460
D.1.28 SYS_TABLES .....	460

D.1.29	SYS_TRIGGERS .....	461
D.1.30	SYS_TYPES .....	462
D.1.31	SYS_URole .....	462
D.1.32	SYS_USERS .....	463
D.1.33	SYS_VIEWS .....	463
D.2	System Tables for Data Synchronization .....	463
D.2.1	SYS_BULLETIN_BOARD .....	464
D.2.2	SYS_PUBLICATION_ARGS .....	464
D.2.3	SYS_PUBLICATION_REPLICA_ARGS .....	464
D.2.4	SYS_PUBLICATION_REPLICA_STMTARGS .....	465
D.2.5	SYS_PUBLICATION_REPLICA_STMTS .....	465
D.2.6	SYS_PUBLICATION_STMTARGS .....	466
D.2.7	SYS_PUBLICATION_STMTS .....	466
D.2.8	SYS_PUBLICATIONS .....	467
D.2.9	SYS_PUBLICATIONS_REPLICA .....	467
D.2.10	SYS_SYNC_BOOKMARKS .....	468
D.2.11	SYS_SYNC_HISTORY_COLUMNS .....	468
D.2.12	SYS_SYNC_INFO .....	468
D.2.13	SYS_SYNC_MASTER_MSGINFO .....	469
D.2.14	SYS_SYNC_MASTER_RECEIVED_BLOB_REFS .....	470
D.2.15	SYS_SYNC_MASTER_RECEIVED_MSGPARTS .....	471
D.2.16	SYS_SYNC_MASTER_RECEIVED_MSGS .....	471
D.2.17	SYS_SYNC_MASTER_STORED_BLOB_REFS .....	471
D.2.18	SYS_SYNC_MASTER_STORED_MSGPARTS .....	472
D.2.19	SYS_SYNC_MASTER_STORED_MSGS .....	472
D.2.20	SYS_SYNC_MASTER_SUBSC_REQ .....	473
D.2.21	SYS_SYNC_MASTER_VERSIONS .....	473
D.2.22	SYS_SYNC_MASTERS .....	474
D.2.23	SYS_SYNC_RECEIVED_BLOB_ARGS .....	474
D.2.24	SYS_SYNC_RECEIVED_STMTS .....	475
D.2.25	SYS_SYNC_REPLICA_MSGINFO .....	475
D.2.26	SYS_SYNC_REPLICA_RECEIVED_BLOB_REFS .....	477
D.2.27	SYS_SYNC_REPLICA_RECEIVED_MSGPARTS .....	477
D.2.28	SYS_SYNC_REPLICA_RECEIVED_MSGS .....	478
D.2.29	SYS_SYNC_REPLICA_STORED_BLOB_REFS .....	478
D.2.30	SYS_SYNC_REPLICA_STORED_MSGS .....	478
D.2.31	SYS_SYNC_REPLICA_STORED_MSGPARTS .....	479
D.2.32	SYS_SYNC_REPLICA_VERSIONS .....	479
D.2.33	SYS_SYNC_REPLICAS .....	480
D.2.34	SYS_SYNC_SAVED_BLOB_ARGS .....	480
D.2.35	SYS_SYNC_SAVED_STMTS .....	480
D.2.36	SYS_SYNC_TRX_PROPERTIES .....	481

D.2.37 SYS_SYNC_USERMAPS .....	481
D.2.38 SYS_SYNC_USERS .....	482
D.3 System Views .....	482
D.3.1 COLUMNS .....	482
D.3.2 SERVER_INFO .....	483
D.3.3 TABLES .....	484
D.3.4 USERS .....	484
D.4 Synchronization-Related Views .....	484
D.4.1 SYNC_FAILED_MESSAGES .....	485
D.4.2 SYNC_FAILED_MASTER_MESSAGES .....	485
D.4.3 SYNC_ACTIVE_MESSAGES .....	486
D.4.4 SYNC_ACTIVE_MASTER_MESSAGES .....	486
E System Stored Procedures .....	489
E.1 Synchronization-Related Stored Procedures .....	489
E.1.1 SYNC_SETUP_CATALOG .....	489
E.1.2 SYNC_REGISTER_REPLICA .....	490
E.1.3 SYNC_UNREGISTER_REPLICA .....	491
E.1.4 SYNC_REGISTER_PUBLICATION .....	493
E.1.5 SYNC_UNREGISTER_PUBLICATION .....	494
E.1.6 SYNC_SHOW_SUBSCRIPTIONS .....	495
E.1.7 SYNC_SHOW_REPLICA_SUBSCRIPTIONS .....	496
E.1.8 SYNC_DELETE_MESSAGES .....	497
E.1.9 SYNC_DELETE_REPLICA_MESSAGES .....	498
E.2 Miscellaneous Stored Procedures .....	499
E.2.1 SYS_GETBACKGROUNDJOB_INFO .....	499
F System Events .....	501
F.1 Miscellaneous Events .....	502
F.2 Errors that Cause SYS_EVENT_ERROR .....	511
F.3 Conditions or Warnings that Cause SYS_EVENT_MESSAGES .....	512
F.4 HotStandby Events .....	513
F.5 Advanced Replication Synchronization Events .....	513
Glossary .....	515
Index .....	519

---

---

# List of Figures

4.1 Sync Pull Notify .....	117
5.1 Referential Constraint .....	153
5.2 Self-referential Constraint .....	154
7.1 Execution Graph 1 .....	189
7.2 Execution Graph 2 .....	190

---



---

# List of Tables

1.1	Typographic Conventions .....	2
1.2	Syntax Notation Conventions .....	3
2.1	Example Database Table .....	5
3.1	Example Database Table .....	13
4.1	Comparison Operators .....	45
4.2	Logical Operators: NOT .....	46
4.3	Logical Operators: AND .....	46
4.4	Logical Operators: OR .....	46
4.5	Determining Data Type from Parameters .....	68
4.6	Statement Atomicity in a Trigger .....	81
4.7	Insert/Update/Delete Operations for BEFORE/AFTER Triggers .....	89
4.8	Example Entry 1 .....	92
4.9	Example Entry 2 .....	93
4.10	Meta Data for the SYS_TRIGGERS System Table .....	104
5.1	Reserved User Names and Roles .....	140
5.2	Viewing Tables and Granting Access .....	144
5.3	Expressions and Operators .....	157
7.1	SQL Info Levels .....	184
7.2	EXPLAIN PLAN FOR Units .....	185
7.3	Explain Plan Table Columns .....	185
7.4	Texts in the Unit INFO Column .....	186
7.5	EXPLAIN PLAN FOR, Example 1 .....	188
7.6	EXPLAIN PLAN FOR, Example 2 .....	189
8.1	Diagnosing Poor Performance .....	206
A.1	Supported Data Types .....	209
A.2	Character Data Types .....	209
A.3	Numeric Data Types .....	210
A.4	Binary Data Types .....	213
A.5	Date Data Type .....	214
A.6	Time Data Type .....	214
A.7	Timestamp Data Type .....	214
A.8	Smallest Possible Non-Zero Numbers .....	215
B.1	ADMIN COMMAND Syntax .....	221
B.2	ALTER TABLE SET HISTORY COLUMNS Return Values .....	242
B.3	ALTER TABLE SET SYNCHISTORY Return Values .....	245
B.4	ALTER USER Return Values .....	248
B.5	Comparison of the Parameter Modes .....	263
B.6	Control Statements .....	264
B.7	CREATE PUBLICATION Return Values .....	281

B.8 CREATE SYNC BOOKMARK Return Values .....	288
B.9 Statement Atomicity in a Trigger .....	298
B.10 DROP MASTER Return Values .....	307
B.11 DROP PUBLICATION Return Values .....	308
B.12 DROP PUBLICATION REGISTRATION Return Values .....	310
B.13 DROP REPLICA Return Values .....	311
B.14 DROP SUBSCRIPTION Return Values .....	315
B.15 DROP SYNC BOOKMARK Return Values .....	317
B.16 EXPORT SUBSCRIPTION Return Values .....	322
B.17 EXPORT SUBSCRIPTION TO REPLICA Return Values .....	324
B.18 GET_PARAM Return Values .....	326
B.19 GRANT REFRESH Return Values .....	329
B.20 Hints .....	332
B.21 IMPORT Return Values .....	337
B.22 LOCK TABLE Return Values .....	343
B.23 MESSAGE APPEND Return Values .....	346
B.24 MESSAGE BEGIN Return Values from Replica .....	348
B.25 MESSAGE BEGIN Return Values from Master .....	348
B.26 MESSAGE DELETE Return Values from Replica .....	350
B.27 MESSAGE DELETE Return Values from Master .....	350
B.28 MESSAGE DELETE CURRENT TRANSACTION Return Values .....	352
B.29 MESSAGE END Return Values from Replica .....	354
B.30 MESSAGE END Return Values from Master .....	354
B.31 MESSAGE EXECUTE Return Values .....	356
B.32 MESSAGE FORWARD Return Values from Replica .....	359
B.33 MESSAGE FORWARD Return Values from Master .....	361
B.34 MESSAGE FROM REPLICA EXECUTE Return Values .....	363
B.35 MESSAGE GET REPLY Return Values from Replica .....	365
B.36 MESSAGE GET REPLY Return Values from Master .....	366
B.37 MESSAGE GET REPLY Result Set Table .....	367
B.38 PUT_PARAM() Return Values .....	370
B.39 REFRESH Return Values .....	372
B.40 REVOKE REFRESH Return Values .....	377
B.41 SAVE Return Values .....	379
B.42 SAVE PROPERTY Return Values .....	381
B.43 SET SYNC Return Values .....	392
B.44 SET SYNC CONNECT Return Values .....	393
B.45 How Different Operations Apply to Synchronization History Tables .....	395
B.46 SET SYNC MODE Return Values .....	395
B.47 SET SYNC NODE Return Values .....	398
B.48 SET SYNC PARAMETER Return Values .....	399
B.49 LOCK TABLE Return Values .....	413

B.50 Table_reference .....	414
B.51 Query_specification .....	416
B.52 Search_condition .....	417
B.53 Check_condition .....	419
B.54 Expression .....	420
B.55 String Functions .....	423
B.56 Numeric Functions .....	424
B.57 Date Time Functions .....	425
B.58 System Functions .....	427
B.59 Miscellaneous Functions .....	428
B.60 Data_type .....	428
B.61 Date and Time Literals .....	429
B.62 Pseudo Columns .....	430
B.63 Wildcard Characters .....	431
C.1 Reserved Words List .....	433
D.1 SQL_LANGUAGES .....	447
D.2 SYS_ATTAUTH .....	447
D.3 SYS_BACKGROUNDJOB_INFO .....	448
D.4 SYS_BLOBS .....	449
D.5 SYS_CARDINAL .....	449
D.6 SYS_CATALOGS .....	450
D.7 SYS_CHECKSTRINGS .....	450
D.8 SYS_COLUMNS .....	450
D.9 SYS_COLUMNS_AUX .....	451
D.10 SYS_DL_REPLICA_CONFIG .....	452
D.11 SYS_DL_REPLICA_DEFAULT .....	452
D.12 SYS_EVENTS .....	453
D.13 SYS_FORKEYPARTS .....	453
D.14 SYS_FORKEYS .....	454
D.15 SYS_INFO .....	454
D.16 SYS_KEYPARTS .....	455
D.17 SYS_KEYS .....	455
D.18 SYS_PROCEDURES .....	456
D.19 SYS_PROCEDURE_COLUMNS .....	457
D.20 SYS_PROPERTIES .....	458
D.21 SYS_RELAUTH .....	458
D.22 SYS_SCHEMAS .....	458
D.23 SYS_SEQUENCES .....	459
D.24 SYS_SYNC_REPLICA_PROPERTIES .....	459
D.25 SYS_SYNONYM .....	459
D.26 SYS_TABLEMODES .....	460
D.27 SYS_TABLES .....	461

D.28 SYS_TRIGGERS .....	461
D.29 SYS_TYPES .....	462
D.30 SYS_UROLE .....	462
D.31 SYS_USERS .....	463
D.32 SYS_VIEWS .....	463
D.33 SYS_BULLETIN_BOARD .....	464
D.34 SYS_PUBLICATION_ARGS .....	464
D.35 SYS_PUBLICATION_REPLICA_ARGS .....	464
D.36 SYS_PUBLICATION_REPLICA_STMTARGS .....	465
D.37 SYS_PUBLICATION_REPLICA_STMTS .....	465
D.38 SYS_PUBLICATION_STMTARGS .....	466
D.39 SYS_PUBLICATION_STMTS .....	466
D.40 SYS_PUBLICATIONS .....	467
D.41 SYS_PUBLICATIONS_REPLICA .....	467
D.42 SYS_SYNC_BOOKMARKS .....	468
D.43 SYS_SYNC_HISTORY_COLUMNS .....	468
D.44 SYS_SYNC_INFO .....	468
D.45 SYS_SYNC_MASTER_MSGINFO .....	469
D.46 SYS_SYNC_MASTER_RECEIVED_BLOB_REFS .....	470
D.47 SYS_SYNC_MASTER_RECEIVED_MSGPARTS .....	471
D.48 SYS_SYNC_MASTER_RECEIVED_MSGS .....	471
D.49 SYS_SYNC_MASTER_STORED_BLOB_REFS .....	471
D.50 SYS_SYNC_MASTER_STORED_MSGPARTS .....	472
D.51 SYS_SYNC_MASTER_STORED_MSGS .....	472
D.52 SYS_SYNC_MASTER_SUBSC_REQ .....	473
D.53 SYS_SYNC_MASTER_VERSIONS .....	473
D.54 SYS_SYNC_MASTERS .....	474
D.55 SYS_SYNC_RECEIVED_BLOB_ARGS .....	474
D.56 SYS_SYNC_RECEIVED_STMTS .....	475
D.57 SYS_SYNC_REPLICA_MSGINFO .....	476
D.58 SYS_SYNC_REPLICA_RECEIVED_BLOB_REFS .....	477
D.59 SYS_SYNC_REPLICA_RECEIVED_MSGPARTS .....	477
D.60 SYS_SYNC_REPLICA_RECEIVED_MSGS .....	478
D.61 SYS_SYNC_REPLICA_STORED_BLOB_REFS .....	478
D.62 SYS_SYNC_REPLICA_STORED_MSGS .....	478
D.63 SYS_SYNC_REPLICA_STORED_MSGPARTS .....	479
D.64 SYS_SYNC_REPLICA_VERSIONS .....	479
D.65 SYS_SYNC_REPLICAS .....	480
D.66 SYS_SYNC_SAVED_BLOB_ARGS .....	480
D.67 SYS_SYNC_SAVED_STMTS .....	481
D.68 SYS_SYNC_TRX_PROPERTIES .....	481
D.69 SYS_SYNC_USERMAPS .....	482

D.70 SYS_SYNC_USERS .....	482
D.71 COLUMNS .....	483
D.72 SERVER_INFO .....	483
D.73 TABLES .....	484
D.74 USERS .....	484
D.75 SYNC_FAILED_MESSAGES .....	485
D.76 SYNC_FAILED_MASTER_MESSAGES .....	485
D.77 SYNC_ACTIVE_MESSAGES .....	486
D.78 SYNC_ACTIVE_MASTER_MESSAGES .....	486
E.1 SYNC_SETUP_CATALOG Error Codes .....	490
E.2 SYNC_REGISTER_REPLICA Error Codes .....	491
E.3 SYNC_UNREGISTER_REPLICA Error Codes .....	492
E.4 SYNC_REGISTER_PUBLICATION Error Codes .....	493
E.5 SYNC_UNREGISTER_PUBLICATION Error Codes .....	494
E.6 CREATE PROCEDURE SYNC_SHOW_SUBSCRIPTIONS Result Set .....	495
E.7 SYNC_SHOW_SUBSCRIPTIONS Error Codes .....	496
E.8 SYNC_SHOW_REPLICA_SUBSCRIPTIONS Result Set .....	497
E.9 SYNC_SHOW_REPLICA_SUBSCRIPTIONS Error Codes .....	497
E.10 SYNC_DELETE_MESSAGES Error Codes .....	498
E.11 SYNC_DELETE_REPLICA_MESSAGES Error Codes .....	499
F.1 Miscellaneous Events .....	502
F.2 Errors that Cause SYS_EVENT_ERROR .....	512
F.3 Warnings that Cause SYS_EVENT_MESSAGES .....	512

---

---

## List of Examples

4.1 Trigger with Referencing Clause Example .....	84
4.2 Flawed Trigger .....	93
4.3 Trigger Example .....	96
4.4 Dropping and Recreating a Trigger .....	102
4.5 Sliced Replicas .....	118
4.6 Event Example 1 .....	125
4.7 Event Example 2 .....	129
4.8 Event Example 3 .....	132
B.1 SET Examples .....	386

---



---

# Chapter 1. Welcome

## 1.1 About This Guide

*IBM solidDB SQL Guide* introduces you to the relational database server theory and the SQL programming language. It also includes appendices that show the syntax of all SQL statement supported by IBM solidDB, and describes the data types that can be used in tables and SQL statements.

### 1.1.1 Organization

This guide contains the following chapters:

- Chapter 2, *Database Concepts*, familiarizes you with the basics of relational databases.
- Chapter 3, *Getting Started with SQL*, familiarizes you with the basics of SQL (Structured Query Language).
- Chapter 4, *Stored Procedures, Events, Triggers, and Sequences*, explains how to use programming features, including Stored Procedures, Triggers, etc.
- Chapter 5, *Using IBM solidDB SQL for Database Administration*, explains the use of SQL for tasks such as limiting access to particular users or roles, etc.
- Chapter 6, *Managing Transactions*, concentrates on transaction management.
- Chapter 7, *Diagnostics and Troubleshooting*, explains how to diagnose and solve some types of problems.
- Chapter 8, *Performance Tuning*, explains how to improve performance of SQL statements.
- Appendix A, *Data Types*, lists the valid SQL data types.
- Appendix B, *IBM solidDB SQL Syntax*, shows the syntax for each SQL statement accepted by the IBM solidDB.
- Appendix C, *Reserved Words*, lists the reserved words in SQL statements.
- Appendix D, *Database System Tables and System Views*, lists system tables and views.
- Appendix E, *System Stored Procedures*, lists stored procedures that are pre-defined by the server.
- Appendix F, *System Events*, lists events that are pre-defined by the server.
- Glossary provides definitions of terms.

## 1.1.2 Audience

This guide is for users who want to learn about SQL in general or who want to learn about IBM solidDB-specific SQL.

## 1.2 Conventions

### 1.2.1 Typographic Conventions

This manual uses the following typographic conventions:

**Table 1.1. Typographic Conventions**

<b>Format</b>	<b>Used for</b>
Database table	This font is used for all ordinary text.
NOT NULL	Uppercase letters on this font indicate SQL keywords and macro names.
<code>solid.ini</code>	These fonts indicate file names and path expressions.
<code>SET SYNC MASTER YES; COMMIT WORK;</code>	This font is used for program code and program output. Example SQL statements also use this font.
<b>run.sh</b>	This font is used for sample command lines.
<code>TRIG_COUNT( )</code>	This font is used for function names.
<code>java.sql.Connection</code>	This font is used for interface names.
<i>LockHashSize</i>	This font is used for parameter names, function arguments, and Windows registry entries.
<i>argument</i>	Words emphasised like this indicate information that the user or the application must provide.
<i>IBM solidDB Administration Guide</i>	This style is used for references to other documents, or chapters in the same document. New terms and emphasised issues are also written like this.
File path presentation	File paths are presented in the Unix format. The slash (/) character represents the installation root directory.
Operating systems	If documentation contains differences between operating systems, the Unix format is mentioned first. The Microsoft Windows format is mentioned in parentheses after the Unix format. Other operating systems are separately mentioned.

## 1.2.2 Syntax Notation

This manual uses the following syntax notation conventions:

**Table 1.2. Syntax Notation Conventions**

Format	Used for
<code>INSERT INTO <i>table_name</i></code>	Syntax descriptions are on this font. Replaceable sections are on <i>this</i> font.
<code>solid.ini</code>	This font indicates file names and path expressions.
<code>[ ]</code>	Square brackets indicate optional items; if in bold text, brackets must be included in the syntax.
<code> </code>	A vertical bar separates two mutually exclusive choices in a syntax line.
<code>{ }</code>	Curly brackets delimit a set of mutually exclusive choices in a syntax line; if in bold text, braces must be included in the syntax.
<code>...</code>	An ellipsis indicates that arguments can be repeated several times.
<code>. . . . .</code>	A column of three dots indicates continuation of previous lines of code.

## 1.3 IBM solidDB Documentation

Below is a complete list of documents available for IBM solidDB. IBM solidDB documentation is distributed in an electronic format, usually PDF files and web pages.

- *Release Notes*. This file contains installation instructions and the most up-to-date information about the specific product version. This file (`releasenotes.txt`) is copied onto your system when you install the software.
- *IBM solidDB Getting Started Guide*. This manual gives you an introduction to IBM solidDB.
- *IBM solidDB SQL Guide*. This manual describes the SQL commands that IBM solidDB supports. This manual also describes some of the system tables, system views, system stored procedures, etc. that the engine makes available to you. This manual contains some basic tutorial material on SQL for those readers

who are not already familiar with SQL. Note that some specialized material is covered in other manuals. For example, the IBM solidDB "administrative commands" related to the High Availability (HotStandby) component are described in the *IBM solidDB High Availability User Guide*, not the *IBM solidDB SQL Guide*.

- *IBM solidDB Administration Guide*. This guide describes administrative procedures for IBM solidDB servers. This manual includes configuration information. Note that some administrative commands use an SQL-like syntax and are documented in the *IBM solidDB SQL Guide*.
- *IBM solidDB Programmer Guide*. This guide explains in detail how to use features such as IBM solidDB Stored Procedure Language, triggers, events, and sequences. It also describes the interfaces (APIs and drivers) available for accessing IBM solidDB and how to use them with a IBM solidDB database.
- *IBM solidDB In-Memory Database User Guide*. This manual describes how to use the IBM solidDB in-memory database and main memory engine (MME).
- *IBM solidDB Advanced Replication Guide*. This guide describes how to use the IBM solidDB advanced replication technology to synchronize data across multiple database servers.
- *IBM solidDB Linked Library Access User Guide*. Linking the client application directly to the server improves performance by eliminating network communication overhead. This guide describes how to use the linked library access, a database engine library that can be linked directly to the client application.

This manual also explains how to use two proprietary Application Programming Interfaces (APIs). The first API is the IBM solidDB SA interface, a low-level C-language interface that allows you to perform simple single-table operations (such as inserting a row in a table) quickly. The second API is SSC API, which allows your C-language program can control the behavior of the embedded (linked) database server

This manual also explains how to set up a IBM solidDB to run without a disk drive.

- *IBM solidDB High Availability User Guide*. IBM solidDB HotStandby allows your system to maintain an identical copy of the database in a backup server or "secondary server". This secondary database server can continue working if the primary database server fails.
- *IBM solidDB Connector Guide*. This guide explains in detail how to use the IBM solidDB Cache solution. IBM solidDB Cache provides a high-performance, low-latency database front-end solution for IBM Data Servers, namely DB2™ and Informix™. IBM solidDB Cache solution uses a number of in-memory front-end databases to handle high-volume traffic from the applications. The connectors are applications that manage data between the back-end and the front-ends in IBM solidDB Cache.

---

# Chapter 2. Database Concepts

If you are not already familiar with relational database servers such as the IBM solidDB family, you may want to read this chapter.

This chapter explains the following concepts:

- Relational Databases
  - Tables, Rows, and Columns
  - Relating data in different tables
- Multi-User Capability / Concurrency Control and Locking
- Client-Server architecture
- Transactions
- Transaction Logging and Recovery

## 2.1 Relational Databases

### 2.1.1 Tables, Rows, and Columns

Most relational database servers, including the IBM solidDB family, use a programming language known as the Structured Query Language (SQL). SQL is a set-oriented programming language that is designed to allow people to query and update tables of information. This chapter discusses tables, and how data is represented within tables. Later in the manual, we will discuss the syntax of the SQL language in more detail.

All information is stored in tables. A table is divided into rows and columns. (SQL theorists refer to columns as "attributes" and rows as "tuples", but we will use the more familiar terms "columns" and "rows". We will also use the terms "record" and "row" interchangeably.) Each database contains 0 or more tables. Most databases contain many tables. An example of a table is shown below.

**Table 2.1. Example Database Table**

<b>ID</b>	<b>NAME</b>	<b>ADDRESS</b>
1	Beethoven	23 Ludwig Lane
2	Dylan	46 Robert Road

---

## 2.1.2 Relating Data In Different Tables

---

<b>ID</b>	<b>NAME</b>	<b>ADDRESS</b>
3	Nelson	79 Willie Way

This table contains 3 rows of data. (The top "row", which has the labels "ID", "NAME", and "ADDRESS" is shown here for the convenience of the reader. The actual table in the database does not have such a row.) The table contains 3 columns (ID, NAME, and ADDRESS).

SQL provides commands to create tables, insert rows into tables, update data in tables, delete rows from tables, and query the rows in tables.

Tables in SQL, unlike arrays in programming languages like C, are not homogeneous. In SQL one column may have one data type (such as INTEGER), while an adjacent column may have a very different data type (such as CHAR(20), which means an array of 20 characters).

A table may have varying numbers of rows. Rows may be inserted and deleted at any time; you do not need to pre-allocate space for a maximum number of rows. (All database servers have some maximum number of rows that they can handle. For example, most database servers that run on 32-bit operating systems have a limit of approximately two billion rows. In most applications, the maximum is far more than you are likely to need.)

Each row ("record") must have at least one value, or combination of values, that is unique. If we have two composers named David Jones to our table, and we need to update the address of only one of them, then we need some way to tell them apart. In some cases, you can find a combination of columns that is unique, even if you can't find any single column that contains unique values. For example, if the name column is not sufficient, then perhaps the combination of name and address will be unique. However, without knowing all the data ahead of time, it is difficult to absolutely guarantee that each value will be unique. Most database designers add an "extra" column that has no purpose other than to uniquely and easily identify each record. In our table above, for example, the ID numbers are unique. As you may have noticed, when we actually try to update or delete a record, we identify it by its unique ID (e.g. "... WHERE id = 1") rather than by using another value, such as name, that might not be unique.

## 2.1.2 Relating Data In Different Tables

If SQL could only handle one table at a time, it would be convenient, but not very powerful. The true power of SQL and relational databases lies in the fact that tables can be related to each other in useful ways, and SQL queries can gather data from multiple tables and display that data in a logical fashion.

We will show how multiple tables are useful by using a bank as an example.

Each customer of a bank may have more than 1 account. There is no real limit to the number of accounts a person might have. One customer might have a checking account, savings account, certificate of deposit, mortgage, credit card, etc. Furthermore, a person may have multiple accounts of the same type. For example,

a customer might have one savings account with retirement money and another savings account (of the same type) that has money for her daughter's college fund. We describe the "relationship" between a person and her accounts as a "one to many" relationship -- one person may have many accounts.

Because there is no limit to the number of accounts a person may have, there is no way to design a record structure ahead of time that can handle all possible combinations of accounts. And if you created a record structure that held the maximum number of accounts that anyone actually owned, you'd have to waste a lot of space. Let's suppose that we tried to build a single table that held all the information about one bank customer and her accounts. Our first draft might look like the following:

```
Customer ID Number
Customer Name
Customer Address
Checking Account #1 ID
Checking Account #1 Balance
CD #1 ID
CD #1 Balance
CD #2 ID
CD #2 Balance
...
```

As you can see, we just don't know when to stop because there is no obvious limit to the number of accounts that each person might own.

Another solution is to create multiple records, one for each account, and duplicate the customer information for each account. So we have a table that looks like:

```
Customer Name
Customer Address
Account ID
Account Balance
```

If a customer has more than one account, we merely create a complete record for each account. This works reasonably well, but it means that every single account record holds all the information about the customer. This wastes storage space and also makes it harder to update the customer's address if the customer moves (you may have to update the address in several places).

Relational databases, such as IBM solidDB's, are designed to solve this problem. We will create one table for customers, and another table for accounts. (In a real bank, we'd probably divide the accounts into multiple tables, too, with one table for checking accounts, another table for savings accounts, etc.) Then we create a

"link" between the customer and each of her accounts. This allows us to waste very little space and yet still have complete information available to us.

As we mentioned earlier, in our example of composers, every record should have a unique value that allows us to identify that record. The unique value is usually just an integer. We'll use that unique integer to help us "relate" a customer to her accounts. This is discussed in more detail in Chapter 3, *Getting Started with SQL*.

When we create an account for a customer, we store that customer's ID number as part of the account information. Specifically, each row in the accounts table has a `customer_id` value, and that `customer_id` value matches the id of the customer who owns that account. Smith has customer id 1, and each of Smith's accounts has a 1 in the `customer_id` field. That means that we can find all of Smith's account records by doing the following:

1. Look up Smith's record in the customers table.
2. When we find Smith's record, look at the id number in that record. (In Smith's case, the id is 1.)
3. Now look up all accounts in the accounts table that have a value of 1 in the `customer_id` field.

It's as though you taped a copy of your home telephone number onto the forehead of each of your children when they went to school. If there is an emergency and you need to send a taxi driver to find and pick up your children at school, you can simply tell the taxi driver your phone number and he can check every child in the school to see if the child has your phone number. (This isn't very efficient, but it works.) By knowing the parent's id number, you can identify all the children. Conversely, by knowing each child, you can identify the parent. If, for example, one of your children is lost on a field trip away from the school, any helpful person can simply read the telephone number off the child's forehead and call you.

As you can see, the parent and child are linked to each other without any sort of physical contact. Simply having the id number (or phone number) is enough to determine which children belong to a parent and which parent belongs to each child. The technique works regardless of how many children you have.

Relational databases use the same technique. Note that join operations are not limited to two tables. It's possible to create joins with an almost arbitrary number of tables. As a realistic extension of our banking example, we might have another table, "checks", which holds information about each check written. Thus we would have not only a 1-to-many relationship from each customer to her accounts, but also a 1-to-many relationship from each checking account to all of the checks written on that account. It's quite possible to write a query that will list all the checks that a customer has written, even if that customer has multiple checking accounts.



## 2.2 Client-Server Architecture

IBM solidDB uses the client-server model. In a client-server model, a single "server" may process requests from 1 or more "clients". This is quite similar to the way that a restaurant works — a single waiter and cook may handle requests from many customers.

In a client-server database model, the server is a specialized computer program that knows how to store and retrieve data efficiently. The server typically accepts four basic types of requests:

- Insert a new piece of information
- Update an existing piece of information
- Retrieve an existing piece of information
- Delete an existing piece of information

The server can store almost any type of data, but generally doesn't know the "meaning" of the data. The server typically knows little or nothing about "business issues", such as accounting, inventory, and so on. It doesn't know whether a particular piece of information is an inventory record, a description of a bank deposit, or a digitized copy of the song "American Pie".

The "clients" are responsible for knowing something about the particular business issues and about the "meaning" of the data. For example, we might write a client program that knows something about accounting. The client program might know how to calculate interest on late payments, for example. Or, the client might recognize that a particular piece of data is a song, and might convert the digital data to analog audio output.

Of course, it's possible to write a single program that does both the "client" and the "server" part of the work. A program that reads digitized music and plays it could also store that data to disk and look it up on request. However, it's not very efficient for every company to write its own data storage and retrieval routines. It is usually more efficient to buy an off-the-shelf data storage solution that is general enough to meet your needs, yet has relatively high performance.

## 2.3 Multi-User Capability

An important advantage of client-server architecture is that it usually makes it easier to have more than one client. IBM solidDB, like most relational database servers, will allow multiple users to access the data in a table.

Of course, when two users try to update the same data, there is potential danger. If the updates aren't the same, then one user's updates could write over the other user's updates. IBM solidDB uses concurrency control mechanisms to prevent this. For more information, see *IBM solidDB Administration Guide*.

## 2.4 Transactions

SQL allows you to group multiple statements into a single "atomic" (indivisible) piece of work called a transaction. For example, if you write a check to a grocery store, then the grocery store's bank account should be given the money at the same instant that the money is withdrawn from your account. It wouldn't make sense for you to pay the money without the grocery store receiving it, and it wouldn't make sense for the grocery store to be paid without your account having the money subtracted. If either of these operations (adding to the grocery store's account or subtracting from yours) fails, then the other one ought to fail, too. If both statements are in the same transaction, and either statement fails, then you can use the `ROLLBACK` command to restore things as they were before the transaction started — this prevents half-successful transactions from occurring. Naturally, if both halves of our financial transaction are successful, then we'd like our database transaction to be successful, too. Successful transactions are preserved with the command `COMMIT WORK`. Below is a simplistic example.

```
COMMIT WORK; -- Finish the previous transaction.
UPDATE stores SET balance = balance + 199.95
  WHERE store_name = 'Big Tyke Bikes';
UPDATE checking_accounts SET balance = balance - 199.95
  WHERE name = 'Jay Smith';
COMMIT WORK;
```

## 2.5 Transaction Logging and Recovery

One of the major advantages of buying a commercial database server is that most such servers have been designed to protect data if the database server shuts down unexpectedly for any reason, such as a power failure, a hardware failure, or a failure in the database software itself.

There are a number of different ways to help protect data. We will focus on one such way, called Transaction Logging.

### 2.5.1 Background

Suppose that you are writing data to a disk drive (or other permanent storage medium) and suddenly the power fails. The data that you write might not be written completely. For example, you might try to write the account balance "122.73", but because of the power failure you just write "12". The person whose account is missing some money will be quite displeased. How do we ensure that we always write complete data? Part of the solution is to use what is called a "transaction log".



### Note

In the world of computers, many different things are called "logs". For example, the IBM solidDB writes multiple log files, including a transaction log file and an error message log file. For the moment, we are discussing only the transaction log file.

As we mentioned previously, work is usually done in "transactions". An entire transaction is either committed or rolled back. No partial transactions are allowed. In the situation described here, where we started to write a person's new account balance to disk but lost power before we could finish, we'd like to roll back this transaction. Any transactions that were already completed and were correctly written to disk should, of course, be preserved.

To help us track what data has been written successfully and what data has not been written successfully, we actually write data to a "transaction log" as well as to the database tables. The transaction log is essentially a linear sequence of the operations that have been performed — that is, the transactions that have been committed. There are markers in the file to indicate the end of each transaction. If the last transaction in the file does not have an "end-of-transaction" marker, then we know that fractional transaction was not completed, and it should be rolled back rather than committed.

When the server re-starts after a failure, it reads the transaction log and applies the completed transactions one by one. In other words, it updates the tables in the database, using the information in the transaction log file. This is called "recovery". When done properly, recovery can even protect against power failures during the recovery process itself.

This is not a complete description of how transaction logging protects against data corruption. We have explained how the server makes sure that it doesn't lose transactions. But we haven't really explained how the server prevents the database file from becoming corrupted if a write failure occurs while the server is in the middle of writing a record to a table in the disk drive. That topic is more advanced and is not discussed here.

## 2.6 Summary

This brief introduction to relational databases has explained the concepts that you need to start using a relational database. You should now be able to answer the following questions:

What are tables, rows, and columns?

Can you work with data in more than one table at a time?

How do transactions help keep data consistent?

Why do we write ("log") transaction data to the disk drive?

---

---

# Chapter 3. Getting Started with SQL

This chapter gives you a quick overview (or refresher) in SQL.

## 3.1 Tables, Rows, and Columns

SQL is a set-oriented programming language that is designed to allow people to query and update tables of information.

All information is stored in tables. A table is divided into rows and columns. (SQL theorists refer to columns as "attributes" and rows as "tuples", but we will use the more familiar terms "columns" and "rows". We will also use the terms "record" and "row" interchangeably.) Each database contains 0 or more tables. Most databases contain many tables. An example of a table is shown below.

**Table 3.1. Example Database Table**

ID	NAME	ADDRESS
1	Beethoven	23 Ludwig Lane
2	Dylan	46 Robert Road
3	Nelson	79 Willie Way

This table contains three rows of data. (The top "row", which has the labels "ID", "NAME", and "ADDRESS" is shown here for the convenience of the reader. The actual table in the database does not have such a row.) The table contains three columns (ID, NAME, and ADDRESS). SQL provides commands to create tables, insert rows into tables, update data in tables, delete rows from tables, and query the rows in tables.

## 3.2 SQL

Below is a complete SQL "program" that creates the preceding table:

```
CREATE TABLE composers (id INTEGER PRIMARY KEY, name CHAR(20),  
address CHAR(50));  
INSERT INTO composers (id, name, address) VALUES (1, 'Beethoven',  
'23 Ludwig Lane');  
INSERT INTO composers (id, name, address) VALUES (2, 'Dylan',  
'46 Robert Road');
```

```
INSERT INTO composers (id, name, address) VALUES (3, 'Nelson',  
'79 Willie Way');
```

We designate the column "id" to be the "primary key" of the table. By doing this we are saying that each row may be uniquely identified by using this column. From now on, the system will guarantee that the value of "id" is unique and it always exists (i.e. it has the NOT NULL property)

If Mr. Dylan moves to 61 Bob Street, you can update his data with the command:

```
UPDATE composers SET ADDRESS = '61 Bob Street' WHERE ID = 2;
```

Because the ID field is unique for each composer, and because the WHERE clause in this command specifies only one ID, this update will be performed on only one composer.

If Mr. Beethoven dies and you need to delete his record, you can do so with the command:

```
DELETE FROM composers WHERE ID = 1;
```

Finally, if you would like to list all the composers in your table, you can use the command:

```
SELECT id, name, address FROM composers;
```

Note that the SELECT statement, unlike the UPDATE and DELETE statements listed above, did not include a WHERE clause. Therefore, the command applied to ALL records in the specified table. Thus the result of this SQL statement is to select (and list) all of the composers listed in the table.

ID	NAME	ADDRESS
1	Beethoven	23 Ludwig Lane
2	Dylan	46 Robert Road
3	Nelson	79 Willie Way

Note that although you entered the strings with quotes, they are displayed without quotes.

Even this simple series of commands helps show some important points about SQL.

- SQL is a relatively "high level" language. A single command can create a table with as many columns as you wish. Similarly, a single command can execute an UPDATE of almost any complexity. Although we didn't show it here, you can update multiple columns at a time, and you can even update more than one row at a time. Operations that might take dozens, or hundreds, of lines of code in languages like C or Java can be executed in a single SQL command.

- Unlike some other computer languages, SQL uses single quotes to delimit strings. For example, 'Beethoven' is a string. "Beethoven" is something different. (Technically, it is a delimited identifier, which we won't discuss in this chapter.) If you are used to programming languages like C, which use double quotes to delimit strings (character arrays) and single quotes to delimit individual characters, you will have to adjust to SQL's way of doing things.

Although the example above doesn't clearly show it, there are several additional points you need to know about basic SQL

- Although SQL is a very powerful high-level language, it is also a very limited one. SQL is designed for table-oriented and record-oriented operations. It has very few low-level operations. For example, there is no direct way to open a file, or to shift bits leftward or rightward. It is also hardware-independent, which is both an advantage and disadvantage. You have very little control over the format of the output from SQL queries; you may choose the order of the columns, and by using the ORDER BY clause you may control the order of the rows, but you can't do things such as control the size of the font on the screen, or print page numbers at the bottom of each printed page of output. SQL simply is not a complete programming language such as C, Java, PASCAL, etc.
- Each SQL implementation has a fixed set of data types. The data types in IBM solidDB (and most other implementations of SQL) include INTEGER, CHARACTER array, FLOATing point, DATE, and TIME.
- SQL is generally an "interpreted" language rather than a "compiled" language. To execute one or more SQL statements, you typically execute a separate program that reads your script and then executes it. No "compiled program" or "executable" is generated and stored for later use. Each time you run the program, it is interpreted again. (Stored Procedures can be re-used without necessarily re-interpreting them. Stored Procedures are discussed briefly in Appendix B, *IBM solidDB SQL Syntax* and extensively in Chapter 4, *Stored Procedures, Events, Triggers, and Sequences*.)
- Table and column names are case-insensitive in SQL. In our examples, keywords (such as CREATE, INSERT, SELECT) are capitalized, and table and columns names are shown in lower case. However, this is only a convention, not a requirement.
- SQL is also not very picky about whether commands are written on a single line or are split across multiple lines. We'll show examples of multi-line statements later in this chapter.
- SQL commands can get extremely complicated, with multiple nested "layers" of queries within queries. Figuring out how to write a complex query can be quite difficult - and figuring out how to understand a query that someone else wrote can be equally difficult. As in any programming language, it's a good idea to document your code!
- To help you document your code, SQL allows "comments". Comments are only for the human reader; they are skipped over by the SQL interpreter. To create a comment, put two dashes at the start of the line.

All the subsequent characters up to the end of the line will be ignored. (There is an exception for "optimizer hints", another advanced topic that we will not discuss in this chapter.)

## 3.3 The Mathematical Origins of SQL

Relational databases and SQL were originally based in part upon the mathematical concept of set theory. If you are familiar with set theory, it will help you understand how relational databases work. If you are not familiar with set theory, then don't worry about it; this is merely one way of looking at relational databases and SQL.

A table can be thought of as a mathematical set, where each element of the set is a row. (In our example above, each person, or composer, is an element of a set. The table contains all of the elements of the set 'composers'.) In mathematics, sets are unordered. Similarly, in SQL, tables are largely treated as unordered, even though of course if you could look at the bits and bytes on the disk you would find that at any given time the records are stored in a particular order.

This lack of ordering is important, because it means that the results of a query may be shown in a different order each time that you run the query. With small data sets stored on a single disk drive, you will usually see the same rows in the same order each time, but this is not necessarily the case when data is spread across multiple files or disk drives.

Because SQL is a set-oriented language, you can use it to perform some set-oriented operations, such as UNIONS (that is, combining two sets of input into one set of output). However, operations such as UNION require that the sets match each other - i.e. that they have the same number of columns, and that they have the same data type (or compatible data type) in corresponding columns. You can't perform a UNION operation if the first column in set1 is of type DATETIME and the first column in set2 is INTEGER, for example.

Again, if you are not comfortable with set theory, don't worry about it. This is just another way of looking at relational databases.

## 3.4 Creating Tables with Related Data

As described in the previous chapter, each customer of a bank may have more than one account. We describe the "relationship" between a person and her accounts as a "one to many" relationship — one person may have many accounts.

Because there is no limit to the number of accounts a person may have, there is no way to design a record structure ahead of time that can handle all possible combinations of accounts.

Relational databases, such as IBM Corporation's, are designed to solve this problem. We will create one table for customers, and another table for accounts. (In a real bank, we'd probably divide the accounts into multiple tables, too, with one table for checking accounts, another table for savings accounts, etc.) Then we create a



"link" between the customer and each of her accounts. This allows us to waste very little space and yet still have complete information available to us.

As we mentioned earlier, in our example of composers, every record should have a primary key that allows us to identify that record. It is usually just an integer. We'll now use that unique integer to help us "relate" a customer to her accounts. Below are the commands to create and populate the customer table:

```
CREATE TABLE customers (id INTEGER PRIMARY KEY, name CHAR(20),
address CHAR(40));
INSERT INTO customers (id, name, address) VALUES (1, 'Smith',
'123 Main Street');
INSERT INTO customers (id, name, address) VALUES (2, 'Jones',
'456 Fifth Avenue');
```

We have inserted two customers, named Smith and Jones. Let us create the account table:

```
CREATE TABLE accounts (id INTEGER PRIMARY KEY, balance FLOAT,
customer_id INT REFERENCES customers);
```

Here, we have designated the column **customer\_id** to be a "foreign key" pointing to the customer table (this is indicated by the REFERENCES keyword). The value of this column is supposed to be exactly the same as the "id" value (the primary key) in the corresponding customer row in the "customers" table. This way we will associate account rows with customer rows. The feature of a database allowing to maintain such relationships in a reliable way is called "referential integrity", and the corresponding SQL syntax elements used to define such relationships are called "referential integrity constraints". For more on referential integrity, see Section 5.5, "Referential Integrity".

Customer Smith has two accounts, and customer Jones has 1 account.

```
INSERT INTO accounts (id, balance, customer_id)
VALUES (1001, 200.00, 1);
INSERT INTO accounts (id, balance, customer_id)
VALUES (1002, 5000.00, 1);
INSERT INTO accounts (id, balance, customer_id)
VALUES (1003, 222.00, 2);
```

As Smith has two accounts, each of Smith's accounts has a 1 in the **customer\_id** field. That means that a user can find all of Smith's account records by doing the following:

1. Look up Smith's record in the customers table.

2. When we find Smith's record, look at the id number in that record. (In Smith's case, the id is 1.)
3. Now look up all accounts in the accounts table that have a value of 1 in the **customer\_id** field.

It's as though you taped a copy of your home telephone number onto the forehead of each of your children when they went to school. If there is an emergency and you need to send a taxi driver to find and pick up your children at school, you can simply tell the taxi driver your phone number and he can check every child in the school to see if the child has your phone number. (This isn't very efficient, but it works.) By knowing the parent's id number, you can identify all the children. Conversely, by knowing each child, you can identify the parent. If, for example, one of your children is lost on a field trip away from the school, any helpful person can simply read the telephone number off the child's forehead and call you.

As you can see, the parent and child are linked to each other without any sort of physical contact. Simply having the id number (or phone number) is enough to determine which children belong to a parent and which parent belongs to each child. The technique works regardless of how many children you have.

Relational databases use the same technique. Now that we've created our customer table and our accounts table, we can show each customer and each of the accounts that she has. To do this, we use what SQL programmers call a "join" operation. The WHERE clause in the SELECT statement "joins" those pairs of records where the account's **customer\_id** number matches the customer's id number.

```
SELECT name, balance
  FROM customers, accounts
 WHERE accounts.customer_id = customers.id;
```

The output of this query is similar to the following:

```
NAME  BALANCE
Smith  200.0
Smith  5000.0
Jones  222.0
```

Of course, if a person has multiple accounts, she might want to know the total amount of money that she has in all accounts. The computer can provide this information by using the following query:

```
SELECT customers.id, SUM(balance)
  FROM customers, accounts
```

```
WHERE accounts.customer_id = customers.id
GROUP BY customers.id;
```

The output of this query is similar to the following:

```
NAME  BALANCE
Smith  5200.0
Jones  222.0
```

Note that this time, Smith appears only once, and she appears with the total amount of money in all her accounts.

This query uses the `GROUP BY` clause and an aggregate function named `SUM()`. The topic of `GROUP BY` clauses is more complex than we want to go into during this simple introduction to SQL. This query is just to give you a little taste of the type of useful work that SQL can do in a single statement. Getting the same result in a language like C would take many statements.

Note that join operations are not limited to two tables. It's possible to create joins with an almost arbitrary number of tables. As a realistic extension of our banking example, we might have another table, "checks", which holds information about each check written. Thus we would have not only a 1-to-many relationship from each customer to her accounts, but also a 1-to-many relationship from each checking account to all of the checks written on that account. It's quite possible to write a query that will list all the checks that a customer has written, even if that customer has multiple checking accounts.

## 3.4.1 Table Aliases

SQL allows you to use an "alias" in place of a table name in some queries. In some cases, aliases are merely an optional convenience. In some queries, however, aliases are actually required (for reasons we won't explain here). We'll introduce the topic of aliases here because they are required for some examples later in this chapter. The query below is the same as an earlier query, except that we've added the table alias "a" for the accounts table and "c" for the customers table.

```
SELECT name, balance
FROM customers c, accounts a
WHERE a.customer_id = c.id;
```

As you can see, we defined an alias in the "FROM" clause and then used it elsewhere in the query (in the WHERE clause in this case).

## 3.4.2 Subqueries

SQL allows one query to contain another query, called a "subquery".

Returning to our bank example, over time, some customers add accounts and other customers terminate accounts. In some cases, a customer might gradually terminate accounts until he has no more accounts. Our bank may want to identify all customers that don't have any accounts so that those customers' records can be deleted, for example. One way to identify the customers who don't have any accounts is to use a subquery and the EXISTS clause.

Of course, to try this out, we need to create a customer who doesn't have any accounts:

```
INSERT INTO customers (id, name, address) VALUES (3, 'Zu', 'B St');
```

Before we list all customers who don't have accounts, let's list all customers who do have accounts.

```
SELECT id, name
FROM customers c
WHERE EXISTS (SELECT * FROM accounts a WHERE a.customer_id = c.id);
```

The subquery (also called the "inner query") is the query inside the parentheses. The inner query is executed once for each record selected by the outer query. (This functions a lot like nested loops would function in another programming language, except that with SQL we can do nested loops in a single statement.) Naturally, if there are any accounts for the particular customer that the outer loop is processing, then those account records are returned to the outer query.

The "EXISTS" clause in the outer query says, effectively, "We don't care what values are in those records; all we care about is whether there are any records or not." Thus EXISTS returns TRUE if the customer has any accounts. If the customer has no accounts, then the EXISTS returns false. The EXISTS clause doesn't care whether there are multiple accounts or single accounts. It doesn't care what values are in the accounts. All the EXISTS wants to know is "Is there at least one record?"

Thus, the entire statement lists those customers who have at least one account. No matter how many accounts the customer has (as long as it's at least 1), the customer is listed only once.

Now let's list all those customers who don't have any accounts:

```
SELECT id, name
```

```
FROM customers c
WHERE NOT EXISTS (SELECT * FROM accounts a WHERE a.customer_id = c.id);
```

Merely adding the keyword NOT reverses the sense of the query.

Subqueries may themselves have subqueries. In fact, subqueries may be nested almost arbitrarily deep.

## 3.5 Which Formats Are Used for Each Data Type?

As we've already shown above, SQL requires that values be expressed in a particular way. For example, character strings must be delimited by single quote marks.

Other values also must be formatted properly. The exact format required depends upon the data type. Several data types other than CHARACTER data types also require single quotes to delimit the values that you enter.

Below are some examples of how to format input data for most of the data types that IBM solidDB supports. We'll show this in the form of a simple SQL script that you can execute if you wish. Note that in this script, many commands are split across multiple lines. This is quite legal in SQL. It's one of the reasons that most SQL interpreters expect a semicolon to separate each SQL statement, even though the ANSI Standard for SQL doesn't actually require a semicolon at the end of each statement.

```
CREATE TABLE one_of_almost_everything (
  int_col INTEGER,
  float_col FLOAT,
  string_col CHAR(20),
  wide_string_col WCHAR(20), -- "wide" means wide chars, e.g. unicode.
  varchar_col VARCHAR, -- Note that we did not have to specify width.
  date_col DATE,
  time_col TIME,
  timestamp_col TIMESTAMP
);
```

```
INSERT INTO one_of_almost_everything (
  int_col,
  float_col,
  string_col,
  wide_string_col,
  varchar_col,
  date_col,
  time_col,
  timestamp_col
```

```
)  
VALUES (  
  1,  
  2.0,  
  'three',  
  'four',  
  'five point zero zero zero zero zero zero zero zero zero zero ...',  
  '2002-12-31',  
  '11:59:00',  
  '1999-12-31 23:59:59.00000'  
);
```

As you can see, timestamp values are entered in order from the "most significant" digit to the "least significant" digit. Similarly, date and time values are also entered from the most significant digit to the least significant digit. And all 3 of these data types (timestamp, date, time) use punctuation to separate individual fields.

The reason for requiring particular formats is that some of the other possible formats are ambiguous. For example, to someone in the U.S., '07-04-1776' is July 4, 1776, since Americans usually write dates in the 'mm-dd-yyyy' (or 'mm/dd/yyyy' format). But to a person from Europe, this date is obviously April 7, not July 4th, since most Europeans write dates in the format 'dd-mm-yyyy'. Although it may seem that the problem of having too many formats is not well solved by adding still another format, there are some advantages to SQL's approach of using a format that starts with the most significant digit and moves steadily toward the least significant digit. First, it means that all three data types (date, time, and timestamp) follow the same rule. Second, the date format and the time format are both perfect subsets of the timestamp format. Third, although it's yet another format to memorize, the rule is reasonably simple and is consistent with the way that "western" languages write numbers (most significant digit is furthest to the left). Finally, by being obviously incompatible with the existing formats, there's no chance that a person will accidentally write one date (e.g. '07-04-1776') and have it interpreted by the machine as another date.

## 3.5.1 BLOBs (or Binary Data Types)

So far, we have discussed data types that store data that is intended to be read by humans. Some types of data are not intended to be read directly by humans, but can still be stored in a database. For example, a picture from a digital camera, or a song from a CD, is stored as a series of numbers. These numbers are almost meaningless to a human. Digitized pictures and sounds can be stored as BINARY data, however. IBM solidDB supports three binary data types: BINARY, VARBINARY, and LONG VARBINARY (or BLOB).

In most cases, you will read and write binary data using the ODBC (Open DataBase Connectivity) API from a C program, or the JDBC API from a Java program. However, it is possible to insert data into a binary field using a utility that executes SQL statements. To insert a value into a binary field, you must represent the value as a series of hexadecimal numbers inside single quotes. For example, if you wanted to insert a series of bytes with the values 1, 9, 11, 255 into a binary field, you would execute:

```
INSERT INTO table1 (binary_col) VALUES (CAST('01090BFF' AS VARBINARY));
```

Because this command instructs the server to CAST the value to type VARBINARY, the server automatically interprets the string as a series of hexadecimal numbers, not as a string literal.

You may also insert a string literal directly, e.g.

```
INSERT INTO table1 (binary_col) VALUES ('Thank you');
```

When you retrieve the data via solsql (IBM solidDB utility for executing SQL statements), the return value from a binary column is expressed in hexadecimal, whether or not you originally entered it as hexadecimal. Thus, after you insert the value 'Thank you', if you select this value from the table you will see:

```
5468616E6B20796F75
```

where 54 represents capital 'T', 68 represents lower-case 'h', 61 represents lower-case 'a', 6E represents lower-case 'n', etc.

Note also that for long values only the first several digits are shown.

## 3.5.2 NULL IS NOT NULL (or "How to say 'None of the above' in SQL")

Sometimes you don't have enough information to fill out a form completely. SQL uses the keyword NULL to represent "Unknown" or "No Value". (This is different from the meaning of NULL in programming languages such as C.) For example, if we are inserting a record for Joni Mitchell into our table of composers, and we don't know Joni Mitchell's address, then we might execute the following:

```
INSERT INTO composers (id, name, address) VALUES (5, 'Mitchell', NULL);
```

If we don't specify the address field, it will contain NULL by default.

```
INSERT INTO composers (id, name) VALUES (5, 'Mitchell');
```

To give you some information about NULL, and also give you some practice reading SQL code, we've written our explanation of NULL as a sample program with comments. You can read this now. When you're ready to run it, simply cut and paste part or all of it into a program that executes SQL, such as the solsql utility provided with the IBM solidDB Development Kit. (For more information about solsql, see *IBM solidDB Administration Guide*.)

### 3.5.2 NULL IS NOT NULL (or "How to say 'None of the above' in SQL")

---

```
-- This sample script shows some unusual characteristics
-- of the value NULL.

-- Data of any data type may contain NULL.
-- For example, a column of type INTEGER may contain not
-- only valid integer values, but also NULL.

-- Set up for experiments...
CREATE TABLE table1 (x INTEGER, name CHAR(30));

-- The value NULL means "there is no value".
-- NULL is not the same as zero, or an empty string.
-- (It's also not a pointer value, as it is in
-- programming languages such as C.)
-- To help show this, we'll insert 3 rows, one of which has
-- "normal" values, one of which has a 0 and an empty string,
-- and one of which has two NULL values.
INSERT INTO table1 (x, name) VALUES (2, 'Ludwig Von Beethoven');
INSERT INTO table1 (x, name) VALUES (0, '');
INSERT INTO table1 (x, name) VALUES (NULL, NULL);
-- This returns only the row containing 0,
-- not the row containing NULL.
SELECT * FROM table1 WHERE x = 0;
-- This returns only the row containing the empty string,
-- not the row containing NULL.
SELECT * FROM table1 WHERE name = '';

-- It's not surprising that NULL doesn't match other values.
-- What IS surprising is that NULL doesn't match even itself.
-- (A mathematician would say that NULL violates the
-- reflexive property "a = a"!)
```

```
SELECT * FROM table1 WHERE x = x;

-- Since NULL doesn't equal NULL, what will the following query return?
SELECT * FROM table1 WHERE x != x;

-- Similarly, although you might think that the
-- expression below is always true, it's actually
-- always false.
```



```
SELECT * FROM table1 WHERE NULL IN (NULL, 2);

-- The result set will contain 2 (since 2 is in
-- the set (NULL, 2)), but the result set will
-- not contain NULL.
SELECT * FROM table1 WHERE x IN (NULL, 2);

-- But suppose that I *want* to find all the records that
-- have NULL values.  How do I do that if I can't say ... = NULL?
SELECT * FROM table1 WHERE x IS NULL;
-- And the opposite query is ...
SELECT * FROM table1 WHERE x IS NOT NULL;

-- Set up for more experiments...
CREATE TABLE parent (id INTEGER, name CHAR(20));
CREATE TABLE children (id INTEGER, name CHAR(12), parent_id INT);
INSERT INTO parent (id, name) VALUES (1, 'Smith');
INSERT INTO children (id, name, parent_id) VALUES (11, 'Smith child', 1);
INSERT INTO children (id, name, parent_id) VALUES (131, 'orphan', NULL);
INSERT INTO parent (id, name) VALUES (NULL, 'Has Null');

-- Since NULL != NULL, if a "parent" record has NULL and a "child"
-- record has NULL, the child's value won't match the parent's value.
-- This result set will contain 'Smith', but not 'Has Null'.
SELECT p.name FROM parent p, children c
  WHERE c.parent_id = p.id;

-- Note that a row that contains nothing but a
-- single NULL is still a row.
-- In the following query, we use an EXISTS clause,
-- which evaluates to TRUE if the subquery returns
-- any rows.  Even a row that contains nothing but a
-- single NULL value is still a row, and so if the
-- subquery returns a single NULL the EXISTS clause
-- still evaluates to TRUE.
-- Even though the subquery below returns NULL rather than a name
-- or ID, the EXISTS expression evaluates to TRUE, and Smith is printed.
```

```
SELECT name FROM parent p
  WHERE EXISTS(SELECT NULL FROM children c WHERE c.parent_id = p.id);
```

```
-- Now that we've trained you to recognize that NULL != NULL,
-- we'll confuse you with something that breaks the pattern.
-- Contrary to what you might expect, the UNIQUE keyword
-- DOES filter out multiple NULL values.
INSERT INTO table1 (x, name) VALUES (NULL, 'any name');
-- Now the table has more than one row in which x is NULL,
-- but a query with UNIQUE nonetheless returns only a
-- single NULL value.
SELECT DISTINCT x FROM table1;
-- You may be interested to know that a UNIQUE index
-- will allow only a single NULL value. (Note that a primary key
-- will not allow any NULL values.)
```

```
-- Clean up.
DROP TABLE parent;
DROP TABLE children;
DROP TABLE table1;
```

## 3.5.3 NOT NULL

As opposed to NULL, NOT NULL is one of the SQL data constraints. NOT NULL indicates that null values are not allowed in any row of the table for the specified column. For more information and examples, refer to Appendix B, *IBM solidDB SQL Syntax*.

## 3.5.4 Expressions and Casts

SQL allows expressions in some parts of SQL statements. For example, the following statement multiplies the value in a column by 12:

```
SELECT monthly_average * 12 FROM table1;
```

As another example, the following statement uses the built-in SQRT function to calculate the square root of each value in the column named "variance".

```
SELECT SQRT(variance) FROM table1;
```

Our next example uses the "REPLACE" function to convert numbers from U.S. format to European format. In U.S. format, numbers use the period character ('.') as the decimal point, but in Europe the comma (',') is used. For example, in the U.S. the approximation of pi is written as "3.14", while in Europe it is written as "3,14". We can use the REPLACE function to replace the '.' character with the ',' character. The following series of statements shows an example of this.

```
CREATE TABLE number_strings (n VARCHAR);
INSERT INTO number_strings (n) VALUES ('3.14'); -- input in US format.
SELECT REPLACE(n, '.', ',') FROM number_strings; -- output in European.
```

The output of course looks like

```
n
-----
3,14
```

Note that one function can call another. The following expression takes the square root of a number and then takes the natural log of that square root:

```
SELECT LOG(SQRT(x)) FROM table1;
```

IBM solidDB SQL does not accept completely general expressions in all clauses. For example, in the SELECT clause, you may use pre-defined functions, but you may not call stored procedures that you have created. Even if you have created a stored procedure named "foo", the following will not work:

```
SELECT foo(column1) FROM table1;
```

When you use expressions, you may want to specify a new name for a column. For example, if you use the expression

```
SELECT monthly_average * 12 FROM table1;
```

you probably don't want the output column to be called "monthly\_average". IBM solidDB server will actually use the expression itself as the name of the column. In this case, the name of the column would be "monthly\_average \* 12". That's certainly descriptive, but for a long expression this can get very messy. You

can use the "AS" keyword to give an output column a specific name. In the following example, the output will have the column heading "yearly\_average".

```
SELECT monthly_average * 12 AS yearly_average FROM table1;
```

Note that the AS clause works for any output column, not just for expressions. If you like, you may do something like the following:

```
SELECT ssn AS SocialSecurityNumber FROM table2;
```

A CASE clause allows you to control the output based on the input. Below is a simple example, which converts a number (1-12) to the name of a month:

```
CREATE TABLE dates (m INT);
INSERT INTO dates (m) VALUES (1);
-- ...etc.
INSERT INTO dates (m) VALUES (12);
INSERT INTO dates (m) VALUES (13);

SELECT
    CASE m
        WHEN 1 THEN 'January'
        -- etc.
        WHEN 12 THEN 'December'
        ELSE 'Invalid value for month'
    END
    AS month_name
FROM dates;
```

Note that this not only allows you to convert valid values, but also allows you to generate appropriate output if there is an error. The "ELSE" clause allows you to specify an alternative value if you get an input value that you weren't expecting.

In some situations, you may want to cast a value to a different data type. For example, when inserting BLOB data, it is convenient to create a string that contains your data, and then insert that string into a BINARY column. You may use a cast as shown below:

```
CREATE TABLE table1 (b BINARY(4));
INSERT INTO table1 VALUES ( CAST('FF00AA55' AS BINARY));
```

This cast allows you to take data that is a series of hexadecimal digits and input it as though it were a string. Each of the hexadecimal pairs in the quoted string represents a single byte of data. There are 8 hexadecimal digits, and thus 4 bytes of input.

A cast can be used to change output as well as input. In the rather complex code sample below, the expression in the CASE clause converts the output from the format '2003-01-20 15:33:40' to '2003-Jan-20 15:33:40'.

```
CREATE TABLE sample1(dt TIMESTAMP);
COMMIT WORK;

INSERT INTO sample1 VALUES ('2003-01-20 15:33:40');
COMMIT WORK;

SELECT
    CASE MONTH(dt)
        WHEN 1 THEN REPLACE(CAST(dt AS varchar), '-01-', '-Jan-')
        WHEN 2 THEN REPLACE(CAST(dt AS varchar), '-02-', '-Feb-')
        WHEN 3 THEN REPLACE(CAST(dt AS varchar), '-03-', '-Mar-')
        WHEN 4 THEN REPLACE(CAST(dt AS varchar), '-04-', '-Apr-')
        WHEN 5 THEN REPLACE(CAST(dt AS varchar), '-05-', '-May-')
        WHEN 6 THEN REPLACE(CAST(dt AS varchar), '-06-', '-Jun-')
        WHEN 7 THEN REPLACE(CAST(dt AS varchar), '-07-', '-Jul-')
        WHEN 8 THEN REPLACE(CAST(dt AS varchar), '-08-', '-Aug-')
        WHEN 9 THEN REPLACE(CAST(dt AS varchar), '-09-', '-Sep-')
        WHEN 10 THEN REPLACE(CAST(dt AS varchar), '-10-', '-Oct-')
        WHEN 11 THEN REPLACE(CAST(dt AS varchar), '-11-', '-Nov-')
        WHEN 12 THEN REPLACE(CAST(dt AS varchar), '-12-', '-Dec-')
    END
    AS formatted_date
FROM sample1;
```

This takes a value from a column named dt, converts that value from timestamp to VARCHAR, then replaces the month number with an abbreviation for the month (for example, it replaces "-01-" with "-Jan-"). By using the CASE/WHEN/END syntax, we can specify exactly what output we want for each possible input. Note that because this expression is so complicated, it is almost mandatory to use an AS clause to specify the column header in the output.

## 3.5.5 Row Value Constructors

This section explains one of the less familiar types of expressions, the Row Value Constructor (RVC), and how it is used with relational operators, such as greater than, less than, etc.

A row value constructor is an ordered sequence of values delimited by parentheses, for example:

```
(1, 4, 9)
('Smith', 'Lisa')
```

You can think of this as constructing a row based on a series of elements/values, just like a row of a table is composed of a series of fields.

Row value constructors, like individual values, may be used in comparisons. For example, just as you may have expressions like:

```
WHERE x > y;
WHERE 2 > 1;
```

you also may have expressions like:

```
WHERE (2, 3, 4) > (1, 2, 3);
WHERE (t1.last_name, t1.first_name) = (t2.last_name, t2.first_name);
```

Comparisons using row value constructors must be done carefully. Rather than give the technical definition of comparisons (which you can find in section 8.2 (comparison predicates) of the SQL-92 standard), we will give examples and an analogy to help you see the pattern.

The following expressions are true:

```
(9, 9, 9) > (1, 1, 1)
('Baker', 'Barbara') > ('Alpert', 'Andy')
(1, 1) = (1, 1)
(3, 2, 1) != (4, 3, 2)
```

The examples above are simple, because the expression is correct for each corresponding pair of elements and is therefore true for the RVCs. For example,

```
'Baker' > 'Alpert' and 'Barbara' > 'Andy', and therefore  
( 'Baker', 'Barbara' ) > ( 'Alpert', 'Andy' )
```

However, when comparing row value constructors, it is not necessary that the expression be true for each corresponding element. In a row value constructor, the further left an element is, the more significance it has. Thus the following expressions are also true:

```
( 9, 1, 1 ) > ( 1, 9, 9 )  
( 'Zoomer', 'Andy' ) > ( 'Alpert', 'Zelda' )
```

In these examples, since the most significant element of the first RCV is greater than the corresponding element of the second RCV, the expression is true, regardless of the values of the remaining elements. Similarly, in the examples below, the first elements are identical, but the expressions overall are true:

```
( 1, 1, 2 ) > ( 1, 1, 1 )  
( 1, 2, 1 ) > ( 1, 1, 1 )  
( 'Baker', 'Zelda' ) > ( 'Baker', 'Allison' )
```

Again, in a row value constructor, the further left an element is, the more significance it has. This is similar to the way that we compare multi-digit numbers. In a 3-digit number, such as 911, the hundreds-place digit is more significant than the tens-place digit, and the tens-place digit is more significant than the ones-place digit. Thus, the number 911 is greater than the number 199, even though not all digits of 911 are greater than the corresponding digits of 199.

This is useful when comparing multiple columns that are related. A practical application of this is when comparing people's names. For example, suppose that we have 2 tables, each of which has an **lname** (last name) and **fname** (first name) column. Suppose that we want to find all people whose names are less than Michael Morley's. In this situation, we want the last name to have more significance than the first name. The following names are shown in the correct alphabetical order (by last name):

Adams, Zelda

Morley, Michael

Young, Anna

If we want to list all persons whose names are less than Michael Morley's, then we do NOT want to use the following:

```
table1.lname < 'Morley' and table1.fname < 'Michael'
```

If we used this expression, we would reject Zelda Adams because her first name is alphabetically after Michael Morley's first name. One correct solution is to use the row value constructor approach:

```
(table1.lname, table1.fname) < ('Morley', 'Michael')
```

Note that when using equality, the expression must be true for ALL elements of the RCVs. E.g.:

```
(1, 2, 3) = (1, 2, 3)
```

Not surprisingly, for inequality the expression must be true for only one element:

```
(1, 2, 1) != (1, 1, 1)
```

## 3.6 More about Transactions

As described in the previous chapter, SQL allows you to group multiple statements into a single "atomic" (indivisible) piece of work called a transaction. Successful transactions are preserved with the command COMMIT WORK. Below is a simplistic example.

```
COMMIT WORK; -- Finish the previous transaction.
UPDATE stores SET balance = balance + 199.95
  WHERE store_name = 'Big Tyke Bikes';
UPDATE checking_accounts SET balance = balance - 199.95
  WHERE name = 'Jay Smith';
COMMIT WORK;
```

If you don't want to keep a particular transaction, you can roll it back by using the command:

```
ROLLBACK WORK;
```

If you do not explicitly commit or roll back your work, then the server will roll it back for you. In other words, unless you confirm that you want to keep the data (by committing it), the data will be discarded.



## 3.7 Summary

This brief introduction to SQL and relational databases has explained the concepts that you need to start using SQL. You should now be able to answer the following questions:

What are tables, rows, and columns?

How do I create a table?

How do I put data into a table?

How do I update data in a table?

How do I delete data from a table?

How do I list data in a table?

How do I list related data in two different tables?

How do I ensure that multiple statements are executed together (so that all fail or all succeed as a group)?

## 3.8 Where to Find Additional Information about SQL

Other chapters in this manual explain more about SQL and IBM solidDB -specific features. However, this manual is neither a complete tutorial nor a comprehensive reference on SQL. You may wish to acquire additional documents on SQL.

There are many books on SQL. These books are not specific to IBM solidDB's implementation of SQL; most of the material is generic and will apply to any database server, such as IBM solidDB's, that conforms to the ANSI standards. General SQL books include:

- *Introduction to SQL: Mastering the Relational Database Language*, by Rick van der Lans, published by Addison-Wesley.

ANSI standards on SQL include:

- Database Language - SQL with Integrity Enhancement, ANSI, 1989 ANSI X3.135-1989.
- Database Language - SQL: ANSI X3H2 and ISO/IEC JTC1/SC21/WG3 9075:1992 (SQL-92).

You may purchase ANSI standards from [www.ansi.org](http://www.ansi.org).

ISO (International Standards Organization) also has standards for SQL. See [www.iso.org](http://www.iso.org) for a list of standards and prices.

---

# Chapter 4. Stored Procedures, Events, Triggers, and Sequences

In IBM solidDB databases, a number of features are available that make it possible to move parts of the application logic into the database. These features include:

- stored procedures
- deferred procedure calls ("Start After Commit")
- event alerts
- triggers
- sequences

## 4.1 Stored Procedures

Stored procedures are simple programs, or procedures, that are executed in IBM solidDB databases. The user can create procedures that contain several SQL statements or whole transactions, and execute them with a single call statement. In addition to SQL statements, 3GL type control structures can be used enabling procedural control. In this way complex, data-bound transactions may be run on the server itself, thus reducing network traffic.

Granting execute rights on a stored procedure automatically invokes the necessary access rights to all database objects used in the procedure. Therefore, administering database access rights may be greatly simplified by allowing access to critical data through procedures.

This section explains in detail how to use stored procedures. In the beginning of this section, the general concepts of using the procedures are explained. Later sections go more in-depth and describe the actual syntax of different statements in the procedures. The end of this section discusses transaction management, sequences and other advanced stored procedure features.

### 4.1.1 Basic Procedure Structure

A stored procedure is a standard IBM solidDB database object that can be manipulated using standard DDL statements CREATE and DROP.

In its simplest form a stored procedure definition looks like:

```
"CREATE PROCEDURE procedure_name
parameter_section
BEGIN
declare_section_local_variables
procedure_body
END";
```

The following example creates a procedure called TEST:

```
"CREATE PROCEDURE test
BEGIN
END"
```

Procedures can be run by issuing a CALL statement followed by the name of the procedure to be invoked:

```
CALL test
```

## 4.1.2 Naming Procedures

Procedure names have to be unique within a database schema.

All the standard naming restrictions applicable to database objects, like using reserved words, identifier lengths, etc., apply to stored procedure names. For an overview and complete list of reserved words, see Appendix C, *Reserved Words*.

## 4.1.3 Parameter Section

A stored procedure communicates with the calling program using parameters. IBM solidDB supports two methods to return values to the calling program. The first method is the standard SQL-99 method, which uses parameters, and the other is a IBM solidDB proprietary method, RETURNS, which uses result sets.

### Using Parameters

Using parameters is the standard SQL-99 method of returning data. Stored procedures accept three types of parameters:

- Input parameters, which are used as input to the procedure. Parameters are input parameters by default. Thus, keyword IN is optional.
- Output parameters, which are returned values from the procedure.

- Input/output parameters, which pass values into the procedure and return a value back to the calling procedure.

Declaring input parameters in the procedure heading make their values accessible inside the procedure by referring to the parameter name. The parameter data type must also be declared. For supported data types, see Appendix A, *Data Types*.

The syntax used in the parameter declaration is (for the complete syntax, see Appendix B, *IBM solidDB SQL Syntax*):

```
parameter_definition ::= [parameter_mode] parameter_name data_type  
parameter_mode ::= IN | OUT | INOUT
```

There can be any number of parameters. Input parameters have to be supplied in the same order as they are defined when the procedure is called.

You can give default values to the parameters when you create the procedure. When you declare the parameter, just add an equals character (=) and the default value after the parameter data type. For example:

```
"CREATE PROCEDURE participants( adults integer = 1,  
children integer = '0',  
pets integer = '0')  
BEGIN  
END"
```

When you call the procedure which has default values for the parameters defined, you don't have to give values for all the parameters. To use default values for all parameters you can simply use the command:

```
call participants()
```

To give a value to a parameter, use the parameter name in the call statement and assign the parameter value by using the equals character as shown in the example below:

```
call participants(children = 2)
```

This command gives value 2 for parameter "children" and default values for parameters "adults" and "pets".

If parameter names are not used in the call statement, IBM solidDB assumes that the parameters are given in same the order as in the create statement.

*Examples:*

```
call participants(1)
```

This command uses value 1 for parameter "adults" and default values for parameters "children" and "pets".

```
call participants(1,2)
```

This command uses value 1 for parameter "adults" and value 2 for parameter "children". The default value is used for parameter "pets".

If a name is given to a parameter, all parameters following it must also have a name. This is why command:

```
call participants(adults = 1,2)
```

returns an error.

```
call participants(1,children = 2)
```

This command uses value 1 for parameter "adults" and value 2 for parameter "children". The default value is used for parameter "pets".

## Using RETURNS

You can use stored procedures to return a result set table with several rows of data in separate columns. This is a IBM solidDB proprietary method to return data and it is performed by using the RETURNS structure.

When you use the RETURNS structure, you must separately declare result set column names for the output data rows. There can be any number of result set column names. The result set column names are declared in the RETURNS section of the procedure definition:

```
"CREATE PROCEDURE procedure_name
[ (IN input_param1 datatype[,
input_param2 datatype, ... ]) ]
[ RETURNS
(output_column_definition1 datatype[,
output_column_definition2 datatype, ... ]) ]
```

```
BEGIN  
END";
```

By default, the procedure only returns one row of data containing the values as they were at the moment when the stored procedure was run or was forced to exit. However, it is also possible to return result sets from a procedure using the following syntax:

```
return row;
```

Every RETURN ROW call adds a new row into the returned result set where column values are the current values of the result set column names.

The following statement creates a procedure that has two input parameters and two result set column names for output rows:

```
"CREATE PROCEDURE PHONEBOOK_SEARCH  
  (IN FIRST_NAME VARCHAR, LAST_NAME VARCHAR)  
  RETURNS (PHONE_NR NUMERIC, CITY VARCHAR)  
BEGIN  
-- procedure_body  
END";
```

This procedure should be called by using two input parameter of data type VARCHAR. The procedure returns an output table consisting of two columns named PHONE\_NR of type NUMERIC and CITY of type VARCHAR.

For example:

```
call phonebook_search ('JOHN', 'DOE');
```

The result looks as follows (when the procedure body has been programmed):

PHONE_NR	CITY
3433555	NEW YORK
2345226	LOS ANGELES

The following statement creates a calculator procedure:

```
"create procedure calc(i1 float, op char(1),  
  i2 float)
```

```
returns (calresult float)
begin
  declare i integer;

  if op = '+' then
    calresult := i1 + i2;
  elseif op = '-' then
    calresult := i1 - i2;
  elseif op = '*' then
    calresult := i1 * i2;
  elseif op = '/' then
    calresult := i1 / i2;
  else
    calresult := 'Error: illegal op';
  end if
end";
```

You can test the calculator with the command:

```
call calc(1,'/',3);
```

With RETURNS, select statements can also be wrapped into database procedures. The following statement creates a procedure that uses a select statement to return backups created from the database:

```
"create procedure show_backups
  returns (backup_number varchar, date_created varchar)
begin
-- First set action for failing statements.
  exec sql  whenever sqlerror rollback, abort;

-- Prepare and execute the select statement
  exec sql prepare sel_cursor select
    replace(property, 'backup ', ''),
    substring(value_str, 1, 19) from sys_info
  where property like 'backup %';
  exec sql execute sel_cursor into (backup_number, date_created);

-- Fetch first row;
  exec sql fetch sel_cursor;
-- Loop until end of table
```



```
while sqlsuccess loop
-- Return the fetched row
  return row;
-- Fetch next
  exec sql fetch sel_cursor;
end loop;
end";
```

### 4.1.4 Declare Section

Local variables that are used inside the procedure for temporary storage of column and control values are defined in a separate section of the stored procedure directly following the `BEGIN` keyword.

The syntax of declaring a variable is:

```
DECLARE variable_name datatype;
```

Note that every declare statement should be ended with a semicolon (;).

The variable name is an alphanumeric string that identifies the variable. The data type of the variable can be any valid SQL data type supported. For supported data types, see Appendix A, *Data Types*.

For example:

```
"CREATE PROCEDURE PHONEBOOK_SEARCH
  (FIRST_NAME VARCHAR, LAST_NAME VARCHAR)
  RETURNS (PHONE_NR NUMERIC, CITY VARCHAR)
BEGIN
DECLARE i INTEGER;

DECLARE dat DATE;

END";
```

Note that input and output parameters are treated like local variables within a procedure with the exception that input parameters have a preset value and output parameter values are returned or can be appended to the returned result set.

## 4.1.5 Procedure Body

The procedure body contains the actual stored procedure program based on assignments, expressions, and SQL statements.

Any type of expression, including scalar functions, can be used in a procedure body. For valid expressions, see Appendix B, *IBM solidDB SQL Syntax*.

## 4.1.6 Assignments

To assign values to variables either of the following syntax is used:

```
SET variable_name = expression;
```

or

```
variable_name := expression;
```

Example:

```
SET i = i + 20 ;  
  
i := 100;
```

## Scalar Functions with Assignments

A scalar function is an operation denoted by a function name followed by a pair of parentheses enclosing zero or more specified arguments. Each scalar function returns one value. Note that scalar functions can be used with assignments, as in:

```
"CREATE PROCEDURE scalar_sample  
RETURNS (string_var VARCHAR(20))  
BEGIN  
-- CHAR(39) is the single quote/apostrophe  
string_var := 'Joe' + {fn CHAR (39)} + 's Garage';  
END";
```

The result of this stored procedure is the output:

Joe's Garage

For a list of IBM solidDB-supported scalar functions (SQL-92), see Appendix B, *IBM solidDB SQL Syntax*. Note that *IBM solidDB Programmer Guide* contains an appendix that describes ODBC scalar functions, which contain some differences for SQL-92.

### Variables, Constants, and Parameters in Assignments

Variables and constants are initialized every time a procedure is executed. By default, variables are initialized to NULL. Unless a variable has been explicitly initialized, its value is NULL, as the following example shows:

```
BEGIN
DECLARE   total   INTEGER;
...
total := total + 1;  -- assigns a null to total
...

```

Therefore, a variable should never be referenced before it has been assigned a value.

The expression following the assignment operator can be arbitrarily complex, but it must yield a data type that is the same as or convertible to the data type of the variable.

When possible, IBM solidDB procedure language can provide conversion of data types implicitly. This makes it possible to use literals, variables and parameters of one type where another type is expected.

Implicit conversion is not possible if:

- information would be lost in the conversion.
- a string to be converted to an integer contains non-numeric data

Examples:

```
DECLARE integer_var INTEGER;
integer_var := 'NR:123';

```

returns an error.

```
DECLARE string_var CHAR(3);  
string_var := 123.45;
```

results in value '123' in variable *string\_var*.

```
DECLARE string_var VARCHAR(2);  
string_var := 123.45;
```

returns an error.

### Single Quotes and Apostrophes in String Assignments

Strings are delimited by single quotes. If you want to have a single quote marks within a string, then you can put two single quote marks ("), side by side, to produce one quote mark in your output. This is commonly known as an "escape sequence." Following is a stored procedure that uses this technique:

```
"CREATE PROCEDURE q  
RETURNS (string_var VARCHAR(20))  
BEGIN  
string_var := 'Joe''s Garage';  
END";  
CALL q;
```

The result is:

Joe's Garage

Here are some other examples:

```
'I''m writing.'
```

becomes:

```
I'm writing.
```

and

```
'Here are two single quotes:'''''
```

becomes:

```
Here are two single quotes:''
```

Note that in the last example there are five single quotes in a row at the end of the string. The last of these is the delimiter (the closing quote mark); the preceding four are part of the data. The four quotes are treated as two pairs of quotes, and each pair of quotes is treated as an escape sequence representing one single quote mark.

## 4.1.7 Expressions

### Comparison Operators

Comparison operators compare one expression to another. The result is always TRUE, FALSE, or NULL. Typically, comparisons are used in conditional control statements and allow comparisons of arbitrarily complex expressions. The following table gives the meaning of each operator:

**Table 4.1. Comparison Operators**

Operator	Meaning
=	is equal to
<>	is not equal to
<	is less than
>	is greater than
<=	is less than or equal to
>=	is greater than or equal to

Note that the != notation cannot be used inside a stored procedure, use the ANSI-SQL compliant <> instead.

### Logical Operators

The logical operators can be used to build more complex queries. The logical operators AND, OR, and NOT operate according to the tri-state logic illustrated by the truth tables shown below. AND and OR are binary operators; NOT is a unary operator.

**Table 4.2. Logical Operators: NOT**

<b>NOT</b>	<b>true</b>	<b>false</b>	<b>null</b>
	false	true	null

**Table 4.3. Logical Operators: AND**

<b>AND</b>	<b>true</b>	<b>false</b>	<b>null</b>
true	true	false	null
false	false	false	false
null	null	false	null

**Table 4.4. Logical Operators: OR**

<b>OR</b>	<b>true</b>	<b>false</b>	<b>null</b>
true	true	true	true
false	true	false	null
null	true	null	null

As the truth tables show, AND returns the value TRUE only if both its operands are true. On the other hand, OR returns the value TRUE if either of its operands is true. NOT returns the opposite value (logical negation) of its operand. For example, NOT TRUE returns FALSE.

NOT NULL returns NULL because nulls are indeterminate.

When not using parentheses to specify the order of evaluation, operator precedence determines the order.

Note that 'true' and 'false' are not literals accepted by SQL parser but values. Logical expression value can be interpreted as a numeric variable:

false = 0 or NULL

true = 1 or any other numeric value

Example:

```
IF expression = TRUE THEN
```

can be simply written

`IF expression THEN`

### **IS NULL Operator**

The IS NULL operator returns the Boolean value TRUE if its operand is null, or FALSE if it is not null. Comparisons involving nulls always yield NULL. To test whether a value is NULL, do not use the expression,

`IF variable = NULL THEN...`

because it never evaluates to TRUE.

Instead, use the following statement:

`IF variable IS NULL THEN...`

Note that when using multiple logical operators in IBM solidDB stored procedures the individual logical expressions should be enclosed in parentheses like:

`((A >= B) AND (C = 2)) OR (A = 3)`

## **4.1.8 Control Structures**

The following sections describe the statements that can be used in the procedure body, including branch and loop statements.

### **IF Statement**

Often, it is necessary to take alternative actions depending on circumstances. The IF statement executes a sequence of statements conditionally. There are three forms of IF statements: IF-THEN, IF-THEN-ELSE, and IF-THEN-ELSEIF.

#### **IF-THEN**

The simplest form of IF statement associates a condition with a statement list enclosed by the keywords THEN and END IF (not ENDIF), as follows:

`IF condition THEN`

```
    statement_list;  
END IF
```

The sequence of statements is executed only if the condition evaluates to TRUE. If the condition evaluates to FALSE or NULL, the IF statement does nothing. In either case, control passes to the next statement. An example follows:

```
IF sales > quota THEN  
    SET pay = pay + bonus;  
END IF
```

### **IF-THEN-ELSE**

The second form of IF statement adds the keyword ELSE followed by an alternative statement list, as follows:

```
IF condition THEN  
    statement_list1;  
ELSE  
    statement_list2;  
END IF
```

The statement list in the ELSE clause is executed only if the condition evaluates to FALSE or NULL. Thus, the ELSE clause ensures that a statement list is executed. In the following example, the first or second assignment statement is executed when the condition is true or false, respectively:

```
IF trans_type = 'CR' THEN  
    SET balance = balance + credit;  
ELSE  
    SET balance = balance - debit;  
END IF
```

THEN and ELSE clauses can include IF statements. That is, IF statements can be nested, as the following example shows:

```
IF trans_type = 'CR' THEN  
    SET balance = balance + credit ;  
ELSE  
    IF balance >= minimum_balance THEN  
        SET balance = balance - debit ;
```



```
ELSE
    SET balance = minimum_balance;
END IF
END IF
```

### IF-THEN-ELSEIF

Occasionally it is necessary to select an action from several mutually exclusive alternatives. The third form of IF statement uses the keyword ELSEIF to introduce additional conditions, as follows:

```
IF condition1 THEN
    statement_list1;
ELSEIF condition2 THEN
    statement_list2;
ELSE
    statement_list3;
END IF
```

If the first condition evaluates to FALSE or NULL, the ELSEIF clause tests another condition. An IF statement can have any number of ELSEIF clauses; the final ELSE clause is optional. Conditions are evaluated one by one from top to bottom. If any condition evaluates to TRUE, its associated statement list is executed and the rest of the statements (inside the IF-THEN-ELSEIF) are skipped. If all conditions evaluate to FALSE or NULL, the sequence in the ELSE clause is executed. Consider the following example:

```
IF sales > 50000 THEN
    bonus := 1500;
ELSEIF sales > 35000 THEN
    bonus := 500;
ELSE
    bonus := 100;
END IF
```

If the value of "sales" is more than 50000, the first and second conditions are true. Nevertheless, "bonus" is assigned the proper value of 1500 since the second condition is never tested. When the first condition evaluates to TRUE, its associated statement is executed and control passes to the next statement following the IF-THEN-ELSEIF.

When possible, use the ELSEIF clause instead of nested IF statements. That way, the code will be easier to read and understand. Compare the following IF statements:

```
IF condition1 THEN
    statement_list1;
ELSE
    IF condition2 THEN
        statement_list2;
    ELSE
        IF condition3 THEN
            statement_list3;
        END IF
    END IF
END IF

IF condition1 THEN
    statement_list1;
ELSEIF condition2 THEN
    statement_list2;
ELSEIF condition3 THEN
    statement_list3;
END IF
```

These statements are logically equivalent, but the first statement obscures the flow of logic, whereas the second statement reveals it.

## Use of Parentheses in IF-THEN Statements

The following code illustrates the rules for using parentheses in IF-THEN statements. Refer also to the release notes for additional information on using parentheses in IF-THEN statements.

```
--- This piece of code shows examples of valid logical conditions in IF
--- statements.
```

```
"CREATE PROCEDURE sample_if_conditions
BEGIN
DECLARE x INT;
DECLARE y INT;
x := 2;
y := 2;
```

```
--- As shown below, a single logical expression in an IF condition may
--- use parentheses.
```

```
IF (x > 0) THEN
x := x - 1;
END IF;
```

```
--- As shown below, although a single logical expression in an IF
--- condition may use parentheses, the parentheses are not required.
```

```
IF x > 0 THEN
x := x - 1;
END IF;
```

--- As shown below, if there are multiple expressions inside a logical condition, parentheses are allowed (and in fact are required) around each subexpression.

```
IF (x > 0) AND (y > 0) THEN
x := x - 1;
END IF;
```

--- The example below is the same as the preceding example, except that this has additional parentheses around the entire expression.

```
IF ((x > 0) AND (y > 0)) THEN
x := x - 1;
END IF;
```

### WHILE-LOOP

The WHILE-LOOP statement associates a condition with a sequence of statements enclosed by the keywords LOOP and END LOOP, as follows:

```
WHILE condition LOOP
    statement_list;
END LOOP
```

Before each iteration of the loop, the condition is evaluated. If the condition evaluates to TRUE, the statement list is executed, then control resumes at the top of the loop. If the condition evaluates to FALSE or NULL, the loop is bypassed and control passes to the next statement. An example follows:

```
WHILE total <= 25000 LOOP
    ...
    total := total + salary;
END LOOP
```

The number of iterations depends on the condition and is unknown until the loop completes. Since the condition is tested at the top of the loop, the sequence might execute zero times. In the latter example, if the initial value of "total" is greater than 25000, the condition evaluates to FALSE and the loop is bypassed altogether.

Loops can be nested. When an inner loop is finished, control is returned to the next loop. The procedure continues from the next statement after END LOOP.

## Leaving Loops

It may be necessary to force the procedure to leave a loop prematurely. This can be implemented using the LEAVE keyword:

```
WHILE total < 25000 LOOP
  total := total + salary;
  IF exit_condition THEN
    LEAVE;
  END IF
END LOOP
statement_list2
```

Upon successful evaluation of the *exit\_condition* the loop is left, and the procedure continues at the *statement\_list2*.



### Note

Although IBM solidDB databases support the ANSI-SQL CASE syntax, the CASE construct cannot be used inside a stored procedure as a control structure.

## Use of Parentheses in WHILE Loops

The following code illustrates the rules for using parentheses in WHILE loops. Refer also to the release notes for additional information on using parentheses in WHILE loops.

```
--- This piece of code shows examples of valid logical conditions in
--- WHILE loops.
"CREATE PROCEDURE sample_while_conditions
BEGIN
DECLARE x INT;
DECLARE y INT;
x := 2;
y := 2;

--- As shown below, a single logical expression in a WHILE condition
--- may use parentheses.
WHILE (x > 0) LOOP
x := x - 1;
END LOOP;
```

--- As shown below, although a single logical expression in a WHILE condition may use parentheses, the parentheses are not required.

```
WHILE x > 0 LOOP
x := x - 1;
END LOOP;
```

--- As shown below, if there are multiple expressions inside a logical condition, then you need parentheses around EACH individual expression.

```
WHILE (x > 0) AND (y > 0) LOOP
x := x - 1;
y := y - 1;
END LOOP;
```

--- The example below is the same as the preceding example, except that this has additional parentheses around the entire expression.

```
WHILE ((x > 0) AND (y > 0)) LOOP
x := x - 1;
y := y - 1;
END LOOP;
```

### Handling Nulls

Nulls can cause confusing behavior. To avoid some common errors, observe the following rules:

- comparisons involving nulls always yield NULL
- applying the logical operator NOT to a null yields NULL
- in conditional control statements, if the condition evaluates to NULL, its associated sequence of statements is not executed

In the example below, you might expect the statement list to execute because "x" and "y" seem unequal. Remember though that nulls are indeterminate. Whether "x" is equal to "y" or not is unknown. Therefore, the IF condition evaluates to NULL and the statement list is bypassed.

```
x := 5;
  y := NULL;
  ...
```

```
IF x <> y THEN -- evaluates to NULL, not TRUE
  statement_list; -- not executed
END IF
```

In the next example, one might expect the statement list to execute because "a" and "b" seem equal. But, again, this is unknown, so the IF condition evaluates to NULL and the statement list is bypassed.

```
a := NULL;
b := NULL;
...
IF a = b THEN -- evaluates to NULL, not TRUE
  statement_list; -- not executed
END IF
```

### NOT Operator

Applying the logical operator NOT to a null yields NULL. Thus, the following two statements are not always equivalent:

```
IF x > y THEN                IF NOT (x > y) THEN
  high := x;                  high := y;
ELSE                           ELSE
  high := y;                  high := x;
END IF                          END IF
```

The sequence of statements in the ELSE clause is executed when the IF condition evaluates to FALSE or NULL. If either or both "x" and "y" are NULL, the first IF statement assigns the value of "y" to "high", but the second IF statement assigns the value of "x" to "high". If neither "x" nor "y" is NULL, both IF statements assign the corresponding value to "high".

### Zero-Length Strings

Zero length strings are treated by a IBM solidDB server like they are a string of zero length, instead of a null. NULL values should be specifically assigned as in the following:

```
SET a = NULL;
```

This also means that checking for NULL values will return FALSE when applied to a zero-length string.

## Example Stored Procedure

Following is an example of a simple procedure that determines whether a person is an adult on the basis of a birthday as input parameter.

Note the usage of {fn ...} on scalar functions, and semicolons to end assignments.

```
"CREATE PROCEDURE grown_up
(birth_date DATE)
RETURNS (description VARCHAR)
BEGIN
DECLARE age INTEGER;
-- determine the number of years since the day of birth
age := {fn TIMESTAMPDIFF(SQL_TSI_YEAR, birth_date, now())};
IF age >= 18 THEN
-- If age is at least 18, then it's an adult
description := 'ADULT';
ELSE
-- otherwise it's still a minor
description := 'MINOR';
END IF
END";
```

## Exiting a Procedure

A procedure may be exited prematurely by issuing the keyword

```
RETURN;
```

at any location. After this keyword, control is directly handed to the program calling the procedure returning the values bound to the result set column names as indicated in the RETURNS section of the procedure definition.

## Returning Data

You can return data with the OUT parameter mode, which is a standard SQL-99 method of returning data. This method allows you to pass data back to the program from the procedure. For syntax information, refer to Appendix B, *IBM solidDB SQL Syntax*.

The OUT parameter mode has the following characteristics:

- The OUT parameter mode allows you to pass data back to the calling program from the procedure. Inside the calling program, the OUT parameter acts like a variable. That means you can use an OUT parameter as if it were a local variable. You can change its value or reference the value in any way.
- The actual parameter that corresponds to an OUT parameter must be a variable; it cannot be a constant or an expression.
- Like variables, OUT parameters are initialized to NULL.

Before exiting a procedure, you must explicitly assign values to all OUT parameters. Otherwise, the corresponding actual parameters will be null. If you exit successfully, IBM solidDB assigns values to the actual parameters. However, if you exit with an unhandled exception, IBM solidDB does not assign values to the actual parameters.

For a IBM solidDB proprietary method of returning data, see the section called “Using RETURNS”.

## 4.2 Remote Stored Procedures

Stored procedures may be called locally or remotely. By "remotely", we mean that one database server may call a stored procedure on another database server. Remote stored procedure calls use a syntax like the following:

```
CALL procedure_name AT node-ref;
```

where *node-ref* indicates which database server the remote stored procedure is on.

Remote stored procedures calls can only be made between two IBM solidDB servers that have a master/replica relationship. The calls can be made in either "direction"; i.e. the master may call a stored procedure on the replica, or the replica may call a stored procedure on the master. A remote stored procedure may be called from any context that allows a local procedure call. Thus, for example, you may call a remote stored procedure directly by using a CALL statement, or you may call the remote procedure from within a trigger, or another stored procedure, or a Start After Commit statement.

A remotely-called stored procedure may contain any command that any other stored procedure may contain. All stored procedures are created using the same syntax rules. A single stored procedure may be called both locally and remotely at different times.

The stored procedure, when called remotely, accepts parameters from the caller, just as if the call was local. However, a remote stored procedure cannot return a result set; it can only return an error code.

Both local and remote stored procedure calls are synchronous; in other words, whether the procedure is called locally or remotely, the caller waits until the value is returned; the caller does not continue on while the stored



procedure executes in the background. (Note that if the stored procedure is called from inside a `START AFTER COMMIT`, then the stored procedure call itself is synchronous, but the `START AFTER COMMIT` was not synchronous, so the stored procedure will execute as an asynchronous background process.)

### **Important**

Transaction handling for remote stored procedures is different from transaction handling for local stored procedures. When a stored procedure is called remotely, the execution of the stored procedure is **NOT** a part of the transaction that contained the call. Therefore, you cannot roll back a stored procedure call by rolling back the transaction that called it.

The full syntax of the command to call a remote stored procedure is:

```
CALL <proc-name>[(param [, param...])] AT node-def;  
node-def ::= DEFAULT | 'replica name' | 'master name'
```

For example:

```
CALL MyProc('Smith', 750) AT replica1;  
CALL MyProcWithoutParameters AT replica2;
```

See Appendix B, *IBM solidDB SQL Syntax*, for more details about the `CALL` statement.

The node definition "DEFAULT" is used only with the `START AFTER COMMIT` statement. See the section on `START AFTER COMMIT` for more details.

### **Note**

You can only list one node definition per `CALL`. If you want to notify multiple replicas, for example, then you must call each of them separately. You can, however, create a stored procedure that contains multiple `CALL` statements, and then simply make a single call to that procedure.

The remote stored procedure is always created on the server that executes the procedure, not on the server that calls the procedure. For example, if the master is going to call procedure `foo()` to execute on `replica1`, then procedure `foo()` must have been created on `replica1`. The master does not know the "content" of the stored procedure that it calls remotely. In fact, the master does not know anything at all about the stored procedure other than the information specified in the `CALL` statement itself, for example:

```
CALL foo(param1, param2) AT replical
```

which of course includes the procedure's name, some parameter values, and the name of the replica on which the procedure is to be executed. The stored procedure is not registered with the caller. This means that the caller in some sense calls the procedure "blindly", without even knowing if it's there. Of course, if the caller tries to call a procedure that doesn't exist, then the caller will get an error message that says that the procedure doesn't exist.

Dynamic parameter binding is supported. For example, the following is legal:

```
CALL MYPROC(?, ?) AT MYREPLICAL;
```

Calls to the stored procedure are not buffered or queued. If you call the stored procedure and the procedure does not exist, the call does not "persist", waiting until the stored procedure appears. Similarly, if the procedure does exist but the server that has that procedure is shut down or is disconnected from the network is not accessible for any other reason, then the call is not held "open" and retried when the server becomes accessible again. This is important to know when using the "Sync Pull Notify" (push synchronization) feature.

### 4.2.1 ACCESS RIGHTS

To call a stored procedure, the caller must have EXECUTE privilege on that procedure. (This is true for any stored procedure, whether it is called locally or remotely.)

When a procedure is called locally, it is executed with the privileges of the caller. When a procedure is called remotely, it may be executed either with the privileges of a specified user on the remote server, or with the privileges of the remote user who corresponds to the local caller. (The replica and master users must already be mapped to each other before the stored procedure is called. For more information about mapping replica users to master users, see *IBM solidDB Advanced Replication Guide*.)

If a remote stored procedure was called from the replica (and is to be executed on the master), then you have the option of specifying which master user's privileges you would like the procedure to be executed with.

If the remote stored procedure was called from the master (and is to be executed on the replica), or if you do not specify which user's privileges to use, then the calling server will figure out which user's privileges should be used, based on which user called the stored procedure and the mapping between replica and master users.

These possibilities are explained in more detail below.

1. If the procedure was called from a replica (and will be executed on the master), then you may execute the SET SYNC USER statement to specify which master user's privileges to use. You must execute SET SYNC USER on the local server before calling the remote stored procedure. Once the sync user has been

specified on the calling server, the calling server will send the user name and password to the remote server (the master server) each time a remote stored procedure is called. The remote server will try to execute the procedure using the user id and password that were sent with the procedure call. The user id and password must exist in the remote server, and the specified user must have appropriate access rights to the database and EXECUTE privilege on the called procedure.

The SET SYNC USER statement is valid only on a replica, so you can only specify the sync user when a replica calls a stored procedure on a master.

2. If the caller is a master, or if the call was made from a replica and you did not specify a sync user before the call, then the servers will attempt to determine which user on the remote server corresponds to the user on the local server.

*If the calling server is a replica ( $R \rightarrow M$ )*

The calling server sends the following information to the remote server when calling a remote procedure:

Name of the master (SYS\_SYNC\_MASTERS.NAME).

Replica id (SYS\_SYNC\_MASTERS.REPLICA\_ID).

Master user id (This master user id is the master user id that corresponds to the user id of the local user who called the procedure. Obviously, this local user must already be mapped to the corresponding master user.)

Note that this method of selecting the master user id is the same as the method used when a replica refreshes data — the replica looks up in the SYS\_SYNC\_USERS table to find the master user who is mapped to the current local replica user.

*If the calling server is a master ( $M \rightarrow R$ )*

The calling server sends the following information to the remote server when calling a remote procedure:

Name of the master (SYS\_SYNC\_REPLICAS.MASTER\_NAME).

Replica id (SYS\_SYNC\_REPLICAS.ID).

User name of the caller.

User id of the caller.

When the replica receives the master user id, the replica looks up the local user who is mapped to that master id. Since more than one replica user may be mapped to a single master user, the server will use

the first local user it finds who is mapped to the specified master user and who has the privileges required to execute this stored procedure.

Before a master server can call a stored procedure on a replica server, the master must of course know the connect string of the replica. If a replica allows calls from a master, then the replica should define its own connect string information in the `solid.ini` file. This information is provided to the master (the replica includes a copy when it forwards any message to master). When the master receives the connect string from the replica, the master replaces the previous value (if the new value differs).

Example:

```
[Synchronizer]
ConnectStrForMaster=tcp replicahost 1316
```

It is also possible to inform the master of the replica's connect string by using the statement:

```
SET SYNC CONNECT <connect-info> TO REPLICA <replica-name>
```

This is useful if the master needs to call the replica but the replica has not yet provided its connect string to the master (i.e. has not yet forwarded any message to the master).

## 4.3 Using SQL in a Stored Procedure

Using SQL statements inside a stored procedure is somewhat different from issuing SQL directly from tools like `solsql`.

A special syntax is required when using SQL statements inside a stored procedure. There are two ways to execute SQL statements inside a procedure: you may use the `EXECDIRECT` syntax to execute a statement, or you may treat the SQL statement as a "cursor". Both possibilities are explained below.

### 4.3.1 EXECDIRECT

The `EXECDIRECT` syntax is particularly appropriate for statements where there is no result set, and where you do not have to use any variable to specify a parameter value. For example, the following statement inserts a single row of data:

```
EXEC SQL EXECDIRECT insert into table1 (id, name) values (1, 'Smith');
```

For more information about `EXECDIRECT`, see Section 4.3.1, "EXECDIRECT".

## 4.3.2 Using a Cursor

Cursors are appropriate for statements where there is a result set, or where you want to repeat a single basic statement but use different values from a local variable as a parameter (e.g. in a loop).

A cursor is a specific allocated part of the server process memory that keeps track of the statement being processed. Memory space is allocated for holding one row of the underlying statement, together with some status information on the current row (in SELECTS) or the number of rows affected by the statement (in UPDATES, INSERTS and DELETES).

In this way query results are processed one row at a time. The stored procedure logic should take care of the actual handling of the rows, and the positioning of the cursor on the required row(s).

There are five basic steps in handling a cursor:

1. Preparing the cursor - the definition
2. Executing the cursor - executing the statement
3. Fetching on the cursor (for select procedure calls) - getting the results row by row
4. Closing the cursor after use - still enabling it to re-execute
5. Dropping the cursor from memory - removing it

### 1. Preparing the Cursor

A cursor is defined (prepared) using the following syntax:

```
EXEC SQL PREPARE cursor_name SQL_statement;
```

By preparing a cursor, memory space is allocated to accommodate one row of the result set of the statement, and the statement is parsed and optimized.

A cursor name given for the statement must be unique within the connection. This means procedures that contain cursors cannot be called recursively (at least not from a statement that is after a PREPARE CURSOR and before the corresponding DROP CURSOR). When a cursor is prepared, a IBM solidDB server checks that no other cursor of this name is currently open. If there is one, error number 14504 is returned.

Note that statement cursors can also be opened using the ODBC API. These cursor names need to be different from the cursors opened from procedures.

Example:

```
EXEC SQL PREPARE sel_tables
  SELECT table_name
  FROM sys_tables
  WHERE table_name LIKE 'SYS%';
```

This statement will prepare the cursor named *sel\_tables*, but will not execute the statement that it contains.

## 2. Executing the Cursor

After a statement has been successfully prepared it can be executed. An execute binds possible input and output variables to it and runs the actual statement.

Syntax of the execute statement is:

```
EXEC SQL EXECUTE cursor_name
  [ INTO ( var1 [, var2...] ) ];
```

The optional section INTO binds result data of the statement to variables.

Variables listed in parentheses after the INTO keyword are used when running a SELECT or CALL statement. The resulting columns of the SELECT or CALL statement are bound to these variables when the statement is executed. The variables are bound starting from the left-most column listed in the statement. Binding of variables continues to the following column until all variables in the list of variables have been bound. For example to extend the sequence for the cursor *sel\_tables* that was prepared earlier we need to run the following statements:

```
EXEC SQL PREPARE sel_tables
  SELECT table_name
  FROM sys_tables
  WHERE table_name LIKE 'SYS%'

EXEC SQL EXECUTE sel_tables INTO (tab);
```

The statement is now executed and the resulting table names will be returned into variable *tab* in the subsequent Fetch statements.

### 3. Fetching on the Cursor

When a SELECT or CALL statement has been prepared and executed, it is ready for fetching data from it. Other statements (UPDATE, INSERT, DELETE, DDL) do not require fetching as there will be no result set. Fetching results is done using the fetch syntax:

```
EXEC SQL FETCH cursor_name;
```

This command fetches a single row from the cursor to the variables that were bound with INTO keyword when the statement was executed.

To complete the previous example to actually get result rows back, the statements will look like:

```
EXEC SQL PREPARE sel_tables
  SELECT table_name
  FROM sys_tables
  WHERE table_name LIKE 'SYS%'
EXEC SQL EXECUTE sel_tables INTO (tab);
EXEC SQL FETCH sel_tables;
```

After this the variable `tab` will contain the table name of the first table found conforming to the WHERE clause.

Subsequent calls to fetch on the cursor `sel_tables` will get the next row(s) if the select found more than one.

To fetch all table names a loop construct may be used:

```
WHILE expression LOOP
  EXEC SQL FETCH sel_tables;
END LOOP
```

Note that after the completion of the loop, the variable `tab` will contain the last fetched table name.

### 4. Closing the Cursor

Cursors may be closed by issuing the statement

```
EXEC SQL CLOSE cursor_name;
```

This will not remove the actual cursor definition from memory; it may be re-executed when the need arises.

## 5. Dropping the Cursor

Cursors may be dropped from memory, releasing all resources by the statement:

```
EXEC SQL DROP cursor_name;
```

### Example Stored procedure

Here is an example of a stored procedure that uses EXECDIRECT in one place and uses a cursor in another place.

```
"CREATE PROCEDURE p2
BEGIN

-- This variable holds an ID that we insert into the table.
DECLARE id INT;

-- Here are simple examples of EXECDIRECT.
EXEC SQL EXECDIRECT create table table1 (id_col INT);
EXEC SQL EXECDIRECT insert into table1 (id_col) values (1);

-- Here is an example of a cursor.
EXEC SQL PREPARE cursor1 INSERT INTO table1 (id_col) values (?);
id := 2;
WHILE id <= 10 LOOP
    EXEC SQL EXECUTE cursor1 USING (id);
    id := id + 1;
END LOOP;
EXEC SQL CLOSE cursor1;
EXEC SQL DROP cursor1;

END";
```

## 4.3.3 Error Handling

### SQLSUCCESS

The return value of the latest EXEC SQL statement executed inside a procedure body is stored into variable SQLSUCCESS. This variable is automatically generated for every procedure. If the previous SQL statement



was successful, the value 1 is stored into `SQLSUCCESS`. After a failed SQL statement, a value 0 is stored into `SQLSUCCESS`.

The value of `SQLSUCCESS` may be used, for instance, to determine when the cursor has reached the end of the result set as in the following example:

```
EXEC SQL FETCH sel_tab;
-- loop as long as last statement in loop is successful
WHILE SQLSUCCESS LOOP
    -- do something with the results, for example, return a row
    EXEC SQL FETCH sel_tab;

END LOOP
```

## **SQLERRNUM**

This variable contains the error code of the latest SQL statement executed. It is automatically generated for every procedure. After successful execution, `SQLERRNUM` contains zero (0).

## **SQLERRSTR**

This variable contains the error string from the last failed SQL statement.

## **SQLROWCOUNT**

After the execution of `UPDATE`, `INSERT` and `DELETE` statements, an additional variable is available to check the result of the statement. Variable `SQLROWCOUNT` contains the number of rows affected by the last statement.

## **SQLERROR**

To generate user errors from procedures, the `SQLERROR` variable may be used to return an actual error string that caused the statement to fail to the calling application. The syntax is:

```
RETURN SQLERROR 'error string'
RETURN SQLERROR char_variable
```

The error is returned in the following format:

User error: *error\_string*

**SQLERROR OF *cursorname***

For error checking of EXEC SQL statements, the SQLSUCCESS variable may be used as described under SQLSUCCESS in the beginning of this section. To return the actual error that caused the statement to fail to the calling application, the following syntax may be used:

```
EXEC SQL PREPARE cursorname sql_statement;  
EXEC SQL EXECUTE cursorname;  
IF NOT SQLSUCCESS THEN  
  RETURN SQLERROR OF cursorname;  
END IF  
  
END IF
```

Processing will stop immediately when this statement is executed and the procedure return code is SQLERROR. The actual database error can be returned using the SQLError function:

```
Solid Database error 10033: Primary key unique constraint violation
```

The generic error handling method for a procedure can be declared with:

```
EXEC SQL WHENEVER SQLERROR [ROLLBACK [WORK],] ABORT;
```

When this statement is included in a stored procedure, all return values of executed SQL statements are checked for errors. If a statement execution returns an error, the procedure is automatically aborted and SQLERROR of the last cursor is returned. Optionally the transaction can also be rolled back.

The statement should be included before any EXEC SQL statements, directly following the DECLARE section of variables.

Below is an example of a complete procedure returning all table names from SYS\_TABLES that start with 'SYS':

```
"CREATE PROCEDURE  sys_tabs  
RETURNS (tab VARCHAR)  
BEGIN  
  -- abort on errors  
EXEC SQL WHENEVER SQLERROR ROLLBACK, ABORT;  
  -- prepare the cursor
```

```
EXEC SQL PREPARE sel_tables
  SELECT table_name
  FROM sys_tables
  WHERE table_name LIKE 'SYS%';
-- execute the cursor
EXEC SQL EXECUTE sel_tables INTO (tab);
-- loop through rows
EXEC SQL FETCH sel_tables;
WHILE sqlsuccess LOOP
  RETURN ROW;
  EXEC SQL FETCH sel_tables;
END LOOP
-- close and drop the used cursors
EXEC SQL CLOSE sel_tables;
EXEC SQL DROP sel_tables;
END";
```

## 4.3.4 Parameter Markers in Cursors

In order to make a cursor more dynamic, a SQL statement can contain parameter markers that indicate values that are bound to the actual parameter values at execute time. The '?' symbol is used as a parameter marker.

Syntax example:

```
EXEC SQL PREPARE sel_tabs
  SELECT table_name
  FROM sys_tables
  WHERE table_name LIKE ?
  AND table_schema LIKE ?;
```

The execution statement is adapted by including a USING keyword to accommodate the binding of a variable to the parameter marker.

```
EXEC SQL EXECUTE sel_tabs USING ( var1, var2 ) INTO (tabs);
```

In this way a single cursor can be used multiple times without having to re-prepare the cursor. As preparing a cursor involves also the parsing and optimizing of the statement, significant performance gains can be achieved by using re-usable cursors.

Note that the USING list only accepts variables; data can not be directly passed in this way. So if for example an insert into a table should be made, one column value of which should always be the same (status = 'NEW') then the following syntax would be wrong:

```
EXEC SQL EXECUTE ins_tab USING (nr, desc, dat, 'NEW');
```

The correct way would be to define the constant value in the prepare section:

```
EXEC SQL PREPARE ins_tab
  INSERT INTO my_tab (id, descript, in_date, status)
  VALUES (?, ?, ?, 'NEW');
EXEC SQL EXECUTE ins_tab USING (nr, desc, dat);
```

Note that variables can be used multiple times in the using list.

The parameters in a SQL statement have no intrinsic data type or explicit declaration. Therefore, parameter markers can be included in a SQL statement only if their data types can be inferred from another operand in the statement.

For example, in an arithmetic expression such as ? + COLUMN1, the data type of the parameter can be inferred from the data type of the named column represented by COLUMN1. A procedure cannot use a parameter marker if the data type cannot be determined.

The following table describes how a data type is determined for several types of parameters.

**Table 4.5. Determining Data Type from Parameters**

Location of Parameter	Assumed Data Type
One operand of a binary arithmetic or comparison operator	Same as the other operand
The first operand in a BETWEEN clause	Same as the other operand
The second or third operand in a BETWEEN clause	Same as the first operand
An expression used with IN	Same as the first value or the result column of the subquery
A value used with IN	Same as the expression
A pattern value used with LIKE	VARCHAR
An update value used with UPDATE	Same as the update column

An application cannot place parameter markers in the following locations:

- As a SQL identifier (name of a table, name of a column etc.)
- In a SELECT list.
- As both expressions in a comparison-predicate.
- As both operands of a binary operator.
- As both the first and second operands of a BETWEEN operation.
- As both the first and third operands of a BETWEEN operation.
- As both the expression and the first value of an IN operation.
- As the operand of a unary + or - operation.
- As the argument of a set-function-reference.

For more information, see the ANSI SQL-92 specification.

In the following example, a stored procedure will read rows from one table and insert parts of them in another, using multiple cursors:

```
"CREATE PROCEDURE  tabs_in_schema (schema_nm VARCHAR)
RETURNS (nr_of_rows INTEGER)
BEGIN
DECLARE tab_nm VARCHAR;
EXEC SQL PREPARE  sel_tab
  SELECT  table_name
  FROM  sys_tables
  WHERE  table_schema = ?;
EXEC SQL PREPARE ins_tab
  INSERT INTO my_table (table_name, schema) VALUES (?,?);

nr_of_rows := 0;

EXEC SQL EXECUTE sel_tab USING (schema_nm) INTO (tab_nm);
EXEC SQL FETCH sel_tab;
WHILE SQLSUCCESS LOOP
  nr_of_rows := nr_of_rows + 1;
  EXEC SQL EXECUTE ins_tab USING(tab_nm, schema_nm);
```

```
IF SQLROWCOUNT <> 1 THEN
  RETURN SQLERROR OF ins_tab;
END IF
EXEC SQL FETCH sel_tab;
END LOOP
END";
```

## 4.4 Calling other Procedures

As calling a procedure forms a part of the supported SQL syntax, a stored procedure may be called from within another stored procedure. The default limit for levels of nested procedures is 16. When the maximum is exceeded, the transaction fails. The maximum nesting level is set in the *MaxNestedProcedures* parameter in the *solid.ini* configuration file. For details, see appendix "Configuration Parameters" in *IBM solidDB Administration Guide*.

Like all SQL statements, a cursor should be prepared and executed like:

```
EXEC SQL PREPARE cp CALL myproc(?, ?);
EXEC SQL EXECUTE cp USING (var1, var2);
```

If procedure *myproc* returns one or more values, then subsequently a fetch should be done on the cursor *cp* to retrieve those values:

```
EXEC SQL PREPARE cp call myproc(?,?);
EXEC SQL EXECUTE cp USING (var1, var2) INTO
(ret_var1, ret_var2);
EXEC SQL FETCH cp;
```

Note that if the called procedure uses a *return row* statement, the calling procedure should utilize a WHILE LOOP construct to fetch all results.

Recursive calls are possible, but discouraged because cursor names are unique at connection level.

### 4.4.1 Positioned Updates and Deletes

In IBM solidDB procedures it is possible to use positioned updates and deletes. This means that an update or delete will be done to a row where a given cursor is currently positioned. The positioned updates and deletes can also be used within stored procedures using the cursor names used within the procedure.

The following syntax is used for positioned updates:

```
UPDATE table_name
SET column = value
WHERE CURRENT OF cursor_name
```

and for deletes

```
DELETE FROM table_name
WHERE CURRENT OF cursor_name
```

In both cases the *cursor\_name* refers to a statement doing a `SELECT` on the table that is to be updated/deleted from.

Positioned cursor update is a semantically suspicious concept in SQL standard that may cause peculiarities also with a IBM solidDB server. Please note the following restriction when using positioned updates.

Below is an example written with pseudo code that will cause an endless loop with a IBM solidDB server (error handling, binding variables and other important tasks omitted for brevity and clarity):

```
"CREATE PROCEDURE ENDLESS_LOOP
BEGIN
EXEC SQL PREPARE MYCURSOR SELECT * FROM TABLE1;
EXEC SQL PREPARE MYCURSOR_UPDATE
  UPDATE TABLE1 SET COLUMN2 = 'new data';
  WHERE CURRENT OF MYCURSOR;"
EXEC SQL EXECUTE MYCURSOR;
EXEC SQL FETCH MYCURSOR;
WHILE SQLSUCCESS LOOP
  EXEC SQL EXECUTE MYCURSOR_UPDATE;
  EXEC SQL COMMIT WORK;
  EXEC SQL FETCH MYCURSOR;
END LOOP
END";
```

The endless loop is caused by the fact that when the update is committed, a new version of the row becomes visible in the cursor and it is accessed in the next `FETCH` statement. This happens because the incremented row version number is included in the key value and the cursor finds the changed row as the next greater key value after the current position. The row gets updated again, the key value is changed and again it will be the next row found.

In the above example, the updated COLUMN2 is not assumed to be part of the primary key for the table, and the row version number was the only part of the index entry that changed. However, if a column value is changed that is part of the index through which the cursor has searched the data, the changed row may jump further forward or backward in the search set.

For these reasons, using positioned update is not recommended in general and searched update should be used instead whenever possible. However, sometimes the update logic may be too complex to be expressed in SQL WHERE clause and in such cases positioned update can be used as follows:

Positioned cursor update works deterministically in IBM solidDB, when the WHERE clause is such that the updated row does not match the criteria and therefore does not reappear in the fetch loop. Constructing such a search criteria may require using additional column only for this purpose.

Note that in an open cursor, user changes do not become visible unless they are committed within the same database session.

## 4.4.2 Transactions

Stored procedures use transactions like any other interface to the database uses transactions. A transaction may be committed or rolled back either inside the procedure or outside the procedure. Inside the procedure a commit or roll back is done using the following syntax:

```
EXEC SQL COMMIT WORK ;  
EXEC SQL ROLLBACK WORK ;
```

These statements end the previous transaction and start a new one.

If a transaction is not committed inside the procedure, it may be ended externally using:

- IBM solidDB SA
- Another stored procedure
- By autocommit, if the connection has AUTOCOMMIT switch set to ON

Note that when a connection has autocommit activated it does not force autocommit inside a procedure. The commit is done when the procedure exits.

## 4.4.3 Default Cursor Management

By default, when a procedure exits, all cursors opened in a procedure are closed. Closing cursors means that cursors are left in a prepared state and can be re-executed.



After exiting, the procedure is put in the procedure cache. When the procedure is dropped from the cache, all cursors are finally dropped.

The number of procedures kept in cache is determined by the `solid.ini` file setting:

```
[SQL]
ProcedureCache = nbr_of_procedures
```

This means that, as long as the procedure is in the procedure cache, all cursors can be re-used as long as they are not dropped. A IBM solidDB server itself manages the procedure cache by keeping track of the cursors declared, and notices if the statement a cursor contains has been prepared.

As cursor management, especially in a heavy multi-user environment, can use a considerable amount of server resources, it is good practice to always close cursors immediately and preferably also drop all cursors that are no longer used. Only the most frequently used cursors may be left non-dropped to reduce the cursor preparation effort.

Note that transactions are not related to procedures or other statements. Commit or rollback therefore does NOT release any resources in a procedure.

### 4.4.4 Notes on SQL

- There is no restriction on the SQL statements used. Any valid SQL statement can be used inside a stored procedure, including DDL and DML statements.
- Cursors may be declared anywhere in a stored procedure. Cursors that are certainly going to be used are best prepared directly following the declare section.
- Cursors that are used inside control structures, and are therefore not always necessary, are best declared at the point where they are activated, to limit the amount of open cursors and hence the memory usage.
- The cursor name is an undeclared identifier, not a variable; it is used only to reference the query. You cannot assign values to a cursor name or use it in an expression.
- Cursors may be re-executed repeatedly without having to re-prepare them. Note that this can have a serious influence on performance; repetitively preparing cursors on similar statements may decrease the performance by around 40% in comparison to re-executing already prepared cursors!
- Any SQL statement will have to be preceded by the keywords EXEC SQL.

## 4.4.5 Functions for Procedure Stack Viewing

The following functions may be included in stored procedures to analyze the current contents of the procedure stack:

- `PROC_COUNT` ( )

This function returns the number of procedures in the procedure stack, including the current procedure.

- `PROC_NAME` (N)

This function returns the Nth procedure name in the stack. The first procedure is in position zero.

- `PROC_SCHEMA` (N)

This function returns the schema name of the Nth procedure in the procedure stack.

These functions allow for stored procedures that behave differently depending on whether they are called from an application or from a procedure.

## 4.5 Procedure Privileges

Stored procedures are owned by the creator, and are part of the creator's schema. Users who need to run stored procedures in other schemas need to be granted EXECUTE privilege on the procedure:

```
GRANT EXECUTE ON Proc_name TO { USER | ROLE };
```

This function returns the schema name of the Nth procedure in the procedure stack.

All database objects accessed within the granted procedure, even subsequently called procedures, are accessed according to the rights of the owner of the procedure. No special grants are necessary.

Since the procedure is run with the privileges of the creator, the procedure not only has the creator's rights to access objects such as tables, but also uses the creator's schema and catalog. For example, suppose that user 'Sally' runs a procedure named 'Proc1' created by user 'Jasmine'. Suppose also that both Sally and Jasmine have a table named 'table1'. By default, the stored procedure Proc1 will use the table1 that is in Jasmine's schema, even if Proc1 was called by user Sally.

See also Section 4.2.1, "ACCESS RIGHTS" for more information about privileges and remote stored procedure calls.

## 4.6 Using Triggers

A trigger activates stored procedure code, which a IBM solidDB server automatically executes when a user attempts to change the data in a table. You may create one or more triggers on a table, with each trigger defined to activate on a specific INSERT, UPDATE, or DELETE command. When a user modifies data within the table, the trigger that corresponds to the command is activated.

Triggers enable you to:

- Implement referential integrity constraints, such as ensuring that a foreign key value matches an existing primary key value.
- Prevent users from making incorrect or inconsistent data changes by ensuring that intended modifications do not compromise a database's integrity.
- Take action based on the value of a row before or after modification.
- Transfer much of the logic processing to the back-end, reducing the amount of work that your application needs to do as well as reducing network traffic.

### 4.6.1 How Triggers Work

The order in which a data manipulation statement is executed when triggers are enabled is the key to understanding how triggers work in IBM solidDB databases.

In IBM solidDB's DML Execution Model, a IBM solidDB server performs a number of validation checks before executing data manipulation statements (INSERT, UPDATE, or DELETE). Following is the execution order for data validation, trigger execution, and integrity constraint checking for a single DML statement.

1. Validate values if they are part of the statement (that is, not bound). This includes null value checking, data type checking (such as numeric), etc.
2. Perform table level security checks.
3. Loop for each row affected by the SQL statement. For each row perform these actions in this order:
  - a. Perform column level security checks.
  - b. Fire BEFORE row trigger.
  - c. Validate values if they are bound in. This includes null value checks, data type checking, and size checking (for example, checking if the character string is too long).

Note that size checking is performed even for values that are not bound.

- d. Execute INSERT/UPDATE/DELETE
  - e. Fire AFTER ROW trigger
4. Commit statement
- a. Perform concurrency conflict checks.
  - b. Perform checks for duplicate values.
  - c. Perform referential integrity checks on invoking DML.



### Note

A trigger itself can cause the DML to be executed, which applies to the steps shown in the above model.

## 4.6.2 Creating Triggers

Use the CREATE TRIGGER statement (described below) to create a trigger. You can disable an existing trigger or all triggers defined on a table by using the ALTER TRIGGER statement. For details, read Section 4.8.6, “Altering Trigger Attributes”. The ALTER TRIGGER statement causes a IBM solidDB server to ignore the trigger when an activating DML statement is issued. With this statement, you can also enable a trigger that is currently inactive.

To drop a trigger from the system catalog, use DROP TRIGGER. For details, read Section 4.8.5, “Dropping Triggers”.

### CREATE TRIGGER Statement

The CREATE TRIGGER statement creates a trigger. To create a trigger you must be a DBA or owner of the table on which the trigger is being defined. To create a trigger, provide the catalog, schema/owner and name of the table on which a trigger is being defined. For an example of the CREATE TRIGGER statement, see Section 4.8.4, “Trigger Example”.

The syntax of the CREATE TRIGGER statement is:

```
create_trigger ::=  
CREATE TRIGGER trigger_name ON table_name time_of_operation  
  triggering_event [REFERENCING column_reference] trigger_body
```

where:

```
trigger_name           ::= literal
table_name            ::= literal
time_of_operation     ::= BEFORE | AFTER
triggering_event      ::= = INSERT | UPDATE | DELETE
column_reference     ::= {OLD | NEW} column_name [AS] col_identifier
                        [, REFERENCING column_reference]
```

```
trigger_body         ::= [declare_statement;...] trigger_statement; [trigger_statement;
```

```
old_column_name       ::= literal
new_column_name      ::= literal
old_col_identifier   ::= literal
new_col_identifier   ::= literal
new_col_identifier   ::= literal
```

## 4.6.3 Keywords and Clauses

Following is a summary of keywords and clauses.

### Trigger\_name

The *trigger\_name* can contain up to 254 characters.

### BEFORE | AFTER Clause

The BEFORE | AFTER clause specifies whether to execute the trigger before or after the invoking DML statement, which modifies data. In some circumstances, the BEFORE and AFTER clauses are interchangeable. However, there are some situations where one clause is preferred over the other.

- It is more efficient to use the BEFORE clause when performing data validation, such as domain constraint and referential integrity checking.
- When you use the AFTER clause, table rows which become available due to the invoking DML statement are processed. Conversely, the AFTER clause also confirms data deletion after the invoking DELETE statement.

You can define up to six triggers per table, one for each combination of table, event (INSERT, UPDATE, DELETE), and time (BEFORE and AFTER). For example, you can define one trigger for each BEFORE and AFTER clause, providing two triggers per DML operation. In addition, if you provide INSERT, UPDATE, and DELETE triggers to these combinations, you have a total maximum of six triggers.

The following example shows trigger trig01 defined BEFORE INSERT ON table t1.

```
"CREATE TRIGGER TRIG01 ON T1
  BEFORE INSERT
  REFERENCING NEW COL1 AS NEW_COL1
BEGIN
  EXEC SQL PREPARE CUR1
    INSERT INTO T2 VALUES (?);
  EXEC SQL EXECUTE CUR1 USING (NEW_COL1);
END"
```

Following are examples (including implications and advantages) of using the BEFORE and AFTER clause of the CREATE TRIGGER command for each DML operation:

- UPDATE Operation

The BEFORE clause can verify that modified data follows integrity constraint rules before processing the UPDATE. If the REFERENCING NEW AS *new\_col\_identifier* clause is used with the BEFORE UPDATE clause, then the updated values are available to the triggered SQL statements. In the trigger, you can set the default column values or derived column values before performing an UPDATE.

The AFTER clause can perform operations on newly modified data. For example, after a branch address update, the sales for the branch can be computed.

If the REFERENCING OLD AS *old\_col\_identifier* clause is used with the AFTER UPDATE clause, then the values that existed prior to the invoking update are accessible to the triggered SQL statements.

- INSERT Operation

The BEFORE clause can verify that new data follows integrity constraint rules before performing an INSERT. Column values passed as parameters are visible to the triggered SQL statements but the inserted rows are not. In the trigger, you can set default column values or derived column values before performing an INSERT.

The AFTER clause can perform operations on newly inserted data. For example, after insertion of a sales order, the total order can be computed to see if a customer is eligible for a discount.

Column values are passed as parameters and inserted rows are visible to the triggered SQL statements.

- DELETE Operation

The BEFORE clause can perform operations on data about to be deleted. Column values passed as parameters and inserted rows that are about to be deleted are visible to the triggered SQL statements.

The AFTER clause can be used to confirm the deletion of data. Column values passed as parameters are visible to the triggered SQL statements. Please note that the deleted rows are visible to the triggering SQL statement.

## INSERT | UPDATE | DELETE Clause

The INSERT | UPDATE | DELETE clause indicates the trigger action when a user action (INSERT, UPDATE, DELETE) is attempted.

Statements related to processing a trigger occur first before commits and autocommits from the invoking DML (INSERT, UPDATE, DELETE) statements on tables. If a trigger body or a procedure called within the trigger body attempts to execute a COMMIT or ROLLBACK, a IBM solidDB server returns an appropriate run-time error.

INSERT specifies that the trigger is activated by an INSERT on the table. Loading n rows of data is considered as n inserts.



### Note

There may be some performance impact if you try to load the data with triggers enabled. Depending on your business need, you may want to disable the triggers before loading and enable them after loading. For details, see Section 4.8.6, “Altering Trigger Attributes”.

DELETE specifies that the trigger is activated by a DELETE on the table.

UPDATE specifies that the trigger is activated by an UPDATE on the table. Note the following rules for using the UPDATE clause:

- Within the REFERENCES clause of a trigger, a column may be referenced (aliased) no more than once in the BEFORE sub-clause and once in the AFTER sub-clause. Also, if the column is referenced in both the BEFORE and AFTER sub-clauses, the column's alias must be different in each sub-clause.
- A IBM solidDB server allows for recursive update to the same table and does not prohibit recursive updates to the same row.

A IBM solidDB server does not detect situations where the actions of different triggers cause the same data to be updated. For example, assume there are two update triggers (one that is a BEFORE trigger and one that is an AFTER trigger) on table1. When an update is attempted on Table1, the two triggers are activated. Both triggers call stored procedures which update the same column, Col3, of a second table, Table2. The first trigger updates Table2.Col3 to 10 and the second trigger updates Table2.Col3 to 20.

Likewise, a IBM solidDB server does not detect situations where the result of an UPDATE which activates a trigger conflicts with the actions of the trigger itself. For example, consider the following SQL statement:

```
UPDATE t1 SET c1 = 20 WHERE c3 = 10;
```

If the trigger activated by this UPDATE then calls a procedure that contains the following SQL statement, the procedure overwrites the result of the UPDATE that activated the trigger:

```
UPDATE t1 SET c1 = 17 WHERE c1 = 20;
```



### Note

The above example can lead to recursive trigger execution, which you should try to avoid.

### Table\_name

The *table\_name* is the name of the table on which the trigger is created. IBM solidDB server allows you to drop a table that has dependent triggers defined on it. When you drop a table all dependent objects including triggers are dropped. Be aware that you may still get run-time errors. For example, assume you create two tables A and B. If a procedure SP-B inserts data into table A, and table A is then dropped, a user will receive a run-time error if table B has a trigger which invokes SP-B.

### Trigger\_body

The *trigger\_body* contains the statement(s) to be executed when a trigger fires. The rules for defining the body of a trigger are the same as the rules for defining the body of a stored procedure. Read Section 4.1, “Stored Procedures” for details on creating a stored procedure body.

A trigger body may also invoke any procedure registered with a IBM solidDB server. IBM solidDB procedure invocation rules follow standard procedure invocation practices.

You must explicitly check for business logic errors and raise an error.

### REFERENCING Clause

This clause is optional when creating a trigger on an INSERT/UPDATE/DELETE operation. It provides a way to reference the current column identifiers in the case of INSERT and DELETE operations, and both the old column identifier and the new updated column identifier by aliasing the column(s) on which an UPDATE operation occurs.



You must specify the OLD or NEW *col\_identifier* to access it. A IBM solidDB server does not provide access to the *col\_identifier* unless you define it using the REFERENCING subclause.

### **{OLD | NEW} *column\_name* AS *col\_identifier***

This subclause of the REFERENCING clause allow you to reference the values of columns both before and after an UPDATE operation. It produces a set of old and new column values which can be passed to a stored procedure; once passed, the procedure contains logic (for example, domain constraint checking) used to determine these parameter values.

Use the OLD AS clause to alias the table's old identifier as it exists before the UPDATE. Use the NEW AS clause to alias the table's new identifier as it exists after the UPDATE.

If you reference both the old and new values of the same column, you must use a different *col\_identifier*.

Each column that is referenced as NEW or OLD should have a separate REFERENCING subclause.

The statement atomicity in a trigger is such that operations made in a trigger are visible to the subsequent SQL statements inside the trigger. For example, if you execute an INSERT statement in a trigger and then also perform a select in the same trigger, then the inserted row is visible.

In the case of AFTER trigger, an inserted row or an updated row is visible in the AFTER insert trigger, but a deleted row cannot be seen for a select performed within the trigger. In the case of a BEFORE trigger, an inserted or updated row is invisible within the trigger and a deleted row is visible. In the case of an UPDATE, the pre-update values are available in a BEFORE trigger.

The table below summarizes the statement atomicity in a trigger, indicating whether the row is visible to the SELECT statement in the trigger body.

**Table 4.6. Statement Atomicity in a Trigger**

<b>Operation</b>	<b>BEFORE TRIGGER</b>	<b>AFTER TRIGGER</b>
INSERT	row is invisible	row is visible
UPDATE	previous value is visible	new value is visible
DELETE	row is visible	row is invisible

## **4.6.4 Triggers Comments and Restrictions**

- To use the stored procedure that a trigger calls, provide the catalog, schema/owner and name of the table on which the trigger is defined and specify whether to enable or disable the triggers on the table. For more details on stored procedures, read Section 4.7, “Triggers and Procedures”.

- To create a trigger on a table, you must have DBA authority or be the owner of the table on which the trigger is being defined.
- You can define, by default, up to one trigger for each combination of table, event (INSERT, UPDATE, DELETE) and time (BEFORE and AFTER). This means there can be a maximum of six triggers per table.



### Note

The triggers are applied to each row. This means that if there are ten inserts, a trigger is executed ten times.

- You cannot define triggers on a view (even if the view is based on a single table).
- You cannot alter a table that has a trigger defined on it when the dependent columns are affected.
- You cannot create a trigger on a system table.
- You cannot execute triggers that reference dropped or altered objects. To prevent this error:
  - Recreate any referenced object that you drop.
  - Restore any referenced object you changed back to its original state (known by the trigger).
- You can use reserved words in trigger statements if they are enclosed in double quotes. For example, the following CREATE TRIGGER statement references a column named "data", which is a reserved word.

```
"CREATE TRIGGER TRIG1 ON TMPT BEFORE INSERT
REFERENCING NEW "DATA" AS NEW_DATA
BEGIN
END"
```

## 4.7 Triggers and Procedures

Triggers can call stored procedures and cause a IBM solidDB server to execute other triggers. You can invoke procedures within a trigger body. In fact, you can define a trigger body that contains only procedure calls. A procedure invoked from a trigger body can invoke other triggers.

When using stored procedures within the trigger body, you must first store the procedure with the CREATE PROCEDURE statement.

In a procedure definition, you can use COMMIT and ROLLBACK statements. But in a trigger body, you *cannot* use COMMIT (including AUTOCOMMIT and COMMIT WORK) and ROLLBACK statements. You can use only the WHENEVER SQLERROR ABORT statement.

You can nest triggers up to 16 levels deep (the limit can be changed using a configuration parameter). If a trigger gets into an infinite loop, a IBM solidDB server detects this recursive action when the 16-level nesting (or system parameter) maximum is reached and returns an error to the user. For example, you could activate a trigger by attempting to insert into the table T1 and the trigger could call a stored procedure which also attempts to insert into T1, recursively activating the trigger.

If a set of nested triggers fails at any time, a IBM solidDB server rolls back the statement which originally activated the triggers.

### 4.7.1 Setting Default or Derived Columns

You can create triggers to set up default or derived column values in INSERT and UPDATE operations. When you create the trigger for this purpose using the CREATE TRIGGER command, the trigger must follow these rules:

- The trigger must be executed BEFORE the INSERT or UPDATE operation. Column values are modified with only a BEFORE trigger. Because the column value must be set before the INSERT or UPDATE operation, using the AFTER trigger to set column values is meaningless. Note also that the DELETE operation does not apply to modifying column values.
- For an INSERT and UPDATE operation, the REFERENCING clause must contain a NEW column value for modification. Note that modifying the OLD column value is meaningless.
- New column values can be set by simply changing the values of variables defined in the referencing section.

### 4.7.2 Using Parameters and Variables

When we update a record and that update invokes a trigger, the trigger itself may change the value of some columns within that record. In some situations, you may want to refer to both the "old" value and the "new" value within the trigger.

The REFERENCING clause allows you to create "aliases" for old and new values so that you can refer to either one within the same trigger. For example, assume there are two tables, one that holds customer information and one that holds invoice information. In addition to storing the amount of money billed for each invoice, the table contains a "total\_bought" field for each customer; this "total\_bought" field contains the cumulative total for all invoices ever sent to this customer. (This field might be used to identify high-volume customers.)

Any time the `total_amount` on an invoice is updated, the `total_bought` value for that customer's record in the customer table is also updated. To do this, the amount of the old value stored in the invoice is subtracted and the amount of the new value in the invoice is added. For example, if a customer's invoice used to be for \$100 and it is changed to \$150, then \$100 is subtracted and \$150 is added to the `total_bought` field. By properly using the `REFERENCING` clause, the trigger can "see" both the old value and the price column, thereby allowing the update of the `total_bought` column.

Note that the column aliases created by the `REFERENCING` clause are valid only within the trigger. Let's look at a pseudo-code example below:

```
CREATE TRIGGER pseudo_code_to_add_tax ON invoices
  AFTER UPDATE
  REFERENCING OLD total_price AS old_total_price,
  REFERENCING NEW total_price AS new_total_price
  BEGIN
    EXEC SQL PREPARE update_cursor
    UPDATE customers
      SET total_bought = total_bought - old_total_price
                                     + new_total_price;
  END
```

This example is "pseudo-code"; a real trigger would require some changes and additions (such as code to execute, close, and drop the cursor). A complete, valid SQL script for this example is provided below.

### Example 4.1. Trigger with Referencing Clause Example

```
-- This SQL sample demonstrates how to use the clause
-- "REFERENCING OLD AS old_col, REFERENCING NEW AS new_col"
-- to have simultaneous access to both the "OLD" and "NEW"
-- column values of the field while inside a trigger.
-- In this scenario, we have customers and invoices.
-- For each customer, we keep track of the cumulative total of
-- all purchases by that customer.
-- Each invoice stores the total amount of all purchases on
-- that invoice. If an total price on an invoice must be
-- adjusted, then the cumulative value of that customer's
-- purchases must also be adjusted.
-- Therefore, we update the cumulative total by subtracting
-- the "old" price on the invoice and adding the "new" price.
-- For example, if the amount on a customer's invoice was
```

```
-- changed from $100 to $150 (an increase of $50), then we
-- would update the customer's cumulative total by
-- subtracting $100 and adding $150 (a net increase of $50).
-- Drop the sample tables if they already exist.
DROP TABLE customers;
DROP TABLE invoices;
CREATE TABLE customers (
    customer_id INTEGER,    -- ID for each customer.
    total_bought FLOAT      -- The cumulative total price of
                            -- all this customer's purchases.
);
-- Each customer may have 0 or more invoices.
CREATE TABLE invoices (
    customer_id INTEGER,
    invoice_id INTEGER,    -- unique ID for each invoice
    invoice_total FLOAT    -- total price for this invoice
);
-- If the total_price on an invoice changes, then
-- update customers.total_bought to take into account
-- the change. Subtract the old invoice price and add the
-- new invoice price.
"CREATE TRIGGER old_and_new ON invoices
AFTER UPDATE
    REFERENCING OLD invoice_total AS old_invoice_total,
    REFERENCING NEW invoice_total AS new_invoice_total,
    -- If the customer_id doesn't change, we could use
    -- either the NEW or OLD customer_id.
    REFERENCING NEW customer_id AS new_customer_id
BEGIN
    EXEC SQL PREPARE upd_curs
        UPDATE customers
            SET total_bought = total_bought - ? + ?
            WHERE customers.customer_id = ?;
    EXEC SQL EXECUTE upd_curs
        USING (old_invoice_total, new_invoice_total,
            new_customer_id);
    EXEC SQL CLOSE upd_curs;
    EXEC SQL DROP upd_curs;
END";
-- When a new invoice is created, we update the total_bought
-- in the customers table.
"CREATE TRIGGER update_total_bought ON invoices
```

```
AFTER INSERT
REFERENCING NEW invoice_total AS new_invoice_total,
REFERENCING NEW customer_id AS new_customer_id
BEGIN
    EXEC SQL PREPARE ins_curs
        UPDATE customers
            SET total_bought = total_bought + ?
            WHERE customers.customer_id = ?;
    EXEC SQL EXECUTE ins_curs
        USING (new_invoice_total, new_customer_id);
    EXEC SQL CLOSE ins_curs;
    EXEC SQL DROP ins_curs;
END";
-- Insert a sample customer.
INSERT INTO customers (customer_id, total_bought)
VALUES (1000, 0.0);
-- Insert invoices for a customer; the INSERT trigger will
-- update the total_bought in the customers table.
INSERT INTO invoices (customer_id, invoice_id, invoice_total)
VALUES (1000, 5555, 234.00);
INSERT INTO invoices (customer_id, invoice_id, invoice_total)
VALUES (1000, 5789, 199.0);
-- Make sure that the INSERT trigger worked.
SELECT * FROM customers;
-- Now update an invoice; the total_bought in the customers
-- table will also be updated and the trigger that does
-- this will use the REFERENCING clauses
--     REFERENCING NEW invoice_total AS new_invoice_total,
--     REFERENCING OLD invoice_total AS old_invoice_total
UPDATE invoices SET invoice_total = 235.00
WHERE invoice_id = 5555;
-- Make sure that the UPDATE trigger worked.
SELECT * FROM customers;
COMMIT WORK;
```

## 4.8 Triggers and Transactions

Triggers require no commit from the invoking transaction in order to fire; DML statements alone cause triggers to fire. COMMIT WORK is also disallowed in a trigger body.

In a procedure definition, you can use COMMIT and ROLLBACK statements. But in a trigger body, you cannot use COMMIT and ROLLBACK statements. You can use only the WHENEVER SQLERROR ABORT statement. Note that if autocommit is on, then each statement inside the trigger is not treated as a separate statement and is not committed when it is executed; instead, the entire trigger body is executed as part of the INSERT, UPDATE, or DELETE statement that fired the trigger. Either the entire trigger (and the statement that fired it) is committed, or else the entire trigger (and the statement that fired it) is rolled back.

### 4.8.1 Recursion and Concurrency Conflict Errors

If a DML statement updates/deletes a row that causes a trigger to be fired, you cannot update/delete the same row again within that trigger. In such cases an AFTER trigger event can cause a recursion error and a BEFORE trigger event can cause a concurrency conflict error.

The following sections explain these terms, provide some examples of triggers that create these problems, and provide a table (shown in the section called "Summary of Trigger Cases"), that indicates the trigger situations that will and will not cause recursion errors or concurrency conflict errors.

#### Triggers and Recursion

A piece of code is "recursive" if the code causes itself to execute again. For example, a stored procedure that calls itself is recursive. Recursion in stored procedures is occasionally useful. On the other hand, triggers can create a slightly more subtle type of recursion, which is invalid and prohibited by the IBM solidDB server. A trigger that contains a statement that causes the same trigger to execute again on the same record is recursive. For example, a delete trigger would be recursive if it tries to delete the same record whose deletion fired the trigger.

If the database server were to allow recursion in triggers, then the server might go into an "infinite loop" and never finish executing the statement that fired the trigger. A concurrency conflict error occurs when a trigger executes an operation that "competes with" the statement that fired the trigger by trying to do the same type of action (for example, delete) within the same SQL statement. For example, if you create a trigger that is supposed to be fired when a record is deleted, and if that trigger tries to delete the same record whose deletion fired the trigger, then there are in essence two different "simultaneous" delete statements "competing" to delete the record; this results in a concurrency conflict. The following section provides an example of a defective delete trigger.

#### *Examples of Defective Triggers Causing Recursion*

The examples in this section explain just a few of the many restrictions and rules involving triggers.

In this scenario, an employee has resigned from a job and his or her medical coverage requires cancellation. The medical coverage also requires cancellation for the employee's dependents. A business rule for this situation is implemented by creating a trigger; the trigger is executed when an employee's record is deleted and the

statements inside the trigger then delete the employee's dependents. (This example assumes that the employees and their dependents are stored in the same table; in the real world, dependents are normally kept in a separate table. This example also assumes that each family has a unique last name.)

```
CREATE TRIGGER do_not_try_this ON employees_and_dependents
AFTER DELETE
REFERENCING OLD last_name AS old_last_name
BEGIN
    EXEC SQL PREPARE del_cursor
    DELETE FROM employees_and_dependents
    WHERE last_name = ?;
    EXEC SQL EXECUTE del_cursor USING (old_last_name);
    -- ... close and drop the cursor.
END;
```

Assume that an employee "John Smith" resigns and his medical coverage is deleted. When you delete "John Smith", the trigger is invoked immediately after John Smith is deleted and the trigger will try to delete ALL people named "John Smith", including not only the employee's dependents, but also the employee himself, since his name meets the criteria in the WHERE clause.

Every time an attempt is made to delete the employee's record, this action fires the trigger again. The code then recursively keeps trying to delete the employee by again firing the trigger, and again trying to delete. If the database server did not prohibit this or detect the situation, the server could go into an infinite loop. If the server detects this situation, it will give you an appropriate error, such as "Too many nested triggers."

A similar situation can happen with UPDATE. Assume that a trigger adds sales tax every time that a record is updated. Here's an example that causes a recursion error:

```
CREATE TRIGGER do_not_do_this_either ON invoice
AFTER UPDATE
REFERENCING NEW total_price AS new_total_price
BEGIN
    -- Add 8% sales tax.
    EXEC SQL PREPARE upd_curs1
    UPDATE invoice SET total_price = 1.08 * total_price
    WHERE ...;
    -- ... execute, close, and drop the cursor...
END;
```

In this scenario, customer Ann Jones calls up to change her order; the new price (with sales tax) is calculated by multiplying the new subtotal by 1.08. The record is updated with the new total price; each time the record



is updated, the trigger is fired, so updating the record once, causes the trigger to update it again and updates are repeated in an infinite loop.

If AFTER triggers can cause recursion or looping, what happens with BEFORE triggers? The answer is that, in some cases, BEFORE triggers can cause concurrency problems. Let's return to the first example of the trigger that deleted medical coverage for employees and their dependents. If the trigger were a BEFORE trigger (rather than an AFTER trigger), then just before the employee is deleted, we would execute the trigger, which in this case deletes everyone named John Smith. After the trigger is executed, the engine resumes its original task of dropping employee John Smith himself, but the server finds either he isn't there or that his record cannot be deleted because it has already been marked for deletion — in other words, there is a concurrency conflict because there are two separate efforts to delete the same record.

## Summary of Trigger Cases

In addition to the examples described in the previous section, the following table summarizes a number of additional cases, including those involving INSERTs, as well as UPDATES and DELETES.

The table is divided into the following five columns:

- *Trigger Mode* (that is, BEFORE or AFTER)
- *Operation* (INSERT, DELETE, or UPDATE)
- *Trigger Action* (what the trigger itself attempts to do, such as update the record that was just inserted)
- *Lock Type* ("optimistic" or "pessimistic")
- *Result* that you will see (for example, that the trigger action was successful, or that the trigger failed for a reason such as a recursion error like the one discussed in the previous section).

For details on interpreting a trigger entry in this table, see *Example Entry 1* later in this chapter.

**Table 4.7. Insert/Update/Delete Operations for BEFORE/AFTER Triggers**

Trigger Mode	Operation	Trigger Action	Lock Type	Result
AFTER	INSERT	UPDATE the same row by adding a number to the value	Optimistic	Record is updated.
AFTER	INSERT	UPDATE the same row by	Pessimistic	Record is updated.

#### 4.8.1 Recursion and Concurrency Conflict Errors

Trigger Mode	Operation	Trigger Action	Lock Type	Result
		adding a number to the value		
BEFORE	INSERT	UPDATE the same row by adding a number to the value	Optimistic	Record is not updated since the WHERE condition of the UPDATE within the trigger body returns a NULL resultset (as the desired row is not yet inserted in the table).
BEFORE	INSERT	UPDATE the same row by adding a number to the value	Pessimistic	Record is not updated since the WHERE condition of the UPDATE within the trigger body returns a NULL resultset (as the desired row is not yet inserted in the table).
AFTER	INSERT	DELETE the same row that is being inserted	Optimistic	Record is deleted.
AFTER	INSERT	DELETE the same row that is being inserted	Pessimistic	Record is deleted.
BEFORE	INSERT	DELETE the same row that is being inserted	Optimistic	Record is not deleted since the WHERE condition of the DELETE within the trigger body returns a NULL resultset (as the desired row is not yet inserted in the table).
BEFORE	INSERT	DELETE the same row that is being inserted	Pessimistic	Record is not updated since the WHERE condition of the UPDATE within the trigger body returns a NULL resultset (as the desired row is not yet inserted in the table).
AFTER	INSERT	INSERT a row	Optimistic	Too many nested triggers.
AFTER	INSERT	INSERT a row	Pessimistic	Too many nested triggers.
BEFORE	INSERT	INSERT a row	Optimistic	Too many nested triggers.
BEFORE	INSERT	INSERT a row	Pessimistic	Too many nested triggers.
AFTER	UPDATE	UPDATE the same row by adding a number to the value	Optimistic	Generates Solid Table Error: Too many nested triggers.
AFTER	UPDATE	UPDATE the same row by	Pessimistic	Generates Solid Table Error: Too many nested triggers.

#### 4.8.1 Recursion and Concurrency Conflict Errors

Trigger Mode	Operation	Trigger Action	Lock Type	Result
		adding a number to the value		
BEFORE	UPDATE	UPDATE the same row by adding a number to the value.	Optimistic	Record is updated, but does not get into a nested loop because the WHERE condition in the trigger body returns a NULL resultset and no rows are updated to fire the trigger recursively.
BEFORE	UPDATE	UPDATE the same row by adding a number to the value.	Pessimistic	Record is updated, but does not get into a nested loop because the WHERE condition in the trigger body returns a NULL resultset and no rows are updated to fire the trigger recursively.
AFTER	UPDATE	DELETE the same row that is being updated.	Optimistic	Record is deleted.
AFTER	UPDATE	DELETE the same row that is being updated.	Pessimistic	Record is deleted.
BEFORE	UPDATE	DELETE the same row that is being updated.	Optimistic	Concurrency conflict error.
BEFORE	UPDATE	DELETE the same row that is being updated.	Pessimistic	Concurrency conflict error.
AFTER	DELETE	INSERT a row with the same value.	Optimistic	Same record is inserted after deleting.
AFTER	DELETE	INSERT a row with the same value.	Pessimistic	Hangs at the time of firing the trigger.
BEFORE	DELETE	INSERT a row with the same value.	Optimistic	Same record is inserted after deleting
BEFORE	DELETE	INSERT a row with the same value.	Pessimistic	Hangs at the time of firing the trigger.

Trigger Mode	Operation	Trigger Action	Lock Type	Result
AFTER	DELETE	INSERT a row with the same value.	Optimistic	Record is deleted.
AFTER	DELETE	UPDATE the same row by adding a number to the value.	Pessimistic	Record is deleted.
BEFORE	DELETE	UPDATE the same row by adding a number to the value.	Optimistic	Record is deleted.
BEFORE	DELETE	UPDATE the same row by adding a number to the value	Pessimistic	Record is deleted.
AFTER	DELETE	DELETE same row	Optimistic	Too many nested triggers.
AFTER	DELETE	DELETE same record	Pessimistic	Too many nested triggers
BEFORE	DELETE	DELETE same record	Optimistic	Concurrency conflict error.
BEFORE	DELETE	DELETE same record	Pessimistic	Concurrency conflict error.

Here's an example entry from the table and an explanation of that entry:

**Table 4.8. Example Entry 1**

Trigger	Operation	Trigger Action	Lock Type	Result
AFTER	INSERT	UPDATE the same row by adding a number to the value	Optimistic	Record is updated.

In this situation, we have a trigger that fires AFTER an INSERT operation is done. The body of the trigger contains statements that update the same row as was inserted (that is, the same row as the one that fired the

trigger). If the lock type is "optimistic", then the result will be that the record gets updated. (Because there is no conflict, the locking [optimistic versus pessimistic] does not make a difference).

Note that in this case there is no recursion issue, even though we update the same row that we just inserted. The action that "fires" the trigger is not the same as the action taken inside the trigger, and so we do not create a recursive/looping situation.

Here's another example from the table:

**Table 4.9. Example Entry 2**

Trigger	Operation	Trigger Action	Lock Type	Result
BEFORE	INSERT	UPDATE the same row by adding a number to the value	Optimistic	Record is not updated since the WHERE condition of the UPDATE within the trigger body returns a NULL resultset (as the desired row is not yet inserted in the table).

In this case, we try to insert a record, but before the insertion takes place the trigger is run. In this case, the trigger tries to update the record (for example, to add sales tax to it). Since the record is not yet inserted, however, the UPDATE command inside the trigger does not find the record, and never adds the sales tax. Thus the result is the same as if the trigger had never fired. There is no error message, so you may not realize immediately that your trigger does not do what you intended.

## Flawed Trigger

Flawed trigger logic occurs in the following example in which the same row is deleted in a BEFORE UPDATE trigger; this causes IBM solidDB to generate a concurrency conflict error.

### Example 4.2. Flawed Trigger

```
DROP EMP ;
COMMIT WORK ;

CREATE TABLE EMP (C1 INTEGER) ;
INSERT INTO EMP VALUES (1) ;
COMMIT WORK ;

"CREATE TRIGGER TRIG1 ON EMP
  BEFORE UPDATE
  REFERENCING OLD C1 AS OLD_C1
BEGIN
```

```
EXEC SQL WHENEVER SQLERROR ABORT;  
EXEC SQL PREPARE CUR1 DELETE FROM EMP WHERE C1 = ?;  
EXEC SQL EXECUTE CUR1 USING (OLD_C1);  
END";
```

```
UPDATE EMP SET C1=200 WHERE C1 = 1;  
SELECT * FROM EMP;
```

```
ROLLBACK WORK;
```



### Note

If the row that is updated/deleted were based on a unique key, instead of an ordinary column (as in the example above), IBM solidDB generates the following error message: *1001: key value not found*.

To avoid recursion and concurrency conflict errors, be sure to check the application logic and take precautions to ensure the application does not cause two transactions to update or delete the same row.

## Error Handling

If a procedure returns an error to a trigger, the trigger causes its invoking DML command to fail with an error. To automatically return errors during the execution of a DML statement, you must use `WHENEVER SQLERROR ABORT` statement in the trigger body. Otherwise, errors must be checked explicitly within the trigger body after each procedure call or SQL statement.

For any errors in the user written business logic as part of the trigger body, users must use the `RETURN SQLERROR` statement. For details, see Section 4.8.3, “Raising Errors from Inside Triggers”.

If `RETURN SQLERROR` is not specified, then the system returns a default error message when the SQL statement execution fails. Any changes to the database due to the current DML statement are undone and the transaction is still active. In effect, transactions are not rolled back if a trigger execution fails, but the current executing statement is rolled back.



### Note

Triggered SQL statements are a part of the invoking transaction. If the invoking DML statement fails due to either the trigger or another error that is generated outside the trigger, all SQL statements within the trigger are rolled back along with the failed invoking DML command.

It is the responsibility of the invoking transaction to commit or rollback any DML statements executed within the trigger's procedure. However, this rule does not apply if the DML command invoking the trigger fails as

a result of the associated trigger. In this case, any DML statements executed within that trigger's procedure are automatically rolled back.

The COMMIT and ROLLBACK statements must be executed outside the trigger body and cannot be executed within the trigger body. If one executes COMMIT or ROLLBACK within the trigger body or within a procedure called from the trigger body or another trigger, the user will get a run-time error.

### **Nested and Recursive Triggers**

If a trigger gets into an infinite loop, a IBM solidDB server detects this recursive action when the 16-level nesting (or *MaxNestedTriggers* system parameter maximum is reached). For example, an insert attempt on table T1 activates a trigger and the trigger could call a stored procedure which also attempts to insert into Table T1, recursively activating the trigger. A IBM solidDB server returns an error on a user's insert attempt.

If a set of nested triggers fails at any time, a IBM solidDB server rolls back the command which originally activated the triggers.

## **4.8.2 Trigger Privileges and Security**

Because triggers can be activated by a user's attempt to INSERT, UPDATE, or DELETE data, no privileges are required to execute them.

When a user invokes a trigger, the user assumes the privileges of the owner of the table on which the trigger is defined. The action statements are executed on behalf of the table owner, not the user who activates the trigger. However, to create a trigger which uses a stored procedure requires that the creator of the trigger meet one of the following conditions:

- You have DBA privileges.
- You are the owner of the table on which the trigger is being defined.
- You were granted all privileges on the table.

If the creator has DBA authority and creates a table for another user, a IBM solidDB server assumes that unqualified names specified in the TRIGGER command belong to the user. For example, the following command is executed under DBA authority:

```
CREATE TRIGGER A.TRIG ON EMP BEFORE UPDATE
```

Since the EMP table is unqualified, the IBM solidDB server assumes that the qualified table name is A.EMP, not DBA.EMP.

## 4.8.3 Raising Errors from Inside Triggers

At times, it is possible to receive an error in executing a trigger. The error may be due to execution of SQL statements or business logic.

Users can receive any errors in a procedure variable using the SQL statement:

```
RETURN SQLERROR error_string
```

or

```
RETURN SQLERROR char_variable
```

The error is returned in the following format:

```
User error: error_string
```

If a user does not specify the RETURN SQLERROR statement in the trigger body, then all trapped SQL errors are raised with a default *error\_string* determined by the system. For details, see the appendix, "Error Codes" in the documentation for your IBM solidDB product.

## 4.8.4 Trigger Example

### Example 4.3. Trigger Example

This example shows how simple triggers work. It contains some triggers that work correctly and some triggers that contain errors. For the successful triggers in the example, a table (named *trigger\_test*) is created and six triggers are created on that table. Each trigger, when fired, inserts a record into another table (named *trigger\_output*). After performing the DML statements (INSERT, UPDATE, and DELETE) that fire the triggers, the results of the triggers are displayed by selecting all records from the *trigger\_output* table.

```
DROP TABLE TRIGGER_TEST;  
DROP TABLE TRIGGER_ERR_TEST;  
DROP TABLE TRIGGER_ERR_B_TEST;  
DROP TABLE TRIGGER_ERR_A_TEST;  
DROP TABLE TRIGGER_OUTPUT;  
COMMIT WORK;  
-- Create a table that has a column for each of the possible trigger
```



## 4.8.4 Trigger Example

---

```
-- types (for example, BI = a trigger that is on Insert
-- operations and that executes as a "Before" trigger).
CREATE TABLE TRIGGER_TEST(
    XX VARCHAR,
    BI VARCHAR,  -- BI = Before Insert
    AI VARCHAR,  -- AI = After Insert
    BU VARCHAR,  -- BU = Before Update
    AU VARCHAR,  -- AU = After Update
    BD VARCHAR,  -- BD = Before Delete
    AD VARCHAR   -- AD = After Delete
);
COMMIT WORK;

-- Table for 'before' trigger errors
CREATE TABLE TRIGGER_ERR_B_TEST(
    XX VARCHAR,
    BI VARCHAR,
    AI VARCHAR,
    BU VARCHAR,
    AU VARCHAR,
    BD VARCHAR,
    AD VARCHAR
);

INSERT INTO TRIGGER_ERR_B_TEST VALUES('x','x','x','x','x',
    'x','x');
COMMIT WORK;

-- Table for 'after X' trigger errors
CREATE TABLE TRIGGER_ERR_A_TEST(
    XX VARCHAR,
    BI VARCHAR,  -- Before Insert
    AI VARCHAR,  -- After Insert
    BU VARCHAR,  -- Before Update
    AU VARCHAR,  -- After Update
    BD VARCHAR,  -- Before Delete
    AD VARCHAR   -- After Delete
);

INSERT INTO TRIGGER_ERR_A_TEST VALUES('x','x','x','x','x',
    'x','x');
COMMIT WORK;
```

## 4.8.4 Trigger Example

---

```
CREATE TABLE TRIGGER_OUTPUT(  
    TEXT VARCHAR,  
    NAME VARCHAR,  
    SCHEMA VARCHAR  
);  
COMMIT WORK;  
  
-----  
-- Successful triggers  
-----  
-- Create a "Before" trigger on insert operations. When a record is  
-- inserted into the table named trigger_test, then this trigger is  
-- fired. When this trigger is fired, it inserts a record into the  
-- "trigger_output" table to show that the trigger actually executed.  
  
"CREATE TRIGGER TRIGGER_BI ON TRIGGER_TEST  
    BEFORE INSERT  
    REFERENCING NEW BI AS NEW_BI  
BEGIN  
    EXEC SQL PREPARE BI INSERT INTO TRIGGER_OUTPUT VALUES(  
        'BI', TRIG_NAME(0), TRIG_SCHEMA(0));  
    EXEC SQL EXECUTE BI;  
    SET NEW_BI = 'TRIGGER_BI';  
END";  
COMMIT WORK;  
  
"CREATE TRIGGER TRIGGER_AI ON TRIGGER_TEST  
    AFTER INSERT  
    REFERENCING NEW AI AS NEW_AI  
BEGIN  
    EXEC SQL PREPARE AI INSERT INTO TRIGGER_OUTPUT VALUES(  
        'AI', TRIG_NAME(0), TRIG_SCHEMA(0));  
    EXEC SQL EXECUTE AI;  
    SET NEW_AI = 'TRIGGER_AI';  
END";  
COMMIT WORK;  
  
"CREATE TRIGGER TRIGGER_BU ON TRIGGER_TEST  
    BEFORE UPDATE  
    REFERENCING NEW BU AS NEW_BU  
BEGIN
```

## 4.8.4 Trigger Example

---

```
        EXEC SQL PREPARE BU INSERT INTO TRIGGER_OUTPUT VALUES(
'BU', TRIG_NAME(0), TRIG_SCHEMA(0));
        EXEC SQL EXECUTE BU;
        SET NEW_BU = 'TRIGGER_BU';
END";
COMMIT WORK;

"CREATE TRIGGER TRIGGER_AU ON TRIGGER_TEST
    AFTER UPDATE
    REFERENCING NEW AU AS NEW_AU
BEGIN
    EXEC SQL PREPARE AU INSERT INTO TRIGGER_OUTPUT VALUES(
'AU', TRIG_NAME(0), TRIG_SCHEMA(0));
    EXEC SQL EXECUTE AU;
    SET NEW_AU = 'TRIGGER_AU';
END";
COMMIT WORK;

"CREATE TRIGGER TRIGGER_BD ON TRIGGER_TEST
    BEFORE DELETE
    REFERENCING OLD BD AS OLD_BD
BEGIN
    EXEC SQL PREPARE BD INSERT INTO TRIGGER_OUTPUT VALUES(
'BD', TRIG_NAME(0), TRIG_SCHEMA(0));
    EXEC SQL EXECUTE BD;
    SET OLD_BD = 'TRIGGER_BD';
END";
COMMIT WORK;

"CREATE TRIGGER TRIGGER_AD ON TRIGGER_TEST
    AFTER DELETE
    REFERENCING OLD AD AS OLD_AD
BEGIN
    EXEC SQL PREPARE AD INSERT INTO TRIGGER_OUTPUT VALUES(
'AD', TRIG_NAME(0), TRIG_SCHEMA(0));
    EXEC SQL EXECUTE AD;
    SET OLD_AD = 'TRIGGER_AD';
END";
COMMIT WORK;
```

```
-----
-- This attempt to create a trigger will fail. The statement
```

## 4.8.4 Trigger Example

---

```
-- specifies the wrong data type for the error variable named
-- ERRSTR.
-----

"CREATE TRIGGER TRIGGER_ERR_AU ON TRIGGER_ERR_A_TEST
    AFTER UPDATE
    REFERENCING NEW AU AS NEW_AU
BEGIN
    -- The following line is incorrect; ERRSTR must be declared
    -- as VARCHAR, not INTEGER;
    DECLARE ERRSTR INTEGER;
    -- ...
    RETURN SQLERROR ERRSTR;
END";
COMMIT WORK;

-----

-- Trigger that returns an error message.
-----

"CREATE TRIGGER TRIGGER_ERR_BI ON TRIGGER_ERR_B_TEST
    BEFORE INSERT
    REFERENCING NEW BI AS NEW_BI
BEGIN
    -- ...
    RETURN SQLERROR 'Error in TRIGGER_ERR_BI';
END";
COMMIT WORK;

-----

-- Success trigger tests. These Insert, Update, and Delete
-- statements will force the triggers to fire. The SELECT
-- statements will show you the records in the trigger_test and
-- trigger_output tables.
-----

INSERT INTO TRIGGER_TEST(XX) VALUES ('XX');
COMMIT WORK;

-- Show the records that were inserted into the trigger_test
-- table. (The records for trigger_output are shown later.)
```

```
SELECT * FROM TRIGGER_TEST;
COMMIT WORK;

UPDATE TRIGGER_TEST SET XX = 'XX updated';
COMMIT WORK;

-- Show the records that were inserted into the trigger_test
-- table. (The records for trigger_output are shown later.)

SELECT * FROM TRIGGER_TEST;
COMMIT WORK;

DELETE FROM TRIGGER_TEST;
COMMIT WORK;

SELECT * FROM TRIGGER_TEST;

-- Show that the triggers did run and did add values to the
-- trigger_output table. You should see 6 records one for
-- each of the triggers that executed. The 6 triggers are:
--     BI, AI, BU, AU, BD, AD.

SELECT * FROM TRIGGER_OUTPUT;
COMMIT WORK;

-----
-- Error trigger test
-----

INSERT INTO TRIGGER_ERR_B_TEST(XX) VALUES ('XX');
COMMIT WORK;
```

## 4.8.5 Dropping Triggers

To drop a trigger defined on a table, use the `DROP TRIGGER` command. This command drops the trigger from the system catalog.

You must be the owner of a table, or a user with DBA authority, to drop a trigger from the table.

The syntax is:

```
DROP TRIGGER [[catalog_name.]schema_name.]trigger_name
DROP TRIGGER trigger_name
DROP TRIGGER schema_name.trigger_name
DROP TRIGGER catalog_name.schema_name.trigger_name
```

The *trigger\_name* is the name of the trigger on which the table is defined.

If the trigger is part of a schema, indicate the schema name as in:

```
schema_name.trigger_name
```

If the trigger is part of a catalog, indicate the catalog name as in:

```
catalog_name.schema_name.trigger_name
```

### Example 4.4. Dropping and Recreating a Trigger

```
DROP TRIGGER TRIGGER_BI;
COMMIT WORK;

"CREATE TRIGGER TRIGGER_BI ON TRIGGER_TEST
    BEFORE INSERT
    REFERENCING NEW BI AS NEW_BI
BEGIN
    EXEC SQL PREPARE BI INSERT INTO TRIGGER_OUTPUT VALUES(
        'BI_NEW', TRIG_NAME(0), TRIG_SCHEMA(0));
    EXEC SQL EXECUTE BI;
    SET NEW_BI = 'TRIGGER_BI_NEW';
END";
COMMIT WORK;

INSERT INTO TRIGGER_TEST(XX) VALUES ('XX');
COMMIT WORK;

SELECT * FROM TRIGGER_TEST;
```

```
SELECT * FROM TRIGGER_OUTPUT;  
COMMIT WORK;
```

## 4.8.6 Altering Trigger Attributes

You can alter trigger attributes using the ALTER TRIGGER command. The valid attributes are ENABLED and DISABLED trigger.

The ALTER TRIGGER SET DISABLED command causes a IBM solidDB server to ignore the trigger when an activating DML statement is issued. With ALTER TRIGGER SET ENABLED statement, you can enable a trigger that is currently inactive.

You must be the owner of a table, or a user with DBA authority to alter a trigger from the table.

```
alter_trigger ::=  
    ALTER TRIGGER trigger_name_attr SET ENABLED | DISABLED  
trigger_name_attr ::= [catalog_name.[schema_name]]trigger_name
```

For example:

```
ALTER TRIGGER trig_on_employee SET ENABLED;
```

## 4.8.7 Obtaining Trigger Information

You obtain trigger information by using trigger functions that return specific information and performing a query on the trigger system table. Each of these sources is described in this section.

## 4.8.8 Trigger Functions

The following system supported triggers stack functions are useful for analyzing and debugging purposes.



### Note

The trigger stack refer to those triggers that are cached, regardless of whether they are executed or detected for execution. Trigger stack functions can be used in the application program like any other function.

The functions are:

- TRIG\_COUNT ( )

This function returns the number of triggers in the trigger stack, including the current trigger. The return value is an integer.

- TRIG\_NAME ( n )

This function returns the nth trigger name in the trigger stack. The first trigger position or offset is zero.

- TRIG\_SCHEMA ( n )

This function returns the nth trigger schema name in the trigger stack. The first trigger position or offset is zero. The return value is a string.

## 4.8.9 SYS\_TRIGGERS System Table

Triggers are stored in a system table called SYS\_TRIGGERS. The following is the meta data for the SYS\_TRIGGERS system table:

**Table 4.10. Meta Data for the SYS\_TRIGGERS System Table**

Column Name	Data Type	Description
ID	INTEGER	unique table identifier (primary key)
TRIGGER_NAME	WVARCHAR	trigger name (unique with schema)
TRIGGER_TEXT	LONG WVARCHAR	trigger body
TRIGGER_BIN	LONG VARBINARY	compiled form of the trigger
TRIGGER_SCHEMA	WVARCHAR	the schema in which the trigger was created
TRIGGER_CATALOG	WVARCHAR	the catalog in which the trigger was created
CREATIME	TIMESTAMP	the creation time of the trigger
TYPE	INTEGER	reserved for future use
REL_ID	INTEGER	the relation id (unique with type)
TRIGGER_ENABLED	WVARCHAR	'YES' if the trigger is enabled; 'NO' if the trigger is disabled.



## 4.8.10 Trigger Parameter Settings

### Setting Nested Trigger Maximum

Triggers can invoke other triggers or a trigger can invoke itself (recursive trigger). The maximum number of nested or recursive triggers can be configured by the *MaxNestedTriggers* system parameter in the *SQL* section of *solid.ini*.

```
[SQL]
MaxNestedTriggers = n;
```

where *n* is the maximum number of nested triggers.

The default number for nested triggers is 16.

### Setting the Trigger Cache

In a IBM solidDB server, triggers are cached in a separate cache. Each user has a separate cache for triggers. As the triggers are executed, the trigger procedure logic is cached in the trigger cache and is reused when the trigger is executed again.

You can set the size of the trigger cache using the *TriggerCache* system parameter in the *SQL* section of *solid.ini*.

```
[SQL]
TriggerCache = n;
```

where *n* is the number of triggers being reserved for the cache.

## 4.9 Deferred Procedure Calls

At the end of a committed transaction, you may want to perform a specific action. For example, if the transaction updated some data in a "master" publication, then you may want to notify a replica that the master data was updated. IBM solidDB allows the *START AFTER COMMIT* statement to specify an SQL statement that will be executed when the current transaction is committed. The specified SQL statement is called the "body" of the *START AFTER COMMIT*. The body is executed asynchronously in a separate connection.

For example, if you would like to call a stored procedure named *my\_proc()* when the transaction commits, then you would write:

```
START AFTER COMMIT NONUNIQUE CALL
    my_proc;
```

This statement may appear anywhere inside the transaction; it may be the first statement, the last statement, or any statement in between. Regardless of where the `START AFTER COMMIT` statement itself appears within the transaction, the "body" (the call to `my_proc`) will be executed only when the transaction is committed. In the example above, we put the body on a separate line, but that is not required syntactically.

Because the body of the statement is not executed at the same time as the `START AFTER COMMIT` statement itself, we say that there are two different phases to the `START AFTER COMMIT` command: the "definition" phase and the "execution" phase. In the definition phase of `START AFTER COMMIT`, you specify the body but don't execute it. The creation phase may occur anywhere inside a transaction; in other words, the statement "`START AFTER COMMIT ...`" may be placed in any order relative to other SQL statements in the same transaction.

In the execution phase, the body of the `START AFTER COMMIT` statement is actually executed. The execution phase occurs when the `COMMIT WORK` statement for the transaction is executed. (It is also possible to execute a `START AFTER COMMIT` in autocommit mode, but there is rarely a reason to do this.)

Below is an example that shows the use of a `START AFTER COMMIT` statement inside a transaction.

```
-- Any valid SQL statement(s)...
...
-- Creation phase. The function my_proc() is not actually called here.
START AFTER COMMIT NONUNIQUE CALL my_proc(x, y);
...
-- Any valid SQL statement(s)...

-- Execution phase: This ends the transaction and starts execution
-- of the call to my_proc().
COMMIT WORK;
```

A `START AFTER COMMIT` does not execute unless and until the transaction is successfully committed. If the transaction containing the `START AFTER COMMIT` is rolled back, then the body of the `START AFTER COMMIT` is not executed. If you want to propagate the updated data from a replica to a master, then this is an advantage because you only want the data propagated if it is committed. If you were to use triggers to start the propagation, the data would be propagated before it was committed.

The `START AFTER COMMIT` command applies only to the current transaction, i.e. the one that the `START AFTER COMMIT` command was issued inside. It does not apply to subsequent transactions, or to any other transactions that are currently open in other connections.

The `START AFTER COMMIT` command allows you to specify only one SQL statement to be executed when the `COMMIT` occurs. However, that one SQL statement may be a call to a stored procedure, and that stored procedure may have many statements, including calls to other stored procedures. Furthermore, you may have more than one `START AFTER COMMIT` command per transaction. The body of each of these `START AFTER COMMIT` statements will be executed when the transaction is committed. However, these bodies will run independently and asynchronously; they will not necessarily execute in the same order as their corresponding `START AFTER COMMIT` statements, and they are likely to have overlapping execution (there is no guarantee that one will finish before the next one starts).

A common use of `START AFTER COMMIT` is to help implement "Sync Pull Notify" ("Push Synchronization"), which is discussed in *IBM solidDB Advanced Replication Guide*.

If the body of your `START AFTER COMMIT` is a call to a stored procedure, that procedure may be local or it may be remote on one remote replica (or master).

If you are using Sync Pull Notify, then you may want to call the same procedure on many replicas. To do this, you must use a slightly indirect method. The simplest method is to write one local procedure that calls many procedures on replicas. For example, if the body of the `START AFTER COMMIT` statement is "`CALL my_proc`", then you could write `my_proc` to be similar to the following:

```
CREATE PROCEDURE my_proc
BEGIN
CALL update_inventory(x) AT replica1;
CALL update_inventory(x) AT replica2;
CALL update_inventory(x) AT replica3;
END;
```

This approach works fine if your list of replicas is static. However, if you expect to add new replicas in the future, you may find it more convenient to update "groups" of replicas based on their properties. This allows you to add new replicas with specific properties and then have existing stored procedures operate on those new replicas. This is done by making use of two features: the `FOR EACH REPLICA` clause in `START AFTER COMMIT`, and the `DEFAULT` clause in remote stored procedure calls.

If the `FOR EACH REPLICA` clause is used in `START AFTER COMMIT`, the statement will be executed once for each replica that meets the conditions in the `WHERE` clause. Note that the statement is executed once `FOR` each replica, not once `ON` each replica. If there is no "`AT node-ref`" clause in the `CALL` statement, then the stored procedure is called locally, i.e. on the same server as the `START AFTER COMMIT` was executed on. To make sure that a stored procedure is called once `ON` each replica, you must use the `DEFAULT` clause. The typical way to do this is to create a local stored procedure that contains a remote procedure calling that uses the `DEFAULT` clause. For example, suppose that `my_local_proc` contains the following:

```
CALL update_sales_statistics AT DEFAULT;
```

and suppose that the `START AFTER COMMIT` statement is

```
START AFTER COMMIT FOR EACH REPLICA
WHERE region = 'north'
UNIQUE
CALL my_local_proc;
```

The `WHERE` clause is

```
WHERE region = 'north'
```

Therefore, for each replica that has the properties

```
region = 'north'
```

we will call the stored procedure named `my_local_proc`. That local procedure, in turn, executes

```
CALL update_sales_statistics() AT DEFAULT
```

The keyword `DEFAULT` is resolved as the name of the replica. Each time that `my_local_proc` is called from inside the body of the `START AFTER COMMIT`, the `DEFAULT` keyword is the name of a different replica that has the property `"region = 'north'"`.

For more information about property/value pairs such as `"region = 'north'"`, please see the *IBM solidDB Advanced Replication Guide's* discussion of Replica Property Names.

Note that it's possible that not all replicas will have a procedure named `update_sales_statistics()`. If this is the case, then the procedure will only be executed on those replicas that have the procedure. (The master will not send each replica a copy of the procedure; the master only calls existing procedures.)

Note also that it's possible that not all replicas that have a procedure named `update_sales_statistics()` will have the `SAME` procedure. Each replica may have its own custom version of the procedure.

Naturally, before executing each statement on each replica, a connection to the replica is established.

When the `START AFTER COMMIT` command is used to call multiple replicas, this enables the use of the optional keyword "DEFAULT" in the syntax of the `CALL` command. For example, suppose that you use the following:

```
START AFTER COMMIT
  FOR EACH REPLICA
  WHERE location = 'India'
  UNIQUE CALL push;
```

Then in the local procedure 'push' you can use the keyword "DEFAULT", which acts as a variable that contains the name of the replica in question.

```
CREATE PROCEDURE push
BEGIN
EXEC SQL EXECDIRECT CALL remoteproc AT DEFAULT;
END
```

Procedure 'push' will be called once for each replica that has a property named 'location' with value 'India'. Each time the procedure is called, "DEFAULT" will be set to the name of that replica. Thus

```
CALL remoteproc AT DEFAULT;
```

will call the procedure on that particular replica.

You can set the replica properties in the master with the statement:

```
SET SYNC PROPERTY propname = 'value' FOR REPLICA replica_name;
```

for example

```
SET SYNC PROPERTY location = 'India' FOR REPLICA asia_hq;
```

For more details about the DEFAULT keyword, see the section titled "More on the DEFAULT keyword..." below.

The statement specified in `START AFTER COMMIT` is executed as an independent transaction. It is not part of the transaction that contained the `START AFTER COMMIT` command. This independent transaction is run as though autocommit mode were on; in other words, you do not need an explicit `COMMIT WORK` to commit the work done in this statement.

In other respects, however, the execution of the statement is not much like a transaction. First, there is no guarantee that the statement will execute to completion. The statement is launched as an independent background task. If the server crashes, or if for some other reason the statement cannot be executed, then the statement disappears without being completely executed.

Second, because the statement is executed as a background task, there is no mechanism for returning an error. Third, there is no way to roll back the statement; if the statement execution is completed, the "transaction" statement is autocommitted regardless of whether any errors were detected. (Note that if the statement is a procedure call, then the procedure itself may contain COMMIT and ROLLBACK commands.)

You may use the "RETRY" clause to try executing the statement more than once if it fails. The RETRY clause allows you to specify the number of times the server should attempt to retry the failed statement. You must specify the number of seconds to wait between each retry.

If you do not use the RETRY clause, the server attempts only once execute the statement, then the statement is discarded. If, for example, the statement tries to call a remote procedure, and if the remote server is down (or cannot be contacted due to a network problem), then the statement will not be executed and you will not get any error message.

Any statement, including the statement specified in a START AFTER COMMIT, executes in a certain "context". The context includes such factors as the default catalog, the default schema, etc. For a statement executed from within a START AFTER COMMIT, the statement's context is based on the context at the time that the START AFTER COMMIT is executed, not on the context at the time of the COMMIT WORK that actually causes the statement inside START AFTER COMMIT to run. In the example below, 'CALL FOO\_PROC' is executed in the catalog foo\_cat and schema foo\_schema, not bar\_cat and bar\_schema.

```
SET CATALOG FOO_CAT;  
SET SCHEMA FOO_SCHEMA;  
START AFTER COMMIT UNIQUE CALL FOO_PROC;  
...  
SET CATALOG BAR_CAT;  
SET SCHEMA BAR_SCHEMA;  
COMMIT WORK;
```

The UNIQUE/NONUNIQUE keywords determine whether the server tries to avoid issuing the same command twice.

The UNIQUE keyword before <stmt> defines that the statement is executed only if there isn't identical statement under execution or "pending" for execution. Statements are compared with simple string compare. So for example 'call foo(1)' is different from 'call foo(2)'. Replicas are also taken into account in the comparison; in other words, UNIQUE does not prevent the server from executing the same trigger call on different

replicas. Note that "unique" only blocks overlapping execution of statements; it does not prevent the same statement from being executed again later if it is called again after the current invocation has finished running.

NONUNIQUE means that duplicate statements can be executed simultaneously in the background.

Examples: The following statements are all considered different and are thus executed even though each contains the UNIQUE keyword. (Name is a unique property of replica.)

```
START AFTER COMMIT UNIQUE call myproc;
START AFTER COMMIT FOR EACH REPLICA WHERE name='R1' UNIQUE call myproc;
START AFTER COMMIT FOR EACH REPLICA WHERE name='R2' UNIQUE call myproc;
START AFTER COMMIT FOR EACH REPLICA WHERE name='R3' UNIQUE call myproc;
```

But if the following statement is executed in the same transaction as the previous ones and if some of the replicas R1, R2, and R3 have the property "color='blue'", then the call is not executed for those replicas again.

```
START AFTER COMMIT FOR EACH REPLICA WHERE color='blue'
UNIQUE call myproc;
```

Note that uniqueness also does not prevent "automatic" execution from overlapping "manual" execution. For example, if you manually execute a command to refresh from a particular publication, and if the master also calls a remote stored procedure to refresh from that publication, the master won't "skip" the call because a manual refresh is already running. Uniqueness applies only to statements started by START AFTER COMMIT.

The START AFTER COMMIT statement can be used inside a stored procedure. For example, suppose that you want to post an event if and only if a transaction completed successfully. You could write a stored procedure that would execute a START AFTER COMMIT statement that would post the event if the transaction was committed (but not if it was rolled back). Your code might look similar to the following:

This sample also contains an example of "receiving" and then using an event parameter. See the stored procedure named "wait\_on\_event\_e" in script #1.

```
-- To run this demo properly, you will need two users/connections.
-- This demo contains 5 separate "scripts", which must be executed
-- in the order shown below:
--     User1 executes the first script.
--     User2 executes the second script.
--     User1 executes the third script.
--     User2 executes the fourth script.
--     User1 executes the fifth script.
```

```
-- You may notice that there are some COMMIT WORK statements
-- in surprising places.  These are to ensure that each user sees the
-- most recent changes of the other user.  Without the COMMIT WORK
-- statements, in some cases one user would see an out-of-date
-- "snapshot" of the database.
--
-- Please set autocommit off for both users/connections!

----- SCRIPT 1 (USER 1) -----
CREATE EVENT e (i int);
CREATE TABLE table1 (a int);

-- This inserts a row into table1. The value inserted into the is copied
-- from the parameter to the procedure.
"CREATE PROCEDURE inserter(i integer)
BEGIN
EXEC SQL PREPARE c_inserter INSERT INTO table1 (a) VALUES (?);
EXEC SQL EXECUTE c_inserter USING (i);
EXEC SQL CLOSE c_inserter;
EXEC SQL DROP c_inserter;
END";

-- This posts the event named "e".
"CREATE PROCEDURE post_event(i integer)
BEGIN
POST EVENT e(i);
END";

-- This demonstrates the use of START AFTER COMMIT inside a
-- stored procedure.  After you call this procedure and
-- call COMMIT WORK, the server will post the event.
"CREATE PROCEDURE sac_demo
BEGIN
DECLARE MyVar INT;
MyVar := 97;
EXEC SQL PREPARE c_sacdemo START AFTER COMMIT NONUNIQUE CALL
  post_event(?);
EXEC SQL EXECUTE c_sacdemo USING (MyVar);
EXEC SQL CLOSE c_sacdemo;
```



```
EXEC SQL DROP c_sacdemo;
END";

-- When user2 calls this procedure, the procedure will wait until
-- the event named "e" is posted, and then it will call the
-- stored procedure that inserts a record into table1.
"CREATE PROCEDURE wait_on_event_e
BEGIN
-- Declare the variable that will be used to hold the event parameter.
-- Although the parameter was declared when the event was created, you
-- still need to declare it as a variable in the procedure that receives
-- that event.
DECLARE i INT;
WAIT EVENT
  WHEN e (i) BEGIN
    -- After we receive the event, insert a row into the table.
    EXEC SQL PREPARE c_call_inserter CALL inserter(?);
    EXEC SQL EXECUTE c_call_inserter USING (i);
    EXEC SQL CLOSE c_call_inserter;
    EXEC SQL DROP c_call_inserter;
  END EVENT
END WAIT
END";

COMMIT WORK;

----- SCRIPT 2 (USER 2) -----
-- Make sure that user2 sees the changes that user1 made.
COMMIT WORK;

-- Wait until user1 posts the event.
CALL wait_on_event_e;
-- Don't commit work again (yet).

----- SCRIPT 3 (USER 1) -----
COMMIT WORK;

-- User2 should be waiting on event e, and should see the event after
-- we execute the stored procedure named sac_demo and then commit work.
-- Note that since START AFTER COMMIT statements are executed
```

```
-- asynchronously, there may be a slight delay between the COMMIT WORK
-- and the associated POST EVENT.
CALL sac_demo;
COMMIT WORK;

----- SCRIPT 4 (USER 2) -----
-- Commit the INSERT that we did earlier when we called inserter()
-- after receiving the event.
COMMIT WORK;

-----SCRIPT 5 (USER 1) -----
-- Ensure that we see the data that user2 inserted.
COMMIT WORK;

-- Show the record that user2 inserted.
SELECT * FROM table1;

COMMIT WORK;
```

There are several important things that you should know about `START AFTER COMMIT`.

- When the body of the deferred procedure call (`START AFTER COMMIT`) is executed, it runs asynchronously in the background. This allows the server to immediately start executing the next SQL command in your program without waiting for the deferred procedure call statement to finish. It also means that you do not have to wait for completion before disconnecting from the server. In most situations, this is an advantage. However, in a few situations this may be a disadvantage. For example, if the body of the deferred procedure call locks records that are needed by subsequent SQL commands in your program, you may not appreciate having the body of the deferred procedure call run in the background while your next SQL command runs in the foreground and has to wait to access those same records. (For a way around this, see below...)
- The statement to be executed will only be executed if the transaction is completed with a `COMMIT`, not a `ROLLBACK`. If the entire transaction is explicitly rolled back, or if the transaction is aborted and thus implicitly rolled back (due to a failed connection, for example), then the body of the `START AFTER COMMIT` will not be executed.
- Although the transaction in which the deferred procedure call occurs can be rolled back (thus preventing the body of the deferred procedure call from running), the body of the deferred procedure call cannot itself be rolled back if it has executed. Because it runs asynchronously in the background, there is no mechanism for cancelling or rolling back the body once it starts executing.

- The statement in the deferred procedure call is not guaranteed to run until completion or to be run as an "atomic" transaction. For example, if your server crashes, then the statement will not resume executing the next time that the server starts, and any actions that were completed before the server crashed may be kept. To prevent inconsistent data in this type of situation, you must program carefully and make proper use of features like referential constraints to ensure data integrity.
- If you execute a `START AFTER COMMIT` statement in autocommit mode, then the body of the `START AFTER COMMIT` will be executed "immediately" (i.e. as soon as the `START AFTER COMMIT` is executed and automatically committed). At first, this might seem useless — why not just execute the body of the `START AFTER COMMIT` directly? There are a few subtle differences, however. First, a direct call to `my_proc` is synchronous; the server will not return control to you until the stored procedure has finished executing. If you call `my_proc` as the body of a `START AFTER COMMIT`, however, then the call is asynchronous; the server does not wait for the end of `my_proc` before allowing you to execute the next SQL statement. In addition, because `START AFTER COMMIT` statements are not truly executed "immediately" (i.e. at the time that the transaction is committed) but may instead be delayed briefly if the server is busy, you might or might not actually start running your next SQL statement before `my_proc` even starts executing. It is rare for this to be desirable behavior. However, if you truly want to launch an asynchronous stored procedure that will run in the background while you continue onward with your program, it is valid to do `START AFTER COMMIT` in autocommit mode.
- If more than one deferred procedure call was executed in the same transaction, then the bodies of all the `START AFTER COMMIT` statements will run asynchronously. This means that they will not necessarily run in the same order as you executed the `START AFTER COMMIT` statements within the transaction.
- The body of a `START AFTER COMMIT` must contain only one SQL statement. That one statement may be a procedure call, however, and the procedure may contain multiple SQL statements, including other procedure calls.
- The `START AFTER COMMIT` statement applies only to the transaction in which it is defined. If you execute `START AFTER COMMIT` in the current transaction, the body of the deferred procedure call will be executed only when the current transaction is committed; it will not be executed in subsequent transactions, nor will it be executed for transactions done by any other connections. `START AFTER COMMIT` statements do not create "persistent" behavior. If you would like the same body to be called at the end of multiple transactions, then you will have to execute a "`START AFTER COMMIT ... CALL my_proc`" statement in each of those transactions.
- The "result" of the execution of the body of the deferred procedure call (`START AFTER COMMIT`) statement is not returned in any way to the connection that ran the deferred procedure call. For example, if the body of the deferred procedure call returns a value that indicates whether an error occurred, that value will be discarded.
- Almost any SQL statement may be used as the body of a `START AFTER COMMIT` statement. Although calls to stored procedures are typical, you may also use `UPDATE`, `CREATE TABLE`, or almost anything

else. (We don't advise putting another `START AFTER COMMIT` statements inside a `START AFTER COMMIT`, however.) Note that a statement like `SELECT` is generally useless inside an deferred procedure call because the result is not returned.

- Because the body is not executed at the time that the `START AFTER COMMIT` statement is executed inside the transaction, `START AFTER COMMIT` statements rarely fail unless the deferred procedure call itself or the body contains a syntax error or some other error that can be detected without actually executing the body.

What if you don't want the next SQL statement in your program to run until deferred procedure call statement has finished running? Here's a workaround:

1. At the end of the deferred procedure call statement (e.g. at the end of the stored procedure called by the deferred procedure call statement), post an Event. (See *IBM solidDB Programmer Guide* for a description of events.)
2. Immediately after you commit the transaction that specified the deferred procedure call, call a stored procedure that waits on the event.
3. After the stored procedure call (to wait on the event), put the next SQL statement that your program wants to execute.

For example, your program might look like the following:

```
...
START AFTER COMMIT ... CALL myproc;
...
COMMIT WORK;
CALL wait_for_sac_completion;
UPDATE ...;
```

The stored procedure `wait_for_sac_completion` would wait for the event that `myproc` will post. Therefore, the `UPDATE` statement won't run until after the deferred procedure call statement finishes.

Note that this workaround is slightly risky. Since deferred procedure call statements are not guaranteed to execute until completion, there is a chance that the stored procedure `wait_for_sac_completion` will never get the event that it is waiting for.

Why would anyone design a command that may or may not run to completion? The answer is that the primary purpose of the `START AFTER COMMIT` feature is to support "Sync Pull Notify". The Sync Pull Notify feature allows a master server to notify its replica(s) that data has been updated and that the replicas may request refreshes to get the new data. If this notification process fails for some reason, it would not result in data

corruption; it would simply mean that there would be a longer delay before the replica refreshes the data. Since a replica is always given all the data since its last successful refresh operation, a delay in receipt of data does not cause the replica to permanently miss any data. For more details, see the section of *IBM solidDB Advanced Replication Guide* that documents the Sync Pull Notify feature.



### Note

The statement inside the body of the START AFTER COMMIT may be any statement, including SELECT. Remember, however, that the body of the START AFTER COMMIT does not return its results anywhere, so a SELECT statement is generally not useful inside a START AFTER COMMIT.



### Note

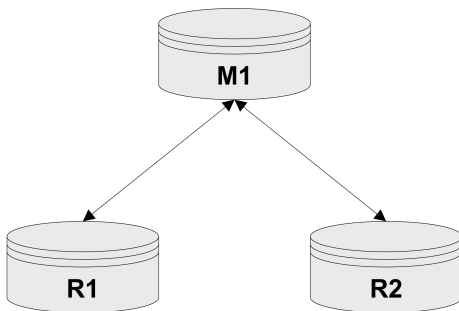
If you are in auto-commit mode and execute START AFTER COMMIT..., then the given statement is started immediately in the background. "Immediately" here actually means "as soon as possible", because it's still executed asynchronously when the server has time to do it.

## 4.9.1 Sync Pull Notify ("Push Synchronization") Example

To implement Sync Pull Notify (i.e. Master notifying all relevant Replicas that there is new data that they can request a refresh of), users can use the START and CALL statements as defined earlier. This particular example also uses triggers.

Let us consider a scenario where there is a Master M1 and Replicas R1 and R2.

**Figure 4.1. Sync Pull Notify**



To carry out Sync Pull Notify, follow the steps listed below:

1. Define a Procedure Pm1 in Master M1. In Procedure Pm1, include the statements:

```
EXECDIRECT CALL Pr1 AT R1;  
EXECDIRECT CALL Pr1 AT R2;
```

(You will have one call for each interested Replica. Note that the replica name changes, but typically the procedure name is the same on each replica.)

2. Define a Procedure Pr1 in Replica R1. If a master is to invoke the Pr1 in more than one replica, then Pr1 should be defined for every replica that is of interest. See the replica procedure example in the example section below.
3. Define a Trigger for all relevant DML operations, such as
  - INSERT
  - UPDATE and
  - DELETE
4. In each trigger body, embed the statement

```
EXECDIRECT START [UNIQUE] CALL Pm1;
```

5. Grant EXECUTE authority to the appropriate user on each replica. (A user Ur1 on the replica should already be mapped to a corresponding user Um1 on the master. The user Um1 must execute the

```
EXECDIRECT START [UNIQUE] CALL Pm1;
```

When Um1 calls the procedure remotely, the call will actually execute with the privileges of Ur1 when the call is executed on the replica.)

### Example 4.5. Sliced Replicas

A sales application has a table named CUSTOMER, which has a column named SALESMAN. The master database contains information for all salespersons. Each salesperson has her own replica database, and that replica has only a "slice" of the master's data; specifically, each salesperson's replica has the slice of data for that salesperson. For example, salesperson Smith's replica has only the data for salesperson Smith. If the salesperson assigned to a particular customer changes, then the correct replicas should be notified. If XYZ corporation is reassigned from salesperson Smith to salesperson Jones, then salesperson Jones's replica database

#### 4.9.1 Sync Pull Notify ("Push Synchronization") Example

---

should add the data related to XYZ corporation, and salesperson Smith's replica should delete the data related to XYZ corporation. Here is the code to update both replica databases:

```
-- If a customer is reassigned to a different salesman, then we
-- must notify both the old and new salesmen.
-- NOTE: This sample shows only the "UPDATE" trigger, but of course in
-- the real world you'd also need to define INSERT and DELETE triggers.
CREATE TRIGGER T_CUST_AFTERUPDATE ON CUSTOMER
AFTER UPDATE
REFERENCING NEW SALESMAN AS NEW_SALESMAN,
REFERENCING OLD SALESMAN AS OLD_SALESMAN
BEGIN
IF NEW_SALESMAN <> OLD_SALESMAN THEN
EXEC SQL EXECDIRECT
START AFTER COMMIT
FOR EACH REPLICa WHERE NAME=OLD_SALESMAN
UNIQUE CALL CUST(OLD_SALESMAN);
EXEC SQL EXECDIRECT
START AFTER COMMIT
FOR EACH REPLICa WHERE NAME=NEW_SALESMAN
UNIQUE CALL CUST(NEW_SALESMAN);
ENDIF
END;
```

Suppose that in the application, the user assigns all customers in sales area 'CA' to salesperson Mike.

```
UPDATE CUSTOMER SET SALESMAN='Mike' WHERE SALES_AREA='CA';
COMMIT WORK;
```

The Master server has the following procedure:

```
CREATE PROCEDURE CUST(salesman VARCHAR)
BEGIN
EXEC SQL EXECDIRECT CALL CUST(salesman) AT salesman;
COMMIT WORK;
END
```

Each replica has the following procedure:

```
CREATE PROCEDURE CUST(salesman VARCHAR)
BEGIN
  MESSAGE s BEGIN;
  MESSAGE s APPEND REFRESH CUSTS(salesman);
  MESSAGE s END;
  COMMIT WORK;
  MESSAGE s FORWARD TIMEOUT FOREVER;
  COMMIT WORK;
END
```

In the procedure `CUST()`, we force the salesperson's replica to refresh from the data in the master. This procedure `CUST()` is defined on all the replicas. If we call the procedure on both the replica that the customer was reassigned to, and the replica that the customer was reassigned from, then the procedure updates both those replicas. Effectively, this will delete the out-of-date data from the replica that no longer has this customer, and will insert the data to the replica that is now responsible for this customer. If the publication and its parameters are properly defined, we don't need to write additional detailed logic to handle each possible operation, such as reassigning a customer from one salesperson to another; instead, we simply tell each replica to refresh from the most current data.

### NOTES:

It is possible to implement a Sync Pull Notify without triggers. The application may call appropriate procedures to implement `SyncPull`. Triggers are a way to achieve Sync Pull Notify in conjunction with the statement `START AFTER COMMIT` and remote procedure calls.

Sometimes, in the Sync Pull Notify process, it is possible that a replica may have to exchange one extra round trip of messages unnecessarily. This could happen if the master invoked procedure tries to send a message to the replica that just sent the changes to the master, and that causes a change in the "hot data" in the master. But this can be avoided with careful usage of the `START AFTER COMMIT` statement. Be careful not to create an "infinite loop", where each update on the master leads to an immediate update on the replica, which leads to an immediate update on the master... The best way to avoid this is to be careful when creating triggers on the replica that might "immediately" send updated data to the master, which in turn "immediately" notifies the replica to refresh again.

## 4.9.2 Tracing the Execution of Background Jobs

The `START AFTER COMMIT` statement returns a result-set with one `INTEGER` column. This integer is a unique "job" id that can be used to query the status of statements that failed to start for one reason or another (invalid SQL statement, no access rights, replica not available, etc.).



If a maximum number of uncommitted deferred procedure call statements is reached, then an error is returned when the deferred procedure call is issued. The maximum number is configurable in `solid.ini`. See *IBM solidDB Administration Guide*.

If a statement cannot be started, the reason is logged to the system table `SYS_BACKGROUNDJOB_INFO`.

```
SYS_BACKGROUNDJOB_INFO
(
  ID INTEGER NOT NULL,
  STMT WVARCHAR NOT NULL,
  USER_ID INTEGER NOT NULL,
  ERROR_CODE INTEGER NOT NULL,
  ERROR_TEXT WVARCHAR NOT NULL,
  PRIMARY KEY(ID)
);
```

Only failed `START AFTER COMMIT` statements are logged into this table. If the statement (e.g. a procedure call) starts successfully, no information is stored into the system tables.

User can retrieve the information from the table `SYS_BACKGROUNDJOB_INFO` using either SQL `SELECT`-query or calling a system procedure `SYS_GETBACKGROUNDJOB_INFO`. The input parameters is the `jobID`. The returned values are: `ID INTEGER`, `STMT WVARCHAR`, `USER_ID INTEGER`, `ERROR_CODE INTEGER`, `ERROR_TEXT INTEGER`.

Also an event `SYS_EVENT_SACFAILED` is posted when a statement fails to start.

```
CREATE EVENT SYS_EVENT_SACFAILED (ENAME WVARCHAR,
POSTSRVTIME TIMESTAMP,
UID INTEGER,
NUMDATAINFO INTEGER,
TEXTDATA WVARCHAR);
```

The `NUMDATAINFO` field contains the `jobID`. The application can wait for this event and use the `jobID` to retrieve the reason from the system table `SYS_BACKGROUNDJOB_INFO`.

The system table `SYS_BACKGROUNDJOB_INFO` can be emptied with the admin command **`cleanbgjobinfo`**. You need `DBA` privileges to execute this command, which means that only a `DBA` can delete the rows from the table.

## 4.9.3 Controlling Background Tasks

Background tasks can be controlled with the SSC API and admin commands (see the linked library access manual for details on the SSC API). The server uses the task type `SSC_TASK_BACKGROUND` for the tasks that execute statements started with `START AFTER COMMIT`. Note that there may be several of these tasks, but you cannot control them individually.

## 4.10 Using Sequences

A sequence object is used to get sequence numbers in an efficient manner. The syntax is:

```
CREATE [DENSE] SEQUENCE sequence_name
```

Depending on how the sequence is created, there may or may not be holes in the sequence (the sequence can be sparse or dense). Dense sequences guarantee that there are no holes in the sequence numbers. The sequence number allocation is bound to the current transaction. If the transaction rolls back, the sequence number allocations are also rolled back. The drawback of dense sequences is that the sequence is locked out from other transactions until the current transaction ends.

If there is no need for dense sequences, a sparse sequence can be used. A sparse sequence guarantees uniqueness of the returned values, but it is not bound to the current transaction. If a transaction allocates a sparse sequence number and later rolls back, the sequence number is simply lost.

A sequence object can be used, for example, to generate primary key numbers. The advantage of using a sequence object instead of a separate table is that the sequence object is specifically fine-tuned for fast execution and requires less overhead than normal update statements.

Both dense and sparse sequence numbers start from 1.

After creating the sequence with the `CREATE SEQUENCE` statement, you can access the Sequence object values by using the following constructs in SQL statements:

- `sequencename.CURRVAL` which returns the current value of the sequence
- `sequencename.NEXTVAL` which increments the sequence by one and returns the next value.

An example of creating unique identifiers automatically for a table is given below:

```
INSERT INTO ORDERS (id, ...) VALUES (order_seq.NEXTVAL, ...);
```

Sequences can also be used inside stored procedures. The current sequence value can be retrieved using the following statement:

```
EXEC SEQUENCE sequence_name.CURRENT INTO variable;
```

New sequence values can be retrieved using the following syntax:

```
EXEC SEQUENCE sequence_name.NEXT INTO variable;
```

It is also possible to set the current value of a sequence to a predefined value by using the following syntax:

```
EXEC SEQUENCE sequence_name SET VALUE USING variable;
```

An example of using a stored procedure to retrieve a new sequence number is given below:

```
"CREATE PROCEDURE get_my_seq  
RETURNS (val INTEGER)  
BEGIN  
EXEC SEQUENCE my_sequence.NEXT INTO (val);  
END";
```

## 4.11 Using Events

Event alerts are special objects in IBM solidDB databases. Events are used primarily to coordinate timing, but may also be used to send a small amount of information. One connection "waits" on an event until another connection "posts" that event.

More than one connection may wait on the same event. If multiple connections wait on the same event, then all waiting connections are notified when the event is posted. A connection may also wait on multiple events, in which case it will be notified when any of those events are posted.

Events generally consume a much smaller amount of resources than polling consumes.

Users may create their own events. The server also has some built-in system events.

The server does not automatically post user-defined events; they must be posted by a stored procedure. Similarly, the events are received (waited on) in a stored procedure. (You may also wait on an event outside a stored procedure by using the `ADMIN EVENT` command.) When an application calls a stored procedure that waits for a specific event to happen, the application is blocked until the event is posted and received. In multi-threaded environments, separate threads and connections can be used to access the database during the event wait.

An event has a name that identifies it and a set of parameters. The name can be any user-specified alphanumeric string. An event object is created with the SQL statement:

```
CREATE EVENT event_name
  [(parameter_name datatype
    [parameter_name datatype...])]
```

The parameter list specifies parameter names and parameter types. The parameter types are normal SQL types. Events are dropped with the SQL statement:

```
DROP EVENT event_name
```

Events are always posted inside stored procedures. Events are usually received inside stored procedures. Special stored procedure statements are used to post and receive events.

The event is posted with the stored procedure statement

```
post_statement ::= POST EVENT event_name [(parameters)]
```

Event parameters must be local variables or parameters in the stored procedure where the event is triggered. All clients that are waiting for the posted event will receive the event.

Each connection has its own event queue. The events to be collected in the event queue are specified with the stored procedure statement

```
wait_register-statement ::=
REGISTER EVENT event_name
```

Events are removed from the event queue with the stored procedure statement

```
UNREGISTER EVENT event_name
```

Event parameters must be local variables or parameters in the stored procedure where the event is triggered.

To make a procedure wait for an event to happen, the `WAIT EVENT` construct is used in the stored procedure:

```
wait_event_statement::=  
  WAIT EVENT  
    [event_specification...]  
  END WAIT  
event_specification::=  
  WHEN event_name [(parameters)] BEGIN  
    statements  
  END EVENT
```

You may also wait on an event by using the `ADMIN EVENT` command. You may use this at the `solsql` command line, for example. Below is an example of the code required to register for and wait on an event using `ADMIN EVENT` commands:

```
ADMIN EVENT 'register sys_event_hsbstateswitch';  
ADMIN EVENT 'wait';
```

You may wait on either system-defined events or user-defined events. Note that you cannot post events using `ADMIN EVENT`. For more details about `ADMIN EVENT`, see Section B.2, “ADMIN EVENT”.

### Example 4.6. Event Example 1

This section includes two examples for using events. Example 1 is a pair of SQL scripts that when used together show how to use events. Example 2 is a pair of SQL scripts, including a stored procedure, that when used together waits for multiple events.

In this first example of using events, we have two scripts. One script waits on an event and the other script posts the event. Once the event has been posted, the event that is waiting will finish waiting and move on to the next command.

To execute this example code, you will need two consoles so that you can start the `WaitOnEvent.sql` script and then run the `PostEvent.sql` script while `WaitOnEvent.sql` is waiting.

In this particular example, the stored procedure that waits does not actually do anything after the event has posted; the script merely finishes the wait and returns to the caller. The caller can then proceed to do whatever it wants, which in this case is to `SELECT` the record that was inserted while we were waiting.

This example waits for only a single event, which is called "record\_was\_inserted". Later in this chapter we will have another script that waits for multiple events using a single "WAIT".

```
===== SCRIPT 1=====
-- SCRIPT NAME: WaitOnEvent.sql
-- PURPOSE:
-- This is one of a set of scripts that demonstrates posting events
-- and waiting on events. The sequence of steps is shown below:
--
-- THIS SCRIPT (WaitOnEvent.sql) PostEvent.sql script
-----
-- CREATE EVENT.
-- CREATE TABLE.
-- WAIT ON EVENT.
--   Insert a record into table.
--   Post event.
-- SELECT * FROM TABLE.
--
-- To perform these steps in the proper order, start running this
-- script FIRST, but remember that this script does not finish running
-- until after the post_event script runs and posts the event.
-- Therefore, you will need two open consoles so that you can leave
-- this running/waiting in one window while you run the other script
-- (post_event) in the other window.
-- Create a simple event that has no parameters.
-- Note that this event (like any event) does not have any
-- commands or data; the event is just a label that allows both the
-- posting process and the waiting process to identify which event has
-- been posted (more than one event may be registered at a time).
-- As part of our demonstration of events, this particular event
-- will be posted by the other user after he or she inserted a record.
CREATE EVENT record_was_inserted;
-- Create a table that the other script will insert into.
CREATE TABLE table1 (int_col INTEGER);
-- Create a procedure that will wait on an event
-- named "record_was_inserted".
-- The other script (PostEvent.sql) will post this event.
```

```
"CREATE PROCEDURE wait_for_event
BEGIN
-- If possible, avoid holding open a transaction. Note that in most
-- cases it's better to do the COMMIT WORK before the procedure,
-- not inside it. See "Waiting on Events" at the end of this example.
EXEC SQL COMMIT WORK;
-- Now wait for the event to be posted.
WAIT EVENT
  WHEN record_was_inserted BEGIN
  -- In this demo, we simply fall through and return from the
  -- procedure call, and then we continue on to the next
  -- statement after the procedure call.
  END EVENT
END WAIT;
END";
-- Call the procedure to wait. Note that this script will not
-- continue on to the next step (the SELECT) until after the
-- event is posted.
CALL wait_for_event();
COMMIT WORK;
-- Display the record inserted by the other script.
SELECT * FROM table1;
```

### *Guidelines for Committing Transaction in Script 1 (WaitOnEvent.sql)*

Whenever possible, complete any current transaction before waiting on an event. If you execute a WAIT inside a transaction, then the transaction will be held open until the event occurs and the next COMMIT or ROLLBACK is executed. This means that during the wait, the server will hold locks, which may lead to excessive bonsai tree growth. For details on the Bonsai Tree and preventing its growth, read the section "Reducing Bonsai Tree Size by Committing Transactions," in *IBM solidDB Administration Guide*.

In this example, we have put COMMIT WORK inside the procedure immediately before the WAIT. However, this is not usually a good solution; putting the COMMIT or ROLLBACK inside the "wait" procedure means that if the procedure is called as part of another transaction, then the COMMIT or ROLLBACK will terminate that enclosing transaction and start a new transaction, which is probably not what you want. If, for example, you were entering data into a "child" table with a referential constraint and you are waiting for the referenced data to be entered into the "parent" table, then breaking the transaction into two transactions would simply cause the insert of the "child" record to fail because the parent would not have been inserted yet.

The best strategy is to design your program so that you do not need to WAIT inside a transaction; instead, your "wait" procedure should be called between transactions if that is possible. By using events/waits, you have some control over the order in which things are done and you can use this to help ensure that dependencies

are met without actually putting everything into a single transaction. For example, in an "asynchronous" situation you might be waiting for both a child and a parent record to be inserted, and if your database server did not have the "events" feature, then you might require that both records be inserted in the same transaction so that you could ensure referential integrity.

By using events/waits, you can ensure that the insertion of the parent is done first; you can then put the insertion of the child record in a second transaction because you can guarantee that the parent will always be present when the child is inserted. (To be more precise, you can ALMOST guarantee that the parent will be present when the child is inserted. If you break up the insertions into two different transactions, then even if you ensure that the parent is inserted before the child, there is a slight chance that the parent would be deleted before the program tried to insert the child record.)

```
===== SCRIPT 2=====
-- SCRIPT NAME: PostEvent.sql
-- PURPOSE:
-- This script is one of a set of scripts that demonstrates posting
-- events and waiting on events. The sequence of steps is shown below:
--
-- WaitOnEvent.sql THIS SCRIPT (PostEvent.sql)
-- -----
-- Create event.
-- Create table.
-- Wait on event.
--   INSERT A RECORD INTO TABLE.
--   POST THE EVENT.
-- Select * from table.
-- Insert a record into the table.
INSERT INTO table1 (int_col) VALUES (99);
COMMIT WORK;
-- Create a stored procedure to post the event.
"CREATE PROCEDURE post_event
BEGIN
  -- Post the event.
  POST EVENT record_was_inserted;
END";
-- Call the procedure that posts the event.
CALL post_event();
```



```
DROP PROCEDURE post_event;  
COMMIT WORK;
```

### Example 4.7. Event Example 2

The previous example showed how to wait on a single event. The next example shows how to write a stored procedure that will wait on multiple events and that will finish the wait when any one of those events is posted.

```
===== SCRIPT 1=====
-- SCRIPT NAME: MultiWaitExamplePart1.sql
-- PURPOSE:
-- This code shows how to wait on more than one event.
-- If you run this demonstration, you will see that a "wait" lasts only
-- until one of the events is received. Thus a wait on multiple events
-- is like an "OR" (rather than an "AND"); you wait until event1 OR
-- event2 OR ... occurs.
--
-- This demo uses 2 scripts, one of which waits for an event(s) and one
-- of which posts an event.
-- To run this example, you will need 2 consoles.
-- 1) Run this script (MultiWaitExamplePart1.sql) in one window. After
-- this script reaches the point where it is waiting for the event, then
-- start Step 2.
-- 2) Run the script MultiWaitExamplePart2.sql in the other window.
-- This will post one of the events.
-- After the event is posted, the first script will finish.
-- Create the 3 different events on which we will wait.
CREATE EVENT event1;
CREATE EVENT event2(i INTEGER);
CREATE EVENT event3(i INTEGER, c CHAR(4));
-- When an event is received, the process that is waiting on the event
-- will insert a record into this table. That lets us see which events
-- were received.
CREATE TABLE event_records(event_name CHAR(10));
-- This procedure inserts a record into the event_records table.
-- This procedure is called when an event is received.
"CREATE PROCEDURE insert_a_record(event_name_param CHAR(10))
BEGIN
  EXEC SQL PREPARE insert_cursor
  INSERT INTO event_records (event_name) VALUES (?);
  EXEC SQL EXECUTE insert_cursor USING (event_name_param);
```

```
EXEC SQL CLOSE insert_cursor;
EXEC SQL DROP insert_cursor;
END";
-- This procedure has a single "WAIT" command that has 3 subsections;
-- each subsection waits on a different event.
-- The "WAIT" is finished when ANY of the events occur, and so the
-- event_records table will hold only one of the following:
-- "event1",
-- "event2", or
-- "event3".
"CREATE PROCEDURE event_wait(i1 INTEGER)
RETURNS (eventresult CHAR(10))
BEGIN
  DECLARE i INTEGER;
  DECLARE c CHAR(4);
  -- The specific values of i and c are irrelevant in this example.
  i := i1;
  c := 'mark';
  -- Set eventresult to an empty string.
  eventresult := '';
  -- Will we exit after any of these 3 events are posted, or must
  -- we wait until all of them are posted? The answer is that
  -- we will exit after any one event is posted and received.
  WAIT EVENT
    -- When the event named "event1" is received...
  WHEN event1 BEGIN
    eventresult := 'event1';
    -- Insert a record into the event_records table showing that
    -- this event was posted and received.
    EXEC SQL PREPARE call_cursor
      CALL insert_a_record(?);
    EXEC SQL EXECUTE call_cursor USING (eventresult);
    EXEC SQL CLOSE call_cursor;
    EXEC SQL DROP call_cursor;
    RETURN;
  END EVENT
  WHEN event2(i) BEGIN
    eventresult := 'event2';
    EXEC SQL PREPARE call_cursor2
      CALL insert_a_record(?);
    EXEC SQL EXECUTE call_cursor2 USING (eventresult);
    EXEC SQL CLOSE call_cursor2;
```

```
EXEC SQL DROP call_cursor2;
RETURN;
END EVENT
WHEN event3(i, c) BEGIN
  eventresult := 'event3';
  EXEC SQL PREPARE call_cursor3
    CALL insert_a_record(?);
  EXEC SQL EXECUTE call_cursor3 USING (eventresult);
  EXEC SQL CLOSE call_cursor3;
  EXEC SQL DROP call_cursor3;
  RETURN;
END EVENT
END WAIT
END";
COMMIT WORK;
-- Call the procedure that waits until one of the events is posted.
CALL event_wait(1);
-- See which event was posted.
SELECT * FROM event_records;
===== SCRIPT 2 =====
-- SCRIPT NAME: MultiWaitExamplePart2.sql
-- PURPOSE:
-- This is script 2 of 2 scripts that show how to wait for multiple
-- events. See the instructions at the top of MultiWaitExamplePart1.sql.
-- Create a stored procedure to post an event.
"CREATE PROCEDURE post_event1
BEGIN
  -- Post the event.
  POST EVENT event1;
END";
--Create a stored procedure to post the event.
"CREATE PROCEDURE post_event2(param INTEGER)
BEGIN
  -- Post the event.
  POST EVENT event2(param);
END";
--Create a stored procedure to post the event.
"CREATE PROCEDURE post_event3(param INTEGER, s CHAR(4))
BEGIN
  -- Post the event.
  POST EVENT event3(param, s);
END";
```

```
COMMIT WORK;
-- Notice that to finish the "wait", only one event needs to be posted.
-- You may execute any one of the following 3 CALL commands to post an
-- event.
-- We've commented out 2 of them; you may change which one is de
-- commented.
CALL post_event1();
--CALL post_event2(2);
--CALL post_event3(3, 'mark');
```

### Example 4.8. Event Example 3

This example shows very simple usage of the REGISTER EVENT and UNREGISTER EVENT commands. You might notice that the previous scripts did not use REGISTER EVENT, yet their WAIT commands succeeded anyway. The reason for this is that when you wait on an event, you will be registered implicitly for that event if you did not already explicitly register for it. Thus you only need to explicitly register events if you want them to start being queued now but you don't want to start WAITing for them until later.

```
CREATE EVENT e0;
CREATE EVENT e1 (param1 int);
COMMIT WORK;

-- Create a procedure to register the events to that when they occur
-- they are put in this connection's event queue.
"CREATE PROCEDURE eeregister
  BEGIN
    REGISTER event e0;
    REGISTER EVENT e1;
  END";

CALL eeregister;
COMMIT WORK;

-- Create a procedure to post the events.
"CREATE PROCEDURE eepost
  BEGIN
    DECLARE x int;
    x := 1;
    POST EVENT e0;
    POST EVENT e1(x);
  END";
```

```
END";

COMMIT WORK;

-- Post the events.  Even though we haven't yet waited on the events,
-- they will be stored in our queue because we registered for them.
CALL eepost;
COMMIT WORK;

-- Now create a procedure to wait for the events.
"CREATE PROCEDURE eewait
  RETURNS (whichEvent VARCHAR(100))
  BEGIN
    DECLARE i INT;

    WAIT EVENT
      WHEN e0 BEGIN
        whichEvent := 'event0';
      END EVENT

      WHEN e1(i) BEGIN
        whichEvent := 'event1';
      END EVENT

    END WAIT

  END";

COMMIT WORK;

-- Since we already registered for the 2 events and we already
-- posted the 2 events, when we call the eewait procedure twice
-- it should return immediately, rather than waiting.
CALL eewait;
CALL eewait;
COMMIT WORK;

-- Unregister for the events.
"CREATE PROCEDURE eeunregister
  BEGIN
    UNREGISTER event e0;
```

```
UNREGISTER EVENT e1;
END";

CALL eeunregister;
COMMIT WORK;
CREATE EVENT e0;
CREATE EVENT e1 (param1 int);
COMMIT WORK;

-- Create a procedure to register the events to that when they occur
-- they are put in this connection's event queue.
"CREATE PROCEDURE eeregister
  BEGIN
    REGISTER event e0;
    REGISTER EVENT e1;
  END";

CALL eeregister;
COMMIT WORK;

-- Create a procedure to post the events.
"CREATE PROCEDURE eepost
  BEGIN
    DECLARE x int;
    x := 1;
    POST EVENT e0;
    POST EVENT e1(x);
  END";

COMMIT WORK;

-- Post the events. Even though we haven't yet waited on the events,
-- they will be stored in our queue because we registered for them.
CALL eepost;
COMMIT WORK;

-- Now create a procedure to wait for the events.
"CREATE PROCEDURE eewait
  RETURNS (whichEvent VARCHAR(100))
  BEGIN
    DECLARE i INT;
```

```
WAIT EVENT
  WHEN e0 BEGIN
    whichEvent := 'event0';
  END EVENT

  WHEN e1(i) BEGIN
    whichEvent := 'event1';
  END EVENT

END WAIT

END";

COMMIT WORK;

-- Since we already registered for the 2 events and we already
-- posted the 2 events, when we call the eewait procedure twice
-- it should return immediately, rather than waiting.
CALL eewait;
CALL eewait;
COMMIT WORK;

-- Unregister for the events.
"CREATE PROCEDURE eeunregister
  BEGIN
    UNREGISTER event e0;
    UNREGISTER EVENT e1;
  END";

CALL eeunregister;
COMMIT WORK;
```





---

# Chapter 5. Using IBM solidDB SQL for Database Administration

You manage a IBM solidDB database, as well as its users and schema, using IBM solidDB SQL statements. This chapter describes the management tasks you perform with IBM solidDB SQL. These tasks include managing roles and privileges, tables, indexes, transactions, catalogs, and schemas.

## 5.1 Using IBM solidDB SQL Syntax

The SQL syntax is based on the ANSI X3H2-1989 (SQL-89) level 2 standard including important SQL-92 and SQL-99 extensions. Refer to Appendix B, *IBM solidDB SQL Syntax*, for a more formal definition of the syntax.

SQL statements must be terminated with a semicolon (;) only when using IBM solidDB SQL Editor. Otherwise, terminating SQL statements with a semicolon leads to a syntax error.

You can use IBM solidDB SQL Editor (or third-party ODBC or JDBC compliant tools) to execute SQL statements. To automate the tasks, you may want to save the SQL statements to a file. You can use these files for rerunning your SQL statements later or as a document of your users, tables, and indexes.

### 5.1.1 IBM solidDB SQL Data Types

IBM solidDB SQL supports data types specified in the SQL-92 Standard Entry Level specifications, as well as important Intermediate Level enhancements. Refer to Appendix A, *Data Types*, for a complete description of the supported data types.

You can also define some data types with the optional length, scale, and precision parameters. In that case, the default properties of the corresponding data type are not used.

### 5.1.2 IBM solidDB ADMIN COMMAND

IBM solidDB SQL provides the extension **ADMIN COMMAND** '*command* [*command\_args*]' to perform basic administrative tasks, such as backups, performance monitoring, and shutdown.

You can use IBM solidDB SQL Editor (teletype) to execute the command options provided by **ADMIN COMMAND**. To access a short description of available **ADMIN COMMANDS**, execute **ADMIN COMMAND 'help'**. For a formal definition of the syntax of these statements, refer to Appendix B, *IBM solidDB SQL Syntax*, in this guide.



### Note

ADMIN COMMAND tasks are also available as administrative commands in IBM solidDB Remote Control (teletype). For details, read the section of *IBM solidDB Administration Guide* titled "IBM solidDB Remote Control (teletype)".

IBM solidDB also provides SQL extensions that implement the data synchronization capability.

## 5.1.3 Using Functions

All IBM solidDB proprietary scalar functions can be used in a normal way, e.g.:

```
select substring(line, 1,4) from test;
```

On the other hand, functions whose name match reserved words, have to be used with escape characters. For example:

```
select "left"(line,4) from test;
```

or:

```
select {fn left(line,4)} from test;
```

The latter one corresponds to the ODBC implementation-independent syntax. It can be used in all API and GUI interfaces.

## 5.2 Managing User Privileges and Roles

You can use IBM solidDB teletype tools, and many ODBC compliant SQL tools to modify user privileges. Users and roles are created and deleted using SQL statements or commands. A file consisting of several SQL statements is called a SQL script.

In the `Solid/solidDB6.0/samples/sql` directory, you will find the SQL script `sample.sql`, which gives an example of creating users and roles. You can run it by using `sosql`. To create your own users and roles, you can make your own script describing your user environment.

## 5.2.1 User Privileges

When using IBM solidDB databases in a multi-user environment, you may want to apply user privileges to hide certain tables from some users. For example, you may not want an employee to see the table in which employee salaries are listed, or you may not want other users to change your test tables.

You can apply five different kinds of user privileges. A user may be able to view, delete, insert, update or reference information in a table or view. Any combination of these privileges may also be applied. A user who has none of these privileges to a table is not able to use the table at all.



### Note

Once user privileges are granted, they take effect when the user who is granted the privileges logs on to the database. If the user is already logged on to the database when the privileges are granted, they take effect only if the user:

- accesses for the first time the table or object on which the privileges are set, or
- disconnects and then reconnects to the database.

## 5.2.2 User Roles

Privileges can also be granted to an entity called a role. A role is a group of privileges that can be granted to users as one unit. You can create roles and assign users to certain roles. A single user may have more than one role assigned, and a single role may have more than one user assigned.



### Note

1. The same string cannot be used both as a user name and a role name.
2. Once a user role is granted, it takes effect when the user who is granted the role logs on to the database. If the user is already logged on to the database when the role is granted, the role takes effect when the user disconnects and then reconnects to the database.

The following user names and roles are reserved:

**Table 5.1. Reserved User Names and Roles**

Reserved Names	Description
PUBLIC	This role grants privileges to all users. When user privileges to a certain table are granted to the role <b>PUBLIC</b> , all current and future users have the specified user privileges to this table. This role is granted automatically to all users.
SYS_ADMIN_ROLE	This is the default role for the database administrator. This role has administration privileges to all tables, indexes and users, as well as the right to use IBM solidDB Remote Control. This is also the database creator role.
_SYSTEM	This is the schema name of all system tables and views.
SYS_CONSOLE_ROLE	This role has the right to use IBM solidDB Remote Control, but does not have other administration privileges.
SYS_SYNC_ADMIN_ROLE	This is the administrator role for data synchronization functions.
SYS_SYNC_REGISTER_ROLE	This role is only for registering and unregistering a replica database to the master.

## 5.2.3 Examples of SQL Statements

Below are some examples of SQL statements for administering users, roles, and user privileges.

### Creating Users

```
CREATE USER username IDENTIFIED BY password;
```

Only an administrator has the privilege to execute this statement. The following example creates a new user named CALVIN with the password HOBBS.

```
CREATE USER CALVIN IDENTIFIED BY HOBBS;
```

### Deleting Users

```
DROP USER username;
```

Only an administrator has the privilege to execute this statement. The following example deletes the user named CALVIN.

```
DROP USER CALVIN;
```

### Changing a Password

```
ALTER USER username IDENTIFIED BY new password;
```

The user *username* and the administrator have the privilege to execute this command. The following example changes CALVIN's password to GUBBES.

```
ALTER USER CALVIN IDENTIFIED BY GUBBES;
```

### Creating Roles

```
CREATE ROLE rolename;
```

The following example creates a new user role named GUEST\_USERS.

```
CREATE ROLE GUEST_USERS;
```

### Deleting Roles

```
DROP ROLE role_name;
```

The following example deletes the user role named GUEST\_USERS.

```
DROP ROLE GUEST_USERS;
```

### Granting Privileges to a User or a Role

```
GRANT user_privilege ON table_name TO username or role_name ;
```

The possible user privileges on tables are SELECT, INSERT, DELETE, UPDATE, REFERENCES and ALL. ALL provides a user or a role all five privileges mentioned above. A new user has no privileges until they are granted.

The following example grants INSERT and DELETE privileges on a table named TEST\_TABLE to the GUEST\_USERS role.

```
GRANT INSERT, DELETE ON TEST_TABLE TO GUEST_USERS;
```

The EXECUTE privilege provides a user the right to execute a stored procedure:

```
GRANT EXECUTE ON procedure_name TO username or role_name ;
```

The following example grants EXECUTE privilege on a stored procedure named SP\_TEST to user CALVIN.

```
GRANT EXECUTE ON SP_TEST TO CALVIN;
```

### **Granting Privileges to a User by Giving the User a Role**

```
GRANT role_name TO username ;
```

The following example gives the user CALVIN the privileges that are defined for the GUEST\_USERS role.

```
GRANT GUEST_USERS TO CALVIN;
```

### **Revoking Privileges from a User or a Role**

```
REVOKE user_privilege ON table_name FROM username or role_name ;
```

The following example revokes the INSERT privilege on the table named TEST\_TABLE from the GUEST\_USERS role.

```
REVOKE INSERT ON TEST_TABLE FROM GUEST_USERS;
```

### **Revoking Privileges by Revoking the Role of a User**

```
REVOKE role_name FROM username ;
```

The following example revokes the privileges that are defined for the GUEST\_USERS role from CALVIN.

```
REVOKE GUEST_USERS FROM CALVIN;
```

### Granting Administrator Privileges to a User

```
GRANT SYS_ADMIN_ROLE TO username ;
```

The following example grants administrator privileges to CALVIN, who now has all privileges to all tables.

```
GRANT SYS_ADMIN_ROLE TO CALVIN;
```

You may also want to grant a user the right to perform data synchronization operations. To do this, execute the command:

```
GRANT SYS_SYNC_ADMIN_ROLE TO HOBBS
```



#### Note

If the autocommit mode is set OFF, you need to commit your work. To commit your work use the following SQL statement: COMMIT WORK; If the autocommit mode is set ON, the transactions are committed automatically.

## 5.3 Managing Tables

IBM solidDB has a dynamic data dictionary that allows you to create, delete and alter tables on-line. IBM solidDB database tables are managed using SQL commands.

In the IBM solidDB directory, you can find a SQL script named `sample.sql`, which gives an example of managing tables. You can run the script using `solsql`.

Below are some examples of SQL statements for managing tables. Refer to Appendix B, *IBM solidDB SQL Syntax* for a formal definition of the IBM solidDB SQL statements.

If you want to see the names of all tables in your database, issue the SQL statement `SELECT * FROM TABLES`. ("TABLES" is a system-defined view.) The table names can be found in the column `TABLE_NAME`.

## 5.3.1 Accessing System Tables

The IBM solidDB system tables store IBM solidDB server information, including user information. Your ability to access specific system tables depends on your user's role and access rights. For example, DBAs can view all information about all stored procedures, including the procedure definition text (i.e. the CREATE PROCEDURE statement). Normal users can see the stored procedures, including the procedure definition text, for procedures that they have created. Normal users who have execute access on a stored procedure, but who did not create that stored procedure, may look at some information about that stored procedure but may not see the procedure definition text. For a list of system tables, refer to Appendix D, *Database System Tables and System Views*.

The table below provides the viewing access and/or object granting privileges for specific system tables and their data by user role and user access rights.

Note that a "User with access rights" in this table refers to a normal user who has any one of the following rights: INSERT, UPDATE, DELETE, or SELECT access.\*

**Table 5.2. Viewing Tables and Granting Access**

Tasks	DBA	Owner	User with access rights*	User with no access rights
Viewing SYS_TABLES	All (no restrictions)	All (no restrictions)	All (no restrictions)	All (no restrictions)
Viewing User tables in SYS_TABLES	All (no restrictions)	Restricted to the owners' tables only	All tables to which the user has INSERT, UPDATE, DELETE, SELECT, or REFERENCES access rights.	No tables can be viewed.
Viewing SYS_COLUMNS	All (no restrictions)	Columns in the owner's tables	Columns in tables to which the user has INSERT, UPDATE, DELETE, SELECT, or REFERENCES access rights.	No columns can be viewed.
Viewing SYS PROCEDURES (excluding the procedure definition text — i.e. the text of the CREATE PROCEDURE statement)	All (no restrictions)	Those procedures created by the user (owner).	Those procedures in which the user has execute access.	No procedures can be viewed.



Tasks	DBA	Owner	User with access rights*	User with no access rights
Viewing Procedure definition text in SYS_PROCEDURES	All (no restrictions)	Those procedures created by the user (owner)	Note that execute access does not allow the user to see the procedure definition text.	No procedures or procedure definition text can be viewed.
Ability to Grant Access rights on procedures	Yes	Yes	No	No
Viewing SYS_TRIGGERS	All (no restrictions)	Those triggers created by the user (owner)	None	No triggers can be viewed.
Viewing Trigger definition text in SYS_TRIGGERS	All (no restrictions)	Those triggers created by the user (owner)	None	No triggers can be viewed.

## 5.3.2 Examples of SQL Statements

Below are some examples of SQL statements for administering tables.

### Creating Tables

```
CREATE TABLE table_name (column_name column_type
    [, column_name column_type]...);
```

All users have privileges to create tables.

The following example creates a new table named TEST with the column I of the column type INTEGER and the column TEXT of the column type VARCHAR.

```
CREATE TABLE TEST (I INTEGER, TEXT VARCHAR);
```

### Removing Tables

```
DROP TABLE table_name;
```

Only the creator of the particular table or users having SYS\_ADMIN\_ROLE have privileges to remove tables.

The following example removes the table named TEST.

```
DROP TABLE TEST;
```



### Note

For catalogs and schemas: The ANSI standard for SQL defines the keywords `RESTRICT` and `CASCADE`. When dropping a catalog or a schema, if you use the keyword `RESTRICT`, then you cannot drop a catalog or schema if it contains other database objects (e.g. tables). Using the keyword `CASCADE` allows you to drop a catalog or schema that still contains database objects — the database objects that it contains will automatically be dropped. The default behavior (if you don't specify either `RESTRICT` or `CASCADE`) is `RESTRICT`.

For database objects other than Catalogs and Schemas: The keywords `RESTRICT` and `CASCADE` are not accepted as part of most `DROP` statements in IBM solidDB SQL. Furthermore, for these database objects, the rules are more complex than simply "pure `CASCADE`" or "pure `RESTRICT`" behavior, but generally objects are dropped with drop behavior `RESTRICT`. For example, if you try to drop `table1` but `table2` has a foreign key dependency on `table1`, or if there are publications that reference `table1`, then you will not be able to drop `table1` without first dropping the dependent table or publication. However, the server does not use `RESTRICT` behavior for all possible types of dependency. For example, if a view or a stored procedure references a table, the referenced table can still be dropped, and the view or stored procedure will fail the next time that it tries to reference that table. Also, if a table has a corresponding synchronization history table, that synchronization history table will be dropped automatically. For more information about synchronization history tables, see *IBM solidDB Advanced Replication Guide*.

## Adding Columns to a Table

```
ALTER TABLE table_name ADD COLUMN column_name column_type;
```

Only the creator of the particular table or users having `SYS_ADMIN_ROLE` have privileges to add or delete columns in a table.

The following example adds the column `C` of the column type `CHAR(1)` to the table `TEST`.

```
ALTER TABLE TEST ADD COLUMN C CHAR(1);
```

### Deleting Columns from a Table

```
ALTER TABLE table_name DROP COLUMN column_name;
```

A column cannot be dropped if it is part of a unique constraint or primary key. For details on primary keys, read Section 5.4, “Managing Indexes”.

The following example statement deletes the column C from the table TEST.

```
ALTER TABLE TEST DROP COLUMN C;
```



#### Note

If the autocommit mode is set OFF, you need to commit your work before you can modify the data in the table you altered. To commit your work after altering a table, use the following SQL statement:

```
COMMIT WORK;
```

If the autocommit mode is set ON, then all statements, including DDL (Data Definition Language) statements, are committed automatically.

## 5.4 Managing Indexes

Indexes are used to speed up access to tables. The database engine uses indexes to access the rows in a table directly. Without indexes, the engine would have to search the whole contents of a table to find the desired row. You can create as many indexes as you like on a single table; however, adding indexes does slow down write operations, such as inserts, deletes, and updates on that table. For details on creating indexes to improve performance, read Section 8.3, “Using Indexes to Improve Query Performance”.

There are two kinds of indexes: non-unique indexes and unique indexes. A unique index is an index where all key values are unique. A unique index is always created, when the UNIQUE restraint is used when creating an index.

You can create and delete indexes using the following SQL statements. Refer to Appendix B, *IBM solidDB SQL Syntax*, for a formal definition of the syntax for these statement.

## 5.4.1 Examples of SQL Statements

Below are some examples of SQL commands for administering indexes.

### Creating an Index on a Table

```
CREATE [UNIQUE] INDEX index_name ON base_table_name  
  column_identifier [ASC | DESC]  
  [, column_identifier [ASC | DESC]] ...
```

Only the creator of the particular table or users having SYS\_ADMIN\_ROLE have privileges to create or drop indexes.

The following example creates an index named X\_TEST on column I of the table TEST.

```
CREATE INDEX X_TEST ON TEST (I);
```

### Creating a Unique Index on a Table

```
CREATE UNIQUE INDEX index_name ON table_name (column_name);
```

The following example creates a unique index named UX\_TEST on column I of the table TEST.

```
CREATE UNIQUE INDEX UX_TEST ON TEST (I);
```

### Deleting an Index

```
DROP INDEX index_name;
```

The following example deletes the index named X\_TEST.

```
DROP INDEX X_TEST;
```



#### Note

After creating or dropping an index, you must commit (or roll back) your work before you can modify the data in the table on which you created or dropped the index.

## 5.4.2 Primary Key Indexes

To retrieve a single specific record from a table, we must be able to uniquely identify that record. IBM solidDB uses "primary keys" to uniquely identify each record in each table. A primary key is a column or combination of columns that contains a unique value or combination of values. Each table has a primary key — either explicit or implicit.

IBM solidDB automatically creates a "primary key index" based on the field(s) of that primary key. A primary key index, like any index, speeds up access to data in the table. Unlike other indexes, however, a primary key index also controls the order in which records are stored in the database. (This is called "clustering".) Records are stored in ascending order based on the primary key values.

If the creator of the table does not specify a primary key, then IBM solidDB automatically creates a primary key for the table. To ensure uniqueness in that primary key, the server uses a hidden internal row identifier. The value of that row identifier may be retrieved and used in queries by way of a symbolic pseudo column name "ROWID".



### Note

In IBM solidDB, it is not possible to add an explicit primary key after the table has been created. If a primary key is not specified by a user, the most efficient query method is not available (unless ROWID is used) for that table. Also, such a table cannot be used in referential integrity constraints as a referenced table. For those reasons, it is strongly recommended that a primary key is always defined at table creation.

Once a primary key is defined (whether by the table creator or by the server), the server will prevent rows with duplicate primary key values from being inserted into the table.

## 5.4.3 Secondary Key Indexes

Since indexes speed up searches, it is often helpful for a table to have one index for each attribute (or combination of attributes) that is used frequently in searches. All indexes other than the primary index are called "secondary indexes".

A table may have as many indexes as you like, as long as each index has a unique combination of columns, order of columns, and order of values (ASCending, DESCending). For example, in the code shown below, the third index duplicates the first and will either generate an error message or will waste disk space with duplicate information.

```
CREATE INDEX i1 ON TABLE t1 (col1, col2);  
-- The following is ok because although the columns are the same as in
```

```
-- index i1, the order of the columns is different.
CREATE INDEX i2 ON TABLE t1 (col2, col1);
-- The following is not ok because index i3 would be exactly the
-- same as index i1.
CREATE INDEX i3 ON TABLE t1 (col1, col2); -- ERROR.
-- The following is ok because although the columns and
-- column order are the same, the order of the index values
-- (ASCending vs. DESCending) is different.
CREATE INDEX i3b ON TABLE t1 (col1, col2) DESC;
```

Note that if one index is a "leading subset" of another (meaning that the columns, column order, and value order of all N columns in index2 are exactly the same as the first N column(s) of index1), then you only need to create the index that is the superset. For example, suppose that you have an index on the combination of DEPARTMENT + OFFICE + EMP\_NAME. This index can be used not only for searches by department, office and emp\_name together, but also for searches of just the department, or just the department and office together. So there is no need to create a separate index on the department name alone, or on the department and office alone. The same is true for ORDER BY operations; if the ORDER BY criterion matches a subset of an existing index, then the server can use that index.

Keep in mind that if you defined a primary key or unique constraint, that key or constraint is implemented as an index. Thus you never need to create an index that is a "leading subset" of the primary key or of an existing unique constraint; such an index would be redundant.

Note that when searching using a secondary index, if the server finds all the requested data in the index key, the server doesn't need to look up the complete row in the table. (This applies only to "read" operations, i.e. SELECT statements. If the user updates values in the table, then of course the data rows in the table as well as the values in the index(es) must be updated.)

## 5.4.4 Protection Against Duplicate Indexes

IBM solidDB contains a protection against duplicate indexes. Occasionally, the recreation of an index (DROP/CREATE) can fail if other indexes were created whereby the original index became a duplicate index. To understand what duplicate indexes are, see the example below:

Let's assume we have created a table containing five columns, named A, B, C, D, E. The following indexes have been created on the table:

- A
- AB
- BCE

- ABC

As you can see, index B is used for searching or filtering column B. Index BCE starts with column B. Therefore, queries that use an index for locating column B can use index BCE. The same is the case with indexes AB and ABC. Thus, indexes B and AB are duplicate indexes.

Duplicate indexes have, for example, the following adverse effects:

- The storage space required increases
- The update performance decreases
- Backup time increases

If you attempt to create duplicate indexes, index creation fails and IBM solidDB issues error:

SOLID Table Error 13199: Duplicate index definition

For more information, see Appendix C, Error Codes, in *IBM solidDB Administration Guide*.

## 5.5 Referential Integrity

Referential integrity is a concept for ensuring that relationships between database tables remain consistent. In other words, references to data must be valid.

A relationship between two database tables, called a referenced table and a referencing table, is created by using a foreign key. A foreign key is a field in the referencing table that matches the primary key column (or other similar unique column) of the referenced table. In other words, the foreign key can be used to represent a conceptual relationship of type 1:n such as "an employee belongs to a department". Now, when the referencing table has a foreign key to the referenced table, the concept of referential integrity states that you cannot add a record to the referencing table (containing the foreign key) unless there is a corresponding record in the referenced table.

As explained above, referential integrity is enforced by using the foreign keys. Foreign keys are maintained with referential constraint definitions. The constraints also specify what referential actions IBM solidDB must take when the constraint is violated. This can happen, for example, when a row with a referenced primary key is deleted from the referenced table. Foreign keys and constraints are explained into more detail in the following chapters.

## 5.5.1 Primary Keys and Candidate Keys

In order for a table to participate in referential constraints as a referenced table, a primary key (preferable) or candidate keys have to be defined. A primary key is defined with the primary key constraint syntax in the CREATE TABLE statement, e.g.:

```
CREATE TABLE customers (  
  cust_id INTEGER PRIMARY KEY,  
  name CHAR(24),  
  city CHAR(40));
```

Another possibility is to define a unique index on a column or a group of columns and enact the NOT NULL constraint for them. Effectively, this will produce a "candidate key". Using an explicit primary key is preferable because of the involved performance gain while deriving joins.

## 5.5.2 Foreign Keys

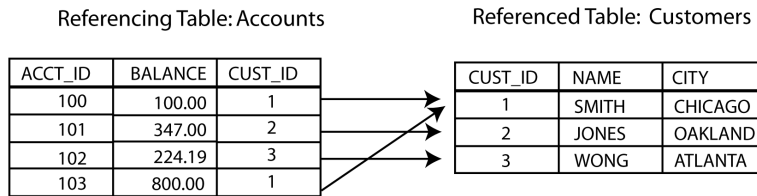
A foreign key is a column (or group of columns) within a table that refers to (or "relates to") a unique value in another table. Each value in the foreign key column must have a matching value in the other table.

To ensure that each record in the referencing table references exactly one record in the referenced table, the referenced column(s) in the referenced table must have a primary key constraint or having both unique and not-null constraints. (Note that having a unique index is not sufficient.)

For example, in a bank, one table might hold customer information, and another table might hold account information. Each account must be related to a customer, and would have a unique customer\_id. This customer\_id would serve as the primary key of the customers table. Each account would also have a copy of the customer\_id of the customer who owns that account; this allows us to look up customer information based on account information. The copy of the customer\_id in the accounts table is a foreign key; it references the matching value in the primary key of the customers table.

Below is an example. In this example, the CUST\_ID column in the CUSTOMERS table is the primary key of the referenced table, and the CUST\_ID column of the ACCOUNTS table is a foreign key that refers to the CUSTOMERS table. As you can see in the diagram below, each account is associated with a corresponding customer. Some customers have more than one account.



**Figure 5.1. Referential Constraint**

The referencing table "accounts" may be created with the following statement:

```
CREATE TABLE accounts (
    acc_id INTEGER PRIMARY KEY,
    balance FLOAT,
    customer_id INTEGER REFERENCES customers);
```

In the REFERENCES clause, only the referenced table is specified, with no referenced column. By default, the primary key is assumed. This is a preferred way allowing to avoid errors possibly made while specifying the referenced columns.

In the example shown above, the primary key and foreign key used a single column. However, primary and foreign keys may be composed of more than one column. Since each foreign key value must exactly match the corresponding primary key value, a foreign key must contain the same number and data type of columns as the primary key, and these key columns must be in the same order. However, a foreign key can have different column names than the primary key, although this is rare. (The foreign key and primary key may also have different default values. However, since values in the referenced table must be unique, default values are not much used and are rarely used for columns that are part of a primary key. Default values are also not used very often for foreign key columns.)

Although primary key values must be unique, foreign key values are not required to be unique. For example, a single customer at a bank might have multiple accounts. The account\_id that appears in the primary key column in the CUSTOMERS table must be unique; however, the same account\_id might occur multiple times in the foreign key column in the ACCOUNTS table. As you can see in the illustration above, customer SMITH has more than one account, and therefore her CUST\_ID appears more than once in the foreign key column of the ACCOUNTS table.

Although it is rare, a foreign key in a table may refer to a primary key in that same table. In other words, the referenced table and the referencing table are the same table. For example, in a table of employees, each employee record might have a field that contains the ID of the manager of that employee. The managers themselves might be stored in the same table. Thus the manager\_id of that table might be a foreign key that refers to the employee\_id of that same table. You can see an example of this below.

**Figure 5.2. Self-referential Constraint**

A SELF-REFERENTIAL TABLE

EMP_ID	MGR_ID	EMP_NAME
1	NULL	ANNAN
10	1	WONG
20	1	SMITH
147	10	JONES
162	20	RAMA

In this example, Rama's manager is Smith (Rama's MGR\_ID is 20, and Smith's EMP\_ID is 20). Smith reports to Annan (Smith's MGR\_ID is 1, and Annan's EMP\_ID is 1.) Jones' manager is Wong, and Wong's manager is Annan. If Annan is the president of the company, then Annan doesn't have a manager, and the value of the foreign key is NULL.

If a primary key is composed of more than one column, it should be defined after the columns. For example:

```
CREATE TABLE DEPT (  
    DIVNO INTEGER,  
    DEPTNO INTEGER,  
    DNAME VARCHAR,  
    PRIMARY KEY (DIVNO, DEPTNO));
```

A similar syntax may be used for foreign keys. However, it is highly recommended that foreign keys are defined with the CONSTRAINT syntax utilizing also a constraint name. This approach allows for removing foreign key dynamically, with the ALTER TABLE statements, after tables have been created. For example:

```
CREATE TABLE EMP (  
    EMPNO INTEGER PRIMARY KEY,  
    DIVNO INTEGER,  
    DEPTNO INTEGER,
```

```
ENAME VARCHAR,  
CONSTRAINT emp_fk1 FOREIGN KEY (DIVNO, DEPTNO) REFERENCES DEPT);
```



## Note

Similarly to other integrity constraints, you can name referential integrity constraints (foreign keys) and manipulate them (drop or add) dynamically, with the ALTER TABLE statement. For more information, see Section 5.5.4, “Dynamic Constraint Management”.

Refer to Appendix B, *IBM solidDB SQL Syntax*, for CREATE TABLE and ALTER TABLE syntax detail.

Not all tables are allowed to have foreign keys. If a table is involved in master/replica synchronization and is in a replica server, that table can not have any foreign key constraints. This limitation applies only to the tables that are in replicas and that are involved in publish/subscribe (refresh) activities. Note that tables in the replica that are not involved in refresh activities may still have foreign keys. Foreign keys are allowed in the tables that are in the master database, even if those tables are involved in publish/refresh activities.

This limitation does not apply to primary keys. Any table may have a primary key (and some tables, such as synchronization tables, must have a primary key).

Defining a foreign key always creates an index on the foreign key column(s). Each time when a referenced record is updated or deleted, the server checks that there are no referencing records that are left without a reference. Giving each foreign key an index improves the performance of foreign key checking.

## 5.5.3 Referential Actions

Referential integrity is maintained by the system, by taking certain actions when referential constraints are violated, for example, in one of the following ways:

- when a row containing an invalid foreign key value is inserted in the referencing table
- when a foreign key in the referencing table is updated to an invalid value
- when a row with a referenced primary key is deleted from the referenced table
- when a referenced primary key is updated in the referenced table

The following actions are available when the constraint is violated:

- *No action*. This option restricts the operation, or rolls back the operation that violates the referential integrity constraint.

- *Cascade*. In the case of operations performed on the referenced table, cascades the operations on the referenced table down to the referencing tables. This includes deleting all the referencing rows (a cascading delete) and updating all the referencing foreign-key values (a cascading update).
- *Set default*. In the case of operations performed on the referenced table, sets the referencing columns to a predefined default value.
- *Set null*. In the case of operations performed on the referenced table, sets the referencing columns to null.
- *Restrict*. Referential integrity actions sometimes allow changes to a table that temporarily violate a referential constraint. The No action allows such violations. If you have a requirement that the table state must never violate any constraint even temporarily, use the Restrict referential action.

If no action is specified, the default 'No action' is assumed.

No cycles are allowed in cascading referential actions. An effort to create a cycle composed of foreign keys having cascading actions results in an error.



### Note

For any two tables, at most one CASCADE UPDATE path between them can be defined. The restriction does not apply to CASCADE DELETE.

## 5.5.4 Dynamic Constraint Management

Constraints can be managed dynamically with the ALTER TABLE clause. The sub-clauses that can be used are:

- **ADD CONSTRAINT**. This clause adds a named constraint to a table.
- **DROP CONSTRAINT**. This clause removes a named constraint from a table.



### Note

In IBM solidDB, when the keyword CONSTRAINT is used, the constraint name is mandatory.

- **CHECK**. This constraint allows you to specify rules to your tables or table columns. Each rule is a condition that must not be false for any row in the table on which it is defined. Otherwise the table cannot be updated.

The rules are Boolean expressions. The rule can check, for example, a range of values, equity, or the rule can be a simple comparison. You can use several checks in one statement. The following expressions and operators are available:

**Table 5.3. Expressions and Operators**

Expression	Explanation
<	less than
>	greater than
=	equal to
<=	less than or equal to
>=	greater than or equal to
<>	not equal to
AND	negation
ANY	in the list that follows or in the table specified
BETWEEN	between
IN	in the list that follows or in the table specified
MAX	maximum value
MIN	minimum value
NOT	conjunction
OR	disjunction
XOR	exclusive or

- **UNIQUE.** The UNIQUE constraint requires that no two rows in a table contain the same value in a given column or list of columns. You can create a unique constraint at either the table level or the column level. Note: primary keys contain the unique constraint.
- **FOREIGN KEY.** The FOREIGN KEY constraint requires that each value in the foreign key column must have a matching value in the referenced table.



### Note

IBM solidDB automatically generates names for unnamed constraints. If you want to view the names, use the command **soldd -x hidddenames**.

For constraint syntax information and examples, see the CREATE TABLE and ALTER TABLE sections in Appendix B, *IBM solidDB SQL Syntax*.

## 5.6 Managing Database Objects

### 5.6.1 Introduction

IBM solidDB allows you to use catalogs and schemas to organize your data. (Catalogs also have other uses, which we will explain later.) IBM solidDB's use of schemas conforms to the SQL standard, while IBM solidDB's use of catalogs is an extension to the SQL standard.

Catalogs and schemas allow you to group database objects (e.g. tables, sequences, etc.) in a hierarchical way. This allows you to put related items into the same group. For example, all the tables related to the accounting system might be in one group (e.g. a catalog), while all the tables related to the human resources system might be in another group. You can also group database objects by user. For example, all of the tables used by Jane Smith might be in a single schema.

Catalogs are the highest (broadest) level of the hierarchy. Schema names are the mid-level. Specific database objects, such as tables, are the lowest (narrowest) level of the hierarchy. Thus, a single catalog may contain multiple schemas, and each of those schemas may contain multiple tables.

Object names must be unique within a group, but do not have to be unique across groups. Thus, for example, Jane Smith's schema and Robin Trower's schema might each have a table named "bills". These two tables have nothing to do with each other. They may have different structures and different data, even though they have the same name. Similarly, the catalog "accounting\_catalog" and the catalog "human\_resources\_catalog" might each have a schema named "david\_jones". Those schemas are unrelated to each other, even though they have the same name.

Not surprisingly, if you want to specify a particular table and that table name is not unique in the database, you can identify it by specifying the catalog, schema, and table name, e.g.

```
accounting_catalog.david_jones.bills
```

The syntax is discussed in more detail later.

If you don't specify the complete name (i.e. if you omit the schema, or the schema and the catalog), then the server uses the current/default catalog and schema name to determine which table to use.

In general, a catalog can be thought of as a logical database. A schema typically corresponds to a user. This is discussed in more detail below.

## 5.6.2 Catalogs

A physical database file may contain more than one logical database. Each logical database is a complete, independent group of database objects, such as tables, indexes, procedures, triggers, etc. Each logical database is a catalog. Note that a IBM solidDB catalog is not just limited to indexes (as in the traditional sense of a library card catalog, which serves to locate an item without containing the full contents of the item).

Catalogs allow you to logically partition databases so you can:

- Organize your data to meet the needs of your business, users, and, applications.
- Specify multiple master or replica databases (by using logical databases) for synchronization within one physical database server. For more details on implementing synchronization in multi-master environments, read "Multi-master synchronization model" in *IBM solidDB Advanced Replication Guide*.

## 5.6.3 Schemas

A catalog may contain one or more schemas. A schema is a persistent database object that provides a definition for part or all of the database. It represents a collection of database objects associated with a specific schema name. These objects include tables, views, indexes, stored procedures, triggers, and sequences. Schemas allow you to provide each user with his or her own database objects (such as tables) within the same logical database (that is, a single catalog). If no schema is specified with a database object, the default schema is the user id of the user creating the object.

## 5.6.4 Uniquely Identifying Objects within Catalogs and Schemas

Schemas make it possible for two different users to create tables with the same names in the same physical database or even in the same logical database. For example, assume in a single physical database, there are two separate catalogs, `employee_catalog` and `inventory_catalog`. Assume also that each catalog contains two separate schemas, named `smith` and `jones`, and that the same Smith owns both "smith" schemas and the same Jones owns both "jones" schemas. If Smith and Jones create a table named `books` in each of their schemas, then we have a total of 4 tables named "books", and these tables are accessible as:

```
employee_catalog.smith.books  
employee_catalog.jones.books  
inventory_catalog.smith.books  
inventory_catalog.jones.books
```

As you can see, the catalog name and schema name can be used to "qualify" (uniquely identify) the name of a database object such as a table. Object names can be qualified in all DML statements by using the syntax:

*catalog\_name.schema\_name.database\_object*

or

*catalog\_name.user\_id.database\_object*

For example:

```
SELECT cust_name FROM accounting_dept.smith.overdue_bills;
```

You can qualify one or more database objects with a schema name, whether or not you specify a catalog name. The syntax is:

*schema\_name.database\_object\_name*

or

*user\_id.database\_object\_name*

For example,

```
SELECT SUM(sales_tax) FROM jones.invoices;
```

To use a schema name with a database object, you must have already created the schema.

By default, database objects that are created without schema names are qualified using the user ID of the database object's creator. For example:

*user\_id.table\_name*

Catalog and schema contexts are set using the SET CATALOG or SET SCHEMA statement.

If a catalog context is not set using SET CATALOG, then all database object names are resolved by using the default catalog name.



 **Note**

When creating a new database or converting an old database to a new format, the user is prompted to specify a default catalog name for the database system catalog. Users can access the default catalog name without knowing this specified default catalog name. For example, users can specify the following syntax to access the system catalog:

```
" "._SYSTEM.table
```

IBM solidDB translates the empty string ("") specified as a catalog name to the default catalog name. IBM solidDB also provides for automatic resolution of `_SYSTEM` schema to the system catalog, even when users provide no catalog name.

The following SQL statements provide examples of creating catalogs and schemas. Refer to Appendix B, *IBM solidDB SQL Syntax*, for a formal definition of the IBM solidDB SQL statements.

## 5.6.5 Examples of SQL Statements

Below are some examples of SQL statements for managing database objects.

### Creating a Catalog

```
CREATE CATALOG catalog_name
```

Only the creator of the database or users having `SYS_ADMIN_ROLE` have privileges to create or drop catalogs.

The following example creates a catalog named C and assumes the userid is SMITH

```
CREATE CATALOG C;  
SET CATALOG C;  
CREATE TABLE T (i INTEGER);  
SELECT * FROM T;  
--The name T is resolved to C.SMITH.T
```

### Setting a Catalog and Schema Context

The following example sets a catalog context to C and the schema context to S.

```
SET CATALOG C;  
SET SCHEMA S;  
CREATE TABLE T (i INTEGER);  
SELECT * FROM T;  
-- The name T is resolved to C.S.T
```

### Deleting a Catalog

```
DROP CATALOG catalog_name
```

The following example deletes the catalog named C.

```
DROP CATALOG C;
```

### Creating a Schema

```
CREATE SCHEMA schema_name
```

Any database user can create a schema; however, the user must have permission to create the objects that pertain to the schema (for example, CREATE PROCEDURE, CREATE TABLE, etc.).

Note that creating a schema does not implicitly make that new schema the current/default schema. You must explicitly set that schema with the SET SCHEMA statement if you want the new schema to become the current schema.

The following example creates a schema named FINANCE and assumes the user id is SMITH:

```
CREATE SCHEMA FINANCE;  
CREATE TABLE EMPLOYEE (EMP_ID INTEGER);  
-- NOTE: The employee table is qualified to SMITH.EMPLOYEE, not  
-- FINANCE.EMPLOYEE. Creating a schema does not implicitly make that  
-- new schema the current/default schema.  
SET SCHEMA FINANCE;  
CREATE TABLE EMPLOYEE (ID INTEGER);
```

```
SELECT ID FROM EMPLOYEE;  
-- In this case, the table is qualified to FINANCE.EMPLOYEE
```

### **Deleting a Schema**

```
DROP SCHEMA schema_name
```

The following example deletes the schema named FINANCE.

```
DROP SCHEMA FINANCE;
```

---

---

# Chapter 6. Managing Transactions

This chapter concentrates on transaction management. To be more specific, this chapter explains how to manage transactions, how to deal with concurrency control and locking, and how to choose the durability level.

## 6.1 Managing Transactions

A transaction is a group of SQL statements treated as a single unit of work; either all the statements are executed as a group, or none are executed. This section assumes you know the fundamentals for creating transactions using standard SQL statements. It describes how IBM solidDB SQL lets you handle transaction behavior, concurrency control, and isolation levels.

### 6.1.1 Defining Read-Only and Read-Write Transactions

To define a transaction to be read-only or read-write, use the following SQL commands:

```
SET TRANSACTION { READ ONLY | READ WRITE }
```

The following options are available with this command.

- READ ONLY

Use this option for a read only transaction.

- READ WRITE

Use this option for a read and write transaction. This option is the default.



#### Note

To detect conflicts between transactions, use the standard ANSI SQL command `SET TRANSACTION ISOLATION LEVEL` to define the transaction with a Repeatable Read or Serializable isolation level. For details, read For details, read chapter Choosing Transaction Isolation Levels in *IBM solidDB Administration Guide*.

Transactions must be ended with the `COMMIT WORK` or `ROLLBACK WORK` commands unless `autocommit` is used.

## 6.1.2 Setting Concurrency Control

Earlier, we discussed the theory behind concurrency control ("locking"). This section explains some of the commands used in specifying what type of concurrency control you would like to use.

### Setting Pessimistic and Mixed Concurrency Control

By default, IBM solidDB uses optimistic concurrency control. When necessary, you can also use pessimistic (row-level locking) or mixed concurrency control methods. Pessimistic concurrency control is more appropriate, for example, in applications that contain small tables that are frequently updated. In the case of these so-called *hotspots*, conflicts are so probable that optimistic concurrency control wastes effort in rolling back conflicting transactions.

Mixed concurrency control is available by setting individual tables to optimistic or pessimistic. Mixed concurrency control is a combination of row-level pessimistic locking and optimistic concurrency control. By turning on row-level locking table-by-table, you can specify that a single transaction use both concurrency control methods simultaneously. This functionality is available for both read-only and read-write transactions.



#### Note

Pessimistic table level locks in shared mode are possible with tables that are synchronized. This functionality provides users with the option to run some operations for synchronization in pessimistic mode even with optimistic tables. For example, when a REFRESH is executed in pessimistic mode in a replica, IBM solidDB locks all tables in shared mode; later, if necessary, the server can "promote" these locks to exclusive table locks. This is done in a few synchronization statements when optional keyword PESSIMISTIC is specified. Note that read operations do not use any locks.

To set individual tables for optimistic or pessimistic concurrency, use the following SQL command:

```
ALTER TABLE base_table_name SET {OPTIMISTIC | PESSIMISTIC}
```

Note that by default all tables are set for optimistic.

You can also set a database-wide default in the *[General]* section of the configuration file with the following parameter:

```
Pessimistic = yes
```

When you specify PESSIMISTIC concurrency control, the server places locks on rows to control the level of consistency and concurrency when users are submitting queries or updates to rows.

## Setting Lock Timeout

The lock timeout setting is the time in seconds that the engine waits for a lock to be released. By default, lock timeout is set to 30 seconds. When the timeout interval is reached, IBM solidDB terminates the timed-out statement. For example, if one user is querying a specific row in a table and a second user tries to update data in the same row, the update will not go through until the first user's query is completed (or times out). If the first user's query is completed and the second user's query hasn't timed out yet, then a lock is issued for the second user's update transaction. If the first user doesn't finish before the second user times out, then the second user's statement is terminated by the server.

You can set the lock timeout with the following SQL command:

```
SET LOCK TIMEOUT timeout_in_seconds
```

By default, the granularity is in seconds. The lock timeout can be set at millisecond granularity by adding "MS" after the value, e.g.

```
SET LOCK TIMEOUT 10MS;
```

Without the "MS", the lock timeout will be in seconds.

Note that the maximum timeout is 1000 seconds (a little over 15 minutes). The server will not accept a longer value.

## Setting Lock Timeout for Optimistic Tables

When you use `SELECT FOR UPDATE`, the selected rows are locked even if the table's locking mode was set to "optimistic". These rows must be locked to ensure that the update will be successful. By default, the lock timeout in this situation is 0 seconds — in other words, either you immediately get the lock, or you get an error message. If you would like the server to wait and try again to get the lock before giving up, then you can use the following SQL command to set the lock timeout separately for optimistic tables.

```
SET OPTIMISTIC LOCK TIMEOUT seconds
```

# 6.2 Concurrency Control and Locking

In a system where multiple users may try to update the same data simultaneously, the system must limit concurrent access; in other words, the system must allow only one user at a time to change the data.

Although concurrency control is a basic capability of any good database system, the topic can become surprisingly complex and there are some subtleties. This section of the manual attempts to explain how concurrency

control and locking behave from a user perspective. (We do not attempt to describe most of the internal mechanisms by which the server actually implements concurrency control.) This section explains:

1. the purpose of concurrency control,
2. exclusive vs. shared locks,
3. pessimistic vs. optimistic concurrency control,
4. table locks,
5. lock duration,
6. transaction isolation levels,
7. miscellaneous info

See also Chapter 6, *Managing Transactions* for more information about locks and transactions.

### 6.2.1 The Purpose of Concurrency Control

The purpose of concurrency control is to prevent two different users (or two different connections by the same user) from trying to update the same data at the same. Concurrency control can also prevent one user from seeing out-of-date data while another user is part way through updating the same data. Below is a simple example of why concurrency control is needed.

Suppose that your checking account contains \$1,000. Suppose also that today you deposit \$300 and you spend \$200 from that account. Obviously, at the end of the day your account should have \$1,100. However, if the account updates are done "simultaneously" rather than in sequence, one update might write over another update.

Suppose that at 11:00 AM bank teller #1 looks up your account and sees that you have \$1,000. She subtracts the \$200 check, but is not able to save the updated account balance (\$800) immediately. At 11:01 AM, bank teller #2 looks up your account and still sees the \$1,000 balance. She adds your \$300 deposit and saves your new account balance as \$1,300. At 11:09 AM, bank teller #1 returns to her terminal, finishes entering and saving the updated value that she calculated (\$800). That \$800 value writes over the \$1300. At the end of the day, your account has \$800 when it should have had \$1,100 ( $\$1000 + 300 - 200$ ).

To prevent two users from "simultaneously" updating data (and potentially writing over each other's updates), database software uses a concurrency control mechanism. IBM solidDB offers two different concurrency control mechanisms. These are called "pessimistic concurrency control" (usually just called "locking") and "optimistic concurrency control". (We will explain the reasons for these terms later.

For simplicity, in this example we will assume the system uses locking as its concurrency control mechanism.



A lock is a mechanism for limiting other users' access to a piece of data. When one user has a lock on a record, the lock prevents other users from changing (and in some cases reading) that record.

When teller #1 starts working on your account, a "lock" is placed on the account; if teller #2 tries to read or update your account while teller #1 is updating your account, teller #2 will not be given access and will typically get an error message. In most database servers, the lock is placed on an individual record in the database. (We will discuss table-level locks later.) Using our banking example, a teller might get a lock on the record that contains your checking account balance without also locking your savings account balance and without locking the records of any other users' accounts.

Locking allows us to increase SAFETY at the cost of CONCURRENCY. We assure data integrity, but we do it by preventing more than one user at a time from working with a particular piece of data.

### 6.2.2 EXCLUSIVE vs. SHARED LOCKS

An exclusive lock allows only one user/connection to access (read or update) a particular piece of data. A shared lock allows multiple users to read data, but doesn't allow any of them to update the data.

If a user is updating data (as in our bank example) and is using pessimistic concurrency control (i.e. locking), then that user must acquire an "exclusive" lock. No other user may read or update that data (e.g. bank account record) while the exclusive lock is held. In addition, if you are using pessimistic concurrency control, no other user may even view the record that has been exclusively locked. That prevents a user from seeing, for example, a mix of updated data and not-yet-updated data. At any given time, only one user may have an exclusive lock on a particular piece of data.

If both users only want to read (not change) the data, then each user can use a "shared" lock. For example, if I am reading, but not updating, a record, then another user can look at that record at the same time. Many users may have shared locks on the same item (record, table, etc.) at the same time. For example, you, your spouse, your banker, and a credit rating agency could all look at your checking account balance simultaneously, as long as none of you try to change it at the same time.

Shared and exclusive locks cannot be mixed. If you have an exclusive lock on a record, I cannot get a shared lock (or an exclusive lock) on that same record.

### 6.2.3 PESSIMISTIC vs. OPTIMISTIC Concurrency Control

As we mentioned previously, IBM solidDB offers two different types of concurrency control mechanisms, which are known as "pessimistic" and "optimistic". We explain both methods below. By default, IBM solidDB uses "optimistic" concurrency control.

Pessimistic concurrency control is also known as "locking". Locks allow multiple users to safely share a database as long as all users are updating different data at the same time. For example, you can update Ms.

Smith's record while I update Mr. Kumar's record. (For the moment, I will simplify the discussion by focusing only on update operations and exclusive locks, not read-only/select operations and shared locks.)

When locks are used, the locks are placed as soon as any piece of the row is updated. Thus it is impossible for two users to update a row at the same time. As soon as one user gets a lock, no one else can process that row. This is a safe, conceptually simple approach. The disadvantage is that it requires overhead for every operation, whether or not two or more users are actually trying to access the same record. This overhead is small, but adds up because every row that is updated requires a lock. Furthermore, every time that a user tries to access a row, the system must also check whether the requested row(s) are already locked by another user (or connection).

To extend our earlier example with bank tellers, when teller #1 gets a lock, teller #2 must check for that lock, no matter how unlikely it is that teller #2 will want to work on the same record at the same time as teller #1. Checking every record that you use will take some time. Furthermore, it's important that during that checking no other teller tries to run the same check as teller #2 (otherwise they might both see at 10:59:59 that record X is not in use, and then they might both try to lock it at 11:00:00). Thus even checking a lock can itself require another lock to prevent two users from changing the locks at the time.

Pessimistic concurrency control (i.e. locking) is called "pessimistic" because the system assumes the worst — it assumes that two users will want to update the same record at the same time, and then prevents that possibility by locking the record, no matter how unlikely conflicts actually are.

An alternative approach to locking is called "optimistic concurrency control". Optimistic concurrency control assumes that although conflicts are possible, they will be very rare. Instead of locking every record every time that it is used, the software merely looks for indications that two users actually did try to update the same record at the same time. If that evidence is found, then one user's updates are discarded (and of course the user is informed).

Below is a description of one way that the server can detect conflicts after they occur (rather than preventing them before they occur). For simplicity, we'll assume that an update is done as the following series of actions:

1. Read the data from a disk drive into memory.
2. Update the data in memory.
3. Write the updated data back to the disk drive.

(The principle is the same even if the updated data is written to a device other than a disk drive.)

When using optimistic locking, each time that the server reads a record to try to update it, the server makes a copy of the "version number" of the record and stores that copy for later reference. When it's time to write the updated data back to the disk drive, the server compares the original version number that it read with the version number that the disk drive now contains. If the version numbers are the same, then no one else changed

the record and we can write our updated value. However, if the value we originally read and the current value on the disk are not the same, then someone has changed the data since we read it, and whatever operation we did is probably out-of-date, so we discard our version of the data and give the user an error message. Naturally, each time that we update a record, we also update the version number.

When you use optimistic locking, you don't find out there's a conflict until just before you write the updated data. In pessimistic locking, you find out there's a conflict as soon as you try to read the data. To use our analogy with banks again, pessimistic locking is like having a guard at the bank door who checks your account number when you try to enter; if someone else (a spouse, or a merchant to whom you wrote a check) is already in the bank accessing your account, then you can't enter until that other person finishes her transaction and leaves. Optimistic locking, on the other hand, allows you to walk into the bank at any time and try to do your business, but at the risk that as you are walking out the door the bank guard will tell you that your transaction conflicted with someone else's and you'll have to go back and do the transaction again.

Optimistic and pessimistic concurrency control differ in another important way besides the time at which conflicts are detected and error messages are issued. Pessimistic locking allows one user to not only block another user from updating the same record, but even from reading that record. If you use pessimistic locking and you get an exclusive lock, then no other user can even read that record. With optimistic locking, however, we don't check for conflicts except at the time that we write updated data to disk. If user1 updates a record and user2 only wants to read it, then user2 simply reads whatever data is on the disk and then proceeds, without checking whether the data is locked. User2 might see slightly out-of-date information if user1 has read the data and updated it but has not yet "committed" the transaction.

IBM solidDB actually implements optimistic concurrency control in a more sophisticated way than this. Rather than giving each user "whatever version of data is on the disk at the moment it is read", IBM solidDB can store multiple versions of each data row temporarily. Each user's transaction sees the database as it was at the time that the transaction started. This way, the data that each user sees is consistent throughout the transaction, and users are able to concurrently access the database. Data is always available to users because locking is not used; access is improved since deadlocks no longer apply. (Again, however, users run the risk that their changes will be thrown out if those changes conflict with another user's changes.) For details about how multiversioning is done, read the section of *IBM solidDB Administration Guide* titled IBM solidDB Bonsai Tree Multiversioning and Concurrency Control.

The descriptions above of optimistic and pessimistic concurrency control are slightly simplified. Even if a table uses pessimistic locking, and even if a record within that table has an exclusive lock, another user may execute read operations on the locked record under specific conditions. If the reader explicitly sets her transaction to be a read-only transaction, then she can use versioning rather than locking. This only occurs if the user explicitly declares the transaction as read only by issuing the command:

```
SET TRANSACTION READ ONLY;
```

Thus, for example, user1 might put an exclusive lock on a record and update it. When the record is updated, its version number changes. User2, who is using a read-only transaction, can read the previous version of the record even though the record has an exclusive lock on it.

Note that pessimistic locking allows you an option that optimistic locking does not offer. We said earlier that pessimistic locks fail "immediately" — i.e., if you try to get an exclusive lock on a record and another user already has a lock (shared or exclusive) on that record, then you will be told that you can't get a lock. In fact, IBM solidDB allows you the option of either failing immediately or of waiting a specified number of seconds before failing. You might specify a wait of 30 seconds; this means that if you initially try to get the lock and cannot, the server will continue trying to get the lock until either it gets the lock or until the 30 seconds has elapsed. In many cases, especially when transactions tend to be very short, you may find that setting a brief wait allows you to continue activities that otherwise would have been blocked by locks.

This wait mechanism applies only to pessimistic locking, not to optimistic concurrency control. There is no such thing as "waiting for an optimistic lock". If someone else changed the data since the time that you read it, no amount of waiting will prevent a conflict that has already occurred. In fact, since optimistic concurrency methods do not place locks, there is literally no "optimistic lock" to wait on.



#### Note

When executing `SELECT FOR UPDATE`, the server uses an update mode lock, which prevents other users from reading or updating that row and ensures that the current user can update the row. For more information, see the section called "Shared, Exclusive, and Update Locks", Section 6.1.2, "Setting Concurrency Control", and the section called "Setting Lock Timeout for Optimistic Tables".

Neither pessimistic nor optimistic concurrency control is "right" or "wrong". When properly implemented, both approaches ensure that your data is properly updated. In most scenarios, optimistic concurrency control is more efficient and offers higher performance, but in some scenarios pessimistic locking is more appropriate. In situations where there are a lot of updates and relatively high chances of users trying to update data at the same time, you probably want to use pessimistic locking. If the odds of conflict are very low (many records and relatively few users, or very few updates and mostly "read" operations), then optimistic concurrency control is usually the best choice. The decision will also be affected by how many records each user updates at a time. In our bank example, we usually update only one account/record at a time. For some applications, however, each operation may update a large number of records at a time (for example, the bank might add interest earnings to every account at the end of each month), virtually assuring that if two such applications are running at the same time then they will have conflicts.

You can override optimistic locking and specify pessimistic locking instead. You can do this at the level of individual tables. One table might follow the rules of optimistic locking while another table follows the rules of pessimistic locking. Both tables can be used within the same transaction and even the same statement; IBM solidDB takes care of the details for you. For more details about how to specify optimistic vs. pessimistic, see the section called "Setting the Concurrency (Locking) Mode to Optimistic or Pessimistic".

You might wonder whether "optimistic locking" is a true locking scheme at all. When we use optimistic locking, we do not actually place any locks. Thus the name "optimistic locking" is misleading. However, optimistic locking serves the same purpose as pessimistic locking (it prevents overlapping updates), so it is labeled "locking", even though the underlying mechanism is not a true lock.

### **Important**

By default, IBM solidDB servers use optimistic locking for disk-based tables. Optimistic locking allows fast performance and high concurrency (access by multiple users), at the cost of occasionally "refusing" to write data that was initially accepted but was found at the last second to conflict with another user's changes.

On the other hand, in in-memory tables, only pessimistic concurrency control is available because it leads to better memory preservation

### **Setting the Concurrency (Locking) Mode to Optimistic or Pessimistic**

For in-memory tables that are used in transactions with an isolation level higher than READ COMMITTED, the server will use pessimistic concurrency control.

For all other tables, the server uses the rules below (shown in declining order of precedence):

1. You may set the concurrency mode for a specific table by using the ALTER TABLE command, e.g.

```
ALTER TABLE MyTable SET PESSIMISTIC;  
ALTER TABLE MyTable SET OPTIMISTIC;
```

2. You may set the default concurrency mode for all tables by setting the `solid.ini` configuration parameter `General.Pessimistic`, e.g.

```
[General]  
Pessimistic=yes
```

Note that this parameter takes effect only at the time that the server starts. If you manually edit the `solid.ini` file, the change will not be visible until the server restarts.

Note also that this parameter cannot be set via an ADMIN COMMAND in versions up through and including 4.0.

3. If none of the above methods are used to specify the concurrency mode, then the server defaults to optimistic concurrency.

Since the value of the *General.Pessimistic* can change, the concurrency control for a table may change. It is quite possible for a table to use optimistic concurrency control during one "execution" of the server, and use pessimistic concurrency control during another execution.

For tables whose setting is based on the value of the *General.Pessimistic* parameter, the table uses the current value of the *General.Pessimistic* parameter, not the value at the time that the table was created.

## Reading the Concurrency Mode

For in-memory tables that are used in transactions with an isolation level higher than READ COMMITTED, the server will use pessimistic concurrency control, and you should ignore the rules below.

For all other tables, there is no single method of reading a table's concurrency mode. You must follow the steps below to determine the concurrency mode for the desired table.

1. If a table's concurrency mode was set explicitly with the ALTER TABLE command, then the concurrency mode for that table is recorded in the system table named SYS\_TABLEMODES. You can read the value by executing the following command:

```
SELECT SYS_TABLEMODES.ID, table_name,  
FROM SYS_TABLES, SYS_TABLEMODES  
WHERE SYS_TABLEMODES.ID = SYS_TABLES.ID;
```

Note that this works ONLY if you explicitly set the table's concurrency mode using the ALTER TABLE command.

2. If a table's concurrency mode was not set with the ALTER TABLE command, then check the concurrency control mode specified by the `solid.ini` file at the time that the server started. You can read this level by executing the command:

**ADMIN COMMAND 'describe parameter general.pessimistic';**

If the value in the `solid.ini` file has not been changed since the server started, and if the value has not been overridden by an ADMIN COMMAND, then of course you can determine the value by looking at the `solid.ini` file.

(Note: Prior to version 4.00.0031, the server did not properly recognize the `ADMIN COMMAND` to display the value of the `General.Pessimistic` variable. This means that for earlier versions of the server you will need to look at the value in the `solid.ini` file. If anyone changed the value in the `solid.ini` file since the time that the server started, then you will not know the correct value.)

3. If none of the above apply, then the server will default to optimistic for all tables.

### Shared, Exclusive, and Update Locks

The following lock modes are used only for rows in tables that are using pessimistic locking:

- **SHARED**

Multiple users can hold shared locks on the same row simultaneously. Shared locks are used on read-only or `SELECT` operations. Shared locks allow many users to read data, but do not allow any users to change that data.

- **EXCLUSIVE**

When one user has an exclusive lock on a row, no other lock of any type may be placed on that row. Thus the user with the exclusive lock has exclusive access to that row. Exclusive locks are used on insert, update, and delete operations.

- **UPDATE**

When a user accesses a row with the `SELECT... FOR UPDATE` statement, the row is locked with an update mode lock. This means that no other user can read or update the row, and ensures the current user can later update the row. Update locks are similar to exclusive locks. The main difference between the two is that you can acquire an update lock when another user already has a shared lock on the same record. This lets the holder of the update lock read data without excluding other users; however, once the holder of the update lock changes the data, the update lock is converted to an exclusive lock. A surprising characteristic of update locks is that they are asymmetric with respect to shared locks. A user may acquire an update lock on a record that already has a shared lock; however, a user may not acquire a shared lock on a record that already has an update lock. Because an update lock prevents subsequent read locks, it is easier to convert the update lock to an exclusive lock.

## 6.2.4 Table Locks

So far, we've talked primarily about locking individual rows in a table, such as the bank account information that contains your checking account balance. The server allows table-level locks as well as row-level locks. Many of the principles that apply to locks on individual records also apply to locks on tables.

Why would you want to lock a table? Imagine that you want to alter a table to add a new column. You don't want anyone else to try to add a column with the same name at the same time.

Therefore, when you execute an ALTER TABLE operation, you get a shared lock on that table. That allows other users to continue to read data from the table, but prevents them from making changes to the table. If another user wants to do DDL operations (such as ALTER TABLE) on the same table at the same time, he or she will either have to wait or will get an error message.

Thus basic table locking has much the same purpose and mechanism as record locking. However, there are some additional situations in which table locking is used; it's not always just because one user is trying to update the structure of the table.

Imagine that you are updating a record in a table; for example, perhaps you are updating a customer's home phone number. Meanwhile, another user decides to change the table, dropping the telephone number column and adding an email address column. If we allowed another user to drop the telephone number column and then allowed you to try to write an updated telephone number to that column that no longer exists, the data would undoubtedly be corrupted. Therefore, when a user acquires a shared lock or an exclusive lock on a record in a table, the user also implicitly acquires a lock (usually a shared lock) on the entire table. This prevents the structure of the table from changing while users are in the middle of using any part of that table.

Table-level locks are always "pessimistic"; the server puts a real lock on the table rather than just looking at versioning information. This is true even if the table is set to optimistic locking. (The terms here may be confusing. Keep in mind that when you set the lock mode for a table, you are really setting the lock mode for the rows in the table, not the table itself. In other words, you are setting the lock mode for row-level locks, not table-level locks.)

Unless you are altering the table, the locks on tables are usually shared locks. These table locks usually have a "timeout" of 0 seconds — if you can't get the lock immediately, then the server does not wait; it just gives you an error message.

There is a third possible reason for locking an entire table. Suppose that you want to change every record in the table within a single transaction. For example, suppose that it's 12:01 AM January 1st, and you want to credit all of the savings accounts with the interest that they earned last year. You could acquire an individual exclusive lock on each record in the table, but this is inefficient. You'd like to get an exclusive lock on the entire table. Checking this one lock is more efficient than checking potential locks on every record in the table. Naturally, if some other user has a lock on the table (such as the shared table lock that she acquires as a result of locking any record in the table), then you won't be able to acquire an exclusive lock on that table. The rules regarding exclusive/shared locks are the same for tables as for records: you can have as many shared locks as you want, but only one exclusive lock may exist at a time; furthermore, you can't have a combination of exclusive and shared locks.



When the server recognizes that a particular operation (such as an UPDATE statement without a where clause) will affect every record in the table, the server itself can lock the entire table if it thinks that would be most efficient, and if no conflicting locks on the table already exist.

Thus we see that table locks can be used for at least three purposes:

1. to protect against two users trying to change the table at the same time
2. to prevent the table from being changed while records within the table are being changed
3. to increase efficiency of operations that do mass updates

Most table-level locks are implicit — in other words, the server itself sets those locks when necessary. However, you can also set table-level locks explicitly by using the LOCK TABLE command. This is useful when using the Maintenance Mode feature set. See the chapter "Updating and Maintaining the Schema of a Distributed System" in *IBM solidDB Advanced Replication Guide* for more details.

### Table-Level Locking

The EXCLUSIVE and SHARED lock modes (see the section called “Shared, Exclusive, and Update Locks”) are used for both pessimistic and optimistic tables.



#### Note

By default, optimistic and pessimistic tables are always locked in shared mode. In addition, some IBM solidDB statements that are optionally run with the PESSIMISTIC keyword use EXCLUSIVE table level locks even when the tables are optimistic.

## 6.2.5 Lock Duration

The purpose of a transaction (a sequence of statements that are all committed or rolled back together) is to ensure that data is internally consistent. This may require locks to be held until the end of the transaction.

Let's review the subject of transactions first. Suppose that you just bought a new bicycle and paid for it by check. The bank must subtract the price of the bicycle from your account and must add the price of the bicycle to the bike store's account. These 2 operations must be done "together" or else money may seem to disappear to, or appear from, nowhere. For example, suppose that we subtracted the money from your account, then committed the transaction, and then failed to update the bike store's account (perhaps because a power failure occurred immediately before we updated the store's account). You would be poorer, but the bike store would be no richer. The money would seem to disappear (and you'd probably have a very angry bicycle dealer demanding that you pay again for something you've already paid for).

If we put the two operations (subtracting from your account and adding to the store's account) into the same transaction, then no money ever disappears. If the transaction is interrupted (and rolled back) for some reason such as a power failure, then we can retry the same operation again later without risking the possibility of charging you twice (or not paying the store at all).

Generally, an update lock is held from the time it is acquired until the time that the transaction completes (via commit or rollback). If the lock were not held until the end of the transaction, then rollback might fail. (Imagine what would happen if someone else updated the record after you updated it but before you finished your transaction. If you have to roll back for some reason, the server would have to figure out whether to roll back the other user's changes — or might simply lose those changes, even if the other user continued on and committed her transaction.)

In IBM solidDB, shared locks ("read locks") are also held until the end of the transaction. IBM solidDB servers differ from some other servers in this regard. Some servers will release shared locks before the end of a transaction if the Transaction Isolation Level is low enough.

You might wonder whether the transaction isolation level affects the server's behavior with regard to shared locks if those shared locks are always held until the end of the transaction. There are still some differences between the isolation levels, even when locks are held until the end of the transaction. For example, **SERIALIZABLE** isolation level does additional checks. It checks also that no new rows are added to the result set that the transaction should have seen. In other words, it prevents other users from inserting rows that would have qualified for the result set that is in the transaction. For example, suppose that I have a **SERIALIZABLE** transaction that has an update command like:

```
UPDATE customers SET x = y WHERE area_code = 415;
```

In a **SERIALIZABLE** transaction, the server does not allow other users to enter records with `area_code=415` until the serializable transaction is committed.

See the next section for a more detailed discussion of transaction isolation.

## 6.2.6 TRANSACTION ISOLATION Levels

In a "simple" world, shared locks would be released as soon as you were done looking at the data. (Update locks, as discussed above, are held until the end of the transaction.)

But the world is not always so simple. In some cases, a user may look at a record more than once in a single transaction. For example, if you write a program that uses scroll cursors, your user may scroll back and forth through a list of records, viewing the same record more than once. If the value of that record changed every time that the user looked at it, even within the same transaction, the user might get quite confused. Therefore, many database servers (especially those that comply with the ANSI and ISO standards for the SQL language), allow you to extend the duration of read/shared locks. The idea is to ensure that the data looks the same every

time you view it (within a single transaction). Once you read a record, you acquire a shared lock on that record, and then hold it until the end of the transaction.

(This is really only one of the factors involved in transaction isolation level. The transaction isolation level affects not only how long you lock a record, but also what you see. For example, on systems that (unlike IBM solidDB's) allow both "read uncommitted" (sometimes called "dirty read") and "read committed", your isolation level affects what YOU see, not just what other users can and can't see because you've locked certain records.)

In IBM solidDB, isolation levels may be set globally, with a configuration parameter, or per session and per transaction. For more details, please refer to For details, read chapter Choosing Transaction Isolation Levels in *IBM solidDB Administration Guide*.

## 6.2.7 Miscellaneous Lock Information

All locks within a particular category (such as shared locks) are "equal". It does not matter who placed the lock. Locks placed by a DBA are no more and no less "strong" than locks placed by any other user. It does not matter whether the lock was executed as part of a statement typed in interactively, called from a compiled remote application, or called from within the local application when using the linked library access. It does not matter whether the lock was placed as a result of a statement inside a stored procedure or trigger.

With pessimistic locking, the first user to request a lock gets it. Once you have the lock, no other user or connection can override your lock. In IBM solidDB, the lock lasts until the end of the transaction or in the case of "long" table locks, the lock lasts until you explicitly release it.

Note that some locks can be "escalated". For example, if you are using a scroll cursor and you acquire a shared lock on a record, and then later within that same transaction you update that record, your shared lock may be upgraded to an exclusive lock. Of course, getting an exclusive lock is only possible if there are no other locks (shared or exclusive) on the table; if you and another user both have shared locks on the same record, then the server cannot upgrade your shared lock to an exclusive lock until the other user drops her shared lock.

### Notes on Table Locks

Although table locks are normally used with "Maintenance Mode" operations, the two features are independent. You may use the table lock feature with or without the Maintenance Mode feature.

In the replica, exclusive table locks are issued implicitly on publication tables if refreshed with the PESSIMISTIC keyword.

IBM solidDB issues implicit table shared locks in all DDL and DML operations. These prevent one user from dropping the table while another user is updating data in the table.

## 6.2.8 Summary of Locking Information

Locks prevent two users from doing conflicting operations at the same time. Operations "conflict" if at least one of the operations involves updating the data (via UPDATE, DELETE, INSERT, ALTER TABLE, etc.). If all the operations are read-only operations (such as SELECT), then there is no conflict. The current version of IBM solidDB does not allow users to specify row-level locks explicitly. There is no "LOCK RECORD" command; the server does all row-level locking for you. The server also does table-level locking for you. If you need to set table-level locks explicitly, you may do so using the LOCK TABLE command.

## 6.3 Choosing the Transaction Durability

If you can afford to lose a small amount of recent data, and if performance is crucial to you, then you may want to use relaxed durability. Relaxed durability is appropriate when each individual transaction is not crucial. For example, if you are monitoring system performance and you want to store data on response times, you may only be interested in average response times, which will not be significantly affected if you are missing a few pieces of data. In fact, since measuring performance will itself affect performance (by using up resources such as CPU time and I/O bandwidth), you probably want your performance tracking operations themselves to have high performance (low cost) rather than high precision. Relaxed durability is appropriate in this situation.

On the other hand, if you are tracking financial data, such as bill payments, then you probably want to ensure that 100% of your committed data is stored and recoverable. In this situation, you will want strict durability.

You should use relaxed durability ONLY when you can afford to lose a few of the most recent transactions. Otherwise, use strict durability. If you are not sure whether strict or relaxed durability is appropriate, use strict durability.

### 6.3.1 Setting the Transaction Durability Level

There are four ways to set the transaction durability level. These are listed below in descending order of precedence:

1. SET TRANSACTION DURABILITY

```
SET TRANSACTION DURABILITY { RELAXED | STRICT }
```

For example

```
SET TRANSACTION DURABILITY RELAXED;  
SET TRANSACTION DURABILITY STRICT;
```

If you use the `SET TRANSACTION DURABILITY` command, then you specify the transaction durability on a per-transaction basis. The command affects only the current transaction.

#### 2. SET DURABILITY

```
SET DURABILITY { RELAXED | STRICT }
```

For example

```
SET DURABILITY RELAXED;  
SET DURABILITY STRICT;
```

If you use the `SET DURABILITY` command, then you specify the transaction durability on a per-session basis. A *session* is the time between connecting and disconnecting to the server. Each user has a separate session, even if the sessions overlap in time. In fact, a single user may establish more than one session (for example, by running multiple copies of `solsql`, or by writing a program that makes multiple connections to the same server). When you specify the transaction durability level by using the `SET DURABILITY` statement, you are specifying it only for the session in which the command is issued. Your choice will not affect any other user, any other open session that you yourself currently have, or any future session that you may have. Each user session may set its own transaction durability level, based on how important it is for the session not to lose any data.

The effect of this statement lasts until the end of the session, or until another `SET DURABILITY` command is issued.

#### 3. Setting the *DurabilityLevel* parameter in the `solid.ini` configuration file.

```
[Logging]  
DurabilityLevel=3
```

See chapter *DurabilityLevel* in *IBM solidDB Advanced Replication Guide*.

This setting affects all users.

This parameter can be changed dynamically. If you want to change the default setting while the server is running, you may do so by using the following command:

```
ADMIN COMMAND 'parameter Logging.DurabilityLevel={1 | 2 | 3}'
```

If you execute this command, it will take effect immediately.

4. By default, if you do not set the transaction durability level using any of the methods above, the server will use strict durability.

If you are using strict durability, you may also set an additional configuration parameter (*LogWriteMode*), which also influences performance. For details about *LogWriteMode*, see its description in *IBM solidDB Administration Guide*.

---

# Chapter 7. Diagnostics and Troubleshooting

This chapter provides information on the following IBM solidDB diagnostic tools:

- SQL info facility and the EXPLAIN PLAN FOR statement used to tune your application and identify inefficient SQL statements in your application.
- Tracing facilities for stored procedures and triggers

You can use these facilities to observe performance, troubleshoot problems, and produce high quality problem reports. These reports let you pinpoint the source of your problems by isolating them under product categories (such as IBM solidDB ODBC API, IBM solidDB ODBC Driver, IBM solidDB JDBC Driver, etc.).

## 7.1 Observing Performance

You can use the SQL Info facility to provide information on a SQL statement and the SQL statement EXPLAIN PLAN FOR to show the execution graph that the SQL optimizer selected for a given SQL statement. Typically, if you need to contact IBM Corporation technical support, you will be asked to provide the SQL statement, EXPLAIN PLAN output, and SQL Info output from the EXPLAIN PLAN run with info level 8 for more extensive trace output.

### 7.1.1 SQL Info Facility

Run your application with the SQL Info facility enabled. The SQL Info facility generates information for each SQL statement processed by IBM solidDB.

The *Info* parameter in the *[SQL]* section specifies the tracing level on the SQL parser and optimizer as an integer between 0 (no tracing) and 8 (IBM solidDB info from every fetched row). Trace information will be output to the file named `soltrace.out` in the IBM solidDB directory.

Example:

```
[SQL]
info = 1
```

**Table 7.1. SQL Info Levels**

Info value	Information
0	no output
1	table, index, and view info in SQL format
2	SQL execution graphs (for IBM Corporation technical support use only)
3	some SQL estimate info, IBM solidDB selected key name
4	all SQL estimate info, IBM solidDB selected key info
5	IBM solidDB info also from discarded keys
6	IBM solidDB table level info
7	SQL info from every fetched row
8	IBM solidDB info from every fetched row

The SQL Info facility can also be turned on with the following SQL statement (this sets SQL Info on only for the client that executes the statement):

```
SET SQL INFO ON LEVEL info_value FILE file_name
```

and turned off with the following SQL statement:

```
SET SQL INFO OFF
```

Example:

```
SET SQL INFO ON LEVEL 1 FILE 'my_query.txt'
```

## 7.1.2 The EXPLAIN PLAN FOR Statement

The syntax of the EXPLAIN PLAN FOR statement is:



EXPLAIN PLAN FOR *sql\_statement*

The EXPLAIN PLAN FOR statement is used to show the execution plan that the SQL optimizer has selected for a given SQL statement. An execution plan is a series of primitive operations, and an ordering of these operations, that IBM solidDB performs to execute the statement. Each operation in the execution plan is called a unit.

**Table 7.2. EXPLAIN PLAN FOR Units**

Unit	Description
JOIN UNIT*	Join unit joins two or more tables. The join can be done by using loop join or merge join.
TABLE UNIT	The table unit is used to fetch the data rows from a table or index.
ORDER UNIT	Order unit is used to order rows for grouping or to satisfy ORDER BY. The ordering can be done in memory or using an external disk sorter.
GROUP UNIT	Group unit is used to do grouping and aggregate calculation (SUM, MIN, etc.).
UNION UNIT*	Union unit performs the UNION operation. The unit can be done by using loop join or merge join.
INTERSECT UNIT*	Intersect unit performs the INTERSECT operation. The unit can be done by using loop join or merge join.
EXCEPT UNIT*	Except unit performs the EXCEPT operation. The unit can be done by using loop join or merge join.

\*This unit is generated also for queries that reference only a single table. In that case no join is executed in the unit; it simply passes the rows without manipulating them.

The table returned by the EXPLAIN PLAN FOR statement contains the following columns.

**Table 7.3. Explain Plan Table Columns**

Column Name	Description
ID	The output row number, used only to guarantee that the rows are unique.
UNIT_ID	This is the internal unit id in the SQL interpreter. Each unit has a different id. The unit id is a sparse sequence of numbers, because the SQL interpreter generates unit ids also for those units that are removed during the optimization phase. If more than one row has the same unit id it means that those rows be-

Column Name	Description
	long to the same unit. For formatting reasons the info from one unit may be divided into several different rows.
PAR_ID	Parent unit id for the unit. The parent id number refers to the id in the UNIT_ID column.
JOIN_PATH	For join, union, intersect, and except units there is a join path which specifies which tables are joined in the unit and the join order for tables. The join path number refers to the unit id in the UNIT_ID column. It means that the input to the unit comes from that unit. The order in which the tables are joined is the order in which the join path is listed. The first listed table is the outermost table in a loop join.
UNIT_TYPE	Unit type is the execution graph unit type.
INFO	Info column is reserved for additional information. It may contain, for example, index usage, the database table name and constraints used in IBM solidDB to select rows. Note that the constraints listed here may not match those constraints given in the SQL statement.

The following texts may exist in the INFO column for different types of units.

**Table 7.4. Texts in the Unit INFO Column**

Unit type	Text in Info column	Description
TABLE UNIT	<i>tablename</i>	The table unit refers to table <i>tablename</i> .
TABLE UNIT	<i>constraints</i>	The constraints that are passed to the database engine are listed. If for example in joins the constraint value is not known in advance, the constraint value is displayed as NULL.
TABLE UNIT	SCAN TABLE	Full table scan is used to search for rows.
TABLE UNIT	SCAN <i>indexname</i>	Index <i>indexname</i> is used to search for rows. If all selected columns are found from an index, sometimes it is faster to scan the index instead of the entire table because the index has fewer disk blocks.
TABLE UNIT	PRIMARY KEY	The primary key is used to search rows. This differs from SCAN in that the whole table is not scanned because there is a limiting constraint to the primary key attributes.

### 7.1.2 The EXPLAIN PLAN FOR Statement

Unit type	Text in Info column	Description
TABLE UNIT	INDEX <i>indexname</i>	Index <i>indexname</i> is used to search for rows. For every matching index row, the actual data row is fetched separately.
TABLE UNIT	INDEX ONLY <i>indexname</i>	Index <i>indexname</i> is used to search for rows. All selected columns are in the index, so the actual data rows are not fetched separately by reading from the table.
JOIN UNIT	MERGE JOIN	Merge join is used to join the tables.
JOIN UNIT	3-MERGE JOIN	A 3-merge join is used to merge the tables.
JOIN UNIT	LOOP JOIN	Loop join is used to join the tables.
ORDER UNIT	NO ORDERING REQUIRED	No ordering is required, the rows are retrieved in correct order from IBM solidDB.
ORDER UNIT	EXTERNAL SORT	External sorter is used to sort the rows. To enable external sorter, the temporary directory name must be specified in the Sorter section of the configuration file.
ORDER UNIT	FIELD <i>n</i> USED AS PARTIAL ORDER	For distinct result sets, an internal sorter (in-memory sorter) is used for sorting and the rows retrieved from IBM solidDB are partially sorted with column number <i>n</i> . The partial ordering helps the internal sorter avoid multiple passes over the data.
ORDER UNIT	<i>n</i> FIELDS USED FOR PARTIAL SORT	An internal sorter (in-memory sorter) is used for sorting and the rows retrieved from IBM solidDB are partially sorted with <i>n</i> fields. The partial ordering helps the internal sorter to avoid multiple passes over the data.
ORDER UNIT	NO PARTIAL SORT	Internal sorter is used for sorting. The rows are retrieved in random order from IBM solidDB to the sorter.
UNION UNIT	MERGE JOIN	Merge join is used to join the tables.
UNION UNIT	3-MERGE JOIN	A 3-merge join is used to merge the tables.
UNION UNIT	LOOP JOIN	Loop join is used to join the tables.
INTERSECT UNIT	MERGE JOIN	Merge join is used to join the tables.
INTERSECT UNIT	3-MERGE JOIN	A 3-merge join is used to merge the tables.

Unit type	Text in Info column	Description
INTERSECT UNIT	LOOP JOIN	Loop join is used to join the tables.
EXCEPT UNIT	MERGE JOIN	Merge join is used to join the tables.
EXCEPT UNIT	3-MERGE JOIN	A 3-merge join is used to merge the tables.
EXCEPT UNIT	LOOP JOIN	Loop join is used to join the tables.

### Example 1

```
EXPLAIN PLAN FOR SELECT * FROM TENKTUP1 WHERE
UNIQUE2_NI BETWEEN 0 AND 99;
```

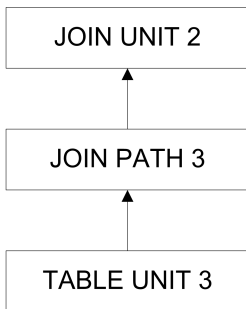
**Table 7.5. EXPLAIN PLAN FOR, Example 1**

ID	UNIT_ID	PAR_ID	JOIN_PATH	UNIT_TYPE	INFO
1	2	1	3	JOIN UNIT	
2	3	2	0	TABLE UNIT	TENKTUP1
3	3	2	0		FULL SCAN
4	3	2	0		UNIQUE2_NI <= 99
5	3	2	0		UNIQUE2_NI >= 0
6	3	2	0		

### Execution graph:

JOIN UNIT 2 gets input from TABLE UNIT 3

TABLE UNIT 3 for table TENKTUP1 does a full table scan with constraints UNIQUE2\_NI <= 99 and UNIQUE2\_NI >= 0

**Figure 7.1. Execution Graph 1****Example 2**

```

EXPLAIN PLAN FOR SELECT * FROM TENKTUP1, TENKTUP2
WHERE TENKTUP1.UNIQUE2 > 4000 AND TENKTUP1.UNIQUE2 < 4500
AND TENKTUP1.UNIQUE2 = TENKTUP2.UNIQUE2;
  
```

**Table 7.6. EXPLAIN PLAN FOR, Example 2**

ID	UNIT_ID	PAR_ID	JOIN_PATH	UNIT_TYPE	INFO
1	6	1	9	JOIN UNIT	MERGE JOIN
2	6	1	10		
3	9	6	0	ORDER UNIT	NO ORDERING RE- QUIRED
4	8	9	0	TABLE UNIT	TENKTUP2
5	8	9	0		PRIMARY KEY
6	8	9	0		UNIQUE2 < 4500
7	8	9	0		UNIQUE2 > 4000
8	8	9	0		
9	10	6	0	ORDER UNIT	NO ORDERING RE- QUIRED
10	7	10	0	TABLE UNIT	TENKTUP1
11	7	10	0		PRIMARY KEY
12	7	10	0		UNIQUE2 < 4500

ID	UNIT_ID	PAR_ID	JOIN_PATH	UNIT_TYPE	INFO
13	7	10	0		UNIQUE2 > 4000
14	7	10	0		

### Execution Graph:

JOIN UNIT 6 the input from order units 9 and 10 are joined using merge join algorithm

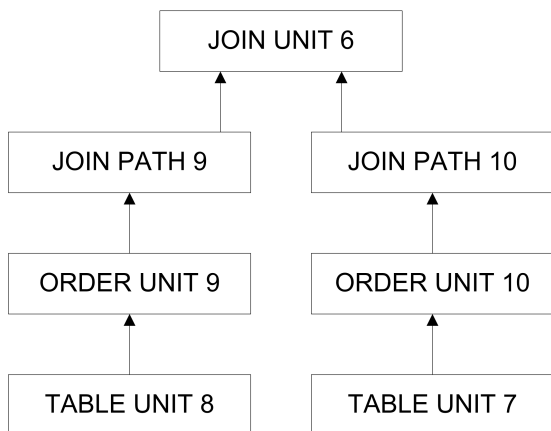
ORDER UNIT 9 orders the input from TABLE UNIT 8. Since the data is retrieved in correct order, no real ordering is needed

ORDER UNIT 10 orders the input from TABLE UNIT 7. Since the data is retrieved in correct order, no real ordering is needed

TABLE UNIT 8: rows are fetched from table TENKTUP2 using primary key. Constraints UNIQUE2 < 4500 and UNIQUE2 > 4000 are used to select the rows

TABLE UNIT 7: rows are fetched from table TENKTUP1 using primary key. Constraints UNIQUE2 < 4500 and UNIQUE2 > 4000 are used to select the rows

**Figure 7.2. Execution Graph 2**



## 7.2 Problem Reporting

IBM solidDB offers sophisticated diagnostic tools and methods for producing high quality problem reports with very limited effort. Use the diagnostic tools to capture all the relevant information about the problem.

All problem reports should contain the following files and information:

- `solid.ini`
- license number
- `solmsg.out`
- `solerror.out`
- `soltrace.out`
- problem description
- steps to reproduce the problem
- all error messages and codes
- contact information, preferably email address of the contact person

## 7.3 Problem Categories

Most problems can be divided into the following categories:

- IBM solidDB ODBC API
- IBM solidDB ODBC or JDBC Driver
- UNIFACE driver for IBM solidDB
- Communication problems between the application or an external application (if using the linked library access) and IBM solidDB.

The following pages include detailed instructions to produce a proper problem report for each problem type. Please follow the guidelines carefully.

### 7.3.1 IBM solidDB ODBC API Problems

If the problem concerns the performance of a specific IBM solidDB ODBC API or SQL statement, you should run SQL info facility at level 4 and include the generated `soltrace.out` file into your problem report. This file contains the following information:

- create table statements

- create view statements
- create index statements
- SQL statement(s)

### 7.3.2 IBM solidDB ODBC Driver Problems

If the problem concerns the performance of IBM solidDB ODBC Driver, please include the following information:

- IBM solidDB ODBC Driver name, version, and size
- ODBC Driver Manager version and size

If the problem concerns the cooperation of IBM solidDB and any third party standard software package, please include the following information:

- Full name of the software
- Version and language
- Manufacturer
- Error messages from the third party software package

Use ODBC trace option to get a log of the ODBC statements and include it in your problem report.

### 7.3.3 IBM solidDB JDBC Driver Problems

If the problem is related to the IBM solidDB JDBC Driver, please include the following information in your problem report:

- Exact version of JDK or JRE used
- Name, size, and date of the `SOLIDDriver` class package
- Contents of `DriverManager.setLogStream(someOutputStream)` output, if available
- Call stack (that is, `Exception.printStackTrace()` output) of the application, if an exception has occurred in the application



## 7.3.4 UNIFACE Driver for IBM solidDB Problems

If the problem concerns the performance of IBM solidDB UNIFACE Driver, please include the following information:

- IBM solidDB UNIFACE Driver version and size
- UNIFACE version and platform
- Contents of the UNIFACE message frame
- Error codes from the driver, \$STATUS, \$ERROR
- All necessary files to reproduce the problem (TRXs, SQL scripts, USYS.ASN etc.)

## 7.3.5 Communication between a Client and Server

If the problem concerns the performance of the communication between a client and server use the Network trace facility and include the generated trace files into your problem report. Please include the following information:

- IBM solidDB communication DLLs used: version and size
- other communication DLLs used: version and size
- description of the network configuration

## 7.4 Tracing Facilities for Stored Procedures and Triggers

When debugging a stored procedure or a trigger, you may want to add "trace" commands to see which parts of the code are executing. Or you may want to trace every statement within the procedure or trigger. The following two sections explain how to do these things.

### 7.4.1 User-Definable Trace Output from Procedure Code

From inside your stored procedure or trigger, you can send "trace" output to the `soltrace.out` file by using the following command:

WRITETRACE (entry VARCHAR)

You can turn the output on or off by using the command:

```
ADMIN COMMAND 'usertrace { on | off }  
user username { procedure | trigger | table } entity_name'
```

The "entity\_name" is the name of the procedure, trigger, or table for which you want to turn tracing on or off. If the keyword "table" is specified, then all triggers on that table are traced.

You may turn on (or off) tracing for a specified procedure, a specified trigger, or for all triggers on a specified table.

Trace is activated only when the specified user calls the procedure / trigger. This is useful, for example, when tracing propagated procedure calls in a advanced replication master.

Turning on tracing turns it on in all procedure/trigger calls by this user, not just calls from the connection that switched the trace on. If you have multiple connections that use the same username, then all of the calls in all of those connections will be traced. Furthermore, the tracing will be done on calls propagated to (executed on) the master, as well as the calls executed on the replica.

## 7.4.2 Procedure Execution Trace

If you must trace EVERY statement in your stored procedure or trigger, then you don't want to spend time to write a WRITETRACE statement for every SQL statement. Instead, you can simply turn on "PROCTRACE", which traces every statement inside the specified stored procedure or trigger. As with USERTRACE, you can turn proctrace on for a specified procedure, a specified trigger, or for all triggers associated with a particular table. The syntax is:

```
ADMIN COMMAND 'proctrace { on | off }  
user username { procedure | trigger | table } entity_name'
```

The "entity\_name" is the name of the procedure, trigger, or table for which you want to turn tracing on or off.

Trace is activated only when the specified user calls the procedure / trigger. This is useful, for example, when tracing propagated procedure calls in a advanced replication master.

Turning on tracing turns it on in all procedure/trigger calls by this user, not just calls from the connection that switched the trace on. If you have multiple connections that use the same username, then all of the calls in

all of those connections will be traced. Furthermore, the tracing will be done on calls propagated to (executed on) the master, as well as the calls executed on the replica.

If the keyword "table" is specified, then all triggers on that table are traced.

Example:

```
"create procedure trace_sample(i integer)
returns(j integer)
begin
    j := 2*i;
    return row;
end";
commit work;
```

```
admin command 'proctrace on user DBA procedure TRACE_SAMPLE';
call trace_sample(2);
```

OUTPUT FROM EXAMPLE:

```
23.01 17:25:17 ---- PROCEDURE 'DBA.DBA.TRACE_SAMPLE' TRACE BEGIN ----
0001:CREATE PROCEDURE TRACE_SAMPLE(I INTEGER)
0002:RETURNS(J INTEGER)
0003:BEGIN
    --> I:=2
    --> J:=NULL
    --> SQLSUCCESS:=1
    --> SQLERRNUM:=NULL
    --> SQLERRSTR:=NULL
    --> SQLROWCOUNT:=NULL

0004:      J := 2*I;
    --> J:=4
0005:      RETURN ROW;
```

```
0006:END  
23.01 17:25:17 ----- PROCEDURE 'DBA.DBA.TRACE_SAMPLE' TRACE END -----
```

## 7.5 Measuring and Improving Performance of START AFTER COMMIT Statements

### 7.5.1 Tuning Performance of START AFTER COMMIT Statements

Background tasks can be controlled with SSC-API and admin commands (see IBM solidDB Linked Library Access User Guide for details). The task type `SSC_TASK_BACKGROUND` is used for the tasks that execute statements started with `START AFTER COMMIT`. You can give this task type higher priority or lower priority, or you may suspend this task type.

Note that there may be more than one of these tasks, but you cannot control them individually. In other words, if you call `SSCSuspendTaskClass` for `SSC_TASK_BACKGROUND`, it will suspend all the background tasks.

### 7.5.2 Analyzing Failures in START AFTER COMMIT Statements

There is a limit on the number of uncommitted `START AFTER COMMIT` statements that may exist simultaneously. (By "uncommitted", we mean that the transaction in which the `START AFTER COMMIT` statement was executed has not yet been committed. At this point, the body of the `START AFTER COMMIT` statement — e.g. the procedure call — has not yet even started to execute.) If the maximum is reached, then an error is returned when the next `START AFTER COMMIT` is issued. The maximum number is configurable in `solid.ini` using the parameter named *MaxStartStatements* (for details, see the description of this parameter in *IBM solidDB Administration Guide*).

If a statement cannot be started, the reason for it is logged into the system table `SYS_BACKGROUNDJOB_INFO`. Only failed `START AFTER COMMIT` statements are logged into this table. For more details about this table, see Section D.1.3, "`SYS_BACKGROUNDJOB_INFO`".

The user can retrieve the information from the table `SYS_BACKGROUNDJOB_INFO` using either an SQL `SELECT` statement or by calling the system procedure `SYS_GETBACKGROUNDJOB_INFO`. The stored procedure `SYS_GETBACKGROUNDJOB_INFO` returns the row that matches the given jobid of the `START AFTER COMMIT` statement. For more details about `SYS_GETBACKGROUNDJOB_INFO`, see Section E.2.1, "`SYS_GETBACKGROUNDJOB_INFO`".

If you want to be notified when a statement fails to start, you can wait on the system event `SYS_EVENT_SACFAILED`. See its description in Section F.1, "Miscellaneous Events" for details about this event. The application can wait for this event and use the jobid to retrieve the error message from the system table `SYS_BACKGROUNDJOB_INFO`.

---

# Chapter 8. Performance Tuning

This chapter discusses techniques that you can use to improve the performance of IBM solidDB. The topics included in this chapter are:

- Tuning SQL statements and applications
- Optimizing single-table SQL queries
- Using indexes to improve query performance
- Waiting on events
- Optimizing batch inserts and updates
- Using Optimizer hints for performance
- Diagnosing poor performance

For tips on optimizing advanced replication data synchronization, see *IBM solidDB Advanced Replication Guide*.

## 8.1 Tuning SQL Statements and Applications

Tuning the SQL statements, especially in applications where complex queries are involved, is generally the most efficient means of improving the database performance.

Be sure to tune your application *before* tuning the RDBMS because:

- during application design you have control over the SQL statements and data to be processed
- you can improve performance even if you are unfamiliar with the internal working of the RDBMS you are going to use
- if your application is not tuned well, it will not run well even on a well-tuned RDBMS

You should know what data your application processes, what are the SQL statements used, and what operations the application performs on the data. For example, you can improve query performance when you keep SELECT statements simple, avoiding unnecessary clauses and predicates.

## 8.1.1 Evaluating Application Performance

To isolate areas where performance is lacking in your application, the IBM solidDB provides the following diagnostic tools for observing database performance:

- SQL info facility
- EXPLAIN PLAN FOR statement

These tools are helpful in tuning your application and identifying any inefficient SQL statements in it. Read Chapter 7, *Diagnostics and Troubleshooting* for additional information on how to use these tools.

In addition, the following commands provide useful information for evaluating performance.

- **ADMIN COMMAND 'status'**

This command returns statistics information from the server. For details, read about this command in *IBM solidDB Administration Guide*.

- **ADMIN COMMAND 'perfmon'**

The command returns detailed performance statistics from the server. For details, read about perfmon and "Detailed DBMS Monitoring and Troubleshooting" in *IBM solidDB Administration Guide*.

- **ADMIN COMMAND 'trace'**

This command switches tracing on for SQL statements and network communication. For complete syntax, see the trace option syntax under Section B.1, "ADMIN COMMAND".

## 8.1.2 Using Stored Procedure Language

Using stored procedures can speed up some operations in two ways:

- Statements in stored procedures are parsed and compiled once and then stored in compiled form. Statements outside stored procedures are re-parsed and compiled every time that they are executed. Thus, putting statements in stored procedures reduces overhead (parsing and compiling) if the statements are executed more than once.
- If you have multiple statements inside a single stored procedure, calling that stored procedure once may use fewer network "trips" than passing each statement individually from the client to the server.

## 8.2 Optimizing Single-Table SQL Queries

IBM solidDB provides a Simple SQL Optimization feature that increases performance with specific types of single-table SQL queries. Performance improvements apply to SELECT, DELETE, and UPDATE statements. The feature does not apply to INSERT statements.

Simple SQL Optimization is enabled/disabled by the *SimpleSQLOpt* parameter in the *[SQL]* section of the *solid.ini* file. By default, this feature is turned on and the *SimpleSQLOpt* parameter does not appear in the *solid.ini* file. To disable the feature, you must add the following lines to the *solid.ini* file:

```
[SQL]
SimpleSQLOpt=No
```

Once you have added these lines to the file, you can always enable the feature by specifying *SimpleSQLOpt=Yes* or removing the parameter from the *[SQL]* section. As always, remember that any changes to the *solid.ini* file do not take effect until the server restarts.

When simple SQL optimization is turned on, IBM solidDB automatically optimizes single-table SQL queries that meet the following conditions:

- The statement accesses only a single table.
- The statement does not contain a view, subquery, UNION, INTERSECT, etc.
- The statement does not use ROWNUM.
- The statement does not use a IBM solidDB sequence object that is used to retrieve sequence numbers.

Note that like other optimization techniques, the Simple SQL Optimization feature speeds up most queries, but reduces performance for a few types of queries. If you find your particular queries run more slowly when you are using simple SQL optimization, you can turn off the feature.

## 8.3 Using Indexes to Improve Query Performance

You can use indexes to improve the performance of queries. A query that references an indexed column in its WHERE clause can use the index. If the query selects only the indexed column, the query can read the indexed column value directly from the index, rather than from the table.

If all the fields in the SELECT list of a query are in an index, then the IBM solidDB optimizer can simply use that index, rather than doing an extra lookup to read the complete record. Similarly, if all the fields of a WHERE clause are in an index, then the optimizer can use that index — if the information in the index is

enough to prove that the record won't qualify for the WHERE clause, then the optimizer can avoid looking up the complete record.

For example, suppose that we have a WHERE clause that refers to two or more columns, e.g.

```
WHERE col1 = x AND col2 >= a AND col2 <=b
```

Suppose further that we have an index that contains both col1 and col2, and that has either col1 or col2 as the leading column of the key. For example, if we have an index on col2 + col3 + col1 then this index contains both columns, and one of those columns (col2) is the leading column in the key. If the user's query is

```
SELECT col1, col4  
  FROM table1  
 WHERE col1 = x AND col2 >= a AND col2 <=b;
```

then we do not need to look up the complete record unless the search criteria are met. After all, if the search criteria are not met, then we don't care what value col4 has and so we don't need to look up the full record.

If a table has a primary key, IBM solidDB orders the rows on disk in the order of the values of the primary key. Since the rows are physically in order by the primary key, the primary key itself serves as an index, and optimization tips that apply to indexes also apply to the primary key.

If the table does not have a user-specified primary key, then the rows are ordered using the ROWID. The ROWID is assigned to each row when it is inserted, and each record gets a larger ROWID than the record inserted before it. Thus, in tables without user-specified primary keys, the records are stored in the order in which those rows were inserted. For more information about primary keys, read Section 5.4.2, "Primary Key Indexes".

Searches with row value constructor constraints are optimized to use an index if an index is available. For efficiency, IBM solidDB uses an index to resolve row value constructor constraints of the form (A, B, C) >= (1, 2, 3), where the operator may be any of the following: <, <=, >= and >. (The server does not use an index to resolve row value constructor constraints that contain the operators =, !=, or <>. The server may, of course, use an index to resolve other types of constraints that use =, !=, or <>.) For more information about row value constructors, see Section 3.5.5, "Row Value Constructors".

Indexes improve the performance of queries that select a small percentage of rows from a table. You should consider using indexes for queries that select less than 15% of table rows.



## 8.3.1 Full Table Scan

If a query can use an index, IBM solidDB must perform a full table scan to execute the query. This involves reading all rows of a table sequentially. Each row is examined to determine whether it meets the criteria of the query's WHERE clause. Finding a single row with an indexed query can be substantially faster than finding the row with a full table scan. On the other hand, a query that selects more than 15% of a table's rows may be performed faster by a full table scan than by an indexed query.

You should check every query using the EXPLAIN PLAN statement. (You should use your real data when doing this, since the best plan will depend upon the actual amount of data and the characteristics of that data.) The output from the EXPLAIN PLAN statement allows you to detect whether an index is really used and if necessary you can redo the query or the index. Full table scans often cause slow response time for SELECT queries, as well as excessive disk activity. To diagnose performance degradation problems, you can request statistics on file operations using ADMIN COMMAND 'perfmon' as described in *IBM solidDB Administration Guide* (see the section titled "Detailed DBMS Monitoring and Troubleshooting").

To perform a full table scan, every block in the table is read. For each block, every row stored in the block is read. To perform an indexed query, the rows are read in the order in which they appear in the index, regardless of which blocks contain them. If a block contains more than one selected row it may be read more than once. So, there are cases when a full table scan requires less I/O than an indexed query, if the result set is relatively large.

## 8.3.2 Concatenated Indexes

An index can be made up of more than one column. Such an index is called a concatenated index. We recommend using concatenated indexes when possible.

Whether or not a SQL statement uses a concatenated index is determined by the columns contained in the WHERE clause of the SQL statement. A query can use a concatenated index if it references a leading portion of the index in the WHERE clause. A leading portion of an index refers to the first column or columns specified in the CREATE INDEX statement.

Example:

```
CREATE INDEX job_sal_deptno ON emp(job, sal, deptno);
```

This index can be used by these queries:

```
SELECT * FROM emp WHERE job = 'clerk' and sal =  
800 and deptno = 20;
```

```
SELECT * FROM emp WHERE sal = 1250 and job = salesman;  
SELECT job, sal FROM emp WHERE job = 'manager';
```

The following query does not contain the first column of the index in its WHERE clause and therefore cannot use the index:

```
SELECT * FROM emp WHERE sal = 6000;
```

### Choosing Columns to Index

The following list gives guidelines in choosing columns to index:

- You should create indexes on columns that are used frequently in WHERE clauses.
- You should create indexes on columns that are used frequently to join tables.
- You should create indexes on columns that are used frequently in ORDER BY clauses.
- You should create indexes on columns that have few of the same values or unique values in the table.
- You should not create indexes on small tables (tables that use only a few blocks) because a full table scan may be faster than an indexed query.
- If possible, choose a primary key that orders the rows in the most appropriate order.
- If only one column of the concatenated index is used frequently in WHERE clauses, place that column first in the CREATE INDEX statement.
- If more than one column in a concatenated index is used frequently in WHERE clauses, place the most selective column first in the CREATE INDEX statement.

## 8.4 Waiting on Events

In many programs, you may have to wait for a particular condition to occur before you can perform a certain task. In some cases, you may use a "while" loop to check whether the condition has occurred. IBM solidDB provides Events, which in some cases allow you to avoid wasting CPU time spinning in a loop waiting for a condition.

One (or more) clients or threads can wait on an event, and another client or thread can post that event. For example, several threads might wait for a sensor to get a new piece of data. Another thread (working with that sensor) can post an event indicating that the data is available. For more information about events, see

Section 4.11, “Using Events” and various sections of Appendix B, *IBM solidDB SQL Syntax*, including Section B.12, “CREATE EVENT”.

# 8.5 Optimizing Batch Inserts and Updates

It is highly recommended that you design a database schema that supports running a batch insert in primary key order. Data in the database file is stored physically in the order defined by the primary key of the table. If no primary key is defined, data is stored in the database file in the order it is written to the database. Database operations (that is, reads and writes) always access data at the page level. The default page size of the database is 8 KB.

If the batch write operations are performed in the order that supports the primary key, the caching algorithms of the server are able to group the database file write operations. In this way, a larger number of rows are written to the disk in one physical disk I/O operation. In the worst case, if the insert order is different from the primary key order, each insert or delete operation requires re-writing a database page where only one row has changed.

For these reasons, it makes sense to ensure that tables of a batch write operation have primary keys that match the access order of the batch write operation. This type of database schema can make a significant difference in the performance of the operation.

For example, assume you have the following kind of table:

```
CREATE TABLE USAGE_EVENT (  
  EVENT_ID INTEGER NOT NULL PRIMARY KEY,  
  DEVICE_ID INTEGER NOT NULL,  
  EVENT_DATA VARCHAR NOT NULL);
```

In this table, `EVENT_ID` is a sequence number. The insert and delete operations are done in the order specified by the `EVENT_ID` column, allowing for maximum efficiency.

Note that performance of batch write operations on this same table can be significantly worse if the first column of the primary key were `DEVICE_ID`, but data was written to the database in the `EVENT_ID` order. In this scenario, the number of file-I/O operations needed to complete the batch write operation increases when the size of the table grows.

## 8.5.1 Increasing Speed of Batch Inserts and Updates

You can optimize the speed for large batch inserts and updates to IBM solidDB. Following are guidelines for increasing speed:

1. Check that you are running the application with the AUTOCOMMIT mode set off.

IBM solidDB ODBC Driver's default setting is AUTOCOMMIT. This is the standard setting according to the ODBC specification. To set your application with AUTOCOMMIT off, call the `SQLSetConnectOption` function as in the following example:

```
rc = SQLSetConnectOption
(hdbc, SQL_AUTOCOMMIT, SQL_AUTOCOMMIT_OFF);
```

2. Do not use large transactions. Five hundred (500) rows is recommended as the initial transaction size. The optimal value for the transaction size is dependent on the particular application; you may need to experiment.
3. To make batch inserts faster, you can turn logging off. This, however, increases the risk of data loss during system failure. In some environments, this trade-off is tolerable.

Number 1 and 2 of these guidelines are the most important actions you can take to increase the speed of batch inserts. The actual rate of insertions also depends on your hardware, on the amount of data per row, and on the existing indices for the table.

## 8.6 Using Optimizer Hints

Due to various conditions with the data, user query, and database, the SQL Optimizer is not always able to choose the best possible execution plan. For example, for more efficiency, you may want to force a merge join because you know, unlike the Optimizer, that your data is already sorted.

Or sometimes specific predicates in queries cause performance problems that the Optimizer cannot eliminate. The Optimizer may be using an index that you know is not optimal. In this case, you may want to force the Optimizer to use one that produces faster results.

Optimizer hints provide a way to have better control over response times to meet your performance needs. Within a query, you can specify directives or hints to the Optimizer, which it then uses to determine its query execution plan. Hints are detected through a pseudo comment syntax from SQL-92.

Hints are available for:

- Selecting merge or nested loop join
- Using a fixed join order as given in the from list
- Selecting internal or external sort

- Selecting a particular index
- Selecting a table scan over an index scan
- Selecting sorting before or after grouping

You can place a hint(s) in a SQL statement as a static string, just after a `SELECT`, `UPDATE`, or `DELETE` keyword. Hints are not allowed after the `INSERT` keyword.

Table name resolution in optimizer hints is the same as in any table name in a SQL statement. This means that if there is a table alias name in the query, then you must use the alias, not the table name, in the optimizer hints. For example:

```
SELECT
  --(* vendor(SOLID), product(Engine), option(hint)
  -- FULL SCAN emp_alias *)--
  emp_alias.emp_id, employee_name, dependent_name
FROM employee_table AS emp_alias LEFT OUTER JOIN dependent_table
AS dep_alias
  ON (dep_alias.emp_id = emp_alias.emp_id)
ORDER BY emp_alias.emp_id;
```

If you specify the table name when you should have specified the alias name, you will get the following error message:

```
102: Unused optimizer hint.
```

If you are not using an alias and you are using a table that is in another schema and/or another catalog, then make sure that in the hint you precede the table name with the schema and/or catalog name. For example:

```
SELECT
  --(* vendor(SOLID), product(Engine), option(hint)
  -- FULL SCAN sally_schema.employee_table *)--
  emp_id, employee_name
FROM sally_schema.employee_table;
```

When there is an error in a hint specification, then the whole SQL statement fails with an error message.

Hints are enabled and disabled using the following configuration parameter in `solid.ini`:

```
[Hints]
```

```
EnableHints=YES | NO
```

The default is set to YES.

For more details on Optimizer Hints, including a description of possible hints and examples, refer to Section B.49, “HINT”.

## 8.7 Diagnosing Poor Performance

There are different areas in IBM solidDB that can result in performance degradation. In order to remedy performance problems, you need to determine the underlying cause. Following is a table that lists common symptoms of poor performance, possible causes, and directs you to the section in this chapter for the remedy.

**Table 8.1. Diagnosing Poor Performance**

Symptoms	Diagnosis	Solution
Slow response time for a single query. Other concurrent access to the database is affected. Disk may be busy.	<ul style="list-style-type: none"> <li>Inefficient usage of indexes in the query.</li> <li>Non-optimal decision from the Optimizer.</li> <li>External sorting is not defined and a large internal sorting is causing excessive swapping to disk.</li> </ul>	<p>If index definitions are missing, create new indices or modify existing ones to match the indexing requirements of the slow query. For more details, read Section 8.3, “Using Indexes to Improve Query Performance”.</p> <p>Run the EXPLAIN PLAN FOR statement for the slow query and verify whether the query optimizer is using the indices. For more details, read Section 7.1.2, “The EXPLAIN PLAN FOR Statement”.</p> <p>If the Optimizer is not choosing the optimal query execution plan, override the Optimizer decision by using optimizer hints. For more details, read Section 8.6, “Using Optimizer Hints”.</p>

Symptoms	Diagnosis	Solution
		Make sure the external sorter is enabled by defining the <i>Sorter.TmpDir</i> configuration parameter. For more details, see the description of "TmpDir_[1...N]" in <i>IBM solidDB Administration Guide</i> .
Slow response time is experienced for all queries. An increase in the number of concurrent users deteriorates the performance more than linearly. When all users are thrown out and then reconnected, performance still does not improve.	Insufficient cache size.	Increase the cache size. Allocate for cache at least 0.5MB per concurrent user or 2-5% of the database size. For more details, read the section titled "Tuning Cache" in <i>IBM solidDB Administration Guide</i> .
Slow response time is experienced for all queries and write operations. When all users are thrown out and are connected, performance only improves temporarily. The disk is very busy.	The Bonsai Tree is too large to fit into the cache.	Make sure that there are no unintentionally long-running transactions. Verify that all transactions (also read-only transactions) are committed in a timely manner. For more details, read "Reducing Bonsai Tree Size by Committing Transactions" in <i>IBM solidDB Administration Guide</i> .
Slow performance during batch write operation as the database size increases. There is an excessive amount of disk I/O.	<ul style="list-style-type: none"> <li>The data is committed to the database in batches that are too small.</li> <li>Data is written to disk in an order that is not supported by the primary key of the table.</li> </ul>	<p>Make sure that the autocommit is switched off and the write operations are committed in batches of at least 100 rows per transaction.</p> <p>Modify the primary keys or batch write processes so that write operations occur in the primary key order. For more details, read Section 8.5, "Optimizing Batch Inserts and Updates".</p>
The server process footprint grows excessively and causes the operating system to swap. The disk is very busy. The ADMIN COMMAND 'report' output shows a long list of currently active statements.	SQL statements have not been closed and dropped after use.	Make sure that the statements that are no longer in use by the client application are closed and dropped in a timely manner.

---



---

# Appendix A. Data Types

## A.1 Supported Data Types

The tables in this appendix list the supported data types by category. The following abbreviations are used in each table.

**Table A.1. Supported Data Types**

Abbreviation	Description
DEFLN	the defined length of the column; for example, for CHAR(24) the precision and length is 24
DEFPREC	the defined precision; for example, for NUMERIC(10,3) it is 10
DEFSCALE	the defined scale; for example, for NUMERIC(10,3), it is 3
MAXLEN	the maximum length of column
N/A	not applicable

### A.1.1 Character Data Types

**Table A.2. Character Data Types**

Data Type	Size	Precision	Scale	Length	Display Size
CHARACTER	2 G - 1*	DEFLN	N/A	DEFLN	DEFLN
CHAR	(2147483647)				
WCHAR	2 G - 1*	DEFLN	N/A	DEFLN	DEFLN
NATIONAL CHARACTER	(2147483647)				
NATIONAL CHAR					
NCHAR					
VARCHAR	2 G - 1**	DEFLN	N/A	DEFLN	DEFLN
CHARACTER VARYING	(2147483647)				
CHAR VARYING					

## A.1.2 Numeric Data Types

Data Type	Size	Precision	Scale	Length	Display Size
WVARCHAR NATIONAL VARCHAR NCHAR VARYING NVARCHAR	2 G - 1** (2147483647)	DEFLN	N/A	DEFLN	DEFLN
LONG VARCHAR CHARACTER LARGE OBJECT CHAR LARGE OBJECT CLOB	2 G - 1 (2147483647)	MAXLEN	N/A	MAXLEN	MAXLEN
LONG WVARCHAR LONG NATIONAL VARCHAR NCHAR LARGE OBJECT NCLOB	2 G - 1 (2147483647)	MAXLEN	N/A	MAXLEN	MAXLEN
* default is 1					
** default is 254					

## A.1.2 Numeric Data Types

**Table A.3. Numeric Data Types**

Data Type	Size	Precision	Scale	Length	Display Size
TINYINT	[-128, 255]	3	0	1 (bytes)	4 (signed) 3 (unsigned)
SMALLINT	[-32768, 65535]	5	0	2 (bytes)	6 (signed) 5 (unsigned)
INTEGER	$[-2^{31}, 2^{31} - 1]$	10	0	4 (bytes)	11 (signed)

## A.1.2 Numeric Data Types

Data Type	Size	Precision	Scale	Length	Display Size
INT					10 (unsigned)
BIGINT	$[-2^{63}, 2^{63} - 1]$	19	0	8 (bytes)	20 (signed)
REAL	Positive numbers: 1.175494351e-38 to 1.7014117e+38 Negative numbers: -1.7014117e+38 to -1.175494351e-38 You can also use value zero (0) with this data type.	7	N/A	4 (bytes)	13
FLOAT	Positive numbers: 2.2250738585072014e-308 - 8.98846567431157854e+307 Negative numbers: -8.98846567431157854e+307 to -2.2250738585072014e-308 You can also use value zero (0) with this data type.	15	N/A	8 (bytes)	22
DOUBLE PRECISION	Positive numbers: 2.2250738585072014e-308 - 8.98846567431157854e+307 Negative numbers:	15	N/A	8 (bytes)	22

Data Type	Size	Precision	Scale	Length	Display Size
	-8.98846567431157854e+307 to -2.2250738585072014e-308  You can also use value zero (0) with this data type.				
DECIMAL*	$\pm 1.0e254$	DEFPREC  Max 52 Default 52	DEFS- CALE  Default 0	2-27 (bytes)	Variable
NUMERIC	$\pm 1.0e254$	DEFPREC  Max 52 Default 52	DEFS- CALE  Default 0	2-27 (bytes)	Variable
* If neither precision nor scale is specified for DECIMAL, the values are represented as (exact) decimal floating point numbers of precision 52 and range $\pm 1.0e254$ .					



## Note

Although integer data types (TINYINT, SMALLINT, INT, and BIGINT) may be interpreted by the client program as either signed or unsigned, IBM solidDB stores and orders them as signed integers. There is no way to tell the server to order the integer data types as though they were unsigned.



## Caution

BIGINT has approximately 19 significant digits. This means that you may lose least significant digits when storing BIGINT into non-integer data types such as FLOAT (which has approximately 15 significant digits), SMALLFLOAT (which has approximately 7 significant digits), DECIMAL (which has 16 significant digits).

## A.1.3 Binary Data Types

**Table A.4. Binary Data Types**

Data Type	Size	Precision	Scale	Length	Display Size
BINARY	2 G*	DEFLEN	N/A	DEFLEN	DEFLEN x 2
VARBINARY	2 G**	DEFLEN	N/A	DEFLEN	DEFLEN x 2
LONG VARBINARY	2 G	MAXLEN	N/A	MAXLEN	MAXLEN x 2
BLOB					
* default is 1					
** default is 254					

### Tip

To insert values into BINARY, VARBINARY, and LONG VARBINARY fields, you may express the value as hexadecimal and use the CAST operator, e.g.:

```
INSERT INTO table1 VALUES (CAST('FF00AA55' AS VARBINARY));
```

Similarly, you may use CAST() expressions in WHERE clauses:

```
CREATE TABLE t1 (x VARBINARY);
INSERT INTO t1 (x) VALUES (CAST('000000A512' AS VARBINARY));
INSERT INTO t1 (x) VALUES (CAST('000000FF12' AS VARBINARY));
```

```
-- To compare the VARBINARY value(s) using LIKE, cast the
-- VARBINARY to VARCHAR.
SELECT * FROM t1 WHERE CAST(x AS VARCHAR) LIKE '000000A5%';
SELECT * FROM t1 WHERE CAST(x AS VARCHAR) LIKE '000000A5__';
```

```
-- NOTE: If you want to use "=" rather than "LIKE" then you
-- can cast either operand.
```

## A.1.4 Date Data Type

---

```
SELECT * FROM t1 WHERE CAST(x AS VARCHAR) = '000000A512';  
SELECT * FROM t1 WHERE x = CAST('000000A512' AS VARBINARY);
```

WARNING: this kind of query cannot use indexed search for the LIKE predicate and results in poor query performance in many cases.

## A.1.4 Date Data Type

**Table A.5. Date Data Type**

Data Type	Size	Precision	Scale	Length	Display Size
DATE	N/A	10*	N/A	6**	10*
* the number of characters in the yyyy-mm-dd format					
** the size of the DATE_STRUCT structure					

## A.1.5 Time Data Type

**Table A.6. Time Data Type**

Data Type	Size	Precision	Scale	Length	Display Size
TIME	N/A	8*	N/A	6**	8*
* the number of characters in the hh:mm:ss format					
** the size of the TIME_STRUCT structure					

## A.1.6 Timestamp Data Type

**Table A.7. Timestamp Data Type**

Data Type	Size	Precision	Scale	Length	Display Size
TIMESTAMP	N/A	19*	9	16**	19/29***
* the number of characters in the 'yyyy-mm-dd hh:mm:ss.ffffff' format					
** the size of the TIMESTAMP_STRUCT structure					

Data Type	Size	Precision	Scale	Length	Display Size
*** size is 29 with a decimal fraction part					

## A.1.7 The Smallest Possible Non-Zero Numbers

**Table A.8. Smallest Possible Non-Zero Numbers**

Data Type	Value
DOUBLE	2.2250738585072014e-308
REAL	1.175494351e-38

### Description of Different Column Values in the Tables

The range of a numeric column refers to the minimum and maximum values the column can store. The size of character columns refers to the maximum length of data that can be stored in the column of that data type.

The precision of a numeric column refers to the maximum number of digits used by the data type of the column. The precision of a non-numeric column refers to the defined length of the column.

The scale of a numeric column refers to the maximum number of digits to the right of the decimal point. Note that for the approximate floating point number columns, the scale is undefined, since the number of digits to the right of the decimal point is not fixed.

The length of a column is the maximum number of bytes returned to the application when data is transferred to its default C type. For character data, the length does not include the null termination byte. Note that the length of a column may differ from the number of bytes needed to store the data on the data source.

The display size of a column is the maximum number of bytes needed to display data in character form.

## A.1.8 BLOBs and CLOBs

IBM solidDB can store binary and character data up to 2147483647 (2G - 1) bytes long. When such data exceeds a certain length, the data is called a BLOB (Binary Large Object) or CLOB (Character Large Object), depending upon the data type that stores the information. CLOBs contain only "plain text" and can be stored in any of the following data types:

CHAR, WCHAR

VARCHAR, WVARCHAR

LONG VARCHAR (mapped to standard type CLOB),

LONG WVARCHAR (mapped to standard type NCLOB)

BLOBs can store any type of data that can be represented as a sequence of bytes, such as a digitized picture, video, audio, a formatted text document. (They can also store plain text, of course, but you'll have more flexibility if you store plain text in CLOBs). BLOBs are stored in any of the following data types:

BINARY

VARBINARY

LONG VARBINARY (mapped to standard type BLOB)

Since character data is of course a sequence of bytes, character data can be stored in BINARY fields, as well as in CHAR fields. CLOBs can be considered a subset of BLOBs.

For convenience, we will use the term BLOBs to refer to both CLOBs and BLOBs.

For most non-BLOB data types, such as integer, float, date, etc., there is a rich set of valid operations that you can do on that data type. For example, you can add, subtract, multiply, divide, and do other operations with FLOAT values. Because a BLOB is a sequence of bytes and the database server does not know the "meaning" of that sequence of bytes (i.e. it doesn't know whether the bytes represent a movie, a song, or the design of the space shuttle), the operations that you can do on BLOBs are very limited.

IBM solidDB does allow you to perform some string operations on CLOBs. For example, you can search for a particular substring (e.g. a person's name) inside a CLOB by using the `LOCATE ( )` function. Because such operations require a lot of the server's resources (memory and/or CPU time), IBM solidDB allows you to limit the number of bytes of the CLOB that are processed. For example, you might specify that only the first 1 megabyte of each CLOB be searched when doing a string search. For more information, see the description of the *MaxBlobExpressionSize* configuration parameter in *IBM solidDB Administration Guide*.

Although it is theoretically possible to store the entire blob "inside" a typical table, if the blob is large, then the server usually performs better if most or all of the blob is not stored in the table. In IBM solidDB, if a blob is no more than N bytes long, then the blob is stored in the table. If the blob is longer than N bytes, then the first N bytes are stored in the table, and the rest of the blob is stored outside the table as disk blocks in the physical database file. The exact value of "N" depends in part upon the structure of the table, the disk page size that you specified when you created the database, etc., but is always at least 256. (Data 256 bytes or shorter is always stored in the table.)

If a data row size is larger than one third of the disk block size of the database file, you must store it partly as a BLOB.



The `SYS_BLOBS` system table is used as a directory for all BLOB data in the physical database file. One `SYS_BLOB` entry can accommodate 50 BLOB parts. If the BLOB size exceeds 50 parts, several `SYS_BLOB` entries per BLOB are needed.

The query below returns an estimate on the total size of BLOBs in the database.

```
select sum(totalsize) from sys_blobs
```

The estimate is not accurate, because the info is only maintained at checkpoints. After two empty checkpoints, this query should return an accurate response.

---

---

# Appendix B. IBM solidDB SQL Syntax

This appendix presents a simplified description of the SQL statements, including some examples.

Note that earlier versions of this manual put the sync-related SQL commands in a separate chapter. This version of the manual puts all the SQL commands into this one appendix.

IBM solidDB SQL syntax is based on the ANSI X3H2-1989 level 2 standard including important ANSI X3H2-1992 (SQL-92) extensions. User and role management services missing from previous standards are based on the ANSI SQL-99 draft.

Most commands listed here are available in IBM solidDB disk-based engine and IBM solidDB main memory engine. Some commands related to advanced replication synchronization are not available if you have not licensed advanced replication.

## B.1 ADMIN COMMAND

```
ADMIN COMMAND 'command_name'
```

```
command_name ::= ABORT | ASSETEXIT | BACKUP |  
BACKGROUNDJOB | BACKUPLIST | CHECKPOINTING | CLEANBGJOBINFO |  
CLOSE | DESCRIBE | ERRORCODE | ERROREXIT | FILESPEC |  
HELP | HOTSTANDBY | INFO | MAKECP | MEMORY | MESSAGES |  
MONITOR | NETBACKUP | NETBACKUPLIST | NETSTAT | NOTIFY |  
OPEN | PARAMETER | PERFMON | PID | PROCTRACE |  
PROTOCOLS | REPORT | RUNMERGE | SAVE | SHUTDOWN |  
SOLCONNECTOR PROPAGATOR SHUTDOWN | SQLLIST | STARTMERGE |  
STATUS | THROWOUT | TID | TRACE | USERID | USERLIST |  
USERTRACE | VERSION
```

### B.1.1 Supported in

ADMIN COMMAND syntax is supported in all IBM solidDB editions.

### B.1.2 Usage

This SQL extension executes administrative commands. The *command\_name* in the syntax is a IBM solidDB SQL Editor (solsql) command string, for example:

ADMIN COMMAND 'backup'

If you are entering these commands using IBM solidDB Remote Control (solcon), be sure to specify the syntax with command name only (without the quotes), for example:

backup

Abbreviations for ADMIN COMMANDs are also available, for example,

ADMIN COMMAND 'bak'

To access a list of abbreviated commands, execute

ADMIN COMMAND 'help'

The result set contains two columns: RC INTEGER and TEXT VARCHAR(254). Integer column RC is a command return code (0 if success), and varchar column TEXT is the command reply.

Note that all options of the ADMIN COMMAND are not transactional and cannot be rolled back.



## Caution

### *ADMIN COMMANDS and Starting Transactions*

Although ADMIN COMMANDs are not transactional, they will start a new transaction if one is not already open. (They do not commit or roll back any open transaction.) This effect is usually insignificant. However, it may affect the 'start time' of a transaction, and that may occasionally have unexpected effects. IBM solidDB's concurrency control is based on a versioning system; you see a database as it was at the time that your transaction started. (See the section of *IBM solidDB Administration Guide* titled 'IBM solidDB Bonsai Tree Multiversioning and Concurrency Control'). So, for example, if you: commit work, and issue an ADMIN COMMAND without doing another commit, and go to lunch and return an hour later, then your next SQL command may see the database as it was an hour ago, i.e. when you first started the transaction with the ADMIN COMMAND.



## Caution


Error codes in ADMIN COMMANDS ADMIN COMMANDS return an error only if the command syntax or parameter values are incorrect. That is, if only the requested operation may be started, the command returns SQLSUCCESS (0). The outcome of the operation itself is written into a result set. The result set has two columns: TC and TEXT. The RC (return code) column contains the return code

of the operation: it is "0" for success, and different numeric values for errors. It is thus necessary to check both the codes (of the ADMIN COMMAND statement and of the operation).

Following is a description of the syntax for each ADMIN COMMAND command option:

**Table B.1. ADMIN COMMAND Syntax**

Option Syntax	Description
ADMIN COMMAND 'abort [backup   netbackup]'	Aborts the active local or network backup process. The backup operation is not guaranteed to be atomic, therefore the cancelled operation may produce an incomplete backup file to the backup directory until the next backup takes place.  If the options are not entered, the default behaviour is similar to command ADMIN COMMAND 'abort backup'.
ADMIN COMMAND 'assertexit' Abbreviation: asex	Asserts the server.
ADMIN COMMAND 'backgroundjob' [LIST [-1] [user]]   [ABORT {jobid   user   ALL }]   [DELETE ERRORINFO {jobid   user   ALL }]'  user ::= USER {username userid}  Abbreviation: bgjob	Lists and possibly aborts running background jobs, that is, SQL statements that have been started by using the START AFTER COMMIT (SAC) statement.  The LIST option lists all user jobs that are running or, alternatively, only those of a specified user. The -1 option refers to a long list (such as AC 'userlist -1').  The ABORT option aborts either jobs by job identification number or all jobs by user identification number. If you give the ABORT without arguments, it aborts all jobs from all users.  The DELETE ERRORINFO option deletes error information from the SYS_BACKGROUNDJOB_INFO system table, where the errors encountered by background jobs are stored. This option performs the same operation as the deprecated ADMIN COMMAND 'CLEANBGJOBINFO' command.
ADMIN COMMAND 'backup [-s] [backup_directory]'	Makes a backup of the database. The operation can be performed as a synchronized or an asynchronous (default) manner. The synchronized operation is specified by using the optional -s parameter.

Option Syntax	Description
Abbreviation: bak	The default backup directory is the one defined in the <i>[General]</i> section of the configuration parameter <i>BackupDirectory</i> . The backup directory may also be given as an argument. For example, backup abc creates a backup in directory 'abc'. All directory definitions are relative to the IBM solidDB working directory.
ADMIN COMMAND 'backuplist' Abbreviation: bls	Displays a status list of last local backups.
ADMIN COMMAND 'cleanbgjobinfo' Abbreviation: cleanbgi	 <p><b>Note</b></p> <p>This command has been deprecated. See the <b>backgroundjob</b> command for more information.</p> <p>Cleans the table SYS_BACKGROUNDJOB_INFO containing status data of background procedures.</p>
ADMIN COMMAND 'checkpointing' Abbreviation: cp	Turns on/off checkpointing.
ADMIN COMMAND 'close' Abbreviation: clo	Closes the server to new connections; no new connections are allowed.
ADMIN COMMAND 'describe parameter <i>param</i> ' Abbreviation: des	<p>Returns a description of the specified parameter.</p> <p>Note that the param should be in the form <i>section_name.param_name</i>. The section and parameter names are case-insensitive.</p> <p>The following example describes parameter <i>Com.Trace</i> = y/n</p>

Option Syntax	Description
	ADMIN COMMAND 'des parameter com.trace'
ADMIN COMMAND 'errorcode {all   <i>SOLID_error_code</i> }' Abbreviation: ec	Displays a description of an error code (or all codes). Give the code number as an argument, for example, <b>errorcode 10033</b> .
ADMIN COMMAND 'errorexit <number>' Abbreviation: erex	Forces the server into an immediate process exit with the given process exit code.
ADMIN COMMAND 'filespec' Abbreviation: fs	Displays database file specifications, current fill ratios and current file sizes.
ADMIN COMMAND 'help' Abbreviation: ?	Displays available commands.
ADMIN COMMAND 'hotstandby [option]' Abbreviation: hsb	A HotStandby command. For list of options see <i>IBM solidDB High Availability User Guide</i> .
ADMIN COMMAND 'info options' Abbreviation: info	Returns server information. The server information consists of 25 rows of data. The information displayed does not explain the meaning of the values. However, by using the list below, you can find out the meaning of each value. The 25 values are, from top to down: <ul style="list-style-type: none"> <li>• numusers - Number of current users.</li> </ul>

Option Syntax	Description
	<ul style="list-style-type: none"> <li>• <code>maxusers</code> - Maximum number of users.</li> <li>• <code>sernum</code> - Server serial number.</li> <li>• <code>dbsize</code> - Database size.</li> <li>• <code>logsize</code> - Size of log files.</li> <li>• <code>uptime</code> - Server up since.</li> <li>• <code>bcktime</code> - Timestamp of last successfully completed local backup.</li> <li>• <code>cptime</code> - Timestamp of last successfully completed checkpoint.</li> <li>• <code>tracestate</code> - Current trace state.</li> <li>• <code>monitorstate</code> - Current monitor state, which is the number of users who have SQL monitoring currently enabled; this value is -1 if all users have SQL monitoring enabled. Note that SQL monitoring is enabled using the ADMIN COMMAND 'monitor {on   off} [user {username   userid}]' (described below).</li> <li>• <code>openstate</code> - Current open or close state — that is, whether the database server accepts new connections or not. "open" means that the database server accepts new connections.</li> <li>• <code>nummerges</code> - Number of merges.</li> <li>• <code>numlocks</code> - Number of locks.</li> <li>• <code>numcursors</code> - Number of open cursors.</li> <li>• <code>numtransactions</code> - Number of open transactions.</li> <li>• <code>memtotal</code> - Total amount of memory allocated bytes.</li> <li>• <code>dbfreesize</code> - Amount of free space remaining in database.</li> </ul>



Option Syntax	Description
	<ul style="list-style-type: none"> <li>• <code>dbpagesize</code> - Database page size.</li> <li>• <code>imdbsize</code> - Amount of space used by in-memory tables (including Temporary Tables and Transient Tables) and the indexes on those tables. The return value is in kilobytes (KB) and is in the form of a VARCHAR.</li> <li>• <code>name</code> - Prints out the server name.</li> <li>• <code>primarystarttime</code> - The time the Primary role has started.</li> <li>• <code>secondarystarttime</code> - The time the Secondary role has started.</li> <li>• <code>dbconfigsize</code> - The configured database size.</li> <li>• <code>dbcreatetime</code> - This option prints out the database creation timestamp. The abbreviation <code>dbcreationtime</code> can also be used.</li> <li>• <code>processsize</code> - This option prints out the system-level virtual process size in kilobytes. The abbreviation <code>psize</code> can also be used.</li> </ul> <p>More than one option can be used per command. Values are returned in the same order as requested, one row for each value.</p> <p>Example command:</p> <p><b>ADMIN COMMAND 'info dbsize logsize'</b></p> <p>Example output:</p> <pre>RC TEXT 0      851968 0      573440</pre>

Option Syntax	Description
ADMIN COMMAND 'makecp [-s]' Abbreviation: mcp	<p>Makes a checkpoint. Requires SYS_ADMIN_ROLE privilege.</p> <p>By default, the checkpoint is asynchronous. With the option <code>-s</code>, the command returns only after the checkpoint has completed.</p>
ADMIN COMMAND 'memory' Abbreviation: mem	<p>Returns the server process memory size. The reported process memory size can differ from the process size reported by your operating system.</p>
ADMIN COMMAND 'messages [{ warnings   errors}] [count]' Abbreviation: mes	<p>Displays server messages. Optional severity and message numbers can also be defined. For example:</p> <p>ADMIN COMMAND 'messages warnings 100' displays last 100 warnings.</p>
ADMIN COMMAND 'monitor {on   off} [ user {username   userid}]' Abbreviation: mon	<p>Sets server monitoring on and off. Monitoring logs user activity and SQL calls to <code>soltrace.out</code> file.</p>
ADMIN COMMAND 'netbackup [options] [DELETE_LOGS   KEEP_LOGS] [connect connect str] [dir backup dir]' Abbreviation: nbak	<p>Makes a network backup of the database. The operation can be performed as a synchronized or an asynchronous (default) manner. The synchronized operation is specified by using the optional <code>-s</code> parameter.</p> <p>If you use the <code>DELETE_LOGS</code> parameter, backed-up log files in the source server are deleted. This is sometimes referred to as Full backup. This is the default value. On the other hand, if you use the <code>KEEP_LOGS</code> parameter, backed-up log files are kept in the source server. This is sometimes referred to as Copy backup. Using the keyword <code>KEEP_LOGS</code> corresponds to setting the <i>General</i> parameter <code>Netbackup-DeleteLog</code> to "no".</p>

Option Syntax	Description
	<p>The default connect string and the default netbackup directory are defined in the <i>NetBackupConnect</i> and in the <i>NetBackupDirectory</i> parameters in the [General] section of the configuration file.</p> <p>Options that are entered with the netbackup command override the values specified in the configuration file. Directory definitions are relative to the IBM solidDB working directory.</p>
<p>ADMIN COMMAND 'netbackuplist' Abbreviation: nbls</p>	<p>Displays a status list of the most recently made network backups of the database server.</p>
<p>ADMIN COMMAND 'netstat'</p>	<p>Displays server settings and the network status.</p>
<p>ADMIN COMMAND 'notify user {username   user id   ALL } message' Abbreviation: not</p>	<p>This command sends an event to a given user with event identifier NOTIFY. This identifier is used to cancel an event-waiting thread when the statement timeout is not long enough for a disconnect or to change the event registration.</p> <p>The following example sends a notify message to a user with user id 5 ; the event then gets the value of the message parameter.</p> <p><b>ADMIN COMMAND 'notify user 5 Canceled by admin'</b></p>
<p>ADMIN COMMAND 'open' Abbreviation: ope</p>	<p>Opens server for new connections; new connections are allowed.</p>
<p>ADMIN COMMAND 'parameter [option][name[= [* value][temporary]]' Abbreviation: par</p>	<p>Displays and sets server parameter values. If you run the command without a specified value, the parameter will be set to its startup value. If you assign a parameter value with an asterisk (*), the parameter will be set to its factory value. The "name" may be either a section name, or it may be a parameter name prefaced by a section name and period (e.g. "com.trace"). For example:</p>

Option Syntax	Description
	<ul style="list-style-type: none"> <li>• <i>parameter</i> used alone displays all parameters.</li> <li>• <i>parameter general</i> displays all parameters from section [<i>General</i>].</li> <li>• <i>parameter general.readonly</i> displays a single parameter named <i>readonly</i> from section [<i>General</i>]. You must place a period between the section name (<i>[General]</i>) and the parameter name (<i>readonly</i>).</li> <li>• <i>parameter com.trace=yes</i> sets communication trace on. You must place a period between the section name (e.g. [<i>Com</i>]) and the parameter name (e.g. <i>trace</i>). You should not put blanks around the equals sign.</li> <li>• <i>parameter com.trace=</i> sets communication trace to its startup value.</li> <li>• <i>parameter com.trace=*</i> sets communication trace to its factory value.</li> </ul> <p>The output may contain three values, as shown below:</p> <pre>0 Logging DurabilityLevel 1 2 3</pre> <p>The three values represent the following:</p> <ul style="list-style-type: none"> <li>• 1 is the current value (may be set dynamically)</li> <li>• 2 is the value in the INI file (startup value)</li> <li>• 3 is the factory value</li> </ul> <p>If the <i>-r</i> option is used, then only the current parameter values are returned.</p>
<pre>ADMIN COMMAND 'perfmon [- c   - r] [options] [diff [ start   stop ]</pre>	<p>Returns server performance counters. The options are:</p> <ul style="list-style-type: none"> <li>• <b>-c</b> - prints actual counter values. If this option is not provided, the output numbers are operations/second where applicable.</li> </ul>

Option Syntax	Description
<pre>[filename interval] [ name_prefix_list]'</pre> <p>Abbreviation: pmon</p>	<ul style="list-style-type: none"> <li>• <b>-r</b> - prints in raw mode, which includes only the latest counter values without any formatting. No option names or additional information is printed. This option is useful if actual monitoring is performed using some other external program that retrieves the counter values from the server.</li> <li>• <b>-xtime</b> - prints the time in seconds</li> <li>• <b>-xtimediff</b> - prints the difference to the last pmon call in milliseconds</li> <li>• <b>-xnames</b> - prints out the column names for the output</li> <li>• <b>-xdiff</b> - indicates the difference to the last perfmon execution instead of the absolute value</li> <li>• <b>diff</b> - starts a server task that prints out all perfmon counters with specified intervals to a file. The interval must be given in milliseconds. The output file is written using "comma-separated values", with the first row including counter names. The file, as it is, can be processed by spreadsheet programs like Excel.</li> <li>• <i>name_prefix_list</i> - limits output to specific counter names. For example, to print all file related counters, the <i>name_prefix_list</i> should be file. You can also specify multiple prefixes.</li> </ul> <p>The following example returns all information:</p> <p><b>ADMIN COMMAND 'perfmon'</b></p> <p>The following example returns all values whose name starts with prefix file and cache as counters.</p> <p><b>ADMIN COMMAND 'perfmon-c file cache'</b></p> <p>Note that the prefix file and cache are matched to those counter names that are in the perfmon output.</p>

Option Syntax	Description
	<p>The following command example starts a diff task that writes to <code>myd.csv</code> file on 1000 milliseconds interval:</p> <p><b>ADMIN COMMAND 'pmon diff start myd.csv 1000'</b></p> <p>For sample output, and description of counters, see the section of <i>IBM solidDB Administration Guide</i> titled "Detailed DBMS Monitoring and Troubleshooting".</p>
<pre>ADMIN COMMAND 'pid' Abbreviation: pid</pre>	<p>Returns server process id.</p>
<pre>ADMIN COMMAND 'proctrace { on   off } user username { procedure   trigger   table } entity_name' Abbreviation: ptrc</pre>	<p>This turns on tracing in stored procedures and triggers.</p> <p>The "username" is the name of the user whose procedure calls (or triggers) you want to trace. If multiple connections are using the same username, then calls from all of those connections will be traced. Furthermore, if you are using advanced replication, the tracing will be done not only for calls on the replica, but also calls that are propagated to the master and then executed on the master.</p> <p>The "entity_name" is the name of the procedure, trigger, or table for which you want to turn tracing on or off. If you specify a procedure or trigger name, then it will generate output for every statement in the specified procedure or trigger. If you specify a table name, then it will generate output for all triggers on that table. Trace is activated only when the specified username calls the procedure / trigger.</p> <p>For more detail about proctrace, see "Tracing Facilities For Stored Procedures And Triggers" in <i>IBM solidDB SQL Guide</i>.</p> <p>See also ADMIN COMMAND 'ustrace'.</p>
<pre>ADMIN COMMAND 'protocols' Abbreviation: prot</pre>	<p>Returns a list of available communication protocols, one row for each protocol.</p> <p>Example:</p>

Option Syntax	Description
	<b>ADMIN COMMAND 'protocols'</b>
ADMIN COMMAND 'report <i>filename</i> ' Abbreviation: rep	Generates a report of server information to a file given as an argument.
ADMIN COMMAND 'runmerge' Abbreviation: rm	Runs an index merge.
ADMIN COMMAND 'save parameters [ <i>filename</i> ]' Abbreviation: save	Saves the set of current configuration parameter values to a file. If no file name is given, the default <code>solid.ini</code> file is rewritten. This operation is performed implicitly at each checkpoint.
ADMIN COMMAND 'shutdown [ <i>force</i> ]' Abbreviation: sd	Stops IBM solidDB.  If the <i>force</i> option is used, the active transactions are aborted and the users are disconnected forcefully.
ADMIN COMMAND 'solconnector propagator shutdown [ <i>all</i>   <i>partition-id</i> ]'	When you issue this command, either one partition replica-specific connector instance or all of them are shut down. Before shutting down, each connector writes and commits all the transactions that have been retrieved in the back-end database.  In the command, <i>partition-id</i> refers to the partition name stored in the replication model.  If the connector runs in the loader role, the command has no effect.
ADMIN COMMAND 'sqlist <i>top number_of_statements</i> '	This command prints out a list of the longest running SQL statements among the currently running statements. The list contains the selected number of statements.

Option Syntax	Description
ADMIN COMMAND 'status' Abbreviation: sta	Displays server statistics.
ADMIN COMMAND 'status backup   netbackup' Abbreviation: sta backup   netbackup	Displays status of the last started local or network backup. The status can be one of the following: <ul style="list-style-type: none"> <li>• If the last backup was successful or no backups have been requested, the output is 0 SUCCESS.</li> <li>• If the backup is in process (for example, started but not ready yet), then the output is 14003 ACTIVE.</li> <li>• If the last backup failed, the output is: <i>errorcode</i> ERROR where the <i>errorcode</i> shows the reason for the failure</li> </ul>
ADMIN COMMAND 'startmerge' Abbreviation: sm	Starts and waits for completion of merge.
ADMIN COMMAND 'throwout {username   userid   all}' Abbreviation: to	Exits users from IBM solidDB. To exit a specified user, give the user id as an argument. To throw out all users, use the keyword ALL as an argument.
ADMIN COMMAND 'tid' Abbreviation: tid	This command returns the ID (a 4-digit code) of the current user thread (in the server).
ADMIN COMMAND 'trace { on   off} sql   rpc   sync   info <level>	Sets server trace on or off. The tracing options are: <ul style="list-style-type: none"> <li>• sql - SQL messages</li> <li>• rpc - Network communications</li> </ul>



Option Syntax	Description
<pre>flowplans   all'</pre> Abbreviation: tra	<ul style="list-style-type: none"> <li>• sync - Synchronization messages</li> <li>• info &lt;level&gt; - SQL execution trace (level is 0...8)</li> <li>• flowplans - plans of Flow SQL statements</li> </ul> <p>If no options are specified, or all is specified, both SQL messages and network communications messages are written to the trace file. The name of the default trace file is <code>sol-trace.out</code>.</p>
ADMIN COMMAND 'userid' Abbreviation: uid	<p>Returns the user identification number of the current connection.</p> <p>Example:</p> <pre>ADMIN COMMAND 'userid'</pre>
ADMIN COMMAND 'userlist [-l] [name   id]' Abbreviation: ul	<p>This command displays a list of users currently logged in to the database, together with a number of primary attributes. These attributes are: User name, User Id, Type, Machine Id, Login time and Appinfo (optional). For attribute descriptions, see the detailed output description below.</p> <p>Option <code>-l</code> (long) displays a more detailed output. The fields in the long output are:</p> <ul style="list-style-type: none"> <li>• <i>Id</i> - The user session identification number within the database. The lifetime of an Id is that of the user session. After a user logs out, the number may be reused.</li> <li>• <i>Type</i> - Client type. Possible values are:             <ul style="list-style-type: none"> <li>• <i>Java</i>, which refers to a client using JDBC</li> <li>• <i>ODBC</i>, which refers to a client using ODBC</li> <li>• <i>SQL</i>, which refers to IBM solidDB SolSql editor</li> </ul> </li> </ul>

Option Syntax	Description
	<ul style="list-style-type: none"> <li>• <i>Machine</i> - The client computer name (host name) and its IP address, if available.</li> <li>• <i>Login tile</i> - The client computer login timestamp.</li> <li>• <i>Appinfo</i> - The value of the client computer's environmental variable <i>SOLAPPINFO</i>, if the client is using ODBC. In the case of JDBC, the Java utility property <i>solid_appinfo</i> has to be set to that value, for it to be visible in the output. Alternatively, the following Java command line may be used to pass the value of the environmental variable to the driver:   <b>java -Dsolid_appinfo=%SOLAPPINFO% java program name</b> <p>Note: the value of SOLAPPINFO must not contain blanks.</p> </li> <li>• <i>Last activity</i> - The time when the client last time sent a request to the server.</li> <li>• <i>Autocommit</i> - If the autocommit mode is switched off (value 0), the current transaction is open until a COMMIT or ROLLBACK statement is issued. After that, a new statement starts a new transaction.   If the autocommit mode is switched on (value 1), each statement is automatically committed. </li> <li>• <i>RPC compression</i> - Indicates whether the data transmission compression is on or off.</li> <li>• <i>Transparent failover</i> - This field indicates if Transparent Failover (TF) is in use. Transparent failover is a characteristic of the HotStandby configuration. It hides the server role change from the user. Because IBM solidDB Tools do not support TF, you will only see a "no" value in this field.</li> </ul>

Option Syntax	Description
	<ul style="list-style-type: none"> <li data-bbox="692 244 1329 335">• <i>Transparent cluster</i> - Transparent cluster indicates whether the load balancing feature (in HSB) is enabled for this connection or not.</li> <li data-bbox="692 366 1329 491">• <i>Transaction active</i> - This field indicates whether there is an open, uncommitted transaction on the connections (value 1) or not (value 0). When the connection is set for Autocommit, the value is, most of the time, 0.</li> <li data-bbox="692 522 1329 612">• <i>Transaction duration</i> - This field indicates the duration of the currently open transaction. After COMMIT or ROLLBACK, the value becomes 0.</li> <li data-bbox="692 644 1329 769">• <i>Transaction isolation</i> - This field indicates the transaction isolation level for the transactions. The isolation level decides how data which is a part of an ongoing transaction is made visible to other transactions.</li> <li data-bbox="692 800 1329 890">• <i>Transaction durability</i> - This field indicates the durability of the currently open transaction. By default, IBM solidDB uses <i>adaptive</i> durability.</li> <li data-bbox="692 921 1329 1046">• <i>Transaction safeness</i> - This field indicates the safeness of the currently open transaction. Safeness is set by using the <i>SafenessLevel</i> parameter. By default, IBM solidDB uses the <i>2safe</i> transaction safeness.</li> <li data-bbox="692 1078 1329 1394">• <i>Transaction autocommit</i> - This field indicates whether the currently open transaction is automatically committed. If the transaction autocommit for the current transaction is switched off (value 0), the current transaction is open until a COMMIT or ROLLBACK statement is issued. After that, a new statement starts a new transaction.  If the autocommit mode is switched on for the current transaction (value 1), each statement is automatically committed.</li> <li data-bbox="692 1425 1329 1454">• <i>Current schema</i> - Indicates the current schema name.</li> <li data-bbox="692 1486 1329 1515">• <i>Current catalog</i> - Indicates the current catalog name.</li> </ul>

Option Syntax	Description
	<ul style="list-style-type: none"> <li data-bbox="694 248 1333 369">• <i>Sortgroubby</i> - Indicates how the GROUP BY statement is performed if explicit information on the number of result groups is not available. There are two possible values: <ul style="list-style-type: none"> <li data-bbox="729 401 1333 522">• ADAPTIVE - GROUP BY input is pre-sorted if the real number of result groups exceeds the number of rows that fit into the central memory array for GROUP BY.</li> <li data-bbox="729 557 1333 644">• STATIC - GROUP BY input is pre-sorted whenever there are at least two items in the GROUP BY list. Otherwise, the GROUP BY input is not pre-sorted.</li> </ul> </li> <li data-bbox="694 678 1333 765">• <i>Simple optimizer rules</i> - Indicates the value of the <code>solid.ini</code> SQL parameter <code>SimpleOptimizerRules</code>. Possible values are Yes/No/Default.</li> <li data-bbox="694 800 1333 956">• <i>Statement max time</i> - Indicates the connection-specific statement maximum execution time in seconds. This setting is effective until a new maximum time is given. Zero time indicates that there is no maximum time. This is the default value.</li> <li data-bbox="694 991 1333 1043">• <i>Lock timeout</i> - Indicates the timeout set by using the SET LOCK TIMEOUT statement.</li> <li data-bbox="694 1078 1333 1130">• <i>Optimistic lock timeout</i> - Indicates the timeout set by using the SET OPTIMISTIC LOCK TIMEOUT statement.</li> <li data-bbox="694 1164 1333 1216">• <i>Idle timeout</i> - Indicates the timeout set by using the SET IDLE TIMEOUT statement.</li> <li data-bbox="694 1251 1333 1303">• <i>Join Path Span</i> - Indicates the join path span value set by using the SET SQL JOINPATHSPAN statement.</li> <li data-bbox="694 1338 1333 1373">• <i>RPC seqno</i> - Internal protocol message sequence number.</li> <li data-bbox="694 1407 1333 1459">• <i>SQL sortarray</i> - The size of user-specific internal sort array.</li> </ul>

Option Syntax	Description
	<ul style="list-style-type: none"> <li>• <i>SQL unionsfromors</i> - The value tells how many (at most) OR operators may be converted to UNIONS. Unions are faster but require more memory to execute.</li> <li>• <i>EVENT QUEUE LENGTH</i> - Indicates the number of posted events in the event queue.</li> <li>• <i>Stmt id</i> - The current statement identification number. The numbers are session specific and they are assigned for each different statement</li> <li>• <i>Stmt state</i> - An internal statement execution state.</li> <li>• <i>Stmt rowcount</i> - The number of rows retrieved or inserted in the current statement.</li> <li>• <i>Stmt starttime</i> - The current statement start date and time.</li> <li>• <i>Stmt duration</i> - Internal statement duration in seconds. Note: this value has no relevance to the externally visible statement latency. Typically, the statement duration is much longer than latency.</li> <li>• <i>Stmt SQL str</i> - The current statement string.</li> </ul>
<pre>ADMIN COMMAND 'usertrace { on   off } user username { procedure   trigger   table } entity_name' Abbreviation: utrc</pre>	<p>This turns on user tracing in stored procedures and triggers. This command will generate output for every WRITETRACE statement in the specified procedure or trigger.</p> <p>The "username" is the name of the user whose procedure calls (or triggers) you want to trace. If multiple connections are using the same username, then calls from all of those connections will be traced. Furthermore, if you are using advanced replication, the tracing will be done not only for calls on the replica, but also calls that are propagated to the master and then executed on the master.</p> <p>The "entity_name" is the name of the procedure, trigger, or table for which you want to turn tracing on or off. If you specify a table name, then it will generate output for all trig-</p>

Option Syntax	Description
	<p>gers on that table. Trace is activated only when the specified user calls the procedure / trigger.</p> <p>For more detail about proctrace, see "Tracing Facilities For Stored Procedures And Triggers" in IBM solidDB SQL Guide.</p> <p>See also the discussion of "proctrace" on page D-10.</p>
<p>ADMIN COMMAND 'version' Abbreviation: ver</p>	<p>Displays server version information and information related to the IBM solidDB software licence in use.</p>

## B.2 ADMIN EVENT

```
ADMIN EVENT 'command'
command_name ::=
    REGISTER { event_name [ , event_name ... ] | ALL } |
    UNREGISTER { event_name [ , event_name ... ] | ALL } |
    WAIT
event_name ::= the name of a system event
```

### B.2.1 Usage

This is a IBM solidDB-specific extension to SQL that allows you to register for and wait for system-generated events without writing and calling a stored procedure.

You must explicitly register for and wait for the event. For example

```
ADMIN EVENT 'register sys_event_hsbstateswitch';
ADMIN EVENT 'wait';
```

After the event is posted by the system, you will see something similar to the following:

```
ENAME                POSTSRVTIME          UID NUMDATAINFO TEXTDATA
-----                -
SYS_EVENT_HSBSTATESWITCH 2003-10-28 18:10:14 -1 NULL          PRIMARY ACTIVE
```

1 rows fetched.

You must register for the event before you wait for it. (This is different from the way that WAIT works in stored procedures. In stored procedures, explicit registration is optional.)



## Note

You cannot register to synchronization events (starting with "SYNC\_") with this command. You may use the procedure language command WAIT EVENT for that purpose.

Once the connection starts to wait for an event, the connection will not be able to do anything else until the event is posted.

You may register for multiple events. When you wait, you cannot specify which type of event to wait for. The wait will continue until you have received any of the events for which you have registered.

You may only wait for system events, not user events, using ADMIN EVENT. If you want to wait for user events, then you must write and call a stored procedure.

The ADMIN EVENT command does not provide an option to post an event.

To use ADMIN EVENT, you must have DBA privileges or be granted the role SYS\_ADMIN\_ROLE.

## B.2.2 Examples

```
ADMIN EVENT 'register sys_event_hsbstateswitch';
ADMIN EVENT 'wait';
ADMIN EVENT 'unregister sys_event_hsbstateswitch';
```

## B.3 ALTER TABLE

```
ALTER TABLE base_table_name
{
  ADD [COLUMN] column_identifier data_type
  [DEFAULT literal | NULL] [NOT NULL] |
  ADD CONSTRAINT constraint_name dynamic_table_constraint |
  DROP CONSTRAINT constraint_name |
  ALTER [ COLUMN ] column_name
  {DROP DEFAULT | {SET DEFAULT literal | NULL} } |
```

```

{{ADD | DROP} NOT NULL }
  DROP [COLUMN] column_identifier |
  RENAME [COLUMN]
    column_identifier column_identifier |
  MODIFY [COLUMN] column_identifier data-type |
MODIFY SCHEMA schema_name} |
SET HISTORY COLUMNS (c1, c2, c3) |
SET {OPTIMISTIC | PESSIMISTIC} |
SET STORE {DISK | MEMORY} |
SET [NO]SYNCHHISTORY |
  SET TABLE NAME new_base_table_name
}
dynamic_table_constraint::=
{FOREIGN KEY (column_identifier [, column_identifier] ...)
REFERENCES table_name [(column_identifier [, column_identifier] ) ...]}
[referential_triggered_action] |
CHECK (check_condition) | UNIQUE (column_identifier)
referential_triggered_action::=
  ON {UPDATE | DELETE} {CASCADE | SET NULL | SET DEFAULT |
  RESTRICT |NO ACTION}

```

## B.3.1 Usage

The structure of a table may be modified through the ALTER TABLE statement. Columns may be added, removed, modified, or renamed. You may change whether the table uses optimistic or pessimistic concurrency control. You may change whether the table is stored in memory or on disk. You may change which schema the table is part of.

The server allows users to change the width of a column using the ALTER TABLE command. A column width can be increased at any time (that is, whether a table is empty [no rows] or non-empty). However, the ALTER TABLE command disallows decreasing the column width when the table is non-empty; a table must be empty to decrease the column width.

Note that a column cannot be dropped if it is part of a unique or primary key.

The owner of a table can be changed using the ALTER TABLE *base\_table\_name* MODIFY SCHEMA *schema\_name* statement. This statement gives all rights, including creator rights, to the new owner. The old owner's access rights to the table, excluding the creator rights, are preserved.

For information about the SET HISTORY COLUMNS clause, see Section B.4, “ALTER TABLE ... SET HISTORY COLUMNS”.



For information about the SET [NO]SYNCHISTORY clause, see Section B.5, “ALTER TABLE ... SET SYNCHISTORY”.

Individual tables can be set to optimistic or pessimistic with the statement ALTER TABLE *base\_table\_name* SET {OPTIMISTIC | PESSIMISTIC}. By default, all tables are optimistic. A database-wide default can be set in the *General* section of the configuration file with the parameter *Pessimistic* = *yes*.

A table may be changed from disk-based to in-memory or vice-versa. (This is only allowed with IBM solidDB main memory engine.) This may be done only if the table is empty. If you try to change a table to the same storage mode that it already uses (e.g. if you try to change an in-memory table to use in-memory storage), then the command has no effect, and no error message is issued.

### B.3.2 Example

```
ALTER TABLE table1 ADD x INTEGER;
ALTER TABLE table1 RENAME COLUMN old_name new_name;
ALTER TABLE table1 MODIFY COLUMN xyz SMALLINT;
ALTER TABLE table1 DROP COLUMN xyz;
ALTER TABLE table1 SET STORE MEMORY;
ALTER TABLE table1 SET PESSIMISTIC;
ALTER TABLE table2 ADD COLUMN col_new CHAR(8) DEFAULT 'VACANT' NOT NULL;
ALTER TABLE table2 ALTER COLUMN col_new SET DEFAULT 'EMPTY';
ALTER TABLE table2 ALTER COLUMN col_new DROP DEFAULT;
ALTER TABLE dept_tab1 ADD CONSTRAINT div_check CHECK(division_id < 12);
ALTER TABLE dept_tab1 DROP CONSTRAINT div_check;
```

## B.4 ALTER TABLE ... SET HISTORY COLUMNS

```
ALTER TABLE table_name SET HISTORY COLUMNS ( col1, col2, colN ...)
```

### B.4.1 Usage

To further optimize the synchronization history process, after you set tables for synchronization history, you can use the SET HISTORY COLUMNS statement to specify which column updates in the master and its corresponding synchronized table cause entries to the history table. If you do not use this statement to specify particular columns, then all update operations (on all columns) in the master database cause a new entry to the history table when the corresponding synchronized table is updated. Generally, we recommend using ALTER TABLE ... SET HISTORY COLUMNS for columns that are used for search criteria or for joining.

## B.4.2 Usage in Master

Use SET SYNCHISTORY and SET HISTORY COLUMNS in the master to enable incremental publications on a table.

## B.4.3 Usage in Replica

Use SET SYNCHISTORY and SET HISTORY COLUMNS in the replica to enable incremental REFRESH on a table.



### Note

In order for ALTER TABLE ... SET HISTORY COLUMNS to succeed, the statement ALTER TABLE ... SET SYNCHISTORY has to be executed first. Executing ALTER TABLE ... SET NOSYNCHISTORY removes also the effect of ALTER TABLE ... SET HISTORY COLUMNS.

## B.4.4 Example

```
ALTER TABLE myLargeTable SET HISTORY COLUMNS (accountid);
```

## B.4.5 Return Values

For details on each error code, see the appendix titled Error Codes in *IBM solidDB Administration Guide*.

**Table B.2. ALTER TABLE SET HISTORY COLUMNS Return Values**

Error code	Description
13047	No privilege for operation
13100	Illegal table mode combination
13134	Table is not a base table
25038	Table is referenced in publication <i>publication_name</i> ; drop or alter operations are not allowed
25039	Table is referenced in subscription to publication <i>publication_name</i> ; drop or alter operations are not allowed.

## B.4.6 See Also

ALTER TABLE ... SET SYNCHISTORY

## B.5 ALTER TABLE ... SET SYNCHISTORY

```
ALTER TABLE table_name SET {SYNCHISTORY | NOSYNCHISTORY}
```

### B.5.1 Usage

*SET [NO]SYNCHISTORY*

The "SET SYNCHISTORY / NOSYNCHISTORY" clause tells the server to use the incremental publications mechanism of IBM solidDB architecture for this table. By default, SYNCHISTORY is not on. When this statement is set to SYNCHISTORY for a specified table, a shadow table is automatically created to store old versions of updated or deleted rows of the main table. The shadow table is called a "synchronization history table" or simply a "history table".

The data in a history table is referred to when a replica gets an incremental REFRESH from a publication in the master. For example, let's suppose that the record with Ms. Smith's telephone bill is deleted from the main table. A copy of her record is stored in the synchronization history table. When the replica refreshes, the master checks the history table and tells the replica that Ms. Smith's record was deleted. The replica can then delete that record, also. If the percentage of records that were deleted or changed is fairly small, then an incremental update is faster than downloading the entire table from the master. (When the user does a full REFRESH, rather than an incremental REFRESH, the history table is not used. The data in the table on the master is simply copied to the replica.)

Versioned data is automatically deleted from the database when there are no longer any replicas that need the data to fulfill REFRESH requests.

You must use this command to turn on synchronization history before a table can participate in master/replica synchronization. You can use this command on a table even if data currently exists in that table; however ALTER TABLE SET SYNCHISTORY can only be used if the specified table is not referenced by an existing publication.

SET SYNCHISTORY must be specified in the tables of both master and replica databases.

You can check if SYNCHISTORY is on for a table from the SYS\_TABLEMODES system table. The MODE column contains the SYNCHISTORY information.

You can use, for example, the query below:

```
SELECT mode
FROM SYS_TABLES, SYS_TABLEMODES
```

```
WHERE table_name = 'MY_TABLE' AND SYS_TABLEMODES.ID = SYS_TABLES.ID;
MODE
----
SYNCHISTORY
1 rows fetched.
```

`SYS_TABLEMODES` only shows the mode of tables for which the mode was explicitly set. In other words, `SYS_TABLEMODES` doesn't show the mode of tables that were left at the default mode. If `SYNCHISTORY` (or `NOSYNCHISTORY`) is not set for the table, the query returns an empty resultset.

## B.5.2 Usage in Master

Use `SET SYNCHISTORY` in the master to enable incremental publications on a table.

## B.5.3 Usage in Replica

Use `SET SYNCHISTORY` in the replica to enable incremental `REFRESHES` on a table.



### Note

If the Replica is read only (no changes are done to the replicated parts of the publication), the statement `ALTER TABLE ... SET SYNCHISTORY` is not needed. In the same time, the following Flow Replica-resident parameter should be set:

```
set sync parameter SYS_SYNC_KEEPLOCALCHANGES 'Yes';
```

## B.5.4 Example

```
ALTER TABLE myLargeTable SET SYNCHISTORY;
ALTER TABLE myVerySmallTable SET NOSYNCHISTORY;
```

## B.5.5 Return Values

For details on each error code, see the appendix titled *Error Codes* in the *IBM solidDB Administration Guide*.

**Table B.3. ALTER TABLE SET SYNCHISTORY Return Values**

Error code	Description
13047	No privilege for operation
13100	Illegal table mode combination
13134	Table is not a base table
25038	Table is referenced in publication <code>publication_name</code> ; drop or alter operations are not allowed
25039	Table is referenced in subscription to publication <code>publication_name</code> ; drop or alter operations are not allowed.

## B.5.6 See Also

ALTER TABLE ... SET HISTORY COLUMNS

## B.6 ALTER TRIGGER

```
ALTER TRIGGER trigger_name_attr SET {ENABLED | DISABLED}
trigger_name_attr ::= [ catalog_name.[ schema_name. ] ] trigger_name
```

### B.6.1 Usage

You can alter trigger attributes using the ALTER TRIGGER statement. The valid attributes are ENABLED and DISABLED trigger.

The ALTER TRIGGER DISABLED statement causes IBM solidDB to ignore the trigger when an activating DML statement is issued. With this command, you can also enable a trigger that is currently inactive or disable a trigger that is currently active.

You must be the owner of the table, or a user with DBA authority, to alter a trigger on the table.

## B.6.2 Example

```
ALTER TRIGGER trig_on_employee SET ENABLED;
```

## B.7 ALTER USER

```
ALTER USER user_name IDENTIFIED BY password
```

### B.7.1 Usage

The password of a user may be modified through the ALTER USER statement.

### B.7.2 Example

```
ALTER USER MANAGER IDENTIFIED BY O2CPTG;
```

## B.8 ALTER USER

```
ALTER USER replica_user SET MASTER master_name USER user_specification
```

where:

```
user_specification ::= { master_user IDENTIFIED BY master_password | NONE }
```

```
ALTER USER user_name SET {PUBLIC | PRIVATE}
```

### B.8.1 Usage

The following statement is used to map replica user ids to specified master user ids.

```
ALTER USER replica_user SET MASTER master_name USER user_specification
```

Mapping user ids is used for implementing security in a multi-master or multi-tier synchronization environment. In such environments, it is difficult to maintain the same username and passwords in separate, geographically dispersed databases. For this reason mapping is effective.

Only a user with DBA authority or `SYS_SYNC_ADMIN_ROLE` can map users. To implement mapping, an administrator must know the master user name and password. Note that it is always a replica user id that is mapped to a master user id. If `NONE` is specified, the mapping is removed.

All replica databases are responsible for subscribing to the `SYNC_CONFIG` system publication to update user information. Public master user names and passwords are downloaded, during this process, to a replica database using the `MESSAGE APPEND SYNC_CONFIG` command. Through mapping of the replica user id with the master user id, the system determines the currently active master user based on the local user id that is logged to the replica database. Note that if during `SYNC_CONFIG` loading, the system does not detect mapping, it determines the currently active master user through the matching user id and password in the master and the replica.

For more details on using mapping for security, read "Implementing Security Through Access Rights And Roles" in *IBM solidDB Advanced Replication Guide*.

It is also possible to limit what master users are downloaded to the replica during `SYNC_CONFIG` loading. This is done by altering users as private or public with the following command:

```
ALTER USER user_name SET PRIVATE | PUBLIC
```

Note that the default is `PUBLIC`. If the `PRIVATE` option is set for the user, that user's information is not included in a `SYNC_CONFIG` subscription, even if they are specified in a `SYNC_CONFIG` request. Only a user with DBA authority or `SYS_SYNC_ADMIN_ROLE` can alter a user's status.

This allows administrators to ensure no user ids with administration rights are sent to a replica. For security reasons, administrators may want to ensure that DBA passwords are never public, for example.

## B.8.2 Usage in Master

You set user ids to `PUBLIC` or `PRIVATE` in a master database.

## B.8.3 Usage in Replica

You map a replica user id to a master user id in a replica database.

## B.8.4 Example

The following example maps a replica user id `smith_1` to a master user id `dba` with a password of `dba`.

```
ALTER USER SMITH_1 SET MASTER MASTER_1 USER DBA IDENTIFIED BY DBA
```

The following example shows how users are set to PRIVATE and PUBLIC.

```
-- this master user should not be downloaded to any replica
ALTER USER dba SET PRIVATE;
```

```
-- this master user should be downloaded to every replica
ALTER USER salesman SET PUBLIC;
```

## B.8.5 Return Values

For details on each error code, see the appendix titled Error Codes in the *IBM solidDB Administration Guide*.

**Table B.4. ALTER USER Return Values**

Error code	Description
13047	No privilege for operation
13060	User name xxx not found
25020	Database is not a master database
25062	User <i>user_id</i> is not mapped to master <i>user_id</i>
25063	User <i>user_id</i> is already mapped to master <i>user_id</i>

## B.9 CALL

```
CALL procedure_name [( parameter [, parameter ...])] [AT node-def]
node-def ::= DEFAULT | <replica name> | <master name>
```

### B.9.1 Supported in

IBM solidDB disk-based engine, IBM solidDB (Note that remote procedure calls are allowed only with IBM solidDB with the advanced replication component)

### B.9.2 Usage

Stored procedures are called with statement CALL.



You may call a stored procedure on another node by using the `AT node_ref` clause. This is valid only if the call is made from a master node to one of its replica nodes or vice-versa.

`DEFAULT` means that the "current replica context" is used. The "current replica context" is only defined when the procedure call is started in the background using the `START AFTER COMMIT` statement with the `FOR EACH REPLICA` option. If the default is not set, then an error 'Default node not defined' is returned. `DEFAULT` can be used inside stored procedures and in a statement started with `START AFTER COMMIT`.

A remote stored procedure cannot return a result set; it can only return an error code.

A single call statement can only call a single procedure on a single node. If you want to call more than one procedure on a single node, you must execute multiple `CALL` statements. If you want to execute the same procedure (i.e. the same procedure name) on more than one node, then you have to either

1) Use

```
START AFTER COMMIT FOR EACH REPLICA.
```

E.g.

```
START AFTER COMMIT FOR EACH REPLICA WHERE NAME LIKE 'REPLICA%'  
UNIQUE CALL MYPROC AT DEFAULT.
```

2) Execute multiple calls.

A procedure call is executed synchronously; it returns after the call is executed.



### Note

The procedure call is executed asynchronously in the background if the procedure call is executed using `START AFTER COMMIT` (e.g. `START AFTER COMMIT UNIQUE CALL FOO AT REPLICA1`). That is due to the nature of the `START AFTER COMMIT` command, not the nature of procedure calls.

## B.9.3 Transactions

A remote procedure call (whether or not it was started by a `START AFTER COMMIT`) is executed in a separate transaction from the transaction that it was called from. The caller cannot roll back or commit the remote procedure call. The procedure that is executing in the called node is responsible for issuing its own commit or rollback statement.

## B.9.4 Return Values from the Remote Procedure

When you call a remote stored procedure, you cannot get a complete result set returned. All that you get is the return value of the stored procedure (a single value) or an error code.



### Note

If the remote procedure is executed in the background (using `START AFTER COMMIT`), then no return value is returned to the user. Even error codes are not returned.

## B.9.5 Access Rights for Remote Stored Procedure Calls

When a stored procedure is called remotely, you must take into account the access rights — i.e. does the caller have the right to execute this procedure on the remote server?

CASE 1. If the Sync user is set with the command `SET SYNC USER`.

The caller sends the user name and password of the "sync user" to the remote server, and the remote server tries to execute the procedure using that user name and password. In this case, the username and password must exist in the remote server (i.e. the server that the stored procedure will be executed on) and the user must have appropriate access rights to the database and the called procedure.

CASE 2. If the Sync user is not set:

The caller sends the following information to the remote server when calling a remote procedure:

If the caller is the master and the remote server is the replica ( $M \rightarrow R$ ):

- Name of the master (`SYS_SYNC_REPLICAS.MASTER_NAME`).
- Replica id (`SYS_SYNC_REPLICAS.ID`).
- User name of the caller.
- User id of the caller.

If the caller is the replica and the remote procedure is the master ( $R \rightarrow M$ ):

- Name of the master (`SYS_SYNC_MASTERS.NAME`).
- Replica id (`SYS_SYNC_MASTERS.REPLICA_ID`).

- Master user id (The same user id is used as when a replica refreshes data. There has to be a mapping from the local replica user to a master user in `SYS_SYNC_USERS` table.)

The following actions are performed in the called node:

If the remote node is a replica (M → R):

- Get the master id from table `SYS_SYNC_MASTERS` according to the master name received from the caller (master itself doesn't know it's id in the replica). From the table `SYS_SYNC_USERMAPS` get the replica user ids according to master user name and master id. Select the first user that has access rights to the procedure.
- If there are no matching rows in `SYS_SYNC_USERMAPS`, then get `NAME` and `PASSWD` from the table `SYS_SYNC_USERS` according to master id and master user name received from the caller and try to execute the procedure using them.

If the remote node is a master (R → M)

- Try to execute the procedure using the user id received from the replica.

If the replica allows calls from any master it should define its own connect string information in the `sol-id.ini` file, for example:

```
[Synchronizer]
ConnectStrForMaster=tcp replicahost 1316
```

The replica sends that connect string automatically to the master when it forwards any message to the master. When the master receives the connect string from the replica, it replaces any previous value (if it differs).

The master can set the connect string to the replica (if the replica has not done any messaging and the master needs to call it and knows that the connect string has changed) using the following statement:

```
SET SYNC CONNECT <connect-info> TO REPLICA <replica-name>
```

## B.9.6 Durability

Remote procedure calls are not durable. If the server goes down right after issuing the remote procedure call, then the call is lost. It will not be executed in recovery phase.

## B.9.7 Example

```
CALL proctest;  
CALL proctest('some string', 14);  
CALL remote_proc AT replica2;  
CALL RemoteProc(?,?) AT MyReplica1;
```

## B.10 COMMIT WORK

COMMIT WORK

### B.10.1 Usage

The changes made in the database are made permanent by the COMMIT statement. It terminates the transaction. To discard the changes, use the ROLLBACK command. Note that if you do not explicitly COMMIT a transaction, and if the program (for example solsql) does not COMMIT for you, then the transaction will be rolled back.

### B.10.2 Example

COMMIT WORK;

### B.10.3 See Also

ROLLBACK WORK

## B.11 CREATE CATALOG

CREATE CATALOG *catalog\_name*

### B.11.1 Usage

Catalogs allow you to logically partition databases so you can organize your data to meet the needs of your business or application. IBM solidDB's use of catalogs is an extension to the SQL standard.

A IBM solidDB physical database file may contain more than one *logical database*. Each logical database is a complete, independent group of database objects, such as tables, indexes, triggers, stored procedures, etc. Each logical database is implemented as a database catalog. Thus, IBM solidDB can have one or more catalogs.

When creating a new database or converting an old database to a new format, users are prompted for a default catalog name. This default catalog name allows for backward compatibility of IBM solidDB databases prior to version 3.x.

A catalog can have zero or more `schema_names`. The default schema name is the user ID of the user who creates the catalog.

A schema can have zero or more database object names. A database object can be qualified by a schema or user ID.

The catalog name is used to qualify a database object name.



### Caution

The catalog name must not contain spaces.

Database object names can be qualified in all DML statements as:

```
catalog_name.schema_name.database_object
```

or

```
catalog_name.user_id.database_object
```

Note that if you use the catalog name, then you must also use the schema name. The converse is not true; you may use the schema name without using the catalog name (if you have already done an appropriate SET CATALOG statement to specify the default catalog).

```
catalog_name.database_object -- Illegal  
schema_name.database_object -- Legal
```

Only a user with DBA authority (SYS\_ADMIN\_ROLE) can create a catalog for a database.

Note that creating a catalog does not automatically make that catalog the current default catalog. If you have created a new catalog and want your subsequent commands to execute within that catalog, then you must also execute the SET CATALOG statement. For example:

```
CREATE CATALOG MyCatalog;
CREATE SCHEMA smith; -- not in MyCatalog
SET CATALOG MyCatalog;
CREATE SCHEMA jones; -- in MyCatalog
```

For more information about SET CATALOG, see the description of the command "SET" in Section B.75, "SET".

To use schemas, a schema name must be created before creating the database object name. However, a database object name can be created without a schema name. In such cases, database objects are qualified using user\_id only. For details on creating schemas, read Section B.17, "CREATE SCHEMA".

A catalog context can be set in a program using:

```
SET CATALOG catalog_name
```

A catalog can be dropped from a database using:

```
DROP CATALOG catalog_name
```

When dropping a catalog name, all objects associated with the catalog name must be dropped prior to dropping the catalog.

Following are the rules for resolving catalog names:

- A fully qualified name (*catalog\_name.schema\_name.database\_object\_name*) does not need any name resolution, but will be validated.
- If a catalog context is not set using SET CATALOG, then all database object names are resolved always using the default catalog name as the catalog name. The database object name is resolved using schema name resolution rules. For details on these rules, read Section B.17, "CREATE SCHEMA".
- If a catalog context is set and the catalog name cannot be resolved using the *catalog\_name* in the context, then *database\_object\_name* resolution fails.
- To access a database system catalog, users do not need to know the system catalog name. Users can specify " " *.\_SYSTEM.table*". IBM solidDB translates the empty string " " used as a catalog name to the default catalog name. IBM solidDB also provides automatic resolution of *\_SYSTEM* schema to the system catalog, even when the catalog name is not provided.

## B.11.2 Examples

```
CREATE CATALOG C;
SET CATALOG C;
CREATE SCHEMA S;
SET SCHEMA S;
CREATE TABLE T (i INTEGER);
SELECT * FROM T;
-- the name T is resolved to C.S.T

-- Assume the userid is SMITH
CREATE CATALOG C;
SET CATALOG C;
CREATE TABLE T (i INTEGER);
SELECT * FROM T;
--The name T is resolved to C.SMITH.T

-- Assume there is no Catalog context set.
-- Meaning the default catalog name is BASE or the setting
-- of the base catalog.
CREATE SCHEMA S;
SET SCHEMA S;
CREATE TABLE T (i INTEGER);
SELECT * FROM T;
--The name T is resolved to <BASE>.S.T

CREATE CATALOG C1;
SET CATALOG C1;
CREATE SCHEMA S1;
SET SCHEMA S1;
CREATE TABLE T1 (c1 INTEGER);

CREATE CATALOG C2;
SET CATALOG C2;
CREATE SCHEMA S2;
SET SCHEMA S2;
CREATE TABLE T1 (c2 INTEGER)

SET CATALOG BASE;
SET SCHEMA USER;
```

```
SELECT * FROM T1;  
-- This select will give an error as it  
-- cannot resolve the T1.
```

## B.12 CREATE EVENT

```
CREATE EVENT event_name [( parameter_definition  
    [,parameter_definition ...])]
```

### B.12.1 Usage

Event alerts are used to signal an event in the database. Events are simple objects with a name. Applications can use event alerts instead of polling, which uses more resources.

An event object is created with the SQL statement

```
CREATE EVENT event_name [parameter_list]
```

The name can be any user-specified alphanumeric string. The parameter list specifies parameter names and parameter types. The parameter types are normal SQL types.

Events are dropped with the SQL statement

```
DROP EVENT event_name
```

Events are sent and received inside stored procedures. Special stored procedure statements are used to send and receive events.

The event is sent with the stored procedure statement

```
post_statement ::= POST EVENT event_name  
    [( parameters) ] [UNIQUE | DATA UNIQUE]
```

Event parameters must be local variables, constant values, or parameters in the stored procedure from which the event is sent.

The keyword UNIQUE means that only last post is kept in event queue for each user and for each event. For example after POST EVENT EV(1) and POST EVENT EV(2) only EV(2) is in event queue if EV(1) is not processed before EV(2) is posted. Event EV(1) is discarded. The keyword DATA UNIQUE means that also



event parameters must be unique. So after calls POST EVENT EV(1), POST EVENT EV(2) and POST EVENT EV(2) events EV(1) and EV(2) are kept in event queue. First EV(2) is discarded.

All clients that are waiting for the posted event will receive the event. Each connection has its own event queue. The events to be collected in the event queue are specified with the stored procedure statement:

```
wait_register_statement ::= REGISTER EVENT event_name
```

Events are removed from the event queue with the stored procedure statement:

```
wait_unregister_statement ::= UNREGISTER EVENT event_name
```

Note that you do not need to register for every event before waiting for it. When you wait on an event, you will be registered implicitly for that event if you did not already explicitly register for it. Thus you only need to explicitly register events if you want them to start being queued now but you don't want to start WAITing for them until later.

To make a procedure wait for an event to happen, the WAIT EVENT construct is used in a stored procedure:

```
wait_event_statement ::=  
    WAIT EVENT  
        [event_specification ...]  
    END WAIT  
  
event_specification ::=  
    WHEN event_name [(parameters)] BEGIN  
        statements  
    END EVENT
```

Each connection has its own event queue. To specify the events to be collected in the event queue, use the command REGISTER EVENT *event\_name*. Events are removed from the event queue by the command UNREGISTER EVENT *event\_name*.

```
"CREATE PROCEDURE register_event  
begin  
    register event test_event  
end";  
  
"CREATE PROCEDURE unregister_event
```

```
begin
    unregister event test_event
end";
```

The creator of an event or the database administrator can grant and revoke access rights on that event. Access rights can be granted to users and roles. If a user has "SELECT" access right on an event, then the user has the right to wait on that event. If a user has the INSERT access right on an event, then the user may post that event.

If you want to stop the stored procedure waiting for an event, you can use ODBC function `SQLCancel()` called from a separate thread in the client application. This function cancels executing statements. Alternatively, you can create a specific user event and send it. The waiting stored procedure must be modified to wait for this additional event. The client application recognises this event and exits the waiting loop.

For in-depth examples of events usage, refer to the section Section 4.11, "Using Events". The example includes a pair of SQL scripts that when used together post and wait for multiple events.

## B.12.2 Example

```
CREATE EVENT ALERT1(I INTEGER, C CHAR(4));
```

## B.12.3 See Also

```
CREATE PROCEDURE
```

## B.13 CREATE INDEX

```
CREATE [UNIQUE] INDEX index_name
    ON base_table_name
    (column_identifier [ASC | DESC]
    [, column_identifier [ASC | DESC]] ...)
```

### B.13.1 Usage

Creates an index for a table based on the given columns.

The keyword UNIQUE specifies that the column(s) being indexed must contain unique values. If more than one column is specified, then the combination of columns must have a unique value, but the individual columns do not need to have unique values. For example, if you create an index on the combination of LAST\_NAME

and FIRST\_NAME, then the following data values are acceptable because although there are duplicate first names and duplicate last names, no 2 rows have the same value for both first name and last name.

```
SMITH, PATTI  
SMITH, DAVID  
JONES, DAVID
```

Keywords ASC and DESC specify whether the given columns should be indexed in ascending or descending order. If neither ASC nor DESC is specified, then ascending order is used.

## B.13.2 Example

```
CREATE UNIQUE INDEX UX_TEST ON TEST (I);  
CREATE INDEX X_TEST ON TEST (I DESC, J DESC);
```

## B.13.3 See Also

Section B.15, “CREATE [OR REPLACE] PUBLICATION”.

## B.14 CREATE PROCEDURE

```
CREATE PROCEDURE procedure_name [(parameter_definition  
  [, parameter_definition ...])] [RETURNS (output_column_definition [, output_column_definition ...])]  
  BEGIN procedure_body END;  
parameter_definition ::= [parameter_mode] parameter_name data_type  
output_column_definition ::= column_name column_type  
procedure_body ::= [declare_statement; ...][procedure_statement; ...]  
  
parameter_mode ::= IN | OUT | INOUT  
  
declare_statement ::= DECLARE variable_name data_type  
  
procedure_statement ::= prepare_statement | execute_statement |  
  fetch_statement | control_statement | post_statement |  
  wait_event_statement | wait_register_statement | exec_direct_statement |  
  writetrace_statement | sql_dml_or_ddl_statement
```

```
prepare_statement ::= EXEC SQL PREPARE
  { cursor_name | CURSORNAME( { string_literal | variable } ) }
  sql_statement

execute_statement ::=
  EXEC SQL EXECUTE cursor_name
    [USING (variable [, variable ...])]
    [INTO (variable [, variable ...])] |
  EXEC SQL CLOSE cursor_name |
  EXEC SQL DROP cursor_name |
  EXEC SQL {COMMIT | ROLLBACK} WORK |
  EXEC SQL SET TRANSACTION {READ ONLY | READ WRITE} |
  EXEC SQL WHENEVER SQLERROR {ABORT | ROLLBACK [WORK], ABORT}
  EXEC SEQUENCE sequence_name.CURRENT INTO variable |
  EXEC SEQUENCE sequence_name.NEXT INTO variable |
  EXEC SEQUENCE sequence_name SET VALUE USING variable

fetch_statement ::= EXEC SQL FETCH cursor_name

cursor_name ::=
  literal

post_statement ::= POST EVENT event_name [(parameters)]

wait_event_statement ::=
  WAIT EVENT
  [event_specification ...]
  END WAIT

event_specification ::=
  WHEN event_name [(parameters)] BEGIN
    statements
  END EVENT

wait_register_statement ::=
  REGISTER EVENT event_name |
  UNREGISTER EVENT event_name

writetrace_statement ::=
  WRITETRACE(string)

control_statement ::=
  SET variable_name = value | variable_name ::= value |
  WHILE expression
```

```
    LOOP procedure_statement... END LOOP |
  LEAVE |
  IF expression THEN procedure_statement ...
    [ ELSEIF procedure_statement ... THEN] ...
    ELSE procedure_statement ... END IF |
  RETURN | RETURN SQLERROR OF cursor_name | RETURN ROW |
  RETURN NO ROW
exec_direct_statement ::=
  EXEC SQL [USING (variable [, variable ...])]
  [CURSORNAME(variable)]
  EXECDIRECT sql_dml_or_ddl_statement |
  EXEC SQL cursor_name
  [USING (variable [, variable ...])]
  [INTO (variable [, variable ...])]
  [CURSORNAME(variable)]
  EXECDIRECT sql_dml_or_ddl_statement
```

## B.14.1 Usage

Stored procedures are simple programs, or procedures, that are executed in the server. The user can create a procedure that contains several SQL statements or a whole transaction and execute it with a single call statement. Usage of stored procedures reduces network traffic and allows more strict control to access rights and database operations.

Procedures are created with the statement

```
CREATE PROCEDURE name body
```

and dropped with the statement

```
DROP PROCEDURE name
```

Procedures are called with the statement

```
CALL name [parameter ...]
```

All SQL stored procedures are executed in the Primary unless they are specified as read-only procedures by way of the SQL standard clause *SQL Data Access Indication*, in the procedure declaration.

```
<SQL-data-access-indication> ::=  
    NO SQL |  
    READS SQL DATA |  
    CONTAINS SQL |  
    MODIFIES SQL DATA
```

To avoid unnecessary handovers of read-only procedures and functions, one of the following values can be declared:

- NO SQL
- READS SQL DATA
- CONTAINS SQL

Only MODIFIES SQL DATA (which is the default) inflicts transaction handover.

The clause comes between the (optional) RETURNS clause and the procedure body. For example:

```
"CREATE PROCEDURE PHONEBOOK_SEARCH  
( IN FIRST_NAME VARCHAR, LAST_NAME VARCHAR )  
RETURNS ( PHONE_NR NUMERIC, CITY VARCHAR )  
READS SQL DATA  
BEGIN  
-- procedure_body  
END" ;  
  
---
```

Stored procedures provide for three different parameter modes: input parameters, output parameters, and input/output parameters. The parameter modes are:

1. Input parameters are passed to the stored procedure from the calling program. The *parameter\_mode* value is IN. This is the default behaviour.
2. Output parameters are returned to the calling program from the stored procedure. The *parameter\_mode* value is OUT.
3. Input/output parameters pass values into the procedure and return a value back to the calling procedure. The *parameter\_mode* is INOUT.

See the table below for a comparison of the parameter modes:

**Table B.5. Comparison of the Parameter Modes**

Feature	IN	OUT	INOUT
Default/specified	Default.	Must be specified.	Must be specified.
Operation	Passes values to a subprogram.	Returns values to the caller.	Passes initial values to a subprogram; returns updated values to the caller.
Action	Formal parameter, acts like a constant.	Formal parameter, acts like an uninitialised variable.	Formal parameter, acts like an initialised variable.
Value assignation	Formal parameter, cannot be assigned a value.	Formal parameter, cannot be used in an expression; must be assigned a value.	Formal parameter, should be assigned a value.
Parameter type	Actual parameter, can be a constant, initialised variable, literal, or expression.	Actual parameter, must be a variable.	Actual parameter, must be a variable.

At programming interfaces, the output parameters are bound to variables as follows:

In JDBC, with the method `CallableStatement.registerOutParameter()`.

In ODBC, with the function `SQLBindParameter()`, where the third argument, *InputOutputType*, may be of type:

`SQL_PARAM_INPUT`

`SQL_PARAM_OUTPUT`

`SQL_PARAM_INPUT_OUTPUT`

For more information on binding parameters to variables, refer to *IBM solidDB Programmer Guide*.

Note that it is syntactically valid, although not useful, to create a stored procedure with an empty body.

Procedures are owned by the creator of the procedure. Specified access rights can be granted to other users. When the procedure is run, it has the creator's access rights to database objects.

The stored procedure syntax is a proprietary syntax modeled from SQL-99 specifications and dynamic SQL. Procedures contain control statements and SQL statements.

The following control statements are available in the procedures:

**Table B.6. Control Statements**

Control Statement	Description
<code>set <i>variable</i> = <i>expression</i></code>	Assigns a value to a variable. The value can be either a literal value (e.g., 10 or 'text') or another variable. Parameters are considered as normal variables.
<code><i>variable</i> ::= <i>expression</i></code>	Alternate syntax for assigning values to variables.
<pre>while   <i>expr</i> loop   <i>statement-list</i> end loop</pre>	Loops while expression is true.
leave	Leaves the innermost while loop and continues executing the procedure from the next statement after the keyword end loop.
<pre>if   <i>expr</i> then   <i>statement-list1</i> else   <i>statement-list2</i> end if</pre>	Executes <i>statements-list1</i> if expression <i>expr</i> is true; otherwise, executes <i>statement-list2</i> .
<pre>if   <i>expr1</i> then   <i>statement-list1</i></pre>	If <i>expr1</i> is true, executes <i>statement-list1</i> . If <i>expr2</i> is true, executes <i>statement-list2</i> . The statement can optionally contain multiple elseif statements and also an else statement.



Control Statement	Description
<pre>elseif     <i>expr2</i> then     <i>statement-list2</i> end if</pre>	
<pre>return</pre>	Returns the current values of output parameters and exits the procedure. If a procedure has a <i>return row</i> statement, return behaves like <i>return norow</i> .
<pre>return sqlerror of <i>cursor-name</i></pre>	Returns the sqlerror associated with the cursor and exits the procedure.
<pre>return row</pre>	Returns the current values of output parameters and continues execution of the procedure. Return row does not exit the procedure and return control to the caller.
<pre>return norow</pre>	Returns the end of the set and exits the procedure.

All SQL DML and DDL statements can be used in procedures. Thus the procedure can, for example, create tables or commit a transaction. Each SQL statement in the procedure is atomic.

The "autocommit" functionality works differently for statements inside a stored procedure than for statements outside a stored procedure. For SQL statements outside a stored procedure, each individual statement is implicitly followed by a COMMIT WORK operation when autocommit is on. For a stored procedure, however, the implicit COMMIT WORK is executed after the stored procedure has returned to the caller. Note that this does not imply that a stored procedure is "atomic". As indicated above, a stored procedure may contain its own COMMIT and ROLLBACK commands. The implicit COMMIT WORK executed after the procedure returns will commit only that portion of the stored procedure statements that were executed since:

- the last COMMIT WORK inside the procedure
- the last ROLLBACK WORK inside the procedure
- the start of the procedure (if no COMMIT or ROLLBACK commands were executed during the procedure)

Note that if one stored procedure is called from inside another, the implicit COMMIT WORK is done only after the end of the OUTERMOST procedure call. There is no implicit COMMIT WORK done after "nested" procedure calls.

For example, in the following script, the implicit COMMIT WORK is executed only after the **CALL outer\_proc();** statement:

```
"CREATE PROCEDURE inner_proc
BEGIN
    ...
END";
CREATE PROCEDURE outer_proc
BEGIN
    ...
    EXEC SQL PREPARE cursor1 CALL inner_proc();
    EXEC SQL EXECUTE cursor1;
    ...
END";
    CALL outer_proc();
```

## B.14.2 Preparing SQL Statements

The SQL statements are first prepared with the statement

```
EXEC SQL PREPARE cursor sql_statement
```

The *cursor* specification is a cursor name that must be given. It can be any unique cursor name inside the transaction. Note that if the procedure is not a complete transaction, other open cursors outside the procedure may have conflicting cursor names.

## B.14.3 Executing Prepared SQL Statements

The SQL *statement* is executed with the statement

```
EXEC SQL EXECUTE cursor [opt_using] [opt_into]
```

The optional *opt-using* specification has the syntax

USING (*variable\_list*)

where *variable\_list* contains a list of procedure variables or parameters separated by a comma. These variables are input parameters for the SQL statement. The SQL input parameters are marked with the standard question mark syntax in the prepare statement. If the SQL statement has no input parameters, the USING specification is ignored.

The optional *opt\_into* specification has the syntax

INTO (*variable\_list*)

where *variable\_list* contains the variables that the column values of the SQL SELECT statement are stored into. The INTO specification is effective only for SQL SELECT statements.

After the execution of UPDATE, INSERT and DELETE statements an additional variable is available to check the result of the statement. Variable SQLROWCOUNT contains the number of rows affected by the last statement.

## B.14.4 Fetching Results

Rows are fetched with the statement

```
EXEC SQL FETCH cursor_name
```

If the fetch completed successfully, then the column values are stored into the variables defined in the *opt\_into* specification of the EXECUTE or EXECDIRECT statement.

## B.14.5 Closing and Dropping Cursors

When you are finished using a cursor, you should CLOSE the cursor and DROP the cursor. If you do not do this, then resources (such as memory) that are allocated to the cursor may not be freed up for re-use.

## B.14.6 Checking for Errors

The result of each EXEC SQL statement executed inside a procedure body is stored into the variable SQLSUCCESS. This variable is automatically generated for every procedure. If the previous SQL statement was successful, a value one is stored into SQLSUCCESS. After a failed SQL statement, a value zero is stored into SQLSUCCESS.

```
EXEC SQL WHENEVER SQLERROR {ABORT | [ROLLBACK [WORK], ABORT]}
```

is used to decrease the need for `IF NOT SQLSUCCESS THEN` tests after every executed SQL statement in a procedure. When this statement is included in a stored procedure all return values of executed statements are checked for errors. If statement execution returns an error, the procedure is automatically aborted. Optionally the transaction can be rolled back.

The error string of latest failed SQL statement is stored into variable `SQLERRSTR`.

### B.14.7 Using Transactions

```
EXEC SQL {COMMIT | ROLLBACK} WORK
```

is used to terminate transactions.

```
EXEC SQL SET TRANSACTION {READ ONLY | READ WRITE}
```

is used to control the type of transactions.

### B.14.8 Using Sequencer Objects and Event Alerts

Refer to the usage of the `CREATE SEQUENCE` and `CREATE EVENT` statements.

### B.14.9 Writetrace

The `writetrace()` function allows you to send a string to the `soltrace.out` trace file. This can be useful when debugging problems in stored procedures.

The output will only be written if you turn tracing on.

For more information about writetrace and how to turn on tracing, see Section 7.4, “Tracing Facilities for Stored Procedures and Triggers”.

### B.14.10 Procedure Stack Functions

The following functions may be used to analyze the current contents of the procedure stack: `PROC_COUNT()`, `PROC_NAME(N)`, `PROC_SCHEMA(N)`.

`PROC_COUNT()` returns the number of procedures in the procedure stack. This includes the current procedure.

`PROC_NAME(N)` returns the Nth procedure name in the stack. First procedure position is zero.

`PROC_SCHEMA(N)` returns the schema name of the Nth procedure in procedure stack.

## B.14.11 Dynamic Cursor Names

```
CURSORNAME(  
    prefix -- VARCHAR  
)
```

The `CURSORNAME()` function allows you to dynamically generate a cursor name rather than hard-coding the cursor name.



### Note

Strictly speaking, `CURSORNAME()` is not a function, despite the syntactic similarity. `CURSORNAME(arg)` does not actually return anything; instead it sets the name of the current statement's cursor based on the given argument. However, it is convenient to refer to it as a function, and therefore we will do so.

Cursor names must be unique within a connection. This causes problems in recursive stored procedures because each invocation uses the same cursor name(s). When the recursive procedure calls itself, the second invocation will find that the first invocation has already created a cursor with the same name as the second invocation wants to use.

To get around this problem, we must generate unique cursor names dynamically, and we must be able to use those names when we declare and use cursors. To enable us to generate unique names and use them as cursors, we use 2 functions:

- `GET_UNIQUE_STRING`
- `CURSORNAME`

The `GET_UNIQUE_STRING` function does just what its name suggests — it generates a unique string. The `CURSORNAME` function (actually a pseudo-function) allows you to use a dynamically generated string as part of a cursor name.

Note that `GET_UNIQUE_STRING` returns a different output each time it is called, even if the input is the same. `CURSORNAME`, on the other hand, returns the same output each time if the input is the same each time.

Below is an example of using `GET_UNIQUE_STRING` and `CURSORNAME` to dynamically generate a cursor name. The dynamically generated cursorname is assigned to the placeholder "cname", which is then used in each statement after the `PREPARE`.

```
DECLARE autoname VARCHAR;  
Autoname := GET_UNIQUE_STRING('CUR_');  
EXEC SQL PREPARE cname CURSORNAME(autoname) SELECT * FROM TABLES;  
EXEC SQL EXECUTE cname USING(...) INTO(...);  
EXEC SQL FETCH cname;  
EXEC SQL CLOSE cname;  
EXEC SQL DROP cname;
```

`CURSORNAME()` can only be used in `PREPARE` statements and `EXECDIRECT` statements. It cannot be used in `EXECUTE`, `FETCH`, `CLOSE`, `DROP`, etc.

By using the `CURSORNAME()` feature and the `GET_UNIQUE_STRING()` function, you can generate unique cursor names in recursive stored procedures. If the procedure calls itself, then each time that this function is called within the stored procedure, this function will return a unique string that can be used as the cursor name in a `PREPARE` statement. See below for some examples of code that you could use inside a stored procedure.

Note that each call to `CURSORNAME(autoname)` returns the same value — i.e. the same cursor name, as long as the input (autoname) does not change.

## B.14.12 EXECDIRECT

The `EXECDIRECT` statement allows you to execute statements inside stored procedures without first "preparing" those statements. This reduces the amount of code required. Note that if the statement is a cursor, you still need to close and drop it; only the `PREPARE` statement can be skipped.

When using

```
EXEC SQL [USING(var_list)] [CURSORNAME(variable)]  
EXECDIRECT <statement>
```

or

```
EXEC SQL <cursor_name> [USING(var_list)] [INTO (var_list)]  
[CURSORNAME(variable)] EXECDIRECT <statement>
```

remember the following rules:

- If the statement specifies a cursor name, then the cursor must be dropped with the EXEC SQL DROP statement.
- If a cursor name is not specified, then you don't need to drop the statement.
- If the statement is a fetch cursor, then the INTO... clause must be specified.
- If the INTO clause is specified, then the cursor\_name must be specified; otherwise the FETCH statement won't be able to specify which cursor name the row should be fetched from. (You may have more than one open cursor at a time.)

Below are several examples of CREATE PROCEDURE statements. Some use the PREPARE and EXECUTE commands, while others use EXECDIRECT.

### B.14.13 CREATE PROCEDURE

```
"create procedure test2(tableid integer)
  returns (cnt integer)
begin
  exec sql prepare c1 select count(*) from sys_tables where id > ?;
  exec sql execute c1 using (tableid) into (cnt);
  exec sql fetch c1;
  exec sql close c1;
  exec sql drop c1;
end";
```

### B.14.14 Using the Explicit RETURN Statement

This example uses the explicit RETURN statement to return multiple rows, one at a time.

```
"create procedure return_tables
  returns (name varchar)
begin
  exec sql execdirect create table table_name (lname char (20));
  exec sql whenever sqlerror rollback, abort;
  exec sql prepare c1 select table_name from sys_tables;
  exec sql execute c1 into (name);
  while sqlsuccess loop
    exec sql fetch c1;
    if not sqlsuccess
```

```
        then leave;
    end if
    return row;
end loop;
exec sql close c1;
exec sql drop c1;
end";
```

## B.14.15 Using EXECDIRECT

```
-- This example shows how to use "execdirect".
"CREATE PROCEDURE p
BEGIN
    DECLARE host_x INT;
    DECLARE host_y INT;

    -- Examples of execdirect without a cursor. Here we create a table
    -- and insert a row into that table.
    EXEC SQL EXECDIRECT create table foo (x int, y int);
    EXEC SQL EXECDIRECT insert into foo(x, y) values (1, 2);

    SET host_x = 1;

    -- Example of execdirect with cursor name.
    -- In this example, "c1" is the cursor name; "host_x" is the
    -- variable whose value will be substituted for the "?";
    -- "host_y" is the variable into which we will store the value of the
    -- column y (when we fetch it).
    -- Note: although you don't need a "prepare" statement, you still
    -- need close/drop.
    EXEC SQL c1 USING(host_x) INTO(host_y) EXECDIRECT
        SELECT y from foo where x=?;
    EXEC SQL FETCH c1;
    EXEC SQL CLOSE c1;
```



```
EXEC SQL DROP c1;
END";
```

## B.14.16 Using CURSORNAME

This example shows the usage of the CURSORNAME() pseudo-function. This shows only part of the body of a stored procedure, not a complete stored procedure.

```
-- Declare a variable that will hold a unique string that we can use
-- as a cursor name.
DECLARE autoname VARCHAR ;
Autoname := GET_UNIQUE_STRING('CUR_') ;
EXEC SQL PREPARE curs_name CURSORNAME(autoname) SELECT * FROM TABLES;
EXEC SQL EXECUTE curs_name USING(...) INTO(...);
EXEC SQL FETCH curs_name;
EXEC SQL CLOSE curs_name;
EXEC SQL DROP curs_name;
```

## B.14.17 Using GET\_UNIQUE\_STRING and CURSORNAME

Here is a more complete example that actually uses the GET\_UNIQUE\_STRING and CURSORNAME functions in a recursive stored procedure.

The stored procedure below demonstrates the use of these two functions in a recursive procedure. Note that the cursor name "curs1" appears to be hard-coded, but in fact has been mapped to the dynamically generated name.

```
-- Demonstrate GET_UNIQUE_STRING and CURSORNAME functions in a
-- recursive stored procedure.
-- Given a number N greater than or equal to 1, this procedure
-- returns the sum of the numbers 1 - N. (We could do this in a loop,
-- of course, but the purpose of the example is to show the use of the
-- CURSORNAME function in a recursive procedure.)
"CREATE PROCEDURE Sum1ToN(n INT)
RETURNS (SumSoFar INT)
BEGIN
    DECLARE SumOfRemainingItems INT;
    DECLARE nMinusOne INT;
    DECLARE autoname VARCHAR;
```

```
SumSoFar := 0;
SumOfRemainingItems := 0;
nMinusOne := n - 1;

IF (nMinusOne > 0) THEN
    Autaname := GET_UNIQUE_STRING('CURSOR_NAME_PREFIX_') ;
    EXEC SQL PREPARE curs1 CURSORNAME(autaname) CALL Sum1ToN(?);
    EXEC SQL EXECUTE curs1 USING(nMinusOne) INTO(SumOfRemainingItems);
    EXEC SQL FETCH curs1;
    EXEC SQL CLOSE curs1;
    EXEC SQL DROP curs1;
END IF;

SumSoFar := n + SumOfRemainingItems;
END";
```

## B.14.18 Example 6

Using EXECDIRECT in CREATE PROCEDURE

```
CREATE TABLE table1 (x INT, y INT);
INSERT INTO table1 (x, y) VALUES (1, 2);

"CREATE PROCEDURE FOO
RETURNS (r INT)
BEGIN
DECLARE autaname VARCHAR;
Autaname := GET_UNIQUE_STRING('CUR_');
EXEC SQL curs_name INTO(r) CURSORNAME(autaname) EXECDIRECT
    SELECT y FROM TABLE1 WHERE x = 1;
EXEC SQL FETCH curs_name;
EXEC SQL CLOSE curs_name;
EXEC SQL DROP curs_name;
END";
```

```
CALL foo();
SELECT * FROM table1;
```

## B.14.19 Creating a Unique Name for a Synchronization Message

Creating a unique name for a synchronization message:

```
DECLARE Autoname VARCHAR;
DECLARE Sqlstr VARCHAR;
Autoname := get_unique_string('MSG_') ;
Sqlstr := 'MESSAGE' + autoname + 'BEGIN';
EXEC SQL EXECDIRECT Sqlstr;
...
Sqlstr := 'MESSAGE' + autoname + 'FORWARD';
EXEC SQL EXECDIRECT Sqlstr;
```

## B.14.20 Using GET\_UNIQUE\_STRING

```
-- This demonstrates how to use the GET_UNIQUE_STRING() function
-- to generate unique message names from within a recursive stored
-- procedure.
```

```
CREATE TABLE table1 (i int, beginMsg VARCHAR, endMsg VARCHAR);
```

```
-- This is a simplified example of recursion.
-- Note that the messages I compose are not actually used! This is
-- not a true example of synchronization; it's only an example of
-- generating unique message names. The "count" parameter is the
-- number of times that you want this function to call
-- itself (not including the initial call).
```

```
"CREATE PROCEDURE repeater(count INT)
```

```
BEGIN
```

```
DECLARE Autoname VARCHAR;
DECLARE MsgBeginStr VARCHAR;
DECLARE MsgEndStr VARCHAR;
```

```
Autoname := GET_UNIQUE_STRING('MSG_');
```

```

MsgBeginStr := 'MESSAGE ' + Autoname + ' BEGIN';
MsgEndStr := 'MESSAGE ' + Autoname + ' END';

EXEC SQL c1 USING (count, MsgBeginStr, MsgEndStr) EXECDIRECT
    INSERT INTO table1 (i, beginMsg, endMsg) VALUES (?, ?, ?);
EXEC SQL CLOSE c1;
EXEC SQL DROP c1;

-- Once you have composed the SQL statement as a string,
-- you can execute it one of two ways:
-- 1) by using the EXECDIRECT feature or
-- 2) by preparing and executing the SQL statement.
-- In this example, we use EXECDIRECT.
EXEC SQL EXECDIRECT MsgBeginStr;
EXEC SQL EXECDIRECT MsgEndStr;
-- Do something useful here.

-- The recursive portion of the function.
IF (count > 1) THEN
    SET count = count - 1;
    -- Note that we can also use our unique name as a cursor name,
    -- as shown below.
    EXEC SQL Autoname USING (count) EXECDIRECT CALL repeater(?);
    EXEC SQL CLOSE Autoname;
    EXEC SQL DROP Autoname;
END IF

RETURN;
END";

CALL repeater(3);

-- Show the message names that we composed.
SELECT * FROM table1;

```

The output from this SELECT statement would look similar to the following:

```

I  BEGINMSG                                ENDMSG
-- -----                                -----
1  MESSAGE MSG_019 BEGIN                    MESSAGE MSG_019 END

```

```
2 MESSAGE MSG_020 BEGIN      MESSAGE MSG_020 END
3 MESSAGE MSG_021 BEGIN      MESSAGE MSG_021 END
```

## B.15 CREATE [OR REPLACE] PUBLICATION

```
"CREATE [OR REPLACE] PUBLICATION publication_name
  [(parameter_definition [,parameter_definition...])]
BEGIN
  main_result_set_definition...
END";
```

```
main_result_set_definition ::=
RESULT SET FOR main_replica_table_name
```

```
BEGIN
  SELECT select_list
  FROM master_table_name
  [ WHERE search_condition ] ;
  [ [DISTINCT] result_set_definition...]
END
```

```
result_set_definition ::=
RESULT SET FOR replica_table_name
```

```
BEGIN
  SELECT select_list
  FROM master_table_name
  [ WHERE search_condition ] ;
  [[ DISTINCT] result_set_definition...]
END
```

NOTE: *Search\_condition* can reference *parameter\_definitions* and/or columns of replica tables defined on previous (higher) levels.

### B.15.1 Usage

Publications define the sets of data that can be REFRESHed from the master to the replica database. A publication is always transactionally consistent, that is, its data has been read from the master database in one transaction and the data is written to the replica database in one transaction.

 **Caution**

The data read from the publication is internally consistent unless the master is using the READ COMMITTED isolation level.

Search conditions of a SELECT clause can contain input arguments of the publication as parameters. The parameter name must have a colon as a prefix.

Publications can contain data from multiple tables. The tables of the publication can be independent or there can be relations between the tables. If there is a relation between tables, the result sets must be nested. The WHERE clause of the SELECT statement of the inner result set of the publication must refer to a column of the table of the outer result set.

If the relation between outer and inner result set of the publication is a N-1 relationship, then the keyword DISTINCT must be used in the result set definition.

The *replica\_table\_name* can be different from the *master\_table\_name*. The publication definition provides the mapping between the master and replica tables. (If you have multiple replicas, all the replicas should use the same name, even if that name is different from the name used in the master.) Column names in the master and replica tables must be the same.

Note that the initial download is always a *full publication*, which means that all data contained in the publication is sent to the replica database. Subsequent downloads (refreshes) for the same publication may be *incremental publications*, which means that they contain only the data that has been changed since the prior REFRESH. To enable usage of incremental publications, SYNCHISTORY has to be set ON for tables included in the publication in both the master and replica databases. For details, read Section B.5, “ALTER TABLE ... SET SYNCHISTORY” and Section B.32, “DROP PUBLICATION REGISTRATION”.

If the optional keywords "OR REPLACE" are used, then if the publication already exists it will be replaced with the new definition. Since the publication was not dropped and recreated, replicas do not need to re-register, and subsequent REFRESHes from that publication can be incremental rather than full, depending upon exactly what changes were made to the publication.

To avoid having a replica refresh from a publication while you are updating that publication, you may temporarily set the catalog's sync mode to Maintenance mode. However, using maintenance mode is not absolutely required when replacing a publication.

If you replace an existing publication, the new definition of the publication will be sent to each replica the next time that replica requests a refresh. The replica does not need to explicitly re-register itself to the publication.

When you replace an existing publication with a new definition, you may change the resultset definitions. You cannot change the parameters of the publication. The only way to change the parameters is to drop the

publication and create a new one, which also means that the replicas must re-register and the replicas will get a full refresh rather than an incremental refresh the next time that they request a refresh.

When you replace an existing publication, the privileges related to that publication are left unchanged. (You do not need to re-create them.)

The `CREATE OR REPLACE PUBLICATION` command can be executed in any situation where it is valid to execute the `CREATE PUBLICATION` command.



### Caution

If you use `CREATE OR REPLACE PUBLICATION` to alter the contents of an existing advanced replication publication, you have to take care of removing invalid rows from Replica.

## B.15.2 Usage in Master

You define the publication in the master database to enable the replicas to get refreshes from it.

## B.15.3 Usage in Replica

There is no need to define the publications in the replicas. Publication subscription functionality depends on the definitions only at the master database. If this command is executed in a replica, it will store the publication definition to the replica, but the publication definition is not used for anything. (Note that if a database is both a replica (for a master above it) and a master (to a replica below it), then of course you may want to create a publication definition in the database.)

## B.15.4 Example

The following sample publication retrieves data from the customer table using the area code of the customer as search criterion. For each customer, the orders and invoices of the customer (1-N relation) as well as the dedicated salesman of the customer (1-1 relation) are also retrieved.

```
"CREATE PUBLICATION PUB_CUSTOMERS_BY_AREA
  ( IN_AREA_CODE VARCHAR )
BEGIN
  RESULT SET FOR CUSTOMER
  BEGIN
    SELECT * FROM CUSTOMER
    WHERE AREA_CODE = :IN_AREA_CODE ;
  RESULT SET FOR CUST_ORDER
  BEGIN
```

```
        SELECT * FROM CUST_ORDER
        WHERE CUSTOMER_ID = CUSTOMER.ID;
    END
    RESULT SET FOR INVOICE
BEGIN
    SELECT * FROM INVOICE
    WHERE CUSTOMER_ID = CUSTOMER.ID;
END
DISTINCT RESULT SET FOR SALESMAN
BEGIN
    SELECT * FROM SALESMAN
    WHERE ID = CUSTOMER.SALESMAN_ID;
END
END
END";
```



## Note

The colon (:) in :IN\_AREA\_CODE is used to designate a reference to a publication parameter with the same name.

### EXAMPLE 2:

Developers decided to add a new column C in table T, which is referenced in publication P. The modification must be made to the master database and all replica databases.

The tasks to execute in the master database are:

```
-- Prevent other users from doing concurrent synchronization operations
-- to this catalog.
SET SYNC MAINTENANCE MODE ON;
ALTER TABLE T ADD COLUMN C INTEGER;
COMMIT WORK;
CREATE OR REPLACE PUBLICATION P ... (column C added also to publication)
COMMIT WORK;
SET SYNC MAINTENANCE MODE OFF;
```

The tasks to execute in all replica databases are:

```
-- Prevent other users from doing concurrent synchronization operations
```



```
-- to this catalog.  
SET SYNC MAINTENANCE MODE ON;  
ALTER TABLE T ADD COLUMN C INTEGER;  
COMMIT WORK;  
SET SYNC MAINTENANCE MODE OFF;
```

## B.15.5 Return Values

For details on each error code, see the appendix titled Error Codes in the *IBM solidDB Administration Guide*.

**Table B.7. CREATE PUBLICATION Return Values**

Error Code	Description
13047	No privilege for operation. You do not have the privileges required to drop this publication or create a publication.
13120	The name is too long for the publication
25015	Syntax error: <i>error_message</i> , line <i>line_number</i>
25021	Database is not master or replica database. Publications can be created only in a master or replica database. (As a practical matter, they should only be created in a master database.)
25033	Publication <i>publication_name</i> already exists
25049	Referenced table <i>table_name</i> not found in subscription hierarchy
25061	Where condition for table <i>table_name</i> must refer to an outer table of the publication

## B.16 CREATE ROLE

```
CREATE ROLE role_name
```

### B.16.1 Usage

Creates a new user role.

## B.16.2 Example

```
CREATE ROLE GUEST_USERS;
```

## B.17 CREATE SCHEMA

```
CREATE SCHEMA schema_name
```

### B.17.1 Usage

Schemas are a collection of database objects, such as tables, views, indexes, events, triggers, sequences, and stored procedures for a database user. The default schema name is the user id. Note that with schemas, there is one default for each user. IBM solidDB's use of schemas conforms to the SQL standard.

The schema name is used to qualify a database object name. Database object names are qualified in all DML statements as:

```
catalog_name.schema_name.database_object_name
```

or

```
user_id.database_object_name
```

To logically partition a database, users can create a catalog before they create a schema. For details on creating a catalog, read Section B.11, “CREATE CATALOG”. Note that when creating a new database or converting an old database to a new format, users are prompted for a default catalog name.

To use schemas, a schema name must be created before creating the database object name (such as a table name or procedure name). However, a database object name can be created without a schema name. In such cases, database objects are qualified using `user_id` only.

You can specify the database object names in a DML statement explicitly by fully qualifying them or implicitly by setting the schema name context using:

```
SET SCHEMA schema_name
```

Creating a schema does not automatically make that schema the current default schema. If you have created a new schema and want your subsequent commands to execute within that schema, then you must also execute the SET SCHEMA statement. For example:

```
CREATE SCHEMA MySchema;  
CREATE TABLE t1; -- not in MySchema  
SET SCHEMA MySchema;  
CREATE TABLE t2; -- in MySchema
```

For more information about SET SCHEMA, see the description of the SET SCHEMA command in Section B.75, “SET”.

A schema can be dropped from a database using:

```
DROP SCHEMA schema_name
```

When dropping a schema name, all objects associated with the schema name must be dropped prior to dropping the schema.

A schema context can be removed using:

```
SET SCHEMA USER
```

Below are the rules for resolving schema names:

- A fully qualified name (*schema\_name.database\_object\_name*) does not need any name resolution, but will be validated.
- If a schema context is not set using SET SCHEMA, then all database object names are resolved always using the user id as the schema name.
- If the database object name cannot be resolved from the schema name, then the database object name is resolved from all existing schema names.
- If name resolution finds either zero matching or more than one matching database object name, then a IBM solidDB server issues a name resolution conflict error.

## B.17.2 Examples

```
-- Assume the userID is SMITH.
CREATE SCHEMA FINANCE;
CREATE TABLE EMPLOYEE (EMP_ID INTEGER);
SET SCHEMA FINANCE;
CREATE TABLE EMPLOYEE (ID INTEGER);
SELECT ID FROM EMPLOYEE;
-- In this case, the table is qualified to FINANCE.EMPLOYEE
SELECT EMP_ID FROM EMPLOYEE;
-- This will give an error as the context is with FINANCE and
-- table is resolved to FINANCE.EMPLOYEE

--The following are valid schema statements: one with a schema context,
--the other without.
SELECT ID FROM FINANCE.EMPLOYEE;
SELECT EMP_ID FROM SMITH.EMPLOYEE
--The following statement will resolve to schema SMITH without a schema
--context
SELECT EMP_ID FROM EMPLOYEE;
```

## B.18 CREATE SEQUENCE

```
CREATE [DENSE] SEQUENCE sequence_name
```

### B.18.1 Usage

Sequencer objects are objects that are used to get sequence numbers.

Using a dense sequence guarantees that there are no holes in the sequence numbers. The sequence number allocation is bound to the current transaction. If the transaction rolls back, then the sequence number allocations are also rolled back. The drawback of dense sequences is that the sequence is locked out from other transactions until the current transaction ends.

Using a sparse sequence guarantees uniqueness of the returned values, but they are not bound to the current transaction. If a transaction allocates a sparse sequence number and later rolls back, the sequence number is simply lost.

Sequence numbers are 8-byte values. Sequence values can be stored in `BIGINT`, `INT`, or `BINARY` data types. `BIGINT` is recommended. Sequence values stored in `INT` variables lose information because an 8-byte sequence number will not fit in a 4-byte `INT`. 8-byte `BINARY` values can store a complete sequence number, but `BINARY` values are not always as convenient to work with as integer data types.



## Note

Because a sequence number is an 8-byte number, storing it in a 4-byte integer (in a stored procedure or in an application program) will omit the highest four bytes. This will lead possibly to unwanted behavior after the sequence number goes beyond  $2^{31} - 1$  (=2147483647). Below is some sample code and the output that demonstrates this behavior:

```
CREATE SEQUENCE seq1;

-- Set the sequence number to 2^31 - 1,
-- then return that value and the "next" value (2^31).
"CREATE PROCEDURE set_seq1_to_2G
RETURNS (x INT, y INT)
BEGIN
DECLARE int1 INTEGER;
int1 := 2147483647;
EXEC SEQUENCE seq1 SET VALUE USING int1;
EXEC SEQUENCE seq1 CURRENT INTO x;
EXEC SEQUENCE seq1 NEXT INTO y;
END";

COMMIT WORK;

CALL set_seq1_to_2G();
```

The return values from the call are:

x	y
2147483647	-2147483648

The value for `x` is correct, but the value for `y` is a negative number instead of the correct positive number.

The advantage of using a sequencer object instead of a separate table is that the sequencer object is specifically fine-tuned for fast execution and requires less overhead than normal update statements.

Sequence values can be incremented and used within SQL statements. These constructs can be used in SQL:

```
sequence_name.CURRVAL  
sequence_name.NEXTVAL
```

Sequences can also be used inside stored procedures. The current sequence value can be retrieved using the following stored procedure statement:

```
EXEC SEQUENCE sequence_name.CURRENT INTO variable
```

The new sequence value can be retrieved using the following stored procedure statement:

```
EXEC SEQUENCE sequence_name.NEXT INTO variable
```

Sequence values can be set with the following stored procedure statement:

```
EXEC SEQUENCE sequence_name SET VALUE USING variable
```

Select access rights are required to retrieve the current sequence value. Update access rights are required to allocate new sequence values. These access rights are granted and revoked in the same way as table access rights.

## B.18.2 Examples

```
CREATE DENSE SEQUENCE SEQ1;  
INSERT INTO ORDER (id) VALUES (order_sequence.NEXTVAL);
```

## B.19 CREATE SYNC BOOKMARK

```
CREATE SYNC BOOKMARK bookmark_name
```

### B.19.1 Supported in

This requires IBM solidDB advanced replication.

## B.19.2 Usage

This statement creates a bookmark in a master database. Bookmarks represent a user-defined version of the database. It is a persistent snapshot of a IBM solidDB database, which provides a reference for performing specific synchronization tasks. Bookmarks are used typically to export data from a master for import into a replica using the EXPORT SUBSCRIPTION command. Exporting and importing data allows you to create a replica from a master more efficiently if you have databases larger than 2GB.

To create a bookmark, you must have administrative DBA privileges or SYS\_SYNC\_ADMIN\_ROLE. There is no limit to the number of bookmarks you can create in a database. A bookmark is created only in a master database. The system issues an error if you attempt to create a bookmark in a replica database.

If a table is set up for synchronization history with the ALTER TABLE SET SYNCHISTORY command, a bookmark retains history information for the table. For this reason, use the DROP SYNC BOOKMARK statement to drop bookmarks when they are not longer needed. Otherwise, extra history data will increase disk space usage.

When you create a new bookmark, the system associates other attributes, such as creator of the bookmark, creation data and time, and a unique bookmark ID. This metadata is maintained in the system table SYS\_SYNC\_BOOKMARKS. For a description of this table, refer to Section D.2.10, “SYS\_SYNC\_BOOKMARKS”.

## B.19.3 Usage in Master

Use the CREATE SYNC BOOKMARK statement to create a bookmark in a master database.

## B.19.4 Usage in Replica

The CREATE SYNC BOOKMARK statement cannot be used in a replica database.

## B.19.5 Example

```
CREATE SYNC BOOKMARK BOOKMARK_AFTER_DATALOAD;
```

## B.19.6 Return Values

For details on each error code, see the appendix titled Error Codes in the *IBM solidDB Administration Guide*.

**Table B.8. CREATE SYNC BOOKMARK Return Values**

Error Code	Description
25066	Bookmark already exists
13047	No privilege for operation

## B.20 CREATE TABLE

```
CREATE [ { [GLOBAL] TEMPORARY | TRANSIENT } ] TABLE base_table_name
(column_element [, column_element] ...) [STORE {MEMORY | DISK}]
```

```
base_table_name ::= base_table_identifier | schema_name.base_table_identifier |
catalog_name.schema_name.base_table_identifier
```

```
column_element ::= column_definition / table_constraint_definition
```

```
column_definition ::= column_identifier
data_type [DEFAULT literal | NULL] [NOT NULL]
[column_constraint_definition [column_constraint_definition] ...]
```

```
column_constraint_definition ::= [CONSTRAINT constraint_name]
UNIQUE | PRIMARY KEY |
REFERENCES ref_table_name [(referenced_columns)] |
CHECK (check_condition)
```

```
table_constraint_definition ::= [CONSTRAINT constraint_name]
UNIQUE (column_identifier [, column_identifier] ...) |
PRIMARY KEY (column_identifier [, column_identifier] ...) |
CHECK (check_condition) |
{FOREIGN KEY (column_identifier [, column_identifier] ...)
REFERENCES table_name [(referenced_columns)]
[referential_triggered_action] }
```

```
referential_triggered_action:: =
```



```
ON {UPDATE | DELETE} {CASCADE | SET NULL | SET DEFAULT |  
RESTRICT | NO ACTION}
```

## B.20.1 Usage

Tables are created through the CREATE TABLE statement. The CREATE TABLE statement requires a list of the columns created, the data types, and, if applicable, sizes of values within each column, in addition to other options, such as creating primary keys.

### Important

Always define a primary key when you create a table. If you do not define a primary key, IBM solidDB will create one automatically. This will lead to unexpected data order on the disk and may cause performance degradation. An appropriate primary key speeds up queries using the primary key.

Constraint definitions are available for both the column and table level. For the column level, constraints defined with NOT NULL specify that a non-null value is required for a column insertion. UNIQUE specifies that no two rows are allowed to have the same value. PRIMARY KEY ensures that the column(s), which is (are) a primary key, does not permit two rows to have the same value and does not permit any NULL values; PRIMARY KEY is thus equivalent to the combination of UNIQUE and NOT NULL. The REFERENCES clause with FOREIGN KEY specifies a table name and a list of columns for a referential integrity constraint. This means that when data is inserted or updated in this table, the data must match the values in the referenced tables and columns.

The CHECK keyword restricts the values that can be inserted into a column (for example, restricting the values with a specific integer range). When defined, the check constraint performs a validation check for any data that is inserted or updated in that column. If the data violates the constraint, then the modification is prohibited. For example:

```
CREATE TABLE table1 (salary DECIMAL CHECK (salary >= 0.0));
```

The check\_condition is a boolean expression that specifies the check constraints for the column. Check constraints are defined with the predicates >, <, =, <>, <=, >= and the keywords BETWEEN, IN, LIKE (which may contain wildcard characters), and IS [NOT] NULL. The expression (similar to the syntax of a WHERE clause) can be qualified with keywords AND and OR. For example:

```
...CHECK (coll = 'Y' OR coll = 'N')...  
...CHECK (last_name IS NOT NULL)...
```

Note that **UNIQUE** and **PRIMARY KEY** constraints can be defined at the column level or the table level. They also automatically create a unique index on the specified columns.

A foreign key is a column or group of columns within a table that refers to, or relates to, some other table through its values. The **FOREIGN KEY** is used to specify that the column(s) listed are foreign keys in this table. The **REFERENCES** keyword in the statement specifies the table and those column(s) that are references of the foreign key(s). Note that although column-level constraints can use a **REFERENCES** clause, only table-level constraints can use the **FOREIGN KEY ... REFERENCES** clause.

To use the **REFERENCES** constraint with **FOREIGN** keys, a foreign key must always include enough columns in its definition to uniquely identify a row in the referenced table. A foreign key must contain the same number and type (data type) of columns as the primary key in the referenced table as well as be in the same order; however, a foreign key can have different column names and default values than the primary key.

Note the following rules about constraints:

- The *check\_condition* cannot contain subqueries, aggregate functions, host variables, or parameters.
- Column check constraints can reference only the columns on which they are defined.
- Table check constraints can reference any columns in the table, that is if all columns in the table have been defined earlier in the statement.
- A table may have only one primary key constraint, but may have multiple unique constraints.
- The **UNIQUE** and **PRIMARY KEY** constraints in the **CREATE TABLE** statement can be used to create indexes. However, if you use the **ALTER TABLE** statement, keep in mind that a column cannot be dropped if it is part of a unique or primary key. You may want to use the **CREATE INDEX** statement to create an index instead because the index will have a name and you can drop it. The **CREATE INDEX** statement also offers some additional features, such as the ability to create non-unique indexes and to specify if the indexes are sorted in ascending or descending order.
- The referential integrity rules for persistent, transient, and temporary table types are different.
  - A temporary table may reference another temporary table, but may not reference any other type of table (i.e. transient or persistent). No other type of table may reference a temporary table.
  - Transient tables may reference other transient tables and may reference persistent tables. They may not reference temporary tables. Neither temporary tables nor persistent tables may reference a transient table.

In a disk-based table, the maximum size of a row (excluding BLOBs) is approximately 1/3 of the page size. In an in-memory table, the maximum size of a row (including BLOBs) is approximately the page size. (There is a small amount of overhead used in both disk-based and in-memory pages, so not quite all of the page is available for user data.) The default page size is 8kB. For more information about page size, see the description of the `solid.ini` configuration parameter `BlockSize` in *IBM solidDB Administration Guide*.

The server does not use simple rules to determine BLOB storage, but as a general rule each BLOB occupies 256 bytes from the page where the row resides, and the rest of the BLOB goes to separate BLOB pages. If the BLOB is shorter than 256 bytes, then it is stored entirely in the main disk page, not BLOB pages.

Each row is limited to 1000 columns.

The `STORE` clause indicates whether the table should be stored in memory or on disk. (This clause is only available in IBM solidDB main memory engine.) For more information about the `STORE` clause, see *IBM solidDB In-Memory Database User Guide*.

In-memory tables may be persistent (normal) tables, temporary tables, or transient tables. For a detailed discussion of temporary tables and transient tables, see *IBM solidDB In-Memory Database User Guide*.

All temporary tables and transient tables must be in-memory tables. You do not need to specify the "STORE MEMORY" clause; temporary tables and transient tables will automatically be created as in-memory tables if you omit the `STORE` clause. (For temporary tables and transient tables, the `solid.ini` configuration parameter `DefaultStoreIsMemory` is ignored.) You will get an error if you try to explicitly create temporary tables or transient tables as disk-based tables, e.g. if you execute a command similar to the following:

```
CREATE TEMPORARY TABLE t1 (i INT) STORE DISK; --Wrong!
```

The keyword "GLOBAL" is included to comply with the SQL:1999 standard for temporary tables. In IBM solidDB, all temporary tables are global, whether or not the `GLOBAL` keyword is used.

### Interactions with Configuration Parameters

The storage location (disk or memory) in the `CREATE TABLE` statement takes precedence over the storage location specified by the `DefaultStoreIsMemory` parameter in the `solid.ini` configuration file.

## B.20.2 Example

```
CREATE TABLE DEPT (DEPTNO INTEGER NOT NULL, DNAME VARCHAR, PRIMARY KEY(DEPTNO));
CREATE TABLE DEPT2 (DEPTNO INTEGER NOT NULL PRIMARY KEY, DNAME VARCHAR);
CREATE TABLE DEPT3 (DEPTNO INTEGER NOT NULL UNIQUE, DNAME VARCHAR);
CREATE TABLE DEPT4 (DEPTNO INTEGER NOT NULL, DNAME VARCHAR, UNIQUE(DEPTNO));
```

```
CREATE TABLE EMP (DEPTNO INTEGER, ENAME VARCHAR, FOREIGN KEY (DEPTNO)
REFERENCES DEPT (DEPTNO)) STORE DISK;
CREATE TABLE EMP2 (DEPTNO INTEGER, ENAME VARCHAR, CHECK (ENAME IS NOT NULL),
FOREIGN KEY (DEPTNO) REFERENCES DEPT (DEPTNO)) STORE MEMORY;
CREATE GLOBAL TEMPORARY TABLE T1 (C1 INT);
CREATE TRANSIENT TABLE T2 (C1 INT);
```

## B.21 CREATE TRIGGER

```
CREATE TRIGGER trigger_name ON table_name time_of_operation
    triggering_event [REFERENCING column_reference]
    BEGIN trigger_body END
```

where:

```
trigger_name ::= literal
table_name ::= literal
time_of_operation ::= BEFORE | AFTER
triggering_event ::= INSERT | UPDATE | DELETE
column_reference ::= {OLD | NEW} column_name [AS] col_identifier
    [, REFERENCING column_reference ]
```

```
trigger_body ::=
    [declare_statement;...]
    [trigger_statement;...]
```

```
old_column_name ::= literal
new_column_name ::= literal
col_identifier ::= literal
```



### Note

This appendix is intended to provide a quick reference to using IBM solidDB SQL commands. For details on when and how to use triggers, read Section 4.7, “Triggers and Procedures”.

### B.21.1 Usage

A trigger provides a mechanism for executing a series of SQL statements when a particular action (an INSERT, UPDATE, or DELETE) occurs. The “body” of the trigger contains the SQL statement(s) that the user wants

to execute. The body of the trigger is written using the Stored Procedure Language (which is described in more detail in section about the CREATE PROCEDURE statement).

You may create one or more triggers on a table, with each trigger defined to activate on a specific INSERT, UPDATE, or DELETE command. When a user modifies data within the table, the trigger that corresponds to the command is activated.

You can only use inline SQL or stored procedures with triggers. If you use a stored procedure in the trigger, then the procedure must be created with the CREATE PROCEDURE command. A procedure invoked from a trigger body can invoke other triggers.

To create a trigger, you must be a DBA or owner of the table on which the trigger is being defined.

Triggers are created with the statement

```
CREATE TRIGGER name body
```

and dropped from the system catalog with the statement

```
DROP TRIGGER name
```

Triggers are disabled by using the statement

```
ALTER TRIGGER name
```

When you disable a trigger defined on a table, a IBM solidDB server ignores the trigger when an activating DML statement is issued. With this command, you can also enable a trigger that is currently inactive.



### **Note**

Following is a brief summary of the keywords and clauses used in the CREATE TRIGGER command. For more information on usage, read Chapter 4, *Stored Procedures, Events, Triggers, and Sequences*.

## **B.21.2 Trigger Name**

The *trigger\_name* identifies the trigger and can contain up to 254 characters.

## B.21.3 BEFORE | AFTER Clause

The BEFORE | AFTER clause specifies whether to execute the trigger before or after the invoking DML statement. In some circumstances, the BEFORE and AFTER clauses are interchangeable. However, there are some situations where one clause is preferred over the other.

- It is more efficient to use the BEFORE clause when performing data validation, such as domain constraint and referential integrity checking.
- When you use the AFTER clause, table rows which become available due to the invoking DML statement are processed. Conversely, the AFTER clause also confirms data deletion after the invoking DELETE statement.

You can define up to six triggers per table, one for each combination of action (INSERT, UPDATE, DELETE) and time (BEFORE and AFTER):

- BEFORE INSERT
- BEFORE UPDATE
- BEFORE DELETE
- AFTER INSERT
- AFTER UPDATE
- AFTER DELETE

The following example shows trigger trig01 defined BEFORE INSERT ON table1.

```
"CREATE TRIGGER TRIG01 ON table1
  BEFORE INSERT
  REFERENCING NEW COL1 AS NEW_COL1
BEGIN
  EXEC SQL PREPARE CUR1
        INSERT INTO T2 VALUES (?);
  EXEC SQL EXECUTE CUR1 USING (NEW_COL1);
  EXEC SQL CLOSE CUR1;
```

```
EXEC SQL DROP CUR1;  
END "
```

Following are examples (including implications and advantages) of using the BEFORE and AFTER clause of the CREATE TRIGGER command for each DML operation:

- UPDATE Operation

The BEFORE clause can verify that modified data follows integrity constraint rules before processing the UPDATE. If the REFERENCING NEW AS *new\_column\_identifier clause* is used with the BEFORE UPDATE clause, then the updated values are available to the triggered SQL statements. In the trigger, you can set the default column values or derived column values before performing an UPDATE.

The AFTER clause can perform operations on newly modified data. For example, after a branch address update, the sales for the branch can be computed.

If the REFERENCING OLD AS *old\_column\_identifier clause* is used with the AFTER UPDATE clause, then the values that existed prior to the invoking update are accessible to the triggered SQL statements.

- INSERT Operation

The BEFORE clause can verify that new data follows integrity constraint rules before performing an INSERT. Column values passed as parameters are visible to the triggered SQL statements but the inserted rows are not. In the trigger, you can set default column values or derived column values before performing an INSERT.

The AFTER clause can perform operations on newly inserted data. For example, after insertion of a sales order, the total order can be computed to see if a customer is eligible for a discount.

Column values are passed as parameters and inserted rows are visible to the triggered SQL statements.

- DELETE Operation

The BEFORE clause can perform operations on data about to be deleted. Column values passed as parameters and inserted rows that are about to be deleted are visible to the triggered SQL statements.

The AFTER clause can be used to confirm the deletion of data. Column values passed as parameters are visible to the triggered SQL statements. Please note that the deleted rows are visible to the triggering SQL statement.

## B.21.4 INSERT | UPDATE | DELETE Clause

The INSERT | UPDATE | DELETE clause indicates the trigger action when a user action (INSERT, UPDATE, DELETE) is attempted.

Statements related to processing a trigger occur first before commits and autocommits from the invoking DML (INSERT, UPDATE, DELETE) statements on tables. If a trigger body or a procedure called within the trigger body attempts to execute a COMMIT or ROLLBACK, a IBM solidDB server returns an appropriate run-time error.

INSERT specifies that the trigger is activated by an INSERT on the table. Loading *n* rows of data is considered as *n* inserts.



### Note

There may be some performance impact if you try to load the data with triggers enabled. Depending on your business need, you may want to disable the triggers before loading and enable them after loading. For details, see Section B.6, “ALTER TRIGGER”.

DELETE specifies that the trigger is activated by a DELETE on the table.

UPDATE specifies that the trigger is activated by an UPDATE on the table. Note the following rules for using the UPDATE clause:

- Within the REFERENCES clause of a trigger, a column may be referenced (aliased) no more than once in the BEFORE sub-clause and once in the AFTER sub-clause. Also, if the column is referenced in both the BEFORE and AFTER sub-clauses, the column's alias must be different in each sub-clause.
- A IBM solidDB server allows for recursive update to the same table and does not prohibit recursive updates to the same row.

A IBM solidDB server does not detect situations where the actions of different triggers cause the same data to be updated. For example, assume there are two update triggers (one that is a BEFORE trigger and one that is an AFTER trigger) on different columns, Col1 and Col2, of table Table1. When an update is attempted on all the columns of Table1, the two triggers are activated. Both triggers call stored procedures which update the same column, Col3, of a second table, Table2. The first trigger updates Table2.Col3 to 10 and the second trigger updates Table2.Col3 to 20.

Likewise, a IBM solidDB server does not detect situations where the result of an UPDATE which activates a trigger conflicts with the actions of the trigger itself. For example, consider the following SQL statement:



```
UPDATE t1 SET c1 = 20 WHERE c3 = 10;
```

If the trigger is activated by this UPDATE then calls a procedure that contains the following SQL statement, the procedure overwrites the result of the UPDATE that activated the trigger:

```
UPDATE t1 SET c1 = 17 WHERE c1 = 20;
```



### Note

The above example can lead to recursive trigger execution, which you should try to avoid.

## B.21.5 Table\_name

The *table\_name* is the name of the table on which the trigger is created. IBM solidDB server allows you to drop a table that has dependent triggers defined on it. When you drop a table all dependent objects including triggers are dropped. Be aware that you may still get run-time errors. For example, assume you create two tables A and B. If a procedure SP-B inserts data into table A, and table A is then dropped, a user will receive a run-time error if table B has a trigger which invokes SP-B.

## B.21.6 Trigger\_body

The *trigger\_body* contains the statement(s) to be executed when a trigger fires. The *trigger\_body* definition is identical to the stored procedure definition. Read Section B.14, “CREATE PROCEDURE” for details on creating a stored procedure body.

Note that it is syntactically valid, although not useful, to create a trigger with an empty body.

A trigger body may also invoke any procedure registered with a IBM solidDB server. IBM solidDB procedure invocation rules follow standard procedure invocation practices.

You must explicitly check for business logic errors and raise an error.

## B.21.7 REFERENCING Clause

This clause is optional when creating a trigger on an INSERT/UPDATE/DELETE operation. It provides a way to reference the current column identifiers in the case of INSERT and DELETE operations, and both the old column identifier and the new updated column identifier by aliasing the column(s) on which an UPDATE operation occurs.

You must specify the OLD or NEW *column\_identifier* to access it. A IBM solidDB server does not provide access to the *column\_identifier* unless you define it using the REFERENCING subclause.

## B.21.8 {OLD | NEW} column\_name AS col\_identifier

This subclause of the REFERENCING clause allow you to reference the values of columns both before and after an UPDATE operation. It produces a set of old and new column values which can be passed to a stored procedure; once passed, the procedure contains logic (for example, domain constraint checking) used to determine these parameter values.

Use the OLD AS clause to alias the table's old identifier as it exists before the UPDATE. Use the NEW AS clause to alias the table's new identifier as it exists after the UPDATE.

If you reference both the old and new values of the same column, you must use different *column\_identifiers*.

Each column that is referenced as NEW or OLD should have a separate REFERENCING subclause.

The statement atomicity in a trigger is such that operations made in a trigger are visible to the subsequent SQL statements inside the trigger. For example, if you execute an INSERT statement in a trigger and then also perform a select in the same trigger, then the inserted row is visible.

In the case of AFTER trigger, an inserted row or an updated row is visible in the AFTER insert trigger, but a deleted row cannot be seen for a select performed within the trigger. In the case of a BEFORE trigger, an inserted or updated row is invisible within the trigger and a deleted row is visible. In the case of an UPDATE, the pre-update values are available in a BEFORE trigger.

The table below summarizes the statement atomicity in a trigger, indicating whether the row is visible to the SELECT statement in the trigger body.

**Table B.9. Statement Atomicity in a Trigger**

<b>Operation</b>	<b>BEFORE TRIGGER</b>	<b>AFTER TRIGGER</b>
INSERT	row is invisible	row is visible
UPDATE	previous value is visible	new value is visible
DELETE	row is visible	row is invisible

## B.21.9 Triggers Comments and Restrictions

- To use the stored procedure that a trigger calls, provide the catalog, schema/owner and name of the table on which the trigger is defined and specify whether to enable or disable the triggers on the table. For more details on stored procedures, read Chapter 4, *Stored Procedures, Events, Triggers, and Sequences*.
- To create a trigger on a table, you must have DBA authority or be the owner of the table on which the trigger is being defined.
- You can define, by default, up to one trigger for each combination of table, action (INSERT, UPDATE, DELETE) and time (BEFORE and AFTER). This means there can be a maximum of six triggers per table.



### Note

The triggers are applied to each row. This means that if there are ten inserts, a trigger is executed ten times.

- You cannot define triggers on a view (even if the view is based on a single table).
- You cannot alter a table that has a trigger defined on it when the dependent columns are affected.
- You cannot create a trigger on a system table.
- You cannot execute triggers that reference dropped or altered objects. To prevent this error:
  - Recreate any referenced object that you drop.
  - Restore any referenced object you changed back to its original state (known by the trigger).
- You can use reserved words in trigger statements if they are enclosed in double quotes. For example, the following CREATE TRIGGER statement references a column named "data" which is a reserved word.

```
"CREATE TRIGGER TRIG1 ON TMPT BEFORE INSERT
REFERENCING NEW "DATA" AS NEW_DATA
BEGIN
END"
```

## Setting the Maximum Number of Nested Triggers

Triggers can invoke other triggers or a trigger can invoke itself (or recursive triggers). You can nest triggers up to 16 levels deep. The maximum number of nested triggers is set in the *MaxNestedTriggers* parameter in the *SQL* section of the *solid.ini* configuration file:

```
[SQL]
MaxNestedTriggers=n
```

where *n* is the maximum number of nested triggers.

The default is 16 triggers.

## Setting the Triggers Cache

Triggers are cached in a separate cache in the IBM solidDB server; each user has a separate cache for triggers. As the triggers are executed, the trigger procedure logic is cached in the trigger cache and is resumed when the trigger is executed again.

The cache size is set in the *TriggerCache* parameter in the *SQL* section of the *solid.ini* configuration file:

```
[SQL]
TriggerCache=n
```

where *n* is the number of triggers that is reserved for the cache.

## Checking for Errors

At times, it is possible to receive an error in executing a trigger. The error may be due to execution of SQL statements or business logic. If a trigger returns an error, it causes its invoking DML command to fail. To automatically return errors during the execution of a DML statement, you must use the *WHENEVER SQLERROR ABORT* statement in the trigger body. Otherwise, errors must be checked explicitly within the trigger body after each procedure call or SQL statement.

For any errors in the user-written business logic as part of the trigger body, users can receive errors in a procedure variable using the SQL statement:

```
RETURN SQLERROR error_string
```

or

```
RETURN SQLERROR char_variable
```

The error is returned in the following format:

User error: *error\_string*

If a user does not specify the RETURN SQLERROR statement in the trigger body, then all trapped SQL errors are raised with a default error\_string determined by the system. For details, see the appendix on Error Codes in *IBM solidDB Administration Guide*.



## Note

Triggered SQL statements are a part of the invoking transaction. If the invoking DML statement fails due to either the trigger or another error that is generated outside the trigger, all SQL statements within the trigger are rolled back along with the failed invoking DML command.

Below is an example of using WHENEVER SQLERROR ABORT to make sure that the trigger catches an error in a stored procedure that it calls.

```
-- If you return an SQLERROR from a stored procedure, the error is
-- displayed. However, if the stored procedure is called from inside
-- a trigger, then the error is not displayed unless you use the
-- SQL statement WHENEVER SQLERROR ABORT.
```

```
CREATE TABLE table1 (x INT);
CREATE TABLE table2 (x INT);
```

```
"CREATE PROCEDURE stprocl
BEGIN
    RETURN SQLERROR 'Here is an error!';
END";
COMMIT WORK;
```

```
"CREATE TRIGGER displays_error ON table1 BEFORE INSERT
BEGIN
    EXEC SQL WHENEVER SQLERROR ABORT;
    EXEC SQL EXECDIRECT CALL stprocl();
END";
COMMIT WORK;
```

```
"CREATE TRIGGER does_not_display_error ON table2 BEFORE INSERT
BEGIN
    EXEC SQL EXECDIRECT CALL stprocl();
END";
```

```
COMMIT WORK;

-- This shows that the error is returned if you execute the stored procedure.
CALL stprocl();

-- Displays an error because the trigger had WHENEVER SQL ERROR ABORT.
INSERT INTO table1 (x) values (1);
-- Does not display an error.
INSERT INTO table2 (x) values (1);
```

### Triggers Stack Functions

The following functions may be used to analyze the current contents of the trigger stack:

`TRIG_COUNT()` returns the number of triggers in the trigger stack. This includes the current trigger. The return value is an integer.

`TRIG_NAME(n)` returns the nth trigger name in the trigger stack. The first trigger position or offset is zero.

`TRIG_SCHEMA(n)` returns the nth trigger schema name in the trigger stack. The first trigger position or offset is zero. The return value is a string.

### Example

```
"CREATE TRIGGER TRIGGER_BI ON TRIGGER_TEST
  BEFORE INSERT
  REFERENCING NEW BI AS NEW_BI
BEGIN
  EXEC SQL PREPARE BI INSERT INTO TRIGGER_OUTPUT VALUES (
    'BI', TRIG_NAME(0), TRIG_SCHEMA(0));
  EXEC SQL EXECUTE BI;
```

```
    SET NEW_BI = 'TRIGGER_BI';  
END";
```

## B.22 CREATE USER

```
CREATE USER user_name IDENTIFIED BY password
```

### B.22.1 Usage

Creates a new user with a given password.

### B.22.2 Example

```
CREATE USER HOBBS IDENTIFIED BY CALVIN;
```

## B.23 CREATE VIEW

```
CREATE VIEW viewed_table_name [( column_identifier  
    [,column_identifier ]... )]  
AS query-specification
```

### B.23.1 Usage

A view can be viewed as a virtual table; that is, a table that does not physically exist, but rather is formed by a query specification against one or more tables.

### B.23.2 Example

```
CREATE VIEW TEST_VIEW
```

```
(VIEW_I, VIEW_C, VIEW_ID)  
AS SELECT I, C, ID FROM TEST;
```

## B.24 DELETE

```
DELETE FROM table_name [WHERE search_condition]
```

### B.24.1 Usage

Depending on your search condition, the specified row(s) will be deleted from a given table.

### B.24.2 Example

```
DELETE FROM TEST WHERE ID = 5;  
DELETE FROM TEST;
```

## B.25 DELETE (positioned)

```
DELETE FROM table_name WHERE CURRENT OF cursor_name
```

### B.25.1 Usage

The positioned DELETE statement deletes the current row of the cursor.

### B.25.2 Example

```
DELETE FROM TEST WHERE CURRENT OF MY_CURSOR;
```

## B.26 DROP CATALOG

```
DROP CATALOG catalog_name [CASCADE | RESTRICT]
```

### B.26.1 Usage

The DROP CATALOG statement drops the specified catalog from the database.



If you use the `RESTRICT` keyword, or if you do not specify either `RESTRICT` or `CASCADE`, then you must drop all database objects in the catalog before you drop the catalog itself.

If you use the `CASCADE` keyword, then if the catalog contains database objects (such as tables), those will automatically be dropped. If you use the `CASCADE` keyword, and if objects in other catalogs reference an object in the catalog being dropped, then the references will automatically be resolved by dropping those referencing objects or updating them to eliminate the reference.

Only the creator of the database or users having `SYS_ADMIN_ROLE` (i.e. DBA) have privileges to create or drop a catalog. Even the creator of a catalog cannot drop that catalog if she loses `SYS_ADMIN_ROLE` privileges.

### B.26.2 Example

```
DROP CATALOG C1;  
DROP CATALOG C2 CASCADE;  
DROP CATALOG C3 RESTRICT;
```

## B.27 DROP EVENT

```
DROP EVENT event_name  
DROP EVENT [[catalog_name.]schema_name.]event_name
```

### B.27.1 Usage

The `DROP EVENT` statement removes the specified event from the database.

### B.27.2 Example

```
DROP EVENT EVENT_TEST;  
-- Using catalog, schema, and event name
```

```
DROP EVENT
HR_database.smith_schema.event1;
```

## B.28 DROP INDEX

```
DROP INDEX index_name
DROP INDEX[[catalog_name.]schema_name.]index_name
```

### B.28.1 Usage

The DROP INDEX statement removes the specified index from the database.

### B.28.2 Example

```
DROP INDEX test_index;
-- Using catalog, schema, and index name
DROP INDEX bank_accounts.bankteller.first_name_index;
```

## B.29 DROP MASTER

```
DROP MASTER master_name
```

### B.29.1 Usage

This statement drops the master database definitions from a replica database. After this operation, the replica cannot synchronize with the master database.



#### Note

1. Unregistering the replica is the preferred way to quit using a master database. The DROP MASTER statement is used only when the MESSAGE APPEND UNREGISTER REPLICAS statement is unable to be executed. For details on unregistering a replica, see Section B.53, “MESSAGE APPEND”.
2. IBM solidDB requires that autocommit be set OFF when using the DROP MASTER command.
3. If *master\_name* is a reserved word, it must be enclosed in double quotes.

## B.29.2 Usage in Master

This statement can not be used in a master.

## B.29.3 Usage in Replica

This statement is used in replica to drop a master from replica.

## B.29.4 Examples

```
DROP MASTER "MASTER";  
DROP MASTER MY_MASTER;
```

## B.29.5 Return Values

For details on each error code, see the appendix titled Error Codes in *IBM solidDB Administration Guide*.

**Table B.10. DROP MASTER Return Values**

Error code	Description
13047	No privilege for operation
25007	Master <i>master_name</i> not found
25019	Database is not a replica database
25056	Autocommit not allowed
25065	Unfinished message <i>message_name</i> found for master <i>master_name</i>

## B.30 DROP PROCEDURE

```
DROP PROCEDURE procedure_name  
DROP PROCEDURE [[catalog_name.]schema_name.]procedure_name
```

### B.30.1 Usage

The DROP PROCEDURE statement removes the specified procedure from the database.

## B.30.2 Example

```
DROP PROCEDURE PROCTEST;  
-- Using catalog, schema, and procedure name  
DROP PROCEDURE telecomm_database.technician1.add_new_IP_address;
```

## B.31 DROP PUBLICATION

```
DROP PUBLICATION publication_name
```

### B.31.1 Usage

This statement drops a publication definition in the master database. All subscriptions to the dropped publication are automatically dropped as well.

### B.31.2 Usage in Master

Dropping a publication from the master will remove it and replicas will not be able to refresh from it.

### B.31.3 Usage in Replica

Using this statement in a replica will drop the publication definition from the replica if you defined a publication on the replica. (However, it is not necessary or useful to define publications in replica databases, so you should not need to use either CREATE PUBLICATION or DROP PUBLICATION in a replica.)

### B.31.4 Example

```
DROP PUBLICATION customers_by_area;
```

### B.31.5 Return Values

For details on each error code, see the appendix titled Error Codes in *IBM solidDB Administration Guide*.

**Table B.11. DROP PUBLICATION Return Values**

Error Code	Description
25010	Publication <i>publication_name</i> not found.

Error Code	Description
13111	Ambiguous entity name name

## B.32 DROP PUBLICATION REGISTRATION

DROP PUBLICATION *publication\_name* REGISTRATION

### B.32.1 Supported in

This requires IBM solidDB advanced replication.

### B.32.2 Usage

This statement drops a registration for a publication in the replica database. The publication definition remains on the master database, but a user will be unable to refresh from the publication. All subscriptions to the registered publication are automatically dropped as well.

### B.32.3 Usage in Master

This statement is not used in a master database.

### B.32.4 Usage in Replica

Using this statement in a replica will drop the registration for the publication in the replica. All subscriptions and their data to this publication are dropped automatically.

### B.32.5 Example

```
DROP PUBLICATION customers_by_area REGISTRATION;
```

### B.32.6 Return Values

For details on each error code, see the appendix titled Error Codes in *IBM solidDB Administration Guide*.

**Table B.12. DROP PUBLICATION REGISTRATION Return Values**

Error Code	Description
13047	No privilege for operation
25019	Database is not a replica database
25025	Node name not defined
25071	Not registered to publication <i>publication_name</i>

## B.33 DROP REPLICA

DROP REPLICA *replica\_name*

### B.33.1 Supported in

This requires IBM solidDB advanced replication.

### B.33.2 Usage

This statement drops a replica database from the master database. After this operation, the dropped replica cannot synchronize with the master database.

#### Note

1. Unregistering the replica is the preferred way to quit using a replica database. The DROP REPLICA statement is used only when the MESSAGE APPEND UNREGISTER REPLICA statement is unable to be executed. For details on unregistering a replica, see Section B.53, “MESSAGE APPEND”.
2. IBM solidDB requires that autocommit be set OFF when using the DROP REPLICA statement.
3. If *replica\_name* is a reserved word, it should be enclosed in double quotes.

### B.33.3 Usage in Master

Use this statement in the master to drop a replica from master.

## B.33.4 Usage in Replica

This statement cannot be used in replica.

## B.33.5 Example

```
DROP REPLICA salesman_smith ;  
DROP REPLICA "REPLICA" ;
```

## B.33.6 Return Values

For details on each error code, see the appendix titled Error Codes in *IBM solidDB Administration Guide*.

**Table B.13. DROP REPLICA Return Values**

Error code	Description
13047	No privilege for operation
25009	Replica <i>replica_name</i> not found
25020	Database is not a master database
25056	Autocommit not allowed
25064	Unfinished message <i>message_name</i> found for replica <i>replica_name</i>

## B.34 DROP ROLE

```
DROP ROLE role_name
```

### B.34.1 Usage

The DROP ROLE statement removes the specified role from the database.

## B.34.2 Example

```
DROP ROLE GUEST_USERS;
```

## B.35 DROP SCHEMA

```
DROP SCHEMA schema_name [CASCADE | RESTRICT]  
DROP SCHEMA [catalog_name.] schema_name [CASCADE | RESTRICT]
```

### B.35.1 Usage

The DROP SCHEMA statement drops the specified schema from the database. If you use the keyword RESTRICT, or if you do not specify either RESTRICT or CASCADE, then all the objects associated with the specified *schema\_name* must be dropped prior to using this statement. If you use the keyword CASCADE, then all the database objects (such as tables) within the specified schema will be dropped automatically.

If you use the CASCADE keyword, and if objects in other schemas reference an object in the schema being dropped, those references will automatically be resolved by dropping those referencing objects or updating them to eliminate the reference.

### B.35.2 Examples

```
DROP SCHEMA finance;  
DROP SCHEMA finance CASCADE;  
DROP SCHEMA finance RESTRICT;  
DROP SCHEMA forecasting_db.securities_schema CASCADE;
```

## B.36 DROP SEQUENCE

```
DROP SEQUENCE sequence_name  
DROP SEQUENCE [[catalog_name.] schema_name.] sequence_name
```

### B.36.1 Usage

The DROP SEQUENCE statement removes the specified sequence from the database.



## B.36.2 Examples

```
DROP SEQUENCE SEQ1;  
-- Using catalog, schema, and sequence name  
DROP SEQUENCE bank_db.checking_acct_schema.account_num_seq;
```

## B.37 DROP SUBSCRIPTION

In replica:

```
DROP SUBSCRIPTION publication_name [{(parameter_list) | ALL}]  
    [COMMITBLOCK number_of_rows] [OPTIMISTIC | PESSIMISTIC]
```

In master:

```
DROP SUBSCRIPTION publication_name [{(parameter_list) | ALL}]  
    REPLICA replica_name
```

### B.37.1 Supported in

This command requires IBM solidDB advanced replication.

### B.37.2 Usage

Data that is no longer needed in a replica database can be deleted from the replica database by dropping the subscription that was used to retrieve the data from the master database.



#### Note

IBM solidDB requires that autocommit be set OFF when dropping subscriptions.

By default, the data of a subscription is deleted in one transaction. If the amount of data is large, for example, tens of thousands of rows, it is recommended that the COMMITBLOCK be defined. When using the COMMITBLOCK option the data is deleted in more than one transaction. This ensures good performance for the operation.

In a replica, you can define the DROP SUBSCRIPTION statement to use table-level pessimistic locking when it is initially executed. If the PESSIMISTIC mode is specified, all other concurrent access to the table affected

is blocked until the drop has completed. Otherwise, if the optimistic mode is used, the `DROP SUBSCRIPTION` may fail due to a concurrency conflict.

A subscription can be dropped also from the master database. In this case, the replica name is included in the command. Names of all replicates that have been registered in the master database can be found in the `SYS_SYNC_REPLICAS` table. This operation deletes only the internal information about the subscription for this replica. The actual data in the replica is kept.

Dropping a subscription from the master is useful when a replica is no longer using the subscription and the replica has not dropped the subscription itself. Dropping old subscriptions releases old history data from the database. This history data is automatically deleted from the master databases after dropping the subscription.

If a replica's subscription has been dropped in the master database, the replica will receive the full data in the next refresh.

If a subscription is dropped in this case, `DROP SUBSCRIPTION` also drops the publication registration if the last subscription to the publication was dropped. Otherwise, registration must be dropped explicitly using the `DROP PUBLICATION REGISTRATION` statement or `MESSAGE APPEND UNREGISTER PUBLICATION`.

You can define the `DROP SUBSCRIPTION` statement to use table-level pessimistic locking when it is initially executed. If the `PESSIMISTIC` mode is specified, all other concurrent access to the tables affected is blocked until the import has completed. Otherwise, if the optimistic mode is used, the `DROP SUBSCRIPTION` may fail due to a concurrency conflict.

When a transaction acquires an exclusive lock to a table, the `TableLockWaitTimeout` parameter setting in the `[General]` section of the `solid.ini` configuration file determines the transaction's wait period until the exclusive or shared lock is released. For details, see the description of this parameter in *IBM solidDB Administration Guide*.

## B.37.3 Usage in Master

Use this statement to drop a subscription for a specified replica.

## B.37.4 Usage in Replica

Use this statement to drop a subscription from this replica.

## B.37.5 Example

Drop subscription from a master database

```
DROP SUBSCRIPTION customers_by_area('south')
FROM REPLICA salesman_joe
```

### *Return Values*

For details on each error code, see the appendix titled Error Codes in *IBM solidDB Administration Guide*.

**Table B.14. DROP SUBSCRIPTION Return Values**

<b>Error Code</b>	<b>Description</b>
13047	No privileges for operation
25004	Dynamic parameters are not supported
25009	Replica <i>replica_name</i> not found
25010	Publication <i>publication_name</i> not found
25019	Database is not a replica database
25020	Database is not a master database
25041	Subscription to publication <i>publication_name</i> not found
25056	Autocommit not allowed

## **B.38 DROP SYNC BOOKMARK**

```
DROP SYNC BOOKMARK bookmark_name
```

### **B.38.1 Supported in**

This command requires IBM solidDB advanced replication.

### **B.38.2 Usage**

This statement drops a bookmark defined on a master database. To drop a bookmark, you must have administrative privileges DBA or SYS\_SYNC\_ADMIN\_ROLE. Bookmarks are typically used when exporting data to a file. After a file is successfully imported to a replica from the master database, it is recommended that you drop the bookmark that you used to export the data to a file.

If a bookmark remains, then all subsequent changes to data on the master including deletes and updates are tracked on the master database for each bookmark to facilitate incremental refreshes.

If you do not drop bookmarks, the history information takes up disk space and unwanted disk I/O is incurred, as well, for each bookmark registered in the master database. This may result in performance degradation.



### Caution

Bookmarks should only be dropped after the exported data is imported into all intended replicas and after all the replicas have synchronized at least once. Be sure to drop a bookmark only when you no longer have replicas to import and those replicas have refreshed once from the publication after the import.

When dropping bookmarks, IBM solidDB uses the following rules to delete history records:

- Finds the oldest REFRESH delivered to any replica on that table
- Finds the oldest bookmark
- Determines which is older, the oldest REFRESH or oldest bookmark
- Deletes all rows from history up to what it determines is older, the oldest REFRESH or oldest bookmark.

## B.38.3 Usage in Master

Use the DROP SYNC BOOKMARK statement to drop a bookmark in a master database.

## B.38.4 Usage in Replica

The DROP SYNC BOOKMARK statement cannot be used in a replica database.

## B.38.5 Example

```
DROP SYNC BOOKMARK new_database;  
DROP SYNC BOOKMARK database_after_dataload;
```

## B.38.6 Return Values

For details on each error code, see the appendix titled Error Codes in *IBM solidDB Administration Guide*.

**Table B.15. DROP SYNC BOOKMARK Return Values**

Error Code	Description
25067	Synchronizer bookmark <i>bookmark_name</i> not found
13047	No privilege for operation

## B.39 DROP TABLE

```
DROP TABLE base_table_name [CASCADE [CONSTRAINTS]]
DROP TABLE [[catalog_name.] schema_name.] table_name [CASCADE
[CONSTRAINTS]]
```



### Note

Objects are usually dropped with drop behavior RESTRICT. There are some exceptions, however, including:

1. If your table has a synchronization history table, that synchronization history table will be dropped automatically. (IBM solidDB 3.7 and later.)
2. If a table has indexes on it, you do not need to drop the indexes first; they will be dropped automatically when the table is dropped.

### B.39.1 Usage

The DROP TABLE statement removes the specified table from the database.

### B.39.2 Examples

```
DROP TABLE table1;
-- Using catalog, schema, and table name
DROP TABLE domains_db.demand_schema.bad_address_table;
```

```
--remove foreign key constraints in referencing tables  
DROP TABLE table2 CASCADE CONSTRAINTS;
```

# B.40 DROP TRIGGER

```
DROP TRIGGER trigger_name  
DROP TRIGGER [[catalog_name.] schema_name.] trigger_name
```

## B.40.1 Usage

Drops (or deletes) a trigger defined on a table from the system catalog.

You must be the owner of a table, or a user with DBA authority, to delete a trigger from the table.

## B.40.2 Examples

```
DROP TRIGGER update_acct_balance;  
-- Using schema and trigger name  
DROP TRIGGER savings_accounts.update_acct_balance;  
-- Using catalog, schema, and trigger name  
DROP TRIGGER accounts.savings_accounts.update_acct_balance;
```

# B.41 DROP USER

```
DROP USER user_name
```

## B.41.1 Usage

The DROP USER statement removes the specified user from the database. All the objects associated with the specified *user\_name* must be dropped prior to using this statement; the DROP USER statement is not a cascaded operation.

## B.41.2 Example

```
DROP USER HOBBS;
```

## B.42 DROP VIEW

```
DROP VIEW view_name  
DROP VIEW [[catalog_name.]schema_name.]view_name
```

### B.42.1 Usage

The DROP VIEW statement removes the specified view from the database.

### B.42.2 Examples

```
DROP VIEW sum_of_acct_balances;  
-- Using schema and view name  
DROP VIEW acct_manager_schema.sum_of_acct_balances;  
-- Using catalog, schema, and view name  
DROP VIEW account_db.acct_manager_schema.sum_of_acct_balances;
```

## B.43 EXPLAIN PLAN FOR

```
EXPLAIN PLAN FOR sql_statement
```

### B.43.1 Usage

The EXPLAIN PLAN FOR statement shows the selected search plan for the specified SQL statement.

### B.43.2 Example

```
EXPLAIN PLAN FOR select * from tables;
```

## B.44 EXPORT SUBSCRIPTION

```
EXPORT SUBSCRIPTION publication_name [(publication_parameters)]
```

```
TO 'filename'  
USING BOOKMARK bookmark_name;  
[WITH [NO] DATA];
```

## B.44.1 Supported in

This command requires IBM solidDB advanced replication.

## B.44.2 Usage

This EXPORT SUBSCRIPTION statement allows you export a version of the data from a master database to a file. You can then use the IMPORT statement to import the data in the file into a replica database.

There are several uses for the EXPORT SUBSCRIPTION statement. Among them are:

- Creating a large replica database (greater than 2GB) from an existing master.

This procedure requires that you export a subscription with or without data to a file first, then import the subscription to the replica. For details, read "Creating A Replica By Exporting A Subscription With Data" or "Creating A Replica By Exporting A Subscription Without Data" in *IBM solidDB Advanced Replication Guide*.

- Exporting specific versions of the data to a replica.

For performance reasons, you may choose to "export" the data rather than to use the MESSAGE APPEND REFRESH to send the data to a replica.

- Export metadata information without the actual row data.

You may want to create a replica that already contains existing data and only needs the schema and version information associated with a publication.

Unlike the MESSAGE APPEND REFRESH statement where a REFRESH is requested from a replica, you request an export directly from the master database. The export output is saved to a user-specified file rather than a IBM solidDB message.

### B.44.2.1 Keywords and Clauses

The *publication\_name* and *bookmark\_name* are identifiers that must exist in the database. For details on creating a publication, read Section B.15, "CREATE [OR REPLACE] PUBLICATION". For details on creating bookmarks, see Section B.19, "CREATE SYNC BOOKMARK". The filename represents a literal value enclosed in single quotes. You can export several publications to a single file by specifying the same file name.



Publication data is exported from the master database as a REFRESH with a set of input parameter values (if they are used in the publication).

The EXPORT SUBSCRIPTION statement is based on a given bookmark, which means that export data is consistent up to this bookmark. When you export data, the EXPORT SUBSCRIPTION statement includes all rows as in a full publication up to the bookmark. However, since export is based on a given bookmark, the subsequent REFRESHes are incremental.

If a bookmark is created in a master for the purpose of exporting and importing data, then the bookmark must exist when:

- The EXPORT SUBSCRIPTION statement is executed on the master database.

If the bookmark does not exist at this point, error message 25067 is generated, indicating the bookmark cannot be found.

- The IMPORT statement is executed on all intended replicas and replicas receive their first set of data ("REFRESH").

During a file import, a connection to a master database is not needed and the existence of the bookmark is not checked. However, if the bookmark does not exist at the time a replica receives its first REFRESH, the REFRESH fails with error message 25067 and the import data is unusable. The remedy is to create a new bookmark on the master, re-export the data, and re-import the data.

An export file can contain more than one publication. You can export subscriptions using either the WITH DATA or WITH NO DATA options:

- Use the WITH DATA option to create a replica when data is exported to an existing database that does not contain master data and requires a subset of data. For details, read "Creating A Replica By Exporting A Subscription With Data" in *IBM solidDB Advanced Replication Guide*.
- Use the WITH NO DATA option to create a replica when a subscription is imported to a database that already contains the data (for example, using a backup copy of an existing master). For details, read "Creating A Replica By Exporting A Subscription Without Data" in *IBM solidDB Advanced Replication Guide*.

By default, the export file is created using the WITH DATA option and includes all rows. If there is more than one publication specified, the exported file can have a combination of "WITH DATA" and "WITH NO DATA."

### **B.44.2.2 Usage Rules**

Note the following rules when using the EXPORT SUBSCRIPTION statement:

- Only one file per subscription is allowed for export. You can use the same file name to include multiple subscriptions on the same file.
- The file size of an export file is dependent upon the underlying operating system. If a respective platform (such as SUN, or HP) allows more than 2GB, you can write files greater than 2GB. This means that replica (recipient) should also have a compatible platform and file system. Otherwise, the replica is not able to accept the export file. If both the operating system on a master and replica support file sizes greater than 2GB, then export files greater than 2GB are permitted.
- An export file can contain more than one subscription. Subscriptions can be exported using either the WITH DATA or WITH NO DATA options. An exported file with multiple subscriptions can have a combination of WITH DATA and WITH NO DATA included.
- When a subscription is exported to a file using the WITH NO DATA option, only metadata (that is, schema and version information corresponding to that publication) is exported to the file.
- IBM solidDB requires that autocommit be set OFF when using the EXPORT SUBSCRIPTION statement.

### B.44.3 Usage in Master

Use this statement to request master data for export to a file.

### B.44.4 Usage in Replica

This statement is not available in a replica database.

### B.44.5 Example

```
EXPORT SUBSCRIPTION FINANCE_PUBLICATION(2004) TO 'FINANCE.EXP'  
USING BOOKMARK BOOKMARK_FOR_FINANCE_DB WITH NO DATA;
```

### B.44.6 Return Values

For details on each error code, see the appendix titled Error Codes in *IBM solidDB Administration Guide*.

**Table B.16. EXPORT SUBSCRIPTION Return Values**

Error Code	Description
25056	Autocommit not allowed
25067	Bookmark is not found.

Error Code	Description
25068	Export file <i>file_name</i> open failure.
25010	Publication <i>name</i> not found.

## B.45 EXPORT SUBSCRIPTION TO REPLICA

```
EXPORT SUBSCRIPTION publication_name [(publication_parameters)]  
  TO REPLICA replica_node_name  
  USING BOOKMARK bookmark_name  
  [COMMITBLOCK number_of_rows]
```

### B.45.1 Supported in

This command requires IBM solidDB advanced replication.

### B.45.2 Usage

The EXPORT SUBSCRIPTION TO REPLICA statement allows you to send large volume of data specified by a publication from master database to a replica database. After the EXPORT operation has completed, the replica may use MESSAGE APPEND REFRESH statement to refresh the data of the subscription in an incremental manner.

Because the EXPORT SUBSCRIPTION TO REPLICA statement does not use the disk-based advanced replication MESSAGEs to send data from master to replica, it provides a significantly more efficient way to send large volumes of data from master to replica as the usage of disk during the operation is minimized.

#### B.45.2.1 Keywords and Clauses

The *publication\_name* and *bookmark\_name* are identifiers that must exist in the database. For details on creating a publication, read Section B.15, “CREATE [OR REPLACE] PUBLICATION”. For details on creating bookmarks, see Section B.19, “CREATE SYNC BOOKMARK”.

Publication data is exported from the master database as a REFRESH with a set of input parameter values (if they are used in the publication).

The EXPORT SUBSCRIPTION TO REPLICA statement is based on a given bookmark, which means that export data is consistent up to this bookmark. When you export data, the EXPORT SUBSCRIPTION statement includes all rows as in a full publication up to the bookmark. However, since export is based on a given bookmark, the subsequent REFRESHes are incremental.

If a bookmark is created in a master for the purpose of exporting data, then the bookmark must exist when the `EXPORT SUBSCRIPTION` statement is executed on the master database. If the bookmark does not exist at this point, error message 25067 is generated, indicating the bookmark cannot be found.

The `COMMIT BLOCK` keywords specify how many rows of the exported data are committed in the replica database in one transaction. Specifying a commit block when a large number of rows are to be exported improves the performance of the operation. However, it is recommended that the replica database is not used by other applications when export operation with commit block is active.

## B.45.3 Usage in Master

Use this statement to request master data for export to a replica database.

## B.45.4 Usage in Replica

This statement is not available in a replica database.

## B.45.5 Example

```
EXPORT SUBSCRIPTION FINANCE_PUBLICATION(2004) TO REPLICA replica_1
USING BOOKMARK BOOKMARK_FOR_FINANCE_DB COMMITBLOCK 10000 ;
```

## B.45.6 Return Values

For details on each error code, see the appendix titled Error Codes in *IBM solidDB Administration Guide*.

**Table B.17. EXPORT SUBSCRIPTION TO REPLICA Return Values**

Error Code	Description
25056	Autocommit not allowed
25067	Bookmark is not found.
25010	Publication <i>name</i> not found.

## B.46 GET\_PARAM()

```
get_param('param_name')
```

### B.46.1 Supported in

This command requires IBM solidDB advanced replication.

### B.46.2 Usage

The `get_param()` function retrieves a parameter that was placed on the transaction bulletin board using the `PUT_PARAM()` function or with commands `SAVE PROPERTY`, `SAVE DEFAULT PROPERTY`, and `SET SYNC PARAMETER`. The parameter that is retrieved is specific to a catalog and each catalog has a different set of parameters. This function returns the `VARCHAR` value of the parameter or `NULL`, if the parameter does not exist in the bulletin board.

Because `get_param()` is an SQL function, it can be used only in a procedure or as part of a `SELECT` statement.

The parameter name must be enclosed in single quotes.

### B.46.3 Usage in Master

Use the `get_param()` function in the master for retrieving parameter values.

### B.46.4 Usage in Replica

Use the `get_param()` function in replicas for retrieving parameter values.

### B.46.5 IBM solidDB System Parameters

IBM solidDB system parameters are divided into the following categories:

- *Read only system parameters* that are maintained by IBM solidDB and can only be read by using `GET_PARAM(parameter_name)` syntax.

The life cycle of parameters in this category is one transaction, that is, values of these parameters will always be initialized at the beginning of the transaction.

- *Updatable system parameters* that can be set and updated by the user through `PUT_PARAM(parameter_name, value)`. Updatable system parameters are used by IBM solidDB.

Like the category above, the life cycle of these parameters is also one transaction.

- *Database catalog level system parameters* that are set using `SET SYNC PARAMETER parameter_name value` syntax.

Parameters in this category are database catalog level parameters that are valid until changed or removed. They are specified as bulletin board parameters.

Full syntax and examples of usage of `GET_PARAM()`, `PUT_PARAM()` and `SET SYNC PARAMETER` functions are described earlier in this chapter.

For more information about specific bulletin board parameters, see *IBM solidDB Advanced Replication Guide*.

## B.46.6 Example

```
SELECT put_param('myparam', '123abc');
SELECT get_param('myparam');
```

## B.46.7 Return Values

For details on each error code, see the appendix titled Error Codes in *IBM solidDB Administration Guide*.

**Table B.18. GET\_PARAM Return Values**

Error Code	Description
13086	Invalid data type in a parameter.

When executed successfully `get_param()` returns the value of the assigned parameter.

## B.46.8 See Also

PUT\_PARAM

SAVE PROPERTY

SET SYNC PARAMETER

## B.47 GRANT

```
GRANT {ALL | grant_privilege [, grant_privilege]...}  
      ON table_name  
TO {PUBLIC | user_name [, user_name]... |  
    role_name [, role_name]... }  
[WITH GRANT OPTION]
```

```
GRANT role_name TO user_name
```

```
grant_privilege ::= DELETE | INSERT | SELECT |  
                  UPDATE [( column_identifier [, column_identifier]... )] |  
                  REFERENCES [( column_identifier [, column_identifier]... )]
```

```
GRANT EXECUTE ON procedure_name  
      TO {PUBLIC | user_name [, user_name]... | role_name [, role_name]... }
```

```
GRANT {SELECT | INSERT} ON event_name  
      TO {PUBLIC | user_name [, user_name]... | role_name [, role_name]... }
```

```
GRANT {SELECT | UPDATE} ON sequence_name  
      TO {PUBLIC | user_name [, user_name]... | role_name [, role_name]... }
```

### B.47.1 Usage

The GRANT statement is used to

1. grant privileges to the specified user or role.
2. grant privileges to the specified user by giving the user the privileges of the specified role.

When you grant a role to a user, the role may be one that you have created, or it may be a system-defined role, such as SYS\_SYNC\_ADMIN\_ROLE or SYS\_ADMIN\_ROLE.

The role SYS\_SYNC\_ADMIN\_ROLE gives the specified user the privileges to execute data synchronization administration operations, including:

- dropping or re-executing stopped synchronization messages,
- dropping a replica database from master database,

- creating a bookmark.

The role `SYS_ADMIN_ROLE` is the role given to the creator of the database. This role has privileges to all tables, indexes, and users, as well as the right to use IBM solidDB Remote Control (teletype).

If you use the optional `WITH GRANT OPTION`, then the user who receives the privilege may grant the privilege to other users.

### B.47.2 Example

```
GRANT GUEST_USERS TO CALVIN;  
GRANT INSERT, DELETE ON TEST TO GUEST_USERS;
```

### B.47.3 See Also

For more information about user privileges, see also:

- Section B.69, “REVOKE (Privilege from Role or User)” and
- Section 5.2, “Managing User Privileges and Roles”.

For more information about pre-defined roles, see chapter *Special Roles for Database Administration* in *IBM solidDB Administration Guide*.

## B.48 GRANT REFRESH

```
GRANT { REFRESH | SUBSCRIBE } ON publication_name TO { PUBLIC |  
user_name ,  
    [ user_name ] ... | role_name , [ role_name ] ... }
```

### B.48.1 Supported in

This command requires IBM solidDB advanced replication.

### B.48.2 Usage

This statement grants access rights on a publication to a user or role defined in the master database.



**Note**

The keywords "SUBSCRIBE" and "REFRESH" are equivalent. However, the keyword "SUBSCRIBE" is deprecated in the GRANT statement.

**B.48.3 Usage in Master**

Use this statement to grant user or role access rights to a publication.

**B.48.4 Usage in Replica**

This statement is not available in a replica database.

**B.48.5 Example**

```
GRANT REFRESH ON customers_by_area TO salesman_jones;
GRANT REFRESH ON customers_by_area TO all_salesmen;
```

**B.48.6 Return Values**

For details on each error code, see the appendix titled Error Codes in *IBM solidDB Administration Guide*.

**Table B.19. GRANT REFRESH Return Values**

Error Code	Description
13137	Illegal grant/revoke mode
13048	No grant option privilege
25010	Publication <i>name</i> not found

**B.49 HINT**

```
--(* vendor (SOLID), product (Engine), option(hint)
--hint *)--
```

```
hint ::=
[MERGE JOIN |
LOOP JOIN |
```

```
JOIN ORDER FIXED |  
INTERNAL SORT |  
EXTERNAL SORT |  
INDEX [REVERSE] table_name.index_name |  
PRIMARY KEY [REVERSE] table_name |  
FULL SCAN table_name |  
[NO] SORT BEFORE GROUP BY]
```

Following is a description of the keywords and clauses used in the syntax:

## B.49.1 Pseudo Comment Identifier

The pseudo comment prefix is followed by identifying information. You must specify the vendor as **SOLID**, product as **Engine**, and the option, which is the pseudo comment class name, as **hint**.



### Note

In the pseudo comment prefix `--(* and *)--`, there must be no space between the parenthesis and the asterisk.

### Hint

Hints always follow the **SELECT**, **UPDATE**, or **DELETE** keyword that applies to it.



### Note

Hints are not allowed after the **INSERT** keyword.



### Caution

If you are using hints and you compose a query as a string and then submit that string using ODBC or JDBC, you **MUST** ensure that appropriate newline characters are embedded within that string to mark the end of the comments. Otherwise, you will get a syntax error. If you don't include any newlines, then all of the statement after the start of the first comment will look like a comment. For example, suppose that your code looks like the following:

```
strcpy(s, "SELECT --(* hint... *)-- col_name FROM table;");
```

Everything after the first `--` looks like a comment, and therefore your statement looks incomplete. You must do something like the following:

```
strcpy(s, "SELECT --(* hint... *)-- \n col_name FROM table;");
```

Note the embedded newline "\n" character to terminate the comment. A useful technique for debugging is to print out the strings to make sure that they look correct. They should look like:

```
SELECT --(* hint ... *)--  
column_name FROM table_name...;
```

or

```
SELECT --(* hint ... *)--  
column_name FROM table_name...;
```

Each subselect requires its own hint; for example, the following are valid uses of hints syntax:

```
INSERT INTO ... SELECT hint FROM ...  
UPDATE hint TABLE ... WHERE column = (SELECT hint ... FROM ...)  
DELETE hint TABLE ... WHERE column = (SELECT hint ... FROM ...)
```

Be sure to specify multiple hints in one pseudo comment separated by commas as shown in the following examples:

## B.49.2 Example 1

```
SELECT  
--(* vendor(SOLID), product(Engine), option(hint)  
--MERGE JOIN  
--JOIN ORDER FIXED *)--  
*  
FROM TAB1 A, TAB2 B;  
WHERE A.INTF = B.INTF;
```

## B.49.3 Example 2

```
SELECT  
--(* vendor(SOLID), product(Engine), option(hint)
```

```
--INDEX TAB1.INDEX1
--INDEX TAB1.INDEX1 FULL SCAN TAB2 *)--
*
FROM TAB1, TAB2
WHERE TAB1.INTF = TAB2.INTF;
```

Hint is a specific semantic, corresponding to a specific behavior. Following is a list of possible hints:

### Table B.20. Hints

Hint	Definition
MERGE JOIN	<p>Directs the Optimizer to choose the merge join access plan in a select query for all tables listed in the FROM clause. The MERGE JOIN option is used when two tables are approximately equal in size and the data is distributed equally. It is faster than a LOOP JOIN when an equal amount of rows are joined. For joining data, MERGE JOIN supports a maximum of three tables. The joining table is ordered by joining columns and combining the results of the columns.</p> <p>You can use this hint when the data is sorted by a join key and the nested loop join performance is not adequate. The Optimizer selects the merge join only where there is an equal predicate between tables. Otherwise, the Optimizer selects LOOP JOIN even if the MERGE JOIN hint is specified.</p> <p>Note that when data is not sorted before performing the merge operation, the IBM solidDB query executor sorts the data.</p> <p>When considering the usage of this hint, keep in mind that the merge join with a sort is more resource intensive than the merge join without the sort.</p>
LOOP JOIN	<p>Directs the Optimizer to pick the nested loop join in a select query for all tables listed in the FROM clause. By default, the Optimizer does not pick the nested loop join. Using the loop join when tables are small and fit in memory may offer greater efficiency than using other join algorithms.</p> <p>The LOOP JOIN loops through both inner and outer tables. It is used when a small and large amount of rows are joined. It finds matches between columns in inner and outer tables. For better performance, the joining columns should be indexed.</p> <p>You can use the loop join when tables are small and fit in memory.</p>
JOIN ORDER FIXED	<p>Specifies that the Optimizer use tables in a join in the order listed in the FROM clause of the query. This means that the Optimizer does not attempt to rearrange any join order and does not try to find alternate access paths to complete the join.</p>

Hint	Definition
	<p>Before using this hint, be sure to run the EXPLAIN PLAN to view the associated plan. This gives you an idea on the access plan used for executing the query with this join order.</p>
INTERNAL SORT	<p>Specifies that the query executor use the internal sorter. Use this hint if the expected resultset is small (hundreds of rows as opposed to thousands of rows), for example, if you are performing some aggregates, ORDER BY with small resultsets, or GROUP BY with small resultsets, etc.</p> <p>This hint avoids the use of the more expensive external sorter.</p>
EXTERNAL SORT	<p>Specifies that the query executor use the external sorter. Use this hint when the expected resultset is large and does not fit in memory, for example, if the expected resultset has thousands of rows.</p> <p>In addition, specify the SORT working directory in the <code>solid.ini</code> file before using the external sort hint. If a working directory is not specified, you will receive a runtime error. The working directory is specified in the <code>[sorter]</code> section of the <code>solid.ini</code> configuration file. For example:</p> <pre data-bbox="396 864 761 921">[sorter] TmpDir_1=c:\solddb\temp1</pre>
INDEX [REVERSE] <i>table_name.index_name</i>	<p>Forces a given index scan for a given table. In this case, the Optimizer does not proceed to evaluate if there are any other indexes that can be used to build the access plan or whether a table scan is better for the given query.</p> <p>Before using this hint, it is recommended that you "test" the hint by running the EXPLAIN PLAN output to ensure that the plan generated is optimal for the given query.</p> <p>The optional keyword REVERSE returns the rows in the reverse order. In this case, the query executor begins with the last page of the index and starts returning the rows in the descending (reverse) key order of the index.</p> <p>Note that in <i>tablename.indexname</i>, the tablename is a fully qualified table name which includes the <i>catalogname</i> and <i>schemaname</i>.</p>
PRIMARY KEY [REVERSE] <i>tablename</i>	<p>Forces a primary key scan for a given table.</p> <p>The optional keyword REVERSE returns the rows in the reverse order.</p>

Hint	Definition
	If the primary key is not available for the given table, then you will receive a run-time error.
FULL SCAN <i>table_name</i>	Forces a table scan for a given table. In this case, the optimizer does not proceed to evaluate if there are any other indexes that can be used to build the access plan or whether a table scan is better for the given query.  Before using this hint, it is recommended that you "test" the hint by running the EXPLAIN PLAN output to ensure that the plan generated is optimal for the given query.
[NO] SORT BEFORE GROUP BY	Indicates whether the SORT operation occurs before the resultset is grouped by the GROUP BY columns.  If the grouped items are few (hundreds of rows) then use NO SORT BEFORE. On the other hand, if the grouped items are large (thousands of rows), then use SORT BEFORE.

## B.49.4 Usage

Due to various conditions with the data, user query, and database, the SQL Optimizer is not always able to choose the best possible execution plan. For more efficiency, you may want to force a merge join because you, unlike the Optimizer, know that your data is already sorted.

Or sometimes specific predicates in queries cause performance problems that the Optimizer cannot eliminate. The Optimizer may be using an index that you know is not optimal. In this case, you may want to force the Optimizer to use one that produces faster results.

Optimizer hints is a way to have better control over response times to meet your performance needs. Within a query, you can specify directives or hints to the Optimizer, which it then uses to determine its query execution plan. Hints are detected through a pseudo comment syntax from SQL-92.

You can place a hint(s) in a SQL statement as a static string, just after a SELECT, INSERT, UPDATE, or DELETE keyword. The hint always follows the SQL statement that applies to it.

Table name resolution in optimizer hints is the same as in any table name in a SQL statement. When there is an error in a hint specification, then the whole SQL statement fails with an error message.

Hints are enabled and disabled using the following configuration parameter in the `solid.ini`:

[Hints]

EnableHints = YES | NO

The default is YES.

## B.49.5 Example

```
SELECT
--(* vendor(SOLID), product(Engine), option(hint)
-- INDEX TAB1.IDX1 *)--
* FROM TAB1 WHERE I > 100
```

```
SELECT
--(* vendor(SOLID), product(Engine), option(hint)
-- INDEX MyCatalog.mySchema.TAB1.IDX1 *)--
* FROM TAB1 WHERE I > 100
```

```
SELECT
--(* vendor(SOLID), product(Engine), option(hint)
-- JOIN ORDER FIXED *)--
* FROM TAB1, TAB2 WHERE TAB1.I >= TAB2.I
```

```
SELECT
--(* vendor(SOLID), product(Engine), option(hint)
-- LOOP JOIN *)--
* FROM TAB1, TAB2 WHERE TAB1.I >= TAB2.I
```

```
SELECT
--(* vendor(SOLID), product(Engine), option(hint)
-- INDEX REVERSE MyCatalog.mySchema.TAB1.IDX1 *)--
* FROM TAB1 WHERE I > 100
```

```
SELECT
--(* vendor(SOLID), product(Engine), option(hint)
-- SORT BEFORE GROUP BY *)--
AVG(I) FROM TAB1 WHERE I > 10 GROUP BY I2
```

```
SELECT
--(* vendor(SOLID), product(Engine), option(hint)
```

```
-- INTERNAL SORT *)--  
* FROM TAB1 WHERE I > 10 ORDER BY I2
```

## B.50 IMPORT

```
IMPORT 'file_name' [COMMITBLOCK number_of_rows]  
[ {OPTIMISTIC | PESSIMISTIC} ]
```

### B.50.1 Usage

This IMPORT command allows you to import data to a replica from a data file created by the EXPORT SUBSCRIPTION command.

The *file\_name* represents a literal value enclosed in single quotes. The import command can accept a single filename only. Therefore, all the data to be imported to a replica must be contained in one file.

The COMMITBLOCK option indicates the number of rows processed before the data is committed. The *number\_of\_rows* is an integer value used with the optional COMMITBLOCK clause to indicate the commitblock size. The use of the COMMITBLOCK option improves the performance of the import and releases the internal transaction resources frequently.

The optimal value for the COMMITBLOCK size varies depending on various resources at the server. A good example is a COMMITBLOCK size of 1000 for 10,000 rows. If you do not specify the COMMITBLOCK option, the IMPORT command uses all rows in the publication as one transaction. This may work well for a small number of rows, but is problematic for thousands and millions of rows.

You can define the import to use table-level pessimistic locking when it is initially executed. If the PESSIMISTIC mode is specified, all other concurrent access to the table affected is blocked until the import has completed. Otherwise, if the optimistic mode is used, the IMPORT may fail due to a concurrency conflict.

When a transaction acquires an exclusive lock to a table, the *TableLockWaitTimeout* parameter setting in the *[General]* section of the *solid.ini* configuration file determines the transaction's wait period until the exclusive or shared lock is released. For details, see the description of this parameter in *IBM solidDB Administration Guide*.

Imported data is not valid in a replica until it is refreshed once after the import. At the time a replica makes its first REFRESH, the bookmark used to export the file must exist in the master database. If it does not exist, then the REFRESH fails. This means that you are required to create a new bookmark on the master database, re-export the data, and re-import the data on the replica.



### B.50.1.1 Usage Rules

Note the following rules when using the IMPORT command:

- Only one file per subscription is allowed for import.
- The file size of an export file is dependent upon the underlying operating system. If a respective platform (such as SUN, or HP) allows more than 2GB, you can write files greater than 2GB. This means that a replica (recipient) should also have a compatible platform and file system. Otherwise, the replica is not able to accept the export file. If both the operating system on a master and replica support file sizes greater than 2GB, then export files greater than 2GB are permitted.
- Back up replica databases before using the IMPORT command. If a COMMITBLOCK option is used and fails, then the imported data is only partially committed; you will need to restore the replica with a backup file.
- IBM solidDB requires that autocommit be set OFF when using the IMPORT command.

## B.50.2 Usage in Master

This statement is not available in a master database.

## B.50.3 Usage in Replica

Use this statement in a replica to import data from a data file created by the EXPORT SUBSCRIPTION statement in a master database.

## B.50.4 Example

```
IMPORT 'FINANCE.EXP';
```

## B.50.5 Return Values

For details on each error code, see the appendix titled Error Codes in *IBM solidDB Administration Guide*.

**Table B.21. IMPORT Return Values**

Error Code	Description
25007	Master <i>master_name</i> not found.
25019	Database is not a replica database.

Error Code	Description
25069	Import file <i>file_name</i> open failure.
13XXX	Table level error
13124	User id <i>num</i> not found  This message is generated, for example, if the user has been dropped.
10006	Concurrency conflict (simultaneous other operation)
13047	No privilege for operation
13056	Insert not allowed for pseudo column
21XXX	Communication error
25024	Master not defined
25026	Not a valid master user
25031	Transaction is active, operation failed
25036	Publication <i>publication_name</i> not found or publication version mismatch
25040	User id <i>user_id</i> is not found  While executing a message reply an attempt to map a master user to a local replica id failed.
25041	Subscription to publication <i>publication_name</i> not found
25048	Publication <i>publication_name</i> request info not found
25054	Table <i>table_name</i> is not set for synchronization history
25056	Autocommit not allowed
25060	Column <i>column_name</i> does not exist on publication <i>publication_name</i> resultset in table <i>table_name</i>

## B.51 INSERT

```
INSERT INTO table_name insert_columns_and_source
```

```
insert_columns_and_source::=  
from_subquery  
| from_constructor  
| from_default
```

```
from subquery ::=
[insert_column_name_list] query expression

insert_column_name_list ::=
([column name [, column name]... ])

from constructor ::=
[insert_column_name_list] VALUES row_constructor[, row_constructor]... ]

row_constructor ::= ([insert_item[, insert_item]...])

insert_item ::= insert_value | DEFAULT | NULL

from default ::= DEFAULT VALUES
```

## B.51.1 Usage

There are several variations of the INSERT statement. In the simplest instance, a value is provided for each column of the new row in the order specified at the time the table was defined (or altered). In the preferable form of the INSERT statement, the columns are specified as part of the statement and they do not need to be in any specific order as long as the orders of the column list matches the order of the value list.

<insert\_value> can be a literal, a scalar function, or a variable (in a procedure).

## B.51.2 Example

```
INSERT INTO TEST (C, ID) VALUES (0.22, 5);
INSERT INTO TEST VALUES (0.35, 9);
```

Multirow inserts can also be done. For example, to insert three rows in one statement, you can use the following command:

```
INSERT INTO employees VALUES
(10021, 'Peter', 'Humlaut'),
(10543, 'John', 'Wilson'),
(10556, 'Bunba', 'Olo');
```

You can insert default values by using the DEFAULT VALUES statement as shown in the second example below. An equivalent form is "INSERT INTO TEST() VALUES()". You can also assign a specific value for one column and use the default value for another column. These methods as shown in the examples below:

```
INSERT INTO TEST () VALUES ();
INSERT INTO TEST DEFAULT VALUES;
INSERT INTO TEST (C, ID) VALUES (0.35, DEFAULT);
```

```
INSERT INTO TEST (C, ID) SELECT A, B FROM INPUT_TO_TEST;
```

## B.52 LOCK TABLE

```
LOCK lock-definition [lock-definition] [wait-option]
lock-definition ::= TABLE tablename [,tablename]
IN { SHARED | [LONG] EXCLUSIVE } MODE
wait-option ::= NOWAIT | WAIT <#seconds>
```

**Tablename:** The name of the table to lock. You can also specify the catalog and schema of the table by qualifying the table name. You may only lock tables, not views.

**SHARED:** Shared mode allows others to perform read and write operations on the table. Also DDL operations are allowed. Shared mode prohibits others from issuing an EXCLUSIVE lock on the same table.

**EXCLUSIVE:** If a table uses pessimistic locking, then an exclusive lock prevents any other user from accessing the table in any way (e.g. reading data, acquiring a lock, etc.). If the table uses optimistic locking, then an exclusive lock allows other users to perform SELECTs on the locked table but prohibits any other activity (like acquiring shared locks) on that table.

**LONG:** By default, locks are released at the end of a transaction. If the LONG option is specified, then the lock is not released when the locking transaction commits. (Note: If the locking transaction aborts or is rolled back, then all locks, including LONG locks, are released.) The user must explicitly unlock long locks using the UNLOCK command described later in this document. LONG duration locks are allowed only in EXCLUSIVE mode. LONG shared locks are not supported.

**NOWAIT:** Specifies that control is returned to you immediately even if any specified table is locked by another user. If requested lock is not granted then an error is returned.

**WAIT:** Specifies a timeout how long in seconds system should wait to get requested locks. If requested lock is not granted within that time then an error is returned.

## B.52.1 Usage

The LOCK and UNLOCK commands allow you to manually lock and unlock tables. Putting a lock on a table (or any other object) limits access to that object. The LOCK TABLE command has an option that allows you to extend the duration of a manual exclusive lock past the end of the current transaction; in other words, you can keep the table exclusively locked through a series of multiple transactions.

Manual locking is not needed very often. The server's automatic locking is usually sufficient. For a detailed discussion of locking in general, and the server's automatic locking in particular, see Section 6.2, “Concurrency Control and Locking”.

Explicit locking of tables is primarily intended to help database administrators execute maintenance operations in a database without being disturbed by other users. (For more information about Maintenance Mode, see the chapter titled “Updating and Maintaining the Schema of a Distributed System” in *IBM solidDB Advanced Replication Guide*.) However, tables can be locked manually even if you are not using “Maintenance Mode”.

Table locks can be either SHARED or EXCLUSIVE.

An EXCLUSIVE lock on a table prohibits any other user or connection from changing the table or any records within the table. If you have an exclusive lock on a table, then other users/connections cannot do any of the following on that table until your exclusive lock is released:

- INSERT, UPDATE, DELETE
- ALTER TABLE
- DROP TABLE
- LOCK TABLE (in shared or exclusive mode)

Furthermore, if the table uses pessimistic locking, then an exclusive lock also prevents others users/connections from doing the following:

- SELECT

If the table uses pessimistic locking, no other user can SELECT from the table when it has an exclusive lock. Note, however, that if the table uses optimistic locking, an exclusive lock does NOT prevent other users from SELECTing records from that table. (Most database servers on the market behave differently — i.e. they do not allow SELECTs on a table that is locked exclusively — because most other database servers use only pessimistic locking.)

A SHARED lock is less restrictive than an exclusive lock. If you have a shared lock on a table, then other users/connections cannot do any of the following on that table until your shared lock is released:

- ALTER TABLE
- DROP TABLE
- LOCK TABLE (in exclusive mode)

If you use a shared lock on a table, other users/connections may insert, update, delete, and of course select from the table.

Note that shared locks on a table are somewhat different from shared locks on a record. If you have a shared lock on a record, then no other user may change data in the record. However, if you have a shared lock on a table, then other users may still change data in that table.

More than one user at a time may have shared locks on a table. Therefore, if you have a shared lock on the table, other users may also get shared locks on the table. However, no user may get an exclusive lock on a table when another user has a shared lock (or exclusive lock) on that table.

The LOCK command takes effect at the time it is executed. If you do not use the LONG option, then the lock will be released at the end of the transaction. If you use the LONG option, the table lock lasts until you explicitly unlock the table. (The table lock will also be released if you roll back the transaction in which the lock was placed. In other words, LONG locks only persist across transactions if you commit the transaction in which you placed the LONG lock.)

The LOCK/UNLOCK TABLE commands apply only to tables. There is no command to manually lock or unlock individual records within a table.

Privileges required: To use the LOCK TABLE command to issue a lock on a table, you must have insert, delete or update privileges on that table. Note that there is no GRANT command to give other users LOCK and UNLOCK privileges on a table.

Note that in one LOCK command you can lock more than one table and specify different modes. If the lock command fails, then none of the tables are locked. If the lock command was successful, then all requested locks are granted.

If the user does not specify a wait option (NOWAIT or WAIT seconds), then the default wait time is used. That is the same as the deadlock detection timeout.

## B.52.2 Examples

```
LOCK TABLE emp IN SHARED MODE ;  
LOCK TABLE emp IN SHARED MODE TABLE dept IN EXCLUSIVE MODE ;
```

```
LOCK TABLE emp,dept IN SHARED MODE NOWAIT;
LOCK TABLE emp IN LONG EXCLUSIVE MODE;
```

## B.52.3 Return Values

For details on each error code, see the appendix titled Error Codes in *IBM solidDB Administration Guide*.

**Table B.22. LOCK TABLE Return Values**

Error code	Description
10014	Resource is locked.
13047	No privilege for operation.
13011	Table <tablename> is not found.

## B.52.4 See Also

```
UNLOCK TABLE
```

```
SET SYNC MODE { MAINTENANCE | NORMAL }
```

## B.53 MESSAGE APPEND

```
MESSAGE unique_message_name APPEND
  [
    PROPAGATE TRANSACTIONS
    [ { IGNORE_ERRORS | LOG_ERRORS | FAIL_ERRORS } ]
    [WHERE { property_name {=<|<=|>|>=|<>} 'value_string' | ALL } ]
  ]
[ { REFRESH | SUBSCRIBE }
publication_name[(publication_parameters)]
timeout[(timeout_in_seconds)]
[FULL]
]
[REGISTER PUBLICATION publication_name]
[UNREGISTER PUBLICATION publication_name]
[REGISTER REPLICA]
```

```
[UNREGISTER REPLICA]
[SYNC_CONFIG ('sync_config_arg')]
```

## B.53.1 Supported in

This command requires IBM solidDB advanced replication.

## B.53.2 Usage

Once a message has been created in the replica database with the MESSAGE BEGIN command, you can append the following tasks to that message:

- Propagate transactions to the master database
- Refresh a publication from the master database
- Register or unregister a publication for replica subscription
- Register or unregister a replica database to the master
- Download master user information (list of user names and passwords) from the master database

The PROPAGATE TRANSACTIONS task may contain a WHERE clause that is used to propagate only those transactions where a transaction property defined with the SAVE PROPERTY statement meets specific criteria. Using the keyword ALL overrides any default propagation condition set earlier with the statement

```
SAVE DEFAULT PROPAGATE PROPERTY
WHERE property_name {=|<|<=|>|>=|<>} 'value'.
```

This enables you to propagate transactions that do not contain any properties.

The REGISTER REPLICA task adds a new replica database to the list of replicas in the master database. Replicas must be registered with the master database before any other synchronization functions can be performed in the replica database.

Synchronizing each master database to the replica in a multi-master environment requires registration of a replica to each master database by setting up catalogs. One replica catalog can register only to one master catalog. This statement performs the actual registration once catalogs are created in a synchronization environment. For synchronization to the replica, a new catalog for each master database is required. Read the section titled "Guidelines For Multi-Master Topology" in *IBM solidDB Advanced Replication Guide* for details on catalogs.





### Note

A single-master environment does not require the use of catalogs. By default, when catalogs are not used, registration of the replica occurs automatically with a base catalog that maps to a master base catalog, whose name is given when the database is created.



### Note

A single replica *node* may have multiple masters, but only if the node has a separate replica *catalog* for each master catalog. A single replica *catalog* may not have multiple masters.

The UNREGISTER REPLICA option removes an existing replica database from the list of replicas in the master database.

The REFRESH task may contain arguments to the publication (if used in the publication). The parameters must be literals; you cannot use stored procedure variables, for example. Using keyword FULL with REFRESH forces fetching full data to the replica. The publication requested must be registered. Note that the keywords REFRESH and SUBSCRIBE are synonymous; however, SUBSCRIBE is deprecated in the MESSAGE APPEND statement.

The REGISTER PUBLICATION task registers a publication in the replica so that it can be refreshed from. Users can only refresh from publications that are registered. In this way, publication parameters are validated, preventing users from accidentally subscribing to unwanted subscriptions or requesting ad hoc subscriptions. All tables that the registered publication refers to must exist in the replica.

The UNREGISTER PUBLICATION option removes an existing registered publication from the list of registered publications in the master database.

The input argument of the SYNC\_CONFIG task defines the search pattern of the user names that are returned from the master database to the replica. SQL wildcards (such as the symbol %) that follow the conventions of the LIKE keyword are used in this argument with a *match\_string*, which is a character string. For details on using the LIKE keyword, see Section B.96, “Wildcard Characters”.

## B.53.3 Usage in Master

The MESSAGE APPEND statement cannot be used in a master database.

## B.53.4 Usage in Replica

Use MESSAGE APPEND in replicas to append tasks to a message that has been created with MESSAGE BEGIN.

## B.53.5 Example

```

MESSAGE MyMsg0001 APPEND PROPAGATE TRANSACTIONS ;
MESSAGE MyMsg0001 APPEND REFRESH PUB_CUSTOMERS_BY_AREA('SOUTH');
MESSAGE MyMsg0001 APPEND REGISTER REPLICA;
MESSAGE MyMsg0001 APPEND SYNC_CONFIG ('S%');
MESSAGE MyMsg0001 APPEND REGISTER PUBLICATION publ_customer;

```

## B.53.6 Return Values

For details on each error code, see the appendix titled Error Codes in *IBM solidDB Administration Guide*.

**Table B.23. MESSAGE APPEND Return Values**

Error Code	Description
13133	Not a valid license for this product
25004	Dynamic parameters are not supported
25005	Message <i>message_name</i> is already active.
25006	Message <i>message_name</i> not active
25015	Syntax error: <i>error_message</i> , line <i>line_number</i>
25018	Illegal message state.  An appending message in the replica must be placed between the MESSAGE BEGIN and MESSAGE END statements.
25024	Master not defined
25025	Node name not defined
25026	Not a valid master user
25028	Message <i>message_name</i> can include only one system subscription
25035	Message <i>message_name</i> is in use.  A user is currently creating or forwarding this message.
25044	SYNC_CONFIG system publication takes only character arguments
25056	Autocommit not allowed
25071	Not registered to publication <i>publication_name</i>
25072	Already registered to publication <i>publication_name</i>

## B.54 MESSAGE BEGIN

```
MESSAGE unique_message_name BEGIN [TO master_node_name]
```

### B.54.1 Supported in

This command requires IBM solidDB advanced replication.

### B.54.2 Usage

Each message that is sent from a replica to the master database must explicitly begin with the MESSAGE BEGIN statement.

Each message must have a name that is unique within a replica. To construct unique message names, you may use the `GET_UNIQUE_STRING()` function, which is documented in Section B.88, “String Functions”. After a message has been processed, that message name may be reused. However, if the message fails for any reason, the master will keep a copy of the failed message, and if you try to reuse the message name before you delete the failed message, then of course the name will not be unique. You may want to use a new message name even in situations where you might be able to re-use an existing name. Note that it is possible for two replicas of the same master to have the same message name.

When registering a replica to a master catalog, other than the master system catalog, you must provide the master node name in the MESSAGE BEGIN command. The master node name is used to resolve the correct catalog at the master database. Note that specifying a master node name only applies when using the REGISTER REPLICA statement. Later messages are automatically sent to the correct master node.

If you use the optional "TO *master\_node\_name*" clause, then you must put double quotes around the *master\_node\_name*.



#### Note

When working with messages, be sure the autocommit mode is always switched off.

### B.54.3 Usage in Master

The MESSAGE BEGIN statement cannot be used in a master database.

### B.54.4 Usage in Replica

Use MESSAGE BEGIN to start building a new message in a replica.

## B.54.5 Example

```
MESSAGE MyMsg0001 BEGIN ;
MESSAGE MyMsg0002 BEGIN TO "BerkeleyMaster";
```

Return Values from Replica

For details on each error code, see the appendix titled Error Codes in *IBM solidDB Administration Guide*.

**Table B.24. MESSAGE BEGIN Return Values from Replica**

Error Code	Description
25005	Message <i>message_name</i> is already active.  A message of the specified name was created and appears to still be active. The message is automatically deleted when the reply of the message has been successfully executed in the replica.
25035	Message <i>message_name</i> is in use.  A user is currently creating or forwarding this message.
25056	Autocommit not allowed

## B.54.6 Return Values from Master

For details on each error code, see the appendix titled Error Codes in *IBM solidDB Administration Guide*.

**Table B.25. MESSAGE BEGIN Return Values from Master**

Error Code	Description
25019	Database is not a replica database.
25025	Node name not defined.
25056	Autocommit not allowed

## B.55 MESSAGE DELETE

```
MESSAGE message_name [FROM REPLICA replica_name] DELETE
```

### B.55.1 Supported in

This command requires IBM solidDB advanced replication.

### B.55.2 Usage

If the execution of a message is terminated because of an error, this command lets you explicitly delete the message from the database to recover from the error. Note that the current transaction and all subsequent transactions that were propagated to the master in this message are permanently lost when the message is deleted. To use this statement, you must have SYS\_SYNC\_ADMIN\_ROLE access.



#### Note

As an alternative, the MESSAGE DELETE CURRENT TRANSACTION command provides better recovery since it lets you delete only the offending transaction.

If the message needs to be deleted from the master database, then the node name of the replica database that forwarded the message needs to also be provided.

When deleting messages, be sure the autocommit mode is always switched off.

### B.55.3 Usage in Master

Use this statement in the master to delete a failed message. Be sure to specify the replica in the syntax: 'FROM REPLICA *replica\_name*'.

### B.55.4 Usage in Replica

This statement is used in the replica to delete a message.

## B.55.5 Example

```
MESSAGE MyMsg0000 DELETE ;
MESSAGE MyMsg0001 FROM REPLICa bills_laptop DELETE ;
```

Return Values from replica

For details on each error code, see the appendix titled Error Codes in *IBM solidDB Administration Guide*.

**Table B.26. MESSAGE DELETE Return Values from Replica**

Error code	Description
25005	Message <i>message_name</i> is already active
25013	Message <i>message_name</i> not found
25035	Message <i>message_name</i> is in use. A user is currently creating or forwarding this message.
25056	Autocommit not allowed

For details on each error code, see the appendix titled Error Codes in *IBM solidDB Administration Guide*.

**Table B.27. MESSAGE DELETE Return Values from Master**

Error code	Description
13047	No privilege for operation
25009	Replica <i>replica_name</i> not found
25013	Message <i>message_name</i> not found
25020	Database is not a master database
25035	Message <i>message_name</i> is in use. A user is currently executing this message.
25056	Autocommit not allowed

## B.56 MESSAGE DELETE CURRENT TRANSACTION

```
MESSAGE message_name FROM REPLICA replica_name  
DELETE CURRENT TRANSACTION
```

### B.56.1 Supported in

This command requires IBM solidDB advanced replication.

### B.56.2 Usage

This statement deletes the current transaction from a given message in the master database. To use this statement requires `SYS_SYNC_ADMIN_ROLE` privilege.

The execution of a message stops, if a DBMS level error such as a duplicate insert occurs during the execution. This kind of error can be resolved by deleting the offending transaction from the message. Once deleted with the `MESSAGE FROM REPLICA DELETE CURRENT TRANSACTION`, an administrator can proceed with the synchronization process.

When deleting the current transaction, be sure the autocommit mode is always switched off.

This statement is used only when the message is in an error state; if used otherwise, an error message is returned. This statement is a transactional operation and must be committed before message execution may continue. To restart the message after the deletion is committed, use the following statement:

```
MESSAGE msgname FROM REPLICA replicaname EXECUTE
```

Note that the deletion is completed first before the `MESSAGE FROM REPLICA EXECUTE` statement is executed; that is, the statement starts the message from replica, but waits until the active statement is completed before actually executing the message. Thus the statement performs asynchronous message execution.



### Caution

Delete a transaction only as a last resort; normally transactions should be written to prevent unresolved conflicts in a master database. `MESSAGE FROM REPLICA DELETE CURRENT TRANSACTION` is intended for use in the development phase, when unresolved conflicts occur more frequently.

Use caution when deleting a transaction. Because subsequent transactions may be dependent on the results of a deleted transaction, the risk incurred is more transaction errors.

## B.56.3 Usage in Master

Use this statement in the master to delete a failed transaction.

## B.56.4 Usage in Replica

This statement is not available in the replica.

## B.56.5 Example

```
MESSAGE somefailures FROM REPLICA laptop1 DELETE
CURRENT TRANSACTION;
COMMIT WORK;
MESSAGE somefailures FROM REPLICA laptop1 EXECUTE;
COMMIT WORK;
```

## B.56.6 Return Values

For details on each error code, see the appendix titled Error Codes in *IBM solidDB Administration Guide*.

**Table B.28. MESSAGE DELETE CURRENT TRANSACTION Return Values**

Error code	Description
13047	No privilege for operation
25009	Replica <i>replica_name</i> not found
25013	Message name <i>message_name</i> not found
25018	Illegal message state. An attempt was made to delete a transaction from a message that is not in error.
25056	Autocommit not allowed



## B.57 MESSAGE END

MESSAGE *unique\_message\_name* END

### B.57.1 Supported in

This command requires IBM solidDB advanced replication.

### B.57.2 Usage

A message must be "wrapped up" and made persistent before it can be sent to the master database. Ending the message with the MESSAGE END command closes the message, i.e. you can no longer append anything to it. Committing the transaction makes the message persistent.



#### Note

When working with messages, be sure the autocommit mode is switched off.

### B.57.3 Usage in Master

The MESSAGE END statement cannot be used in a master database.

### B.57.4 Usage in Replica

Use the MESSAGE END statement in replicas to end a message.

Example

```
MESSAGE MyMsg001 END ;  
COMMIT WORK ;
```

The following example shows a complete message that propagates transactions and refreshes from publication PUB\_CUSTOMERS\_BY\_AREA.

```
MESSAGE MyMsg001 BEGIN ;  
MESSAGE MyMsg001 APPEND PROPAGATE TRANSACTIONS ;  
MESSAGE MyMsg001 APPEND REFRESH PUB_CUSTOMERS_BY_AREA( '~SOUTH' ) ;
```

```
MESSAGE MyMsg001 END ;
COMMIT WORK ;
```

## B.57.5 Return Values from Replica

For details on each error code, see the appendix titled Error Codes in *IBM solidDB Administration Guide*.

**Table B.29. MESSAGE END Return Values from Replica**

Error Code	Description
13133	Not a valid license for this product
25005	Message <i>message_name</i> is already active
25013	Message <i>message_name</i> not found
25018	Illegal message state.  The MESSAGE BEGIN statement must exist to begin a transaction and the MESSAGE END statement can be executed only once per message.
25026	Not a valid master user
25035	Message <i>message_name</i> is in use  A user is currently creating or forwarding this message.
25056	Autocommit not allowed

## B.57.6 Return Values from Master

For details on each error code, see the appendix titled Error Codes in *IBM solidDB Administration Guide*.

**Table B.30. MESSAGE END Return Values from Master**

Error Code	Description
25019	Database is not a replica database
25056	Autocommit not allowed

## B.58 MESSAGE EXECUTE

```
MESSAGE message_name EXECUTE [{OPTIMISTIC | PESSIMISTIC}]
```

### B.58.1 Supported in

This command requires IBM solidDB advanced replication.

### B.58.2 Usage

This statement allows a message to be re-executed if the execution of a reply message fails in a replica. This can occur, for example, if the database server detects a concurrency conflict between a REFRESH and an ongoing user transaction.

If you anticipate concurrency conflicts to happen often and the re-execution of the message fails because of a concurrency conflict, you can execute the message using the PESSIMISTIC option for table-level locking; this ensures the message execution is successful.

In this mode, all other concurrent access to the table affected is blocked until the synchronization message has completed. Otherwise, if the optimistic mode is used, the MESSAGE EXECUTE statement may fail due to a concurrency conflict.

When a transaction acquires an exclusive lock to a table, the *TableLockWaitTimeout* parameter setting in the *General* section of the *solid.ini* configuration file determines the transaction's wait period until the exclusive or shared lock is released. For details, see the description of this parameter in *IBM solidDB Administration Guide*.



#### Note

When working with messages, be sure the autocommit mode is always switched off.

### B.58.3 Usage in Master

This statement is not available in the master. See Section B.61, “MESSAGE FROM REPLICA EXECUTE”.

### B.58.4 Usage in Replica

Use this statement in the replica to re-execute a failed message execution in the replica.

## B.58.5 Result Set

MESSAGE EXECUTE returns a result set. The returned result set is the same as with command MESSAGE GET REPLY.

## B.58.6 Example

```
MESSAGE MyMsg0002 EXECUTE;
```

## B.58.7 Return Values

For details on each error code, see the appendix titled Error Codes in *IBM solidDB Administration Guide*.

**Table B.31. MESSAGE EXECUTE Return Values**

Error code	Description
13XXX	Table level error
10006	Concurrency conflict (simultaneous other operation)
13047	No privilege for operation
13056	Insert not allowed for pseudo column
25005	Message <i>message_name</i> is already active
25013	Message name <i>message_name</i> not found
25018	Illegal message state
25024	Master not defined
25026	Not a valid master user
25031	Transaction is active, operation failed
25035	Message <i>message_name</i> is in use A user is currently creating or forwarding this message.
25040	User id <i>user_id</i> is not found While executing a message reply an attempt to map a master user to a local replica id failed.
25041	Subscription to publication <i>publication_name</i> not found
25048	Publication <i>publication_name</i> request info not found

Error code	Description
25056	Autocommit not allowed

## B.59 MESSAGE FORWARD

```
MESSAGE unique_message_name FORWARD
[TO {'connect_string' | node_name | "node_name"} ]
[TIMEOUT {number_of_seconds | FOREVER} ]
[COMMITBLOCK block_size_in_rows]
[{OPTIMISTIC | PESSIMISTIC}]
```

### B.59.1 Supported in

This command requires IBM solidDB advanced replication.

### B.59.2 Usage

After a message has been completed and made persistent with the MESSAGE END statement, it can be sent to the master database using the MESSAGE FORWARD statement.

It is only necessary to specify the recipient of the message with keyword TO when a new replica is being registered with the master database; that is, when the first message from a replica to the master server is sent.

The *connect\_string* is a valid connect string, such as:

```
tcp [host_computer_name] server_port_number
```

For more information about connect strings, read the section of *IBM solidDB Administration Guide* titled "Communication Protocols".

In the context of a MESSAGE FORWARD command, a connect string must be delimited in single quotes.

The *node\_name* (without quotes) is a valid alphanumeric sequence that is not a reserved word. The "*node\_name*" (in double quote marks) is used if the node name is a reserved word; in this case, the double quotes ensure that the node name is treated as a delimited identifier. For example, since the word "master" is a reserved word, the word is placed in double quotes when it is used as a node name:

```
-- On master
SET SYNC NODE "master";
```

```
--On replica
MESSAGE refresh_severe_bugs2 FORWARD TO "master" TIMEOUT FOREVER;
```

Each sent message has a reply message. The `TIMEOUT` property defines how long the replica server will wait for the reply message.

If a `TIMEOUT` is not defined, the message is forwarded to the master and the replica does not fetch the reply. In this case the reply can be retrieved with a separate `MESSAGE GET REPLY` call.

If the reply of the sent message contains `REFRESHes` of large publications, the size of the `REFRESH`'s commit block, that is, the number of rows that are committed in one transaction, can be defined using the `COMMITBLOCK` property. This has a positive impact on the performance of the replica database. It is recommended that there are no on-line users accessing the database when the `COMMITBLOCK` property is being used.

As part of the `MESSAGE FORWARD` operation, you can specify table-level pessimistic locking when the reply message is initially executed in the replica. If the `PESSIMISTIC` mode is specified, all other concurrent access to the table affected is blocked until the synchronization message has completed. Otherwise, if the optimistic mode is used, the `MESSAGE FORWARD` operation may fail due to a concurrency conflict.

When a transaction acquires an exclusive lock to a table, the `TableLockWaitTimeout` parameter setting in the *General* section of the `solid.ini` configuration file determines the transaction's wait period until the exclusive or shared lock is released. For details, see the description of this parameter in *IBM solidDB Administration Guide*.

If a forwarded message fails in delivery due to a communication error, you must explicitly use the `MESSAGE FORWARD` to resend the message. Once re-sent, `MESSAGE FORWARD` re-executes the message.



#### Note

When working with the messages, be sure the autocommit mode is always switched off.

### B.59.3 Example

Forward message, wait for the reply for 60 seconds

```
MESSAGE MyMsg001 FORWARD TIMEOUT 60 ;
```

Forward message to a master server that runs on the "mastermachine.acme.com" machine. Do not wait for the reply message.

```
MESSAGE MyRegistrationMsg FORWARD TO
'tcp mastermachine.acme.com 1313' ;
```

Forward message, wait for the reply for 5 minutes (300 seconds) and commit the data of the refreshed publications to replica database in transactions of max. 1000 rows.

```
MESSAGE MyMsg001 FORWARD TIMEOUT 300 COMMITBLOCK 1000 ;
```

## B.59.4 Return Values from Replica

For details on each error code, see the appendix titled Error Codes in *IBM solidDB Administration Guide*.

**Table B.32. MESSAGE FORWARD Return Values from Replica**

Error Code	Description
13XXX	Table level error
21XXX	Communication error
10006	Concurrency conflict (simultaneous other operation)
13047	No privilege for operation
13056	Insert not allowed for pseudo column
25005	Message <i>message_name</i> is already active
25013	Message name <i>message_name</i> not found
25018	Illegal message state  In the replica, the message can only be executed using the MESSAGE FORWARD statement if the message is ended and the ending transaction is committed.
25024	Master not defined  This message is produced if double quotes, rather than single quotes, are used around the <i>connect_string</i> in a MESSAGE FORWARD statement.  For example, if the master node is given the node name "master" (which is a reserved word and therefore should be delimited by double quotes), and if that node's connect string is:  <b>tcp localhost 1315</b>

## B.59.4 Return Values from Replica

Error Code	Description
	<p>then the MESSAGE statements shown below are correct:</p> <pre>--On the replica ... --double quotes MESSAGE msg1 BEGIN TO "master"; ... --single quotes MESSAGE msg2 FORWARD TO 'tcp localhost 1315';</pre> <p>Note that the MESSAGE BEGIN statement defines (within the replica server) what the node name of the master is. The MESSAGE FORWARD statement may contain the connect string to the server.</p>
25026	Not a valid master user
25031	Transaction is active, operation failed
25035	<p>Message <i>message_name</i> is in use.</p> <p>A user is currently creating or forwarding this message.</p>
25040	<p>User id <i>user_id</i> is not found.</p> <p>While executing a message reply an attempt to map a master user to a local replica id failed.</p>
25041	Subscription to publication <i>publication_name</i> not found
25048	Publication <i>publication_name</i> request info not found
25052	Failed to set node name to <i>node_name</i> .
25054	Table <i>table_name</i> is not set for synchronization history
25055	<p>Connect information is allowed only when not registered</p> <p>The connect info in MESSAGE <i>message_name</i> FORWARD TO <i>connect_info options</i> is allowed only if the replica has not yet been registered to the master database.</p>
25056	Autocommit not allowed
25057	The replica database has already been registered to a master database



Error Code	Description
25060	Column <i>column_name</i> does not exist on publication <i>publication_name</i> resultset in table <i>table_name</i>

## B.59.5 Return Values from Master

For details on each error code, see the appendix titled Error Codes in *IBM solidDB Administration Guide*.

**Table B.33. MESSAGE FORWARD Return Values from Master**

Error code	Description
13XXX	Table level error
13124	User id <i>num</i> not found  This message is generated, for example, if the user has been dropped.
25016	Message not found, replica ID <i>replica_id</i> , message ID <i>message_id</i>
25056	Autocommit not allowed

### Result Set

If the MESSAGE FORWARD also retrieves the reply, the statement returns a result set. The result set returned is the same as the one returned with the statement MESSAGE GET REPLY. See Section B.63, “MESSAGE GET REPLY”.

## B.60 MESSAGE FROM REPLICA DELETE

```
MESSAGE msgid FROM REPLICA replicaname DELETE;
MESSAGE msgid FROM REPLICA replicaname DELETE CURRENT TRANSACTION;
```

This command can only be executed on the master.

## B.61 MESSAGE FROM REPLICA EXECUTE

```
MESSAGE message_name FROM REPLICA replica_name EXECUTE
```

## B.61.1 Supported in

This command requires IBM solidDB advanced replication.

## B.61.2 Usage

The execution of a message stops if a DBMS level error such as a duplicate insert occurs during the execution or if an error is raised from a procedure by putting the `SYS_ROLLBACK` parameter to the transactions bulletin board. This kind of error is recoverable by fixing the reason for the error, for example, by removing the duplicate row from the database, and then executing the message.

When the transaction in error is deleted with `MESSAGE DELETE CURRENT TRANSACTION`, the deletion is completed first before the `MESSAGE FROM REPLICA EXECUTE` command is executed; that is, the statement starts the message from replica, but waits until the active statement is completed before actually executing the message. Thus the command performs asynchronous message execution.



### Note

When working with the messages, be sure the autocommit mode is always switched off.

## B.61.3 Usage in Master

Use this command in the master to execute a failed message.

## B.61.4 Usage in Replica

This command is not available in the replica. See Section B.58, “MESSAGE EXECUTE” for an alternative.

## B.61.5 Example

```
MESSAGE MyMsg0002 FROM REPLICA bills_laptop EXECUTE;
```

## B.61.6 Return Values

For details on each error code, see the appendix titled Error Codes in *IBM solidDB Administration Guide*.

**Table B.34. MESSAGE FROM REPLICA EXECUTE Return Values**

Error code	Description
13047	No privilege for operation
25009	Replica <i>replica_name</i> not found
25013	Message name <i>message_name</i> not found
25018	Illegal message state. An attempt was made to delete a transaction from a message that is not in error.
25056	Autocommit not allowed

## B.62 MESSAGE FROM REPLICA RESTART

```
MESSAGE msgid FROM REPLICA replicaname RESTART <err-options>;
```

Where <*err-options*> can be IGNORE\_ERRORS or LOG\_ERRORS or FAIL\_ERRORS

This command can only be executed on the master.

This command allows you to re-execute a failed transaction that has been stored in the system tables and that can be retrieved using the SYNC\_FAILED\_MESSAGES view.

## B.63 MESSAGE GET REPLY

```
MESSAGE unique_message_name GET REPLY
[TIMEOUT {FOREVER | seconds}]
[COMMITBLOCK block_size_in_rows]
[NO EXECUTE]
[{OPTIMISTIC | PESSIMISTIC}]
```

### B.63.1 Supported in

This command requires IBM solidDB advanced replication.

## B.63.2 Usage

If the reply to a sent message has not been received by the MESSAGE FORWARD statement, it can be requested separately from the master database by using the MESSAGE GET REPLY statement in the replica database.

If the reply message contains REFRESHes of large publications, the size of the REFRESH's commit block, that is, the number of rows that are committed in one transaction, can be limited using the COMMITBLOCK property. This has a positive impact on the performance of the replica database. It is recommended that there are no on-line users in the database when the COMMITBLOCK property is in use.

If the execution of a reply message with the COMMITBLOCK property fails in the replica database, it cannot be re-executed. The failed message must be deleted from the replica database and refreshed from the master database.

If NO EXECUTE is specified, when the reply message is available at the master, it is only read and stored for later execution. Otherwise, the reply message is downloaded from the master and executed in the same statement. Using NO EXECUTE reduces bottlenecks in communication lines by allowing reply messages for later execution in different transactions.

You can define the reply message to use table-level pessimistic locking when it is initially executed. If the PESSIMISTIC mode is specified, all other concurrent access to the table affected is blocked until the synchronization message has completed. Otherwise, if the optimistic mode is used, the MESSAGE GET REPLY operation may fail due to a concurrency conflict.

When a transaction acquires an exclusive lock to a table, the *TableLockWaitTimeout* parameter setting in the *General* section of the *solid.ini* configuration file determines the transaction's wait period until the exclusive or shared lock is released. For details, see the description of this parameter in *IBM solidDB Administration Guide*.

If a reply message fails in delivery due to a communication error (without COMMITBLOCK), you must explicitly use the MESSAGE GET REPLY to resend the message. Once resent, MESSAGE GET REPLY re-executes the message.



### Note

When working with the messages, be sure the autocommit mode is always switched off.

## B.63.3 Usage in Master

MESSAGE GET REPLY cannot be used in the master.

## B.63.4 Usage in Replica

Use MESSAGE GET REPLY in the replica to fetch a reply of a message from the master.

## B.63.5 Example

```
MESSAGE MyMessage001 GET REPLY TIMEOUT 120
MESSAGE MyMessage001 GET REPLY TIMEOUT 300 COMMITBLOCK 1000
```

## B.63.6 Return Values from Replica

Fatal errors in transaction propagation abort the message and return an error code to the replica. To propagate the aborted message you need to correct the fatal errors and restart the message with command MESSAGE FROM REPLICA EXECUTE.

If a REFRESH fails in the master, an error message about the failed REFRESH is added to the result set. Other parts of the message are executed normally. The failed REFRESH must be REFRESHed from the master in a separate synchronization message.

If a REFRESH (that is, the execution of the reply message) fails in the replica, the message is still available in the replica database and can be restarted with the MESSAGE *msg\_name* EXECUTE command.

For details on each error code, see the appendix titled Error Codes in *IBM solidDB Administration Guide*.

**Table B.35. MESSAGE GET REPLY Return Values from Replica**

Error code	Description
13XXX	Table level error
13124	User id <i>num</i> not found  This message is generated, for example, if the user has been dropped.
10006	Concurrency conflict (simultaneous other operation)
13047	No privilege for operation
13056	Insert not allowed for pseudo column
21XXX	Communication error
25005	Message <i>message_name</i> is already active
25013	Message name <i>message_name</i> not found

## B.63.7 Return Values from Master

Error code	Description
25018	Illegal message state  In the replica, the message can only be executed using the MESSAGE GET REPLY statement if the message is forwarded to the master.
25024	Master not defined
25026	Not a valid master user
25031	Transaction is active, operation failed
25035	Message <i>message_name</i> is in use. A user is currently creating or forwarding this message.
25036	Publication <i>publication_name</i> not found or publication version mismatch
25040	User id <i>user_id</i> is not found  While executing a message reply, an attempt to map a master user to a local replica id failed.
25041	Subscription to publication <i>publication_name</i> not found
25048	Publication <i>publication_name</i> request info not found
25054	Table <i>table_name</i> is not set for synchronization history
25056	Autocommit not allowed
25057	Already registered to master <i>master_name</i>
25060	Column <i>column_name</i> does not exist on publication <i>publication_name</i> resultset in table <i>table_name</i>

## B.63.7 Return Values from Master

For details on each error code, see the appendix titled Error Codes in *IBM solidDB Administration Guide*.

**Table B.36. MESSAGE GET REPLY Return Values from Master**

Error code	Description
13XXX	Table level error
13124	User id <i>num</i> not found  This message is generated, for example, if the user has been dropped.
25012	Message reply timed out

Error code	Description
25016	Message not found, replica id <i>replica-id</i> , message id <i>message-id</i>
25043	Reply message is too long ( <i>size_of_messages</i> bytes). Maximum is set to <i>max_message_size</i> bytes.
25056	Autocommit not allowed

## B.63.8 Result Set

MESSAGE GET REPLY returns a result set table. The columns of the result set are as follows:

**Table B.37. MESSAGE GET REPLY Result Set Table**

Column Name	Description
Partno	Message part number
Type	The type of result set row. Possible types are: 0: Message part start 1: This type is not in use 2: The message was a propagation message and the status of that operation is stored in the return message 3: Task 4: Subscription task 5: Type of refresh (FULL or INCREMENTAL) 6: MESSAGE DELETE status
Masterid	Master ID
Msgid	Message ID
Errcode	Message error code. Zero if successful.
Errstr	Message error string. NULL is successful.
Insertcount	Number of inserted rows to replica. Type=3: Total number of insert Type=4: Row inserts restored from replica history to replica base table

Column Name	Description
	Type=5: Insert operations received from master
Deletecount	Type = 3: Total number of deletes Type = 4: Row deletes restored from replica base table Type = 5: Delete operations received from master
Bytecount	Size of message in bytes. Indicated in result received from command MESSAGE END. Otherwise 0.
Info	Information of the current task. Type = 0: then Message name Type = 3: Publication name Type = 4: Table name Type = 5: FULL/INCREMENTAL

## B.64 POST EVENT

The POST EVENT command is allowed only inside stored procedures. See Section B.14, “CREATE PROCEDURE” for more details.

## B.65 PUT\_PARAM()

```
put_param(param_name, param_value)
```

### B.65.1 Supported in

This command requires IBM solidDB advanced replication.

### B.65.2 Usage

With IBM solidDB Intelligent Transaction, SQL statements or procedures of a transaction can communicate with each other by passing parameters to each other using a parameter bulletin board. The bulletin board is a storage of parameters that is visible to all statements of a transaction.



Parameters are specific to a catalog. Different replica and master catalogs have their own set of bulletin board parameters that are not visible to each other.

Use the `put_param()` function to place a parameter on the bulletin board. If the parameter already exists, the new value overwrites the previous one.

These parameters are not propagated to the master. You can use the `SAVE PROPERTY` statement to propagate properties from the replica to the master. For details, read Section B.73, “`SAVE PROPERTY`”.

Because `put_param()` is a SQL function, it can be used only within a procedure or in a SQL statement.

Both the parameter name and value are of type `VARCHAR`.

### **B.65.3 Usage in Master**

`Put_param()` function can be used in the master for setting parameters to the parameter bulletin board of the current transaction.

### **B.65.4 Usage in Replica**

`Put_param()` function can be used in replicas for setting parameters to the parameter bulletin board of the current transaction.

### **B.65.5 Differences between "PUT\_PARAM( )" and "SAVE PROPERTY property\_name VALUE property\_value;"**

You typically use `put_param` inside the (running) transaction to pass parameters between procedures. These parameter values disappear from the bulletin board when the transaction terminates (commits or rolls back).

You typically use the `SAVE PROPERTY` statement in the replica to set properties for the entire transaction. These properties can be used in the `WHERE` clause of the `PROPAGATE TRANSACTIONS` statement. When the transaction is executed in the master, the properties of the transaction are put to the parameter bulletin board of the transaction in the beginning of the transaction. Hence, they can be accessed by all procedures of the transaction by using the `GET_PARAM(param_name)` function.

## B.65.6 Example

```
Select put_param('myparam', '123abc');
```

## B.65.7 Return Values

For details on each error code, see the appendix titled Error Codes in *IBM solidDB Administration Guide*.

**Table B.38. PUT\_PARAM() Return Values**

Error Code	Description
13086	Invalid data type in a parameter.

When executed successfully, `put_param()` returns the new value of the assigned parameter.

## B.65.8 See Also

GET\_PARAM

SAVE PROPERTY

SET SYNC PARAMETER

## B.66 REFRESH

```
REFRESH publication [parameters] [FULL]  
[OPTIMISTIC|PESSIMISTIC]  
[COMMITBLOCK number_of_rows]  
[TIMEOUT {DEFAULT | FOREVER | timeout_ms} ]
```

### B.66.1 Usage

The REFRESH statement is a storageless refresh command. It conserves memory by streaming the associated data. It also saves I/O bandwidth because no messages are written to disk. Each command blocks until it is successfully executed.

The optional properties OPTIMISTIC | PESSIMISTIC define the way the replica table is being locked.

- The `OPTIMISTIC` mode (the default value) defines that the concurrency control method depends on the table type and the isolation level. For D-tables in the `OPTIMISTIC` mode, the `REFRESH` will always succeed. For M-tables in general, and for D-tables in the `PESSIMISTIC` mode, row-level locking will be used. If a lock cannot be obtained, `PESSIMISTIC` fails and returns an error.
- `PESSIMISTIC` defines that the table is exclusively locked, regardless of the table type and isolation level chosen, for the time of refresh. If the lock cannot be obtained, the refresh request fails and returns an error.

If the reply to the `REFRESH` request contains `REFRESHes` of large publications, the size of the `REFRESH`'s commit block, that is, the number of rows that are committed in one transaction, can be defined using the `COMMITBLOCK` property. This has a positive impact on the performance of the replica database. It is recommended that there are no on-line users accessing the database when the `COMMITBLOCK` property is being used.

If `COMMITBLOCK` is not used, the execution of `REFRESH` is a part of the current transaction. The effect of `REFRESH` can be revoked by issuing the `ROLLBACK` command. In order to make the effect of `REFRESH` durable, `COMMIT WORK` has to be issued. `REFRESH` is idempotent in the sense that it can be issued repeatedly, over the rollbacks and commits, and the effects are (in the quiescent state of the database) always the same.

If the `COMMITBLOCK` clause is used, each transfer part (of the specified size) is committed in Replica implicitly. The `ROLLBACK` statement removes the effect of the latest transfer part only. `COMMIT WORK` commits the last transfer part.

The `TIMEOUT` property defines how long the replica server will wait for the reply message. If `TIMEOUT` is not defined, then `FOREVER` is used.

## B.66.2 Example

*Synchronous, messageless refresh:*

```
REFRESH publ_states;  
PESSIMISTIC;  
COMMITBLOCK 1000;  
COMMIT WORK;
```

## B.66.3 Return Values

For details on each error code, see the appendix titled Error Codes in *IBM solidDB Administration Guide*.

**Table B.39. REFRESH Return Values**

<b>Error Code</b>	<b>Description</b>
13133	Not a valid license for this product
25004	Dynamic parameters are not supported
25015	Syntax error: <i>error_message</i> , line <i>line_number</i>
25024	Master not defined
25025	Node name not defined
25026	Not a valid master user
25044	SYNC_CONFIG system publication takes only character arguments
25056	Autocommit not allowed
25071	Not registered to publication <i>publication_name</i>
25072	Already registered to publication <i>publication_name</i>
13XXX	Table level error
21XXX	Communication error
10006	Concurrency conflict (simultaneous other operation)
13047	No privilege for operation
13056	Insert not allowed for pseudo column
25005	Message <i>message_name</i> is already active
25018	Illegal message state  In the replica, the message can only be executed using the MESSAGE FORWARD statement if the message is ended and the ending transaction is committed.
25024	Master not defined  This message is produced if double quotes, rather than single quotes, are used around the <i>connect_string</i> in a MESSAGE FORWARD statement.  For example, if the master node is given the node name "master" (which is a reserved word and therefore should be delimited by double quotes), and if that node's connect string is:  <b>tcp localhost 1315</b>  then the MESSAGE statements shown below are correct:

Error Code	Description
	<pre> --On the replica ... --double quotes MESSAGE msg1 BEGIN TO "master"; ... --single quotes MESSAGE msg2 FORWARD TO 'tcp localhost 1315'; </pre> <p>Note that the MESSAGE BEGIN statement defines (within the replica server) what the node name of the master is. The MESSAGE FORWARD statement may contain the connect string to the server.</p>
25026	Not a valid master user
25031	Transaction is active, operation failed
25035	<p>Message <i>message_name</i> is in use.</p> <p>A user is currently creating or forwarding this message.</p>
25040	<p>User id <i>user_id</i> is not found.</p> <p>While executing a message reply an attempt to map a master user to a local replica id failed.</p>
25041	Subscription to publication <i>publication_name</i> not found
25048	Publication <i>publication_name</i> request info not found
25052	Failed to set node name to <i>node_name</i> .
25054	Table <i>table_name</i> is not set for synchronization history
25055	<p>Connect information is allowed only when not registered</p> <p>The connect info in MESSAGE <i>message_name</i> FORWARD TO <i>connect_info_options</i> is allowed only if the replica has not yet been registered to the master database.</p>
25056	Autocommit not allowed
25057	The replica database has already been registered to a master database
25060	Column <i>column_name</i> does not exist on publication <i>publication_name</i> resultset in table <i>table_name</i>
13XXX	Table level error

Error Code	Description
13124	User id <i>num</i> not found  This message is generated, for example, if the user has been dropped.
25056	Autocommit not allowed

## B.67 REGISTER EVENT

Registering an event tells the server that you would like to be notified of all future occurrences of this event, even if you are not yet waiting for it. By separating the "register" and "wait" commands, you can start queuing events immediately, while waiting until later to actually start processing them.

Note that you do not need to register for every event before waiting for it. When you wait on an event, you will be registered implicitly for that event if you did not already explicitly register for it. Thus you only need to explicitly register events if you want them to start being queued now but you don't want to start WAITing for them until later.

You cannot register to synchronization events, because the ADMIN EVENT 'wait' command is not able to return variable resultsets. Instead, you must use stored procedures to handle synchronization events.

The REGISTER EVENT command is allowed only inside stored procedures. See the CREATE PROCEDURE statement and the CREATE EVENT statement for more details.

## B.68 REVOKE (Role from User)

```
REVOKE { role_name [, role_name ]... }  
      FROM { PUBLIC | user_name [, user_name ]... }
```

### B.68.1 Usage

The REVOKE statement is used to take a role away from users.

## B.68.2 Example

```
REVOKE GUEST_USERS FROM HOBBS;
```

## B.69 REVOKE (Privilege from Role or User)

```
REVOKE
  {ALL | revoke_privilege [, revoke_privilege]... } ON table-name
  FROM {PUBLIC | user_name [, user_name]... | role_name [, role_name]... }
```

```
revoke-privilege ::= DELETE | INSERT | SELECT |
  UPDATE [( column_identifier [, column_identifier]... )] |
  REFERENCES
```

```
REVOKE EXECUTE ON procedure_name
  FROM {PUBLIC | user_name [, user_name]... | role_name [, role_name]... }
```

```
REVOKE {SELECT | INSERT} ON event_name FROM
  {PUBLIC | user_name [, user_name]... | role_name [, role_name]... }
```

```
REVOKE {SELECT | INSERT} ON sequence_name
  FROM {PUBLIC | user_name [, user_name]... | role_name [, role_name]... }
```



### Note

IBM solidDB does not support the keywords CASCADE and RESTRICT in REVOKE statements.

### B.69.1 Usage

The REVOKE statement is used to take privileges away from users and roles.

## B.69.2 Example

```
REVOKE INSERT ON TEST FROM GUEST_USERS;
```

## B.69.3 See Also

For more information about user privileges, see also:

- Section B.47, “GRANT”, and
- Section 5.2, “Managing User Privileges and Roles”.

## B.70 REVOKE REFRESH

```
REVOKE { REFRESH | SUBSCRIBE } ON publication_name FROM { PUBLIC |  
    user_name, [ user_name ] ... |  
    role_name , [ role_name ] ... }
```

### B.70.1 Supported in

This command requires IBM solidDB advanced replication.

### B.70.2 Usage

This statement revokes access rights to a publication from a user or role defined in the master database.



#### Note

The keywords "REFRESH" and "SUBSCRIBE" are synonymous. However, "SUBSCRIBE" is deprecated in the REVOKE statement.

### B.70.3 Usage in Master

Use this statement to revoke access rights to a publication from a user or role.

### B.70.4 Usage in Replica

This statement is not available in a replica database.



## B.70.5 Example

```
REVOKE REFRESH ON customers_by_area FROM joe_smith;
REVOKE REFRESH ON customers_by_area FROM all_salesmen;
```

## B.70.6 Return Values

**Table B.40. REVOKE REFRESH Return Values**

Error Code	Description
13137	Illegal grant/revoke mode
13048	No grant option privilege
25010	Publication <i>name</i> not found

## B.71 ROLLBACK WORK

```
ROLLBACK WORK
```

### B.71.1 Usage

The changes made in the database by the current transaction are discarded by the ROLLBACK WORK statement. It terminates the transaction.

### B.71.2 Example

```
ROLLBACK WORK;
```

## B.72 SAVE

```
SAVE [NO CHECK] [ { IGNORE_ERRORS | LOG_ERRORS | FAIL_ERRORS } ]
[ { AUTOSAVE | AUTOSAVEONLY } ] sql_statement
```

### B.72.1 Supported in

This command requires IBM solidDB advanced replication.

## B.72.2 Usage

The statements of a transaction that need to be propagated to the master database must be explicitly saved to the transaction queue of the replica database. Adding a SAVE statement before the transaction statements does this.

Only master users are allowed to save statements. This is because when the saved statements are executed on the master, they must be executed using the appropriate access rights of a user on the master. The saved statements are executed in the master database using the access rights of the master user that was active in the replica when the statement was saved. If a user in the replica was mapped to a user in the master, the SAVE statement uses the access rights of the user in the master.

The default behavior for error handling with transaction propagation is that a failed transaction halts execution of the message; this aborts the currently-executing transaction and prevents execution of any subsequent transactions that are in that same message. However, you may choose a different error-handling behavior.

The options for the SAVE command are explained below:

**NO CHECK:** This option means that the statement is not prepared in the replica. This option is useful if the command would not make sense on the replica. For example, if the SQL command calls a stored procedure that exists on the master but not on the replica, then you don't want the replica to try to prepare the statement. If you use this option, then the statement can not have parameter markers.

**IGNORE\_ERRORS:** This option means that if a statement fails while executing on the master, then the failed statement is ignored and the transaction is aborted. However, only the transaction, not the entire message, is aborted. The master continues executing the message, resuming with the first transaction after the failed one.

**LOG\_ERRORS:** This means that if a statement failed while executing on the master, then the failed statement is ignored and the current transaction is aborted. The failed transaction's statements are saved in SYS\_SYNC\_RECEIVED\_STMTS system table for later execution or investigation. The failed transactions can be examined using SYNC\_FAILED\_MESSAGES system view and they can be re-executed from there using MESSAGE <msg\_id> FROM REPLICA <replica\_name> RESTART -statement.

Note that, as with the IGNORE\_ERROR option, aborting the transaction does not abort the entire message. The master continues executing the message, resuming with the first transaction after the failed one.

**FAIL\_ERRORS:** This option means that if a statement fails, the master stops executing the message. This is the default behavior.

**AUTOSAVE:** This option means that the statement is executed in the master and automatically saved for further propagation if the master is also a replica to some other master (i.e. a middle-tier node)

AUTOSAVEONLY: This option means that the statement is NOT executed in the master but instead is automatically saved for further propagation if the master is also a replica to some other master (i.e. is a middle-tier node)

## B.72.3 Usage in Master

This statement cannot be used in the master.

## B.72.4 Usage in Replica

Use this statement in the replica to save statements for propagation to the master.

## B.72.5 Example

```
SAVE INSERT INTO mytbl (col1, col2) VALUES ('calvin', 'hobbes')
SAVE CALL SP_UPDATE_MYTBL('calvin_1', 'hobbes')
SAVE CALL SP_DELETE_MYTBL('calvin')
SAVE NO CHECK IGNORE_ERRORS insert into mytab values(1,2)
```

## B.72.6 Return Values

For details on each error code, see the appendix titled Error Codes in *IBM solidDB Administration Guide*.

**Table B.41. SAVE Return Values**

Error Code	Description
25001	Internal error Master database has exceeded the database size limit required to save the statement.
25003	Cannot save SAVE statements
25070	Statement can be saved only for one master in transaction.

## B.73 SAVE PROPERTY

```
SAVE PROPERTY property_name VALUE 'value_string'
SAVE PROPERTY property_name VALUE NONE
SAVE DEFAULT PROPERTY property_name VALUE 'value_string'
SAVE DEFAULT PROPERTY property_name VALUE NONE
```

```
SAVE DEFAULT PROPAGATE PROPERTY WHERE name {=|<|<=|>|>=|<>} 'value'  
SAVE DEFAULT PROPAGATE PROPERTY NONE
```

## B.73.1 Supported in

This command requires IBM solidDB advanced replication.

## B.73.2 Usage

It is possible to assign properties to the current active transaction with the following command:

```
SAVE PROPERTY property_name VALUE 'value_string'
```

The statements of the transaction in the master database can access these properties by calling the GET\_PARAM( ) function. Properties are only available in the replica database that apply to the command

```
MESSAGE APPEND unique_message_name PROPAGATE TRANSACTIONS  
WHERE property > 'value_string'
```

When the transaction is executed in the master database, the saved properties are placed on the parameter bulletin board of the transaction. If the saved property already exists, the new value overwrites the previous one.

It is also possible to define default properties that are saved to all transactions of the current connection. The statement for this is:

```
SAVE DEFAULT PROPERTY property_name VALUE 'value_string'
```

A SAVE DEFAULT PROPAGATE PROPERTY WHERE statement can be used to save default transaction propagation criteria. This can be used for example to set the propagation priority of transactions created in the current connection.

SAVE DEFAULT PROPAGATE PROPERTY WHERE *property* > '*value*' can be used in a connection level to append all MESSAGE *unique\_message\_name* APPEND PROPAGATE TRANSACTIONS statements to have the default WHERE statement. If the WHERE statement is entered also in the PROPAGATE statement, it will override the statement set with the DEFAULT PROPAGATE PROPERTY.

A property or a default property can be removed by re-saving the property with value string NONE.

## B.73.3 Usage in Master

This statement cannot be used in the master database.

## B.73.4 Usage in Replica

You can use these statements in the replica to set properties for a transaction that is saved for propagation to the master. The property's value can be read in the master database.

## B.73.5 Differences between "PUT\_PARAM( )" and "SAVE PROPERTY property\_name VALUE property\_value;"

See the description of the PUT\_PARAM( ) function for a discussion of the differences between "SAVE PROPERTY" and "PUT\_PARAM()".

## B.73.6 Example

```
SAVE PROPERTY conflict_rule VALUE 'override'  
SAVE DEFAULT PROPERTY userid VALUE 'scott'  
SAVE DEFAULT PROPERTY userid VALUE NONE  
SAVE DEFAULT PROPAGATE PROPERTY WHERE priority > '2'
```

## B.73.7 Return Values

For details on each error code, see the appendix titled Error Codes in *IBM solidDB Administration Guide*.

**Table B.42. SAVE PROPERTY Return Values**

Error Code	Description
13086	Invalid data type in a parameter.

## B.73.8 Result set

SAVE PROPERTY does not return a result set.

## B.74 SELECT

```
SELECT [ALL | DISTINCT] select-list
    LEVEL
    FROM table_reference_list
    [WHERE search_condition]
    [GROUP BY column_name [, column_name]... ]
    [HAVING search_condition]
    [hierarchical_condition]
    [[UNION | INTERSECT | EXCEPT] [ALL] select_statement]...
    [ORDER BY expression]
    [ASC | DESC]]
    [LIMIT row_count [OFFSET skipped_rows] | LIMIT skipped_rows,row_count]
hierarchical_condition ::=
START WITH search_condition CONNECT BY [PRIOR] search_condition
```

### B.74.1 Usage

The SELECT statement allows you to select 0 or more records from one or more tables.

The non-standard clause `LIMIT row_count OFFSET skipped_rows` allows to mask out a portion of a result set with a sliding window having the size of *row\_count* and positioned at the *skipped\_rows*+1 row. A negative value of *skipped\_rows* results in an error, while the negative value of *row\_count* results in the whole result set produced. Note that two forms are available: for example `LIMIT 24 OFFSET 10` is equal to `LIMIT 10, 24`.

If your table contains hierarchical data, you can select rows in a hierarchical order using a hierarchical query clause. In a hierarchical query clause, `START WITH` specifies the root row(s) of the hierarchy and `CONNECT BY` specifies the relationship between parent rows and child rows of the hierarchy. The `CONNECT BY` condition cannot contain a subquery.

`LEVEL` is a pseudocolumn valid in the context of the hierarchical query only. If the result set is viewed as a tree of interreferenced rows, the `LEVEL` column produces the tree level number, assigning "1" to the top-level row.

`ORDER SIBLINGS BY` causes the rows at any level to be ordered accordingly.

In a hierarchical query, one expression in the condition must be qualified with the `PRIOR` operator to refer to the parent row. `PRIOR` is a unary operator and has the same precedence as the unary + and - arithmetic operators. It evaluates the immediately following expression for the parent row of the current row in a hier-

archical query. PRIOR is most commonly used when comparing column values with the equality operator. The PRIOR keyword can be on either side of the operator.

## B.74.2 Examples

```
SELECT ID FROM TEST;
SELECT DISTINCT ID, C FROM TEST WHERE ID = 5;
SELECT DISTINCT ID FROM TEST ORDER BY ID ASC;
SELECT NAME, ADDRESS FROM CUSTOMERS
UNION
SELECT NAME, DEP FROM PERSONNEL;
SELECT dept, count(*) FROM person
GROUP BY dept
ORDER BY dept
LIMIT 20 OFFSET 10
```

## B.74.3 START WITH Example

```
SELECT last_name, employee_id, manager_id, LEVEL
FROM employees
START WITH employee_id = 100
CONNECT BY PRIOR employee_id = manager_id
ORDER SIBLINGS BY last_name;
```

LAST_NAME	EMPLOYEE_ID	MANAGER_ID
King	100	
Cambrault	148	100
Bates	172	148
Bloom	169	148
Fox	170	148
Kumar	173	148
Ozer	168	148
Smith	171	148
De Haan	102	100
Hunold	103	102
Austin	105	103
Ernst	104	103

Lorentz	107	103
Pataballa	106	103
Errazuriz	147	100
Ande	166	147
Banda	167	147

## B.74.4 LEVEL and ORDER SIBLINGS BY Example

```
SELECT last_name, employee_id, manager_id, LEVEL
FROM employees
START WITH last_name = 'King'
CONNECT BY PRIOR employee_id = manager_id
ORDER SIBLINGS BY last_name
ORDER BY LEVEL;
```

LAST_NAME	EMPLOYEE_ID	MANAGER_ID	LEVEL
-----	-----	-----	-----
King	100	NULL	1
Cambrault	148	100	2
De Haan	102	100	2
Bates	172	148	3
Bloom	169	148	3
Gates	104	148	3
Hunold	103	102	3
Hope	202	172	4
Smith	201	172	4

## B.75 SET

### B.75.1 Usage

SET commands apply to the user session (connection) in which they are executed. They do not affect other user sessions.

SET statements may be issued at any time; however, they do not all take effect immediately. The following statements take effect immediately:

- SET CATALOG
- SET IDLE TIMEOUT



- SET SCHEMA
- SET STATEMENT MAXTIME

The following statements take effect after the next COMMIT WORK:

- SET DURABILITY
- SET OPTIMISTIC LOCK TIMEOUT
- SET LOCK TIMEOUT
- SET ISOLATION LEVEL
- SET { READ ONLY | READ WRITE | WRITE }

SET statements are not subject to rollback, i.e. they remain in force even if the transaction they have been issued in has been aborted or rolled back. It is a good practice to issue them before any DDL/DML SQL statement in a transaction.

The settings continue in effect until the end of the session (connection) or until another SET command changes the settings, or in some cases until a higher-precedence command (e.g. SET TRANSACTION) is executed.

### **B.75.2 Differences between SET and SET TRANSACTION**

IBM solidDB SQL gives you two different commands to set the transaction isolation level, the read level, and the durability level. In addition to the SET command described in this section

```
SET { READ ONLY | READ WRITE | WRITE };  
SET ISOLATION LEVEL {READ COMMITTED...};  
SET DURABILITY ...;
```

there is also the SET TRANSACTION command described in Section B.75.13, “SET TRANSACTION”.

```
SET TRANSACTION { READ ONLY | READ WRITE | WRITE };  
SET TRANSACTION ISOLATION LEVEL {READ COMMITTED ...};  
SET TRANSACTION DURABILITY ...;
```

For information about the differences between these commands, see Section B.75.2, “Differences between SET and SET TRANSACTION”.

## Example B.1. SET Examples

```
SET CATALOG myCatalog;
SET DURABILITY STRICT;
SET IDLE TIMEOUT 30;
SET ISOLATION LEVEL REPEATABLE READ;
SET OPTIMISTIC LOCK TIMEOUT 30;
SET LOCK TIMEOUT 30;
SET LOCK TIMEOUT 500MS;
SET READ ONLY;
SET SCHEMA 'accounting_info';
SET SCHEMA 'john_smith';
SET STATEMENT MAXTIME 180;
```

## B.75.3 SET (Read/Write Level)

```
SET {READ ONLY | READ WRITE | WRITE}
```

SET {READ ONLY | READ WRITE | WRITE} allows you to specify whether the connection be allowed only to read, read and write, or whether it be allowed to write only.

See also Section B.75.6, “SET ISOLATION LEVEL”.

## B.75.4 SET CATALOG

```
SET CATALOG catalog_name
```

SET CATALOG sets the current catalog context in a connection.

## B.75.5 SET DURABILITY

```
SET DURABILITY { RELAXED | STRICT | DEFAULT }
```

SET DURABILITY sets the transaction durability level. For details about the possible settings, see the discussion of "Logging and Transaction Durability" in *IBM solidDB Administration Guide*.

## B.75.6 SET ISOLATION LEVEL

```
SET ISOLATION LEVEL {  
    READ COMMITTED |  
    REPEATABLE READ |  
    SERIALIZABLE }  
}
```

SET ISOLATION LEVEL allows you to specify the isolation level. For more information about isolation levels, see Section 6.2.6, “TRANSACTION ISOLATION Levels”.

If the assigned workload server is Secondary, it can be changed programmatically to the Primary. At the session level, the following statements change the workload connection server to the Primary:

```
SET WRITE (nonstandard)  
SET ISOLATION LEVEL REPEATABLE READ  
SET ISOLATION LEVEL SERIALIZABLE
```

The statement takes effect immediately, if it is a first statement of a transaction, or from the next transaction otherwise.

If the above statement is not applicable, it returns SQL\_SUCCESS, with no action performed. For example, such is the case when SET WRITE is applied to a standalone server. In that case the semantics of SET WRITE is equal to that of SET READ WRITE.

The effect of the SET WRITE statement may be reverted with the statement SET READ WRITE or ... READ ONLY (SQL:1999). Also, the isolation level statement has the same effect:

```
SET ISOLATION LEVEL READ COMMITTED
```

## B.75.7 SET SAFENESS

```
SET SAFENESS {1SAFE | 2SAFE | DEFAULT}
```

SET SAFENESS determines whether the replication protocol is synchronous (2-safe) or asynchronous (1-safe).

- 1-safe: the transaction is first committed at Primary and then transmitted to Secondary
- 2-safe: the transaction is not committed before it has been acknowledged by Secondary (default).

SET SAFENESS sets the safeness level for the current session.

## B.75.8 SET SCHEMA

```
SET SCHEMA { 'schema_name' | USER | 'user_name' }
```

### Usage

IBM solidDB supports SQL89 style schemas. Schemas are used to help uniquely identify entities (tables, views, etc.) within a database. By using schemas, each user may create entities without worrying about whether her names overlap the names chosen by other users/schemas.

To uniquely identify an entity (such as a table), you "qualify" it by specifying the catalog name and schema name. Below is an example of a fully-qualified table name:

```
FinanceCatalog.AccountsReceivableSchema.CustomersTable
```

In keeping with the ANSI SQL-92 standard, the `user_name` or `schema_name` may be enclosed in single quotes.

The default schema can be changed with the SET SCHEMA statement. The schema can be changed to the current user name by using the SET SCHEMA USER statement. Alternatively, the schema can be set to `'user_name'` which must be a valid user name in the database.

The algorithm to resolve entity names [`schema_name`].`table_identifier` is the following:

1. If `schema_name` is given, then `table_identifier` is searched only from that schema.
2. If `schema_name` is not given, then
  - a. First `table_identifier` is searched from default schema. Default schema is initially the same as user name, but can be changed with SET SCHEMA statement
  - b. Then `table_identifier` is searched from all schemas in the database. If more than one entity with same `identifier` and type (table, stored procedure, ...) is found, a new error code 13110 (Ambiguous entity name `table_identifier`) is returned.

The SET SCHEMA statement affects only the default entity name resolution and it does not change any access rights to database entities. It sets the default schema name for unqualified names in statements that are prepared in the current session by an EXECDIRECT statement or a prepare statement.

## Example

```
SET SCHEMA 'CUSTOMERS';
```

## See Also

Catalogs are also used to qualify (uniquely identify) the names of tables and other database entities, so you may also want to read about the SET CATALOG command.

## B.75.9 SET SQL

```
SET SQL INFO {ON | OFF} [FILE {file_name | "{file_name" | '{file_name'}]  
    [LEVEL info_level]  
SET SQL SORTARRAYSIZE {array-size | DEFAULT}  
SET SQL JOINPATHSPAN {  
path-span  
    | DEFAULT}  
SET SQL CONVERTORSTOUNIONS  
    {YES [COUNT  
value  
] | NO | DEFAULT}
```

## Usage

All the settings are read per user session (unlike the settings in the `solid.ini` file, which are automatically read each time IBM solidDB is started).

**SET SQL INFO** The SET SQL INFO command allows you to turn on trace information that may allow you to debug problems or tune queries. For SQL INFO, the default file is a global `soltrace.out` shared by all users. If the file name is given, all future INFO ON settings will use that file unless a new file is set. It is recommended that the file name is given in single quotes, because otherwise the file name is converted to uppercase. The info output is appended to the file and the file is never truncated, so after the info file is not needed anymore, the user must manually delete the file. If the file open fails, the info output is silently discarded.

The default SQL INFO LEVEL is 4. A good way to generate useful info output is to set info on with a new file name and then execute the SQL statement using EXPLAIN PLAN FOR syntax. This method gives all necessary estimator information but does not generate output from the fetches (which may generate a huge output file).

**SET SQL SORTARRAYSIZE** This command sets the size of the array that SQL uses when ordering the result set of a query. The units are "rows" — e.g. if you specify a value of 1000, then the server will create an array big enough to sort 1000 rows.

**SET SQL JOINPATHSPAN** This command is obsolete. The syntax is accepted, but the command has no effect.

**SET SQL CONVERTORSTOUNIONS** allows you to convert a query that contains "OR" operations into an equivalent query that uses "UNION" operations. The following operations are logically equivalent:

```
select ... where x = 1 OR y = 1;  
select ... where x = 1 UNION select... where y = 1;
```

By setting **CONVERTORSTOUNIONS**, you tell the optimizer that it may use equivalent UNION operations instead of OR operations if the UNIONS seem more efficient based on the volume and distribution of data. The **COUNT** parameter in **SQL CONVERTORSTOUNIONS** ("Convert ORs to UNIONS") specifies the maximum number of OR operations that may be converted to UNION operations. Note that you can also specify **CONVERTORSTOUNIONS** by using the `solid.ini` configuration parameter named *ConvertORsToUNIONS* (for details, see the description of this parameter in *IBM solidDB Administration Guide*). The default value is 100, which should be enough in almost all cases.

### Example

```
SET SQL INFO ON FILE 'sqlinfo.txt' LEVEL 5
```

## B.75.10 SET STATEMENT MAXTIME

```
SET STATEMENT MAXTIME minutes
```

**SET STATEMENT MAXTIME** sets connection-specific maximum execution time in minutes. The setting is effective until a new maximum time is set. Zero time means no maximum time, which is also the default.

## B.75.11 SET SYNC

The following chapters describe different **SET SYNC** commands.

### B.75.11.1 SET SYNC master\_or\_replica

```
SET SYNC master_or_replica yes_or_no
```

where:

```
master_or_replica ::= MASTER | REPLICa  
yes_or_no ::= YES | NO
```

#### Supported in

This command requires IBM solidDB advanced replication.

#### Usage

When a database catalog is created and configured for synchronization use, you must use this command to specify whether the database is a master, replica, or both. Only a DBA or a user with SYS\_SYNC\_ADMIN\_ROLE can set the database role.

The database catalog is a master database if there are replicas in the domain that refresh from publications from this database and/or propagate transactions to it. The database catalog is a replica catalog if it can refresh from publications that are in a master database. In multi-tier synchronization, intermediate level databases serve a dual role, as both master and replica databases.

Note that to use this command requires that you have already set the node name for the master or replica using the SET SYNC NODE command. For details, read Section B.75.11.4, “SET SYNC NODE”.

When you set the database for a dual role, you can use the statement once or twice. For example:

```
SET SYNC MASTER YES;  
SET SYNC REPLICa YES;
```

Note that when you set the database for dual roles, SET SYNC REPLICa YES does not override SET SYNC MASTER YES. Only the following explicit statement can override the status of the master database:

```
SET SYNC MASTER NO;
```

Once overridden, the current database is set as replica only.

## Examples

```
-- configure as replica
SET SYNC REPLICA YES;
-- configure as master
SET SYNC MASTER YES;
```

## Return Values

For details on each error code, see the appendix titled Error Codes in *IBM solidDB Administration Guide*.

**Table B.43. SET SYNC Return Values**

Error code	Description
13047	No privilege for operation
13107	Illegal set operation
13133	Not a valid license for this product
25051	Unfinished messages found

### B.75.11.2 SET SYNC CONNECT

```
SET SYNC CONNECT 'connect_string [,connect_string]' TO MASTER
master_name
SET SYNC CONNECT 'connect_string' TO REPLICA replica_name
```

## Supported in

This command requires IBM solidDB advanced replication.

## Usage

This statement changes the network name associated with the database name. Use this statement in a replica (or master) whenever you have changed network names in databases that a replica (or master) connects to. Network names are defined in the *Listen* parameter of the *solid.ini* configuration file.

The second connect string in SET SYNC CONNECT ... TO MASTER facilitates transparent failover of a Replica server to a standby Master server, should the Primary Master server fail. The order of the connect strings is not significant. The connection is automatically maintained to the currently active Primary server.



## Usage in Master

Use this statement in a master to change the replica's network name.

## Usage in Replica

Use this statement in a replica to change the master's network name.

## Example

```
SET SYNC CONNECT 'tcp server.company.com 1313' TO MASTER hq_master;
```

## Return Values

For details on each error code, see the appendix titled Error Codes in *IBM solidDB Administration Guide*.

**Table B.44. SET SYNC CONNECT Return Values**

Error code	Description
13047	No privilege for operation
13107	Illegal set operation
21300	Illegal network protocol
25007	Master <i>master_name</i> not found
25019	Database is not a replica database

### B.75.11.3 SET SYNC MODE

```
SET SYNC MODE { MAINTENANCE | NORMAL }
```

## Supported in

This command requires IBM solidDB advanced replication.

## Usage

This command sets the current catalog's sync mode to either Maintenance mode or Normal mode.

This command applies only to catalogs that are involved in synchronization (i.e. are "master" catalogs or "replica" catalogs, or are both master and replica in a hierarchy with 3 or more levels).

This command applies only to the current catalog. If you want to set more than one catalog's sync mode to Maintenance, then you will have to switch to each catalog (by using the SET CATALOG command) and then issue the SET SYNC MODE MAINTENANCE command for that catalog.

While a catalog's sync mode is Maintenance, the following rules apply:

- The catalog will not send or receive synchronization messages and therefore will not engage in synchronization activities (e.g. refresh or respond to a refresh request).
- DDL commands (e.g. ALTER TABLE) will be allowed on tables that are referenced by publications.
- When the sync mode changes, the server will send the system event SYNC\_MAINTENANCEMODE\_BEGIN or SYNC\_MAINTENANCEMODE\_END.
- If the master catalog's publications are altered (dropped and recreated) by using the REPLACE option, then the publication's metadata (internal publication definition data) is refreshed automatically to each replica the next time that replica refreshes from the changed publication. (This is true whether or not the database was in Maintenance sync mode when the publication was REPLACed.)
- Each catalog has a read-only parameter named SYNC\_MODE in the parameter bulletin board so that applications can check the catalog's mode. Values for that parameter are either 'MAINTENANCE' if the catalog is in maintenance sync mode or 'NORMAL' if the catalog is not in maintenance sync mode. The value is NULL if the catalog is not a master or a replica.
- The user must have DBA or synchronization administrations privileges to set sync mode to Maintenance or Normal.
- A user may have more than one catalog in Maintenance sync mode at a time.
- If the session that set the mode ON disconnects, then mode is set off.
- The normal synchronization history operations are disabled. For example, when a delete or update operation is done on a table that has synchronization history on, the synchronization history tables will not store the "original" rows (i.e. the rows before they were deleted or updated). Note, however, that deletes and updates apply to the synchronization history table; e.g.

```
DELETE * FROM T WHERE c = 5
```

will delete rows from the history table as well as from the base table. The table below shows how various operations (INSERT, DELETE, etc.) apply to the synchronization history tables in master and replica when sync mode is set to Maintenance.

**Table B.45. How Different Operations Apply to Synchronization History Tables**

Operation	Master	Replica
INSERT	Rows are inserted to base table.	Rows are inserted to base table and marked as official.
UPDATE	Both base table and history is updated.	Both base table and history is updated. Tentative/official status is not updated so tentative rows remains tentative and official rows remains official.
DELETE	Rows are deleted from base table and from history.	Rows are deleted from base table and from history.
Add, alter, drop column	Same operation is done to history also.	Same operation is done to history also.
Altering table mode	History mode is not altered	History mode is not altered
Create index	Same index is created to history also	Same index is created to history also
Create triggers	Triggers are not created on history	Triggers are not created on history

## Example

```
SET SYNC MODE MAINTENANCE SET SYNC MODE NORMAL
```

## Return Values

For details on each error code, see the appendix titled Error Codes in *IBM solidDB Administration Guide*.

**Table B.46. SET SYNC MODE Return Values**

Error code	Description
13047	No privilege for operation.
13133	Not a valid license for this product.

Error code	Description
25021	Database is not master or replica database. This operation only applies to master and replica databases.
25088	Catalog already in maintenance mode. You have set the mode on already.
25089	Not allowed to set maintenance mode off. Someone else has set the mode on so you can not set it off.
25090	Catalog is already in maintenance mode. Someone else has set the mode on so you can not set it on.
25091	Catalog is not in maintenance mode. You tried to set mode off and it is not currently on.

### B.75.11.4 SET SYNC NODE

```
SET SYNC NODE {unique_node_name | NONE}
```

#### Supported in

This command requires IBM solidDB advanced replication.

#### Usage

Assigning the node name is part of the registration process of a replica database. Each catalog of a IBM solidDB environment must have a node name that is unique within the domain. One catalog can have only one node name. Two catalogs cannot have the same node name.

You can use the SET SYNC NODE *unique\_node\_name* option to rename a node name if:

- If the node is a replica database and it is not registered to a master

and/or

- If the node is a master database and there are no replicas registered in the master database

Following are examples for renaming a node name:

```
SET SYNC NODE A; -- Now the node name is A.
SET SYNC NODE B; -- Now the node name is B.
COMMIT WORK;
SET SYNC NODE C; -- Now the node name is C.
```

```
ROLLBACK WORK; -- Now the node name is rolled back to B.  
SET SYNC NODE NONE; -- Now the node has no name.  
COMMIT WORK;
```

The *unique\_node\_name* must conform to the rules that are used for naming other objects (such as tables) in the database. Do not put single quotes around the node name.

If you specify NONE, then this command will remove the current node name.

If you want to use a reserved word, such as "NONE", as a node name, then you must put the keyword in double quote marks to ensure that it is treated as a delimited identifier. For example:

```
SET SYNC NODE "NONE"; -- Now the node name is "NONE"
```

You can verify the node name assignment with the following statement:

```
SELECT GET_PARAM( 'SYNC NODE' )
```

The SET SYNC NODE NONE option removes the node name from the current catalog. This option is used when you are dropping a synchronized database and removing its registration.



### Note

When using the SET SYNC NODE NONE option, be sure the catalog associated with the node name is not defined as a master, replica, or both. To remove the node name, the catalog must be defined as SET SYNC MASTER NO and/or SET SYNC REPLICA NO. If you do try to set the node name to NONE on a master and/or replica catalog, IBM solidDB returns error message 25082.

### Usage in Master

Use this statement in the master to set or remove the node name from the current catalog.

### Usage in Replica

Use this statement in the replica to set or remove the node name from the current catalog.

## Example

```
SET SYNC NODE SalesmanJones;
```

## Return Values

For details on each error code, see the appendix titled Error Codes in *IBM solidDB Administration Guide*.

**Table B.47. SET SYNC NODE Return Values**

Error code	Description
13047	No privilege for operation
13107	Illegal set operation
25059	After registration nodename cannot be changed
25082	Node name can not be removed if node is master or replica.

### B.75.11.5 SET SYNC PARAMETER

```
SET SYNC PARAMETER parameter_name 'value_as_string';  
SET SYNC PARAMETER parameter_name NONE;
```

## Supported in

This command requires IBM solidDB advanced replication.

## Usage

This statement defines persistent catalog-level parameters that are visible via the parameter bulletin board to all transactions that are executed in that catalog. Each catalog has a different set of parameters.

If the parameter already exists, the new value overwrites the previous one. An existing parameter can be deleted by setting its value to NONE. All parameters are stored in the SYS\_BULLETIN\_BOARD system table.

These parameters are not propagated to the master.

In addition to system specific-parameters, you can also store in the system table a number of system parameters that configure the synchronization functionality. Available system parameters are listed at the end of the SQL reference.

## Usage in Master

Use the SET SYNC PARAMETER in the master for setting database parameters.

## Usage in Replica

Use the SET SYNC PARAMETER in replicas for setting database parameters.

## Example

```
SET SYNC PARAMETER db_type 'REPLICA'  
SET SYNC PARAMETER db_type NONE
```

## Return Values

For details on each error code, see the appendix titled Error Codes in *IBM solidDB Administration Guide*.

**Table B.48. SET SYNC PARAMETER Return Values**

Error Code	Description
13086	Invalid data type in a parameter

## See Also

GET\_PARAM

PUT\_PARAM

### B.75.11.6 SET SYNC PROPERTY

*Syntax in Master:*

```
SET SYNC PROPERTY <propertyname> = { 'value' | NONE } FOR REPLICA  
<replicaname>
```

*Syntax in Replica:*

```
SAVE SET SYNC PROPERTY <propertyname> = {'value' | NONE }
```

### Supported in

This command requires IBM solidDB advanced replication.

### Usage

This command allows you to specify a property name and value for a replica. Replicas that have properties may be grouped, and a group may be specified when using the START AFTER COMMIT statement. For example, you might have some replicas that are related to the bicycle industry and others that are related to the surfboard industry, and you may want to update each of those groups of replicas separately. You can use Property Names to group these replicas. All members of a group have the same property and have the same value for that property.

For more information, see the section titled "Replica Property Names" in *IBM solidDB Advanced Replication Guide*.

### Examples

*Master:*

```
SET SYNC PROPERTY color = 'red' FOR REPLICA replical;  
SET SYNC PROPERTY color = NONE FOR REPLICA replical;
```

*Replica:*

```
SAVE SET SYNC PROPERTY color = 'red';  
SAVE SET SYNC PROPERTY color = NONE;
```

### B.75.11.7 SET SYNC USER

```
SET SYNC USER master_username IDENTIFIED BY password  
SET SYNC USER NONE
```

### Supported in

This command requires IBM solidDB advanced replication.



## Usage

This statement is used to define the username and password for the registration process when the replica database is being registered in the master database. To use this command, you are required to have `SYS_SYNC_ADMIN_ROLE` access.



### Note

The `SET SYNC USER` statement is used for replica registration only. Aside from registration, all other synchronization operations require a valid master user ID in a replica database. If you want to designate a different master user for a replica, you must map the replica ID on the replica database with the master ID on the master database. For details, read the section titled "Mapping Replica User ID With Master User ID" in *IBM solidDB Advanced Replication Guide*.

You define the registration username in the master database. The name you specify must have sufficient rights to execute the replica registration tasks. You can provide registration rights for a master user in the master database by designating the user with the `SYS_SYNC_REGISTER_ROLE` or the `SYS_SYNC_ADMIN_ROLE` using the `GRANT rolename TO user` statement.

After the registration has been successfully completed, you must reset the sync user to `NONE`; otherwise, if a master user saves statements, propagates messages, or refreshes from or registers to publications, the following error message is returned:

```
User definition not allowed for this operation.
```

## Usage in Master

This statement is not available in the master database.

## Usage in Replica

Use this statement in the replica to set the user name.

## Example

```
SET SYNC USER homer IDENTIFIED BY marge;  
SET SYNC USER NONE;
```

## B.75.12 SET TIMEOUT

```
SET IDLE TIMEOUT { timeout_in_seconds |  
                  timeout_in_millisecondsMS | DEFAULT }  
SET LOCK TIMEOUT { timeout_in_seconds |  
                  timeout_in_millisecondsMS }  
SET OPTIMISTIC LOCK TIMEOUT { timeout_in_seconds |  
                              timeout_in_millisecondsMS }
```

SET IDLE TIMEOUT sets the connection-specific maximum timeout in seconds. The setting is effective until a new timeout is given. If the timeout is set to DEFAULT, it means no maximum time.

SET LOCK TIMEOUT sets the time in seconds that the engine waits for a lock to be released. By default, lock timeout is set to 30 seconds. The maximum lock timeout is 1000 seconds. SET LOCK TIMEOUT of more than 1000 seconds fails.

By default, the granularity is in seconds. The lock timeout can be set at millisecond granularity by adding "MS" after the value, e.g.

```
SET LOCK TIMEOUT 500MS;  
SET LOCK TIMEOUT 1500 MS;
```

Spacing of the "MS" is not significant, and you may use upper or lower case. Without the "MS", the lock timeout will be in seconds. When the timeout interval is reached, IBM solidDB terminates the timed-out statement. For more information, see the section called "Setting Lock Timeout".

## B.75.13 SET TRANSACTION

### B.75.13.1 Usage

The settings apply only to the current transaction.

### B.75.13.2 Background Information on Transaction Logging and Durability

The server uses transaction logging to ensure that it can recover data in the event of an abnormal shutdown. "Strict" durability means that as soon as a transaction is committed, the server writes the information to the transaction log file. "Relaxed" durability means that the server may not write the information as soon as the transaction is committed; instead, the server may wait, for example, until it is less busy, or until it can write multiple transactions in a single write operation. If you use relaxed durability, then if the server shuts down abnormally, you may lose a few of the most recent transactions. For more information about durability, see *IBM solidDB In-Memory Database User Guide*.

If the SET TRANSACTION DURABILITY statement matches the level of durability already set for the session, the statement has no effect, and status "SUCCESS" is returned.

### B.75.13.3 Differences between SET and SET TRANSACTION

IBM solidDB SQL gives you two different commands to set the transaction isolation level, the read level, and the transaction durability level. In addition to the SET TRANSACTION command described in this section:

```
SET TRANSACTION { READ ONLY | READ WRITE | WRITE }
SET TRANSACTION ISOLATION LEVEL { READ COMMITTED ... }
SET TRANSACTION DURABILITY ... ;
```

there are also the SET commands described in Section B.75, "SET".

```
SET { READ ONLY | READ WRITE | WRITE }
SET ISOLATION LEVEL { READ COMMITTED ... }
SET DURABILITY ... ;
```

The commands that have the "TRANSACTION" keyword are called transaction-level commands, while the commands that do not have the "TRANSACTION" keyword are sometimes called session-level commands.

The transaction-level commands follow different rules from the session-level commands. These differences are listed below.

- The transaction-level commands take effect in the transaction in which they are issued; the session-level commands take effect in the next transaction, i.e. after the next COMMIT WORK.
- The transaction-level commands apply to only the current transaction; the session-level commands apply to all subsequent transactions — i.e. until the end of the session (connection) or until another SET command changes them.

- The transaction-level commands must be executed at the beginning of a transaction, i.e. before any DML or DDL statements. (They may be executed after other SET statements, however.) If this rule is violated, an error is returned. The session-level commands may be executed at any point in a transaction.
- The transaction-level commands take precedence over the session-level commands. However, the transaction-level commands apply only to the current transaction. After the current transaction is finished, the settings will return to the value set by the most recent previous SET command (if any). For example:

```
COMMIT WORK; -- Finish previous transaction;
SET ISOLATION LEVEL SERIALIZABLE;
COMMIT WORK;
-- Isolation level is now SERIALIZABLE
...
COMMIT WORK;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
-- Isolation level is now REPEATABLE READ because
-- transaction-level settings take precedence
-- over session-level settings.
COMMIT WORK;
-- Isolation level is now back to SERIALIZABLE, since the
-- transaction-level settings applied only to that
-- transaction.
```

The complete precedence hierarchy for isolation level and read level settings is below. Items closer to the top of the list have higher precedence.

1. SET TRANSACTION... (i.e. transaction-level settings)
2. SET ... (session-level settings)
3. The server-level settings specified by the value in `solid.ini` configuration parameter (e.g. `IsolationLevel` or `DurabilityLevel` (there is no `solid.ini` parameter for the READ ONLY / READ WRITE setting)). You may change these settings by editing the `solid.ini` file, or by issuing a command like the following:

```
ADMIN COMMAND 'parameter Logging.DurabilityLevel = 2';
```

Note that if you change the `solid.ini` parameter, the new setting will not take effect until the next time that the server starts.

4. The server's default (REPEATABLE READ, STRICT, or READ WRITE).

### B.75.13.4 Warnings Regarding Durability

- Unless you can afford to lose some transactions if the server is shut down unexpectedly, you should use strict durability.
- There is no "DEFAULT" option to set the value to whatever value the *DurabilityLevel* parameter has specified. Also, there is no way to read the durability level that applies to the current session. Therefore, once you have explicitly set the durability by executing the SET DURABILITY statement, you cannot restore the "default" durability level specified by the *DurabilityLevel* parameter. You can, of course, switch from RELAXED to STRICT durability and back whenever you wish, but you cannot "undo" your change and restore the default level without actually knowing what that default level was.



### Caution

The behavior of the SET TRANSACTION command changed in IBM solidDB Version 4.0. In previous IBM solidDB product versions, the SET TRANSACTION command applied to all subsequent transactions, rather than to the current transaction. If you want to keep the Section B.75, "SET" or you may set the `solid.ini` configuration parameter `SetTransCompatibility3` (see *IBM solidDB Administration Guide* for a description of this parameter).

The SET TRANSACTION command is based on ANSI SQL. However, the IBM solidDB implementation has some differences from the ANSI definition. The ANSI definition allows the two ANSI-defined "clauses" (isolation level and read level) to be combined, e.g.:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE, READ WRITE;
```

IBM solidDB does not support this syntax. IBM solidDB does, however, support multiple SET statements in a single transaction, e.g.:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;  
SET TRANSACTION READ WRITE;
```

### B.75.13.5 SET TRANSACTION Examples

```
SET TRANSACTION DURABILITY RELAXED;
```

```
SET TRANSACTION ISOLATION REPEATABLE READ;  
SET TRANSACTION READ WRITE;
```

### **B.75.13.6 See Also**

Section B.75, "SET".

Section 6.2.6, "TRANSACTION ISOLATION Levels".

"Logging and Transaction Durability" in *IBM solidDB Administration Guide*

### **B.75.13.7 SET TRANSACTION (Read/Write Level)**

```
SET TRANSACTION {READ ONLY | READ WRITE | WRITE}
```

The command `SET TRANSACTION { READ ONLY | READ WRITE | WRITE }` is based on ANSI SQL. It allows the user to specify whether the transaction is allowed to make any changes to data.

### **B.75.13.8 SET TRANSACTION DURABILITY**

```
SET TRANSACTION DURABILITY {RELAXED | STRICT}
```

The command `SET TRANSACTION DURABILITY { RELAXED | STRICT }` controls whether the server uses "strict" or "relaxed" durability for transaction logging. This command is a IBM solidDB extension to SQL; it is not part of the ANSI standard.

Your choice will not affect any other user, any other open session that you yourself currently have, or any future session that you may have. Each user session may set its own durability level, based on how important it is for the session not to lose any data.

Note that if the new transaction durability setting is `STRICT`, then any previous transactions that have not yet been written to disk will be written at the time that the current transaction is committed. (Note that those transactions are not written to disk as soon as the transaction durability level is changed to `STRICT`; the writes wait until the current transaction is committed.)

If the assigned workload server is `Secondary`, it can be changed programmatically to the `Primary` for the time of one transaction. At the transaction level, the following statements change the workload connection server to `Primary` for the time of one transaction:

```
SET TRANSACTION WRITE (nonstandard)
```

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
```

The affected transaction is the one that is started with the statement, or the next one, in other cases. After the transaction has been executed at the Primary, the workload connection server is reverted to the default one, for the session.

If the above statement is not applicable, it returns `SQL_SUCCESS`, with no action performed. For example, such is the case when `SET TRANSACTION WRITE` is applied to a standalone server. In that case the semantics of `SET TRANSACTION WRITE` is equal to that of `SET TRANSACTION READ WRITE`.

The effect of the `SET TRANSACTION WRITE` statement may be reverted with the statement `SET TRANSACTION READ WRITE` or `... READ ONLY (SQL:1999)`. Also, the isolation level statement has the same effect:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
```

### B.75.13.9 SET TRANSACTION ISOLATION LEVEL

```
SET TRANSACTION ISOLATION LEVEL {
    READ COMMITTED |
    REPEATABLE READ |
    SERIALIZABLE }
```

The command `SET TRANSACTION ISOLATION` is based on ANSI SQL. It sets the transaction isolation level (`READ COMMITTED`, `REPEATABLE READ`, or `SERIALIZABLE`) and the read level (`READ ONLY` or `READ WRITE`). For more information about isolation levels, see Section 6.2.6, “`TRANSACTION ISOLATION Levels`”.

### B.75.13.10 SET TRANSACTION SAFENESS

```
SET TRANSACTION SAFENESS {1SAFE | 2SAFE | DEFAULT}
```

`SET TRANSACTION SAFENESS` determines whether the replication protocol is synchronous (2-safe) or asynchronous (1-safe).

- 1-safe: the transaction is first committed at Primary and then transmitted to Secondary
- 2-safe: the transaction is not committed before it has been acknowledged by Secondary (default).

SET TRANSACTION SAFENESS sets the safeness level for the current transaction.

## B.76 START AFTER COMMIT

START AFTER COMMIT

```
[FOR EACH REPLICA WHERE search_condition [RETRY retry_spec]]  
{UNIQUE | NONUNIQUE} stmt;
```

*stmt* ::= any SQL statement.

*search\_condition* ::= *search\_item* | *search\_item* {AND|OR } *search\_item*

*search\_item* ::= {*search\_test* | (*search\_condition*)}

*search\_test* ::= *comparison\_test* | *like\_test*

*comparison\_test* ::= *property\_name* { = | >> | > | >= | > | >= } value

*property\_name* ::= name of a replica property

*like\_test* ::= *property\_name* [NOT] LIKE value [ESCAPE value]

value ::= *literal*

*retry\_spec* ::= *seconds,count*

### B.76.1 Usage

The START AFTER COMMIT statement specifies an SQL statement (such as a call to a stored procedure) that will be executed when the current transaction commits. (If the transaction is rolled back, then the specified SQL statement will not be executed.)

The START AFTER COMMIT statement returns a result set with one INTEGER column. This integer is a unique "job" id and can be used to query the status of a statement that failed to start due to an invalid SQL statement, insufficient access rights, replica not available etc.

If you use the UNIQUE keyword before the <stmt> then that the statement will be executed only if there isn't already an identical statement executing or "pending". Statements are compared using simple string compare. For example 'call foo(1)' is different from 'call foo(2)'. The server also takes into account whether the statement already being executed (or pending for execution) is on the same replica or a different replica; only identical statements on the same replica are discarded.

#### Important

Remember that when duplicate statements are discarded by using the UNIQUE keyword, the most recent statements are the ones thrown out, and the oldest one is the one that keeps running. It is quite possible to create a situation where you do multiple updates, for example, and you trigger multiple



START AFTER COMMIT operations, but only the oldest one executes and thus the newest updated data may not get sent to the replicas immediately.

NONUNIQUE means that duplicate statements can be executed simultaneously in the background.

FOR EACH REPLICA specifies that the statement is executed for each replica that fulfills the property conditions given in the search\_condition part of the WHERE clause. Before executing the statement, a connection to the replica is established. If a procedure call is started, then the procedure can get the "current" replica name using the keyword "DEFAULT".

If RETRY is specified, then the operation is re-executed after N seconds (defined by seconds in the retry\_spec) if the replica is not reached on the first attempt. The count specifies how many times a retry is attempted.

See Chapter 4, *Stored Procedures, Events, Triggers, and Sequences* for a more detailed description of the START AFTER COMMIT command.

## B.76.2 Transactions

A statement started in the background using START AFTER COMMIT is executed in a separate transaction. That transaction is executed in autocommit mode, i.e. it cannot be rolled back once it has started.

## B.76.3 Context of the Background Statements

Statements started in the background are executed in the context of the user who issued the START AFTER COMMIT statement, and are executed in the catalog and schema in which the START AFTER COMMIT statement executed.

In the example below, 'CALL FOO' is executed in the catalog 'katmandu' and the schema 'steinbeck'.

```
SET CATALOG katmandu;  
SET SCHEMA steinbeck;  
START AFTER COMMIT UNIQUE CALL FOO;  
COMMIT WORK;  
SET CATALOG irrelevant_catalog;  
SET SCHEMA irrelevant_schema
```

## B.76.4 Durability

Background statements are NOT durable. In other words, the execution of statements started with START AFTER COMMIT is not guaranteed.

## B.76.5 Rollback

Background statements cannot be rolled back after they have been started. So after a statement that has been started with `START AFTER COMMIT` has executed successfully, there is no way to roll it back.

The `START AFTER COMMIT` statement itself can of course be rolled back, and this will prevent the specified statement from executing. For example,

```
START AFTER COMMIT UNIQUE INSERT INTO MyTable VALUES (1);  
ROLLBACK;
```

In the example above, the transaction rolls back and thus `"INSERT INTO MyTable VALUES (1)"` will not be executed.

## B.76.6 Order of Execution

Background statements are executed asynchronously and they don't have any guaranteed order even inside a transaction.

## B.76.7 Examples

Start local procedure in the background.

```
START AFTER COMMIT NONUNIQUE CALL myproc;
```

Start the call if `"CALL myproc"` is not running in the background already.

```
START AFTER COMMIT UNIQUE call myproc;
```

Start procedure in the background using replicas which have property `"color" = "blue"`.

```
START AFTER COMMIT FOR EACH REPLICAS WHERE color='blue' UNIQUE CALL myproc;
```

The following statements are all considered different and therefore each is executed, despite the presence of the keyword `UNIQUE`. (Note that `"name"` is a unique property of each replica.)

```
START AFTER COMMIT UNIQUE call myproc;  
START AFTER COMMIT FOR EACH REPLICAS WHERE name='R1' UNIQUE call myproc;
```

```
START AFTER COMMIT FOR EACH REPLICA WHERE name='R2' UNIQUE call myproc;  
START AFTER COMMIT FOR EACH REPLICA WHERE name='R3' UNIQUE call myproc;
```

But if the following statement is executed in the same transaction as the previous ones and the condition "color='blue'" matches some of the replicas R1, R2 or R3, then the call is not executed for those replicas again.

```
START AFTER COMMIT FOR EACH REPLICA WHERE color='blue' UNIQUE call myproc;
```

For additional examples, see Chapter 4, *Stored Procedures, Events, Triggers, and Sequences*.

## B.77 TRUNCATE TABLE

```
TRUNCATE TABLE tablename
```

### B.77.1 Usage

This statement is, from the caller's point of view, semantically equivalent to "DELETE FROM *tablename*". However, it is much more efficient thanks to relaxed isolation. During the execution of this statement, the defined isolation level is not maintained in concurrent transactions. The effect of removing the rows will be immediately seen in all concurrent transactions. Therefore this statement is recommended for maintenance purposes only.

## B.78 UNLOCK TABLE

```
UNLOCK TABLE { ALL | tablename [, tablename] }  
tablename ::= The name of the table to unlock
```

The keyword ALL releases all table-level locks on all tables.

You can also specify the catalog and schema of the table by qualifying the table name.

### B.78.1 Usage

This command allows you to unlock tables that you manually locked (using the LOCK TABLE command) with the LONG option. The LONG option allows you to hold a lock past the end of the transaction in which the lock was placed. Since there is no natural endpoint for the lock (other than the end of the transaction), you must explicitly release a LONG lock by using the UNLOCK command.

The UNLOCK TABLE command does not apply to the server's automatic locks, or to manual locks that were not locked with the LONG option. If a lock is automatic, or if it is manual and not LONG, then the server will automatically release the lock at the end of the transaction in which the lock was placed. Thus there is no need to manually unlock those locks.

When the UNLOCK TABLE command is used, it does not take effect immediately; instead, the locks are released when the current transaction is committed.



### Caution

If the current transaction (the one in which the UNLOCK TABLE command was executed) is not committed (e.g. if it is rolled back), then the tables are not unlocked; they will remain locked until another UNLOCK TABLE command is successfully executed and committed.

The LOCK/UNLOCK commands apply only to tables. There is no command to manually lock or unlock individual records.

Note that if you have a table named "ALL", then you should use the delimited identifier feature to specify the table name. (See the examples at the end of this section.)

## B.78.2 Examples of Using LOCK and UNLOCK

```
LOCK TABLE emp IN SHARED MODE;
LOCK TABLE emp IN SHARED MODE TABLE dept IN EXCLUSIVE MODE;
LOCK TABLE emp,dept IN SHARED MODE NOWAIT;

-- Get an exclusive lock that will persist past the end of the current
-- transaction.  If you can't get an exclusive lock immediately, then
-- wait up to 60 seconds to get it.
LOCK TABLE emp, dept IN LONG EXCLUSIVE MODE WAIT 60;
-- Make the schema changes (or do whatever you needed the exclusive
-- lock for).
CALL DO_SCHEMA_CHANGES_1;
COMMIT WORK;
CALL DO_SCHEMA_CHANGES_2;
UNLOCK TABLE ALL; -- at the end of this transaction, release locks.
...
COMMIT WORK;
```

```
...
UNLOCK TABLE "ALL"; -- Unlock the table named "ALL".
```

## B.78.3 Return Values

For details on each error code, see the appendix titled Error Codes in *IBM solidDB Administration Guide*.

**Table B.49. LOCK TABLE Return Values**

Error code	Description
10083	Table <table_name> not locked.
13011	Table <tablename> not found.

## B.78.4 See Also

LOCK TABLE

SET SYNC MODE { MAINTENANCE | NORMAL }

## B.79 UNREGISTER EVENT

The UNREGISTER EVENT command is allowed only inside stored procedures. See the CREATE PROCEDURE statement and the CREATE EVENT statement for more details.

## B.80 UPDATE (Positioned)

```
UPDATE table_name
  SET [table_name.]column_identifier = {expression | NULL}
  [, [table_name.]column_identifier = {expression | NULL}]...
  WHERE CURRENT OF cursor_name
```

### B.80.1 Usage

The positioned UPDATE statement updates the current row of the cursor. The name of the cursor is defined using ODBC API function named SQLSetCursorName.

## B.80.2 Example

```
UPDATE TEST SET C = 0.33
WHERE CURRENT OF MYCURSOR
```

## B.81 UPDATE (Searched)

```
UPDATE table-name
  SET [table_name.]column_identifier = {expression | NULL}
  [, [table_name.]column_identifier = {expression | NULL}]...
  [WHERE search_condition]
```

### B.81.1 Usage

The UPDATE statement is used to modify the values of one or more columns in one or more rows, according to the search conditions.

### B.81.2 Example

```
UPDATE TEST SET C = 0.44
WHERE ID = 5
```

## B.82 WAIT EVENT

The WAIT EVENT command is allowed only inside stored procedures. See the CREATE PROCEDURE and CREATE EVENT statements for more details.

## B.83 Table\_reference

**Table B.50. Table\_reference**

Table_reference	
<i>table_reference_list</i>	::= <i>table_reference</i> [ , <i>table-reference</i> ... ]

<b>Table_reference</b>	
<i>table_reference</i>	::= <i>table_name</i> [[AS] <i>correlation_name</i> ]   <i>derived_table</i> [[AS] <i>correlation_name</i> [( <i>derived_column_list</i> )]]   <i>joined_table</i>
<i>table_name</i>	::= <i>table_identifier</i>   <i>schema_name.table_identifier</i>
<i>derived_table</i>	::= <i>subquery</i>
<i>de- rived_column_list</i>	::= <i>column_name_list</i>
<i>joined_table</i>	::= <i>cross_join</i>   <i>qualified_join</i>   ( <i>joined_table</i> )
<i>cross_join</i>	::= <i>table_reference</i> CROSS JOIN <i>table_reference</i>
<i>qualified_join</i>	::= <i>table_reference</i> [NATURAL] [ <i>join_type</i> ] JOIN <i>table_reference</i> [ <i>join_specification</i> ]
<i>join_type</i>	::= INNER   <i>outer_join_type</i> [OUTER]   UNION
<i>outer_join_type</i>	::= LEFT   RIGHT   FULL

<b>Table_reference</b>	
<i>join_specification</i>	::= <i>join_condition</i>   <i>named_columns_join</i>
<i>join_condition</i>	::= ON <i>search_condition</i>
<i>named_columns_join</i>	::= USING ( <i>column_name_list</i> )
<i>column_name_list</i>	::= <i>column_identifier</i> [ { , <i>column_identifier</i> } ... ]

## B.84 Query\_specification

**Table B.51. Query\_specification**

<b>Query_specification</b>	
<i>query_specification</i>	::= SELECT [DISTINCT   ALL] <i>select_list</i> <i>table_expression</i>
<i>select_list</i>	::= *   <i>select_sublist</i> [ { , <i>select_sublist</i> } ... ]
<i>select_sublist</i>	::= <i>derived_column</i>   [ <i>table_name</i>   <i>table_identifier</i> ].*



<b>Query_specification</b>	
<i>derived_column</i>	<code>::= expression [ [AS] column_alias ] ]</code>
<i>table_expression</i>	<code>::= FROM table_reference_list [WHERE search_condition ] [GROUP BY column_name_list [[UNION   INTERSECT   EXCEPT] [ALL] [CORRESPONDING [BY ( column_name_list ) ]] query_specification ] [HAVING search_condition ]</code>

## B.85 Search\_condition

**Table B.52. Search\_condition**

<b>Search_condition</b>	
<i>search_condition</i>	<code>::= search_item   search_item { AND   OR } search_item</code>
<i>search_item</i>	<code>::= [NOT] { search_test   ( search_condition )}</code>
<i>search_test</i>	<code>::= comparison_test   between_test   like_test   null_test   set_test   quantified_test   existence_test</code>

<b>Search_condition</b>	
<i>comparison_test</i>	<pre>::= expression { =   &lt;&gt;   &lt;   &lt;=   &gt;   &gt;= } { expression / subquery }</pre> <p>Note: Spaces on each side of the operator are optional.</p>
<i>between_test</i>	<pre>::= column_identifier [NOT] BETWEEN expression AND expression</pre>
<i>like_test</i>	<pre>::= column_identifier [NOT] LIKE value [ESCAPE value]</pre>
<i>null_test</i>	<pre>::= column_identifier IS [NOT] NULL</pre>
<i>set_test</i>	<pre>::= expression [NOT] IN ( { value [, value ]...   subquery } )</pre>
<i>quantified_test</i>	<pre>::= expression { =   &lt;&gt;   &lt;   &lt;=   &gt;   &gt;= } [ALL   ANY   SOME] subquery</pre>
<i>existence_test</i>	<pre>::= EXISTS subquery</pre>

## B.86 Check\_condition

**Table B.53. Check\_condition**

<b>Check_condition</b>	
<i>check_condition</i>	<code>::= check_item   check_item { AND   OR } check_item</code>
<i>check_item</i>	<code>::= [NOT] { check_test   ( check_condition ) }</code>
<i>check_test</i>	<code>::= comparison_test   between_test   like_test   null_test   list_test</code>
<i>comparison_test</i>	<code>::= expression { =   &lt;&gt;   &lt;   &lt;=   &gt;   &gt;= } { expression   subquery }</code>
<i>between_test</i>	<code>::= column_identifier [NOT] BETWEEN expression AND expression</code>
<i>like_test</i>	<code>::= column_identifier [NOT] LIKE value [ESCAPE value ]</code>
<i>null_test</i>	<code>::= column_identifier IS [NOT] NULL</code>

Check_condition	
<i>list_test</i>	<code>::= expression [NOT] IN ( { value [, value ]...} )</code>

## B.87 Expression

**Table B.54. Expression**

Expression	
<i>expression</i>	<code>::= expression_item   expression_item { +   -   *   / } expression_item</code>  Note: Spaces on each side of the operator are optional.
<i>expression_item</i>	<code>::= [ +   - ] { value   column_identifier   function   case_expression   cast_expression   ( expression ) }</code>
<i>value</i>	<code>::= literal   USER   variable</code>
<i>function</i>	<code>::= set_function   null_function   string_function   numeric_function   datetime_function   system_function   datatypeconversion_function</code>  NOTE: The string, numeric, datetime, and datatypeconversion functions are scalar functions, in which an operation denoted by a function name is followed by a pair of parenthesis enclosing zero or more specified arguments. Each scalar function returns a single value.

<b>Expression</b>	
<i>set_function</i>	<pre> ::= COUNT (*)    { AVG   MAX   MIN   SUM   COUNT }  ( { ALL   DISTINCT } expression )</pre>
<i>null_function</i>	<pre> ::= { NULLVAL_CHAR( )   NULLVAL_INT( ) }</pre>
<i>datatypeconversion_function</i>	<pre> ::= CONVERT_CHAR(value_exp)    CONVERT_DATE(value_exp)    CONVERT_DECIMAL(value_exp)    CONVERT_DOUBLE(value_exp)    CONVERT_FLOAT(value_exp)    CONVERT_INTEGER(value_exp)    CONVERT_LONGVARCHAR(value_exp)    CONVERT_NUMERIC(value_exp)    CONVERT_REAL(value_exp)    CONVERT_SMALLINT(value_exp)    CONVERT_TIME(value_exp)    CONVERT_TIMESTAMP(value_exp)    CONVERT_TINYINT(value_exp)    CONVERT_VARCHAR(value_exp)</pre> <p>Note: These functions are used to implement the {fn CONVERT(<i>value</i>, <i>odbc_typename</i>)} escape clauses defined by ODBC. The preferred way, however, is to use CAST(<i>value</i> AS <i>sql_typename</i>) which is defined in SQL-92 and fully supported by IBM solidDB. For details, see Appendix F of <i>IBM solidDB Programmer Guide</i>.</p>
<i>case_expression</i>	<pre> ::= case_abbreviation   case_specification</pre>
<i>case_abbreviation</i>	<pre> ::= NULLIF(value_exp, value_exp)  </pre>

Expression	
	<p><code>COALESCE(value_exp {, value_exp }...)</code></p> <p>The NULLIF function returns NULL if the first parameter is equal to the second parameter; otherwise, it returns the first parameter. It is equivalent to <code>IF (p1 = p2) THEN RETURN NULL ELSE RETURN p1</code>; The NULLIF function is useful if you have a special value that serves as a flag to indicate NULL. You can use NULLIF to convert that special value to NULL. In other words, it behaves like <code>IF (p1 = NullFlag) THEN RETURN NULL ELSE RETURN p1</code>;</p> <p>COALESCE returns the first non-NULL argument. The list of arguments may be of almost any length. All arguments should be of the same (or compatible) data types.</p>
<i>case_specification</i>	<pre> ::= CASE [value_exp] WHEN value_exp THEN {value_exp } [WHEN value_exp THEN { value_exp } ...] [ELSE { value_exp }] END </pre>
<i>cast_expression</i>	<pre> ::= CAST (value_exp AS -data-type) </pre>
<i>row value constructor expression</i>	<p>A row value constructor (RVC) is an ordered sequence of values delimited by parentheses, for example:</p> <p>(1, 4, 9)</p> <p>('Smith', 'Lisa')</p> <p>You can think of this as constructing a row based on a series of elements/values, just like a row of a table is composed of a series of fields.</p>

<b>Expression</b>	
	For more information about row value constructors, see Section 3.5.5, “Row Value Constructors”.

## B.88 String Functions

**Table B.55. String Functions**

Function	Purpose
ASCII( <i>str</i> )	Returns the integer equivalent of string <i>str</i>
CHAR( <i>code</i> )	Returns the character equivalent of <i>code</i>
CONCAT( <i>str1</i> , <i>str2</i> )	Concatenates <i>str2</i> to <i>str1</i>
<i>str1</i> { +    } <i>str2</i>	Concatenates <i>str2</i> to <i>str1</i> .  For example:  SELECT <i>str1</i> + <i>str2</i> , col1 ...  SELECT <i>str1</i>    <i>str2</i> , col1 ...
GET_UNIQUE_STRING( <i>str</i> )	This function generates a unique string, based on a "prefix" (the input string, which may be any string you choose) and a sequence number (which is created and used internally). If the input is NULL, then the function still returns a string based on the unique sequence number.
INSERT( <i>str1</i> , <i>start</i> , <i>length</i> , <i>str2</i> )	Merges strings by deleting <i>length</i> characters from <i>str1</i> and inserting <i>str2</i>
LCASE( <i>str</i> )	Converts string <i>str</i> to lowercase
LEFT( <i>str</i> , <i>count</i> )	Returns leftmost <i>count</i> characters of string <i>str</i>
LENGTH( <i>str</i> )	Returns the number of characters in <i>str</i>
LOCATE( <i>str1</i> , <i>str2</i> [, <i>start</i> ])	Returns the starting position of <i>str1</i> within <i>str2</i> . If the optional argument, <i>start</i> , is specified, the search begins with the character position indicated by the value of <i>start</i> . If <i>str1</i> is not found within <i>str2</i> , the function returns 0. For both the return value and the input parameter <i>start</i> , string positions are numbered starting from 1 (not 0).

Function	Purpose
LTRIM( <i>str</i> )	Removes leading spaces of <i>str</i>
POSITION( <i>str1</i> IN <i>str2</i> )	Returns starting position of <i>str1</i> within <i>str2</i>
REPEAT( <i>str</i> , <i>count</i> )	Returns characters of <i>str</i> repeated <i>count</i> times
REPLACE( <i>str1</i> , <i>str2</i> , <i>str3</i> )	Replaces occurrences of <i>str2</i> in <i>str1</i> with <i>str3</i>
RIGHT( <i>str</i> , <i>count</i> )	Returns the rightmost count characters of string <i>str</i>
RTRIM( <i>str</i> )	Removes trailing spaces in <i>str</i>
SOUNDEX( <i>str</i> )	Calculate 4-character soundex (phonetic) code
SPACE( <i>count</i> )	Returns a string of count spaces
SUBSTRING( <i>str</i> , <i>start</i> , <i>length</i> )	Derives substring <i>length</i> bytes long from <i>str</i> beginning at <i>start</i> . For example, if <i>str</i> ="First Second Third", then SUBSTRING( <i>str</i> , 7, 6) would return "Second".  Note that string positions are numbered starting from 1 (not 0).
TRIM( <i>str</i> )	Removes leading and trailing spaces in <i>str</i>
UCASE( <i>str</i> )	Converts <i>str</i> to uppercase

If you are using wildcard characters in your string operations, then see also Section B.96, "Wildcard Characters".

## B.89 Numeric Functions

**Table B.56. Numeric Functions**

Function	Purpose
ABS( <i>numeric</i> )	Absolute value of <i>numeric</i>
ACOS( <i>float</i> )	Arccosine of <i>float</i> , where <i>float</i> is expressed in radians
ASIN( <i>float</i> )	Arcsine of <i>float</i> , where <i>float</i> is expressed in radians
ATAN( <i>float</i> )	Arctangent of <i>float</i> , where <i>float</i> is expressed in radians



Function	Purpose
ATAN2( <i>float1</i> , <i>float2</i> )	Arctangent of the <i>x</i> and <i>y</i> coordinates, specified by <i>float1</i> and <i>float2</i> , respectively, as an angle, expressed in radians
CEILING( <i>numeric</i> )	Smallest integer greater than or equal to <i>numeric</i>
COS( <i>float</i> )	Cosine of <i>float</i> , where <i>float</i> is expressed in radians
COT( <i>float</i> )	Cotangent of <i>float</i> , where <i>float</i> is expressed in radians
DEGREES( <i>numeric</i> )	Converts <i>numeric</i> radians to degrees
DIFFERENCE( <i>str1</i> , <i>str2</i> )	Return the value of phonetic difference: 0 - 4
EXP( <i>float</i> )	Exponential value of <i>float</i>
FLOOR( <i>numeric</i> )	Largest integer less than or equal to <i>numeric</i>
LOG( <i>float</i> )	Natural logarithm of <i>float</i>
LOG10( <i>float</i> )	Base 10 log of <i>float</i>
MOD( <i>integer1</i> , <i>integer2</i> )	Modulus of <i>integer1</i> divided by <i>integer2</i>
PI()	Pi as a floating point number
POWER( <i>numeric</i> , <i>integer</i> )	Value of <i>numeric</i> raised to the power of <i>integer</i>
RADIANS( <i>numeric</i> )	Converts from <i>numeric</i> degrees to radians
ROUND( <i>numeric</i> , <i>integer</i> )	<i>Numeric</i> rounded to <i>integer</i>
SIGN( <i>numeric</i> )	Sign of <i>numeric</i>
SIN( <i>float</i> )	Sine of <i>float</i> , where <i>float</i> is expressed in radians
SQRT( <i>float</i> )	Square root of <i>float</i>
TAN( <i>float</i> )	Tangent of <i>float</i> , where <i>float</i> is expressed in radians
TRUNCATE( <i>numeric</i> , <i>integer</i> )	<i>Numeric</i> truncated to <i>integer</i>

## B.90 Date Time Functions

**Table B.57. Date Time Functions**

Function	Purpose
CURDATE()	Returns the current date

## B.90 Date Time Functions

Function	Purpose
CURTIME()	Returns the current time
DAYNAME( <i>date</i> )	Returns a string with the day of the week
DAYOFMONTH( <i>date</i> )	Returns the day of the month as an integer between 1 and 31
DAYOFWEEK( <i>date</i> )	Returns the day of the week as an integer between 1 and 7, where 1 represents Sunday
DAYOFYEAR( <i>date</i> )	Returns the day of the year as an integer between 1 and 366
EXTRACT ( <i>date field</i> FROM <i>date_exp</i> )	Isolates a single field of a datetime or a interval and converts it to a number.
HOUR( <i>time_exp</i> )	Returns the hour as an integer between 0 and 23
MINUTE( <i>time_exp</i> )	Returns the minute as an integer between 0 and 59
MONTH( <i>date</i> )	Returns the month as an integer between 1 and 12
MONTHNAME( <i>date</i> )	Returns the month name as a string
NOW()	Returns the current date and time as a timestamp
QUARTER( <i>date</i> )	Returns the quarter as an integer between 1 and 4
SECOND( <i>time_exp</i> )	Returns the second as an integer between 0 and 59
TIMESTAMPADD( <i>interval</i> , <i>integer_exp</i> , <i>timestamp_exp</i> )	<p>Calculates a timestamp by adding <i>integer_exp</i> intervals of type <i>interval</i> to <i>timestamp_exp</i></p> <p>Keywords used to express valid TIMESTAMPADD interval values are:</p> <p>SQL_TSI_FRAC_SECOND</p> <p>SQL_TSI_SECOND</p> <p>SQL_TSI_MINUTE</p> <p>SQL_TSI_HOUR</p> <p>SQL_TSI_DAY</p> <p>SQL_TSI_WEEK</p> <p>SQL_TSI_MONTH</p>

Function	Purpose
	SQL_TSI_QUARTER  SQL_TSI_YEAR
TIMESTAMPDIFF(interval, <i>timestamp-exp1</i> , <i>timestamp-exp2</i> )	Returns the integer number of intervals by which <i>timestamp-exp2</i> is greater than <i>timestamp-exp1</i>  Keywords used to express valid TIMESTAMPDIFF interval values are:  SQL_TSI_FRAC_SECOND  SQL_TSI_SECOND  SQL_TSI_MINUTE  SQL_TSI_HOUR  SQL_TSI_DAY  SQL_TSI_WEEK  SQL_TSI_MONTH  SQL_TSI_QUARTER  SQL_TSI_YEAR
WEEK( <i>date</i> )	Returns the week of the year as an integer between 1 and 52
YEAR( <i>date</i> )	Returns the year as an integer

## B.91 System Functions

The system functions return special information about the IBM solidDB database.

**Table B.58. System Functions**

Function	Purpose
UIC()	Returns the connection id associated with the connection

Function	Purpose
CURRENT_USERID ()	Returns the current user id
LOGIN_USERID ()	Returns the login userid
CURRENT_CATALOG ()	Returns the current catalog
LOGIN_CATALOG ()	Returns the login catalog
CURRENT_SCHEMA ()	Returns the current schema
LOGIN_SCHEMA ()	Returns the login schema

## B.92 Miscellaneous Functions

**Table B.59. Miscellaneous Functions**

Function	Purpose
BIT_AND( <i>integer1</i> , <i>integer2</i> )	Returns the result of the bit-wise AND operation.
IFNULL( <i>exp</i> , <i>value</i> )	If <i>exp</i> is null, returns <i>value</i> ; if not, returns <i>exp</i> (if <i>value</i> is returned, it is converted to type of <i>exp</i> )
SLEEP( <i>milliseconds</i> )	This can only be called from a stored procedure or a trigger. This causes the stored procedure or trigger to "sleep" (temporarily suspend activity) for the specified number of milliseconds. Resolution is accurate to approximately 1 second (i.e. 1000 milliseconds). The exact length of the sleep also depends upon how busy the computer is with other processes and threads. The value must be a literal, not a variable or expression.

## B.93 Data\_type

**Table B.60. Data\_type**

Data_type	
data_type	::= {BIGINT   BINARY   BLOB   CHAR [ <i>length</i> ]

Data_type	
	CHARACTER LARGE OBJECT   CHAR LARGE OBJECT   CLOB   DATE   DECIMAL [ ( <i>precision</i> [ , <i>scale</i> ] ) ]   DOUBLE PRECISION   FLOAT [ ( <i>precision</i> ) ]   INTEGER   LONG NATIONAL VARCHAR   LONG VARBINARY   LONG VARCHAR   LONG WVARCHAR   NCHAR LARGE OBJECT   NUMERIC [ ( <i>precision</i> [ , <i>scale</i> ] ) ]   NATIONAL CHAR   NATIONAL CHARACTER   NATIONAL VARCHAR   NCHAR   NCHAR VARYING   NCLOB   NVARCHAR   REAL   SMALLINT   TIME   TIMESTAMP [ ( <i>timestamp precision</i> ) ]   TINYINT   VARBINARY   VARCHAR [ ( <i>length</i> ) ] }   WCHAR   WVARCHAR [ <i>length</i> ]

## B.94 Date and Time Literals

**Table B.61. Date and Time Literals**

Date/time literal	
<i>date_literal</i>	'YYYY-MM-DD'

<b>Date/time literal</b>	
<i>time_literal</i>	'HH:MM:SS'
<i>timestamp_literal</i>	'YYYY-MM-DD HH:MM:SS'

## B.95 Pseudo Columns

The following pseudo columns may also be used in the select-list of a SELECT statement:

**Table B.62. Pseudo Columns**

Pseudo column	Type	Explanation
ROWVER	VARBINARY(10)	Version of the row in a table.
ROWID	VARBINARY(254)	Persistent id for a row in a table.
ROWNUM	DECIMAL(16,2)	<p>Row number indicates the sequence in which a row was selected from a table or set of joined rows. The first row selected has a ROWNUM of 1, the second row has 2, etc.</p> <p>Because ROWNUM is given to a row before the order by clause is evaluated, ROWNUM should not be used to identify sorted rows.</p> <p>ROWNUM is chiefly useful for limiting the number of rows returned by a query for example, WHERE ROWNUM &lt; 10).</p>



### Note

Since ROWID and ROWVER refer to a single row, they may only be used with queries that return rows from a single table.

## B.96 Wildcard Characters

The following may be used as wildcard characters in certain expressions, such as LIKE '<string>':

**Table B.63. Wildcard Characters**

Character	Explanation
_ (underscore)	The underscore character matches any single character. For example, 'J_NE' matches 'JANE' and 'JUNE'.
% (percent sign)	The percent sign character matches any group of 0 or more characters. For example 'ED%' matches 'EDWARD' and 'EDITOR'. As another example, '%ED%' matches 'EDWARD', 'TEDDY', and 'FRED'.

## B.96.1 Using SQL Wildcards

Exact match searches are conducted by specifying literal values, as in:

```
SELECT * FROM table1 WHERE name = 'SMITH';
```

The string 'SMITH' is a literal value.

Similar match searches are conducted by specifying a SQL wildcard that represents a character string that is similar to another character string. Logical expressions (such as those used in **WHERE** clauses and **CHECK** constraints) may use the "wildcard" characters and the keyword **LIKE** to match strings that are similar.

The underscore character ( **\_** ) is a wildcard character that matches any single character. For example, the following query:

```
SELECT * FROM table1 WHERE first_name LIKE 'J_NE';
```

returns both JANE and JUNE (as well as any other four-character name where the first letter is J and the last two letters are NE).

The percent character ( **%** ) is a wildcard character that matches any occurrence of 0 or more characters. For example, the following query:

```
SELECT * FROM table1 WHERE first_name LIKE 'JOHN%';
```

could return JOHN, JOHNNY, JOHNATHAN, etc.

The **%** wildcard is used most often at the end of strings, but it can be used anywhere. For example, the following search pattern:

```
LIKE '%JO%'
```

returns all people who have JO somewhere in their name, included but not limited to:

JOANNE, BILLY JO, and LONG JOHN SILVER

Multiple wildcards are allowed in a single string. For example, the string `J_V_` matches `JAVA` and `JIVE` and any other four-character words or names that start with `J` and have `V` as the third character. Note that because the underscore (`_`) only matches exactly one character, the string `J_V_` does not match the string `JOVIAL`, which has more than four characters.

### B.96.2 Wildcard Characters as Literals

A wildcard character may be used in one part of a string while the literal character `%` (percent) or underscore (`_`) may be used in another part of the same string. To use a wildcard character as a literal, the wildcard character is prefaced with an escape character; the escape character itself must be specified as part of the query. For example, the expression below uses the backslash character (`\`) as the escape character:

```
LIKE 'MY\_EXPRESSION_' ESCAPE '\\';
```

matches the following:

```
MY_EXPRESSION1 MY_EXPRESSIONA MY_EXPRESSION_
```

but not:

```
MY#EXPRESSION1
```

ANSI standard SQL specifies that character strings must be delimited by single quotes. For example:

```
...LIKE 'J_N_'; -- CORRECT
...LIKE "J_N_"; --WRONG
```

Double quotes are used for delimited identifiers, not data. (C and Java programmers may find this confusing because the C language uses double quotes to delimit strings as in "*C-language string*" and single quotes `'C'` to delimit single characters.



---

# Appendix C. Reserved Words

This appendix contains reserved words in several SQL standards: ODBC 3.0, X/Open and SQL Access Group SQL CAE specification, Database Language - SQL: ANSI X3H2 (SQL-92). Some words are used by IBM solidDB SQL. Applications should avoid using any of these keywords for other purposes. The following table contains also potential reserved words; these markings are enclosed in parenthesis.

Some of the reserved words in this appendix can be used as identifiers (such as table name, column name, etc.) by surrounding the word in double quotes (""). Identifiers in double quote marks are known as delimited identifiers and conform to the ANSI standard for SQL. In the following SQL statement example, the reserved word "NULL" is used as a table name identifier:

```
CREATE TABLE "NULL" (column_1 INTEGER);
```



## Note

IBM solidDB SQL allows some reserved words to be used as identifiers even if those words are not in double quotes. However, we strongly recommend that you use double quotes around any reserved word that you want to use as an identifier; this will increase portability.

**Table C.1. Reserved Words List**

Reserved word	ODBC	X/Open SQL	ANSI SQL-92	Solid SQL
ABSOLUTE	•		•	
ACTION	•		•	
ADA	•			
ADD	•	•	•	•
ADMIN				•
AFTER			(•)	•
ALIAS			(•)	
ALL	•	•	•	•
ALLOCATE	•	•	•	
ALTER	•	•	•	•
AND	•	•	•	•
ANY	•	•	•	•

Reserved word	ODBC	X/Open SQL	ANSI SQL-92	Solid SQL
APPEND				•
ARE	•		•	
AS	•	•	•	•
ASC	•	•	•	•
ASSERTION	•		•	
ASYNC			(•)	•
AT	•		•	
AUTHORIZATION	•		•	•
AVG	•	•	•	
BEFORE			(•)	•
BEGIN	•	•	•	•
BETWEEN	•	•	•	•
BINARY				•
BIT	•		•	
BIT_LENGTH	•		•	
BOOKMARK				•
BOOLEAN			(•)	
BOTH	•		•	
BREADTH			(•)	
BY	•	•	•	•
CALL			(•)	•
CASCADE	•	•	•	•
CASCADED	•		•	•
CASE	•		•	•
CAST	•		•	•
CATALOG	•		•	•
CHAR	•	•	•	•
CHAR_LENGTH	•		•	
CHARACTER	•	•	•	•

Reserved word	ODBC	X/Open SQL	ANSI SQL-92	Solid SQL
CHARACTER_LENGTH	•		•	
CHECK	•	•	•	•
CLOSE	•	•	•	•
COALESCE				
COLLATE	•		•	
COLLATION	•		•	
COLUMN	•		•	•
COMMIT	•	•	•	•
COMMITBLOCK				•
COMMITTED				•
COMPLETION			(•)	
CONNECT	•	•	•	•
CONNECTION	•	•	•	
CONSTRAINT	•		•	•
CONSTRAINTS	•		•	
CONTINUE	•	•	•	
CONVERT	•		•	
CORRESPONDING	•		•	•
COUNT	•	•	•	
CREATE	•	•	•	•
CROSS	•		•	•
CURRENT	•	•		•
CURRENT_DATE	•		•	
CURRENT_TIME	•		•	
CURRENT_TIMESTAMP	•		•	
CURRENT_USER	•		•	
CURSOR	•	•	•	•
CYCLE			(•)	

<b>Reserved word</b>	<b>ODBC</b>	<b>X/Open SQL</b>	<b>ANSI SQL-92</b>	<b>Solid SQL</b>
DATA			(*)	•
DATE				
DAY				
DEALLOCATE				
DEC	•	•	•	•
DECIMAL	•	•	•	•
DECLARE	•	•	•	•
DEFAULT	•	•	•	•
DEFERRABLE	•		•	
DEFERRED	•		•	
DELETE	•	•	•	•
DENSE				•
DEPTH			(*)	
DESC	•	•	•	•
DESCRIBE	•	•	•	
DESCRIPTOR	•	•	•	
DIAGNOSTICS	•	•	•	
DICTIONARY			(*)	
DISCONNECT	•	•	•	
DISTINCT	•	•	•	•
DOMAIN	•		•	•
DOUBLE	•	•	•	•
DROP	•	•	•	•
EACH			(*)	
ELSE	•		•	•
ELSEIF			(*)	•
ENABLE				•
END	•	•	•	•
END-EXEC	•		•	

<b>Reserved word</b>	<b>ODBC</b>	<b>X/Open SQL</b>	<b>ANSI SQL-92</b>	<b>Solid SQL</b>
EQUALS			(•)	
ESCAPE	•		•	•
EVENT				•
EXCEPT	•		•	•
EXCEPTION				
EXEC	•	•	•	•
EXECUTE	•	•	•	•
EXISTS	•	•	•	•
EXPLAIN				•
EXPORT				•
EXTERNAL	•		•	•
EXTRACT	•		•	•
FALSE	•		•	
FETCH	•	•	•	•
FIRST	•		•	
FIXED				•
FLOAT	•	•	•	•
FOR	•	•	•	•
FOREIGN	•	•	•	•
FOREVER				•
FORTRAN	•			
FORWARD				•
FOUND	•	•	•	
FROM	•	•	•	•
FROMFIXED				•
FULL	•		•	•
GENERAL			(•)	
GET	•	•	•	•
GLOBAL	•		•	

Reserved word	ODBC	X/Open SQL	ANSI SQL-92	Solid SQL
GO	•		•	
GOTO	•	•	•	
GRANT	•	•	•	•
GROUP	•	•	•	•
HAVING	•	•	•	•
HINT				•
HOUR	•		•	
IDENTIFIED				•
IDENTITY	•		•	
IF			(•)	•
IGNORE	•		(•)	
IMMEDIATE	•	•	•	
IMPORT				•
IN	•	•	•	•
INCLUDE	•	•		
INDEX	•	•		•
INDICATOR	•		•	
INITIALLY	•		•	
INNER	•		•	•
INPUT	•		•	
INSENSITIVE	•		•	
INSERT	•	•	•	•
INT	•	•	•	•
INTEGER	•	•	•	•
INTERNAL				•
INTERSECT	•		•	•
INTERVAL	•		•	
INTO	•	•	•	•
IS	•	•	•	•

Reserved word	ODBC	X/Open SQL	ANSI SQL-92	Solid SQL
ISOLATION	•		•	•
JAVA				•
JOIN	•		•	•
KEY	•	•	•	•
LANGUAGE	•		•	
LAST	•		•	
LEADING	•		•	
LEAVE			(•)	•
LEFT	•		•	•
LESS			(•)	
LEVEL	•		•	•
LIKE	•	•	•	•
LIMIT			(•)	
LOCAL	•		•	•
LOCK				•
LONG				•
LOOP			(•)	•
LOWER	•		•	
MAINMEMORY				•
MASTER				•
MATCH	•		•	
MAX	•	•	•	
MERGE				•
MESSAGE				•
MIN	•	•	•	
MINUTE	•		•	
MODIFY			(•)	•
MODULE	•		•	
MONTH	•		•	

Reserved word	ODBC	X/Open SQL	ANSI SQL-92	Solid SQL
NAMES	•		•	
NATIONAL	•		•	
NATURAL	•		•	•
NCHAR	•		•	
NEW			(•)	•
NEXT	•		•	•
NO	•		•	•
NONE	•		(•)	
NOT	•	•	•	•
NULL	•	•	•	•
NULLIF	•		•	•
NUMERIC	•	•	•	•
OBJECT			(•)	
OCTET_LENGTH	•		•	
OF	•	•	•	•
OFF				
OID			(•)	
OLD			(•)	•
ON	•	•	•	•
ONLY	•		•	•
OPEN	•	•	•	
OPERATION			(•)	
OPERATORS			(•)	
OPTIMISTIC				•
OPTION				
OR				•
ORDER				
OTHERS				
OUTER				•



Reserved word	ODBC	X/Open SQL	ANSI SQL-92	Solid SQL
OUTPUT	•		•	
OVERLAPS	•		•	
PARAMETERS			(•)	
PARTIAL	•		•	
PASCAL	•			
PENDANT			(•)	
PESSIMISTIC				•
PLAN				•
PLI	•			
POSITION	•		•	
POST				•
PRECISION	•	•	•	•
PREORDER			(•)	
PREPARE				
PRESERVE				
PRIMARY	•	•	•	•
PRIOR	•		•	
PRIVATE			(•)	
PRIVILEGES	•		•	•
PROCEDURE	•		•	•
PROPAGATE				•
PROTECTED			(•)	
PUBLIC	•	•	•	•
PUBLICATION				•
READ			•	•
REAL		•	•	•
RECURSIVE			(•)	
REF			(•)	
REFERENCES	•	•	•	•

Reserved word	ODBC	X/Open SQL	ANSI SQL-92	Solid SQL
REFERENCING			(•)	•
REFRESH				•
REGISTER				•
RELATIVE	•		•	
RENAME				•
REPEATABLE				•
REPLACE			(•)	
REPLICA				•
REPLY				•
RESIGNAL			(•)	
RESTART				•
RESTRICT	•	•	•	•
RESULT				•
RETURN			(•)	•
RETURNS			(•)	•
REVERSE				•
REVOKE	•	•	•	•
RIGHT	•		•	•
ROLE			(•)	•
ROLLBACK	•	•	•	•
ROUTINE			(•)	
ROW			(•)	
ROWID				•
ROWNUM				•
ROWSPERMES- SAGE				•
ROWVER				•
ROWS	•		•	
SAVEPOINT			(•)	•
SCAN				•

Reserved word	ODBC	X/Open SQL	ANSI SQL-92	Solid SQL
SCHEMA	•		•	•
SCROLL	•		•	
SEARCH			(•)	
SECOND	•		•	
SECTION	•	•	•	
SELECT	•	•	•	•
SENSITIVE			(•)	
SEQUENCE			(•)	•
SERIALIZABLE				•
SESSION	•		•	
SESSION_USER	•		•	
SET	•	•	•	•
SIGNAL			(•)	
SIMILAR			(•)	
SIZE	•		•	
SMALLINT	•	•	•	•
SOME	•		•	•
SORT				•
SPACE	•			
SQL	•	•	•	•
SQLCA	•	•		
SQLCODE	•		•	
SQLERROR	•	•	•	•
SQLEXCEPTION			(•)	
SQLSTATE				
SQLWARNING	•		(•)	
START				•
STRUCTURE			(•)	
SUBSCRIBE				•

Reserved word	ODBC	X/Open SQL	ANSI SQL-92	Solid SQL
SUBSCRIPTION				•
SUBSTRING	•		•	
SUM	•	•	•	
SYNC_CONFIG				•
SYSTEM	•			
SYSTEM_USER			•	
TABLE	•	•	•	•
TEMPORARY	•		•	
TEST			(•)	
THEN	•		•	•
THERE			(•)	
TIME	•		•	•
TIMEOUT				•
TIMESTAMP	•		•	•
TIMEZONE_HOUR	•		•	
TIMEZONE_MINUTE	•		•	
TINYINT				•
TO	•	•	•	•
TRAILING			•	
TRANSACTION	•		•	•
TRANSACTIONS				•
TRANSLATE	•		•	
TRANSLATION	•		•	
TRIGGER			(•)	•
TRIM	•		•	
TRUE	•		•	
TRUNCATE				•
TYPE			(•)	
UNDER			(•)	

Reserved word	ODBC	X/Open SQL	ANSI SQL-92	Solid SQL
UNION	•	•	•	•
UNIQUE	•	•	•	•
UNKNOWN	•		•	
UNREGISTER				•
UPDATE	•	•	•	•
UPPER	•		•	
USAGE	•		•	
USER	•	•	•	•
USING	•	•	•	•
VALUE	•	•	•	•
VALUES	•	•	•	•
VARBINARY				•
VARCHAR	•	•	•	•
VARIABLE			(•)	
VARWCHAR				•
VARYING	•	•	•	
VIEW	•	•	•	•
VIRTUAL			(•)	
VISIBLE			(•)	
WAIT			(•)	•
WCHAR				•
WHEN	•		•	•
WHENEVER	•	•	•	
WHERE	•	•	•	•
WHILE			(•)	•
WITH	•	•	•	•
WITHOUT			(•)	
WORK	•	•	•	•
WRITE			•	•

---

<b>Reserved word</b>	<b>ODBC</b>	<b>X/Open SQL</b>	<b>ANSI SQL-92</b>	<b>Solid SQL</b>
WVARCHAR				•
YEAR	•		•	
ZONE			•	



### **Note**

CASCADED: The word CASCADED is reserved in IBM solidDB; however, the word is not currently used in any IBM solidDB SQL statements.

---

# Appendix D. Database System Tables and System Views

## D.1 System Tables

### D.1.1 SQL\_LANGUAGES

The SQL\_LANGUAGES system table lists the SQL standards and SQL dialects which are supported.

**Table D.1. SQL\_LANGUAGES**

Column name	Data type	Description
SOURCE	WVARCHAR	The organization that defined this specific SQL version.
SOURCE_YEAR	WVARCHAR	The year the relevant standard was approved.
CONFORMANCE	WVARCHAR	The conformance level at which conformance to the relevant standard.
INTEGRITY	WVARCHAR	Indicates whether the Integrity Enhancement Feature is supported.
IMPLEMENTATION	WVARCHAR	Identifies uniquely the vendor's SQL language; NULL if SOURCE is 'ISO'.
BINDING_STYLE	WVARCHAR	The binding style 'DIRECT', '*EMBED' or 'MODULE'.
PROGRAMMING_LANG	WVARCHAR	The host language used.

### D.1.2 SYS\_ATTAUTH

**Table D.2. SYS\_ATTAUTH**

Column name	Data type	Description
REL_ID	INTEGER	Table identifier.
UR_ID	INTEGER	User or role identifier.
ATTR_ID	INTEGER	Column identifier.
PRIV	INTEGER	Privilege info.
GRANT_ID	INTEGER	Grantor identifier.

Column name	Data type	Description
GRANT_TIM	TIMESTAMP	Grant time.

## D.1.3 SYS\_BACKGROUNDJOB\_INFO

If the body of a `START AFTER COMMIT` statement cannot be started, the reason is logged in the system table `SYS_BACKGROUNDJOB_INFO`. Only failed `START AFTER COMMIT` statements are logged in this table. If the statement (e.g. a procedure call) starts successfully, no information is stored in this system table. Statements that start successfully but do not finish executing are not stored in this system table either.

The user can retrieve information from the table `SYS_BACKGROUNDJOB_INFO` by using either an SQL `SELECT` statement or by calling a system procedure `SYS_GETBACKGROUNDJOB_INFO`. See Section D.1.3, “`SYS_BACKGROUNDJOB_INFO`” for more details.

Also a system-defined event `SYS_EVENT_SACFAILED` is posted when a `START AFTER COMMIT` statement fails to start. See its description Section F.1, “Miscellaneous Events” for more details. The application can wait for this event and use the jobid to retrieve the error message from the system table `SYS_BACKGROUNDJOB_INFO`.

The system table `SYS_BACKGROUNDJOB_INFO` can be emptied with the admin command:

**ADMIN COMMAND 'cleanbgjobinfo';**

Only a DBA can execute the '`cleanbgjobinfo`' command.

### Table D.3. SYS\_BACKGROUNDJOB\_INFO

Column name	Data type	Description
ID	INTEGER	Job identifier.
STMT	WVARCHAR	The statement that could not be executed.
USER_ID	INTEGER	User or role identifier.
ERROR_CODE	INTEGER	The error that occurred when we tried to execute the statement.
ERROR_TEXT	WVARCHAR	A description of the error.

## D.1.4 SYS\_BLOBS

This table includes information about the blobs stored into the database. Furthermore, this table sees to it that the BLOB is physically saved on disk once only even if it is logically saved several times.



**Table D.4. SYS\_BLOBS**

Column name	Data type	Description
ID	BIGINT	Blob identifier.
STARTPOS	BIGINT	Byte offset from the beginning of the blob — the start position of the pages.
ENDSIZE	BIGINT	Byte offset of the end of the last page +1.
TOTALSIZE	BIGINT	Total size of the blob.
REFCOUNT	INTEGER	The number of references, that is, the number of existing instances of the same blob.
COMPLETE	INTEGER	Indicates whether the write to the blob is ready or not.
STARTCPNUM	INTEGER	Indicates on what checkpoint level the writing of the blob started.
NUMPAGES	INTEGER	The number of pages the blob consist of.
P01_ADDR	INTEGER	First page's byte offset from the beginning of the blob.
P01_ENDSIZE	BIGINT	Last byte of the first page + 1.
P[02...50]_ADDR	INTEGER	Byte offset of pages [2...50] from the beginning of the blob.
P[02...50]_ENDSIZE	BIGINT	Last byte of the pages [2...50] +1.

## D.1.5 SYS\_CARDINAL

The data in this table is refreshed within every checkpoint, not at another time.

**Table D.5. SYS\_CARDINAL**

Column name	Data type	Description
REL_ID	INTEGER	The relation identifier as in SYS_TABLES.
CARDIN	INTEGER	The number of rows in the table.
SIZE	INTEGER	The size of the data in the table.
LAST_UPD	TIMESTAMP	The timestamp of the last update in the table.

## D.1.6 SYS\_CATALOGS

The SYS\_CATALOGS lists available catalogs.

**Table D.6. SYS\_CATALOGS**

Column name	Data type	Description
ID	INTEGER	Catalog identifier.
NAME	WVARCHAR	Catalog name.
CREATIME	TIMESTAMP	Create date and time.
CREATOR	WVARCHAR	Creator name.

## D.1.7 SYS\_CHECKSTRINGS

The SYS\_CHECKSTRINGS lists CHECK constraints of the tables.

**Table D.7. SYS\_CHECKSTRINGS**

Column name	Data type	Description
ID	INTEGER	Table identifier referring to SYS_TABLES.
CONSTRAINT_NAME	WVARCHAR	Name of the CHECK constraint (unique for the table) or an empty string for unnamed constraints (one string for all unnamed CHECK constraints. They are AND-concatenated).
CONSTRAINT	WVARCHAR	The constraint string itself. It is checked by the SQL interpreter while performing inserts/updates to the given table.

## D.1.8 SYS\_COLUMNS

This table lists all system table columns.

There are no owner or user viewing restrictions for viewing the system columns, which means owners can view columns other than those they have created in this table and users with no access rights or with specific access rights can still view any system column in this table.

**Table D.8. SYS\_COLUMNS**

Column name	Data type	Description
ID	INTEGER	Unique column identifier.
REL_ID	INTEGER	The relation identifier as in SYS_TABLES.

Column name	Data type	Description
COLUMN_NAME	WVARCHAR	The name of the column.
COLUMN_NUMBER	INTEGER	The number of the column in the table (in creation order).
DATA_TYPE	WVARCHAR	The data type of the column.
SQL_DATA_TYPE_NUM	SMALLINT	ODBC compliant data type number.
DATA_TYPE_NUMBER	INTEGER	Internal data type number.
CHAR_MAX_LENGTH	INTEGER	Maximum length for a CHAR field.
NUMERIC_PRECISION	INTEGER	Numeric precision.
NUMERIC_PREC_RADIX	SMALLINT	Numeric precision radix.
NUMERIC_SCALE	SMALLINT	Numeric scale.
NULLABLE	CHAR	Are NULL values allowed (Yes, No).
NULLABLE_ODBC	SMALLINT	ODBC, are NULL values allowed (1,0).
FORMAT	WVARCHAR	Reserved for future use.
DEFAULT_VAL	WVARCHAR	Current default value (if set).
ATTR_TYPE	INTEGER	User defined (0) or internal (>0).
REMARKS	LONG WVARCHAR	Reserved for future use.

## D.1.9 SYS\_COLUMNS\_AUX

If you insert a column with a default value to a table that has existing rows, the column default value is not appended to the existing rows. Instead, the default value defined in the column insert statement is written to the SYS\_COLUMNS\_AUX table. If an SQL query is targeted at a row that was inserted to the table before the column, the column value is read from the SYS\_COLUMNS\_AUX table unless the new column value on the row has been changed after it was inserted. Only the original default value is saved in the SYS\_COLUMNS\_AUX table.

**Table D.9. SYS\_COLUMNS\_AUX**

Column name	Data type	Description
ID	INTEGER	Table identifier.
ORIGINAL_DEFAULT	WVARCHAR	The original default value.

## D.1.10 SYS\_DL\_REPLICA\_CONFIG

This table contains the diskless configurations in the master. This table is intended for updates only through the **soldlsetup** command. Users should not modify this table directly. Doing so can have adverse repercussions.

**Table D.10. SYS\_DL\_REPLICA\_CONFIG**

Column name	Data type	Description
CFG_NAME	WVARCHAR (254) PRIMARY KEY NOT NULL	The name of the diskless replica configuration.
INI_FILE	LONG WVARCHAR	The name of the replica configuration file. The <code>solid.ini</code> file contents are inserted into this column as a blob.
LIC_FILE	LONG WVARCHAR	The name of the replica license file. The <code>solid.lic</code> file contents are inserted into this column as a blob.
SCHEMA_FILE	LONG WVARCHAR	The name of the replica schema. The schema file contents are inserted into this column as a blob.

## D.1.11 SYS\_DL\_REPLICA\_DEFAULT

This table contains the diskless default configurations in the master. This table is intended for updates only through the **soldlsetup** command. Users should not modify this table directly. Doing so can have adverse repercussions.

**Table D.11. SYS\_DL\_REPLICA\_DEFAULT**

Column name	Data type	Description
REPLICA_NAME	VARCHAR(254) NOT NULL PRIMARY KEY	The name of the replica.
INI_CFG	VARCHAR(254) REFERENCE SYS_DL_REPLICA_CONFIG(CFG_NAME)	The name of the replica configuration file.

Column name	Data type	Description
LIC_CFG	VARCHAR(254) REFERENCE SYS_DL_REPLICA_CONFIG(CFG_NAME)	The name of the replica license file.
SCHEMA_CFG	VARCHAR(254) REFERENCE SYS_DL_REPLICA_CONFIG(CFG_NAME)	The name of the replica schema.

## D.1.12 SYS\_EVENTS

**Table D.12. SYS\_EVENTS**

Column name	Data type	Description
ID	INTEGER	Unique event identifier.
EVENT_NAME	WVARCHAR	The name of the event.
EVENT_PARAMCOUNT	INTEGER	Number of parameters.
EVENT_PARAMTYPES	LONG VARBINARY	Types of parameters.
EVENT_TEXT	WVARCHAR	The body of the event.
EVENT_SCHEMA	WVARCHAR	The owner of the event.
EVENT_CATALOG	WVARCHAR	The owner of the event.
CREATIME	TIMESTAMP	Creation time.
TYPE	INTEGER	Reserved for future use.

## D.1.13 SYS\_FORKEYPARTS

**Table D.13. SYS\_FORKEYPARTS**

Column name	Data type	Description
KEY_CATALOG	INTEGER	Creator name or the owner of the key.
ID	INTEGER	Foreign key identifier.
KEYP_NO	INTEGER	Keypart number.

Column name	Data type	Description
ATTR_NO	INTEGER	Column number.
ATTR_ID	INTEGER	Column identifier.
ATTR_TYPE	INTEGER	Column type.
CONST_VALUE	VARBINARY	Possible internal constant value; otherwise NULL.

## D.1.14 SYS\_FORKEYS

**Table D.14. SYS\_FORKEYS**

Column name	Data type	Description
ID	INTEGER	Foreign key identifier.
REF_REL_ID	INTEGER	Referenced table identifier.
CREATE_REL_ID	INTEGER	Creator table identifier.
REF_KEY_ID	INTEGER	Referenced key identifier.
REF_TYPE	INTEGER	Reference type.
KEY_SCHEMA	WVARCHAR	Creator name.
KEY_CATALOG	WVARCHAR	Creator name or the owner of the key.
KEY_NREF	INTEGER	Number of referenced key parts.

## D.1.15 SYS\_HOTSTANDBY

Deprecated. Relevant to versions before 4.0.

## D.1.16 SYS\_INFO

**Table D.15. SYS\_INFO**

Column name	Data type	Description
PROPERTY	WVARCHAR	The name of the property.
VALUE_STR	WVARCHAR	Value as a string.
VALUE_INT	INTEGER	Value as an integer.

## D.1.17 SYS\_KEYPARTS

**Table D.16. SYS\_KEYPARTS**

Column name	Data type	Description
ID	INTEGER	This column is a foreign key reference to sys_keys.id, so that you can determine which key each keypart is part of.
REL_ID	INTEGER	The relation identifier as in SYS_TABLES.
KEYP_NO	INTEGER	Keypart identifier.
ATTR_ID	INTEGER	Column identifier.
ATTR_NO	INTEGER	The number of the column in the table (in creation order).
ATTR_TYPE	INTEGER	The type of the column.
CONST_VALUE	VARBINARY	Constant value or NULL.
ASCENDING	CHAR	Is the key ascending (Yes) or descending (No).

## D.1.18 SYS\_KEYS

All database tables must have one clustering key. This key defines the physical sorting order of the data. It has no capacity impact. If a primary key is defined, the primary key is used as the clustering key. If no primary key is defined, an entry with key\_name "\$CLUSTKEY\_xxxxx" will be automatically created in SYS\_KEYS.

If there is a primary key definition for the table, there will be an entry in SYS\_KEYS with a key\_name like "\$PRIMARYKEY\_xxxx" for this entry. The key\_primary and key\_clustering columns will have a value YES.

If there is no primary key definition for the table, there will be an entry in SYS\_KEYS with a key\_name like "\$CLUSTKEY\_xxxxx". The key\_primary column will have a value NO and key\_clustering column will have a value YES.

**Table D.17. SYS\_KEYS**

Column name	Data type	Description
ID	INTEGER	Unique key identifier.
REL_ID	INTEGER	The relation identifier as in SYS_TABLES.
KEY_NAME	WVARCHAR	The name of the key.
KEY_UNIQUE	CHAR	Is the key unique (Yes, No).

Column name	Data type	Description
KEY_NON_UNIQUE_ODBC	SMALLINT	ODBC, is the key NOT unique (1, 0).
KEY_CLUSTERING	CHAR	Is the key a clustering key (Yes, No).
KEY_PRIMARY	CHAR	Is the key a primary key (Yes, No).
KEY_PREJOINED	CHAR	Reserved for future use.
KEY_SCHEMA	WVARCHAR	The owner of the key.
KEY_NREF	INTEGER	When creating a primary key, the server uses ALL fields of the table, even if the user specified N fields (the N fields specified by the user become the first N fields of the key). KEY_NREF = N, i.e. the number of fields specified by the user.

## D.1.19 SYS\_PROCEDURES

This system table lists procedures.

Specific users are restricted from viewing procedures. Owners are restricted to viewing procedures they have created. Users can only view procedures to which they have execute access to see the procedure definition. If users have no access rights, they are restricted from viewing all procedures. Note that execute access does not allow users to see procedure definitions. No restrictions apply to DBAs.

**Table D.18. SYS\_PROCEDURES**

Column name	Data type	Description
ID	INTEGER	Unique procedure identifier.
PROCEDURE_NAME	WVARCHAR	Procedure name.
PROCEDURE_TEXT	LONG WVARCHAR	Procedure body.
PROCEDURE_BIN	LONG VARBINARY	Compiled form of the procedure.
PROCEDURE_SCHEMA	WVARCHAR	The name of the schema containing PROCEDURE_NAME.
PROCEDURE_CATALOG	WVARCHAR	The name of the catalog containing PROCEDURE_NAME.
CREATIME	TIMESTAMP	Creation time.
TYPE	INTEGER	Reserved for future use.



## D.1.20 SYS\_PROCEDURE\_COLUMNS

The SYS\_PROCEDURE\_COLUMNS defines input parameters and result set columns.

**Table D.19. SYS\_PROCEDURE\_COLUMNS**

Column name	Data type	Description
PROCEDURE_ID	INTEGER	Procedure identifier.
COLUMN_NAME	WVARCHAR	Procedure column name.
COLUMN_TYPE	SMALLINT	Procedure column type (SQL_PARAM_INPUT or SQL_RESULT_COL).
DATA_TYPE	SMALLINT	Column's SQL data type.
TYPE_NAME	WVARCHAR	Column's SQL data type name.
COLUMN_SIZE	INTEGER	Size of the procedure column.
BUFFER_LENGTH	INTEGER	Column size in bytes.
DECIMAL_DIGITS	SMALLINT	Decimal digits of the procedure column.
NUM_PREC_RADIX	SMALLINT	Radix for numeric data types (2, 10, or NULL if not applicable).
NULLABLE	SMALLINT	Whether the procedure column accepts a NULL value.
REMARKS	WVARCHAR	A description of the procedure column.
COLUMN_DEF	WVARCHAR	Column's default value. Always NULL, that is, no default value is specified.
SQL_DATA_TYPE	SMALLINT	SQL data type.
SQL_DATETIME_SUB	SMALLINT	Subtype code for datetime. Always NULL.
CHAR_OCTET_LENGTH	INTEGER	Maximum length in bytes of a character or binary data type column.
ORDINAL_POSITION	INTEGER	Ordinal position of the column.
IS_NULLABLE	WVARCHAR	Always "YES".

## D.1.21 SYS\_PROPERTIES

This table is for internal use of HSB only.

**Table D.20. SYS\_PROPERTIES**

Column name	Data type	Description
KEY	WVARCHAR	Property identifier.
VALUE	WVARCHAR	Value of a property.
MODTIME	TIMESTAMP	Creation time for the property.

## D.1.22 SYS\_RELAUTH

This table contains GRANT privileges issued for each table name and user name combination. When a database is created with no GRANT statements executed, this table is empty.

**Table D.21. SYS\_RELAUTH**

Column Name	Description
REL_ID	Table or object identifier.
UR_ID	User or role identifier.
PRIV	Information about privileges of a user or a role. Each privilege is related to someone (GRANT_ID) who has granted it.
GRANT_ID	Grantor identifier.
GRANT_TIM	Grant time.
GRANT_OPT	If set to "Yes", the user who receives the privilege may grant the privilege to other users. The possible values are "Yes" or "No".

## D.1.23 SYS\_SCHEMAS

The SYS\_SCHEMAS lists available schemas.

**Table D.22. SYS\_SCHEMAS**

Column name	Data type	Description
ID	INTEGER	Schema identifier.
NAME	WVARCHAR	Schema name.
OWNER	WVARCHAR	Schema owner name.
CREATIME	TIMESTAMP	Create date and time.
SCHEMA_CATALOG	WVARCHAR	Schema catalog.

## D.1.24 SYS\_SEQUENCES

**Table D.23. SYS\_SEQUENCES**

Column name	Data type	Description
SEQUENCE_NAME	WVARCHAR	Sequence name.
ID	INTEGER	Unique identifier.
DENSE	CHAR	Is the sequence dense or sparse.
SEQUENCE_SCHEMA	WVARCHAR	The name of the schema containing SEQUENCE_NAME.
SEQUENCE_CATALOG	WVARCHAR	The name of the catalog containing SEQUENCE_NAME.
CREATIME	TIMESTAMP	Creation time.

## D.1.25 SYS\_SYNC\_REPLICA\_PROPERTIES

**Table D.24. SYS\_SYNC\_REPLICA\_PROPERTIES**

Column name	Data type	Description
ID	INTEGER	Replica ID.
NAME	VARCHAR	Property name.
VALUE	VARCHAR	Property value.

The primary key is on the ID and NAME fields.

## D.1.26 SYS\_SYNONYM

**Table D.25. SYS\_SYNONYM**

Column name	Data type	Description
TARGET_ID	INTEGER	Reserved for future use.
SYNON	INTEGER	Reserved for future use.

## D.1.27 SYS\_TABLEMODES

**Table D.26. SYS\_TABLEMODES**

Column name	Data type	Description
ID	INTEGER	Relation identifier.
MODE	WVARCHAR	Concurrency control mode (allowed values: OPTIMISTIC, PESSIMISTIC, MAINMEMORY, or MAINMEMORY PESSIMISTIC).
MODIFY_TIME	TIMESTAMP	Last modify time.
MODIFY_USER	WVARCHAR	Last user that modified.

`SYS_TABLEMODES` shows the mode only of tables for which the mode was explicitly set. `SYS_TABLEMODES` doesn't show the mode of tables that were left at the default mode. (The default mode is "optimistic" unless you set the `solid.ini` configuration parameter `Pessimistic=Yes`.)

To list the names and modes of tables that were explicitly set to optimistic or pessimistic, execute the command:

```
SELECT SYS_TABLEMODES.ID, table_name, mode
FROM SYS_TABLES, SYS_TABLEMODES
WHERE SYS_TABLEMODES.ID = SYS_TABLES.ID;
```

The output will look like:

```
   ID  TABLE_NAME  MODE
   --  -
10054 TABLE2      OPTIMISTIC
10056 TABLE3      PESSIMISTIC
```

For more information about setting the concurrency control mode, see the section called “Setting the Concurrency (Locking) Mode to Optimistic or Pessimistic”.

## D.1.28 SYS\_TABLES

This table lists all the system tables.

There are no restrictions for viewing the system tables, which means even users with no access rights can view them. However, specific users are restricted from viewing the user table information. Owners are restricted to viewing user tables they have created and users can only view tables to which they have `INSERT`, `UPDATE`,

DELETE, or SELECT access. Users are restricted from viewing any user tables if they have no access rights. No restrictions apply to DBAs.

**Table D.27. SYS\_TABLES**

Column name	Data type	Description
ID	INTEGER	Unique table identifier.
TABLE_NAME	WVARCHAR	The name of the table.
TABLE_TYPE	WVARCHAR	The type of the table (BASE TABLE or VIEW).
TABLE_SCHEMA	WVARCHAR	The name of the schema containing TABLE_NAME
TABLE_CATALOG	WVARCHAR	The name of the catalog containing TABLE_NAME.
CREATIME	TIMESTAMP	The creation time of the table.
CHECKSTRING	LONG WVARCHAR	Possible check option defined for the table.
REMARKS	LONG WVARCHAR	Reserved for future use.

## D.1.29 SYS\_TRIGGERS

This system table lists procedures.

Specific users are restricted from viewing triggers. Owners are restricted to viewing only those triggers that they have created. Normal users are restricted from viewing triggers. No restrictions apply to DBAs.

**Table D.28. SYS\_TRIGGERS**

Column name	Data type	Description
ID	INTEGER	Unique table identifier.
TRIGGER_NAME	WVARCHAR	Trigger name.
TRIGGER_TEXT	LONG WVARCHAR	Trigger body.
TRIGGER_BIN	LONG VARBINARY	Compiled form of the trigger.
TRIGGER_SCHEMA	WVARCHAR	The name of the schema containing TRIGGER_NAME.
TRIGGER_CATALOG	WVARCHAR	The name of the catalog containing TRIGGER_NAME.
TRIGGER_ENABLED	CHAR	If triggers are enabled "YES"; otherwise "NO".
CREATIME	TIMESTAMP	The creation time of the trigger.
TYPE	INTEGER	Reserved for future use.
REL_ID	INTEGER	The relation identifier.

## D.1.30 SYS\_TYPES

**Table D.29. SYS\_TYPES**

Column name	Data type	Description
TYPE_NAME	WVARCHAR	The name of the data type.
DATA_TYPE	SMALLINT	ODBC, data type number.
PRECISION	INTEGER	ODBC, the precision of the data type.
LITERAL_PREFIX	WVARCHAR	ODBC, possible prefix for literal values.
LITERAL_SUFFIX	WVARCHAR	ODBC, possible suffix for literal values.
CREATE_PARAMS	WVARCHAR	ODBC, the parameters needed to create a column of the data type.
NULLABLE	SMALLINT	ODBC, can the data type contain NULL values.
CASE_SENSITIVE	SMALLINT	ODBC, is the data type case sensitive.
SEARCHABLE	SMALLINT	ODBC, the supported search operations.
UNSIGNED_ATTRIBUTE	SMALLINT	ODBC, is the data type unsigned.
MONEY	SMALLINT	ODBC, whether the data is a money data type.
AUTO_INCREMENT	SMALLINT	ODBC, whether the data type is autoincrementing.
LOCAL_TYPE_NAME	WVARCHAR	ODBC, has the data type another implementation defined name.
MINIMUM_SCALE	SMALLINT	ODBC, the minimum scale of the data type.
MAXIMUM_SCALE	SMALLINT	ODBC, the maximum scale of the data type.

## D.1.31 SYS\_URole

The SYS\_URole contains mapping of users to roles.

**Table D.30. SYS\_URole**

Column name	Data type	Description
U_ID	INTEGER	User identifier.
R_ID	INTEGER	Role identifier.

## D.1.32 SYS\_USERS

The SYS\_USERS list information about users and roles.

**Table D.31. SYS\_USERS**

Column name	Data type	Description
ID	INTEGER	User or role identifier.
NAME	WVARCHAR	User or role name.
TYPE	WVARCHAR	User type, either USER or ROLE.
PRIV	INTEGER	Privilege information.
PASSW	VARBINARY	Password in encrypted format.
PRIORITY	INTEGER	Reserved for future use.
PRIVATE	INTEGER	Specifies whether user is private or public.
LOGIN_CATALOG	WVARCHAR	Reserved for future use.

## D.1.33 SYS\_VIEWS

**Table D.32. SYS\_VIEWS**

Column name	Data type	Description
V_ID	INTEGER	Unique identifier for this view.
TEXT	LONG WVARCHAR	View definition.
CHECKSTRING	LONG WVARCHAR	Possible CHECK OPTION defined for the view.
REMARKS	LONG WVARCHAR	Reserved for future use.

## D.2 System Tables for Data Synchronization

IBM solidDB contains a number of system tables that are used for implementing synchronization functionality. In general, these tables are for internal use only. However, you may need to know the contents of these tables when developing and troubleshooting a new application.

Note that the tables are presented in alphabetical order.

## D.2.1 SYS\_BULLETIN\_BOARD

This table contains persistent parameters that are always available in the parameter bulletin board when transactions are executed in this database catalog.

**Table D.33. SYS\_BULLETIN\_BOARD**

Column Name	Description
PARAM_NAME	Name of the persistent parameter.
PARAM_VALUE	Value of the parameter.
PARAM_CATALOG	Defines the master/replica catalog.

## D.2.2 SYS\_PUBLICATION\_ARGS

This table contains the publication input arguments in this master database

**Table D.34. SYS\_PUBLICATION\_ARGS**

Column Name	Description
PUBL_ID	Internal ID of the publication.
ARG_NUMBER	Sequence number of the argument.
NAME	Name of the argument.
TYPE	Type of the argument.
LENGTH_OR_PRECISION	Length or precision of the argument.
SCALE	Scale of the argument.

## D.2.3 SYS\_PUBLICATION\_REPLICA\_ARGS

This table contains the definition of the publication arguments in a replica database.

**Table D.35. SYS\_PUBLICATION\_REPLICA\_ARGS**

Column Name	Description
MASTER_ID	Internal ID of the master from which the data is refreshed.
PUBL_ID	Internal ID of the publication.
ARG_NUMBER	Sequence number of the argument.



Column Name	Description
NAME	Name of the argument.
LENGTH_OR_PRECISION	Length or precision of the argument.
SCALE	Scale of the argument.

## D.2.4 SYS\_PUBLICATION\_REPLICA\_STMTARGS

This table contains the mapping between the publication arguments and the statements in the replica.

**Table D.36. SYS\_PUBLICATION\_REPLICA\_STMTARGS**

Column Name	Description
MASTER_ID	Internal ID of the master from which the data is refreshed.
PUBL_ID	Internal ID of the publication.
STMT_NUMBER	Sequence number of the statement.
STMT_ARG_NUMBER	Sequence number of the statement.
PUBL_ARG_NUMBER	Sequence number of the publication argument.

## D.2.5 SYS\_PUBLICATION\_REPLICA\_STMTS

This table contains the definition of the publication statements in a replica database.

**Table D.37. SYS\_PUBLICATION\_REPLICA\_STMTS**

Column Name	Description
MASTER_ID	Internal ID of the master from which the data is refreshed.
PUBL_ID	Internal ID of the publication.
STMT_NUMBER	Sequence number of the statement.
REPLICA_CATALOG	Name of the target catalog in the replica database.
REPLICA_SCHEMA	Name of the target schema in the replica database.
REPLICA_TABLE	Name of the target table in the replica database.
TABLE_ALIAS	Alias name of the target table.
REPLICA_FROM_STR	SQL FROM tables as string.
WHERE STR	SQL WHERE arguments as string.

Column Name	Description
LEVEL	Level of this SQL statement in this publication hierarchy.

## D.2.6 SYS\_PUBLICATION\_STMTARGS

This table contains mapping between the publication arguments and the statements in the master database.

**Table D.38. SYS\_PUBLICATION\_STMTARGS**

Column Name	Description
PUBL_ID	Internal ID of the publication.
STMT_NUMBER	Sequence number of the statement.
STMT_ARG_NUMBER	Sequence number of the statement argument.
PUBL_ARG_NUMBER	Sequence number of the publication argument.

## D.2.7 SYS\_PUBLICATION\_STMTS

This table contains the publication statements in the master database.

**Table D.39. SYS\_PUBLICATION\_STMTS**

Column Name	Description
PUBL_ID	Internal ID of the publication.
MASTER_SCHEMA	Name of the publication schema in the master database.
MASTER_TABLE	Name of the table in the master database.
REPLICA_SCHEMA	Name of the schema in the replica database.
REPLICA_TABLE	Name of the table in the replica database.
TABLE_ALIAS	The alias name of the target table.
MASTER_SELECT_STR	SQL SELECT INTO columns as string.
REPLICA_SELECT_STR	SQL SELECT INTO columns as string.
MASTER_FROM_STR	SQL SELECT FROM tables as string.
REPLICA_FROM_STR	SQL SELECT FROM tables as string.
WHERE_STR	SQL WHERE arguments as a string.
DELETEFLAG_STR	For internal use.

Column Name	Description
LEVEL	Level of this SQL statement in the publication hierarchy.

## D.2.8 SYS\_PUBLICATIONS

This table contains the publications that have been defined in this master database.

**Table D.40. SYS\_PUBLICATIONS**

Column Name	Description
ID	Internal ID of the publication.
NAME	Name of the publication.
CREATOR	User ID of the creator of the publication.
CREATETIME	Date and time when the publication was created.
ARGCOUNT	Number of input arguments for this publication.
STMTCOUNT	Number of statement contained in this publication.
TIMEOUT	N/A.
TEXT	Contents of the CREATE PUBLICATION statement.
PUBL_CATALOG	Defines the master catalog.

## D.2.9 SYS\_PUBLICATIONS\_REPLICA

This table contains publications that are being used in this replica database.

**Table D.41. SYS\_PUBLICATIONS\_REPLICA**

Column Name	Description
MASTER_ID	Internal ID of the master from which the data is refreshed.
ID	Internal ID of the publication.
NAME	Name of the publication.
CREATOR	User ID of the creator of the publication.
ARGCOUNT	Number of input arguments for this publication.
STMTCOUNT	Number of statements contained by this publication.

## D.2.10 SYS\_SYNC\_BOOKMARKS

This table contains bookmarks that are being used in a master database.

**Table D.42. SYS\_SYNC\_BOOKMARKS**

Column Name	Description
BM_ID	Internal ID of the bookmark.
BM_CATALOG	Reserved for future use.
BM_NAME	Name of the bookmark.
BM_VERSION	Internal version information of the bookmark in the master.
BM_CREATOR	User ID of the creator of the bookmark.
BM_CREATIME	Create time of the bookmark.

## D.2.11 SYS\_SYNC\_HISTORY\_COLUMNS

If you turn on synchronization history for a table, you may turn it on for all columns, or only for a subset of columns. If you turn it on for a subset of columns, then the SYS\_SYNC\_HISTORY\_COLUMNS table records which columns you are keeping synchronization history information for. There is one row in SYS\_SYNC\_HISTORY\_COLUMNS for each column that you keep synchronization history for.

**Table D.43. SYS\_SYNC\_HISTORY\_COLUMNS**

Column Name	Description
REL_ID	The ID of the table to keep sync history for.
COLUMN_NUMBER	The ordinal number of the column in that table that we keep sync history for. (E.g. if we keep sync history for the second column in the table, then this field will hold the number 2.)

## D.2.12 SYS\_SYNC\_INFO

This table contains synchronization information, one row for each node.

**Table D.44. SYS\_SYNC\_INFO**

Column Name	Description
NODE_NAME	Master or replica node.

Column Name	Description
NODE_CATALOG	Catalog where node belongs.
IS_MASTER	IF YES, this node is a master.
IS_REPLICA	If YES, this node is a replica.
CREATIME	Node create data and time.
CREATOR	Node creator user name.

## D.2.13 SYS\_SYNC\_MASTER\_MSGINFO

This table contains information about the currently active message in the master database.

Data in this table is used to control the synchronization process between the replica and master database. This table also contains information that is useful for troubleshooting purposes. If the execution of a message halts in the master database due to an error, you can query this table to obtain the reason for the problem, as well as the transaction and statement that caused the error.

**Table D.45. SYS\_SYNC\_MASTER\_MSGINFO**

Column Name	Description
STATE	<p>Current state of the message. The following values are possible:</p> <ul style="list-style-type: none"> <li>• 0 = DELETED N/A (internal non-persistent state)</li> <li>• 1 = ERROR - Error has occurred during message processing; the reason for the error was recorded in the error-columns of the row.</li> <li>• 10 = RECEIVED - master has received a message from the replica</li> <li>• 11 = SAVED - message has been saved in the master database and is being processed</li> <li>• 12 = READY - master has processed the message</li> <li>• 13 = SENT - N/A (internal non-persistent state)</li> </ul>
REPLICA_ID	ID of the replica database from which the message was sent.
MASTER_ID	ID of the database to which the master is sent.
MSG_ID	Internal ID of the message.
MSG_NAME	Name of the message given by the user.

Column Name	Description
MSG_TIME	Create time of the message.
MSG_BYTE_COUNT	Size of the message in bytes.
CREATE_UID	ID of the user who created the message.
FORWARD_UID	ID of the user who forwarded the message.
ERROR_CODE	Code of the error that caused the termination of the message execution. You can determine the transaction and statement that caused the error from the TRX_ID and STMT_ID information.
ERROR_TEXT	Description of the error that caused the termination of the message execution.
TRX_ID	Sequence number of the transaction that caused the error.
STMT_ID	Sequence number of the statement of a transaction that caused an error.
ORD_ID_COUNT	N/A (internal use only).
ORD_ID	N/A (internal use only).
FLAGS	NULL or 0 = Normal message.  1 = Message is deleted when reply is sent to replica.
FAILED_MSG_ID	This is an INTEGER column which is part of the primary key. The value is zero for normal messages. The value is msg_id if LOG_ERRORS option is ON and any errors exists.

## D.2.14 SYS\_SYNC\_MASTER\_RECEIVED\_BLOB\_REFS

The received BLOBs are stored in this table on the master. The implementation sees to it that the BLOB is physically saved on disk once only even if it is logically saved several times.

**Table D.46. SYS\_SYNC\_MASTER\_RECEIVED\_BLOB\_REFS**

Column Name	Description
REPLICA_ID	Internal ID of the replica database from which the message was received.
MSG_ID	Internal ID of the message.
BLOB_NUM	The number that identifies the BLOB.
DATA	A reference to the BLOB.

## D.2.15 SYS\_SYNC\_MASTER\_RECEIVED\_MSGPARTS

This table contains parts of the messages that were received in the master database from a replica database, but not yet processed in the master database.

**Table D.47. SYS\_SYNC\_MASTER\_RECEIVED\_MSGPARTS**

Column Name	Description
REPLICA_ID	Internal ID of the replica database from which the message was received.
MSG_ID	Internal ID of the message.
PART_NUMBER	Sequence number of the message part.
DATA_LENGTH	Length of the data in the message part.
DATA	Data of the message part.

## D.2.16 SYS\_SYNC\_MASTER\_RECEIVED\_MSGS

This table contains messages that were received in the master database from a replica database, but are not yet processed in the master database.

**Table D.48. SYS\_SYNC\_MASTER\_RECEIVED\_MSGS**

Column Name	Description
REPLICA_ID	Internal ID of the replica database from which the message has been received.
MSG_ID	Internal ID of the message.
CREATIME	Create time of the message.
CREATOR	User ID of the user who created the message.

## D.2.17 SYS\_SYNC\_MASTER\_STORED\_BLOB\_REFS

The BLOBs to be sent are stored in this table on the master. The implementation sees to it that the BLOB is physically saved on disk once only even if it is logically saved several times.

**Table D.49. SYS\_SYNC\_MASTER\_STORED\_BLOB\_REFS**

Column Name	Description
REPLICA_ID	Internal ID of the replica database to which the message will be sent.

Column Name	Description
MSG_ID	Internal ID of the message.
BLOB_NUM	The number that identifies the BLOB.
DATA	A reference to the BLOB.

## D.2.18 SYS\_SYNC\_MASTER\_STORED\_MSGPARTS

This table contains parts of the message result sets that were created in the master database, but not yet sent to the replica database.

**Table D.50. SYS\_SYNC\_MASTER\_STORED\_MSGPARTS**

Column Name	Description
REPLICA_ID	Internal ID of the replica database to which the message will be sent.
MSG_ID	Internal ID of the message.
ORDER_ID	Sequence number of the result set.
RESULT_SET_ID	Internal ID of the result set.
RESULT_SET_TYPE	Type of the result set.
PART_NUMBER	Sequence number of the message part in the result set.
DATA_LENGTH	Length of the data in the message part in the result set.
DATA	Data of the message part.

## D.2.19 SYS\_SYNC\_MASTER\_STORED\_MSGS

This table contains messages that were created in the master database, but not yet sent to the replica database.

**Table D.51. SYS\_SYNC\_MASTER\_STORED\_MSGS**

Column Name	Description
REPLICA_ID	Internal ID of the replica database to which the message will be sent.
MSG_ID	Internal ID of the message.
CREATIME	Create time of the message.
CREATOR	User ID of the user who created the message.



## D.2.20 SYS\_SYNC\_MASTER\_SUBSC\_REQ

This table contains the list of requested subscriptions waiting to be executed in the master.

**Table D.52. SYS\_SYNC\_MASTER\_SUBSC\_REQ**

Column Name	Description
REPLICA_ID	Internal ID of the replica from which the statement has arrived.
MSG_ID	Internal ID of the message in which the statement has arrived.
ORD_ID	Sequence number of the subscription.
TRX_ID	Internal ID of the transaction to which the subscription belongs.
STMT_ID	Internal ID of the statement in the subscription.
REQUEST_ID	N/A.
PUBL_ID	Internal ID of the subscribed/refreshed publication.
VERSION	Internal version information of the subscription in the master.
REPLICA_VERSION	Internal version information of the subscription in the replica.
FULLSUBSC	Indicates if the subscription is full or incremental.

## D.2.21 SYS\_SYNC\_MASTER\_VERSIONS

This table contains the list of subscriptions (that have been subscribed) to replica databases from the master database.

**Table D.53. SYS\_SYNC\_MASTER\_VERSIONS**

Column Name	Description
REPLICA_ID	Internal ID of the replica database.
REQUEST_ID	Sequence number of the subscription.
VERS_TIME	Create time of the subscription.
PUBL_ID	ID of the publication.
TABNAME	Name of the table of the publication.
TABSCHEMA	Name of the schema of the table.
PARAM_CRC	N/A (for internal use only).
PARAM	Parameters of the publication in binary format.

Column Name	Description
VERSION	Version of the data that has been requested from the replica database.

## D.2.22 SYS\_SYNC\_MASTERS

This table contains the list of master databases accessed by the replica.

**Table D.54. SYS\_SYNC\_MASTERS**

Column Name	Description
NAME	Given name of the master database.
ID	Internal ID of the master database.
REMOTE_NAME	N/A.
REPLICA_NAME	Given name of the replica database.
REPLICA_ID	Surrogate identifier for the replica database.
REPLICA_CATALOG	Defines the replica catalog which is registered to this master.
CONNECT	Connect string of the master database.
CREATOR	ID of the user who set the database as a master.
ISDEFAULT	Reserved for future use.

## D.2.23 SYS\_SYNC\_RECEIVED\_BLOB\_ARGS

This table is on the master. The BLOB parameters are saved in this table when the message from the replica is extracted. The rows only exist until the transaction in the message has been executed.

**Table D.55. SYS\_SYNC\_RECEIVED\_BLOB\_ARGS**

Column Name	Description
REPLICA	Internal ID of the replica from which the BLOB parameters have arrived.
MSG	Internal ID of the message.
ORD_ID	Sequence number of the BLOB part.
TRX_ID	The transaction ID identifies the transaction.
ID	Internal ID of the user.
ARGNO	Number of the parameter.
ARG_VALUE	Value of the parameter in binary format.

## D.2.24 SYS\_SYNC\_RECEIVED\_STMTS

This table contains the propagated statements that have been received in the master database.

**Table D.56. SYS\_SYNC\_RECEIVED\_STMTS**

Column Name	Description
REPLICA	Internal ID of the replica from which the statement has arrived.
MSG	Internal ID of the message in which the statement has arrived.
ORD_ID	N/A.
TXN_ID	Internal ID of the transaction to which the statement belongs.
ID	Sequence number of the statement within the transaction.
CLASS	Type of the constant.
STRING	the SQL statement as a string.
ARG_COUNT	Number of parameters bound to the statement.
ARG_TYPES	Types of the parameters bound to the statement.
ARG_VALUES	Values of the parameters in binary format.
USER_ID	ID of the user who has saved the statement.
REQUEST_ID	N/A.
FLAGS	This indicates the error-handling mode (e.g. IGNORE_ERRORS, LOG_ERRORS, etc.).
ERRCODE	This has the error code if a statement failed while executing on the master.
ERR_STR	This has a description of the error that occurred if a statement failed while executing on the master.

## D.2.25 SYS\_SYNC\_REPLICA\_MSGINFO

This table contains information about currently active messages in the replica database.

Data in this table is used to control the synchronization process between the replica and master database. This table also contains information that is useful for troubleshooting purposes. If the execution of a message halts in the replica database due to an error, you can query this table to obtain the reason for the problem, as well as the transaction and statement that caused the error.

**Table D.57. SYS\_SYNC\_REPLICA\_MSGINFO**

Column Name	Description
STATE	<p>Current state of the message. The following values are possible:</p> <ul style="list-style-type: none"> <li>• 0 = DELETED N/A (internal non-persistent state)</li> <li>• 1 = ERROR - Internal error has occurred during message processing; the reason for the error was recorded in the error-columns of the row.</li> <li>• 20 = R_INIT - N/A (internal non-persistent state)</li> <li>• 21 = R_INITEND - N/A (internal non-persistent state)</li> <li>• 22 = R_SAVED - Replica has saved an outgoing message</li> <li>• 23 = R_SENT - Replica has sent a message to the master</li> <li>• 24 = R_RECEIVED - Replica has received a reply message from the master</li> <li>• 25 = R_EXECUTE - The reply message in a replica is ready for execution</li> <li>• 26 = R_EXECUTE_NOTIFYMASTER - Replica has received a reply, but not yet confirmed it with the master</li> </ul>
MASTER_ID	ID of the master database to which the message is sent.
MASTER_NAME	Name of the master database to which the message is sent.
MSG_ID	Internal ID of the message.
MSG_NAME	Name of the message given by the user.
MSG_TIME	Create time of the message.
MSG_BYTE_COUNT	Size of the message in bytes.
CREATE_UID	ID of the user who created the message.
FORWARD_UID	ID of the user who sent the message.
ERROR_CODE	Code of the error that caused the message execution to terminate.
ERROR_TEXT	Description of the error that caused the message execution to terminate.
FLAGS	NULL or 0 = Normal message.

Column Name	Description
	1 = Message is deleted when a reply is received from master. 3 = Message is a registration message.

## D.2.26 SYS\_SYNC\_REPLICA\_RECEIVED\_BLOB\_REFS

The received BLOBs are stored in this table. The implementation sees to it that the BLOB is physically saved on disk once only even if it is logically saved several times.

**Table D.58. SYS\_SYNC\_REPLICA\_RECEIVED\_BLOB\_REFS**

Column Name	Description
MASTER_ID	Internal ID of the master database from which the message has been received.
MSG_ID	Internal ID of the message.
BLOB_NUM	The number that identifies the BLOB.
DATA	A reference to the BLOB.

## D.2.27 SYS\_SYNC\_REPLICA\_RECEIVED\_MSGPARTS

This table contains parts of the reply messages that have been received into the replica database from the master database, but are not yet processed in the replica database.

**Table D.59. SYS\_SYNC\_REPLICA\_RECEIVED\_MSGPARTS**

Column Name	Description
MASTER_ID	Internal ID of the master database from which the message has been received.
MSG_ID	Internal ID of the message.
PART_NUMBER	Sequence number of the message part.
DATA_LENGTH	Length of the data in the message part.
RESULT_SET_TYPE	Type of the result set.
DATA	Data of the message part.

## D.2.28 SYS\_SYNC\_REPLICA\_RECEIVED\_MSGS

This table contains reply messages that were received in the replica database from the master database, but not yet processed in the replica database.

**Table D.60. SYS\_SYNC\_REPLICA\_RECEIVED\_MSGS**

Column Name	Description
MASTER_ID	Internal ID of the master database from which the message has been received.
MSG_ID	Internal ID of the message.
CREATIME	Create time of the message.
CREATOR	User ID of the user who created the message.

## D.2.29 SYS\_SYNC\_REPLICA\_STORED\_BLOB\_REFS

The BLOBs in the flow message are stored in this table. The implementation sees to it that the BLOB is physically saved on disk once only even if it is logically saved several times.

**Table D.61. SYS\_SYNC\_REPLICA\_STORED\_BLOB\_REFS**

Column Name	Description
MASTER_ID	Internal ID of the master database from which the message has been received.
MSG_ID	Internal ID of the message.
BLOB_NUM	The number that identifies the BLOB.
DATA	A reference to the BLOB.

## D.2.30 SYS\_SYNC\_REPLICA\_STORED\_MSGS

This table contains messages that were created in the replica database, but not yet sent to the master database.

**Table D.62. SYS\_SYNC\_REPLICA\_STORED\_MSGS**

Column Name	Description
MASTER_ID	Internal ID of the master database to which the message will be sent.
MSG_ID	Internal ID of the message.

Column Name	Description
CREATIME	Create time of the message.
CREATOR	User ID of the user who has created the message.

## D.2.31 SYS\_SYNC\_REPLICA\_STORED\_MSGPARTS

This table contains parts of the messages that were created in the replica database, but not yet sent to the master database.

**Table D.63. SYS\_SYNC\_REPLICA\_STORED\_MSGPARTS**

Column Name	Description
MASTER_ID	Internal ID of the master database to which the message will be sent.
MSG_ID	Internal ID of the message.
PART_NUMBER	Sequence number of the message part.
DATA_LENGTH	Length of the data in the message part.
DATA	Data of the message part.

## D.2.32 SYS\_SYNC\_REPLICA\_VERSIONS

This table contains the list of subscriptions (that have been subscribed) to this replica database from the master database.

**Table D.64. SYS\_SYNC\_REPLICA\_VERSIONS**

Column Name	Description
BOOKMARK_ID	Internal ID of the bookmark in the subscription.
REQUEST_ID	Internal ID of the publication request in the subscription.
VERS_TIME	Create time of the subscription.
PUBL_ID	ID of the subscribed publication.
MASTER_ID	ID of the master database from which the publication has been subscribed.
PARAM_CRC	Internal use only.
PARAM	Parameters of the subscription.
VERSION	Version number of subscribed publication in the master database.
LOCAL_VERSION	Version number of subscribed publication in the replica database.

Column Name	Description
PUBL_NAME	Name of the publication.
REPLY_ID	ID of the publication reply.

## D.2.33 SYS\_SYNC\_REPLICAS

This table contains the list of replica databases registered with the master.

**Table D.65. SYS\_SYNC\_REPLICAS**

Column Name	Description
NAME	Given name of the replica database.
ID	Internal ID of the replica database.
MASTER_NAME	N/A.
MASTER_CATALOG	Defines the catalog where the replica is registered
CONNECT	This contains the connect string (e.g. 'tcp MyWorkstation 1315') of the replica.

## D.2.34 SYS\_SYNC\_SAVED\_BLOB\_ARGS

If the user saves a transaction with a BLOB parameter at the replica, a reference to the BLOB is saved in the SYS\_SYNC\_SAVED\_BLOB\_ARGS table. The reference points to the SYS\_SYNC\_REPLICA\_STORED\_BLOB\_REFS table. The rows only exist until the sent message has been prepared.

**Table D.66. SYS\_SYNC\_SAVED\_BLOB\_ARGS**

Column Name	Description
MASTER	ID of the master database to which the parameters are sent.
TRX_ID	The transaction ID identifies the transaction.
ID	Internal ID of the user.
ARGNO	Number of the parameter.
ARG_VALUE	Value of the parameter in binary format.

## D.2.35 SYS\_SYNC\_SAVED\_STMTS

This table contains statements that have been saved in replica database for later propagation.



**Table D.67. SYS\_SYNC\_SAVED\_STMTS**

Column Name	Description
MASTER	Internal ID of the master database to which the statement will be propagated.
TRX_ID	Internal ID of the transaction to which the statement belongs.
ID	Sequence number of the statement within the transaction.
CLASS	Type of the constant.
STRING	The SQL statement as a string.
ARG_COUNT	Number of parameters bound to the statement.
ARG_TYPES	Types of parameters bound to the statement.
ARG_VALUES	Values of the parameters in binary format.
USER_ID	ID of the user who has saved the statement.
REQUEST_ID	N/A.
FLAGS	This indicates the error-handling mode (e.g. IGNORE_ERRORS, LOG_ERRORS, etc.).

## D.2.36 SYS\_SYNC\_TRX\_PROPERTIES

When you save transactions, you can assign properties for them. These properties can later be used to select transactions for propagation. The properties are saved in the SYS\_SYNC\_TRX\_PROPERTIES table.

**Table D.68. SYS\_SYNC\_TRX\_PROPERTIES**

Column Name	Description
TRX_ID	The transaction ID identifies the transaction.
NAME	The transaction property name (for example, COLOR).
VALUE_STR	The transaction property value (for example, RED).

## D.2.37 SYS\_SYNC\_USERMAPS

This table maps replica user ids to master users in the SYS\_SYNC\_USERS table.

**Table D.69. SYS\_SYNC\_USERMAPS**

Column Name	Description
REPLICA_UID	Replica user ID mapped to master user.
MASTER_ID	Master ID.
REPLICA_USERNAME	Replica user name.
MASTER_USERNAME	Master user name.
PASSW	Encrypted password for master user name.

## D.2.38 SYS\_SYNC\_USERS

This table contains a list of users that have access to the synchronization functions of the replica database. These functions include saving transactions and creating synchronization messages.

In a replica the data of this table is downloaded from the master in a message with the command:

```
MESSAGE unique-message-name APPEND SYNC_CONFIG  
[ 'sync-config-arg' ]
```

**Table D.70. SYS\_SYNC\_USERS**

Column Name	Description
MASTER_ID	Internal ID of the master database.
ID	Internal ID of the user.
NAME	User name.
PASSW	Encrypted password of the user.

## D.3 System Views

IBM solidDB supports views as specified in the X/Open SQL Standard.

### D.3.1 COLUMNS

The COLUMNS system view identifies the columns which are accessible to the current user.

**Table D.71. COLUMNS**

Column name	Data type	Description
TABLE_CATALOG	WVARCHAR	The name of the catalog containing TABLE_NAME.
TABLE_SCHEMA	WVARCHAR	The name of the schema containing TABLE_NAME.
TABLE_NAME	WVARCHAR	The name of the table or view.
COLUMN_NAME	WVARCHAR	The name of the column of the specified table or view.
DATA_TYPE	WVARCHAR	The data type of the column.
SQL_DATA_TYPE_NUM	SMALLINT	ODBC compliant data type number.
CHAR_MAX_LENGTH	INTEGER	Maximum length for a character data type column; for others NULL.
NUMERIC_PRECISION	INTEGER	The number of digits of mantissa precision of the column, if DATA_TYPE is approximate numeric data type, NUMERIC_PREC_RADIX indicates the units of measurement; for other numeric types contains the total number of decimal digits allowed in the column; for character data types NULL.
NUMERIC_PREC_RADIX	SMALLINT	The radix of numeric precision if DATA_TYPE is one of the approximate numeric data types; otherwise NULL.
NUMERIC_SCALE	SMALLINT	Total number of significant digits to the right of the decimal point; for INTEGER and SMALLINT 0; for others NULL.
NULLABLE	CHAR	If column is known to be not nullable 'NO'; otherwise 'YES'.
NULLABLE_ODBC	SMALLINT	ODBC, if column is known to be not nullable '0'; otherwise '1'.
REMARKS	LONG WVARCHAR	Reserved for future use.

## D.3.2 SERVER\_INFO

The SERVER\_INFO system view provides attributes of the current database system or server.

**Table D.72. SERVER\_INFO**

Column name	Data type	Description
SERVER_ATTRIBUTE	WVARCHAR	Identifies an attribute of the server.
ATTRIBUTE_VALUE	WVARCHAR	The value of the attribute.

## D.3.3 TABLES

The TABLES system view identifies the tables accessible to the current user.

**Table D.73. TABLES**

Column name	Data type	Description
TABLE_CATALOG	WVARCHAR	The name of the catalog containing TABLE_NAME.
TABLE_SCHEMA	WVARCHAR	The name of the schema containing TABLE_NAME.
TABLE_NAME	WVARCHAR	The name of the table or view.
TABLE_TYPE	WVARCHAR	The type of the table.
REMARKS	LONG WVARCHAR	Reserved for future use.

## D.3.4 USERS

The USERS system view identifies users and roles.

**Table D.74. USERS**

Column name	Data type	Description
ID	INTEGER	User or role identifier.
NAME	WVARCHAR	User or role name.
TYPE	WVARCHAR	User type, either USER or ROLE.
PRIV	INTEGER	Privilege information.
PRIORITY	INTEGER	Reserved for future use.
PRIVATE	INTEGER	Specifies whether user is private or public.

## D.4 Synchronization-Related Views

IBM solidDB provides four views that show information about synchronization messages between masters and replicas. One pair of views (SYNC\_FAILED\_MESSAGES and SYNC\_FAILED\_MASTER\_MESSAGES) shows failed messages. The other pair (SYNC\_ACTIVE\_MESSAGES and SYNC\_ACTIVE\_MASTER\_MESSAGES) shows active messages.

## D.4.1 SYNC\_FAILED\_MESSAGES

This table is on the master and holds information about messages received from the replica. It is possible to view all necessary information about failed messages using one simple view:

```
SELECT * FROM SYNC_FAILED_MESSAGES.
```

This returns the following columns:

**Table D.75. SYNC\_FAILED\_MESSAGES**

Column name	Data type	Description
REPLICA_NAME	WVARCHAR	Given node name of the replica from which the message was sent.
MESSAGE_NAME	WVARCHAR	Name of the message given by the user.
TRANSACTION_ID	BINARY	Internal ID of the replica transaction that has failed.
STATEMENT_ID	INTEGER	Sequence number of the statement within the transaction.
STATEMENT_STRING	WVARCHAR	SQL statement as a string.
ERROR_CODE	INTEGER	Code of the error that caused the termination of the message execution.
ERROR_MESSAGE	VARCHAR	Description of the error.

All users have access to this view; no particular privileges are required.

## D.4.2 SYNC\_FAILED\_MASTER\_MESSAGES

This table is on the replica and holds information about messages sent to the master. It is possible to view all necessary information about failed messages using one simple view:

```
SELECT * FROM SYNC_FAILED_MASTER_MESSAGES.
```

This returns the following columns:

**Table D.76. SYNC\_FAILED\_MASTER\_MESSAGES**

Column name	Data type	Description
MASTER_NAME	WVARCHAR	Given node name of the master.

Column name	Data type	Description
MESSAGE_NAME	WVARCHAR	Name of the message given by user.
ERROR_CODE	INTEGER	Code of the error that caused the termination of the message execution.
ERROR_MESSAGE	VARCHAR	Description of the error.

All users have access to this view; no particular privileges are required.

## D.4.3 SYNC\_ACTIVE\_MESSAGES

This table is on the master and holds information about messages received from the replica. This returns the following columns:

**Table D.77. SYNC\_ACTIVE\_MESSAGES**

Column name	Data type	Description
REPLICA_NAME	WVARCHAR	Given node name of the replica.
MESSAGE_NAME	WVARCHAR	Name of the message given by user.
MESSAGE STATE	VARCHAR	Current state of the message as a string. See details in system table SYS_SYNC_MASTER_MSGINFO.

All users have access to this view; no particular privileges are required.

## D.4.4 SYNC\_ACTIVE\_MASTER\_MESSAGES

This table is on the replica and holds information about messages sent to the master. It is possible to view all necessary information about failed messages using one simple view:

```
SELECT * FROM SYNC_FAILED_MASTER_MESSAGES.
```

This returns the following columns:

**Table D.78. SYNC\_ACTIVE\_MASTER\_MESSAGES**

Column name	Data type	Description
MASTER_NAME	WVARCHAR	Given node name of the master.
MESSAGE_NAME	WVARCHAR	Name of the message given by user.

#### D.4.4 SYNC\_ACTIVE\_MASTER\_MESSAGES

---

<b>Column name</b>	<b>Data type</b>	<b>Description</b>
MESSAGE STATE	VARCHAR	Current state of the message as a string. See details in system table SYS_SYNC_REPLICA_MSGINFO.

All users have access to this view; no particular privileges are required.

---



---

# Appendix E. System Stored Procedures

This chapter documents stored procedures that are provided with the IBM solidDB to help simplify tasks. These stored procedures are built into the server and can be thought of as a library for you to use.

## E.1 Synchronization-Related Stored Procedures

These system procedures simplify routine sync tasks. To maintain this ease of use, "unnecessary" error situations should be avoided.

To execute synchronization system procedures, you must have administrator or sync administrator access rights.

### E.1.1 SYNC\_SETUP\_CATALOG

```
CALL SYNC_SETUP_CATALOG (  
    catalog_name,    -- WVARCHAR  
    node_name,      -- WVARCHAR  
    is_master,      -- INTEGER  
    is_replica      -- INTEGER  
)
```

EXECUTES ON: master or replica.

The SYNC\_SETUP\_CATALOG() procedure creates a catalog, assigns it a node name, and sets the role of the catalog to be master, replica, or both.

If the *catalog\_name* parameter is NULL, then the current catalog is assigned the specified node name and role(s).

For *is\_master* and *is\_replica*, a value of 0 means "no"; any other value means "yes". At least one of these should be non-zero, of course. Note that because a single catalog can be both a replica and a master, it is legal to set both *is\_master* and *is\_replica* to non-zero values.

**Table E.1. SYNC\_SETUP\_CATALOG Error Codes**

RC	Text	Description
13047	No privilege for operation	
13110	NULL not allowed	Only the catalog name can be NULL; all other parameters must be non-NULL.
13133	Not a valid license for this product.	
25031	Transaction is active, operation failed.	The user has made some changes that have not yet been committed.
25052	Failed to set node name to <i>node_name</i> .	The <i>node_name</i> may be invalid.
25059	After registration nodename cannot be changed.	Catalog has a name already and has one or more replicas.

## E.1.2 SYNC\_REGISTER\_REPLICA

```
CALL SYNC_REGISTER_REPLICA (
    replica_node_name,      -- WVARCHAR
    replica_catalog_name,  -- WVARCHAR
    master_network_name,   -- WVARCHAR
    master_node_name,      -- WVARCHAR
    user_id,               -- WVARCHAR
    password                -- WVARCHAR
)
```

EXECUTES ON: replica.

The SYNC\_REGISTER\_REPLICA() system procedure creates a new catalog and registers the replica with the specified master. User must have Administrator or Synchronization Administrator access rights.

The *master\_network\_name* is the connect string of the master database server.

If the specified catalog does not exist, then it is created automatically.

If the specified replica catalog name is NULL, then the current catalog is used. Also, the master nodename can be NULL. No other parameter may be NULL.

If registration fails, both master and replica end are reset back to their original status. If any of the parameters have illegal values, then an error is returned.

If there are any open transactions that have modified data, then this function returns an error.

This system procedure does not return a resultset.

**Table E.2. SYNC\_REGISTER\_REPLICA Error Codes**

RC	Text	Description
13047	No privilege for operation	
13110	NULL not allowed	Only the catalog name and master node name can be NULL; all other parameters must be non-NULL.
13133	Not a valid license for this product.	
21xxx	Communication error	Was not able to connect to master. For more details about 21xxx errors, see the appendix of <i>IBM solidDB Administration Guide</i> titled "Error Codes".
25005	Message is already active.	
25031	Transaction is active, operation failed.	The user has made some changes that have not yet been committed.
25035	Message is in use.	
25051	Unfinished messages found.	
25052	Failed to set node name to <i>node_name</i> .	The <i>node_name</i> may be invalid.
25056	Autocommit not allowed.	You must run this stored procedure with autocommit off.
25057	The replica database has already been registered to a master database.	
25059	After registration nodename cannot be changed.	

## E.1.3 SYNC\_UNREGISTER\_REPLICA

```
CALL SYNC_UNREGISTER_REPLICA (
    replica_catalog_name,  -- WVARCHAR
    drop_catalog,         -- INTEGER
```

```

    force                -- INTEGER
)

```

EXECUTES ON: replica.

The SYNC\_UNREGISTER\_REPLICA() system procedure unregisters the specified replica catalog from the master and optionally drops the replica catalog if the *drop\_catalog* parameter has nonzero value. Any possibly hanging messages for this replica are deleted in both ends of the system. User must have Administrator or Synchronization Administrator access rights.

If the replica catalog name is NULL, then the current catalog is used. If force is non-zero, then the master accepts unregistration even if messages for this replica exist in the master. In that case, those messages are deleted.

If the user has any uncommitted changes (i.e. open transactions), then the call will fail with an error.

This system procedure does not return a resultset.

**Table E.3. SYNC\_UNREGISTER\_REPLICA Error Codes**

RC	Text	Description
13047	No privilege for operation	
13110	NULL not allowed	Catalog name cannot be NULL if drop_catalog is non-zero.
13133	Not a valid license for this product.	
21xxx	Communication error	Was not able to connect to master. For more details about 21xxx errors, see the appendix of <i>IBM solidDB Administration Guide</i> titled "Error Codes".
25005	Message is already active.	
25019	Database is not a replica database.	
25020	Database is not a master database.	
25023	Replica not registered.	
25031	Transaction is active, operation failed.	The user has made some changes that have not yet been committed.
25035	Message is in use.	
25051	Unfinished messages found.	

RC	Text	Description
25056	Autocommit not allowed.	You must run this stored procedure with autocommit off.
25079		
25093		

## E.1.4 SYNC\_REGISTER\_PUBLICATION

```
CALL SYNC_REGISTER_PUBLICATION (
    replica_catalog_name,    -- WVARCHAR
    publication_name        -- WVARCHAR
)
```

EXECUTES ON: replica.

The SYNC\_REGISTER\_PUBLICATION() system procedure registers a publication from the master database.

If the replica catalog name is NULL, then the current catalog is used.

If the user has uncommitted changes, then the call will fail with an error.

This system procedure does not return a resultset.

**Table E.4. SYNC\_REGISTER\_PUBLICATION Error Codes**

RC	Text	Description
13047	No privilege for operation	
13110	NULL not allowed	Only the catalog name can be NULL; all other parameters must be non-NULL.
13133	Not a valid license for this product.	
21xxx	Communication error	Was not able to connect to master. For more details about 21xxx errors, see the appendix of <i>IBM solidDB Administration Guide</i> titled "Error Codes".
25005	Message is already active.	
25010	Publication not found	
25019	Database is not a replica database	

RC	Text	Description
25020	Database is not a master database.	
25023	Replica not registered.	
25035	Message is in use.	
25056	Autocommit not allowed.	You must run this stored procedure with autocommit off.
25072	Already registered to publication.	

## E.1.5 SYNC\_UNREGISTER\_PUBLICATION

```
CALL SYNC_UNREGISTER_PUBLICATION (
    replica_catalog_name,    -- WVARCHAR
    publication_name,       -- WVARCHAR
    drop_data                -- INTEGER
)
```

EXECUTES ON: replica.

The SYNC\_UNREGISTER\_PUBLICATION() system procedure unregisters a publication. If the *drop\_data* flag is set to a non-zero value, then all subscriptions to the publication are automatically dropped.

If the replica catalog name is NULL, then the current catalog is used.

If the user has uncommitted changes, then the call will fail with an error.

This system procedure does not return a resultset.

**Table E.5. SYNC\_UNREGISTER\_PUBLICATION Error Codes**

RC	Text	Description
13047	No privilege for operation	
13110	NULL not allowed	Only the catalog name can be NULL; all other parameters must be non-NULL.
13133	Not a valid license for this product.	
21xxx	Communication error	Was not able to connect to master. For more details about 21xxx errors, see the appendix of <i>IBM</i>

RC	Text	Description
		<i>solidDB Administration Guide</i> titled "Error Codes".
25005	Message is already active.	
25010	Publication not found.	
25019	Database is not a replica database.	
25020	Database is not a master database.	
25023	Replica not registered.	
25031	Transaction is active, operation failed.	User has made some changes that are not yet committed.
25035	Message is in use.	
25056	Autocommit not allowed.	You must run this stored procedure with autocommit off.
25071	Not registered to publication.	

## E.1.6 SYNC\_SHOW\_SUBSCRIPTIONS

```
CREATE PROCEDURE SYNC_SHOW_SUBSCRIPTIONS (
publication_name      -- WVARCHAR
)
```

EXECUTES ON: replica.

Often it is useful for the application to know which subscriptions (i.e. publication name and parameters as string representation) of a publication are active in replica or master database(s). This functionality is available in both master and replica catalogs. Use this function (SYNC\_SHOW\_SUBSCRIPTIONS) in the replica catalog. Use the function SYNC\_SHOW\_REPLICA\_SUBSCRIPTIONS in the master catalog.

The resultset of this procedure call is:

**Table E.6. CREATE PROCEDURE SYNC\_SHOW\_SUBSCRIPTIONS Result Set**

Column Name	Data Type	Description
SUBSCRIPTION	WVARCHAR	Publication name and parameters as a string
SUBSCRIPTION_TIME	TIMESTAMP	Time of last subscription.

**Table E.7. SYNC\_SHOW\_SUBSCRIPTIONS Error Codes**

RC	Text	Description
13047	No privilege for operation	
13133	Not a valid license for this product.	
25009	Replica not found.	
25010	Publication not found	
25019	Database is not a replica database	
25020	Database is not a master database.	
25023	Replica not registered.	
25071	Not registered to publication.	

*See Also:*

Section E.1.7, “SYNC\_SHOW\_REPLICA\_SUBSCRIPTIONS”.

## E.1.7 SYNC\_SHOW\_REPLICA\_SUBSCRIPTIONS

*Syntax in master:*

```
CREATE PROCEDURE SYNC_SHOW_REPLICA_SUBSCRIPTIONS (  
    replica_name,          -- WVARCHAR  
    publication_name      -- WVARCHAR  
)
```

EXECUTES ON: master.

Often it is useful for the application to know which subscriptions (i.e. publication name and parameters as string representation) of a publication are active in a specified replica database(s). This functionality is available in both master and replica catalogs.

If the publication name is NULL, then subscriptions to all publications are listed.

The resultset of this procedure call is:



**Table E.8. SYNC\_SHOW\_REPLICA\_SUBSCRIPTIONS Result Set**

Column Name	Data Type	Description
REPLICA_NAME	WVARCHAR	Replica name.
SUBSCRIPTION	WVARCHAR	Publication name and parameters as a string
SUBSCRIPTION_TIME	TIMESTAMP	Time of last subscription.

**Table E.9. SYNC\_SHOW\_REPLICA\_SUBSCRIPTIONS Error Codes**

RC	Text	Description
13047	No privilege for operation	
13133	Not a valid license for this product.	
25009	Replica not found.	
25010	Publication not found	
25019	Database is not a replica database	
25020	Database is not a master database.	
25023	Replica not registered.	
25071	Not registered to publication.	

*See Also:*

Section E.1.6, “SYNC\_SHOW\_SUBSCRIPTIONS”.

## E.1.8 SYNC\_DELETE\_MESSAGES

```
CALL SYNC_DELETE_MESSAGES (  
    replica_catalog_name,    -- WVARCHAR  
)
```

EXECUTES ON: replica.

If the replica catalog name is NULL, then the current catalog is used.

If a replica application creates lots of messages and does not check / handle errors properly, then there may be lots of messages hanging. Sometimes, the right way to recover is to delete all of them, regardless of the

state of the messages, in both master and replica ends. This procedure deletes the messages in the replica database.

This procedure does not return a resultset.

**Table E.10. SYNC\_DELETE\_MESSAGES Error Codes**

RC	Text	Description
13047	No privilege for operation	
13133	Not a valid license for this product.	
25005	Message is already active.	
25009	Replica not found.	
25019	Database is not a replica database	
25020	Database is not a master database.	
25035	Message is in use.	

*See Also:*

Section E.1.9, “SYNC\_DELETE\_REPLICA\_MESSAGES”.

## E.1.9 SYNC\_DELETE\_REPLICA\_MESSAGES

```
CALL SYNC_DELETE_REPLICA_MESSAGES (  
    master_catalog_name    -- WVARCHAR,  
    replica_name           -- WVARCHAR  
)
```

EXECUTES ON: master.

If a replica application creates lots of messages and does not check / handle errors properly, then there are lots of messages hanging. Sometimes, the right way to recover is to delete all of them, regardless of the state of the messages, in both master and replica ends. This procedure deletes the messages of a specified replica in the master database. The `master_catalog_name` parameter specifies the catalog in the master database from which the messages of the specified replica are searched. If the `master_catalog_name` is set to `NULL`, the current catalog is used.

This procedure does not return a resultset.

**Table E.11. SYNC\_DELETE\_REPLICA\_MESSAGES Error Codes**

RC	Text	Description
13047	No privilege for operation	
13133	Not a valid license for this product.	
25005	Message is already active.	
25009	Replica not found.	
25019	Database is not a replica database	
25020	Database is not a master database.	
25035	Message is in use.	

*See Also:*

Section E.1.8, “SYNC\_DELETE\_MESSAGES”.

## E.2 Miscellaneous Stored Procedures

### E.2.1 SYS\_GETBACKGROUNDJOB\_INFO

```
CREATE PROCEDURE SYS_GETBACKGROUNDJOB_INFO(
    jobid INTEGER)
RETURNS (
    ID INTEGER,
    STMT WVARCHAR,
    USER_ID INTEGER,
    ERROR_CODE INTEGER,
    ERROR_TEXT INTEGER)
```

The user can retrieve information from the table `SYS_BACKGROUNDJOB_INFO` using either an SQL `SELECT` statement or by calling the system stored procedure `SYS_GETBACKGROUNDJOB_INFO`. The procedure `SYS_GETBACKGROUNDJOB_INFO` returns the row that matches the given jobid. The jobid is the job ID of the `START AFTER COMMIT` statement that was executed. (The job ID is returned by the server when the `START AFTER COMMIT` statement is executed.)

---

---

# Appendix F. System Events

This chapter documents System Events. These events are provided with the IBM solidDB to allow programs to be notified when certain actions occur. You can use these events to monitor the progress of activities such as synchronization between master and replica databases.

These events follow most of the same rules as any other events. For information about events in general, see

- Section B.12, “CREATE EVENT”
- Section B.12, “CREATE EVENT”, which describes how to post events and wait on events.
- Chapter 4, *Stored Procedures, Events, Triggers, and Sequences*, which discusses events extensively.

Because these events are pre-defined, you do not create them. Furthermore, you should not post any system event. You should only wait on system events.

Many, although not all, system events have the same five parameters:

- *ename*: The event name.
- *postsrvtime*: The time that the server posted the event.
- *uid*: The user ID (if applicable).
- *numdatainfo*: Miscellaneous numeric data — the exact meaning depends upon the event. For example, the event `SYS_EVENT_BACKUP` is posted both when a backup is started and when a backup is completed. The value in the *numdatainfo* parameter indicates which case applies — i.e. whether the backup has just started or has just completed. This parameter may be NULL if there is no numeric data.
- *textdata*: Miscellaneous text data — the exact meaning depends upon the event. This parameter may be NULL if there is no numeric data.

This appendix contains the following tables:

1. Miscellaneous Events
2. Errors that cause the `SYS_EVENT_ERROR` event to be posted.
3. Conditions or warnings that cause the `SYS_EVENT_MESSAGES` event to be posted.

## F.1 Miscellaneous Events

The following events are mostly related to the server's internal scheduling and "housekeeping". For example, there are events related to backups, checkpoints, and merges. Although users do not post these events, in many cases users may indirectly cause events, for example when requesting a backup, or when turning on "Maintenance Mode". You can monitor these events if you want.

**Table F.1. Miscellaneous Events**

EVENT NAME	EVENT DESCRIPTION	PARAMETERS
SYS_EVENT_BACKUP	<p>The system has started or completed a backup operation. The "state" parameter (NUMDATAINFO) indicates:</p> <p>0: backup completed.</p> <p>1: backup started.</p> <p>Note that the server also posts a second event (SYS_EVENT_MESSAGES) when it starts or completes a backup.</p>	ENAME WVARCHAR, POSTSRVTIME TIMESTAMP, UID INTEGER, NUMDATAINFO INTEGER, TEXTDATA WVARCHAR
SYS_EVENT_BACKUPREQ	<p>A backup operation has been requested (but has not yet started).</p> <p>If the user application's callback function returns non-zero, then backup is not performed.</p> <p>This event can be caught by the user <i>only</i> if the user is using the linked library access.</p> <p>None of the parameters are used.</p>	ENAME WVARCHAR, POSTSRVTIME TIMESTAMP, UID INTEGER, NUMDATAINFO INTEGER, TEXTDATA WVARCHAR
SYS_EVENT_CHECKPOINT	<p>The system has started or completed a checkpoint operation.</p> <p>If the system started a checkpoint, then the "state" parameter (NUM-</p>	ENAME WVARCHAR, POSTSRVTIME TIMESTAMP, UID INTEGER,

F.1 Miscellaneous Events

EVENT NAME	EVENT DESCRIPTION	PARAMETERS
	<p>DATAINFO) is 1, and the message (TEXTDATA) parameter is "started".</p> <p>If the system completed a checkpoint, then the "state" parameter (NUMDATAINFO) is 0, and the message (TEXTDATA) parameter is "completed".</p>	<p>NUMDATAINFO INTEGER, TEXTDATA WVARCHAR</p>
SYS_EVENT_CHECKPOINTREQ	<p>A checkpoint operation has been requested (but has not yet started). Checkpoints are typically executed each time a certain number of log writes has completed.</p> <p>If the user application's callback function returns non-zero, then the merge is not performed.</p> <p>This event can be caught by the user <i>only</i> if the user is using the linked library access.</p> <p>None of the parameters are used.</p>	<p>ENAME WVARCHAR, POSTSRVTIME TIMESTAMP, UID INTEGER, NUMDATAINFO INTEGER, TEXTDATA WVARCHAR</p>
SYS_EVENT_ERROR	<p>Some type of server error has occurred. The message parameter (TEXTDATA) contains the error text. See Section F.2, "Errors that Cause SYS_EVENT_ERROR" for a list of server errors that can cause this event to be posted.</p>	<p>ENAME WVARCHAR, POSTSRVTIME TIMESTAMP, UID INTEGER, NUMDATAINFO INTEGER, TEXTDATA WVARCHAR</p>
SYS_EVENT_IDLE	<p>The system is idle. (Note that some tasks have a priority of "idle" and are only run when the system is not running any other tasks. Because very low priority tasks may be running in an "idle" system, the system</p>	<p>ENAME WVARCHAR, POSTSRVTIME TIMESTAMP, UID INTEGER, NUMDATAINFO INTEGER,</p>

F.1 Miscellaneous Events

EVENT NAME	EVENT DESCRIPTION	PARAMETERS
	<p>is not necessarily truly idle in the sense of not doing anything.)</p> <p>This event can be caught by the user <i>only</i> if the user is using the linked library access.</p> <p>None of the parameters are used.</p>	<p>TEXTDATA WVARCHAR</p>
<p>SYS_EVENT_IMDB_MEMORY</p>	<p>The system has encountered an event related to in-memory database memory limits.</p> <p>The <i>NUMDATAINFO</i> parameter indicates the current memory allocation in Kilobytes.</p> <p>The <i>TEXTDATA</i> parameter can have one of the following values:</p> <ul style="list-style-type: none"> <li>• <i>IMDB_LIMIT_ABOVE</i> - The amount of available virtual memory is above the limit specified by using the <i>Imdb-MemoryLimit</i> parameter</li> <li>• <i>IMDB_LIMIT_BELOW</i> - The amount of available virtual memory is below the limit specified by using the <i>Imdb-MemoryLimit</i> parameter</li> <li>• <i>IMDB_LOW_LEVEL_ABOVE</i> - The amount of available virtual memory is above the limit specified by using the <i>Imdb-MemoryLowPercentage</i> parameter</li> <li>• <i>IMDB_LOW_LEVEL_BELOW</i> - The amount of available virtual memory is below the limit</li> </ul>	<p>ENAME WVARCHAR,</p> <p>POSTSRVTIME TIMESTAMP,</p> <p>UID INTEGER,</p> <p>NUMDATAINFO INTEGER,</p> <p>TEXTDATA WVARCHAR</p>



EVENT NAME	EVENT DESCRIPTION	PARAMETERS
	<p>specified by using the <i>Imdb-MemoryLowPercentage</i> parameter</p> <ul style="list-style-type: none"> <li>• <i>IMDB_WARN-ING_LEVEL_ABOVE</i> - The amount of available virtual memory is above the limit specified by using the <i>Imdb-MemoryWarningPercentage</i> parameter</li> <li>• <i>IMDB_WARNING_LEVEL_BELOW</i> - The amount of available virtual memory is below the limit specified by using the <i>ImdbMemoryWarningPercentage</i> parameter</li> </ul>	
SYS_EVENT_ILL_LOGIN	<p>There has been an illegal login attempt. The username (TEXTDATA) and userid (NUMDATAINFO) indicate the user who tried to log in.</p>	<p>ENAME WVARCHAR, POSTSRVTIME TIMESTAMP, UID INTEGER, NUMDATAINFO INTEGER, TEXTDATA WVARCHAR</p>
SYNC_MAINTENANCEMODE_BEGIN	<p>When the sync mode changes from NORMAL to MAINTENANCE, the server will send this system event. The node_name is the name of the node in which maintenance mode started. (Remember that a single IBM solidDB server can have multiple "nodes" (catalogs).) For more details about sync mode, see Section B.75.11.3, "SET SYNC MODE".</p>	<p>node_name WVARCHAR.</p>

F.1 Miscellaneous Events

EVENT NAME	EVENT DESCRIPTION	PARAMETERS
SYNC_MAINTENANCEMODE_END	When the sync mode changes from MAINTENANCE to NORMAL, the server will send this system event. The node_name is the name of the node in which maintenance mode started. (Remember that a single IBM solidDB server can have multiple "nodes" (catalogs).) For more details about sync mode, see Section B.75.11.3, "SET SYNC MODE".	node_name WVARCHAR
SYS_EVENT_MERGE	An event associated with the "merge" operation (merging data from the Bonsai Tree to the main storage tree) has occurred. The parameter STATE (NUMDATAINFO) gives more details:  0: stop the merge  1: start the merge  2: merge progressing  3: merge accelerated.	ENAME WVARCHAR,  POSTSRVTIME TIMESTAMP,  UID INTEGER,  NUMDATAINFO INTEGER,  TEXTDATA WVARCHAR
SYS_EVENT_MERGREQ	A merge operation has been requested (but has not yet started).  If the user application's callback function returns non-zero, then the merge is not performed.  This event can be caught by the user <i>only</i> if the user is using the linked library access.  None of the parameters are used.	ENAME WVARCHAR,  POSTSRVTIME TIMESTAMP,  UID INTEGER,  NUMDATAINFO INTEGER,  TEXTDATA WVARCHAR
SYS_EVENT_MESSAGES	This event is posted when the server has a message (error message or warning message) to log to	ENAME WVARCHAR,  POSTSRVTIME TIMESTAMP,

EVENT NAME	EVENT DESCRIPTION	PARAMETERS
	<p>solerror.out or solmsg.out. In this case, the TEXTDATA contains the message text and NUMDATAINFO the code. If the message to be written is an error, then <i>both</i> SYS_EVENT_ERROR and SYS_EVENT_MESSAGES will be posted. If the message is only a warning, then only SYS_EVENT_MESSAGES is posted. For a list of the warnings that can cause SYS_EVENT_MESSAGES, see Section F.3, "Conditions or Warnings that Cause SYS_EVENT_MESSAGES".</p>	<p>UID INTEGER, NUMDATAINFO INTEGER, MESSAGE WVARCHAR</p>
SYS_EVENT_NOTIFY	<p>Event sent with the admin command 'notify'.</p>	<p>ENAME WVARCHAR, POSTSRVTIME TIMESTAMP, UID INTEGER, NUMDATAINFO INTEGER, TEXTDATA WVARCHAR</p>
SYS_EVENT_PARAMETER	<p>This event is posted if a configuration parameter is changed with the command</p> <p><b>ADMIN COMMAND 'parameter..';</b></p> <p>The parameter MESSAGE (TEXTDATA) contains the section name (e.g. <i>SRV</i>) and the parameter name.</p>	<p>ENAME WVARCHAR, POSTSRVTIME TIMESTAMP, UID INTEGER, NUMDATAINFO INTEGER, TEXTDATA WVARCHAR</p>
SYS_EVENT_PROCESS_MEMORY	<p>The system has encountered an event related to process size memory limits.</p>	<p>ENAME WVARCHAR, POSTSRVTIME TIMESTAMP,</p>

EVENT NAME	EVENT DESCRIPTION	PARAMETERS
	<p>The <i>NUMDATAINFO</i> parameter indicates the current memory allocation in Kilobytes.</p> <p>The <i>TEXTDATA</i> parameter can have one of the following values:</p> <ul style="list-style-type: none"> <li>• <i>PROCESS_LIMIT_ABOVE</i> - The amount of available virtual memory is above the limit specified by using the <i>ProcessMemoryLimit</i> parameter</li> <li>• <i>PROCESS_LIMIT_BELOW</i> - The amount of available virtual memory is below the limit specified by using the <i>ProcessMemoryLimit</i> parameter</li> <li>• <i>PROCESS_LOW_LEVEL_ABOVE</i> - The amount of available virtual memory is above the limit specified by using the <i>ProcessMemoryLowPercentage</i> parameter</li> <li>• <i>PROCESS_LOW_LEVEL_BELOW</i> - The amount of available virtual memory is below the limit specified by using the <i>ProcessMemoryLowPercentage</i> parameter</li> <li>• <i>PROCESS_WARNING_LEVEL_ABOVE</i> - The amount of available virtual memory is above the limit specified by using the <i>ProcessMemoryWarningPercentage</i> parameter</li> </ul>	<p>UID INTEGER,            NUMDATAINFO INTEGER,            TEXTDATA WVARCHAR</p>

F.1 Miscellaneous Events

EVENT NAME	EVENT DESCRIPTION	PARAMETERS
	<ul style="list-style-type: none"> <li>PROCESS_WARNING_LEVEL_BELOW- The amount of available virtual memory is below the limit specified by using the <i>ProcessMemoryWarningPercentage</i> parameter</li> </ul>	
SYS_EVENT_ROWS2MERGE	This event indicates that there are rows that need to be merged from the Bonsai Tree to the main storage tree. The rows parameter (NUMDATAINFO) indicates the number of non-merged rows in the Bonsai Tree.	ENAME WVARCHAR, POSTSRVTIME TIMESTAMP, UID INTEGER, NUMDATAINFO INTEGER, TEXTDATA WVARCHAR
SYS_EVENT_SACFAILED	This event is posted when a START AFTER COMMIT (SAC) fails. The application can wait for this event and use the job ID (which is in the NUMDATAINFO field) to retrieve the error message from the system table SYS_BACKGROUNDJOB_INFO. (The job ID in NUMDATAINFO matches the job ID that is returned when the START AFTER COMMIT statement is executed.)	ENAME WVARCHAR, POSTSRVTIME TIMESTAMP, UID INTEGER, NUMDATAINFO INTEGER, TEXTDATA WVARCHAR
SYS_EVENT_SHUTDOWNREQ	<p>A shutdown request has been received. If the user application's callback function returns non-zero, then shutdown is not performed.</p> <p>This event can be caught by the user <i>only</i> if the user is using the linked library access.</p> <p>None of the parameters are used.</p>	ENAME WVARCHAR, POSTSRVTIME TIMESTAMP, UID INTEGER, NUMDATAINFO INTEGER, TEXTDATA WVARCHAR

F.1 Miscellaneous Events

EVENT NAME	EVENT DESCRIPTION	PARAMETERS
SYS_EVENT_STATE_MONITOR	<p>This event is posted when monitoring settings are changed.</p> <p>State (NUMDATAINFO) is one of the following:</p> <p>0: monitoring off.</p> <p>1: monitoring on.</p> <p>UID is the user ID of the user for whom monitoring was turned on or off.</p>	<p>ENAME WVARCHAR, POSTSRVTIME TIMESTAMP, UID INTEGER, NUMDATAINFO INTEGER, TEXTDATA WVARCHAR</p>
SYS_EVENT_STATE_OPEN	<p>This event is posted when the "state" of the database is changed. The parameter STATE (NUMDATAINFO) indicates the new state:</p> <p>0: Closed. No new connections allowed.</p> <p>1: Opened: New connections allowed.</p>	<p>ENAME WVARCHAR, POSTSRVTIME TIMESTAMP, UID INTEGER, NUMDATAINFO INTEGER, TEXTDATA WVARCHAR</p>
SYS_EVENT_STATE_SHUTDOWN	<p>This event is posted when a server shutdown is started. Note that the NUMDATAINFO and TEXTDATA parameters have no useful information.</p>	<p>ENAME WVARCHAR, POSTSRVTIME TIMESTAMP, UID INTEGER, NUMDATAINFO INTEGER, TEXTDATA WVARCHAR</p>
SYS_EVENT_STATE_TRACE	<p>Server trace is turned on or off with</p> <p>ADMIN COMMAND 'trace';</p> <p>The parameter STATE (NUMDATAINFO) indicates the new trace state:</p>	<p>ENAME WVARCHAR, POSTSRVTIME TIMESTAMP, UID INTEGER, NUMDATAINFO INTEGER,</p>

## F.2 Errors that Cause SYS\_EVENT\_ERROR

EVENT NAME	EVENT DESCRIPTION	PARAMETERS
	0: tracing off. 1: tracing on.	TEXTDATA WVARCHAR
SYS_EVENT_TMCMD	This event is posted when an "AT" command (i.e. a timed command) is executed. The message parameter (TEXTDATA) contains the command.	ENAME WVARCHAR, POSTSRVTIME TIMESTAMP, UID INTEGER, NUMDATAINFO INTEGER, TEXTDATA WVARCHAR
SYS_EVENT_TRX_TIMEOUT	This event is currently not used.	ENAME WVARCHAR, POSTSRVTIME TIMESTAMP, UID INTEGER, NUMDATAINFO INTEGER, TEXTDATA WVARCHAR
SYS_EVENT_USERS	The parameter REASON (NUMDATAINFO) contains the reason for the event:  0: User connected.  1: User disconnected.  2: User disconnected abnormally.  4: User disconnected because of timeout.	ENAME WVARCHAR, POSTSRVTIME TIMESTAMP, UID INTEGER, NUMDATAINFO INTEGER, TEXTDATA WVARCHAR

## F.2 Errors that Cause SYS\_EVENT\_ERROR

The table below shows the errors that can cause the server to post the event SYS\_EVENT\_ERROR.

The numbers in the "Error Code" column match the error code numbers in the appendix "Error Codes" in the *IBM solidDB Administration Guide*. These values get passed in the NUMDATAINFO event parameter.

**Table F.2. Errors that Cause SYS\_EVENT\_ERROR**

<b>Error code</b>	<b>Error description</b>
30104	Shutdown aborted; denied by user callback
30208	Merge not started; denied by user callback
30284	Checkpoint not started; denied by user callback
30302	Backup start failed. Shutdown is in progress
30302	Backup start failed. Backup is already active
30303	Backup aborted
30304	Backup failed. <error description>
30305	Backup not started; denied by user callback
30306	Backup not started; Backup is not supported on diskless server.
30307	Backup not started, index check failed. Errors written to file ssdebug.log.
30360	AT command failed. <reason>
30403	Log file write failure.
30454	Failed to save configuration file <file name>
30573	Network backup failed. <reason>
30640	<Server RPC error message>

## F.3 Conditions or Warnings that Cause SYS\_EVENT\_MESSAGES

The table below shows the warning messages that can cause the server to post the event SYS\_EVENT\_MESSAGES.

**Table F.3. Warnings that Cause SYS\_EVENT\_MESSAGES**

<b>Error code</b>	<b>Error description</b>
30010	User '<username>' failed to connect, version mismatch. Client version <version>, server version <version>.
30011	User '<username>' failed to connect, collation version mismatch.
30012	User '<username>' failed to connect, there are too many connected clients.
30020	Server is in fatal state, no new connections are allowed



Error code	Error description
30282	Checkpoint creation not started because shutdown is in progress.
30283	Checkpoint creation not started because it's disabled.
30300	Backup completed successfully.  Note that the server also posts a second event (SYS_EVENT_BACKUP) when it starts or completes a backup.
30301	Backup started to <directory path>.  Note that the server also posts a second event (SYS_EVENT_BACKUP) when it starts or completes a backup.
30359	Server noticed time inconsistency during at-command execution. If the system time has been changed, please restart server.
30361	Illegal at command <command> ignored.
30362	Illegal immediate at command <command> ignored.
30405	Unable to open message log file 'file name'
30800	Unable to reserve requested <number> memory blocks for external sorter.  Only <number> memory blocks were available. SQL: <sql statement>
30801	Unable to reserve requested <number> memory blocks for external sorter.  Only <number> memory blocks were available.

## F.4 HotStandby Events

For a description of events related to HotStandby, see *IBM solidDB High Availability User Guide*.

## F.5 Advanced Replication Synchronization Events

For a description of events related to advanced replication, see *IBM solidDB Advanced Replication Guide*.

---

---

# Glossary

This glossary gives you a description of the terminology used in this guide.

## A

### Application Programming Interface (API)

An API is a way for one piece of code to use another piece of code. In the case of IBM solidDB, client programs use an API (such as ODBC or JDBC) to communicate with the server and make requests of it. For example, your client application program may compose an SQL query and then use an ODBC or JDBC function call to send that query to the server for processing.

## C

### Client/server computing

Client/server computing divides a large piece of software into modules that need not all be executed within the same memory space nor on the same processor. The calling module becomes the 'client' that requests services, and the called module becomes the 'server' that provides services. Client and server processes exchange information by sending messages through a computer network. They may run on different hardware and software platforms as appropriate for their special functions.

### Communication protocol

A communication protocol is a set of rules and conventions used in the communication between servers and clients. The server and client have to use the same communication protocol in order to establish a connection. TCP/IP is an example of a common communication protocol.

## D

### Database administrator

The database administrator is a person responsible for tasks such as:

- managing users, tables, and indices
- backing up data
- allocating disk space for the database files

---

## Database Management System (DBMS)

A DBMS is a system that stores information in and retrieves information from a database. A DBMS typically consists of a database server, administration utilities, an application interface, and development tools.

## Database procedures (DBMS)

See stored procedures.

# I

## Index

An index of records has an entry for each key field (for example, employee name, identification number, etc.) and the location of the record. Indexes are used to speed up access to tables. IBM solidDB uses indexes to access the rows in a table directly. Without indexes, the engine would have to search the whole contents of a table to find the desired row. A single table can have more than one index; however, adding indexes does slow down write operations, such as inserts, deletes, and updates on that table. There are two kinds of indexes: non unique indexes and unique indexes. A unique index is an index where all key values are unique.

# O

## Open Database Connectivity (ODBC)

ODBC is a programming interface standard for SQL database programs. IBM solidDB offers a native ODBC programming interface.

## Optimizer Hints

Optimizer hints (which are an extension of SQL) are directives specified through embedded pseudo comments within query statements. The Optimizer detects these directives or hints and bases its query execution plan accordingly. Optimizer hints allow applications to be optimized under various conditions to the data, query type, and the database. They not only provide solutions to performance problems occasionally encountered with queries, but shift control of response times from the system to the user.

# R

## Relational database management system (RDBMS)

IBM solidDB is an RDBMS, which stores and retrieves information that is organized into two-dimensional tables. This name derives from the relational theory that formalizes the data manipulation requests as set operations and allows mathematical analysis of these sets. RDBMSs typically support the SQL language for data manipulation requests.

---

## S

### SQL Access Group's Call Level Interface (SAG CLI)

SAG CLI is a programming interface standard that defines the functions that are used to submit dynamic SQL clauses to a database server for execution. The ODBC interface is also based on SAG CLI. The IBM solidDB SQL API conforms to the SAG CLI standard.

### Stored procedures

Stored procedures allow programmers to split the application logic between the client and the server. These procedures are stored in the database, and they accept parameters in the activation call from the client application. This arrangement is used by intelligent transactions that are implemented with calls to stored procedures.

### Structured Query Language (SQL)

SQL is a standardized query language designed for handling database requests and administration. The SQL syntax used in IBM solidDB is based on the ANSI X3H2-1989 Level 2 standard including important ANSI X3H2-1992 (SQL-92) extensions. Refer to Appendix B, *IBM solidDB SQL Syntax*, for a more formal definition of the syntax.

## T

### Triggers

Triggers are pieces of logic that IBM solidDB automatically executes when a user attempts to change the data in a table. When a user modifies data within the table, the trigger that corresponds to the command (such as insert, delete, or update) is activated.

---

---

# Index

## Symbols

% , 431  
\* (asterisk), 420  
+ (plus), 420, 423  
- (minus), 420  
/ (slash), 420  
< (less than), 418  
<= (less than or equal to), 418  
<> (not equal to), 418  
= (equal to), 418  
> (greater than), 418  
>= (greater than or equal to), 418  
\_ (underscore), 431  
|| (concatenation operator), 423

## A

ABS, 424  
ACCESS RIGHTS, 58  
access rights  
    publications, 328, 376  
    registration user, 400  
ACOS, 424  
ADD CONSTRAINT, 156  
ADMIN COMMAND  
    abort, 221  
    assertexit, 221  
    backgroundjob, 221  
    backup, 221  
    backuplist, 222  
    checkpointing, 222  
    cleanbgjobinfo, 222  
    close, 222  
    commands, 219  
    describe, 222  
    errorcode, 223  
    errorexit, 223  
    filespec, 223

    help, 223  
    hotstandby, 223  
    info, 223  
    makecp  
        and checkpoint, 226  
    memory, 226  
    messages, 226  
    monitor, 226  
    netbackup, 226  
    netbackuplist, 227  
    netstat, 227  
    notify, 227  
    open, 227  
    parameter, 227  
    perfmon, 228  
    pid, 230  
    proctrace, 230  
    protocols, 230  
    runmerge, 231  
    save parameters, 231  
    shutdown, 231  
    solconnector propagator shutdown, 231  
    sqllist, 231  
    startmerge, 232  
    status, 232  
    throwout, 232  
    tid, 232  
    trace, 232  
    userid, 233  
    userlist, 233  
    usertrace, 237  
    version, 238  
ADMIN EVENT, 238  
ALL (keyword)  
    PROPAGATE TRANSACTIONS, 344  
ALTER TABLE SET HISTORY COLUMNS, 241  
ALTER TABLE SET NOSYNCHISTORY  
    described, 243  
ALTER TABLE SET SYNCHISTORY  
    described, 243  
ALTER TABLE statement, 239  
ALTER TRIGGER statement, 103, 245

---

ALTER USER statement, 246  
amount of memory used by in-memory tables and indexes, 225  
AND (operator), 45, 419  
API  
    defined, 515  
APPEND (keyword), 343  
Application Programming Interface  
    defined, 515  
AS, 28  
AS clause in SELECT statement, 28  
ASCending, 149  
ASCII, 423  
ASIN, 424  
ATAN, 424  
ATAN2, 425  
autocommit, 265  
AVG (function), 421

## B

backup  
    and SYS\_EVENT\_BACKUP, 502  
batch inserts and updates  
    optimizing, 203  
bcktime, 224  
BEGIN, 259  
BIGINT data type, 211  
BINARY  
    using CAST to enter values, 213  
BINARY data type, 213  
Binary Data Types, 213  
BIT\_AND function (bit-wise AND operator), 428  
BLOB, 22, 215  
    using CAST to enter values, 213  
BLOBs and CLOBs, 215  
bookmarks  
    dropping, 286, 315  
bulletin board, 325  
    (see also parameter bulletin board)

## C

CALL  
    example of using with EXECDIRECT and parameter, 276  
CALL statement, 248  
    invoking procedures, 36  
candidate key, 152  
CASCADE, 146, 304, 312  
CASCADE keyword in REVOKE statements, 375  
CASCADED  
    reserved word, 446  
CASE, 28, 422  
CAST, 28, 421, 422  
    entering binary values with, 213  
catalogs  
    creating, 161, 252  
    deleting, 162  
    described, 159  
CEILING, 425  
CHAR, 423  
CHAR data type, 209  
CHAR LARGE OBJECT data type, 210  
CHAR VARYING data type, 209  
CHARACTER data type, 209  
Character Data Types, 209, 210  
CHARACTER LARGE OBJECT data type, 210  
CHARACTER VARYING data type, 209  
CHECK, 156  
checkpoint  
    and 'makecp' command, 226  
    and SYS\_EVENT\_CHECKPOINT, 502  
Client-Server Architecture  
    described, 9  
    multi-user capability, 9  
CLOB, 215  
CLOB data type, 210  
clustering, 149  
clustering key, 455  
COALESCE, 422  
column, 5, 13  
columns



---

adding to a table, 146  
 deleting from a table, 147  
 COLUMNS system view, 482  
 commit block  
   defining refresh size, 358, 364  
 COMMIT statements  
   stored procedures, 72  
 COMMIT WORK, 10, 32  
 COMMIT WORK statement, 252  
 COMMITBLOCK (keyword)  
   DROP SUBSCRIPTION, 313  
   MESSAGE FORWARD, 358  
   MESSAGE GET REPLY, 364  
   REFRESH, 371  
 committing work  
   after altering table, 147  
   after altering users and roles, 143  
 Communication protocol  
   defined, 515  
 Comparison operators  
   described, 45  
 CONCAT, 423  
 concatenated indexes, 201  
 CONCURRENCY, 169  
 Concurrency control  
   optimistic, 169  
   pessimistic, 169  
   PESSIMISTIC vs. OPTIMISTIC, 169  
   purpose, 168  
   setting, 166  
     mixed, 166  
     pessimistic, 166  
 Concurrency Control and Locking, 167  
 Concurrency control mode  
   displaying, 460  
   MAINMEMORY, 460  
   MAINMEMORY PESSIMISTIC, 460  
   OPTIMISTIC, 460  
   PESSIMISTIC, 460  
 Conditions or warnings that cause  
 SYS\_EVENT\_ERROR, 512  
 configuring synchronization  
   setting system parameters, 398  
 connect string  
   changing to master name, 392  
 ConnectStrForMaster, 251  
 constraints  
   foreign key, 290  
 Control structures  
   in stored procedures, 47  
 CONVERT\_CHAR, 421  
 CONVERT\_DATE, 421  
 CONVERT\_DECIMAL, 421  
 CONVERT\_DOUBLE, 421  
 CONVERT\_FLOAT, 421  
 CONVERT\_INTEGER, 421  
 CONVERT\_LONGVARCHAR, 421  
 CONVERT\_NUMERIC, 421  
 CONVERT\_REAL, 421  
 CONVERT\_SMALLINT, 421  
 CONVERT\_TIME, 421  
 CONVERT\_TIMESTAMP, 421  
 CONVERT\_TINYINT, 421  
 CONVERT\_VARCHAR, 421  
 CONVERTORSTOUNIONS, 390  
 COS, 425  
 COT, 425  
 COUNT (function), 421  
 cptime, 224  
 CREATE CATALOG statement, 161, 252  
 CREATE EVENT statement, 124, 256  
 CREATE INDEX statement, 258  
 CREATE PROCEDURE statement, 259  
   declare section, 41  
   parameter section, 36  
 CREATE PUBLICATION  
   described, 277  
 CREATE ROLE statement, 281  
 CREATE SCHEMA statement, 282  
 CREATE SEQUENCE statement, 122, 284  
 CREATE SYNC BOOKMARK  
   described, 286  
 CREATE TABLE statement, 288  
 CREATE TRIGGER statement, 76, 292

---

---

CREATE USER statement, 303

CREATE VIEW statement, 303

creating

publications, 277

CURDATE, 425

CURRENT\_CATALOG (system function), 428

CURRENT\_SCHEMA (system function), 428

CURRENT\_USERID (system function), 428

CURSORNAME, 259, 269, 270

example usage, 270, 273

Cursors

closing in stored procedures, 63

default management in stored procedures, 72

dropping in stored procedures, 64

executing in stored procedures, 62

fetching in stored procedures, 63

handling in stored procedures, 61

in stored procedures, 73

parameter markers, 67

preparing in stored procedures, 61

CURTIME, 426

## D

data

exporting to file, 319

importing from a file, 336

returning in a stored procedure, 55

data management

using IBM solidDB SQL, 165

data types, 15, 428

IBM solidDB SQL, 137

supported, 209

database

column, 5, 13

free space in, 224

relational, 5

row, 5, 13

table, 5, 13

database creation time, 225

Database Management System

defined, 516

database objects

managing, 158

date and time literals, 429

DATE data type, 214

date time functions, 425

DAYNAME, 426

DAYOFMONTH, 426

DAYOFWEEK, 426

DAYOFYEAR, 426

dbconfigsize, 225

dbcreatetime, 225

dbfreesize, 224

DBMS

defined, 516

DBMS level errors

recovering, 351

recovering from, 362

dbpagesize, 225

dbsize, 224

DECIMAL data type, 212

DEFAULT, 57

DEFAULT (in START AFTER COMMIT), 409

Deferred Procedure Calls, 105

DEGREES, 425

DELETE (positioned) statement, 304

DELETE statement, 304

deleting

failed messages, 351

messages, 349

DESCending, 149

Diagnosing poor performance

diagnosis, 206

solutions, 206

symptoms, 206

DIFFERENCE, 425

Differences between SET and SET TRANSACTION,  
403

DOUBLE data type, 211, 215

DROP BOOKMARK

described, 286

DROP CATALOG statement, 304

DROP CONSTRAINT, 156

---

DROP EVENT statement, 124, 305  
DROP INDEX statement, 306  
DROP MASTER  
    described, 306  
DROP PROCEDURE statement, 307  
DROP PUBLICATION  
    described, 308  
DROP PUBLICATION REGISTRATION  
    described, 309  
DROP PUBLICATION REGISTRATION statement,  
309  
DROP REPLICA  
    described, 310  
DROP ROLE statement, 311  
DROP SCHEMA statement, 312  
DROP SEQUENCE statement, 312  
DROP SUBSCRIPTION  
    described, 313  
DROP SYNC BOOKMARK  
    described, 315  
DROP TABLE statement, 317  
DROP TRIGGER statement, 101, 318  
DROP USER statement, 318  
DROP VIEW statement, 319  
dropping  
    bookmarks, 286, 315  
    master database, 306  
    publications, 308, 309  
    replica databases, 310  
    subscriptions, 313  
duplicate inserts  
    fixing, 362

## E

EnableHints (parameter), 205  
END, 259  
END LOOP, 264  
ending  
    messages, 353  
Error handling  
    in stored procedures, 64

errors, 365  
    (see also fatal errors, synchronization errors)  
    DBMS, 351, 362  
    problem reporting, 190  
Errors that cause SYS\_EVENT\_ERROR, 511  
escape character, 432  
escape sequencefn, 44  
Evaluating application performance, 198  
EVENT  
    dropping an event, 305  
    posting an event, 259  
    registering for an event, 259  
    unregistering for an event, 259  
    waiting on an event, 259  
Events  
    code example, 125  
    using, 123  
    waiting on, 202  
events  
    ADMIN EVENT command, 238  
    HotStandby, 513  
EXCLUSIVE (lock mode), 175  
exclusive locks, 169, 175  
EXECDIRECT, 270  
    example usage, 275  
    using an SQL statement in a VARCHAR variable,  
275  
executing  
    failed messages, 361  
    messages, 355  
EXP, 425  
EXPLAIN PLAN FOR statement, 184, 206, 319  
EXPORT SUBSCRIPTION  
    described, 319  
expression, 420  
Expressions  
    in stored procedures, 45  
EXTRACT FROM, 426

## F

fatal errors

---

recovering from, 365  
FLOAT data type, 211  
FLOOR, 425  
fn  
    usage in {fn func\_name}, 42, 55  
FOR EACH REPLICA, 107  
foreign key, 151, 152  
FOREIGN KEY, 157  
foreign key constraints, 290  
free space in database, 224  
FULL (keyword), 343  
full table scan, 201  
functions, 368  
    (see also SQL functions)  
    AVG, 421  
    COUNT, 421  
    MAX, 421  
    MIN, 421  
    scalar, 42  
    SUM, 421  
Functions  
    for triggers, 103  
    SET\_PARAM(), 326  
    stack viewing in stored procedures, 74

## G

GET\_PARAM()  
    described, 325  
GET\_UNIQUE\_STRING, 269, 423  
    example usage, 270, 273, 275  
GLOBAL  
    keyword in CREATE TABLE command, 291  
GRANT EXECUTE ON statement, 74  
GRANT REFRESH ON  
    described, 328  
GRANT statement, 327

## H

HINT statement, 329  
hints  
    optimizer, 516

history table, 243  
HotStandby Events, 513  
HOUR, 426

## I

IBM solidDB  
    data management, 165  
IBM solidDB JDBC Driver  
    troubleshooting, 192  
IBM solidDB ODBC API  
    troubleshooting, 191  
IBM solidDB ODBC Driver  
    troubleshooting, 192  
IBM solidDB SQL  
    data management, 165  
    data types, 137  
    extensions, 137  
    functions, 138  
    using for database administration, 137  
IBM solidDB SQL Syntax  
    compliance, 137  
    using, 137  
IBM solidDB UNIFACE Driver  
    troubleshooting, 193  
IF statement  
    described, 47  
IF-THEN construct  
    described, 47  
IF-THEN-ELSE construct  
    described, 48  
IF-THEN-ELSEIF construct  
    described, 49  
IFNULL (system function), 428  
imdbsize, 225  
IMPORT  
    described, 336  
incremental publication  
    specifying, 243  
Index  
    defined, 516  
    primary key index, 149

---

secondary key index, 149  
 Indexes  
   creating, 148  
   creating a unique index, 148  
   deleting, 148  
   foreign key, 152  
   managing, 147  
   primary key indexes, 149  
   secondary key indexes, 149  
 indexes, 199  
   concatenated, 201  
   multi-column, 201  
 indexing  
   columns, 202  
 INSERT, 423  
   multirow, 339  
   using default values, 339  
 INSERT statement, 338  
 INT data type, 210  
 INTEGER data type, 210  
 Intelligent Transaction  
   parameter bulletin board, 380  
   using saved properties, 379  
 IS NULL (operator)  
   described, 47

**L**

large replicas  
   creating, 320  
 LCASE, 423  
 LEFT, 423  
 LENGTH, 423  
 LIKE, 289, 418, 419, 431  
 LIKE (in START AFTER COMMIT), 408  
 Listing users, 233  
 LOCATE, 423  
 Lock  
   EXCLUSIVE LOCK, 169, 175  
   SHARED LOCK, 169, 175  
   UPDATE LOCK, 175  
 lock duration, 177  
 lock modes  
   EXCLUSIVE, 175  
   SHARED, 175  
   UPDATE, 175  
 LOCK TABLE statement, 340  
 Locking  
   concurrency control, 167  
   described, 166  
   optimistic, 166, 169  
   pessimistic, 166, 169  
 locking mode  
   displaying, 460  
 locks  
   exclusive, 169, 175  
   shared, 169, 175  
   update, 175  
 LOG, 425  
 LOG10, 425  
 Logical conditions  
   described, 47  
 logical database, 253  
 Logical operators  
   AND, 45  
   described, 45  
   IS NULL, 47  
   NOT, 45, 54  
   OR, 45  
 LOGIN\_CATALOG (system function), 428  
 LOGIN\_SCHEMA (system function), 428  
 LOGIN\_USERID (system function), 428  
 logsize, 224  
   from 'info' command, 224  
 LONG NATIONAL VARCHAR data type, 210  
 LONG VARBINARY  
   using CAST to enter values, 213  
 LONG VARBINARY data type, 213  
 LONG VARCHAR data type, 210  
 LONG WVARCHAR data type, 210  
 LOOP, 264  
 Loops  
   in stored procedures, 51  
 LTRIM, 424

---

---

## M

### MAINTENANCE

set sync mode maintenance, 393

Maintenance Mode, 394

Managing indexes, 147

master database, 325, 344

(see also retrieving parameter values from)

changing network name, 392

dropping, 306

granting access to publications, 328

propagating transactions to, 344

requesting reply messages from, 363

setting node name, 396

setting parameters in, 398

user information, 344

master databases

properties in, 379

revoking access to publications, 376

setting parameters in, 369

master users

downloading list of, 344

MAX (function), 421

MaxStartStatements (parameter), 196

maxusers, 224

memtotal, 224

MESSAGE APPEND PROPAGATE TRANSACTIONS

described, 343

MESSAGE APPEND PROPAGATE WHERE

using properties, 380

MESSAGE APPEND REFRESH

described, 343

MESSAGE APPEND REGISTER PUBLICATION

described, 343

MESSAGE APPEND REGISTER REPLICA

described, 343

MESSAGE APPEND SUBSCRIBE, 343

(see also Message Append Refresh)

MESSAGE APPEND SYNC\_CONFIG

described, 343

MESSAGE APPEND UNREGISTER PUBLICATION

described, 343

MESSAGE APPEND UNREGISTER REPLICA

described, 343

MESSAGE BEGIN

described, 347

MESSAGE DELETE

described, 349

MESSAGE END

described, 353

MESSAGE EXECUTE

described, 355

MESSAGE FORWARD

described, 357

MESSAGE FROM REPLICA DELETE, 361

described, 351

MESSAGE FROM REPLICA EXECUTE

described, 361

MESSAGE FROM REPLICA RESTART, 363

MESSAGE GET REPLY

described, 363

messages, 351, 361

(see also error messages, failed messages, reply messages)

beginning, 347

deleting, 349

ending, 353

executing, 355

re-executing, 355

requesting replies from the master database, 363

saving, 353

sending, 357

metadata

exporting, 320

MIN (function), 421

MINUTE, 426

miscellaneous functions, 428

MOD, 425

monitorstate, 224

MONTH, 426

MONTHNAME, 426

---

multi-column indexes, 201

## N

name, 225

NATIONAL CHAR data type, 209

NATIONAL CHARACTERdata type, 209

NATIONAL VARCHAR data type, 210

NCHAR data type, 209

NCHAR LARGE OBJECT data type, 210

NCHAR VARYING data type, 210

NCLOB data type, 210

Network communication

troubleshooting, 193

node

setting, 396

node-def, 57

NONUNIQUE, 110

NORMAL

set sync mode normal, 393

NOT (operator), 45, 419

NOT NULL, 26

NOTUNIQUE, 408

NOW, 426

NULL, 23

NULLIF, 422

Nulls

handling, 53

numcursors, 224

NUMERIC data type, 212

numeric functions, 424

numlocks, 224

nummerges, 224

numtransactions, 224

numusers, 223

NVARCHAR data type, 210

## O

ODBC

defined, 516

Open Database Connectivity

defined, 516

openstate, 224

optimistic locking, 166, 169

optimizer hints

using, 204

Optimizer Hints

defined, 516

optimizing batch inserts and updates, 203

OR (operator), 45, 419

## P

parameter bulletin board

defining database-level parameters, 398

described, 368

Intelligent Transaction, 380

parameter modes, 262

Input parameters, 262

Input/output parameters, 262

Output parameters, 262

Parameters

MaxStartStatements, 196

using in triggers, 83

parameters

database-level, 325

defining persistent database-level, 398

deleting, 398

EnableHints, 205

GET\_PARAM(), 325

get\_param(), 325

placing on bulletin board, 368

put\_param(), 325

PUT\_PARAM(), 368

read-only, 325

retrieving from bulletin board, 325

SimpleSQLOpt, 199

updatable, 325

passwords

changing, 141

entering, 140, 141

percent sign character, 431

performance

diagnosing problems, 206

---

- indexes, 199
  - observing, 183
  - single-table SQL queries, 199
  - tuning, 197
  - using indexes to improve, 199
- pessimistic locking, 166, 169
- PI, 425
- POSITION, 424
- POWER, 425
- PRECISION data type, 211
- primary key, 147, 152
  - and index, 149
- Primary Key Indexes, 149
- primarystarttime, 225
- Privileges
  - managing, 138
  - stored procedures, 74
- problem reporting, 190
- PROC\_COUNT function
  - stored procedure stack, 74
- PROC\_NAME (N) function
  - stored procedure stack, 74
- PROC\_SCHEMA (N) function
  - stored procedure, 74
- Procedures, 36
  - (see also Stored procedures)
- processsize, 225
- proctrace, 194
- propagating
  - terminated messages, 365
- propagating transactions, 344
  - SAVE command, 377
  - setting default properties, 380
  - setting priority, 380
- properties
  - assigning, 379
  - saving as default, 380
  - saving default transaction propagation criteria, 380
- pseudo columns, 430
- psize, 225
- publications
  - creating, 277

- dropping, 308, 309
- granting access, 328
- refreshing, 344
- revoking access, 376

Push Synchronization, 107

- Example, 117

PUT\_PARAM()

- described, 368

## Q

QUARTER, 426

## R

RADIANS, 425

re-executing

- messages, 355

READ COMMITTED, 384

REAL data type, 211, 215

Recovery

- and transaction logging, 10

recovery

- DBMS level error, 351, 362

referenced table, 151

REFERENCES (keyword), 289, 290, 327, 375

referencing table, 151

Referential actions

- Cascade, 156
- No action, 155
- Restrict, 156
- Set default, 156
- Set null, 156

Referential Integrity, 151, 290

- and transient tables, 290
- constraints, 155
- dynamic constraint management, 156

REFRESH

- defining commit block, 358

REFRESH statement, 370

refreshes

- handling failure in the master database, 365
- handling failure in the replica database, 365



---

refreshing  
  publications, 344  
REGISTER EVENT statement, 374  
registering  
  replica databases, 344  
  setting replica node names, 396  
registrating databases  
  registration user, 400  
relational databases, 5  
Remote stored procedures, 56  
REPEAT, 424  
REPEATABLE READ, 384  
REPLACE, 424  
replica database, 325  
  (see also retrieving parameter values from)  
  refreshing from publications, 344  
  setting parameters in, 398  
replica databases  
  deleting messages, 349  
  dropping, 310  
  properties in, 379  
  registering, 344, 396, 400  
  requesting reply messages from the master data-  
  base, 363  
  saving transactions, 377  
  setting parameters in, 369  
  unregistering, 344  
Replica Property Names, 108  
reply messages  
  requesting from the master database, 363  
  setting timeout, 358  
RESTRICT, 146, 304, 312, 317  
RESTRICT keyword in REVOKE statements, 375  
RETURN keyword, 55  
REVOKE (Privilege from Role or User) statement,  
375  
REVOKE (Role from User) statement, 374  
REVOKE REFRESH ON  
  described, 376  
REVOKE SUBSCRIBE, 376  
  (see also Revoke Refresh)  
RIGHT, 424

roles  
  \_SYSTEM, 140  
  PUBLIC, 140  
  SYS\_ADMIN\_ROLE, 140  
  SYS\_CONSOLE\_ROLE, 140  
  SYS\_SYNC\_ADMIN\_ROLE, 140  
  SYS\_SYNC\_REGISTER\_ROLE, 140  
ROLLBACK, 10  
ROLLBACK statements  
  stored procedures, 72  
ROLLBACK WORK statement, 377  
ROUND, 425  
row, 5, 13  
Row Value Constructors, 30  
ROWID, 200  
ROWNUM, 199, 430, 442  
RTRIM, 424  
RVC, 30  
  (see also Row Value Constructors)

## **S**

SAVE  
  described, 377  
SAVE DEFAULT PROPAGATE PROPERTY  
WHERE  
  described, 380  
SAVE DEFAULT PROPERTY  
  described, 380  
SAVE PROPERTY  
  described, 379  
SAVE PROPERTY statement, 379  
saving  
  messages, 353  
Scalar functions, 42  
  described, 42  
scalar functions  
  described, 420  
schema  
  creating, 162  
  deleting, 163  
schemas

---

described, 159, 282  
**SECOND**, 426  
 secondary key  
     and index, 149  
 Secondary Key Indexes, 149  
 secondarystarttime, 225  
**SELECT** statement, 382  
 sending  
     messages, 357  
 Sequences  
     Using, 122  
**SERIALIZABLE**, 384  
 sernum, 224  
**SERVER\_INFO** system view, 483  
**SET**  
     differences between **SET** and **SET TRANSACTION**, 403  
**SET CATALOG** catalog\_name, 384  
**SET CATALOG** statement, 160  
**SET DURABILITY**, 181, 384  
**SET HISTORY COLUMNS**  
     described, 243  
**SET IDLE TIMEOUT**, 384  
**SET ISOLATION LEVEL**, 384  
**SET LOCK TIMEOUT**, 384  
**SET NOSYNCHISTORY**  
     described, 243  
**SET OPTIMISTIC LOCK TIMEOUT**, 384  
**SET READ-ONLY**, 384  
**SET READ-WRITE**, 384  
**SET SAFENESS**, 384  
**SET SCHEMA**, 384  
**SET SCHEMA** statement, 160, 388  
**SET SCHEMA USER** statement, 388  
**SET SQL** statement, 389  
**SET** statement, 384  
     in stored procedures, 42  
**SET STATEMENT MAXTIME**, 384  
**SET SYNC CONNECT**, 251  
     described, 392  
**SET SYNC MODE** statement, 393  
**SET SYNC NODE**  
     described, 396  
**SET SYNC PARAMETER**  
     described, 398  
**SET SYNC USER IDENTIFIED BY**  
     described, 400  
**SET SYNCHISTORY**, 241  
     described, 243  
 set theory, 16  
**SET TRANSACTION**  
     differences between **SET** and **SET TRANSACTION**, 403  
**SET TRANSACTION DURABILITY**, 180  
**SET TRANSACTION** statement, 402  
**SET TRANSACTION WRITE**, 402  
**SET WRITE**, 384  
 setting concurrency control, 166  
 Setting Lock Timeout, 167  
     for optimistic tables, 167  
 Setting the Concurrency (Locking) Mode to Optimistic or Pessimistic, 173  
**SHARED** (lock mode), 175  
 shared locks, 169, 175  
**SIGN**, 425  
 Simple SQL Optimization, 199  
 SimpleSQLOpt (parameter), 199  
**SIN**, 425  
**SLEEP**, 428  
**SMALLINT** data type, 210  
 soltrace.out, 193  
**SOUNDEX**, 424  
 space, 224  
**SPACE**, 424  
**SQL**  
     defined, 517  
     getting started, 13  
     mathematical origins of, 16  
     subqueries, 20  
     using in stored procedures, 73  
**SQL** functions  
     **GET\_PARAM()**, 325, 326  
     **PUT\_PARAM()**, 368  
**SQL** Info Facility, 183

---

SQL scripts, 138  
  sample.sql, 143  
  users.sql, 138

SQL statements  
  examples for administering indexes, 148  
  examples for managing database objects, 161  
  examples for managing indexes, 148  
  examples for managing users, roles, and user privileges, 140  
  examples of, 143  
  tuning, 197  
  using, 137

SQL-92, 137

SQL-99, 137

SQL\_LANGUAGES system table, 447

SQL\_TSI\_DAY, 426, 427

SQL\_TSI\_FRAC\_SECOND, 426, 427

SQL\_TSI\_HOUR, 426, 427

SQL\_TSI\_MINUTE, 426, 427

SQL\_TSI\_MONTH, 426, 427

SQL\_TSI\_QUARTER, 426, 427

SQL\_TSI\_SECOND, 426, 427

SQL\_TSI\_WEEK, 426, 427

SQL\_TSI\_YEAR, 426, 427

SQLERRNUM (variable)  
  error code, 65

SQLERROR (variable)  
  error string, 65

SQLERROR OF cursorname (variable), 66

SQLERRSTR (variable)  
  error string, 65

SQLROWCOUNT (variable)  
  row count, 65

SQLSUCCESS (variable)  
  stored procedure, 64

SQRT, 425

SSC\_TASK\_BACKGROUND, 196

START AFTER COMMIT statement, 408  
  analyzing failures in, 196  
  tuning performance of, 196

STORE  
  STORE clause of the CREATE TABLE command, 291

Stored procedures  
  assigning values to variables, 42  
  autocommit, 265  
  CREATE PROCEDURE statement, 35  
  cursors, 73  
  declaring local variables, 41  
  default cursor management, 72  
  default values, 37  
  defined, 517  
  described, 35  
  error handling, 64  
  exiting, 55  
  input parameters, 36  
  input/output parameters, 36  
  loops, 51  
  nesting procedures, 70  
  output parameters, 36  
  parameter markers in cursors, 67  
  positioned updates and deletes, 70  
  privileges, 74  
  procedure body, 42  
  procedure stack viewing, 74  
  remote, 56  
  tracing facilities for, 193  
  transactions, 72  
  triggers, 82  
  using events, 123  
  using parameters, 36  
  using SQL, 73  
  using SQL in, 60

string function, 423

Strings  
  zero-length, 54

Structured Query Language  
  defined, 517

SUBSCRIBE, 343  
  (see also Refresh)

subscriptions  
  defining commit block, 364  
  dropping, 313

---

---

exporting, 319  
 importing, 336  
 SUBSTRING, 424  
 SUM (function), 421  
 Sync Pull Notify, 107  
     Example, 117  
 SYNC\_CONFIG, 345  
 SYNC\_DELETE\_MESSAGES, 497  
 SYNC\_DELETE\_REPLICA\_MESSAGES, 498  
 SYNC\_MAINTENANCEMODE\_BEGIN  
     event, 394  
 SYNC\_MAINTENANCEMODE\_BEGIN (event),  
 505  
 SYNC\_MAINTENANCEMODE\_END  
     event, 394  
 SYNC\_MAINTENANCEMODE\_END (event), 506  
 SYNC\_REGISTER\_PUBLICATION, 493  
 SYNC\_REGISTER\_REPLICA, 490  
 SYNC\_SETUP\_CATALOG, 489  
 SYNC\_SHOW\_REPLICA\_SUBSCRIPTIONS, 496  
 SYNC\_SHOW\_SUBSCRIPTIONS, 495  
 SYNC\_UNREGISTER\_PUBLICATION, 494  
 SYNC\_UNREGISTER\_REPLICA, 491  
 SYNCHISTORY, 241  
 synchronization history table, 243  
 synchronization messages, 368  
 SYS\_ADMIN\_ROLE, 327  
 SYS\_ATTAUTH system table, 447  
 SYS\_BACKGROUNDJOB\_INFO, 196  
 SYS\_BACKGROUNDJOB\_INFO system table, 448  
 SYS\_BLOBS system table, 448  
 SYS\_BULLETIN\_BOARD system table, 464  
 SYS\_CARDINAL system table, 449  
 SYS\_CATALOGS system table, 449  
 SYS\_CHECKSTRINGS system table, 450  
 SYS\_COLUMNS system table, 450  
 SYS\_COLUMNS\_AUX system table, 451  
 SYS\_DL\_REPLICA\_CONFIG system table, 452  
 SYS\_DL\_REPLICA\_DEFAULT system table, 452  
 SYS\_EVENT\_BACKUP, 502  
 SYS\_EVENT\_BACKUPREQ, 502  
 SYS\_EVENT\_CHECKPOINT (event), 502  
 SYS\_EVENT\_CHECKPOINTREQ, 503  
 SYS\_EVENT\_ERROR, 503, 511  
 SYS\_EVENT\_IDLE, 503  
 SYS\_EVENT\_ILL\_LOGIN, 505  
 SYS\_EVENT\_IMDB\_MEMORY, 504  
 SYS\_EVENT\_MERGE, 506  
 SYS\_EVENT\_MERGEREQ, 506  
 SYS\_EVENT\_MESSAGES, 506  
 SYS\_EVENT\_NOTIFY, 507  
 SYS\_EVENT\_PARAMETER, 507  
 SYS\_EVENT\_PROCESS\_MEMORY, 507  
 SYS\_EVENT\_ROWS2MERGE, 509  
 SYS\_EVENT\_SACFAILED, 196, 509  
 SYS\_EVENT\_SHUTDOWNREQ, 509  
 SYS\_EVENT\_STATE\_MONITOR, 510  
 SYS\_EVENT\_STATE\_OPEN, 510  
 SYS\_EVENT\_STATE\_SHUTDOWN, 510  
 SYS\_EVENT\_STATE\_TRACE, 510  
 SYS\_EVENT\_TMCMD, 511  
 SYS\_EVENT\_TRX\_TIMEOUT, 511  
 SYS\_EVENT\_USERS, 511  
 SYS\_EVENTS system table, 453  
 SYS\_FORKEYPARTS system table, 453  
 SYS\_FORKEYS system table, 454  
 SYS\_GETBACKGROUNDJOB\_INFO, 196, 499  
 SYS\_HOTSTANDBY system table, 454  
 SYS\_KEYPARTS system table, 455  
 SYS\_KEYS system table, 455  
 SYS\_PROCEDURE\_COLUMNS system table, 457  
 SYS\_PROCEDURES system table, 456  
 SYS\_PROPERTIES system table, 457  
 SYS\_PUBLICATION\_ARGS system table, 464  
 SYS\_PUBLICATION\_REPLICA\_ARGS system  
 table, 464  
 SYS\_PUBLICATION\_REPLICA\_STMTARGS  
 system table, 465  
 SYS\_PUBLICATION\_REPLICA\_STMTS system  
 table, 465  
 SYS\_PUBLICATION\_STMTARGS system table,  
 466  
 SYS\_PUBLICATION\_STMTS system table, 466  
 SYS\_PUBLICATIONS system table, 467

---

SYS\_PUBLICATIONS\_REPLICA system table, 467  
 SYS\_RELAUTH system table, 458  
 SYS\_SCHEMAS system table, 458  
 SYS\_SEQUENCES system table, 459  
 SYS\_SYNC\_ADMIN\_ROLE, 327  
 SYS\_SYNC\_BOOKMARKS system table, 468  
 SYS\_SYNC\_HISTORY\_COLUMNS system table, 468  
 SYS\_SYNC\_INFO system table, 468  
 SYS\_SYNC\_MASTER\_MSGINFO system table, 469  
 SYS\_SYNC\_MASTER\_RECEIVED\_BLOB\_REFS system table, 470  
 SYS\_SYNC\_MASTER\_RECEIVED\_MSGPARTS system table, 471  
 SYS\_SYNC\_MASTER\_RECEIVED\_MSGS system table, 471  
 SYS\_SYNC\_MASTER\_STORED\_BLOB\_REFS system table, 471  
 SYS\_SYNC\_MASTER\_STORED\_MSGPARTS system table, 472  
 SYS\_SYNC\_MASTER\_STORED\_MSGS system table, 472  
 SYS\_SYNC\_MASTER\_SUBSC\_REQ system table, 473  
 SYS\_SYNC\_MASTER\_VERSIONS system table, 473  
 SYS\_SYNC\_MASTERS system table, 474  
 SYS\_SYNC\_RECEIVED\_BLOB\_ARGS system table, 474  
 SYS\_SYNC\_RECEIVED\_STMTS system table, 475  
 SYS\_SYNC\_REPLICA\_MSGINFO system table, 475  
 SYS\_SYNC\_REPLICA\_PROPERTIES system table, 459  
 SYS\_SYNC\_REPLICA\_RECEIVED\_BLOB\_REFS system table, 477  
 SYS\_SYNC\_REPLICA\_RECEIVED\_MSGPARTS system table, 477  
 SYS\_SYNC\_REPLICA\_RECEIVED\_MSGS system table, 478  
 SYS\_SYNC\_REPLICA\_STORED\_BLOB\_REFS system table, 478  
 SYS\_SYNC\_REPLICA\_STORED\_MSGPARTS system table, 479  
 SYS\_SYNC\_REPLICA\_STORED\_MSGS system table, 478  
 SYS\_SYNC\_REPLICA\_VERSIONS system table, 479  
 SYS\_SYNC\_REPLICAS system table, 480  
 SYS\_SYNC\_SAVED\_BLOB\_ARGS system table, 480  
 SYS\_SYNC\_SAVED\_STMTS system table, 480  
 SYS\_SYNC\_TRX\_PROPERTIES system table, 481  
 SYS\_SYNC\_USERMAPS system table, 481  
 SYS\_SYNC\_USERS system table, 482  
 SYS\_SYNONYM system table, 459  
 SYS\_TABLEMODES system table, 460  
 SYS\_TABLES system table, 460  
 SYS\_TRIGGERS (system table), 104  
 SYS\_TRIGGERS system table, 461  
 SYS\_TYPES system table, 462  
 SYS\_URole system table, 462  
 SYS\_USERS system table, 463  
 SYS\_VIEWS system table, 463  
 system functions, 427  
 system parameters, 325  
     (see also parameters)  
 System table  
     for triggers, 104  
 system tables, 447  
     described, 144  
     granting access, 144  
     viewing, 144  
 system views, 482

**T**

table, 5, 13  
     aliases, 19  
 table locks, 175, 179  
 tables  
     adding columns to, 146

---

- committing work after altering, 147
- creating, 145
- deleting columns from, 147
- managing, 143
- removing, 145
- TABLES system view, 484
- TAN, 425
- temporary table, 291
- THEN
  - keyword in CASE statement, 422
- TIME data type, 214
- timeout
  - setting for reply messages, 358
- TIMEOUT (keyword)
  - MESSAGE FORWARD, 357
  - MESSAGE GET REPLY, 358
- TIMESTAMP data type, 214
- TIMESTAMPADD, 426
- TIMESTAMPDIFF, 427
- TINYINT data type, 210
- TO (keyword)
  - MESSAGE FORWARD, 357
- tracestate, 224
- Tracing facilities for stored procedures and triggers, 193
- Transaction
  - described, 177
- transaction bulletin board, 325
  - (see also parameter bulletin board)
- Transaction durability level
  - choosing, 180
  - improving performance with, 181
  - setting, 180
- TRANSACTION ISOLATION Levels, 178
- Transaction Log, 10
- transaction propagation, 343
- Transactions, 32
  - COMMIT WORK, 10
  - described, 10
  - read-only
    - defining, 165
  - read-write
    - defining, 165
  - ROLLBACK, 10
  - stored procedures, 72
  - transaction log, 10
  - using triggers in, 86
- transactions, 344, 377
  - (see also Intelligent Transactions)
  - assigning properties, 379
  - propagating, 344
  - saving, 377
  - saving default properties , 380
  - setting default properties for propagation, 380
  - setting propagation priority, 380
- transient table, 291
- Triggers
  - altering attributes, 103
  - code example, 96
  - comments and restrictions, 81, 299
  - creating, 76
  - defined, 517
  - dropping, 101
  - error handling, 94
  - functions for analyzing and debugging, 103
  - how they work, 75
  - nested triggers, 95
  - obtaining information, 103
  - parameter settings, 105
  - privileges and security, 95
  - procedures, 82
  - recursive triggers, 95
  - setting cache, 105
  - setting default or derived columns, 83
  - setting nested maximum, 105
  - tracing facilities for, 193
  - transactions, 86
  - using, 75
  - using parameters and variables, 83
- TRIM, 424
- troubleshooting
  - IBM solidDB JDBC Driver, 192
  - IBM solidDB ODBC API, 191
  - IBM solidDB ODBC Driver, 192

---

- IBM solidDB UNIFACE Driver, 193
- Network communication, 193
- problem reporting, 190
- TRUNCATE, 425
- TRUNCATE TABLE statement, 411
- tuning
  - SQL statements, 197
- Tuning SQL Statements, 197
- Tuning SQL Statements and Applications, 197

## U

- UCASE, 424
- UIC (system function), 427
- underline, 431
- underscore, 431
- UNIQUE, 110, 157, 408
- unique constraint, 147
- UNLOCK TABLE statement, 411
- unregistering
  - replica databases, 344
- UPDATE (lock mode), 175
- UPDATE (Positioned) statement, 413
- UPDATE (Searched) statement, 414
- update locks, 175
- uptime, 224
- user names
  - reserved names, 139
- user privileges, 139
  - granting, 141
  - granting administrator privileges, 143
  - revoking, 142
- user roles, 139
  - administrator, 140, 143
  - changing password, 141
  - creating, 141
  - deleting, 141
  - giving a user a role, 142
  - granting privileges to, 141, 142
  - reserved role names, 139
  - revoking privileges from, 142
  - revoking the role of a user, 142

- system console role, 140
- userlist, 233
- users
  - creating, 140
  - deleting, 140
- users and roles
  - committing work after altering, 143
- USERS system view, 484
- usertrace, 193
- Using Events, 123
- Using indexes to improve query performance, 199
- Using SQL in stored procedures, 60
- using SQL wildcards, 431

## V

- VARBINARY
  - using CAST to enter values, 213
- VARBINARY data type, 213
- VARCHAR data type, 209
- Variables
  - assigning in stored procedures, 42
  - SQLERRNUM, 65
  - SQLERROR, 65
  - SQLERROR OF cursorname, 66
  - SQLERRSTR, 65
  - SQLROWCOUNT, 65
  - SQLSUCCESS, 64
  - using in triggers, 83

## W

- WCHAR data type, 209
- WEEK, 427
- WHEN
  - in case\_specification, 422
  - keyword in event specification, 257
- WHERE (keyword)
  - PROPAGATE TRANSACTIONS, 344
- WHILE-LOOP statement
  - described, 51
- wildcard characters, 430
- WRITETRACE, 193

---

WVARCHAR data type, 210

## **Y**

YEAR, 427

## **Z**

Zero-length strings, 54