IBM solidDB
IBM solidDB Universal Cache

バージョン 6.5

SQL ガイド

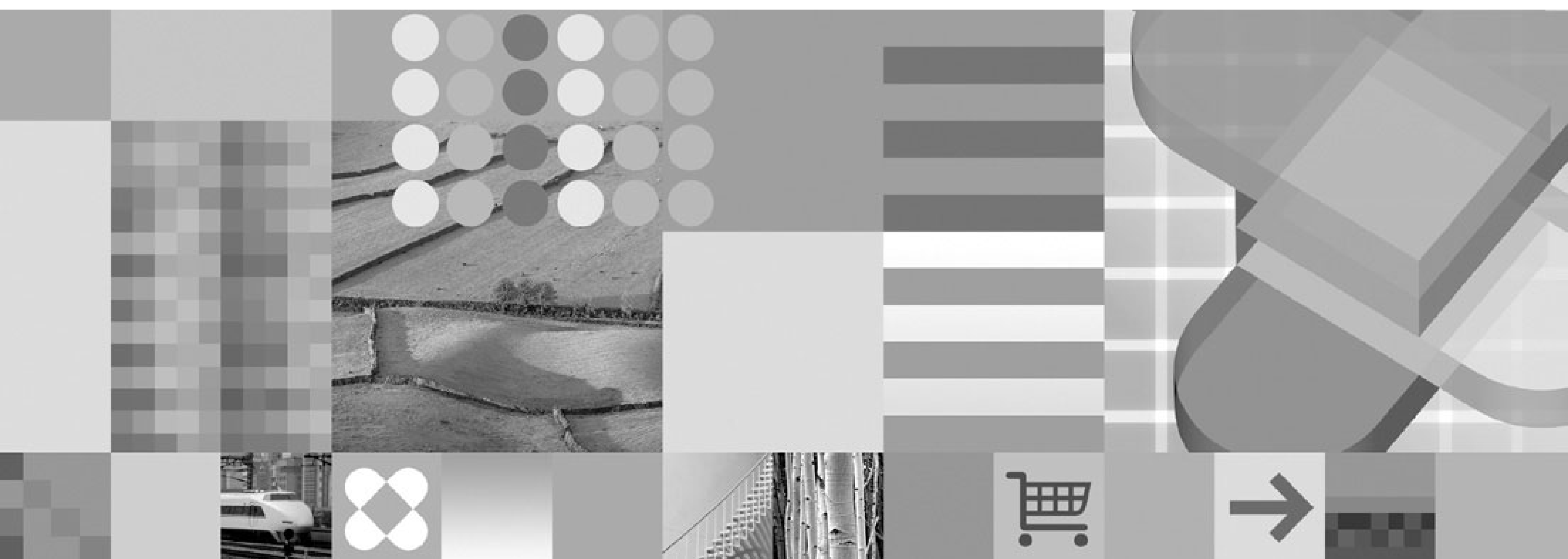
Information Management software

SC88-8168-00
(英文原典：SC23-9871-00)

IBM solidDB
IBM solidDB Universal Cache

バージョン 6.5

IBM



SQL ガイド

Information Management software

SC88-8168-00
(英文原典：SC23-9871-00)

注記

本書および本書で紹介する製品をご使用になる前に、 449 ページの『特記事項』に記載されている情報をお読みください。

本書は、バージョン 6 リリース 5 の IBM solidDB (製品番号 5724-V17) および IBM solidDB Universal Cache (製品番号 5724-W91)、および新しい版で明記されていない限り、以降のすべてのリリースおよびモディフィケーションに適用されます。

お客様の環境によっては、資料中の円記号がバックスラッシュと表示されたり、バックスラッシュが円記号と表示されたりする場合があります。

原典： SC23-9871-00
IBM solidDB
IBM solidDB Universal Cache
Version 6.5
SQL Guide

発行： 日本アイ・ビー・エム株式会社

担当： トランスレーション・サービス・センター

第1刷 2009.10

© Solid Information Technology Ltd. 1993, 2009

目次

図 xi

表. xiii

本書について xvii
書体の規則 xvii
構文表記法の規則. xviii

1 データベース概念 1
リレーショナル・データベース 1
表、行、および列. 1
異なる表のデータの関連付け 2
クライアント/サーバー・アーキテクチャー 4
マルチユーザー機能 5
トランザクション. 5
トランザクション・ロギングおよびリカバリー . . . 5
バックグラウンド. 6
要約 7

2 SQL の概要 9
表、行、および列. 9
SQL 9
SQL の数学的起源 11
関連データを持つ表の作成 12
表の別名 14
副照会 14
各データ型にどの形式を使用するか 15
BLOB (またはバイナリー・データ型). 17
NULL IS NOT NULL (つまり、「上記のどれでもないことを SQL で何というか」) 18
NOT NULL 20
式およびキャスト 20
行値コンストラクター. 22
トランザクションの詳細 24
要約. 24
SQL に関する追加情報の検索先 24

3 ストアード・プロシージャー、イベント、トリガー、およびシーケンス 27
ストアード・プロシージャー 27
基本的なプロシージャー構造 27
プロシージャーのネーミング 28
パラメーター・セクション 28
宣言セクション 31
プロシージャー本体 32
代入. 32
式 34
制御構造 36
リモート・ストアード・プロシージャー. 42
アクセス権限. 44
ストアード・プロシージャーでの SQL の使用. . . 46
EXECDIRECT. 47
カーソルの使用 47
エラー処理 50
カーソル内のパラメーター・マーカー 52
他のプロシージャーの呼び出し 54
位置付け更新および位置付け削除 54
トランザクション 55
デフォルト・カーソル管理 56
SQL についての注 57
プロシージャー・スタックを表示する関数 . . . 57
プロシージャー特権 57
トリガーの使用 58
トリガーの仕組み 58
トリガーの作成 59
キーワードおよび節 60
トリガーのコメントおよび制限事項 64
トリガーおよびプロシージャー. 65
デフォルト列または派生列の設定 65
パラメーターおよび変数の使用. 65
トリガーおよびトランザクション 68
再帰エラーおよび並行性競合エラー 68
トリガーの特権およびセキュリティー 75
トリガー内からのエラーの発生. 76
トリガーの例. 76
トリガーのドロップ 80
トリガー属性の変更 80
トリガー情報の入手 81
トリガー関数. 81
SYS_TRIGGERS システム表. 81
トリガー・パラメーターの設定. 82
据え置きプロシージャー呼び出し 83
プル同期通知 (「プッシュ同期」) の例 92
バックグラウンド・ジョブの実行のトレース . . 95
バックグラウンド・タスクの制御 96
シーケンスの使用 96
イベントの使用 97

4 データベース管理のための solidDB SQL の使用 107
solidDB SQL 構文の使用 107
solidDB SQL データ型 107
solidDB ADMIN COMMAND 107
関数の使用 108
ユーザー特権およびロールの管理. 108
ユーザー特権 108
ユーザー・ロール 109
SQL ステートメントの例 109
表の管理 111
システム表へのアクセス. 112
SQL ステートメントの例 113

索引の管理 115
SQL ステートメントの例 115
主キー索引 116
副次キー索引 116
重複索引に対する保護 117
参照整合性 118
主キーと候補キー 118
外部キー 118
参照アクション. 121
制約の動的な管理 122
データベース・オブジェクトの管理 123
概要 123
カタログ 124
スキーマ 124
カタログおよびスキーマ内でオブジェクトを一意的に識別する 125
SQL ステートメントの例 126

5 トランザクションの管理 129

トランザクションの管理. 129
読み取り専用トランザクションおよび読み取り/書き込みトランザクションの定義. 129
並行性制御の設定 129
並行性制御とロック方式. 131
並行性制御の目的 132
排他ロックと共有ロック. 133
ペシミスティック並行性制御およびオプティミスティック並行性制御 133
表ロック 139
ロック期間 141
トランザクション分離レベル 142
各種のロック情報 143
情報ロックの要約 143
トランザクション持続性の選択 144
トランザクション持続性レベルの設定 144

6 診断およびトラブルシューティング 147

パフォーマンスの監視 147
SQL 情報機能 147
EXPLAIN PLAN FOR ステートメント 148
ストアード・プロシージャーおよびトリガーのトレース機能 154
ユーザー定義可能な、プロシージャー・コードからのトレース出力 154
プロシージャー実行トレース 155
START AFTER COMMIT ステートメントのパフォーマンスの測定および向上 156
START AFTER COMMIT ステートメントのパフォーマンスのチューニング 156
START AFTER COMMIT ステートメントでの障害の分析 156

7 パフォーマンスのチューニング . . . 159

SQL ステートメントとアプリケーションのチューニング. 159
アプリケーション・パフォーマンスの評価. . . 159
ストアード・プロシージャー言語の使用 . . . 160
単一表 SQL 照会の最適化 160
索引を使用した照会パフォーマンスの向上. . . . 161
全表スキャン 162
連結索引 163
イベント待ち 164
バッチ挿入および更新の最適化 164
バッチ挿入および更新の高速化 165
オプティマイザーのヒントの使用. 165
パフォーマンス低下の診断 166

付録 A. データ型 169

文字データ型 169
数値データ型 170
バイナリー・データ型 172
日付データ型 173
TIME データ型. 173
TIMESTAMP データ型 173
最小限の非ゼロ数値 174
BLOB および CLOB 174
表内のさまざまな列値の説明 175

付録 B. solidDB SQL 構文 177

ADMIN COMMAND 177
ADMIN EVENT 186
使用法. 187
例 187
ALTER REMOTE SERVER 187
ALTER TABLE. 188
使用法. 188
例 189
ALTER TABLE ... SET HISTORY COLUMNS . . 189
使用法. 190
マスターでの使用 190
レプリカでの使用 190
例 190
戻り値. 190
関連項目 191
ALTER TABLE ... SET SYNCHISTORY 191
使用法. 191
マスターでの使用 192
レプリカでの使用 192
例 192
戻り値. 192
関連項目 193
ALTER TRIGGER 193
使用法. 193
例 193
ALTER USER 193
使用法. 193
例 193
ALTER USER (レプリカ) 193
使用法. 193
マスターでの使用 194
レプリカでの使用 194
例 195

戻り値. 195
CALL 195
サポート条件 195
使用法. 195
トランザクション 196
リモート・プロシージャーからの戻り値 . . . 196
リモート・ストアード・プロシージャー呼び出しのアクセス権限. 197
持続性. 198
例 198
COMMIT WORK 198
使用法. 198
例 198
関連項目 199
CREATE CATALOG 199
使用法. 199
例 201
CREATE EVENT 201
使用法. 201
例 203
関連項目 203
CREATE INDEX 203
使用法. 203
例 204
関連項目 204
CREATE PROCEDURE 204
使用法. 205
SQL ステートメントの準備. 209
準備済み SQL ステートメントの実行 209
結果のフェッチ. 210
カーソルのクローズとドロップ 210
エラーの検査 210
トランザクションの使用. 210
シーケンサー・オブジェクトおよびイベント・アラートの使用 211
writetrace 211
プロシージャー・スタック関数 211
動的カーソル名. 211
EXECDIRECT 212
CREATE PROCEDURE 213
明示的な RETURN ステートメントの使用. . . 213
EXECDIRECT の使用. 214
CURSORNAME の使用 214
GET_UNIQUE_STRING と CURSORNAME の使用 214
例 6 215
同期メッセージ用の固有の名前の作成 215
GET_UNIQUE_STRING の使用 215
CREATE [OR REPLACE] PUBLICATION 216
使用法. 217
マスターでの使用 218
レプリカでの使用 218
例 219
戻り値. 220
CREATE [OR REPLACE] REMOTE SERVER. . . 220
CREATE ROLE. 221
使用法. 221
例 221
CREATE SCHEMA 221
使用法. 221
例 222
CREATE SEQUENCE. 223
使用法. 223
例 224
CREATE SYNC BOOKMARK 224
サポート条件 224
使用法. 224
マスターでの使用 225
レプリカでの使用 225
例 225
戻り値. 225
CREATE TABLE 225
使用法. 226
例 228
CREATE TRIGGER 229
使用法. 229
トリガー名 230
BEFORE | AFTER 節. 230
INSERT | UPDATE | DELETE 節 232
table_name 233
trigger_body 233
REFERENCING 節. 234
{OLD | NEW} column_name AS col_identifier 234
トリガーのコメントおよび制限事項 235
CREATE USER. 238
使用法. 238
例 238
CREATE VIEW. 238
使用法. 238
例 238
DELETE 238
使用法. 238
例 239
DELETE (位置付け) 239
使用法. 239
例 239
DESCRIBE 239
DROP CATALOG 240
使用法. 241
例 241
DROP EVENT 241
使用法. 241
例 241
DROP INDEX 241
使用法. 241
例 242
DROP MASTER 242
使用法. 242
マスターでの使用 242
レプリカでの使用 242
例 242
戻り値. 242

DROP PROCEDURE . . . 243
使用法. . . 243
例 . . . 243
DROP PUBLICATION . . . 243
使用法. . . 243
マスターでの使用 . . . 243
レプリカでの使用 . . . 243
例 . . . 243
戻り値. . . 244
DROP PUBLICATION REGISTRATION. . . 244
サポート条件 . . . 244
使用法. . . 244
マスターでの使用 . . . 244
レプリカでの使用 . . . 244
例 . . . 244
戻り値. . . 244
DROP REMOTE SERVER . . . 245
DROP REPLICA . . . 245
サポート条件 . . . 245
使用法. . . 245
マスターでの使用 . . . 246
レプリカでの使用 . . . 246
例 . . . 246
戻り値. . . 246
DROP ROLE . . . 246
使用法. . . 246
例 . . . 246
DROP SCHEMA . . . 246
使用法. . . 247
例 . . . 247
DROP SEQUENCE. . . 247
使用法. . . 247
例 . . . 247
DROP SUBSCRIPTION . . . 247
サポート条件 . . . 247
使用法. . . 248
マスターでの使用 . . . 249
レプリカでの使用 . . . 249
例 . . . 249
DROP SYNC BOOKMARK . . . 249
サポート条件 . . . 249
使用法. . . 250
マスターでの使用 . . . 250
レプリカでの使用 . . . 250
例 . . . 250
戻り値. . . 250
DROP TABLE . . . 251
使用法. . . 251
例 . . . 251
DROP TRIGGER . . . 251
使用法. . . 251
例 . . . 252
DROP USER. . . 252
使用法. . . 252
例 . . . 252
DROP VIEW . . . 252
使用法. . . 252
例 . . . 252
EXPLAIN PLAN FOR . . . 252
使用法. . . 252
例 . . . 252
EXPORT SUBSCRIPTION . . . 253
サポート条件 . . . 253
使用法. . . 253
マスターでの使用 . . . 255
レプリカでの使用 . . . 255
例 . . . 255
戻り値. . . 255
EXPORT SUBSCRIPTION TO REPLICA . . . 256
サポート条件 . . . 256
使用法. . . 256
マスターでの使用 . . . 257
レプリカでの使用 . . . 257
例 . . . 257
戻り値. . . 257
GET_PARAM() . . . 257
サポート条件 . . . 257
使用法. . . 257
マスターでの使用 . . . 258
レプリカでの使用 . . . 258
solidDB システム・パラメーター . . . 258
例 . . . 258
戻り値. . . 259
関連項目 . . . 259
GRANT . . . 259
使用法. . . 259
例 . . . 260
関連項目 . . . 260
GRANT PASSTHROUGH . . . 260
GRANT REFRESH. . . 260
サポート条件 . . . 261
使用法. . . 261
マスターでの使用 . . . 261
レプリカでの使用 . . . 261
例 . . . 261
戻り値. . . 261
HINT . . . 261
疑似コメント ID . . . 262
例 1 . . . 263
例 2 . . . 263
使用法. . . 265
例 . . . 266
IMPORT . . . 267
使用法. . . 267
マスターでの使用 . . . 268
レプリカでの使用 . . . 268
例 . . . 268
戻り値. . . 268
INSERT . . . 269
使用法. . . 270
例 . . . 270
LIST . . . 270

LOCK TABLE 272
使用法. 272
例 274
戻り値. 274
関連項目 274
MESSAGE APPEND 275
サポート条件 275
使用法. 275
マスターでの使用 277
レプリカでの使用 277
例 277
戻り値. 277
MESSAGE BEGIN 278
サポート条件 278
使用法. 278
マスターでの使用 278
レプリカでの使用 279
例 279
マスターからの戻り値 279
MESSAGE DELETE 279
サポート条件 279
使用法. 280
マスターでの使用 280
レプリカでの使用 280
例 280
MESSAGE DELETE CURRENT TRANSACTION 281
サポート条件 281
使用法. 281
マスターでの使用 282
レプリカでの使用 282
例 282
戻り値. 282
MESSAGE END 283
サポート条件 283
使用法. 283
マスターでの使用 283
レプリカでの使用 283
レプリカからの戻り値 283
マスターからの戻り値 284
MESSAGE EXECUTE. 284
サポート条件 284
使用法. 284
マスターでの使用 285
レプリカでの使用 285
結果セット 285
例 285
戻り値. 285
MESSAGE FORWARD 286
サポート条件 286
使用法. 286
例 288
レプリカからの戻り値 288
マスターからの戻り値 290
MESSAGE FROM REPLICA DELETE 290
MESSAGE FROM REPLICA EXECUTE. 290
サポート条件 290
使用法. 291
マスターでの使用 291
レプリカでの使用 291
例 291
戻り値. 291
MESSAGE FROM REPLICA RESTART. 292
MESSAGE GET REPLY 292
サポート条件 292
使用法. 292
マスターでの使用 293
レプリカでの使用 293
例 293
レプリカからの戻り値 293
マスターからの戻り値 295
結果セット 295
POST EVENT 297
PUT_PARAM() 297
サポート条件 297
使用法. 297
マスターでの使用 297
レプリカでの使用 297
「PUT_PARAM()」と「SAVE PROPERTY property_name VALUE property_value;」の違い . 297
例 298
戻り値. 298
関連項目 298
REFRESH. 298
使用法. 298
例 299
戻り値. 299
REGISTER EVENT 302
REVOKE (ユーザーからロールを) 302
使用法. 302
例 302
REVOKE (ロールまたはユーザーから特権を). . . 303
使用法. 303
例 303
関連項目 303
REVOKE PASSTHROUGH 303
REVOKE REFRESH 304
サポート条件 304
使用法. 304
マスターでの使用 304
レプリカでの使用 304
例 304
戻り値. 304
ROLLBACK WORK 304
使用法. 305
例 305
SAVE 305
サポート条件 305
使用法. 305
マスターでの使用 306
レプリカでの使用 306
例 306
戻り値. 306

SAVE PROPERTY 307
サポート条件 307
使用法. 307
マスターでの使用 308
レプリカでの使用 308
「PUT_PARAM()」と「SAVE PROPERTY property_name VALUE property_value;」の違い . 308
例 308
戻り値. 308
結果セット 308
SELECT 308
使用法. 309
例 309
START WITH の例 309
LEVEL と ORDER SIBLINGS BY の例 . . . 310
SET 310
使用法. 310
SET および SET TRANSACTION の違い . . . 311
SET (読み取り/書き込みレベル) 311
SET CATALOG. 311
SET DELETE CAPTURE 312
SET DURABILITY. 312
SET ISOLATION LEVEL 312
SET PASSTHROUGH 313
SET SAFENESS 313
SET SCHEMA 314
SET SQL 315
SET STATEMENT MAXTIME 316
SET SYNC 316
SET TIMEOUT 324
SET TRANSACTION 324
START AFTER COMMIT 329
使用法. 330
トランザクション 330
バックグラウンド・ステートメントのコンテキスト 331
持続性. 331
ロールバック 331
実行の順序 331
例 331
TRUNCATE TABLE 332
UNLOCK TABLE 332
使用法. 333
LOCK および UNLOCK の使用例 333
戻り値. 333
関連項目 334
UNREGISTER EVENT 334
UPDATE (位置付け) 334
使用法. 334
例 334
UPDATE (検索付き) 334
使用法. 334
例 334
WAIT EVENT 335
table_reference 335
query_specification 335
search_condition 336
Check_condition 337
式 337
ストリング関数. 339
数字関数 340
日時関数 342
システム関数 343
各種関数 344
data_type 345
日時リテラル 345
疑似列. 345
ワイルドカード文字 346
SQL ワイルドカードの使用. 346
リテラルとしてのワイルドカード文字 347

付録 C. 予約語 349

付録 D. データベース・システム表とシステム・ビュー 363
システム表 363
SQL_LANGUAGES 363
SYS_ATTAUTH. 363
SYS_AUDIT_TRAIL 364
SYS_BACKGROUNDJOB_INFO 365
SYS_BLOBS 365
SYS_CARDINAL 366
SYS_CATALOGS 366
SYS_CHECKSTRINGS 367
SYS_COLUMNS 367
SYS_COLUMNS_AUX 368
SYS_DL_REPLICA_CONFIG 368
SYS_DL_REPLICA_DEFAULT 369
SYS_EVENTS 369
SYS_FEDT_DB_PARTITION 370
SYS_FEDT_TABLE_PARTITION 370
SYS_FORKEYPARTS 371
SYS_FORKEYS 372
SYS_HOTSTANDBY 372
SYS_INFO 372
SYS_KEYPARTS 372
SYS_KEYS 373
SYS_LOGPOS 374
SYS_PROCEDURES 374
SYS_PROCEDURE_COLUMNS. 375
SYS_PROPERTIES 376
SYS_RELAUTH. 376
SYS_SCHEMAS. 377
SYS_SEQUENCES 377
SYS_SERVER 377
SYS_SYNC_REPLICA_PROPERTIES 378
SYS_SYNONYM 378
SYS_TABLEMODES 378
SYS_TABLES 379
SYS_TRIGGERS 380
SYS_TYPES 380
SYS_UROLE. 381

SYS_USERS 381
SYS_VIEWS 382
データ同期に使用されるシステム表 382
SYS_BULLETIN_BOARD 382
SYS_PUBLICATION_ARGS 383
SYS_PUBLICATION_REPLICA_ARGS 383
SYS_PUBLICATION_REPLICA_STMTARGS . . 383
SYS_PUBLICATION_REPLICA_STMTS 384
SYS_PUBLICATION_STMTARGS 384
SYS_PUBLICATION_STMTS 385
SYS_PUBLICATIONS 385
SYS_PUBLICATIONS_REPLICA 386
SYS_SYNC_BOOKMARKS 386
SYS_SYNC_HISTORY_COLUMNS 387
SYS_SYNC_INFO 387
SYS_SYNC_MASTER_MSGINFO 387
SYS_SYNC_MASTER_RECEIVED_BLOB_REFS 389
SYS_SYNC_MASTER_RECEIVED_MSGPARTS 389
SYS_SYNC_MASTER_RECEIVED_MSGS . . . 389
SYS_SYNC_MASTER_STORED_BLOB_REFS . . 390
SYS_SYNC_MASTER_STORED_MSGPARTS . . 390
SYS_SYNC_MASTER_STORED_MSGS 390
SYS_SYNC_MASTER_SUBSC_REQ 391
SYS_SYNC_MASTER_VERSIONS 391
SYS_SYNC_MASTERS 392
SYS_SYNC_RECEIVED_BLOB_ARGS 392
SYS_SYNC_RECEIVED_STMTS 393
SYS_SYNC_REPLICA_MSGINFO 394
SYS_SYNC_REPLICA_RECEIVED_BLOB_REFS 395
SYS_SYNC_REPLICA_RECEIVED_MSGPARTS 395
SYS_SYNC_REPLICA_RECEIVED_MSGS . . . 396
SYS_SYNC_REPLICA_STORED_BLOB_REFS 396
SYS_SYNC_REPLICA_STORED_MSGS 396
SYS_SYNC_REPLICA_STORED_MSGPARTS . . 397
SYS_SYNC_REPLICA_VERSIONS. 397
SYS_SYNC_REPLICAS 398
SYS_SYNC_SAVED_BLOB_ARGS 398
SYS_SYNC_SAVED_STMTS 398
SYS_SYNC_TRX_PROPERTIES 399
SYS_SYNC_USERMAPS 399
SYS_SYNC_USERS 400
システム・ビュー 400
COLUMNS 400
SERVER_INFO 401
TABLES 401
USERS 402
同期関連ビュー. 402
SYNC_FAILED_MESSAGES 402
SYNC_FAILED_MASTER_MESSAGES 403
SYNC_ACTIVE_MESSAGES 403
SYNC_ACTIVE_MASTER_MESSAGES 404

付録 E. データベース仮想表 405

SYS_LOG. 405

付録 F. システム・ストアード・プロシージャー. 411

同期関連ストアード・プロシージャー 411
SYNC_SETUP_CATALOG 411
SYNC_REGISTER_REPLICA 412
SYNC_UNREGISTER_REPLICA 413
SYNC_REGISTER_PUBLICATION 415
SYNC_UNREGISTER_PUBLICATION 416
SYNC_SHOW_SUBSCRIPTIONS 417
SYNC_SHOW_REPLICA_SUBSCRIPTIONS. . . 418
SYNC_DELETE_MESSAGES 419
SYNC_DELETE_REPLICA_MESSAGES 420
各種ストアード・プロシージャー. 420
SYS_GETBACKGROUNDJOB_INFO 420

付録 G. システム・イベント 423

各種イベント 424
SYS_EVENT_ERROR の原因となるエラー. . . . 431
SYS_EVENT_MESSAGES の原因となる状態または警告 432

索引 435

特記事項. 449

図

1. プル同期通知 93
2. 参照制約 119
3. 自己参照制約 120
4. 実行グラフ 1 152
5. 実行グラフ 2 154

表

1. 書体の規則 xvii
2. 構文表記法の規則 xviii
3. データベース表の例 1
4. データベース表の例 9
5. 比較演算子 34
6. 論理演算子: NOT 35
7. 論理演算子: AND 35
8. 論理演算子: OR 35
9. パラメーターからのデータ型の判別 53
10. トリガーのステートメント・アトミシティ 64
11. BEFORE/AFTER トリガーの INSERT/UPDATE/DELETE 操作 70
12. 項目例 1 73
13. 項目例 2 73
14. SYS_TRIGGERS システム表のメタデータ 81
15. 予約済みのユーザー名およびユーザー・ロール 109
16. 表の表示とアクセス権限の付与 112
17. 式および演算子 122
18. SQL Info のレベル 147
19. EXPLAIN PLAN FOR のユニット 148
20. EXPLAIN PLAN 表の列 149
21. ユニットの INFO 列内のテキスト 150
22. EXPLAIN PLAN FOR の例 1 151
23. EXPLAIN PLAN FOR の例 2 152
24. パフォーマンス低下の診断 167
25. サポートされているデータ型 169
26. 文字データ型 169
27. 数値データ型 170
28. バイナリー・データ型 172
29. 日付データ型 173
30. TIME データ型 173
31. TIMESTAMP データ型 173
32. 最小限の非ゼロ数値 174
33. ADMIN COMMAND 構文とオプション 179
34. ALTER TABLE SET HISTORY COLUMNS の戻り値 190
35. ALTER TABLE SET SYNCHISTORY の戻り値 192
36. ALTER USER の戻り値 195
37. パラメーター・モードの比較 206
38. 制御ステートメント 207
39. CREATE PUBLICATION の戻り値 220
40. CREATE SYNC BOOKMARK の戻り値 225
41. トリガーのステートメント・アトミシティ 234
42. DROP MASTER の戻り値 242
43. DROP PUBLICATION の戻り値 244
44. DROP PUBLICATION REGISTRATION の戻り値 244
45. DROP REPLICA の戻り値 246
46. DROP SUBSCRIPTION の戻り値 249
47. DROP SYNC BOOKMARK の戻り値 251
48. EXPORT SUBSCRIPTION の戻り値 255
49. EXPORT SUBSCRIPTION TO REPLICA の戻り値 257
50. GET_PARAM の戻り値 259
51. GRANT REFRESH の戻り値 261
52. ヒント 263
53. IMPORT の戻り値 268
54. LOCK TABLE の戻り値 274
55. MESSAGE APPEND の戻り値 277
56. レプリカからの MESSAGE BEGIN の戻り値 279
57. マスターからの MESSAGE BEGIN の戻り値 279
58. レプリカからの MESSAGE DELETE の戻り値 280
59. マスターからの MESSAGE DELETE の戻り値 281
60. MESSAGE DELETE CURRENT TRANSACTION の戻り値 282
61. レプリカからの MESSAGE END の戻り値 283
62. マスターからの MESSAGE END の戻り値 284
63. MESSAGE EXECUTE の戻り値 285
64. レプリカからの MESSAGE FORWARD の戻り値 288
65. マスターからの MESSAGE FORWARD の戻り値 290
66. MESSAGE FROM REPLICA EXECUTE の戻り値 291
67. レプリカからの MESSAGE GET REPLY の戻り値 294
68. マスターからの MESSAGE GET REPLY の戻り値 295
69. MESSAGE GET REPLY 結果セット表 295
70. PUT_PARAM() の戻り値 298
71. REFRESH の戻り値 299
72. REVOKE REFRESH の戻り値 304
73. SAVE の戻り値 306
74. SAVE PROPERTY の戻り値 308
75. SET SYNC の戻り値 317
76. SET SYNC CONNECT の戻り値 318
77. 同期履歴表への各種操作の適用方法 319
78. SET SYNC MODE の戻り値 320
79. SET SYNC NODE の戻り値 321
80. SET SYNC PARAMETER の戻り値 322
81. LOCK TABLE の戻り値 334
82. table_reference 335
83. query_specification 335
84. search_condition 336
85. Check_condition 337
86. 式 337
87. ストリング関数 339
88. 数字関数 340

89. 日時関数 342
90. システム関数 343
91. 各種関数 344
92. data_type 345
93. 日時リテラル 345
94. 疑似列 346
95. ワイルドカード文字 346
96. 予約語リスト 349
97. SQL_LANGUAGES 363
98. SYS_ATTAUTH 363
99. SYS_AUDIT_TRAIL 表の定義 364
100. SYS_BACKGROUNDJOB_INFO 365
101. SYS_BLOBS 366
102. SYS_CARDINAL 366
103. SYS_CATALOGS 367
104. SYS_CHECKSTRINGS 367
105. SYS_COLUMNS 367
106. SYS_COLUMNS_AUX 368
107. SYS_DL_REPLICA_CONFIG 369
108. SYS_DL_REPLICA_DEFAULT 369
109. SYS_EVENTS 369
110. SYS_FEDT_DB_PARTITION 表の定義 370
111. SYS_FEDT_TABLE_PARTITION 表の定義 370
112. SYS_FORKEYPARTS 371
113. SYS_FORKEYS 372
114. SYS_INFO 372
115. SYS_KEYPARTS 372
116. SYS_KEYS 373
117. SYS_LOGPOS 表の定義 374
118. SYS_PROCEDURES 375
119. SYS_PROCEDURE_COLUMNS 375
120. SYS_PROPERTIES 376
121. SYS_RELAUTH 376
122. SYS_SCHEMAS 377
123. SYS_SEQUENCES 377
124. SYS_SERVER 表の定義 377
125. SYS_SYNC_REPLICA_PROPERTIES . . . 378
126. SYS_SYNONYM 378
127. SYS_TABLEMODES 378
128. SYS_TABLES 379
129. SYS_TRIGGERS 380
130. SYS_TYPES 380
131. SYS_UROLE 381
132. SYS_USERS 381
133. SYS_VIEWS 382
134. SYS_BULLETIN_BOARD 382
135. SYS_PUBLICATION_ARGS 383
136. SYS_PUBLICATION_REPLICA_ARGS 383
137. SYS_PUBLICATION_REPLICA_STMTARGS 383
138. SYS_PUBLICATION_REPLICA_STMTS 384
139. SYS_PUBLICATION_STMTARGS 384
140. SYS_PUBLICATION_STMTS 385
141. SYS_PUBLICATIONS 385
142. SYS_PUBLICATIONS_REPLICA 386
143. SYS_SYNC_BOOKMARKS 386
144. SYS_SYNC_HISTORY_COLUMNS 387
145. SYS_SYNC_INFO 387
146. SYS_SYNC_MASTER_MSGINFO 388
147. SYS_SYNC_MASTER_
RECEIVED_BLOB_REFS 389
148. SYS_SYNC_MASTER_RECEIVED
_MSGPARTS 389
149. SYS_SYNC_MASTER_RECEIVED_MSGS 389
150. SYS_SYNC_MASTER_STORED_BLOB_REFS 390
151. SYS_SYNC_MASTER_STORED_MSGPARTS 390
152. SYS_SYNC_MASTER_STORED_MSGS 391
153. SYS_SYNC_MASTER_SUBSC_REQ 391
154. SYS_SYNC_MASTER_VERSIONS 391
155. SYS_SYNC_MASTERS 392
156. SYS_SYNC_RECEIVED_BLOB_ARGS 392
157. SYS_SYNC_RECEIVED_STMTS 393
158. SYS_SYNC_REPLICA_MSGINFO 394
159. SYS_SYNC_REPLICA_RECEIVED_
BLOB_REFS 395
160. SYS_SYNC_REPLICA_RECEIVED_
MSGPARTS 395
161. SYS_SYNC_REPLICA_RECEIVED_MSGS 396
162. SYS_SYNC_REPLICA_STORED_BLOB_REFS 396
163. SYS_SYNC_REPLICA_STORED_MSGS 396
164. SYS_SYNC_REPLICA_STORED_MSGPARTS 397
165. SYS_SYNC_REPLICA_VERSIONS 397
166. SYS_SYNC_REPLICAS 398
167. SYS_SYNC_SAVED_BLOB_ARGS 398
168. SYS_SYNC_SAVED_STMTS 398
169. SYS_SYNC_TRX_PROPERTIES 399
170. SYS_SYNC_USERMAPS 399
171. SYS_SYNC_USERS 400
172. COLUMNS 400
173. SERVER_INFO 401
174. TABLES 401
175. USERS 402
176. SYNC_FAILED_MESSAGES 403
177. SYNC_FAILED_MASTER_MESSAGES 403
178. SYNC_ACTIVE_MESSAGES 404
179. SYNC_ACTIVE_MASTER_MESSAGES 404
180. SYS_LOG 表の定義 405
181. レコード ID の説明 406
182. 行データ用の DATA 型の説明 407
183. DDL データ用の DATA 型の説明 408
184. SYNC_SETUP_CATALOG のエラー・コード 411
185. SYNC_REGISTER_REPLICA のエラー・コード 412
186. SYNC_UNREGISTER_REPLICA のエラー・コード 414
187. SYNC_REGISTER_PUBLICATION のエラー・コード 415
188. SYNC_UNREGISTER_PUBLICATION のエラー・コード 416
189. CREATE PROCEDURE
SYNC_SHOW_SUBSCRIPTIONS の結果セット 417
190. SYNC_SHOW_SUBSCRIPTIONS のエラー・コード 417

191. SYNC_SHOW_REPLICA_SUBSCRIPTIONS の結果セット 418
192. SYNC_SHOW_REPLICA_SUBSCRIPTIONS のエラー・コード 418
193. SYNC_DELETE_MESSAGES のエラー・コード 419
194. SYNC_DELETE_REPLICA_MESSAGES のエラー・コード 420
195. 各種イベント 424
196. SYS_EVENT_ERROR の原因となるエラー 432
197. SYS_EVENT_MESSAGES の原因となる警告 432

本書について

本書では、リレーショナル・データベース・サーバーの理論および SQL プログラミング言語についての概要を述べます。また、本書には、IBM® solidDB® がサポートするすべての SQL ステートメントの構文を示す付録があり、表および SQL ステートメントで使用できるデータ型についての説明もあります。

本書は、SQL 全般について理解したいユーザーも、solidDB 固有の SQL について理解したいユーザーも対象にしています。

書体の規則

solidDB の資料では、以下の書体の規則を使用します。

表 1. 書体の規則

フォーマット	**用途**
データベース表	このフォントは、すべての通常テキストに使用します。
NOT NULL	このフォントの大文字は、SQL キーワードおよびマクロ名を示しています。
`solid.ini`	これらのフォントは、ファイル名とパス式を表しています。
`SET SYNC MASTER YES;` `COMMIT WORK;`	このフォントは、プログラム・コードとプログラム出力に使用します。SQL ステートメントの例にも、このフォントを使用します。
run.sh	このフォントは、サンプル・コマンド行に使用します。
TRIG_COUNT()	このフォントは、関数名に使用します。
java.sql.Connection	このフォントは、インターフェース名に使用します。
LockHashSize	このフォントは、パラメーター名、関数引数、および Windows® レジストリー項目に使用します。
argument	このように強調されたワードは、ユーザーまたはアプリケーションが指定すべき情報を示しています。
管理者ガイド	このスタイルは、他の資料、または同じ資料内の他の章の参照に使用します。新しい用語や強調事項もこのように記述します。
ファイル・パス表示	特に明記していない場合、ファイル・パスは UNIX® フォーマットで示します。スラッシュ (/) 文字は、インストール・ルート・ディレクトリーを表します。

表 1. 書体の規則 (続き)

フォーマット	用途
オペレーティング・システム	資料にオペレーティング・システムによる違いがある場合は、最初に UNIX フォーマットで記載します。UNIX フォーマットに続いて、小括弧内に Microsoft® Windows フォーマットで記載します。その他のオペレーティング・システムについては、別途記載します。異なるオペレーティング・システムに対して、別の章を設ける場合があります。

構文表記法の規則

solidDB の資料では、以下の構文表記法の規則を使用します。

表 2. 構文表記法の規則

フォーマット	用途
`INSERT INTO` *`table_name`*	構文の記述には、このフォントを使用します。置き換え可能セクションには、この フォントを使用します。
`solid.ini`	このフォントは、ファイル名とパス式を表しています。
[ ]	大括弧は、オプション項目を示します。太字テキストの場合には、大括弧は構文に組み込む必要があります。
|	垂直バーは、構文行で、互いに排他的な選択項目を分離します。
{ }	中括弧は、構文行で互いに排他的な選択項目を区切ります。太字テキストの場合には、中括弧は構文に組み込む必要があります。
...	省略符号は、引数が複数回繰り返し可能なことを示します。
. . .	3 つのドットの列は、直前のコード行が継続することを示します。

1 データベース概念

solidDB のようなリレーショナル・データベース・サーバーについて、まだあまり詳しくない場合は、この章を参照してください。

この章では、以下の概念について説明します。

- リレーショナル・データベース
  - 表、行、および列
  - 異なる表のデータの関連付け
- マルチユーザー機能/並行性制御とロック方式
- クライアント/サーバー・アーキテクチャー
- トランザクション
- トランザクション・ロギングおよびリカバリー

リレーショナル・データベース

表、行、および列

solidDB ファミリーを含めたリレーショナル・データベース・サーバーのほとんどは、構造化照会言語 (SQL) と呼ばれるプログラミング言語を使用します。SQL は、表形式の情報を照会および更新できるように設計された集合指向のプログラミング言語です。この章では、表、および表における情報の表現について説明します。SQL 言語の詳しい構文については、このマニュアルで後述します。

すべての情報は表に格納されます。表は行と列に分けられています (SQL の理論に詳しい人は、列を「属性」、行を「タプル」と呼びますが、ここではよく知られた「列」と「行」という用語を使用します。また、「レコード」と「行」という用語を同じ意味で使用します)。各データベースには 0 個以上の表が含まれています。ほとんどのデータベースは多数の表で構成されています。表の一例を以下に示します。

表 3. データベース表の例

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

この表は 3 行のデータを含んでいます (「ID」、「NAME」、「ADDRESS」というラベルの付いた最上行は、便宜上追加したものです。データベース内の実際の表にこのような行はありません)。この表には 3 つの列があります (ID、NAME、および ADDRESS)。

SQL には、表の作成、表への行の挿入、表内のデータの更新、表からの行の削除、および表内の行の照会を行うためのコマンドが用意されています。

SQL での表は、C のようなプログラミング言語での配列とは異なり、均質ではありません。SQL では、ある列のデータ型 (INTEGER など) が隣接する列のデータ型 (20 文字の配列を意味する CHAR(20) など) とまったく異なっている場合があります。

表の行数は一定ではありません。行はいつでも挿入および削除することができます。最大行数分のスペースを事前に割り振る必要はありません (どのデータベース・サーバーにも処理できる行数の上限が設けられています。例えば、32 ビットのオペレーティング・システムで稼働するデータベース・サーバーのほとんどには、約 20 億行という制限があります。ほとんどのアプリケーションでは、必要となりそうな行数がこの上限をはるかに下回ります)。

各行 (レコード) には、少なくとも 1 つのユニークな値または値の組み合わせが必要です。上記の表に David Jones という名前の作曲家が 2 人含まれていて、その一方の住所のみを更新する必要がある場合は、なんらかの方法で両者を区別する必要があります。場合によっては、どの 1 つの列も固有値がないのに、列の組み合わせがユニークとなることがあります。例えば、名前の列だけでは不十分でも、名前と住所の組み合わせがユニークとなる可能性があります。ただし、すべてのデータを事前に把握しておかないと、各値がユニークであることを絶対的に保証することは困難です。ほとんどのデータベース設計者は、各レコードを一意的に、かつ容易に識別することのみを目的とした「余分な」列を追加します。例えば上記の表の ID 番号はユニークです。したがって、レコードを実際に更新または削除する際には、ユニークでない可能性がある名前のような値を使用するのではなく、ユニークな ID でレコードを識別します (例えば「... WHERE id = 1」と指定)。

異なる表のデータの関連付け

SQL で一度に処理できる表が 1 つだけだとしたら、便利ですが、あまり強力ではないでしょう。SQL とリレーショナル・データベースの真の能力は、有用な方法で表を相互に関連付けることができ、SQL 照会で複数の表からデータを収集し、そのデータを論理的な方法で表示できるという点にあります。

銀行の例を使用して、複数の表を使用することがどのくらい有用かを示します。

銀行の各顧客は複数の口座を持っていることがあります。1 人で持つことができる口座の数に制限はありません。 1 人の顧客が当座預金口座、普通預金口座、譲渡性預金証書、住宅ローン、クレジット・カードを持つことができます。さらに、1 人が同じ種類の口座を複数持つこともできます。例えば、顧客は、退職金用の普通預金口座を 1 つと、娘の教育資金用に 1 つの普通預金口座 (同じ種類の口座) を持つことができます。顧客と口座の「関係」は、1 人が複数の口座を持つことができる「1 対多」の関係であると言えます。

1 人が持つことのできる口座の数に制限がないため、すべての可能な口座の組み合わせを処理できるレコード構造を事前に設計することはできません。また、実際に所有できる最大数の口座を保持するレコード構造を作成した場合、大量のスペースが浪費されます。ここで、1 人の銀行顧客とその口座に関するすべての情報を保持する単一の表を構築したとします。最初のドラフトは、以下のようになります。

```
顧客の ID 番号
顧客名
顧客の住所
当座預金口座 1 の ID
当座預金口座 1 の 残高
CD 1 の ID
CD 1 の 残高
CD 2 の ID
CD 2 の 残高
...
```

ここでわかるように、各顧客が所有している口座の数には明確な制限がないため、どこで止めればよいかがわかりません。

別の解決策として、口座ごとに 1 つずつ、複数のレコードを作成し、各口座に顧客情報を複製する方法があります。その場合の表は、以下のようになります。

```
顧客名
顧客の住所
口座の ID
口座の残高
```

顧客が複数の口座を持っていても、口座ごとに完全なレコードを作成するだけです。これは合理的に機能しますが、すべての単一口座レコードに顧客のすべての情報が保持されることになります。これは、ストレージ・スペースを浪費します。また、顧客が引っ越した場合、顧客の住所を更新することが困難です (複数の場所で住所を更新する必要があります)。

solidDB のようなリレーショナル・データベースは、この問題を解決するように設計されています。顧客用に 1 つの表を作成し、口座用に別の表を作成します。(実際の銀行では、多くの場合、当座預金口座用と普通預金口座用など、口座を複数の表に分割します。) 次に、顧客と口座の間に「リンク」を作成します。これによって、スペースを浪費せずに、完全な情報を使用可能にできます。

既に説明したように、この作成者の例では、すべてのレコードにそのレコードを識別するための固有値があります。固有値は、通常、単なる整数です。このユニーク整数を使用して、顧客と口座を「関連付ける」ことができます。これについては、9 ページの『2 章 SQL の概要』で詳しく説明します。

顧客の口座を作成するとき、口座情報の一部として顧客の ID 番号を保管します。具体的には、口座表の各行に customer_id 値があり、この customer_id 値が、その口座を所有する顧客の ID と一致します。Smith は顧客 ID 1 で、Smith の各口座の customer_id フィールドは 1 です。つまり、以下のようにすることで、 Smith のすべての口座のレコードを検索できます。

1. Smith のレコードを顧客表で参照します。
2. Smith のレコードが検索されたら、そのレコードの ID 番号を調べます。(Smith の場合、ID は 1 です。)
3. 次に、口座表で、customer_id フィールドの値が 1 であるすべての口座を参照します。

これは子供が学校に行くときに、子供の額に自宅の電話番号のコピーを貼り付けるようなものです。緊急事態が発生して、学校に子供を迎えに行ってもらうようタクシーの運転手に頼む場合、タクシーの運転手に電話番号を伝えれば、運転手は学校ですべての子供を調べ、その電話番号を付けている子供を探すことができます。(効

率的ではありませんが、目的は達成されます。) 親の ID 番号を知ることで、すべての子を特定できます。逆に、子がわかっていれば、親を特定できます。例えば、学校から離れた遠足の途中で子供が迷子になった場合、親切な人が子供の額に付いている電話番号を読み、親に電話をすることができます。

このように、親と子は物理的な接触なしに、相互に結び付けられます。ID 番号 (または電話番号) を持つだけで、どの子がどの親に属し、どの親がどの子に属しているかを判別できます。この手法は、子の数にかかわらず機能します。

リレーショナル・データベースは、これと同じ手法を使用します。結合操作は、2 つの表に限られないことに注意してください。ほぼ任意数の表の結合が可能です。銀行の例を現実的に広げ、発行された小切手に関する情報を保持する別の表「checks」を作成した場合を考えてみます。このとき、顧客から口座の 1 対多の関係だけではなく、当座預金口座からその口座で発行されたすべての小切手への 1 対多の関係もできます。顧客が複数の当座預金口座を持っていても、顧客が発行したすべての小切手をリストする照会を作成できます。

クライアント/サーバー・アーキテクチャー

solidDB では、クライアント/サーバー・モデルを使用します。クライアント/サーバー・モデルでは、単一の「サーバー」が 1 つ以上の「クライアント」からの要求を処理できます。これは、レストランの仕事に大変よく似ています。1 人のウエイター兼料理人が多数の顧客からの要求を処理できます。

クライアント/サーバー・データベース・モデルでは、サーバーは、データの効率的な保管とリトリーブの方法を知っている特殊なコンピューター・プログラムです。一般に、サーバーは以下に示す 4 つの基本的なタイプの要求を受け入れます。

- 新しい情報部分の挿入
- 既存の情報部分の更新
- 既存の情報部分のリトリーブ
- 既存の情報部分の削除

サーバーは、ほとんどすべてのタイプのデータを保管できますが、一般に、データの「意味」は知りません。サーバーは会計処理や在庫などの「ビジネスの問題」に関して、ほとんど知らないか、まったく知らない場合がよくあります。特定の情報部分が在庫レコードなのか、銀行預金の説明なのか、それとも「American Pie」という歌のデジタル化コピーなのかも分かりません。

「クライアント」は、特定のビジネスの問題とデータの「意味」に関して、多少の知識を持っている必要があります。例えば、会計処理について何らかのことを知っているクライアント・プログラムを作成する場合があります。例えば、クライアント・プログラムは支払の遅延に対する利子の計算方法を知っているかもしれません。あるいは、クライアントはデータの特定の部分が歌であることを認識したり、デジタル・データをアナログのオーディオ出力に変換したりする場合もあります。

作業の「クライアント」と「サーバー」の両方の部分を実行する単一のプログラムを作成することもできます。デジタル化された音楽を読み取って再生するプログラムは、そのデータをディスクに保管し、要求に応じて検索することもできます。ただし、すべての会社が独自のデータ保管ルーチンとデータ・リトリーブ・ルーチン

を作成するのは、あまり効率的ではありません。通常は、必要を満たすだけの汎用性があり、比較的にパフォーマンスが高い既製のデータ保管ソリューションを購入した方が効率的です。

マルチユーザー機能

クライアント/サーバー・アーキテクチャーの重要な利点は、通常の場合、複数のクライアントに対処するのが容易になることです。 solidDB では、ほとんどのリレーショナル・データベース・サーバーと同様に、複数のユーザーが 1 つの表のデータにアクセスできます。

2 人のユーザーが同じデータを更新しようとすると、潜在的な危険性があります。更新が同じものでない場合、1 人のユーザーの更新が他のユーザーの更新を上書きする可能性があります。これを防止するために、solidDB は並行性制御メカニズムを使用します。詳しくは、「*IBM solidDB 管理者ガイド*」を参照してください。

トランザクション

SQL では、複数のステートメントをトランザクションと呼ばれる「アトミックな」(分割できない) 作業単位にまとめることができます。例えば、食料品店で客が小切手を切った場合は、客の銀行口座から代金が引き落とされるのと同時に、店の銀行口座に代金が入金される必要があります。客が代金を支払っても店が受け取らなければ意味がありません。また、店が支払を受けても客の口座から代金が引き落とされなければ意味がありません。2 つの操作 (店の口座への入金と客の口座からの出金) のいずれかが失敗した場合は、もう一方の操作も失敗するようにする必要があります。 両方のステートメントが同じトランザクションにあれば、どちらかのステートメントが失敗した場合に ROLLBACK コマンドを使用してトランザクション開始前の状態に戻すことができます。これにより、トランザクションが半分だけ成功する事態を回避できます。この会計トランザクションの両方の操作が成功した場合は、データベース・トランザクションも成功した状態にする必要があります。成功したトランザクションは、コマンド COMMIT WORK で保存されます。以下に、単純化した例を示します。

```
COMMIT WORK; -- 前のトランザクションを終了。
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;
```

トランザクション・ロギングおよびリカバリー

市販のデータベース・サーバーを購入する大きな利点の 1 つは、このようなサーバーのほとんどが、電源障害、ハードウェア障害、データベース・ソフトウェア自体の障害などが原因でデータベース・サーバーが予期せずシャットダウンした場合にデータを保護するように設計されていることです。

データはさまざまな方法で保護されます。ここでは、その 1 つであるトランザクション・ロギングに焦点を当てます。

バックグラウンド

ディスク・ドライブ (またはその他の永続ストレージ・メディア) にデータを書き込んでいて、突然、電源に障害が起きたとします。書き込んだデータが完全には書き込まれていないことが考えられます。例えば、「122.73」という勘定残高を書き込もうとして、電源障害のために「12」としか書き込まれませんでした。口座の金額の一部が欠落した人は、大変に不快になるでしょう。どのようにすれば、常に完全なデータが書き込まれることを保証できるでしょうか。解決方法の 1 つは、「トランザクション・ログ」と呼ばれるものを使用することです。

注:

コンピューターの世界では、さまざまなものが「ログ」と呼ばれています。例えば、solidDB はトランザクション・ログ・ファイルやエラー・メッセージ・ログ・ファイルを含む、複数のログ・ファイルを書き込みます。ここでは、少しの間、トランザクション・ログ・ファイルだけについて説明します。

前に述べたように、作業は通常、「トランザクション」単位で行われます。1 つのトランザクション全体は、コミットされるかロールバックされます。部分的なトランザクションは許されません。ここで説明する状況では、ある人の新規勘定残高のディスクへの書き込みを開始しましたが、終了する前に電源が失われたので、このトランザクションをロールバックしたいところです。既に完了して、ディスクに正しく書き込まれたトランザクションは、保存される必要があります。

ここでは、どのデータが正常に書き込まれ、どのデータが正常に書き込まれなかったかを追跡するために役立つよう、実際にデータベース表だけでなく「トランザクション・ログ」にもデータを書き込んでいます。トランザクション・ログは基本的に、実行された操作 (つまり、コミットが完了したトランザクション) を 1 つの線のように並べたものです。このファイルには、それぞれのトランザクションの終わりを示すマーカーが存在します。ファイル内の最後のトランザクションに「トランザクションの終わり」マーカーがなければ、その部分トランザクションは完了しておらず、コミットでなくロールバックを行う必要があることが分かります。

サーバーは障害の後に再始動すると、トランザクション・ログを読み取り、完了したトランザクションを 1 つずつ適用します。言い換えれば、トランザクション・ログ・ファイル内の情報を使用して、データベース内の表を更新します。これを「リカバリー」と呼びます。リカバリーが適正に行われた場合、リカバリー・プロセス自体のときに電源障害が起きても、システムは保護されます。

これは、トランザクション・ロギングによる、データ破壊からの保護の方法を完全に説明したものではありません。ここでは、サーバーがどのようにしてトランザクションが失われないようにするかを説明しました。しかし、サーバーがディスク・ドライブ内の表にレコードを書き込んでいる途中で書き込みの失敗が起きた場合に、データベース・ファイルが壊れないようにする方法については、実際には説明していません。そのトピックはもっと高度な事項なので、ここでは説明しません。

要約

この簡単なリレーショナル・データベース紹介では、リレーショナル・データベースを使い始めるために必要な概念について説明しました。これで、以下の質問に答えられるはずです。

表、行、および列とは何ですか。

同時に複数の表のデータを処理することはできますか。

トランザクションは、どのようにしてデータの整合性を維持していますか。

トランザクション・データをディスク・ドライブに書き込む (「ログに記録する」) 理由は何ですか。

2 SQL の概要

この章では SQL について概説します (復習にも役立ちます)。

表、行、および列

SQL は、表形式の情報を照会および更新できるように設計された集合指向のプログラミング言語です。

すべての情報は表に格納されます。表は行と列に分けられています (SQL の理論に詳しい人は、列を「属性」、行を「タプル」と呼びますが、ここではよく知られた「列」と「行」という用語を使用します。また、「レコード」と「行」という用語を同じ意味で使用します)。各データベースには 0 個以上の表が含まれています。ほとんどのデータベースは多数の表で構成されています。表の一例を以下に示します。

表 4. データベース表の例

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

この表は 3 行のデータを含んでいます (「ID」、「NAME」、「ADDRESS」というラベルの付いた最上行は、便宜上追加したものです。データベース内の実際の表にこのような行はありません)。この表には 3 つの列があります (ID、NAME、および ADDRESS)。SQL には、表の作成、表への行の挿入、表内のデータの更新、表からの行の削除、および表内の行の照会を行うためのコマンドが用意されています。

SQL

前のトピックで示した表を作成する SQL「プログラム」全体を以下に示します。

```
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');
```

ここでは、列「id」を表の「主キー」に指定しています。これにより、各行がこの列で一意的に識別されるようにします。これ以降は、「id」の値がユニークであり、かつ必ず値が存在する (つまり NOT NULL プロパティーが設定される) ことがシステムで保証されます。

Dylan 氏が 61 Bob Street に住所を移した場合は、Dylan 氏のデータを以下のコマンドで更新できます。

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

ID フィールドは作曲家ごとにユニークであり、かつこのコマンドの WHERE 節では 1 つの ID のみが指定されているため、この更新は 1 人の作曲家のみに対して実行されます。

Beethoven 氏が死去し、そのレコードを削除する必要がある場合は、以下のコマンドを使用します。

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

最後に、表内のすべての作曲家をリストする場合は、以下のコマンドを使用します。

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

この SELECT ステートメントには、上記の UPDATE ステートメントや DELETE ステートメントとは異なり、WHERE 節が含まれていません。したがって、指定した表のすべてのレコードにこのコマンドが適用されます。こうして、この SQL ステートメントでは、表に格納されているすべての作曲家が選択され、リストされます。

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

ストリングは引用符で囲んで入力しましたが、そのストリングが引用符なしで表示されていることに注意してください。

この単純な一連のコマンドにも、SQL に関するいくつかの重要なポイントが示されています。

- SQL は比較的「高水準」の言語です。1 つのコマンドで必要な数の列を含んだ表を作成できます。同様に、1 つのコマンドでどのように複雑な更新でもほぼすべて実行できます。ここでは示していませんが、一度に複数の列を更新することも、複数の行を更新することも可能です。C や Java™ のような言語では多数のコード行を必要とする操作も、1 つの SQL コマンドで実行できます。
- 他のコンピューター言語とは異なり、SQL ではストリングを区切る際に単一引用符を使用します。例えば、'Beethoven' はストリングです。"Beethoven" は別のものになります (技術的にはこれは区切り ID ですが、この章では説明しません)。ストリング (文字配列) を二重引用符で区切り、個々の文字を単一引用符で区切る C のようなプログラミング言語に慣れている場合は、SQL の方式に合わせる必要があります。

上記の例では明確に示していませんが、基本的な SQL に関して知っておく必要があるポイントが他にもいくつかあります。

- SQL はきわめて強力な高水準言語ですが、同時にきわめて制限的な言語でもあります。SQL は表単位およびレコード単位の操作を対象として設計されています。対応している低レベルの操作はごくわずかです。例えば、ファイルを開く操作やビットを左右にシフトする操作を直接実行することはできません。また、SQL はハードウェアに依存しません。これには利点と欠点があります。SQL 照会からの

出力のフォーマットはほとんど制御できません。列の順序を選択したり、ORDER BY 節を使用して行の順序は制御したりすることはできますが、画面上のフォント・サイズを変更したり、出力の各印刷ページの最下部にページ番号を印刷したりすることはできません。SQL は C、Java、PASCAL のような完全なプログラミング言語ではありません。

- それぞれの SQL 実装には、一定のデータ型があります。solidDB (およびその他ほとんどの SQL 実装) でのデータ型には、INTEGER、CHAR (文字配列)、FLOAT (浮動小数点)、DATE、および TIME があります。
- SQL は一般には「コンパイルされる」言語ではなく「インタープリットされる」言語です。1 つ以上の SQL ステートメントを実行する場合、通常はスクリプトを読み取って実行する別のプログラムを実行します。後で使用できるように「コンパイルされたプログラム」や「実行可能プログラム」が生成されて保管されることはありません。プログラムは、実行するたびに再度インタープリットされます (ストアード・プロシージャーは再利用が可能で、必ずしも再度インタープリットされるわけではありません。ストアード・プロシージャーの簡単な説明については 177 ページの『付録 B. solidDB SQL 構文』を、詳しい説明については 27 ページの『3 章 ストアード・プロシージャー、イベント、トリガー、およびシーケンス』を参照してください)。
- SQL では、表名と列名の大/小文字を区別しません。ここで示した例では、キーワード (CREATE、INSERT、SELECT など) を大文字表記し、表名と列名を小文字で表記していますが、これは単なる表記規則であり、必要条件ではありません。
- SQL には、コマンドを 1 行に記述するか、複数行に分けて記述するかに関してそれほど細かいルールがありません。複数行のステートメントの例については、この章で後ほど示します。
- SQL コマンドは、照会内に照会がネストした複数の「階層」があると非常に複雑になる可能性があります。複雑な照会の記述方法を理解するのはかなり困難です。また、他人が記述した照会を理解することも同じように困難です。どのプログラミング言語でも同じですが、作成したコードは文書化することを推奨します。
- コードの文書化を支援するために、SQL では「コメント」を付けられるようになっています。コメントは人間が読むためのもので、SQL インタープリターではスキップされます。コメントを作成する場合は、行の先頭に 2 つのダッシュを入力します。そこから行末までのすべての文字が無視されます (「オプティマイザー・ヒント」の場合は例外ですが、これは別の高度なトピックであり、この章では取り上げません)。

SQL の数学的起源

リレーショナル・データベースと SQL は本来、集合論という数学的概念にある程度基づいています。集合論に関する知識があると、リレーショナル・データベースの仕組みを理解しやすくなります。集合論に関する知識がなくても心配する必要はありません。これはリレーショナル・データベースと SQL を考察する方法の 1 つに過ぎません。

表は数学的な集合と考えることができ、その集合の各要素が行となります (前の例では、各人物、つまり作曲家が集合の要素です。この表には「作曲家」という集合の要素がすべて格納されています)。数学での集合には順序がありません。同様に、

SQL では一般に表は順序付けされていないと見なされます (ただし、ディスク上のビットやバイトを見ることができたとしたら、レコードが常に特定の順序で格納されることがわかるはずです)。

この順序性の欠如は重要です。なぜなら、照会を実行するたびにその結果が異なる順序で示される可能性があるからです。1 つのディスク・ドライブに格納されている小規模なデータ・セットであれば、通常は毎回同じ行が同じ順序で表示されますが、データが複数のファイルやディスク・ドライブに散在している場合は必ずしもそのようにはなりません。

SQL は集合指向の言語であるため、SQL を使用して UNION (2 つの入力の集合を組み合わせて 1 つの出力の集合にする操作) などの集合指向の操作を実行することができます。ただし、UNION などの操作では、集合が互いに一致していることが必要です。つまり列数が同じで、かつ対応する列のデータ型が同じ (または互換性がある) でなければなりません。例えば、集合 1 の最初の列の型が DATETIME で、集合 2 の最初の列の型が INTEGER である場合、UNION は実行できません。

繰り返しますが、集合論になじめなくても心配しないでください。これはリレーショナル・データベースの 1 つの見方に過ぎません。

関連データを持つ表の作成

前の章で説明したように、銀行の各顧客は複数の口座を持っていることがあります。顧客と口座の「関係」は、1 人が複数の口座を持つことができる「1 対多」の関係であると言えます。

1 人が持つことのできる口座の数に制限がないため、すべての可能な口座の組み合わせを処理できるレコード構造を事前に設計することはできません。

IBM Corporation のデータベースのようなリレーショナル・データベースは、この問題を解決するように設計されています。顧客用に 1 つの表を作成し、口座用に別の表を作成します。(実際の銀行では、多くの場合、当座預金口座用と普通預金口座用など、口座を複数の表に分割します。) 次に、顧客と口座の間に「リンク」を作成します。これによって、スペースを浪費せずに、完全な情報を使用可能にできます。

既に説明したように、この作成者の例では、すべてのレコードにそのレコードを識別するための主キーがあります。これは、通常、単なる整数です。このユニーク整数を使用して、顧客と口座を「関連付ける」ことができます。以下は、顧客表を作成してデータを設定するコマンドです。

```
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');
```

Smith と Jones という 2 人の顧客を挿入しました。次に、口座表を作成します。

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

ここで、列 *customer_id* を、顧客表を指す「外部キー」に設計しました (REFERENCES キーワードで示されています)。この列の値は、「customers」表の対応する顧客行の「id」値 (主キー) とまったく同じになるはずです。このようにして、口座行と顧客行を関連付けます。信頼性の高い方法で、このような関係を保守できるようにするデータベースの機能を、「参照整合性」と呼び、このような関係の定義に使用される、対応する SQL 構文要素は「参照整合性制約」と呼ばれます。参照整合性について詳しくは、『118 ページの『参照整合性』』を参照してください。

顧客 Smith には 2 つの口座があり、顧客 Jones には 1 つの口座があります。

```
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);
```

Smith には 2 つの口座があるため、 Smith の各口座の *customer_id* フィールドは 1 になります。これは、以下のようにすることで、ユーザーが Smith の口座のレコードのすべてを検索できるという意味になります。

1. Smith のレコードを顧客表で参照します。
2. Smith のレコードが検索されたら、そのレコードの ID 番号を調べます。(Smith の場合、ID は 1 です。)
3. 次に、口座表で、*customer_id* フィールドの値が 1 であるすべての口座を参照します。

これは子供が学校に行くときに、子供の額に自宅の電話番号のコピーを貼り付けるようなものです。緊急事態が発生して、学校に子供を迎えに行ってもらうようタクシーの運転手に頼む場合、タクシーの運転手に電話番号を伝えれば、運転手は学校ですべての子供を調べ、その電話番号を付けている子供を探すことができます。(効率的ではありませんが、目的は達成されます。) 親の ID 番号を知ることで、すべての子を特定できます。逆に、子がわかっていれば、親を特定できます。例えば、学校から離れた遠足の途中で子供が迷子になった場合、親切な人が子供の額に付いている電話番号を読み、親に電話をすることができます。

このように、親と子は物理的な接触なしに、相互に結び付けられます。ID 番号 (または電話番号) を持つだけで、どの子がどの親に属し、どの親がどの子に属しているかを判別できます。この手法は、子の数にかかわらず機能します。

リレーショナル・データベースは、これと同じ手法を使用します。顧客表と口座表を作成したので、次は、各顧客と顧客が持っている各口座を表示できます。そのためには、SQL プログラマーが「結合」操作と呼ぶ方法を使用します。SELECT ステートメントの WHERE 節で、口座の *customer_id* 番号が顧客の ID 番号と一致するレコードのペアを「結合」します。

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

この照会の出力は、以下のようになります。

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

複数の口座を持っている利用者は、自分のすべての口座の合計金額を知りたいと思うことがあります。コンピューターは、以下の照会を使用して、この情報を提供します。

```
SELECT customers.id, SUM(balance)
  FROM customers, accounts
  WHERE accounts.customer_id = customers.id
  GROUP BY customers.id;
```

この照会の出力は、以下のようになります。

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

ここでは、Smith は 1 回だけ表示され、すべての口座の合計金額が表示されていることに注意してください。

この照会では、GROUP BY 節と、SUM() という集約関数が使用されています。GROUP BY 節のトピックは、この簡単な SQL 紹介で扱うには複雑です。この照会は、SQL が単一ステートメントで実行できる便利な作業のタイプを少し体験してみることを目的としています。C などの言語で同じ結果を取得するには、多くのステートメントが必要です。

結合操作は、2 つの表に限られないことに注意してください。ほぼ任意数の表の結合が可能です。銀行の例を現実的に広げ、発行された小切手に関する情報を保持する別の表「checks」を作成した場合を考えてみます。このとき、顧客から口座の 1 対多の関係だけではなく、当座預金口座からその口座で発行されたすべての小切手への 1 対多の関係もできます。顧客が複数の当座預金口座を持っていても、顧客が発行したすべての小切手をリストする照会を作成できます。

表の別名

SQL では、一部の照会で表名の代わりに「別名」を使用することができます。別名は、単に便利なオプションとして使用されることもありますが、照会によっては別名が実際に必要となることがあります (理由についてはここでは説明しません)。この章の後半に別名を必要とする例がいくつか示されているため、ここでは別名について概説します。以下の照会は前に示した照会とほぼ同じですが、表の別名として accounts 表に「a」および customers 表に「c」を追加しています。

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

このように、別名は「FROM」節で定義し、照会内の別の場所 (この場合は WHERE 節) で使用します。

副照会

SQL では、1 つの照会に「副照会」と呼ばれる別の照会を含めることができます。

銀行の例に戻ると、時間の経過とともに、口座を追加で開く顧客がいる一方で、口座を閉じる顧客も出てきます。場合によっては、ある顧客が徐々に口座を閉じて口座をいっさい持たない状態になることもあります。例えば、この例の銀行が、口座を持たない顧客のレコードを削除できるように、該当する顧客を識別するとします。口座を持たない顧客を識別する方法の 1 つは、副照会と EXISTS 節を使用することです。

この操作を試すには、口座を持たない顧客を作成する必要があります。

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

口座を持たないすべての顧客をリストする前に、口座を持つすべての顧客をリストしてみましょう。

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

副照会 (「内部照会」とも呼ばれます) は、括弧で囲まれた照会です。内部照会は、外部照会によって選択された各レコードに対して 1 回ずつ実行されます (これは、別のプログラミング言語のネストされたループによく似ていますが、SQL ではネストされたループを 1 つのステートメントで実行できます)。外部ループが処理している特定の顧客に口座がある場合は、その口座レコードが外部照会に返されます。

外部照会の「EXISTS」節の事実上の意味は、「必要な情報は、返されるレコードの値ではなく、レコードが存在するかどうか」ということです。したがって、顧客に口座がある場合は、EXISTS から TRUE が返されます。顧客に口座がない場合は FALSE が返されます。口座の数が複数であるか 1 つであるかは、EXISTS 節に必要な情報ではありません。口座に含まれる値も重要ではありません。EXISTS に必要な情報は、「1 つ以上のレコードがあるか」ということだけです。

こうして、ステートメント全体で 1 つ以上の口座を持つ顧客がリストされます。所有する口座の数に関係なく (少なくとも 1 つの口座を持っていれば)、顧客は 1 回だけリストされます。

では、口座を持たない顧客をすべてリストしてみましょう。

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

キーワード NOT を追加するだけで照会の意味が逆になります。

副照会自体に副照会を組み込むこともできます。事実上、副照会はほぼ任意の深さにネストできます。

各データ型にどの形式を使用するか

これまでに述べたように、SQL では値を特定の方法で表現する必要があります。例えば、文字ストリングは単一引用符で区切る必要があります。

その他の値も、正しくフォーマット設定する必要があります。必要となる正確なフォーマットは、データ型によって異なります。文字 (CHAR) データ型以外のいくつかのデータ型も、ユーザーの入力値を区切るために単一引用符を必要とします。

以下のいくつかの例は、solidDB がサポートしている大部分のデータ型について、入力データのフォーマットの設定方法を示しています。これは、読者が希望すれば実行できるよう、単純な SQL スクリプトの形で示してあります。このスクリプトでは、多数のコマンドが複数の行に分割されていることに注意してください。これは、SQL では、まったく合法です。そのために、実際の ANSI 規格の SQL では、各ステートメントの末尾にセミコロンは必要ありませんが、大部分の SQL インタープリターは、各 SQL ステートメントを分離するためにセミコロンを予期しています。

```
CREATE TABLE one_of_almost_everything (
 int_col INTEGER,
 float_col FLOAT,
 string_col CHAR(20),
 wide_string_col WCHAR(20), -- 「wide」はユニコードなどのワイド文字を意味します。
 varchar_col VARCHAR, -- 幅を指定する必要がないことに注意してください。
 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'
);
```

上記のように、タイム・スタンプ値は「最上位」桁から「最下位」桁への順序で入力されます。同様に、日付と時刻の値も最上位桁から最下位桁への順に入力されます。また、これら 3 つのデータ型 (タイム・スタンプ、日付、時刻) はすべて、句読点を使用して個々のフィールドを分離します。

特定のフォーマットを必要とする理由は、他の可能なフォーマットの中には、意味が未確定のものがあるからです。例えば、米国内の人間にとって ’07-04-1776’ は 1776 年 7 月 4 日を意味します。なぜなら、アメリカ人は通常、日付を ’mm-dd-yyyy’ (または ’mm/dd/yyyy’) のフォーマットで書くからです。しかし、ヨーロッパ出身の人間にとって、この日付は明らかに 4 月 7 日であり、7 月 4 日ではありません。なぜなら、ほとんどのヨーロッパ人は日付を ’dd-mm-yyyy’ のフォーマットで書くからです。フォーマットの数が多すぎるという問題は、さらに別のフォーマットを追加することでは、うまく解決できないように思えるかもしれませんが、最上位桁から始まり、一貫して最下位桁に向かって進むという SQL の手法には、いくつかの利点があります。第 1 に、3 つのデータ型 (日付、時刻、およびタイム・スタンプ) のすべてが、同じルールに従うことを意味します。第 2 に、日付フォーマットと時刻フォーマットは、どちらもタイム・スタンプ・フォーマット

の完全なサブセットです。第 3 に、別のフォーマットを覚えなければならないとしても、そのルールはかなり単純であり、「西側」の言語で数値を書く (左端が最上位桁になる) 方法と整合しています。最後に、明らかに既存のフォーマットと互換性がないことにより、ある人が誤って 1 つの日付 (例えば '07-04-1776') を書き、それをマシンに別の日付として解釈させる可能性がありません。

BLOB (またはバイナリー・データ型)

これまで、人間によって読み取られることを意図したデータを保管するデータ型について、説明してきました。一部のデータ型は、人間によって直接読み取られることを意図したものでなくても、データベース内に保管できます。例えば、デジタル・カメラの画像や CD からの歌などは、一連の数値として保管されます。これらの数値は、人間に対してほとんど意味を成しません。しかし、デジタル化した画像や音声は BINARY データとして保管できます。solidDB は、3 つのバイナリー・データ型をサポートしています。BINARY、VARBINARY、および LONG VARBINARY (または BLOB) です。

ほとんどの場合、バイナリー・データの読み書きには、C プログラムから ODBC (Open Database Connectivity) API を使用するか、Java プログラムから JDBC API を使用します。しかし、SQL ステートメントを実行するユーティリティーを使用して、バイナリー・フィールドにデータを挿入することができます。バイナリー・フィールドに値を挿入するには、その値を、単一引用符で囲んだ一連の 16 進数として表現する必要があります。例えば、値が 1、9、11、255 である一連のバイトを 1 つのバイナリー・フィールドに挿入したい場合は、以下を実行します。

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

このコマンドはサーバーに、値を型 VARBINARY に CAST するよう指示するので、サーバーは自動的に、ストリングをストリング・リテラルでなく、一連の 16 進数として解釈します。

ストリング・リテラルを直接挿入することもできます。以下に例を示します。

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

データを solsql (SQL ステートメントを実行するための solidDB ユーティリティー) によってリトリーブすると、バイナリー列からの戻り値は、その値が当初に 16 進数として入力されたかどうかにかかわりなく、16 進数で表現されます。このため、値 'Thank you' を挿入した後、この値を表から選択すると、以下のように表示されます。

```
5468616E6B20796F75
```

ここで、54 は大文字の「T」を表し、68 は小文字の「h」、61 は小文字の「a」、6E は小文字の「n」をそれぞれ表しています。

長い値の場合は、最初のいくつかの数字だけが表示されることにも注意してください。

NULL IS NOT NULL (つまり、「上記のどれでもないことを SQL で何というか」)

フォームに完全に記入するのに十分な情報を持っていない場合もあります。SQL では、キーワード NULL を使用して、「不明」または「値なし」を表します。(これは、C などのプログラミング言語における NULL の意味と異なります。) 例えば、ジョニ・ミッチェルに関するレコードを作曲者の表に挿入する場合、ジョニ・ミッチェルの住所が分からなければ、以下を実行できます。

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

address フィールドを指定しなければ、このフィールドにはデフォルトで NULL が格納されます。

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

NULL に関する情報を提供するために、また、SQL コードを読む練習として、NULL の説明をコメント付きのサンプル・プログラムとして作成しました。それを、ここに示します。実行する準備ができたら、SQL を実行するプログラム (例えば、solidDB Development Kit に添付されている solsql ユーティリティーなど) の中にこの一部または全部を単純にカット・アンド・ペーストしてください。(solsql について詳しくは、「*IBM solidDB 管理者ガイド*」を参照してください。)

```
-- このサンプル・スクリプトは、値 NULL の通常と異なる特性の
-- いくつかを示しています。

-- すべてのデータ型のデータに NULL を含めることができます。
-- 例えば、型 INTEGER の列には、
-- 有効な整数値だけでなく、NULL も含めることができます。

-- 実際の試行用のセットアップ...
CREATE TABLE table1 (x INTEGER, name CHAR(30));

-- 値 NULL は「値が存在しない」ことを意味します。
-- NULL はゼロや空ストリングと同じものではありません。
-- (また、C などのプログラミング言語におけるポインター値
-- でもありません。)
-- これを示すために、ここでは 3 つの行を挿入します。そのうちの 1 行は
-- 「通常」の値を持ち、1 行は 0 と空ストリングを持っています。
-- 残る 1 行は、2 つの NULL 値を持っています。
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);
-- これは、0 が入っている行だけを返し、
-- NULL が入っている行を返しません。
SELECT * FROM table1 WHERE x = 0;
-- これは空ストリングが入っている行だけを返し、
-- NULL が入っている行を返しません。
SELECT * FROM table1 WHERE name = '';


-- NULL が他の値に一致しないことは驚くことではありません。
-- 真に驚くべきことは、NULL がそれ自体にさえ一致しないことです。
-- (数学者に言わせれば、NULL は反射律「a = a」に
-- 違反しているのです!)
SELECT * FROM table1 WHERE x = x;

-- NULL は NULL に等しくないとなると、次の照会は何を返すでしょうか。
SELECT * FROM table1 WHERE x != x;


-- 同様に、普通なら次の式は常に
-- 真であると考えられますが、実際には
```

```
-- 常に偽になります。
SELECT * FROM table1 WHERE NULL IN (NULL, 2);


-- 結果セットには 2 が含まれます (2 は、
-- セット (NULL, 2) に入っているからです)。しかし、
-- 結果セットに NULL は含まれません。
SELECT * FROM table1 WHERE x IN (NULL, 2);


-- しかし、見つけたいものは、NULL 値を持つすべてのレコード
-- であるとします。... = NULL と言えないなら、どうすればよいでしょう
SELECT * FROM table1 WHERE x IS NULL;
-- また、反対の照会は ...
SELECT * FROM table1 WHERE x IS NOT NULL;


-- さらに実際の試行を続けるためのセットアップ...
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');


-- NULL != NULL なので、「親」レコードが NULL を持ち「子」
-- レコードが NULL を持つ場合でも、子の値は親の値に一致しません。
-- この結果セットには「Smith」が含まれますが、「Has Null」は含まれません。
SELECT p.name FROM parent p, children c
 WHERE c.parent_id = p.id;


-- 注意すべき点は、単一の NULL 以外に何も入っていない行でも
-- 行であることです。
-- 次の照会では、EXISTS 節を使用しています。
-- これは、副照会が行を返す場合、TRUE に評価
-- されます。単一の NULL 値以外に何も入っていない行でも、
-- 行であるので、
-- 副照会が単一の NULL を返した場合でも、EXISTS 節は
-- TRUE に評価されます。
-- 下記の副照会が、名前や ID でなく NULL を返す場合でも、
-- EXISTS 式は TRUE に評価され、Smith が出力されます。
SELECT name FROM parent p
 WHERE EXISTS(SELECT NULL FROM children c WHERE c.parent_id = p.id);


-- NULL != NULL であることを認識する訓練が終わったところで、
-- このパターンを壊すもので読者の頭を混乱させることにしましょう。
-- 読者の期待に反して、UNIQUE キーワードは
-- 複数の NULL 値をフィルターに掛けて除去します。
INSERT INTO table1 (x, name) VALUES (NULL, 'any name');
-- これで、表には x が NULL である複数の行が存在します。
-- しかし、UNIQUE を使用した照会では、
-- 単一の NULL 値だけが返されます。
SELECT DISTINCT x FROM table1;
-- おもしろいことに、UNIQUE 索引は
-- 単一の NULL 値のみを許可します。(主キーは
-- NULL 値を許可しないことに注意してください。)


-- 終結処理
DROP TABLE parent;
DROP TABLE children;
DROP TABLE table1;
```

NOT NULL

NULL とは逆に、NOT NULL は SQL データ制約の 1 つです。NOT NULL は、表のすべての行で、指定された列に NULL 値が許可されないことを示します。詳細と例については、 177 ページの『付録 B. solidDB SQL 構文』を参照してください。

式およびキャスト

SQL では、SQL ステートメント内で部分的に式を使用できます。例えば、以下のステートメントでは列の値を 12 で乗算します。

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

もう 1 つの例として、以下のステートメントでは組み込みの SQRT 関数を使用して「variance」という名前の列に含まれる各値の平方根を計算しています。

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

次に示す例では、「REPLACE」関数を使用して数値を米国のフォーマットからヨーロッパのフォーマットに変換します。米国のフォーマットでは、数値の小数点にピリオド文字 (.) が使用されますが、ヨーロッパではコンマ (,) が使用されます。例えば、米国では円周率の近似値が「3.14」と表記されますが、ヨーロッパでは「3,14」と表記されます。REPLACE 関数を使用して、「.」文字を「,」文字に置き換えることができます。以下の一連のステートメントはこの例を示しています。

```
CREATE TABLE number_strings (n VARCHAR);
INSERT INTO number_strings (n) VALUES ('3.14'); -- 米国のフォーマットで入力。
SELECT REPLACE(n, '.', ',') FROM number_strings; -- ヨーロッパのフォーマットで出力。
```

出力は以下のようになります。

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

ある関数から別の関数を呼び出すことができることに注意してください。以下の式では、数値の平方根を計算し、その平方根の自然対数を計算します。

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

solidDB SQL では、すべての節で完全な汎用の式を使用できるわけではありません。例えば、SELECT 節では、事前定義関数を使用できますが、各自で作成したストアード・プロシージャーを呼び出すことはできません。「foo」という名前のストアード・プロシージャーを作成しても、以下のステートメントは機能しません。

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

式を使用するときに、列に新しい名前を指定したい場合があります。例えば、以下の式を使用するとします。

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

出力される列の名前を「monthly_average」(月間平均) にするのは望ましくありません。solidDB サーバーでは、実際には式自体が列の名前として使用されます。この例の場合は、列の名前が「monthly_average * 12」となります。確かに記述的です

が、長い式の場合は煩雑になるおそれがあります。「AS」キーワードを使用すれば、出力列に特定の名前を指定できます。以下の例では、出力の列見出しを「yearly_average」にします。

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

AS 節は式だけでなく、あらゆる出力列に使用できることに注意してください。必要であれば、以下のような操作も実行できます。

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

CASE 節では、入力に基づいて出力を制御できます。以下に示す単純な例では、数値 (1 から 12) を月の名前に変換します。

```
CREATE TABLE dates (m INT);
INSERT INTO dates (m) VALUES (1);
-- ...以下続行。
INSERT INTO dates (m) VALUES (12);
INSERT INTO dates (m) VALUES (13);

SELECT
     CASE m
        WHEN 1 THEN 'January'
        -- 以下続行。
        WHEN 12 THEN 'December'
        ELSE 'Invalid value for month'
    END
  AS month_name
 FROM dates;
```

ここでは、有効な値を変換するだけでなく、エラーがあった場合に適切な出力を生成していることに注意してください。「ELSE」節を使用することで、予期しない値が入力された場合に代わりの値を指定できます。

状況によっては、値を別のデータ型にキャストしたいことがあります。例えば、BLOB データを挿入するときに、データを含んだストリングを作成し、それを BINARY 列に挿入できると便利です。キャストは以下のように使用できます。

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

このキャストによって、一連の 16 進数字で構成されるデータをストリングのように入力できます。引用符付きストリング内の 16 進数のペアは、それぞれが 1 バイトのデータを表します。8 つの 16 進数字があるので、入力は 4 バイトです。

キャストを使用して、入力だけでなく出力も変更することができます。以下に示すやや複雑なコード例では、CASE 節の式によって出力のフォーマットを「2003-01-20 15:33:40」から「2003-Jan-20 15:33:40」に変換します。

```
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;
```

この例では、dt という列の値をタイム・スタンプから VARCHAR に変換し、月の数字を月の略語に置き換えます (例えば「-01-」を「-Jan-」に置換)。
CASE/WHEN/END 構文を使用することで、各入力に対応する望ましい出力を正確に指定できます。この式はかなり複雑であるため、AS 節を使用して出力での列見出しを指定することがほとんどの場合必要となります。

行値コンストラクター

このセクションでは、あまり知られていない式のタイプの 1 つである行値コンストラクター (RVC) について説明し、より大きい、より小さいなどの、関係演算子で使用する方法を説明します。

行値コンストラクターは、以下のような、括弧で区切られた値のオーダー・シーケンスです。

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

これは、表の行が一連のフィールドで構成されるのと同様に、一連の要素/値を基に行を構成する処理と見なすことができます。

行値コンストラクターは、個別の値と同様に、比較に使用できます。例えば、以下のような式を使用できます。

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

これと同様に、以下のような式も使用できます。

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

行値コンストラクターを使用する比較は、慎重に実行する必要があります。比較の技術定義 (SQL-92 規格のセクション 8.2 (比較述部) にあります) を示す代わりに、パターンがわかるように、例とそれに似たものを示します。

以下の式は、真です。

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

上の例は単純で、式は対応する要素の各ペアで正しく、よって、RVC で真になります。以下に例を示します。

```
'Baker' > 'Alpert' かつ 'Barbara' > 'Andy' なので、
('Baker', 'Barbara') > ('Alpert', 'Andy')
```

ただし、行値コンストラクターを比較するとき、必ずしも、対応する各要素で式が真である必要はありません。行値コンストラクターでは、左にある要素ほど重要度が高くなります。そのため、以下の式も真です。

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

これらの例では、最初の RCV で最も重要度の高い要素が、2 番目の RCV の対応する要素より大きいため、残りの要素の値にかかわらず、式は真になります。同様に、以下の例では、最初の要素は同一ですが、式全体は真になります。

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

繰り返しますが、行値コンストラクターでは、左にある要素ほど重要度が高くなります。これは、複数の桁の数値を比較するのと似ています。911 のような 3 桁の数値では、百の位の数字は十の位の数字よりも重要度が高く、十の位の数字は一の位の数字より重要度が高くなります。そのため、911 のすべての桁が 199 の対応する桁より大きいわけではありませんが、数値 911 は数値 199 より大きくなります。

これは、関係する複数の列を比較するときに役に立ちます。実用的な応用として、人名の比較があります。例えば、2 つの表があり、それぞれに *lname* (姓) という列と *fname* (名) という列があるとします。ここで、Michael Morley よりも小さい名前の人をすべて検索するとします。この場合、名よりも姓の重要度を高くします。以下の名前は、正しいアルファベット順で表示されています (姓の順)。

Adams, Zelda

Morley, Michael

Young, Anna

Michael Morley より小さい名前の人をすべてリストする場合、以下のようにはしません。

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

この式を使用すると、Zelda Adams が拒否されます。この人の名 (ファーストネーム) は、アルファベット順で、Michael Morley の名より後なためです。正しい解決策の 1 つとして、行値コンストラクターのアプローチを使用する方法があります。

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

等価を使用する場合、式は、RCV のすべての要素で真である必要があります。以下に例を示します。

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

当然、比較演算子では、式は 1 つの要素でだけ真であればかまいません。

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

トランザクションの詳細

前の章で述べたように、SQL では、複数のステートメントをグループ化して、トランザクションと呼ばれる単一の「アトミック」な (分割できない) 作業の部分にすることができます。成功したトランザクションは、コマンド COMMIT WORK で保存されます。以下に、単純化した例を示します。

```
COMMIT WORK; -- 前のトランザクションを終了します。
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;
```

特定のトランザクションを保持したくない場合は、以下のコマンドを使用してトランザクションをロールバックできます。

```
ROLLBACK WORK;
```

作業を明示的にコミットまたはロールバックしなかった場合、サーバーはユーザーに代わってロールバックします。言い換えれば、保持したいデータをユーザーが (コミットすることによって) 確認しなければ、そのデータは廃棄されます。

要約

SQL およびリレーショナル・データベースについて簡単に紹介するこの章では、SQL の使用を開始するユーザーに必要な概念を説明しました。これで、以下について理解できていることになります。

表、行、および列とはなにか

表を作成する方法

表にデータを挿入する方法

表のデータを更新する方法

表からデータを削除する方法

表のデータをリストする方法

2 つの表の関連データをリストする方法

複数のステートメントをまとめて実行する方法 (すべてのステートメントが 1 つのグループとして失敗または成功するようにする)

SQL に関する追加情報の検索先

本書の別の各章で、SQL と solidDB 固有の機能に関する詳細が解説されています。ただし、本書は SQL についての完全なチュートリアルでも、包括的な解説書でもありません。SQL についての追加資料を入手するとよいでしょう。

SQL については、多数の書籍があります。それらの書籍は、solidDB の SQL の実装に固有のものではありません。大半の資料は汎用であり、ANSI 規格に準拠する

すべてのデータベース・サーバー (例えば、solidDB のデータベース・サーバーなど) に適用できます。一般的な SQL の書籍としては、以下のものがあります。

- 「*Introduction to SQL: Mastering the Relational Database Language*」(Rick van der Lans 著、Addison-Wesley 社刊)

SQL に関する ANSI 規格には、以下のものがあります。

- 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).

ANSI 規格は、www.ansi.org から購入できます。

ISO (国際標準化機構) も SQL の規格を持っています。各規格と価格のリストについては、www.iso.org を参照してください。

3 ストアード・プロシージャー、イベント、トリガー、およびシーケンス

solidDB データベースでは、アプリケーション・ロジックの一部をデータベースに移動することができる数々の機能が用意されています。これには以下の機能が含まれます。

- ストアード・プロシージャー
- 据え置きプロシージャー呼び出し (Start After Commit)
- イベント・アラート
- トリガー
- シーケンス

ストアード・プロシージャー

ストアード・プロシージャーは、solidDB データベース内で実行される単純なプログラム、つまりプロシージャーです。ユーザーは、複数の SQL ステートメントまたはトランザクション全体を含んだプロシージャーを作成し、単一の呼び出しステートメントでそのプロシージャーを実行できます。SQL ステートメントの他に、3GL タイプの制御構造を使用してプロシージャー型制御を有効にすることもできます。こうして、データ・バインドされた複雑なトランザクションをサーバー自体で実行して、ネットワーク・トラフィックを削減できます。

ストアード・プロシージャーでの実行権限を付与することで、そのプロシージャーで使用されるすべてのデータベース・オブジェクトに対する必要なアクセス権限が自動的に呼び出されます。このようにプロシージャーを介して重要なデータへのアクセスを許可することで、データベース・アクセス権限の管理を大幅に簡素化できます。

ここでは、ストアード・プロシージャーの使用方法について詳しく説明します。初めに、プロシージャーの使用に関する一般的な概念を説明します。その後のセクションでは、さらに詳しい解説を行うとともに、プロシージャーにおける各種ステートメントの実際の構文について説明します。最後に、トランザクション管理、シーケンス、およびその他の高度なストアード・プロシージャー機能について説明します。

基本的なプロシージャー構造

ストアード・プロシージャーは標準 solidDB データベース・オブジェクトであり、標準 DDL ステートメントの CREATE および DROP を使用して操作できます。

ストアード・プロシージャー定義の最も単純な形式は、以下のとおりです。

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

以下の例では、TEST というプロシージャーを作成します。

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

プロシージャーを実行するには、CALL ステートメントと、それに続けて、呼び出したいプロシージャーの名前を発行します。

```
CALL test
```

プロシージャーのネーミング

プロシージャー名は、1 つのデータベース・スキーマの中で固有でなければなりません。

データベース・オブジェクトに適用できる標準的な命名上の制限 (予約語の使用、ID の長さなど) は、すべてストアード・プロシージャー名にも適用されます。予約語の概要と完全なリストについては、 349 ページの『付録 C. 予約語』を参照してください。

パラメーター・セクション

ストアード・プロシージャーは、パラメーターを使用して呼び出し側プログラムと通信します。solidDB は、呼び出し側プログラムへ値を返すための 2 つの方式をサポートしています。最初の方式はパラメーターを使用する標準的な SQL-99 方式です。もう 1 つは solidDB 独自の方式である RETURNS で、これは結果セットを使用します。

パラメーターの使用

パラメーターを使用することは、データを返すための SQL-99の標準的な方法です。ストアード・プロシージャーは、以下の 3 つのタイプのパラメーターを受け入れます。

- 入力パラメーター。これは、プロシージャーへの入力として使用されます。パラメーターは、デフォルトでは入力パラメーターです。このため、キーワード IN はオプションです。
- 出力パラメーター。これは、プロシージャーから返される値です。
- 入出力パラメーター。これはプロシージャーに値を渡し、呼び出し側プロシージャーに値を返します。

プロシージャー見出しで入力パラメーターを宣言すると、プロシージャーの内部でそれらのパラメーター名を参照することにより、パラメーターの値にアクセスできます。パラメーター・データ型も宣言する必要があります。サポートされているデータ型については、 169 ページの『付録 A. データ型』を参照してください。

パラメーター宣言内で使用される構文は、以下のとおりです (完全な構文については、 177 ページの『付録 B. solidDB SQL 構文』を参照してください)。

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

パラメーターは、いくつあってもかまいません。入力パラメーターは、プロシージャーを呼び出すときに定義されたのと同じ順序で提供する必要があります。

プロシージャーの作成時に、パラメーターにデフォルト値を指定できます。パラメーターを宣言するときに、単にパラメーター・データ型の後に等号 (=) とデフォルト値を追加します。以下に例を示します。

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

定義されたパラメーターにデフォルト値があるプロシージャーを呼び出すときは、すべてのパラメーターに値を指定する必要はありません。すべてのパラメーターにデフォルト値を使用するには、単に以下のコマンドを使用します。

```
call participants()
```

パラメーターに値を渡すには、呼び出しステートメントの中でパラメーター名を使用し、以下の例に示すように、等号を使用してパラメーター値を割り当てます。

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

このコマンドは、パラメーター「children」に値 2 を指定し、パラメーター「adults」および「pets」にデフォルト値を指定します。

呼び出しステートメントの中でパラメーター名を使用しなかった場合、solidDB はパラメーターが作成ステートメント内と同じ順序で指定されたものと見なします。

例:

```
call participants(1)
```

このコマンドは、パラメーター「adults」に値 1 を使用し、パラメーター「children」および「pets」にデフォルト値を使用します。

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

このコマンドはパラメーター「adults」に値 1 を、パラメーター「children」に値 2 を使用します。パラメーター「pets」にはデフォルト値が使用されます。

パラメーターに名前を指定した場合は、それ以降のすべてのパラメーターも名前を持つ必要があります。このため、コマンド、

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

は、エラーを返します。

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

このコマンドはパラメーター「adults」に値 1 を、パラメーター「children」に値 2 を使用します。パラメーター「pets」にはデフォルト値が使用されます。

RETURNS の使用

ストアード・プロシージャーを使用して、データが別々の列に入っている複数行の結果セット表を返すことができます。これは、solidDB に所有権があるデータ返却方式であり、RETURNS 構造を使用して実行されます。

RETURNS 構造を使用する場合、出力データ行の結果セット列名を別個に宣言する必要があります。結果セット列名は、いくつあってもかまいません。結果セット列名は、プロシージャー定義の RETURNS セクションで宣言します。

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

デフォルトでは、プロシージャーは、ストアード・プロシージャーが実行された時点、または強制終了した時点での値が入っている 1 行のデータだけを返します。しかし、以下の構文を使用して、プロシージャーから結果セットを返すこともできます。

```
return row;
```

RETURN ROW 呼び出しごとに、返される結果セット内に新しい 1 行が追加されます。その行の列値は、結果セット列名の現行値です。

以下のステートメントは、2 つの入力パラメーターを持ち、出力行用に 2 つの結果セット列名を持つプロシージャーを作成します。

```
"CREATE PROCEDURE PHONEBOOK_SEARCH
 (IN FIRST_NAME VARCHAR, LAST_NAME VARCHAR)
 RETURNS (PHONE_NR NUMERIC, CITY VARCHAR)
BEGIN
-- プロシージャー本体
END";
```

このプロシージャーは、データ型 VARCHAR の 2 つの入力パラメーターを使用して呼び出す必要があります。このプロシージャーは、型が NUMERIC の PHONE_NR という列と、型が VARCHAR の CITY という列からなる出力表を返します。

以下に例を示します。

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

結果は、以下のようになります (プロシージャー本体がプログラムされた場合)。

```
PHONE_NR    CITY
3433555     NEW YORK
2345226     LOS ANGELES
```

以下のステートメントは、計算器プロシージャーを作成します。

```
"create procedure calc(i1 float, op char(1),
 i2 float)
 returns (calcresult float)
begin
 declare i integer;
```

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

この計算器は、次のコマンドでテストできます。

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

RETURNS を使用すると、SELECT ステートメントをデータベース・プロシージャーの中に包み込むこともできます。以下のステートメントは、SELECT ステートメントを使用して、データベースから作成されたバックアップを返すプロシージャーを作成します。

```
"create procedure show_backups
 returns (backup_number varchar, date_created varchar)
begin
-- 失敗するステートメント用の最初の設定アクション。
 exec sql  whenever sqlerror rollback, abort;

-- SELECT ステートメントを準備し、実行します。
 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);

-- 最初の行をフェッチします。
 exec sql fetch sel_cursor;
-- 表の終わりまでループします。
 while sqlsuccess loop
-- フェッチした行を返します。
  return row;
-- 次をフェッチします。
  exec sql fetch sel_cursor;
 end loop;
end";
```

宣言セクション

列の一時ストレージにプロシージャー内で使用するローカル変数、および制御値は、ストアード・プロシージャーで、BEGIN キーワードの直後の別のセクションで定義されます。

変数を宣言する構文は、以下のとおりです。

```
DECLARE variable_name datatype;
```

各 declare ステートメントは、セミコロン (;) で終了することに注意してください。

変数名は、変数を識別する英数字ストリングです。変数のデータ型は、任意のサポートされている有効な SQL データ型です。サポートされているデータ型については、 169 ページの『付録 A. データ型』を参照してください。

以下に例を示します。

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

DECLARE dat DATE;

END";
```

入力パラメーターと出力パラメーターは、プロシージャー内のローカル変数のように扱われます。ただし、違う点は、入力パラメーターには事前設定値があり、出力パラメーター値は返されるか、返される結果セットに追加できることです。

プロシージャー本体

プロシージャー本体には、割り当て、式、SQL ステートメントに基づいた実際のストアード・プロシージャー・プログラムが含まれます。

プロシージャー本体には、スカラー関数を含む、任意のタイプの式を使用できます。有効な式については、 337 ページの『式』を参照してください。

代入

変数に値を代入するには、以下のいずれかの構文を使用できます。

SET *variable_name* = *expression*;

または

variable_name := *expression*;

例:

```
SET i = i + 20 ;

i := 100;
```

スカラー関数と代入

スカラー関数とは、関数名の後に 1 対の括弧で囲んだ 0 個以上の引数の指定を伴う演算です。各スカラー関数は、1 つの値を返します。スカラー関数は、以下のように、代入と共に使用できることに注意してください。

```
"CREATE PROCEDURE scalar_sample
RETURNS (string_var VARCHAR(20))
BEGIN
-- CHAR(39) は、単一引用符 (アポストロフィ)
string_var := 'Joe' + {fn CHAR (39)} + 's Garage';
END";
```

このストアード・プロシージャーの結果は、以下の出力になります。

```
Joe's Garage
```

solidDB がサポートするスカラー関数 (SQL-92) のリストについては、 177 ページの『付録 B. solidDB SQL 構文』を参照してください。「*solidDB プログラマー・ガイド*」の付録では、SQL-92 とは少し異なる ODBC スカラー関数について説明しています。

代入での変数、定数、およびパラメーター

変数および定数は、プロシージャーが実行されるたびに初期化されます。デフォルトでは、変数は NULL に初期化されます。変数が明示的に初期化された場合を除いて、変数の値は、以下の例が示すように NULL になります。

```
BEGIN
DECLARE   total   INTEGER;
...
total := total + 1;  -- total に NULL を代入します。
...
```

したがって、変数に値が代入される前に、その変数を参照しないでください。

代入演算子の直後にある式は、いくら複雑でもかまいませんが、式が生成するデータ型は、変数のデータ型と同じであるか、それに変換可能なデータ型でなければなりません。

solidDB プロシージャー言語は、可能な場合、暗黙にデータ型の変換を行うことができます。このため、ある型のリテラル、変数、およびパラメーターを、別の型が予期されている場所に使用することができます。

以下の場合は、暗黙の変換ができません。

- 変換すると情報が失われる場合
- 整数に変換されるべきストリングに非数値データが入っている場合

例:

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

これはエラーを返します。

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

この結果、変数 string_var には値「123」が入ります。

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

これはエラーを返します。

ストリング割り当てでの単一引用符およびアポストロフィ

ストリングは単一引用符で区切られます。ストリング内で単一引用符を使用する場合は、2 つの単一引用符を並べて記述することで (’’) 単一引用符が 1 つだけ出力されます。これは一般に「エスケープ・シーケンス」として知られています。 この技法を使用するストアード・プロシージャーを以下に示します。

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

結果は以下のようになります。

```
Joe's Garage
```

別の例を示します。

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

この結果は以下のようになります。

```
I'm writing.
```

さらにもう 1 つ例を示します。

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

この結果は以下のようになります。

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

最後の例では、行内のストリングの末尾に 5 つの単一引用符が記述されていることに注意してください。このうちの最後が区切り文字 (終了引用符) で、その前の 4 つはデータの一部です。4 つの引用符は 2 組の引用符のペアとして処理され、各ペアは 1 つの単一引用符を表すエスケープ・シーケンスとして処理されます。

式

比較演算子

比較演算子は、1 つの式を別の式と比較します。結果は常に、TRUE、FALSE、NULL のいずれかです。一般に、比較は条件付き制御ステートメントの中で使用され、どのように複雑な式でも比較できます。次の表は、各演算子の意味を示しています。

表 5. 比較演算子

演算子	意味
=	に等しい
<>	に等しくない
<	より小さい
>	より大きい
<=	より小か等しい
>=	より大か等しい

!= の表記をストアード・プロシージャーの内部で使用できないことに注意してください。代わりに、ANSI-SQL 準拠の <> を使用してください。

論理演算子

論理演算子を使用して、より複雑な照会を構築できます。論理演算子 AND、OR、および NOT は、以下の真理値表に示すトライステート・ロジックに従って演算を行います。AND および OR は 2 項演算子で、NOT は単項演算子です。

表 6. 論理演算子: NOT

NOT	**true**	**false**	**NULL**
	false	true	NULL

表 7. 論理演算子: AND

AND	**true**	**false**	**NULL**
true	true	false	NULL
false	false	false	false
NULL	NULL	false	NULL

表 8. 論理演算子: OR

OR	**true**	**false**	**NULL**
true	true	true	true
false	true	false	NULL
NULL	true	NULL	NULL

真理値表に示すように、AND は両方のオペランドが真の場合にのみ値 TRUE を返します。一方、OR はどちらかのオペランドが真の場合に、値 TRUE を返します。NOT は、オペランドの反対の値 (論理否定) を返します。例えば、NOT TRUE は FALSE を返します。

NOT NULL は NULL を返します。ヌルは不定であるからです。

評価の順序を示すために小括弧を使用しなかった場合、演算子優先順位によって評価の順序が決まります。

「true」および「false」は SQL パーサーによって受け入れられるリテラルでなく、値であることに注意してください。論理式の値は、以下のように数値変数として解釈できます。

false = 0 または NULL

true = 1 またはそれ以外の任意の数値

例:

```
IF expression = TRUE THEN
```

これは、以下のように単純に書くことができます。

```
IF expression THEN
```

IS NULL 演算子

IS NULL 演算子は、そのオペランドがヌルである場合にブール値 TRUE を返し、ヌルでない場合に FALSE を返します。ヌルが関与する比較では、必ず NULL が生成されます。値が NULL かどうかを調べる場合は、以下の式を使用しないでください。

```
IF variable = NULL THEN...
```

これは、この式が TRUE に評価されることがないためです。

代わりに以下のステートメントを使用します。

```
IF variable IS NULL THEN...
```

solidDB のストアード・プロシージャーで複数の論理演算子を使用する場合は、個々の論理式を以下のように括弧で囲む必要があります。

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

制御構造

以下のセクションでは、プロシージャー本体内で使用できるステートメントについて、分岐ステートメントとループ・ステートメントも含めて説明します。

IF ステートメント

状況に応じて別のアクションを取らなければならないことがよくあります。IF ステートメントでは、一連のステートメントが条件付きで実行されます。IF ステートメントには、IF-THEN、IF-THEN-ELSE、および IF-THEN-ELSEIF の 3 つの形式があります。

IF-THEN

最も単純な形式の IF ステートメントでは、以下のようにキーワード THEN と END IF (ENDIF ではありません) で囲まれたステートメント・リストに条件が関連付けられます。

```
IF condition THEN
 statement_list;
END IF
```

この一連のステートメントは、条件が TRUE に評価された場合にのみ実行されます。条件が FALSE または NULL に評価された場合、IF ステートメントでは何も実行されません。いずれの場合も、制御は次のステートメントに渡されます。以下に例を示します。

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

IF-THEN-ELSE

2 番目の形式の IF ステートメントでは、以下のようにキーワード ELSE が追加され、その後に別のステートメント・リストが指定されます。

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

ELSE 節のステートメント・リストは、条件が FALSE または NULL に評価された場合にのみ実行されます。したがって、ELSE 節によって確実にステートメント・リストが実行されます。以下の例では、条件が TRUE または FALSE である場合に、1 番目または 2 番目の代入ステートメントがそれぞれ実行されます。

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

THEN 節と ELSE 節に IF ステートメントを組み込むこともできます。つまり、以下の例のように IF ステートメントをネストすることができます。

```
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

相互に排他的な複数の選択肢からアクションを選択しなければならないことがあります。3 番目の形式の IF ステートメントでは、以下のようにキーワード ELSEIF を使用して条件を追加します。

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

1 番目の条件が FALSE または NULL に評価されると、ELSEIF 節で別の条件が検査されます。IF ステートメントには任意の数の ELSEIF 節を指定できます。最後の ELSE 節は任意指定です。条件は上から下へ 1 つずつ評価されます。いずれかの条件が TRUE に評価されると、その関連するステートメント・リストが実行され、残りのステートメント (IF-THEN-ELSEIF 内) はスキップされます。すべての条件が FALSE または NULL に評価された場合は、ELSE 節内のシーケンスが実行されます。以下の例を考えてみましょう。

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

「sales」の値が 50000 を超えている場合は、1 番目と 2 番目の条件が TRUE となります。ただし、2 番目の条件は検査されないため、「bonus」に正しい値 1500 が代入されます。1 番目の条件が TRUE に評価されると、その関連するステートメントが実行され、IF-THEN-ELSEIF に続く次のステートメントに制御が渡されます。

可能であれば、ネストした IF ステートメントの代わりに ELSEIF 節を使用します。それによりコードが判読しやすくなり、理解しやすくなります。以下の IF ステートメントを比較してください。

```
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
```

この 2 つのステートメントは論理的には同等ですが、最初のステートメントではロジックの流れが不明確で、2 番目のステートメントではロジックの流れが明確です。

IF-THEN ステートメントでの小括弧の使用

以下のコードは、IF-THEN ステートメントでの小括弧の使用に関するルールを示した例です。IF-THEN ステートメントでの小括弧の使用に関する追加情報については、リリース・ノートも参照してください。

```
--- この部分コードは、IF ステートメントでの有効な論理条件の例を示して
--- います。
"CREATE PROCEDURE sample_if_conditions
BEGIN
DECLARE x INT;
DECLARE y INT;
x := 2;
y := 2;

--- 以下に示すように、IF 条件内の単一の論理式で
--- 小括弧を使用できます。
IF (x > 0) THEN
x := x - 1;
END IF;

--- 以下に示すように、IF 条件内の単一の論理式で
--- 小括弧を使用できますが、小括弧は必須ではありません。
IF x > 0 THEN
x := x - 1;
END IF;

--- 以下に示すように、複数の式が 1 つの
--- 論理条件内に存在する場合は、それぞれの副次式の前後に小括弧を
--- 使用できます (また、実際には必須です)。
IF (x > 0) AND (y > 0) THEN
x := x - 1;
END IF;

--- 以下の例は、前の例と同じものですが、
--- 式全体が追加の小括弧で囲まれている点
--- だけが異なります。
IF ((x > 0) AND (y > 0)) THEN
x := x - 1;
END IF;
```

WHILE-LOOP

WHILE-LOOP ステートメントは、以下に示すように、ある条件を、キーワード LOOP と END LOOP によって囲まれたステートメントのシーケンスに関連付けます。

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

ループのそれぞれの反復の前に、条件が評価されます。条件が TRUE に評価された場合、ステートメント・リストが実行され、その後、制御はループの先頭で再開されます。条件が FALSE または NULL に評価された場合、ループはバイパスされ、制御は次のステートメントへ渡されます。以下に例を示します。

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

反復回数は条件に依存し、ループが完了するまで分かりません。条件がループの先頭でテストされるため、シーケンスが 1 回も実行されない場合があります。後者の例では、「total」の初期値が 25000 より大きい場合、条件は FALSE に評価され、ループは完全にバイパスされます。

ループをネストさせることもできます。内側のループが終了すると、制御は次のループへ返されます。プロシージャーは、END LOOP の後にある次のステートメントから続行されます。

ループの終了

プロシージャーでループの中断の強制が必要となる場合があります。この操作を実装するには、LEAVE キーワードを使用します。

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

exit_condition の評価が成功するとループが中断し、プロシージャーは *statement_list2* から処理を続行します。

注:

solidDB データベースは ANSI-SQL の CASE 構文をサポートしていますが、ストアード・プロシージャー内で CASE 構造を制御構造として使用することはできません。

WHILE ループでの小括弧の使用

以下のコードは、WHILE ループでの小括弧の使用に関するルールを示した例です。WHILE ループでの小括弧の使用に関する追加情報については、リリース・ノートも参照してください。

```
--- この部分コードは、WHILE ループでの有効な論理条件の例を示して
--- います。
"CREATE PROCEDURE sample_while_conditions
```

```
BEGIN
DECLARE x INT;
DECLARE y INT;
x := 2;
y := 2;

--- 以下に示すように、WHILE 条件内の単一の論理式で
--- 小括弧を使用できます。
WHILE (x > 0) LOOP
x := x - 1;
END LOOP;

--- 以下に示すように、WHILE 条件内の単一の論理式で
--- 小括弧を使用できますが、小括弧は必須ではありません。
WHILE x > 0 LOOP
x := x - 1;
END LOOP;

--- 以下に示すように、複数の式が 1 つの
--- 論理条件内にある場合は、個々の式を、式ごとに
--- 小括弧で囲む必要があります。
WHILE (x > 0) AND (y > 0) LOOP
x := x - 1;
y := y - 1;
END LOOP;

--- 以下の例は、前の例と同じものですが、
--- 式全体が追加の小括弧で囲まれている点
--- だけが異なります。
WHILE ((x > 0) AND (y > 0)) LOOP
x := x - 1;
y := y - 1;
END LOOP;
```

ヌルの処理

ヌルが原因で動作が紛らわしくなることがあります。よくあるエラーを回避するために、以下のルールに従ってください。

- ヌルが関与する比較では、必ず NULL が生成される
- 論理演算子 NOT をヌルに適用すると、NULL が生成される
- 条件付き制御ステートメントで条件が NULL に評価されると、関連する一連のステートメントが実行されない

以下の例では、「x」と「y」が等しくないように見えるため、ステートメント・リストが実行されることが予測されます。ただし、ヌルが不確定な値であることを思い出してください。「x」が「y」と等しいかどうかは不明です。このため、IF 条件が NULL に評価され、ステートメント・リストの実行は回避されます。

```
x := 5;
    y := NULL;
    ...
    IF x <> y THEN  -- TRUE ではなく NULL に評価。
        statement_list;  -- 実行されない。
    END IF
```

以下の例では、「a」と「b」が等しいように見えるため、ステートメント・リストが実行されることが予測されます。ところがこの場合も、等しいかどうかが不明となるために IF 条件が NULL に評価され、ステートメント・リストの実行が回避されます。

```
a := NULL;
   b := NULL;
   ...
   IF a = b THEN  -- TRUE ではなく NULL に評価。
       statement_list;  -- 実行されない。
   END IF
```

NOT 演算子

論理演算子 NOT をヌルに適用すると、NULL が生成されます。このため、以下の 2 つのステートメントは必ずしも常に同等とは限りません。

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

ELSE 節内のステートメントのシーケンスは、IF 条件が FALSEまたは NULL に評価されたときに実行されます。x と y のどちらか、または両方が NULL の場合、最初の IF ステートメントは y の値を high に代入しますが、2 番目の IF ステートメントは、x の値を high に代入します。 x と y がどちらも NULL でない場合、両方の IF ステートメントは対応する値を high に代入します。

長さゼロのストリング

長さゼロのストリングは、solidDB サーバーによって、NULL でなく長さがゼロのストリングのように処理されます。NULL 値は、以下の例のように、明示的に割り当てる必要があります。

```
SET a = NULL;
```

これはまた、NULL 値かどうかのチェックが長さゼロのストリングに適用された場合に、FALSE が返されることを意味します。

ストアード・プロシージャーの例

以下に示す単純なプロシージャーの例では、入力パラメーターである誕生日を基に、対象の人物が成人かどうかを判別します。

スカラー関数での {fn ...} の使用、および代入を終了するセミコロンに注意してください。

```
"CREATE PROCEDURE grown_up
(birth_date DATE)
RETURNS (description VARCHAR)
BEGIN
DECLARE age INTEGER;
-- 誕生した日から経過した年数を特定。
age := {fn TIMESTAMPDIFF(SQL_TSI_YEAR, birth_date, now())};
IF age >= 18  THEN
-- 年齢が 18 歳以上であれば成人。
description := 'ADULT';
ELSE
-- そうでない場合は未成年。
description := 'MINOR';
END IF
END";
```

プロシージャーの終了

以下のキーワードを発行することで、どの場所でも完了前にプロシージャーを終了することができます。

```
RETURN;
```

このキーワードの後に、プロシージャーを呼び出したプログラムに制御が直接渡され、プロシージャー定義の RETURNS セクションで指定された結果セットの列名にバインドされた値が返されます。

データの戻り

OUT パラメーター・モードでデータを返すことができます。これは、データを返す標準 SQL-99 方式です。この方式で、プロシージャーからプログラムにデータを戻すことができます。構文情報については、 177 ページの『付録 B. solidDB SQL 構文』を参照してください。

OUT パラメーター・モードには、以下の特性があります。

- OUT パラメーター・モードを使用して、プロシージャーから呼び出し側プログラムにデータを戻すことができます。呼び出し側プログラムの中では、OUT パラメーターは変数のように機能します。つまり、OUT パラメーターをローカル変数のように使用できます。あらゆる方法で、値の変更または値の参照ができます。
- OUT パラメーターに対応する実パラメーターは、変数である必要があります。定数または式は使用できません。
- 変数と同様に、OUT パラメーターは NULL に初期化されます。

プロシージャーが終了する前に、明示的にすべての OUT パラメーターに値を割り当てる必要があります。そうしない場合、対応する実パラメーターが NULL になります。正常に終了した場合は、solidDB が実パラメーターに値を割り当てます。ただし、処理されない例外で終了した場合は、solidDB は実パラメーターに値を割り当てません。

データを返す solidDB 専用の方式については、 30 ページの『RETURNS の使用』を参照してください。

リモート・ストアード・プロシージャー

ストアード・プロシージャーは、ローカル側またはリモート側で呼び出すことができます。「リモート側で」とは、あるデータベース・サーバーが別のデータベース・サーバーのストアード・プロシージャーを呼び出すことができるという意味です。リモート・ストアード・プロシージャー呼び出しは、以下のような構文を使用します。

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

ここで、*node-ref* は、リモート・ストアード・プロシージャーがあるデータベース・サーバーを示します。

リモート・ストアード・プロシージャー呼び出しは、マスター/レプリカの関係を持つ 2 台の solidDB サーバーの間でのみ実行できます。呼び出しは、どちらの「方向」でも可能です。つまり、マスターがレプリカのストアード・プロシージャーを呼び出すことも、レプリカがマスターのストアード・プロシージャーを呼び出すこ

ともできます。リモート・ストアード・プロシージャーは、ローカル・プロシージャー呼び出しが可能なすべてのコンテキストから呼び出すことができます。そのため、例えば、CALL ステートメントを使用してリモート・ストアード・プロシージャーを直接呼び出したり、トリガーまたは別のストアード・プロシージャーの中から、または Start After Commit ステートメントからリモート・プロシージャーを呼び出したりできます。

リモート側から呼び出されるストアード・プロシージャーには、その他のすべてのストアード・プロシージャーに含めることができる、すべてのコマンドを含めることができます。すべてのストアード・プロシージャーは、同じ構文ルールで作成されます。単一のストアード・プロシージャーを別のタイミングで、ローカル側とリモート側の両方で呼び出すことができます。

ストアード・プロシージャーがリモート側で呼び出された場合、呼び出しがローカルだった場合と同じように、呼び出し側からのパラメーターを受け入れます。ただし、リモート・ストアード・プロシージャーは結果セットを返すことができません。返すことができるのはエラー・コードだけです。

ローカルおよびリモート・ストアード・プロシージャー呼び出しは、どちらも同期的です。つまり、プロシージャーがローカルとリモートのどちらで呼び出されても、呼び出し側は値が戻されるまで待機します。呼び出し側は、ストアード・プロシージャーがバックグラウンドで実行されている間、続行されません。 (ストアード・プロシージャーが START AFTER COMMIT から呼び出された場合、ストアード・プロシージャー呼び出し自体は同期的ですが、START AFTER COMMIT は同期的でないため、ストアード・プロシージャーは非同期バックグラウンド・プロセスのように実行されることに注意してください。)

重要:

リモート・ストアード・プロシージャーを処理するトランザクションは、ローカル・ストアード・プロシージャーを処理するトランザクションと異なります。ストアード・プロシージャーがリモート側で呼び出された場合、ストアード・プロシージャーの実行は、呼び出しを含むトランザクションの一部ではありません。そのため、ストアード・プロシージャーを呼び出したトランザクションをロールバックしてストアード・プロシージャー呼び出しをロールバックすることはできません。

リモート・ストアード・プロシージャーを呼び出す完全な構文は、以下のとおりです。

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

以下に例を示します。

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

CALL ステートメントについて詳しくは、 195 ページの『CALL』を参照してください。

ノード定義「DEFAULT」は、START AFTER COMMIT ステートメントでのみ使用します。詳しくは、START AFTER COMMIT のセクションを参照してください。

注:

1 つの CALL でリストできるノード定義は 1 つだけです。例えば、複数のレプリカに通知する場合は、それぞれを個別に呼び出す必要があります。ただし、複数の CALL ステートメントを含むストアード・プロシージャーを作成して、そのプロシージャーを 1 回呼び出すことができます。

リモート・ストアード・プロシージャーは、常に、プロシージャーを呼び出すサーバーではなく、プロシージャーを実行するサーバーに作成します。例えば、マスターが、レプリカ 1 で実行するプロシージャー foo() を呼び出す場合、プロシージャー foo() はレプリカ 1 に作成します。マスターは、リモート側で呼び出すストアード・プロシージャーの「内容」を知りません。実際には、マスターはストアード・プロシージャーについて、CALL ステートメント自体で指定する情報以外、何も知りません。以下に例を示します。

```
CALL foo(param1, param2) AT replica1
```

これには、プロシージャーの名前、いくつかのパラメーター値、プロシージャーを実行するレプリカの名前が含まれています。ストアード・プロシージャーは、呼び出し側には登録されません。つまり、呼び出し側は、プロシージャーが存在するかどうかすらわからずに、ある意味「やみくもに」呼び出します。存在しないプロシージャーを呼び出し側が呼び出そうとした場合、呼び出し側は、プロシージャーが存在しない旨のエラー・メッセージを受け取ります。

動的パラメーター・バインディングがサポートされます。例えば、以下は有効です。

```
CALL MYPROC(?, ?) AT MYREPLICA1;
```

ストアード・プロシージャー呼び出しは、バッファーまたはキューに入れられません。ストアード・プロシージャーを呼び出し、そのプロシージャーが存在しない場合、ストアード・プロシージャーが現れるまで呼び出しが「永続」することはありません。同様に、プロシージャーは存在するが、そのプロシージャーを持つサーバーがシャットダウンしたか、ネットワークから切断された、またはその他の理由でアクセス不可になった場合、呼び出しの「開いた」状態は維持されず、サーバーが再びアクセス可能になったときに呼び出しが再試行されます。このことは、「プル同期通知」(プッシュ同期) 機能を使用するときに知っておく必要があります。

アクセス権限

ストアード・プロシージャーを呼び出すには、呼び出し元がそのプロシージャーに対する EXECUTE 特権を持っている必要があります。(これは、ローカルとリモートのどちら側で呼び出す場合でも、すべてのストアード・プロシージャーに言えることです。)

ローカル側で呼び出されたプロシージャーは、呼び出し元の特権で実行されます。リモート側で呼び出されたプロシージャーは、リモート・サーバー上の指定されたユーザーの特権か、ローカルの呼び出し元に対応するリモート・ユーザーの特権のどちらでも実行できます。(レプリカ・ユーザーとマスター・ユーザーは、ストアード・プロシージャーが呼び出される前に、既にお互いへマップされている必要があ

ります。レプリカ・ユーザーからマスター・ユーザーへのマッピングについて詳しくは、「*IBM solidDB 拡張レプリケーション・ユーザー・ガイド*」を参照してください。)

リモート・ストアード・プロシージャーがレプリカから呼び出された場合 (そして、マスター上で実行される場合) は、どのマスター・ユーザーの特権を使用してプロシージャーを実行するかを指定するオプションがあります。

リモート・ストアード・プロシージャーがマスターから呼び出された場合 (そして、レプリカ上で実行される場合)、または、どのユーザーの特権を使用するかを指定しなかった場合、呼び出し側サーバーは、どのユーザーがそのストアード・プロシージャーを呼び出したか、およびレプリカ・ユーザーとマスター・ユーザーの間のマッピングに基づいて、どのユーザーの特権を使用すべきかを判断します。

これらの可能性について、以下で詳しく説明します。

1. プロシージャーがレプリカから呼び出された場合 (そして、マスター上で実行される場合) は、SET SYNC USER ステートメントを実行して、どのマスター・ユーザーの特権を使用するかを指定できます。SET SYNC USER は、リモート・ストアード・プロシージャーを呼び出す前に、ローカル・サーバー上で実行する必要があります。呼び出し側サーバー上で同期ユーザーが指定された後、呼び出し側サーバーはリモート・ストアード・プロシージャーが呼び出されるたびに、ユーザー名とパスワードをリモート・サーバー (マスター・サーバー) へ送信します。リモート・サーバーはプロシージャー呼び出しで送信されたユーザー ID とパスワードを使用して、プロシージャーの実行を試みます。そのユーザー ID とパスワードはリモート・サーバー内に存在する必要があり、指定されたユーザーはデータベースに対する適切なアクセス権限と、呼び出されたプロシージャーに対する EXECUTE 特権を持っている必要があります。

   SET SYNC USER ステートメントはレプリカ上でのみ有効なので、同期ユーザーを指定できるのは、レプリカがマスター上のストアード・プロシージャーを呼び出すときだけです。

2. 呼び出し元がマスターであるか、呼び出しがレプリカから行われ、呼び出しの前に同期ユーザーを指定しなかった場合、サーバーがリモート・サーバー上のどのユーザーがローカル・サーバー上のユーザーに対応するかの判別を試行します。

   呼び出し側サーバーがレプリカである場合 (R → M)

   呼び出し側サーバーはリモート・プロシージャーを呼び出すとき、以下の情報をリモート・サーバーに送信します。

   マスターの名前 (SYS_SYNC_MASTERS.NAME)。

   レプリカ ID (SYS_SYNC_MASTERS.REPLICA_ID)。

   マスター・ユーザー ID (このマスター・ユーザー ID は、プロシージャーを呼び出したローカル・ユーザーのユーザー ID に対応するマスター・ユーザー ID です。言うまでもなく、このローカル・ユーザーは、対応するマスター・ユーザーへ既にマップされている必要があります)。

このマスター・ユーザー ID 選択方式は、レプリカがデータをリフレッシュするときに使用される方式と同じものであることに注意してください。つまり、レプリカは SYS_SYNC_USERS 表を検索して、現行のローカル・レプリカ・ユーザーにマップされているマスター・ユーザーを見つけます。

呼び出し側サーバーがマスターである場合 (M → R)

呼び出し側サーバーはリモート・プロシージャーを呼び出すとき、以下の情報をリモート・サーバーに送信します。

マスターの名前 (SYS_SYNC_REPLICAS.MASTER_NAME)。

レプリカ ID (SYS_SYNC_REPLICAS.ID)。

呼び出し元のユーザー名。

呼び出し元のユーザー ID。

レプリカはマスター・ユーザー ID を受信すると、そのマスター ID にマップされているローカル・ユーザーをローカル・ユーザーを検索します。1 つのマスター・ユーザーに複数のレプリカ・ユーザーがマップされている場合があるため、サーバーは、指定されたマスター・ユーザーにマップされていて、そのストアード・プロシージャーを実行するために必要な特権を持っている、最初に見つかったローカル・ユーザーを使用します。

マスター・サーバーがレプリカ・サーバー上のストアード・プロシージャーを呼び出すには、マスターがレプリカの接続ストリングを知っている必要があります。レプリカは、マスターからの呼び出しを許可する場合、`solid.ini` ファイルの中で独自の接続ストリング情報を定義する必要があります。この情報はマスターに提供されます (レプリカは、どのようなメッセージをマスターに転送するときにも、コピーを組み込みます)。マスターはレプリカから接続ストリングを受信すると、以前の値を置き換えます (新しい値が異なっている場合)。

例:

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

以下のステートメントを使用して、レプリカの接続ストリングをマスターに知らせることもできます。

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

これは、マスターがレプリカを呼び出す必要があり、まだレプリカが接続ストリングをマスターに提供していない (つまり、マスターに何もメッセージを転送していない) 場合にも便利です。

ストアード・プロシージャーでの SQL の使用

ストアード・プロシージャーの内部で SQL ステートメントを使用することは、solsql のようなツールから SQL を直接発行することとは、いくらか違いがあります。

SQL ステートメントをストアード・プロシージャーの内部で使用するときは、特殊な構文が必要です。プロシージャーの内部で SQL ステートメントを実行する方法は 2 つあります。つまり、EXECDIRECT 構文を使用してステートメントを実行するか、SQL ステートメントを「カーソル」として処理することができます。以下で、両方の可能性について説明します。

EXECDIRECT

EXECDIRECT 構文は、結果セットがないステートメントや、変数を使用してパラメーター値を指定する必要がないステートメントに特に適しています。例えば、以下のステートメントは単一行のデータを挿入します。

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

EXECDIRECT について詳しくは、『EXECDIRECT』を参照してください。

カーソルの使用

カーソルは、ステートメントに結果セットが存在する場合、または、単一の基本ステートメントを繰り返し使用し、その際、ローカル変数からのさまざまな値をパラメーターとして (例えば、ループ内で) 使用したいという場合に適しています。

カーソルはサーバー・プロセス・メモリーのうち、処理中のステートメントを追跡するために割り振られる特定の部分です。メモリー・スペースは、下層にあるステートメントの 1 行を、(SELECT での) 現在行、またはそのステートメントによって影響を受ける (UPDATE、INSERT、および DELETE 内の) いくつかの行に関する何らかの状況情報と一緒に保持するために割り振られます。

このため、照会結果は一度に 1 行ずつ処理されます。ストアード・プロシージャー・ロジックは、行の実際の処理と、必要な行へのカーソルの位置決めを行う必要があります。

カーソルの処理には、以下の 5 つの基本ステップがあります。

1. カーソルの準備 - 定義
2. カーソルの実行 - ステートメントの実行
3. カーソル上でのフェッチ (選択プロシージャー呼び出しの場合) - 行ごとの結果の取得
4. 使用後のカーソルのクローズ - 再実行は依然として可能
5. メモリーからのカーソルの除去 - カーソルの削除

1. カーソルの準備

カーソルを定義 (準備) するには、以下の構文を使用します。

EXEC SQL PREPARE *cursor_name SQL_statement*;

カーソルを準備することにより、ステートメントの結果セットの 1 行を収容するためにメモリー・スペースが割り振られ、ステートメントの構文解析と最適化が行われます。

ステートメントに付けるカーソル名は、その接続内で固有の名前でなければなりません。これは、カーソルを含んでいるプロシージャーを再帰的に (少なくとも、PREPARE CURSOR の後にあり、対応する DROP CURSOR の前にあるステートメ

ントからは) 呼び出せないことを意味します。カーソルを準備する場合、solidDB サーバーは、その名前で現在開かれている別のカーソルがないかどうかを検査します。ある場合は、エラー番号 14504 が返されます。

ステートメント・カーソルも ODBC API を使用して開くことができる点に注意してください。それらのカーソル名は、プロシージャーから開かれたカーソルと異なっている必要があります。

例:

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

このステートメントは *sel_tables* という名前のカーソルを準備しますが、それに含まれているステートメントは実行しません。

2. カーソルの実行

ステートメントの準備が正常に完了したら、そのステートメントを実行できます。実行によって、適切な入力変数と出力変数がステートメントにバインドされ、実際のステートメントが実行されます。

実行ステートメントの構文は以下のとおりです。

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

オプションの INTO セクションは、ステートメントの結果データを変数にバインドします。

INTO キーワード後の括弧内にリストされた変数は、SELECT ステートメントまたは CALL ステートメントを実行する際に使用されます。これらの変数には、SELECT ステートメントまたは CALL ステートメントが実行されたときに、その結果の列がバインドされます。変数は、ステートメントでリストされた左端の列から順にバインドされます。変数のバインドは、リスト内のすべての変数がバインドされるまで次の列に対して継続して実行されます。例えば、前に準備したカーソル *sel_tables* に対してシーケンスを拡張するには、以下のステートメントを実行する必要があります。

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

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

これでステートメントが実行され、結果の表名が後続のフェッチ・ステートメントで変数タブに返されるようになります。

3. カーソルでのフェッチ

SELECT または CALL ステートメントの準備および実行が完了すると、そのステートメントからデータをフェッチできる状態になります。それ以外のステートメント

(UPDATE、INSERT、DELETE、DDL) では、結果セットが生成されないため、フェッチする必要がありません。結果のフェッチは以下のフェッチ構文を使って実行されます。

```
EXEC SQL FETCH cursor_name;
```

このコマンドは、カーソルから 1 行をフェッチして、ステートメント実行時に INTO キーワードにバインドされた変数に入れます。

前の例を完了して実際に結果の行を取得するには、以下のステートメントを実行します。

```
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;
```

これを実行すると、WHERE 節と一致した最初の表の表名が変数タブに格納されます。

SELECT で複数の表が検出された場合は、カーソル *sel_tables* でフェッチを行う後続の呼び出しで次の行が取得されます。

すべての表名をフェッチするには、ループ構成体を使用できます。

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

ループが完了すると、変数タブには最後にフェッチした表名が格納されることに注意してください。

4. カーソルのクローズ

カーソルは、ステートメントを発行することによって閉じることができます。

```
EXEC SQL CLOSE cursor_name;
```

これは、実際にメモリーからカーソル定義を除去するのではないので、必要になった時点でカーソルを再実行できます。

5. カーソルのドロップ

以下のステートメントでカーソルをメモリーからドロップすることで、すべてのリソースを解放できます。

```
EXEC SQL DROP cursor_name;
```

ストアード・プロシージャーの例

以下に示すストアード・プロシージャーの例では、EXECDIRECT とカーソルを別々の場所で使用します。

```
"CREATE PROCEDURE p2
BEGIN

-- 表に挿入する ID を保持する変数。
DECLARE id INT;

-- EXECDIRECT の単純な例。
```

```
EXEC SQL EXECDIRECT create table table1 (id_col INT);
EXEC SQL EXECDIRECT insert into table1 (id_col) values (1);

-- カーソルの例。
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";
```

エラー処理

SQLSUCCESS

プロシージャー本体で最後に実行された EXEC SQL ステートメントの戻り値は、変数 SQLSUCCESS に格納されます。この変数は、すべてのプロシージャーで自動的に生成されます。直前の SQL ステートメントが成功した場合は、SQLSUCCESS に値 1 が格納されます。SQL ステートメントが失敗した場合は、SQLSUCCESS に値 0 が格納されます。

例えば以下の例のように、SQLSUCCESS の値を使用してカーソルが結果セットの最後に到達したタイミングを特定できます。

```
EXEC SQL FETCH sel_tab;
-- ループ内の最後のステートメントが成功であるかぎりループ
WHILE SQLSUCCESS LOOP
     -- 結果を処理 (行を返すなど)
 EXEC SQL FETCH sel_tab;

END LOOP
```

SQLERRNUM

この変数には、最後に実行された SQL ステートメントのエラー・コードが格納されています。この変数はすべてのプロシージャーで自動的に生成されます。実行が成功すると、SQLERRNUM にはゼロ (0) が格納されます。

SQLERRSTR

この変数には、最後に失敗した SQL ステートメントのエラー・ストリングが格納されています。

SQLROWCOUNT

UPDATE、INSERT、および DELETE の各ステートメントが実行されると、ステートメントの結果を検査するための追加の変数が使用可能になります。変数 SQLROWCOUNT には、最後のステートメントの影響を受けた行の数が格納されています。

SQLERROR

プロシージャーからユーザー・エラーを生成するには、SQLERROR 変数を使用してステートメントが失敗する原因となった実際のエラー・ストリングを呼び出し側アプリケーションに返します。構文は以下のとおりです。

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

エラーは、以下のフォーマットで返されます。

User error: *error_string*

SQLERROR OF *cursorname*

EXEC SQL ステートメントのエラー・チェックには、このセクションの冒頭の SQLSUCCESS で説明したように SQLSUCCESS 変数を使用できます。ステートメントが失敗する原因となった実際のエラーを呼び出し側アプリケーションに返すには、以下の構文を使用します。

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

END IF
```

このステートメントが実行され、プロシージャーの戻りコードが SQLERROR であった場合は、処理が直ちに停止します。実際のデータベース・エラーは、SQLError 関数を使用して返すことができます。

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

プロシージャーの汎用的なエラー処理方法は、以下のステートメントで宣言できます。

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

このステートメントをストアード・プロシージャーに組み込むと、実行された SQL ステートメントのすべての戻り値でエラーが検査されます。ステートメント実行でエラーが返された場合は、プロシージャーが自動的に異常終了し、最後のカーソルの SQLERROR が返されます。オプションでトランザクションをロールバックすることもできます。

このステートメントは、EXEC SQL ステートメントの前の DECLARE 変数セクションの直後に挿入する必要があります。

例として、SYS_TABLES から「SYS」で始まるすべての表名を返すプロシージャー全体を以下に示します。

```
"CREATE PROCEDURE  sys_tabs
RETURNS (tab VARCHAR)
BEGIN
-- エラーの場合は異常終了
EXEC SQL WHENEVER SQLERROR ROLLBACK, ABORT;
-- カーソルを準備
EXEC SQL PREPARE sel_tables
 SELECT table_name
 FROM  sys_tables
 WHERE table_name LIKE 'SYS%';
-- カーソルを実行
EXEC SQL EXECUTE sel_tables INTO (tab);
-- 行をループ処理
EXEC SQL FETCH sel_tables;
WHILE sqlsuccess LOOP
      RETURN ROW;
```

```
      EXEC SQL FETCH sel_tables;
END LOOP
-- 使用したカーソルをクローズしてドロップ
EXEC SQL CLOSE sel_tables;
EXEC SQL DROP sel_tables;
END";
```

カーソル内のパラメーター・マーカー

カーソルをより動的にするために、SQL ステートメントにパラメーター・マーカーを含めて、実行時に実際のパラメーター値へバインドされる値を指示することができます。「?」シンボルがパラメーター・マーカーとして使用されます。

構文の例:

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

実行ステートメントを適合させるには、USING キーワードを組み込んで、変数をパラメーター・マーカーにバインドします。

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

この方法では、カーソルを再準備しなくても、単一のカーソルを複数回使用できます。カーソルを準備するには、ステートメントの構文解析と最適化も必要になるので、再利用可能なカーソルを使用することにより、大幅なパフォーマンスのゲインを達成できます。

USING リストは、変数のみを受け入れることに注意してください。この方法では、データを直接渡すことはできません。したがって、例えば、表の 1 つの列値が常に同じ (status = 'NEW') でなければならない表に挿入を行う必要がある場合、以下の構文は誤りになります。

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

正しい方法は、次のように PREPARE セクションで定数値を定義することです。

```
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);
```

USING リストの中で変数を複数回使用できることに注意してください。

SQL ステートメント内のパラメーターには、組み込みデータ型または明示宣言がありません。したがって、パラメーター・マーカーを SQL ステートメントに組み込むことができるのは、マーカーのデータ型をステートメント内の別のオペランドから推論できる場合だけです。

例えば、? + COLUMN1 のような算術式の場合、パラメーターのデータ型は、COLUMN1 によって表された名前付き列のデータ型から推論できます。データ型を判別できない場合、プロシージャーでパラメーター・マーカーを使用することはできません。

以下の表は、いくつかのパラメーター・タイプについて、データ型の判別方法を説明したものです。

表 9. パラメーターからのデータ型の判別

パラメーターの位置	**想定されるデータ型**
2 項演算子または比較演算子の一方のオペランド	他方のオペランドと同じ
BETWEEN 節内の第 1 オペランド	他方のオペランドと同じ
BETWEEN 節内の第 2 または第 3 オペランド	第 1 オペランドと同じ
IN と一緒に使用される式	副照会の最初の値または結果列と同じ
IN と一緒に使用される値	式と同じ
LIKE と一緒に使用されるパターン値	VARCHAR
UPDATE で使用される更新値	更新列と同じ

アプリケーションでは、以下の位置にパラメーター・マーカーを配置できません。

- SQL ID (表の名前、列の名前など) として
- SELECT リストの中
- 比較述部内の両方の式として
- 2 項演算子の両方のオペランドとして
- BETWEEN 演算の第 1 と第 2 の両方のオペランドとして
- BETWEEN 演算の第 1 と第 3 の両方のオペランドとして
- IN 演算の式と最初の値の両方として
- 単項の + または - 演算のオペランドとして
- set 関数参照の引数として

詳しくは、ANSI SQL-92 仕様を参照してください。

以下の例では、ストアード・プロシージャーは複数のカーソルを使用して 1 つの表から行を読み取り、それらの行の一部分を別の表に挿入します。

```
"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";
```

他のプロシージャーの呼び出し

プロシージャーの呼び出しは、サポートされている SQL 構文に含まれているので、ストアード・プロシージャー中から別のストアード・プロシージャーを呼び出すことができます。ネストされたプロシージャーのレベルの限度は、デフォルトでは 16 です。この最大値を超えた場合、トランザクションは失敗します。最大ネスト・レベルは、solid.ini 構成ファイルの MaxNestedProcedures パラメーターで設定します。詳しくは、「*solidDB 管理者ガイド*」の付録『構成パラメーター』を参照してください。

すべての SQL ステートメントの場合と同様に、カーソルを準備し、次のように実行する必要があります。

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

プロシージャー *myproc* が 1 つ以上の値を返す場合は、引き続き、カーソル *cp* に対してフェッチを実行し、それらの値をリトリーブする必要があります。

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

呼び出されたプロシージャーが *return row* ステートメントを使用する場合、呼び出し側プロシージャーは WHILE LOOP 構文を使用してすべての結果をフェッチする必要があることに注意してください。

再帰呼び出しは可能ですが、カーソル名は接続レベルで固有なため、推奨されません。

位置付け更新および位置付け削除

solidDB プロシージャーでは、位置付け更新と位置付け削除を使用できます。これは、所定のカーソルが現在置かれている行に対して、更新または削除が行われることを意味しています。位置付け更新および位置付け削除は、プロシージャー内でカーソル名を使用しているストアード・プロシージャーの中でも使用できます。

位置付け更新には、以下の構文を使用します。

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

また、削除には以下の構文を使用します。

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

どちらの場合も、*cursor_name* は更新/削除される表に対して SELECT を実行するステートメントを参照します。

位置付けカーソル更新は、セマンティックに関して疑念のある SQL 規格の概念であり、solidDB サーバーでも、そのためにいくつかの特異な点が生じています。位置付け更新を使用する場合は、以下の制限に注意してください。

以下は、疑似コードで書かれた例で、これは solidDB サーバーではエンドレス・ループの原因となります (簡潔で分かりやすくするために、エラー処理、変数のバインディング、およびその他の重要なタスクは省略してあります)。

```
"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";
```

エンドレス・ループを引き起こす原因は、更新がコミットされたとき、新しいバージョンの行がカーソル内で可視となり、次の FETCH ステートメント内でその行がアクセスされるという事実にあります。なぜそうなるかというと、インクリメントされた行バージョン番号がキー値に組み込まれ、カーソルは変更された行を、現行位置の後にある次に大きなキー値として検出するからです。行は再び更新されてキー値が変更され、再びその行が次に検出される行になります。

上記の例では、更新される COLUMN2 は表の主キーの一部としては想定されておらず、索引項目の唯一の部分だった行バージョン番号が変更されました。しかし、カーソルがデータの検索に使用した索引の一部である列値が変更された場合、変更された行が検索セット内のはるか前方または後方へジャンプする可能性があります。

これらの理由から、位置付け更新の使用は一般的には推奨されず、可能なときは必ず、検索付き更新を使用する必要があります。しかし、更新ロジックが複雑すぎて、SQL の WHERE 節では表現できない場合もあり、そのような場合は、以下に示すように位置付け更新を使用できます。

位置付けカーソル更新が solidDB で確定的に機能するのは、WHERE 節において、更新される行が条件に一致せず、したがって、フェッチ・ループ内で再び出現することがない場合です。そのような検索基準を構築するには、その目的だけに追加の列を使用しなければならない場合もあります。

オープン・カーソルでは、ユーザーによる変更は、同じデータベース・セッション内でそれらの変更がコミットされなければ可視にならないことに注意してください。

トランザクション

ストアード・プロシージャーは、データベースに対する他のインターフェースと同じようにトランザクションを使用します。トランザクションは、プロシージャーの

内部または外部でコミットまたはロールバックされます。プロシージャー内部では、以下の構文を使用してコミットまたはロールバックが行われます。

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

上記のステートメントは、直前のトランザクションを終了して新しいトランザクションを開始します。

トランザクションがプロシージャー内でコミットされない場合は、以下の機能を使用して外部からトランザクションを終了することができます。

- solidDB SA
- 別のストアード・プロシージャー
- 自動コミット (接続の AUTOCOMMIT スイッチが ON に設定されている場合)

接続の自動コミットがアクティブ化されていても、プロシージャー内部では自動コミットが強制されないので注意してください。コミットはプロシージャーが終了するときに行われます。

デフォルト・カーソル管理

デフォルトでは、プロシージャーが終了するとき、プロシージャーでオープンされたすべてのカーソルはクローズされます。カーソルのクローズは、カーソルが準備状態のままになり、再実行できることを意味します。

終了後、プロシージャーはプロシージャー・キャッシュに入れられます。プロシージャーがキャッシュからドロップされると、すべてのカーソルが最終的にドロップされます。

キャッシュで保持されるプロシージャーの数は、 `solid.ini` ファイルの以下の設定で決まります。

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

つまり、プロシージャーがプロシージャー・キャッシュにある間、すべてのカーソルは、ドロップされない限り再使用可能です。solidDB サーバー自身が、宣言したカーソルの追跡を続けることでプロシージャー・キャッシュを管理し、カーソルに含まれるステートメントの準備ができたかどうかを通知します。

特に、大量マルチユーザー環境では、カーソル管理で相当の量のサーバー・リソースを使用する可能性があるため、常にカーソルをすぐにクローズするようにして、できれば不要になったすべてのカーソルのドロップも行うようにしてください。カーソル準備の手間を軽減するため、特に頻繁に使用するカーソルだけは、ドロップせずに残してかまいません。

トランザクションは、プロシージャーまたはその他のステートメントに関連しないことに注意してください。そのため、コミットまたはロールバックでは、プロシージャーでどのリソースも解放されません。

SQL についての注

- 使用される SQL ステートメントに制限はありません。有効な任意の SQL ステートメントを、DDL ステートメントおよび DML ステートメントも含め、ストアード・プロシージャーの中で使用できます。
- カーソルは、ストアード・プロシージャー内の任意の場所で宣言できます。使用されることが確かなカーソルは、宣言セクションの直後に準備するのが最良の方法です。
- 制御構造の内部で使用され、したがって常に必要なわけではないカーソルは、そのカーソルがアクティブ化されるポイントで宣言するのが最良の方法です。これは、オープン・カーソルの量を制限し、ひいてはメモリー使用量を制限するためです。
- カーソル名は変数ではなく、宣言されない ID であり、照会を参照するためにのみ使用されます。カーソル名に値を割り当てることはできず、式の中でカーソル名を使用することもできません。
- カーソルは、再準備しなくても、繰り返し再実行できます。これはパフォーマンスに重大な影響を及ぼす可能性があることに注意してください。同様なステートメントでカーソルを繰り返し準備すると、既に準備されたカーソルを再実行する場合に比べて、パフォーマンスが 40 % 近くも低下する場合があります。
- すべての SQL ステートメントは、前にキーワード EXEC SQL を付ける必要があります。

プロシージャー・スタックを表示する関数

以下の関数をストアード・プロシージャーに組み込むことで、プロシージャー・スタックの現在の内容を分析できます。

- PROC_COUNT ( )

  この関数は、プロシージャー・スタック内のプロシージャーの数を返します。これには現在のプロシージャーも含まれます。

- PROC_NAME (N)

  この関数は、スタック内の N 番目のプロシージャー名を返します。最初のプロシージャーの位置はゼロです。

- PROC_SCHEMA (N)

  この関数は、プロシージャー・スタック内の N 番目のプロシージャーのスキーマ名を返します。

上記の関数では、呼び出し元がアプリケーションであるかプロシージャーであるかによって動作が変わるストアード・プロシージャーを考慮しています。

プロシージャー特権

ストアード・プロシージャーは、作成者が所有し、作成者のスキーマの一部です。別のスキーマでストアード・プロシージャーを実行する必要があるユーザーは、そのプロシージャーの EXECUTE 特権が付与されている必要があります。

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

この関数は、プロシージャー・スタック内の N 番目のプロシージャーのスキーマ名を返します。

付与されたプロシージャー内でアクセスされるすべてのデータベース・オブジェクト (後で呼び出されるプロシージャーも含む) は、プロシージャーの所有者の権限に従ってアクセスされます。特別な権限付与は必要ありません。

作成者の特権で実行されるため、そのプロシージャーは、作成者が持つ、表などのオブジェクトへのアクセス権限を持つだけでなく、使用するスキーマとカタログも作成者のものを使用します。例えば、ユーザー「Jasmine」が作成したプロシージャー「Proc1」をユーザー「Sally」が実行するとします。また、Sally と Jasmine の両方が、「table1」という表を持つとします。デフォルトで、ストアード・プロシージャー Proc1 は、ユーザー Sally が Proc1 を呼び出したとしても、Jasmine のスキーマにある table1 を使用します。

特権とリモート・ストアード・プロシージャー呼び出しについて詳しくは、44 ページの『アクセス権限』も参照してください。

トリガーの使用

トリガーは、ユーザーが表内のデータを変更しようとしたときに solidDB サーバーが自動的に実行する、ストアード・プロシージャー・コードをアクティブにします。表に 1 つ以上のトリガーを作成できます。各トリガーは、特定の INSERT、UPDATE、または DELETE コマンドでアクティブ化するように定義されます。ユーザーが表のデータを変更すると、そのコマンドに対応するトリガーがアクティブ化されます。

トリガーを使用すると、以下のことができます。

- 外部キー値が既存の主キー値に一致するようにするなど、参照整合性制約を実装する。
- 意図された変更がデータベースの整合性を危険にさらさないようにすることにより、ユーザーが誤ったデータ変更または矛盾するデータ変更を行わないようにする。
- 変更の前または後に、行の値に基づいてアクションを実行する。
- ロジック処理の多くをバックエンドに転送し、アプリケーションで実行する必要がある作業の量を減らすと共に、ネットワーク・トラフィックを減らす。

トリガーの仕組み

solidDB データベースでのトリガーの仕組みを理解するには、トリガーが使用可能であるときにデータ操作ステートメントが実行される順序が重要となります。

solidDB の DML 実行モデルでは、solidDB サーバーが数々の妥当性検査を行ってから、データ操作ステートメント (INSERT、UPDATE、または DELETE) を実行します。以下に、1 つの DML ステートメントに対するデータ妥当性検査、トリガー実行、および保全性制約検査の実行順序を示します。

1. ステートメントの一部となっている値 (つまりバインドされていない値) を検証します。これには、NULL 値の検査やデータ型 (数値など) の検査などが含まれます。

2. 表レベルのセキュリティー検査を実行します。
3.

   対象の SQL ステートメントの影響を受ける各行で繰り返し処理を行います。各行に対して以下のアクションが上から順に実行されます。
   a. 列レベルのセキュリティー検査を実行します。
   b. BEFORE 行トリガーを起動します。
   c. バインドされる値を検証します。これには、NULL 値の検査、データ型の検査、サイズの検査 (文字ストリングが長すぎないかなど) が含まれます。

      サイズの検査はバインドされない値にも実行されることに注意してください。
   d. INSERT/UPDATE/DELETE を実行します。
   e. AFTER ROW トリガーを起動します。
4. ステートメントをコミットします。
   a. 並行性競合検査を実行します。
   b. 重複値の検査を実行します。
   c. DML を呼び出すときに参照整合性検査を実行します。

注: トリガーによって DML が実行されるようにすることもできます。これは上記モデルに示す手順に適用されます。

トリガーの作成

CREATE TRIGGER ステートメント (以下で説明) を使用して、トリガーを作成します。ALTER TRIGGER ステートメントを使用して、既存のトリガーまたは表に定義したすべてのトリガーを使用不可にできます。詳しくは、 80 ページの『トリガー属性の変更』を参照してください。 ALTER TRIGGER ステートメントを使用すると、solidDB サーバーは、アクティブ化する DML ステートメントが発行されたときにトリガーを無視するようになります。このステートメントで、現在非アクティブであるトリガーを使用可能にすることもできます。

トリガーをシステム・カタログからドロップするには、DROP TRIGGER を使用します。詳しくは、 80 ページの『トリガーのドロップ』を参照してください。

CREATE TRIGGER ステートメント

CREATE TRIGGER ステートメントは、トリガーを作成します。トリガーを作成するには、DBA、またはトリガーを定義している表の所有者である必要があります。トリガーを作成するには、トリガーを定義している表のカタログ、スキーマ/所有者および名前を提供します。CREATE TRIGGER ステートメントの例については、 76 ページの『トリガーの例』を参照してください。

CREATE TRIGGER ステートメントの構文は、以下のとおりです。

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

キーワードおよび節

以下のトピックは、キーワードおよび節の要約です。

trigger_name

trigger_name には、最大 254 文字を指定できます。

BEFORE | AFTER 節

BEFORE | AFTER 節は、トリガーを、データを変更する DML ステートメントの呼び出しの前と後のどちらで実行するかを指定します。状況によっては、BEFORE 節と AFTER 節は交換可能です。ただし、一方の節がもう一方の節より望ましい場合もあります。

- ドメイン制約および参照整合性の検査など、データの妥当性検査を行う場合は、BEFORE 節を使用した方が効率的です。
- AFTER 節を使用すると、DML ステートメントの呼び出しによって使用可能になった、表の行が処理されます。逆に、AFTER 節は DELETE ステートメントを呼び出した後に、データ削除の確認も行います。

1 つの表に定義できるトリガーは 6 つまでで、表、イベント (INSERT、UPDATE、DELETE)、および時期 (BEFORE および AFTER) からなる 1 つの組み合わせに 1 つ定義できます。例えば、BEFORE 節と AFTER 節のそれぞれに 1 つずつトリガーを定義することにより、1 つの DML 操作につき 2 つずつトリガーを提供できます。さらに、それらの組み合わせに INSERT、UPDATE、および DELETE トリガーを提供すれば、合計で最大値の 6 つのトリガーを使用することになります。

以下の例は、table 1 の BEFORE INSERT に対して定義されたトリガー trig01 を示しています。

```
"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"
```

以下に、CREATE TRIGGER コマンドの BEFORE および AFTER 節をそれぞれの DML 操作に使用した例を (意味と利点も含めて) 示します。

- UPDATE 操作

BEFORE 節は、UPDATE を処理する前に、変更されたデータが整合性制約ルールに従っているかどうかを検査できます。BEFORE UPDATE 節と一緒に REFERENCING NEW AS *new_col_identifier* 節を使用すると、更新された値をトリガー SQL ステートメントから使用できます。トリガー内で、UPDATE の実行前にデフォルトの列値または派生した列値を設定できます。

AFTER 節は、新規に変更されたデータに対して操作を行うことができます。例えば、支社のアドレスを更新後に、その支社の売上高を計算できます。

AFTER UPDATE 節と一緒に REFERENCING OLD AS *old_col_identifier* 節を使用すると、更新を呼び出す前に存在していた値に、トリガー SQL ステートメントからアクセスできます。

- INSERT 操作

  BEFORE 節は、INSERT を実行する前に、新しいデータが整合性制約ルールに従っているかどうかを検査できます。パラメーターとして引き渡された列値は、トリガー SQL ステートメントで可視ですが、挿入された行は可視ではありません。トリガー内で、INSERT の実行前にデフォルトの列値または派生した列値を設定できます。

  AFTER 節は、新規に挿入されたデータに対して操作を行うことができます。例えば、販売注文の挿入後に、注文の合計を計算して、顧客が割引きの対象になるかどうかを調べることができます。

  列値はパラメーターとして引き渡され、挿入された行は、トリガー SQL ステートメントで可視です。

- DELETE 操作

  BEFORE 節は、削除されようとするデータに対して操作を行うことができます。パラメーターとして引き渡された列値と、削除される挿入された行は、トリガー SQL ステートメントで可視です。

  AFTER 節を使用して、データの削除を確認できます。パラメーターとして引き渡された列値は、トリガー SQL ステートメントで可視です。削除された行がトリガー SQL ステートメントで可視であることに注意してください。

INSERT | UPDATE | DELETE 節

INSERT | UPDATE | DELETE 節は、ユーザー・アクション (INSERT、UPDATE、DELETE) が試行されたときのトリガー・アクションを示します。

トリガーの処理に関連するステートメントは、まず表での呼び出し側 DML ステートメント (INSERT、UPDATE、DELETE) からのコミットおよび自動コミットの前に発生します。トリガー本体またはトリガー本体の中で呼び出されたプロシージャーが COMMIT または ROLLBACK を実行しようとすると、solidDB サーバーは対応するランタイム・エラーを返します。

INSERT では、表での INSERT によってトリガーがアクティブ化されるように指定されます。n 行のデータをロードすることは、n 回の挿入と見なされます。

注: トリガーを使用可能にしてデータをロードしようとすると、パフォーマンスに影響が生じることがあります。ビジネス・ニーズに応じて、ロードの前はトリガーを使用不可にし、ロードの後にトリガーを使用可能にすることができます。詳しくは、 80 ページの『トリガー属性の変更』を参照してください。

DELETE では、表での DELETE によってトリガーがアクティブ化されるように指定されます。

UPDATE では、表での UPDATE によってトリガーがアクティブ化されるように指定されます。UPDATE 節を使用するときは、以下のルールに注意してください。

- トリガーの REFERENCES 節の中で列を参照できる (列に別名を付けることができる) のは、BEFORE サブ節で 1 回、AFTER サブ節で 1 回までです。また、列が BEFORE と AFTER の両方のサブ節で参照される場合は、各サブ節で列の別名が異なっていなければなりません。
- solidDB サーバーでは、同じ表に対する再帰的更新が可能で、同じ行に対する再帰的更新が禁止されません。

solidDB サーバーでは、複数の異なるトリガーのアクションによって同じデータが更新される状況は検出されません。例えば、Table1 に 2 つの更新トリガーがあるとします (1 つは BEFORE トリガー、もう 1 つは AFTER トリガー) があるとします。Table1 で更新が試行されると、2 つのトリガーがアクティブ化されます。どちらのトリガーも、2 番目の表 (Table2) の同じ列 (Col3) を更新するストアード・プロシージャーを呼び出します。最初のトリガーが、Table2.Col3 を 10 に更新し、2 番目のトリガーが Table2.Col3 を 20 に更新します。

同様に、solidDB サーバーでは、トリガーをアクティブ化する UPDATE の結果がトリガー自体のアクションと競合する状況は検出されません。例えば、以下の SQL ステートメントがあるとします。

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

この UPDATE によってトリガーがアクティブ化されると、以下の SQL ステートメントを含むプロシージャーが呼び出され、そのプロシージャーによってトリガーをアクティブ化した UPDATE の結果が上書きされます。

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

注: 上の例は再帰的なトリガーの実行につながる可能性があります。これは回避する必要があります。

table_name

table_name は、トリガーが作成される表の名前です。solidDB サーバーでは、従属トリガーが定義されている表をドロップすることができます。表をドロップすると、トリガーを含むすべての従属オブジェクトがドロップされます。それでもランタイム・エラーが発生することがあるので注意してください。例えば、A と B の 2 つの表を作成したとします。プロシージャー SP-B が表 A にデータを挿入した後に表 A がドロップされた場合、表 B に SP-B を呼び出すトリガーがあるとランタイム・エラーが発生します。

trigger_body

trigger_body には、トリガーが起動されたときに実行されるステートメントが格納されています。トリガーの本体を定義するルールは、ストアード・プロシージャーの本体を定義するルールと同じです。ストアード・プロシージャー本体の作成について詳しくは、 27 ページの『ストアード・プロシージャー』を参照してください。

トリガー本体が、solidDB サーバーに登録されているプロシージャーを呼び出すこともできます。solidDB プロシージャー呼び出しルールは、標準のプロシージャー呼び出し方法に従います。

ビジネス・ロジックのエラーは明示的に検査し、エラーを発生させる必要があります。

REFERENCING 節

この節は、 INSERT/UPDATE/DELETE 操作でトリガーを作成するときのオプションです。INSERT 操作および DELETE 操作の場合、現行の列 ID を参照できるようにします。UPDATE 操作の場合、操作が発生する列に別名を割り当てることで、古い列 ID と新しい更新された列 ID の両方を参照できるようにします。

アクセスするときは、OLD または NEW *col_identifier* を指定する必要があります。REFERENCING サブ節を使用して定義しない場合、solidDB サーバーは *col_identifier* へのアクセスを提供しません。

{OLD | NEW} *column_name* AS *col_identifier*

この REFERENCING 節のサブ節を使用して、UPDATE 操作の前と後の両方で、列の値を参照できます。これは、ストアード・プロシージャーに渡すことができる新旧の列値セットを生成します。渡されたストアード・プロシージャーには、それらのパラメーター値を判別するためのロジック (例えば、ドメイン制約の検査など) が含まれています。

OLD AS 節を使用して、UPDATE の前に存在していた表の古い ID に別名を割り当てます。NEW AS 節を使用して、UPDATE の後に存在する表の新しい ID に別名を割り当てます。

同じ列の新旧両方の値を参照する場合は、異なる *col_identifier* を使用する必要があります。

NEW または OLD として参照される各列は、別個の REFERENCING サブ節を持っている必要があります。

トリガー内のステートメント・アトミシティは、トリガーで実行される操作が、トリガー内の後続の SQL ステートメントで可視になる単位です。例えば、トリガー内で INSERT ステートメントを実行し、その後、同じトリガー内で選択を実行する場合、挿入された行は可視です。

AFTER トリガーの場合、挿入された行または更新された行は AFTER 挿入トリガーの中で可視ですが、削除された行は、そのトリガー内で実行される選択には見ることができません。BEFORE トリガーの場合、挿入された行または更新された行はトリガー内で可視でなく、削除された行は可視です。UPDATE の場合、更新前の値を BEFORE トリガーの中で使用できます。

以下の表で、トリガーのステートメント・アトミシティの要約を示し、トリガー本体の SELECT ステートメントで行が可視かどうかを示します。

表 10. トリガーのステートメント・アトミシティ

操作	**BEFORE トリガー**	**AFTER トリガー**
INSERT	行は可視でない	行は可視である
UPDATE	前の値は可視である	新しい値は可視である
DELETE	行は可視である	行は可視でない

トリガーのコメントおよび制限事項

- トリガーが呼び出すストアード・プロシージャーを使用するには、トリガーが定義されている表のカタログ、スキーマ/所有者、および名前を指定し、表でトリガーを使用可能にするか使用不可にするかを指定します。ストアード・プロシージャーについて詳しくは、 65 ページの『トリガーおよびプロシージャー』を参照してください。
- 表にトリガーを作成するには、DBA 権限があるか、トリガーを定義している表の所有者である必要があります。
- デフォルトでは、表、イベント (INSERT、UPDATE、DELETE)、およびタイミング (BEFORE および AFTER) の各組み合わせにトリガーを 1 つずつ定義できます。つまり、表ごとに最大 6 つのトリガーを作成できます。

注: トリガーは、各行に適用されます。つまり、10 回の挿入があると、トリガーが 10 回実行されます。

- ビューにはトリガーを定義できません (ビューが単一表に基づいている場合でも同様です)。
- トリガーが定義されている表は、従属列が影響を受ける場合、変更できません。
- システム表にはトリガーを作成できません。
- ドロップまたは変更されたオブジェクトを参照するトリガーは実行できません。このエラーを防ぐには、以下のようにします。
  - ドロップした参照先オブジェクトを再作成します。
  - 変更したすべての参照先オブジェクトを元の状態 (トリガーが認識する状態) にリストアします。
- トリガー・ステートメントでは、二重引用符で囲むことで予約語を使用できます。例えば、以下の CREATE TRIGGER ステートメントは、予約語である「data」という名前の列を参照します。

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

トリガーおよびプロシージャー

トリガーは、ストアード・プロシージャーを呼び出して、solidDB サーバーに他のトリガーを実行させせることができます。プロシージャーは、トリガー本体の中で呼び出すことができます。実際に、プロシージャー呼び出しのみを含んだトリガー本体を定義できます。トリガー本体から呼び出されるプロシージャーは、別のトリガーを呼び出すことができます。

トリガー本体の中でストアード・プロシージャーを使用する場合は、まず CREATE PROCEDURE ステートメントでプロシージャーを格納する必要があります。

プロシージャー定義では、COMMIT ステートメントおよび ROLLBACK ステートメントを使用できます。ただしトリガー本体では、COMMIT ステートメント (AUTOCOMMIT および COMMIT WORK を含む) と ROLLBACK ステートメントを使用できません。使用できるのは WHENEVER SQLERROR ABORT ステートメントのみです。

トリガーは最大 16 レベルの深さまでネストできます (この制限は構成パラメーターで変更できます)。トリガーが無限ループに入ると、solidDB サーバーは上限である 16 レベルのネスト (またはシステム・パラメーター) に達した時点でこの再帰的なアクションを検出し、ユーザーにエラーを返します。例えば、表 T1 に挿入を試みることでトリガーをアクティブ化し、そのトリガーが同じく T1 への挿入を試みるストアード・プロシージャーを呼び出すと、トリガーが再帰的にアクティブ化されます。

一連のネストされたトリガーが途中で失敗した場合は、solidDB サーバーがこれらのトリガーを最初にアクティブ化したステートメントをロールバックします。

デフォルト列または派生列の設定

INSERT 操作と UPDATE 操作でデフォルト列または派生列の値を設定するトリガーを作成することができます。この目的で CREATE TRIGGER コマンドを使用してトリガーを作成する場合は、トリガーが以下のルールに従っている必要があります。

- トリガーは、INSERT 操作または UPDATE 操作の前 (BEFORE) に実行する必要があります。列の値は BEFORE トリガーでのみ変更されます。列の値は INSERT 操作または UPDATE 操作の前に設定する必要があるため、AFTER トリガーを使用して列の値を設定しても無意味です。また、DELETE 操作は列値の変更に該当しないことにも注意してください。
- INSERT 操作および UPDATE 操作の REFERENCING 節で、変更後の新しい列値を示す NEW を指定する必要があります。元の列値 (OLD) を変更しても意味はありません。
- 新しい列値は、参照元のセクションで定義されている変数の値を変更するだけで設定できます。

パラメーターおよび変数の使用

レコードを更新し、その更新によってトリガーが起動された場合、トリガー自体がそのレコード内のいくつかの列の値を変更することがあります。状況によっては、「古い」値と「新しい」値の両方をトリガー内で参照したい場合もあります。

REFERENCING 節を使用すると、同じトリガー内で古い値と新しい値のどちらも参照できるよう、それらの値に「別名」を作成できます。例えば、2 つの表があり、1 つに顧客情報が、もう 1 つに請求書情報が入っているとします。表は、各請求書の請求金額を格納しているだけでなく、各顧客の「total_bought」フィールドを含んでいます。この「total_bought」フィールドには、それまでにその顧客へ送られたすべての請求書の累計が入っています。(このフィールドを使用して、大口の顧客を識別することもできます。)

1 つの請求書の total_amount が更新されると、顧客表にあるその顧客のレコードの「total_bought」値も更新されます。これを行うために、請求書に保管されている古い値の金額が減算され、請求書内の新しい値の金額が加算されます。例えば、ある顧客の $100 だった請求書が $150 に変更された場合、「total_bought」フィールドから $100 が減算され、$150 が加算されます。REFERENCING 節を正しく使用すると、トリガーは古い値と価格の列の両方を「見る」ことができ、それによって、total_bought 列を更新できます。

REFERENCING 節によって作成される列別名は、トリガー内でのみ有効であることに注意してください。以下の疑似コードの例を見てみましょう。

```
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
```

この例は「疑似コード」であり、実際のトリガーではいくつかの変更と追加 (カーソルの実行、クローズ、およびドロップを行うコードなど) が必要になります。この例の完全で有効な SQL スクリプトを以下に示します。

REFERENCING 節を持つトリガーの例

```
-- 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;
```

トリガーおよびトランザクション

トリガーを起動するために呼び出し側トランザクションからコミットを実行する必要はありません。トリガーを起動するのは DML ステートメントだけです。トリガー本体で COMMIT WORK を実行することもできません。

プロシージャー定義では、COMMIT ステートメントおよび ROLLBACK ステートメントを使用できます。ただしトリガー本体では、COMMIT ステートメントおよび ROLLBACK ステートメントを使用できません。使用できるのは WHENEVER SQLERROR ABORT ステートメントのみです。自動コミットがオンになっている場合は、トリガー内の各ステートメントが個別のステートメントとして処理されず、実行時にコミットされないので注意してください。代わりに、トリガー本体全体が、そのトリガーを起動した INSERT、 UPDATE、または DELETE ステートメントの一部として実行されます。トリガー全体 (およびそれを起動したステートメント) がコミットされるか、またはロールバックされます。

再帰エラーおよび並行性競合エラー

DML ステートメントが、トリガーを発生させるような行の更新または削除を行った場合、そのトリガーで同じ行を更新または削除できません。この場合、AFTER トリガー・イベントは再帰エラーを発生させ、BEFORE トリガー・イベントは並行性競合エラーを発生させることがあります。

以下のセクションで、これらの条件について説明し、このような問題を発生させるトリガーの例を示し、再帰エラーまたは並行性競合エラーを発生させるトリガー状態と発生させないトリガー状態を示す表 (70 ページの『トリガーの事例の要約』を参照) を示します。

トリガーおよび再帰

コードの一部分が自身を再度実行する場合、そのコードは「再帰的」です。例えば、自身を呼び出すストアード・プロシージャーは再帰的です。ストアード・プロシージャーでは再帰を利用すると便利な場合があります。一方、トリガーではもう少し複雑なタイプの再帰を作成できますが、 solidDB サーバーではこの再帰を無効として禁止しています。同じレコードで同じトリガーを再実行するステートメントを含んだトリガーは再帰的です。例えば、あるレコードの削除によって起動された DELETE トリガーが同じレコードを削除しようとすると、そのトリガーは再帰的となります。

データベース・サーバーでトリガーでの再帰が許可されていると、サーバーが「無限ループ」に入り、トリガーを起動したステートメントの実行が終了しなくなる場合があります。トリガーがそのトリガーを起動したステートメントと同じタイプのアクション (DELETE など) を同じ SQL ステートメント内で実行してそのステートメントと競合した場合は、並行性競合エラーが発生します。例えば、レコードが削除されたときに起動されるトリガーを作成したとします。あるレコードが削除されたことでそのトリガーが起動され、同じレコードを削除しようとすると、実質的には 1 つのレコードを 2 つの削除ステートメントが同時に削除しようと競合することになり、並行性競合が発生します。以下のセクションでは、問題のある DELETE トリガーの例を示します。

再帰の原因となる問題のあるトリガーの例

このセクションの例では、トリガーに関連する多数の制限事項やルールのごく一部のみを説明します。

このシナリオでは、ある従業員が辞職したので、その従業員の医療保険を解約する必要があります。また、この従業員の扶養家族の医療保険も解約する必要があります。この状況に対応するビジネス・ルールは、トリガーを作成することで実装されます。このトリガーは、従業員のレコードが削除されるときに実行され、トリガー内のステートメントによって従業員の扶養家族が削除されます (この例では、従業員とその扶養家族が同じ表に格納されていることを前提としています。現実には、扶養家族は別の表に格納されるのが一般的です。また、この例では各家族の姓がユニークであることを前提としています)。

```
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);
  -- ... カーソルをクローズしてドロップします。
 END;
```

従業員「John Smith」が辞職し、この従業員の医療保険が削除されるとします。「John Smith」を削除すると、削除の直後にトリガーが呼び出され、「John Smith」という名前の人員をこの従業員の扶養家族だけでなく従業員本人も含めてすべて削除しようとします。これは、この従業員の名前が WHERE 節の条件と一致するためです。

この従業員のレコードの削除が試行されるたびに、このトリガーが再度起動されます。つまりこのコードは、トリガーの再起動、および削除の再試行によって、再帰的に従業員の削除を試行することになります。データベース・サーバーでこの操作が禁止されていなかったりこの状況が検出されなかったりすると、サーバーは無限ループに入る可能性があります。サーバーでこの状況が検出された場合は、「ネストされたトリガーが多すぎます」などの適切なエラーが表示されます。

同様の状況は UPDATE でも発生します。レコードが更新されるたびに売上税を加算するトリガーがあるとします。再帰エラーを引き起こす例を以下に示します。

```
CREATE TRIGGER do_not_do_this_either ON invoice
     AFTER UPDATE
     REFERENCING NEW total_price AS new_total_price
     BEGIN
         -- 8% の売上税を加算します。
         EXEC SQL PREPARE upd_curs1
          UPDATE invoice SET total_price = 1.08 * total_price
           WHERE ...;
         -- ... カーソルを実行、クローズ、およびドロップします...
     END;
```

このシナリオでは、Ann Jones という顧客から注文を変更する電話が入ります。新しい価格 (売上税込み) は、新しい小計に 1.08 を乗算することで計算されます。この新しい合計価格でレコードは更新されます。レコードが更新されるたびにトリガーが起動されるため、レコードを一度更新すると、トリガーがそのレコードを再度更新し、更新が無限ループで繰り返されます。

AFTER トリガーは再帰やループの原因となる可能性がありますが、BEFORE トリガーはどうでしょうか。BEFORE トリガーは、場合によって並行性の問題を引き起こす可能性があります。従業員とその扶養家族の医療保険を削除する最初の例のトリガーに戻ってみましょう。このトリガーが BEFORE トリガー (AFTER トリガーではなく) である場合は、従業員が削除される直前にトリガーが実行され、John Smith という名前の人物が全員削除されます。トリガーの実行後に、エンジンは従業員 John Smith 本人をドロップする当初のタスクを再開しますが、この従業員が存在しないか、そのレコードを削除できない (削除対象として既にマークされているため) ことを検出します。つまり、同じレコードを削除する 2 つの操作が存在するために、並行性競合が生じます。

トリガーの事例の要約

前のセクションで説明した例に加えて、UPDATE と DELETE の他に INSERT も関係する多数の事例を以下の表にまとめました。

この表は以下の 5 つの列で構成されています。

- *トリガー・モード* (つまり BEFORE または AFTER)
- *操作* (INSERT、DELETE、または UPDATE)
- *トリガー・アクション* (トリガー自体が実行する操作 (挿入されたレコードの更新など))
- *ロック・タイプ* (「オプティミスティック」または「ペシミスティック」)
- *結果* (例えば、トリガー・アクションの成功、または前のセクションで説明したような再帰エラーなどを原因とするトリガー・アクションの失敗)

この表のトリガー項目の解釈について詳しくは、この章の後半の『*項目例 1*』を参照してください。

表 11. BEFORE/AFTER トリガーの INSERT/UPDATE/DELETE 操作

トリガー・モード	操作	トリガー・アクション	ロック・タイプ	結果
AFTER	INSERT	値に数値を加算して同じ行を UPDATE	オプティミスティック	レコードが更新されます。
AFTER	INSERT	値に数値を加算して同じ行を UPDATE	ペシミスティック	レコードが更新されます。
BEFORE	INSERT	値に数値を加算して同じ行を UPDATE	オプティミスティック	レコードは更新されません。これは、トリガー本体の UPDATE の WHERE 条件から NULL の結果セットが返されるためです (目的の行がまだ表に挿入されていないため)。

表 11. BEFORE/AFTER トリガーの INSERT/UPDATE/DELETE 操作 (続き)

トリガー・モード	操作	トリガー・アクション	ロック・タイプ	結果
BEFORE	INSERT	値に数値を加算して同じ行を UPDATE	ペシミスティック	レコードは更新されません。これは、トリガー本体の UPDATE の WHERE 条件から NULL の結果セットが返されるためです (目的の行がまだ表に挿入されていないため)。
AFTER	INSERT	挿入された同じ行を DELETE	オプティミスティック	レコードが削除されます。
AFTER	INSERT	挿入された同じ行を DELETE	ペシミスティック	レコードが削除されます。
BEFORE	INSERT	挿入された同じ行を DELETE	オプティミスティック	レコードは削除されません。これは、トリガー本体の DELETE の WHERE 条件から NULL の結果セットが返されるためです (目的の行がまだ表に挿入されていないため)。
BEFORE	INSERT	挿入された同じ行を DELETE	ペシミスティック	レコードは更新されません。これは、トリガー本体の UPDATE の WHERE 条件から NULL の結果セットが返されるためです (目的の行がまだ表に挿入されていないため)。
AFTER	INSERT	行を INSERT	オプティミスティック	ネストされたトリガーが多すぎます。
AFTER	INSERT	行を INSERT	ペシミスティック	ネストされたトリガーが多すぎます。
BEFORE	INSERT	行を INSERT	オプティミスティック	ネストされたトリガーが多すぎます。
BEFORE	INSERT	行を INSERT	ペシミスティック	ネストされたトリガーが多すぎます。
AFTER	UPDATE	値に数値を加算して同じ行を UPDATE	オプティミスティック	Solid® 表エラーを生成: ネストされたトリガーが多すぎます。
AFTER	UPDATE	値に数値を加算して同じ行を UPDATE	ペシミスティック	Solid 表エラーを生成: ネストされたトリガーが多すぎます。

表 11. BEFORE/AFTER トリガーの INSERT/UPDATE/DELETE 操作 (続き)

トリガー・モード	操作	トリガー・アクション	ロック・タイプ	結果
BEFORE	UPDATE	値に数値を加算して同じ行を UPDATE	オプティミスティック	レコードは更新されますが、ネストされたループにはなりません。これは、トリガー本体の WHERE 条件から NULL の結果セットが返されて行が更新されず、トリガーが再帰的に起動されないからです。
BEFORE	UPDATE	値に数値を加算して同じ行を UPDATE	ペシミスティック	レコードは更新されますが、ネストされたループにはなりません。これは、トリガー本体の WHERE 条件から NULL の結果セットが返されて行が更新されず、トリガーが再帰的に起動されないからです。
AFTER	UPDATE	更新された同じ行を DELETE	オプティミスティック	レコードが削除されます。
AFTER	UPDATE	更新された同じ行を DELETE	ペシミスティック	レコードが削除されます。
BEFORE	UPDATE	更新された同じ行を DELETE	オプティミスティック	並行性競合エラー。
BEFORE	UPDATE	更新された同じ行を DELETE	ペシミスティック	並行性競合エラー。
AFTER	DELETE	同じ値で行を INSERT	オプティミスティック	削除した後に同じレコードが挿入されます。
AFTER	DELETE	同じ値で行を INSERT	ペシミスティック	トリガーを起動した時点でハングします。
BEFORE	DELETE	同じ値で行を INSERT	オプティミスティック	削除した後に同じレコードが挿入されます。
BEFORE	DELETE	同じ値で行を INSERT	ペシミスティック	トリガーを起動した時点でハングします。
AFTER	DELETE	同じ値で行を INSERT	オプティミスティック	レコードが削除されます。
AFTER	DELETE	値に数値を加算して同じ行を UPDATE	ペシミスティック	レコードが削除されます。
BEFORE	DELETE	値に数値を加算して同じ行を UPDATE	オプティミスティック	レコードが削除されます。

表 11. BEFORE/AFTER トリガーの INSERT/UPDATE/DELETE 操作 (続き)

トリガー・モード	操作	トリガー・アクション	ロック・タイプ	結果
BEFORE	DELETE	値に数値を加算して同じ行を UPDATE	ペシミスティック	レコードが削除されます。
AFTER	DELETE	同じ行を DELETE	オプティミスティック	ネストされたトリガーが多すぎます。
AFTER	DELETE	同じレコードを DELETE	ペシミスティック	ネストされたトリガーが多すぎます。
BEFORE	DELETE	同じレコードを DELETE	オプティミスティック	並行性競合エラー。
BEFORE	DELETE	同じレコードを DELETE	ペシミスティック	並行性競合エラー。

次に、この表の項目の 1 つを例に挙げて説明します。

表 12. 項目例 1

トリガー	操作	トリガー・アクション	ロック・タイプ	結果
AFTER	INSERT	値に数値を加算して同じ行を UPDATE	オプティミスティック	レコードが更新されます。

これは、INSERT 操作の実行後 (AFTER) に起動されるトリガーです。トリガーの本体には、挿入された行を更新するステートメントが含まれています (つまり、トリガーを起動した行と同じ行)。ロック・タイプが「オプティミスティック」の場合は、結果的にレコードが更新されます (競合は生じないため、ロック方式がオプティミスティックでもペシミスティックでも違いはありません)。

この場合、挿入した行を更新していますが、再帰の問題は発生しません。トリガーを「起動」するアクションがトリガー内部で実行されるアクションと同じでないため、再帰/ループの状況にはなりません。

この表からもう 1 つの例を示します。

表 13. 項目例 2

トリガー	操作	トリガー・アクション	ロック・タイプ	結果
BEFORE	INSERT	値に数値を加算して同じ行を UPDATE	オプティミスティック	レコードは更新されません。これは、トリガー本体の UPDATE の WHERE 条件から NULL の結果セットが返されるためです (目的の行がまだ表に挿入されていないため)。

この場合は、レコードを挿入しようとしますが、挿入が実行される前にトリガーが実行されます。このトリガーはレコードを更新しようとします (例えば売上税をレコードに追加するなど)。ただし、レコードはまだ挿入されていないため、トリガーの UPDATE コマンドはレコードを見つけられず、売上税は追加されません。このため、結果はトリガーが起動されなかった場合と同じになります。エラー・メッセージが生成されないので、ユーザーはトリガーが目的の操作を実行しなかったことにすぐに気付かない可能性があります。

トリガーの誤り

以下の例では、BEFORE UPDATE トリガーで同じ行が削除されるという誤りがトリガー・ロジックに発生します。これによって、solidDB で並行性競合エラーが生成されます。

トリガーの誤り

```
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;
```

注:

更新/削除される行が通常の列ではなくユニーク・キーに基づいている場合 (上の例のように) は、solidDB で次のエラー・メッセージが生成されます: *1001: key value not found*

再帰エラーや並行性競合エラーを回避するために、アプリケーション・ロジックを確認し、アプリケーションで 2 つのトランザクションが同じ行を更新または削除しないように予防措置を取ってください。

エラー処理

プロシージャーがトリガーにエラーを返した場合、トリガーは呼び出し側の DML コマンドがエラーで失敗するようにします。DML ステートメントの実行中に、自動的にエラーを返すには、WHENEVER SQLERROR ABORT ステートメントをトリガー本体で使用する必要があります。そうしない場合は、各プロシージャー呼び出しまたは SQL ステートメントの後、トリガー本体でエラーを明示的に検査する必要があります。

トリガー本体の一部であるユーザー作成のビジネス・ロジックで発生するエラーの場合は、RETURN SQLERROR ステートメントを使用する必要があります。詳しくは、76 ページの『トリガー内からのエラーの発生』を参照してください。

RETURN SQLERROR が指定されていない場合、SQL ステートメントの実行が失敗したときに、システムはデフォルトのエラー・メッセージを返します。現行 DML ステートメントによるデータベースへの変更は取り消され、トランザクションはアクティブのままです。実際には、トリガーの実行が失敗してもトランザクションはロールバックされませんが、現在実行中のステートメントはロールバックされます。

注:

トリガー SQL ステートメントは、呼び出し側トランザクションの一部です。トリガーが原因で、またはトリガーの外部で生成されたその他のエラーが原因で呼び出し側 DML ステートメントが失敗した場合、トリガーのすべての SQL ステートメントと失敗した呼び出し側 DML コマンドがロールバックされます。

トリガーのプロシージャー内で実行された DML ステートメントのコミットまたはロールバックは、呼び出し側トランザクションが実行します。ただし、トリガーを呼び出した DML コマンドが、関連付けられているトリガーの結果として失敗した場合、このルールは適用されません。この場合は、そのトリガーのプロシージャー内で実行されたすべての DML ステートメントが自動的にロールバックされます。

COMMIT ステートメントおよび ROLLBACK ステートメントは、トリガー本体の外部で実行する必要があり、トリガー本体の中で実行することはできません。トリガー本体の中、あるいはトリガー本体または別のトリガーから呼び出されたプロシージャーの内部で COMMIT または ROLLBACK を実行すると、ランタイム・エラーが発生します。

ネストしたトリガーと再帰的トリガー

トリガーが無限ループに入ると、solidDB サーバーは 16 レベルのネスト (つまり MaxNestedTriggers システム・パラメーターの最大値) に到達した時点でその再帰的アクションを検出します。例えば、表 T1 への挿入を行おうとするとトリガーがアクティブ化され、トリガーはストアード・プロシージャーを呼び出し、そのストアード・プロシージャーも再帰的にトリガーをアクティブ化して、表 T1 への挿入を試みる可能性があります。solidDB サーバーは、ユーザーの挿入の試みに対してエラーを返します。

ネストした一連のトリガーがいずれかの時点で失敗した場合、solidDB サーバーはトリガーを最初にアクティブ化したコマンドをロールバックします。

トリガーの特権およびセキュリティー

トリガーはユーザーがデータの INSERT、UPDATE、または DELETE を試みることでアクティブ化できるため、トリガーを実行するための特権は必要ありません。

ユーザーがトリガーを呼び出すと、トリガーが定義されている表の所有者の特権がそのユーザーに与えられます。アクション・ステートメントは、トリガーをアクティブ化したユーザーではなく、表の所有者のために実行されます。ただし、ストア

ード・プロシージャーを使用するトリガーを作成するには、トリガーの作成者が以下のいずれかの条件を満たしている必要があります。

- DBA 特権を持っている。
- トリガーが定義されている表の所有者である。
- 表に対するすべての特権を付与されている。

DBA 権限を持つ作成者が別のユーザー向けに表を作成すると、solidDB サーバーは TRIGGER コマンドに指定されている修飾されていない名前がそのユーザーに属していると見なします。例えば、以下のコマンドは DBA 権限の下で実行されます。

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

EMP 表は修飾されていないため、solidDB サーバーは修飾された表名が DBA.EMP ではなく A.EMP であると見なします。

トリガー内からのエラーの発生

トリガーの実行中に、エラーを受け取ることがあります。エラーは、SQL ステートメントまたはビジネス・ロジックの実行が原因で発生することがあります。

以下の SQL ステートメントを使用して、プロシージャー変数でエラーを受け取ることができます。

```
RETURN SQLERROR error_string
```

または

```
RETURN SQLERROR char_variable
```

エラーは、以下のフォーマットで返されます。

```
User error: error_string
```

ユーザーが RETURN SQLERROR ステートメントをトリガー本体で指定しない場合、トラップされたすべての SQL エラーは、システムが決定するデフォルトの error_string で発生します。詳しくは、solidDB 製品の資料の付録の『エラー・コード』を参照してください。

トリガーの例

トリガーの例

この例では、単純なトリガーが動作する仕組みを説明します。ここで示すトリガーには、正常に機能するものとエラーを含んでいるものがあります。この例の正常なトリガーとして、表 (*trigger_test*) が作成され、その表に 6 個のトリガーが作成されます。それぞれのトリガーは、起動されると別の表 (*trigger_output*) にレコードを挿入します。トリガーを起動する DML ステートメント (INSERT、UPDATE、および DELETE) が実行された後に、*trigger_output* 表のレコードをすべて選択することでトリガーの結果が表示されます。

```
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;
-- 想定される各トリガー用の列を含んだ表を作成します。
```

```
-- (例えば、BI は、INSERT 操作に対して BEFORE トリガーとして
-- 実行されるトリガーです。)
CREATE TABLE TRIGGER_TEST(
        XX VARCHAR,
        BI VARCHAR,  -- BI = Before Insert (挿入前)
        AI VARCHAR,  -- AI = After Insert (挿入後)
        BU VARCHAR,  -- BU = Before Update (更新前)
        AU VARCHAR,  -- AU = After Update (更新後)
        BD VARCHAR,  -- BD = Before Delete (削除前)
        AD VARCHAR   -- AD = After Delete (削除後)
);
COMMIT WORK;

-- BEFORE トリガー・エラー用の表
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;

-- 「AFTER X」トリガー・エラー用の表
CREATE TABLE TRIGGER_ERR_A_TEST(
        XX VARCHAR,
        BI VARCHAR,  -- Before Insert (挿入前)
        AI VARCHAR,  -- After Insert (挿入後)
        BU VARCHAR,  -- Before Update (更新前)
        AU VARCHAR,  -- After Update (更新後)
        BD VARCHAR,  -- Before Delete (削除前)
        AD VARCHAR   -- After Delete (削除後)
);

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

CREATE TABLE TRIGGER_OUTPUT(
        TEXT VARCHAR,
        NAME VARCHAR,
        SCHEMA VARCHAR
);
COMMIT WORK;

------------------------------------------------------------------
-- 正常なトリガー
------------------------------------------------------------------
-- INSERT 操作に対して「BEFORE」トリガーを作成します。レコードが
-- trigger_test という表に挿入されるときに、このトリガーが
-- 起動されます。このトリガーは、起動されるとレコードを
-- trigger_output 表に挿入して、トリガーが実際に実行されたことを示します。

"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
        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;

------------------------------------------------------------------
-- トリガーを作成するこの操作は失敗します。ステートメントで
-- ERRSTR というエラー変数に間違ったデータ型が指定されています。
------------------------------------------------------------------

"CREATE TRIGGER TRIGGER_ERR_AU ON TRIGGER_ERR_A_TEST
        AFTER UPDATE
        REFERENCING NEW AU AS NEW_AU
BEGIN
 -- 以下の行は正しくありません。ERRSTR は INTEGER ではなく
```

```
 -- VARCHAR として宣言する必要があります。DECLARE ERRSTR INTEGER;
 -- ...
        RETURN SQLERROR ERRSTR;
END";
COMMIT WORK;

-----------------------------------------------------------------
-- エラー・メッセージを返すトリガー
-----------------------------------------------------------------
"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;


-----------------------------------------------------------------
-- 正常なトリガーのテストです。以下の INSERT、UPDATE、および DELETE
-- の各ステートメントは強制的にトリガーを起動します。SELECT
-- ステートメントによって、trigger_test 表および trigger_output 表の
-- レコードが表示されます。
-----------------------------------------------------------------

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

-- trigger_test 表に挿入されたレコードを表示します。
-- (trigger_output のレコードは後で表示されます。)

SELECT * FROM TRIGGER_TEST;
COMMIT WORK;

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

-- trigger_test 表に挿入されたレコードを表示します。
-- (trigger_output のレコードは後で表示されます。)

SELECT * FROM TRIGGER_TEST;
COMMIT WORK;

DELETE FROM TRIGGER_TEST;
COMMIT WORK;

SELECT * FROM TRIGGER_TEST;

-- トリガーが実行されて、trigger_output 表に値が追加された
-- ことを示します。実行されたトリガーごとに 1 つずつ、
-- つまり 6 つのレコードが表示されます。6 つのトリガーとは、
--  BI、AI、BU、AU、BD、AD です。

SELECT * FROM TRIGGER_OUTPUT;
COMMIT WORK;

-----------------------------------------------------------------
-- エラー・トリガーのテスト
-----------------------------------------------------------------

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

トリガーのドロップ

表に定義されているトリガーをドロップするには、DROP TRIGGER コマンドを使用します。このコマンドは、システム・カタログからトリガーをドロップします。

表からトリガーをドロップするには、表の所有者であるか、DBA 権限を持っている必要があります。

構文は以下のとおりです。

```
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
```

trigger_name は、表で定義されているトリガーの名前です。

トリガーがスキーマの一部である場合は、以下のようにスキーマ名を指定します。

schema_name.trigger_name

トリガーがカタログの一部である場合は、以下のようにカタログ名を指定します。

catalog_name.schema_name.trigger_name

トリガーのドロップおよび再作成

```
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;
```

トリガー属性の変更

ALTER TRIGGER コマンドを使用して、トリガー属性を変更できます。有効な属性は、ENABLED および DISABLED トリガーです。

ALTER TRIGGER SET DISABLED コマンドを実行すると、solidDB サーバーは、アクティブ化 DML ステートメントが発行されたとき、トリガーを無視します。ALTER TRIGGER SET ENABLED ステートメントを使用すると、現在アクティブでないトリガーを使用可能にすることができます。

表からのトリガーを変更するには、その表の所有者であるか、DBA 権限を持つユーザーであることが必要です。

```
alter_trigger ::=
    ALTER TRIGGER trigger_name_att SET ENABLED | DISABLED
trigger_name_attr ::= [catalog_name.[schema_name]]trigger_name
```

以下に例を示します。

```
ALTER TRIGGER trig_on_employee SET ENABLED;
```

トリガー情報の入手

トリガー情報を入手するには、具体的な情報を返すトリガー関数を使用し、トリガー・システム表に対して照会を行います。このセクションでは、それらの各ソースについて説明します。

トリガー関数

システムでサポートされている以下のトリガー・スタック関数は、分析とデバッグに役立ちます。

注: トリガー・スタックとは、実行されるか、または実行対象として検出されるかにかかわらずキャッシュされるトリガーです。トリガー・スタック関数は、他の関数と同じようにアプリケーション・プログラムで使用できます。

以下に各関数を示します。

- TRIG_COUNT()

  この関数は、トリガー・スタック内のトリガー (現在のトリガーも含む) の数を返します。戻り値は整数です。

- TRIG_NAME(n)

  この関数は、トリガー・スタック内の n 番目のトリガー名を返します。最初のトリガー位置またはオフセットはゼロです。

- TRIG_SCHEMA(n)

  この関数は、トリガー・スタック内の n 番目のトリガー・スキーマ名を返します。最初のトリガー位置またはオフセットはゼロです。戻り値はストリングです。

SYS_TRIGGERS システム表

トリガーは、SYS_TRIGGERS と呼ばれるシステム表に格納されます。SYS_TRIGGERS システム表のメタデータを以下に示します。

表 14. SYS_TRIGGERS システム表のメタデータ

列名	データ型	説明
ID	INTEGER	ユニークな表 ID (主キー)。
TRIGGER_NAME	WVARCHAR	トリガー名 (スキーマと組み合わせてユニーク)。
TRIGGER_TEXT	LONG WVARCHAR	トリガー本体。
TRIGGER_BIN	LONG VARBINARY	コンパイルされた形式のトリガー。

表 14. SYS_TRIGGERS システム表のメタデータ (続き)

列名	データ型	説明
TRIGGER_SCHEMA	WVARCHAR	トリガーが作成されたスキーマ。
TRIGGER_CATALOG	WVARCHAR	トリガーが作成されたカタログ。
CREATIME	TIMESTAMP	トリガーの作成時刻。
TYPE	INTEGER	将来の使用のために予約済み。
REL_ID	INTEGER	関係 ID (タイプと組み合わせてユニーク)。
TRIGGER_ENABLED	WVARCHAR	トリガーが使用可能である場合は「YES」、使用不可である場合は「NO」。

トリガー・パラメーターの設定

ネストされたトリガーの最大数の設定

トリガーが他のトリガーを呼び出したり、トリガーが自分自身を呼び出したり (再帰トリガー) することができます。ネストされたトリガーまたは再帰トリガーの最大数は、solid.ini の SQL セクションの MaxNestedTriggers システム・パラメーターで設定できます。

```
[SQL]
MaxNestedTriggers = n;
```

n はネストされたトリガーの最大数です。

ネストされたトリガーのデフォルトの数は 16 です。

トリガー・キャッシュの設定

solidDB サーバーでは、トリガーが別のキャッシュにキャッシュされます。各ユーザーにトリガー専用のキャッシュが用意されます。トリガーが実行されると、トリガー・プロシージャー・ロジックがトリガー・キャッシュにキャッシュされ、トリガーが再び実行されるときに再利用されます。

トリガー・キャッシュのサイズは、solid.ini の SQL セクションの TriggerCache システム・パラメーターで設定できます。

```
[SQL]
TriggerCache = n;
```

n はキャッシュに確保されるトリガーの数です。

据え置きプロシージャー呼び出し

コミットされたトランザクションの最後で、特定のアクションを実行できます。例えば、トランザクションが「マスター」パブリケーションのデータを更新する場合、マスター・データが更新されたことをレプリカに通知できます。solidDB では、START AFTER COMMIT ステートメントで、現行トランザクションがコミットされるときに実行する SQL ステートメントを指定できます。指定された SQL ステートメントは、START AFTER COMMIT の「本体」と呼ばれます。本体は、個別の接続で、非同期に実行されます。

例えば、トランザクションがコミットされるときに、my_proc() というストアード・プロシージャーを呼び出すには、以下のステートメントを作成します。

```
START AFTER COMMIT NONUNIQUE CALL
    my_proc;
```

このステートメントは、トランザクション内のどこででも使用できます。最初のステートメント、最後のステートメント、または間のステートメントにできます。トランザクションのどこに START AFTER COMMIT ステートメントがあるかにかかわらず、「本体」(my_proc の呼び出し) は、トランザクションがコミットされたときにだけ実行されます。上の例では個別の行に本体を置きましたが、構文的にこのようにする必要があるわけではありません。

ステートメントの本体は、START AFTER COMMIT ステートメント自身と同時には実行されないため、START AFTER COMMIT コマンドには、「定義」フェーズと「実行」フェーズという 2 つの異なるフェーズがあります。START AFTER COMMIT の定義フェーズでは、本体を指定しますが、実行はしません。作成フェーズは、トランザクション内の任意の場所に出現します。言い換えれば、「START AFTER COMMIT ...」は、同じトランザクションにある他の SQL ステートメントに対して、任意の相対順序で配置できます。

実行フェーズでは、START AFTER COMMIT ステートメントの本体が実際に実行されます。実行フェーズは、トランザクションの COMMIT WORK ステートメントが実行されたときに発生します。(START AFTER COMMIT を自動コミット・モードで実行することもできますが、そうする理由はほとんどありません。)

以下に、トランザクション内で START AFTER COMMIT ステートメントを使用する例を示します。

```
-- 任意の有効な SQL ステートメント
...
-- 作成フェーズ。関数 my_proc() は、実際にはここでは呼び出されません。
START AFTER COMMIT NONUNIQUE CALL my_proc(x, y);
...
-- 任意の有効な SQL ステートメント

-- 実行フェーズ。ここでトランザクションが終了し、
-- my_proc() 呼び出しの実行が開始されます。
COMMIT WORK;
```

START AFTER COMMIT は、トランザクションが正常にコミットされるまで、実行されません。START AFTER COMMIT を含むトランザクションがロールバックされた場合、START AFTER COMMIT の本体は実行されません。更新したデータ

をレプリカからマスターに伝搬する場合、コミットされたときにだけデータが伝搬されるため、これは利点になります。トリガーを使用して伝搬を開始すると、コミットされる前にデータが伝搬されます。

START AFTER COMMIT コマンドは、現行トランザクション、つまり、その中で START AFTER COMMIT コマンドが発行されたトランザクションにだけ適用されます。後続のトランザクションや、他の接続で現在開いている他のトランザクションには適用されません。

START AFTER COMMIT コマンドでは、COMMIT が発生したときに実行する SQL ステートメントを 1 つだけ指定できます。ただし、その SQL ステートメントでストアード・プロシージャーを呼び出すことができ、そのストアード・プロシージャーには、他のストアード・プロシージャーの呼び出しも含め、多数のステートメントを含めることができます。さらに、トランザクションごとに複数の START AFTER COMMIT コマンドを含めることもできます。各 START AFTER COMMIT ステートメントの本体は、トランザクションがコミットされるときに実行されます。ただし、これらの本体は、独立して非同期で実行されます。必ずしも、対応する START AFTER COMMIT ステートメントと同じ順序で実行されるわけではなく、実行がオーバーラップすることもあります (次の本体が開始する前に、前の本体が終了しているとは限りません)。

START AFTER COMMIT の一般的な使用法として、「*IBM solidDB 拡張レプリケーション・ユーザー・ガイド*」で説明する「プル同期通知」(「プッシュ同期」) の実装があります。

START AFTER COMMIT の本体がストアード・プロシージャー呼び出しの場合、そのプロシージャーは、ローカル・プロシージャーでも、リモート・レプリカ (またはマスター) のリモート・プロシージャーでもかまいません。

プル同期通知を使用する場合、同じプロシージャーを多くのレプリカで呼び出すことができます。そのためには、やや間接的な方式を使用します。最も単純な方式は、レプリカで多数のプロシージャーを呼び出す 1 つのローカル・プロシージャーを作成する方式です。例えば、START AFTER COMMIT ステートメントの本体が「CALL my_proc」の場合、my_proc を以下のように作成できます。

```
CREATE PROCEDURE my_proc
BEGIN
CALL update_inventory(x) AT replica1;
CALL update_inventory(x) AT replica2;
CALL update_inventory(x) AT replica3;
END;
```

このアプローチは、レプリカのリストが静的な場合、正常に機能します。ただし、将来、新しいレプリカを追加する場合は、プロパティーに基づいてレプリカの「グループ」を更新する方が簡単です。この方式では、特定のプロパティーを含む新しいレプリカを追加して、既存のストアード・プロシージャーがその新しいレプリカで動作するようにできます。これを行うには、START AFTER COMMIT の FOR EACH REPLICA 節と、リモート・ストアード・プロシージャー呼び出しの DEFAULT 節という 2 つの機能を使用します。

START AFTER COMMIT で FOR EACH REPLICA 節を使用すると、WHERE 節の条件に合うレプリカごとに 1 回、ステートメントが実行されます。このステート

メントは、レプリカごとに 1 回実行されるのであって、各レプリカで 1 回ずつ実行されるわけではありません。CALL ステートメントに「AT node-ref」節がない場合は、ストアード・プロシージャーがローカルに呼び出されます。つまり、START AFTER COMMIT が実行されたサーバーと同じサーバーで呼び出されます。ストアード・プロシージャーを確実に各レプリカで 1 回ずつ呼び出すには、DEFAULT 節を使用する必要があります。これを実行する標準的な方法は、DEFAULT 節を使うリモート・プロシージャー呼び出しを含むローカル・ストアード・プロシージャーを作成することです。例えば、my_local_proc に、以下が含まれているとします。

```
CALL update_sales_statistics AT DEFAULT;
```

このとき、START AFTER COMMIT ステートメントが、以下のようだとします。

```
START AFTER COMMIT FOR EACH REPLICA
WHERE region = 'north'
UNIQUE
CALL my_local_proc;
```

WHERE 節は、以下のとおりです。

```
WHERE region = 'north'
```

したがって、以下のプロパティー

```
region = 'north'
```

が含まれているレプリカごとに、my_local_proc というストアード・プロシージャーを呼び出します。次にこのローカル・プロシージャーで、以下を実行します。

```
CALL update_sales_statistics() AT DEFAULT
```

キーワード DEFAULT は、レプリカの名前に解決されます。 my_local_proc が START AFTER COMMIT の本体内から呼び出されるたびに、DEFAULT キーワードは、「region = 'north'」というプロパティーが含まれている異なるレプリカの名前になります。

「region = 'north'」のようなプロパティーと値のペアについて詳しくは、「*IBM solidDB 拡張レプリケーション・ユーザー・ガイド*」の『レプリカ・プロパティー名』のセクションを参照してください。

すべてのレプリカに、update_sales_statistics() というプロシージャーがあるとは限らないことに注意してください。その場合、プロシージャーがあるレプリカでのみ、プロシージャーが実行されます。(マスターは、プロシージャーのコピーを各レプリカに送信しません。マスターは、既存のプロシージャーを呼び出すだけです。)

また、update_sales_statistics() というプロシージャーがあるすべてのレプリカに同じプロシージャーがあるとは限らないことに注意してください。レプリカごとに、プロシージャーの独自のカスタム・バージョンを持つ場合があります。

もちろん、各レプリカでステートメントを実行する前に、レプリカへの接続が確立されます。

START AFTER COMMIT コマンドを使用して複数のレプリカを呼び出す場合、CALL コマンドの構文でオプション・キーワード「DEFAULT」を使用できます。例えば、以下を使用するとします。

```
START AFTER COMMIT
  FOR EACH REPLICA
  WHERE location = 'India'
  UNIQUE CALL push;
```

ローカル・プロシージャー「push」で、キーワード「DEFAULT」を使用できます。このキーワードは、問題のレプリカの名前を含む変数として機能します。

```
CREATE PROCEDURE push
BEGIN
EXEC SQL EXECDIRECT CALL remoteproc AT DEFAULT;
END
```

プロシージャー「push」は、プロパティー名「location」の値が「India」であるレプリカごとに 1 回呼び出されます。プロシージャーが呼び出されるたびに、「DEFAULT」はレプリカの名前に設定されます。したがって、

```
CALL remoteproc AT DEFAULT;
```

は、その特定のレプリカのプロシージャーを呼び出します。

レプリカ・プロパティーは、以下のステートメントを使用して、マスターで設定できます。

```
SET SYNC PROPERTY propname = 'value' FOR REPLICA replica_name;
```

例を示します。

```
SET SYNC PROPERTY location = 'India' FOR REPLICA asia_hq;
```

START AFTER COMMIT で指定されたステートメントは、独立したトランザクションとして実行されます。START AFTER COMMIT コマンドが含まれていたトランザクションの一部ではありません。この独立したトランザクションは、自動コミット・モードがオンである場合と同じように実行されます。つまり、このステートメントで実行した作業を、COMMIT WORK で明示的にコミットする必要はありません。

ただし、別の観点から見ると、このステートメントの実行はトランザクションとは似ていません。まず、ステートメントの実行が完了するという保証がありません。ステートメントは、独立したバックグラウンド・タスクとして起動されます。サーバーが異常終了した場合、またはその他の理由でステートメントを実行できない場合、ステートメントは実行を完了せずに消滅します。

2 番目に、ステートメントはバックグラウンド・タスクとして実行されるため、エラーを返すメカニズムがありません。3 番目に、ステートメントをロールバックする方法がありません。ステートメントの実行が完了すると、「トランザクション」ステートメントは、エラーが検出されたかどうかにかかわらず、自動コミットされます。(ステートメントがプロシージャー呼び出しの場合は、プロシージャー自身に COMMIT および ROLLBACK コマンドを含めることができることに注意してください。)

「RETRY」節を使用して、ステートメントが失敗した場合、複数回実行を試行することができます。RETRY 節を使用して、失敗したステートメントをサーバーが再試行する回数を指定できます。各再試行間の待ち時間を秒数で指定する必要があります。

RETRY 節を使用しない場合、サーバーはステートメントの実行を 1 回だけ試行し、ステートメントを廃棄します。例えば、ステートメントがリモート・プロシージャーを呼び出そうとして、リモート・サーバーが停止していた (または、ネットワークの問題で接続できなかった) 場合、ステートメントは実行されず、エラー・メッセージも表示されません。

START AFTER COMMIT に指定されたステートメントを含め、すべてのステートメントは、特定の「コンテキスト」で実行されます。コンテキストには、デフォルト・カタログ、デフォルト・スキーマなどの因子が含まれています。START AFTER COMMIT から実行されるステートメントの場合、ステートメントのコンテキストは、COMMIT WORK が START AFTER COMMIT 内のステートメントを実際に実行する時点ではなく、START AFTER COMMIT が実行される時点のコンテキストに基づきます。下の例で、「CALL FOO_PROC」は、カタログ foo_cat とスキーマ foo_schema で実行されます。bar_cat と bar_schema ではありません。

```
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;
```

UNIQUE/NONUNIQUE キーワードは、サーバーが同じコマンドを 2 回発行することを防止するかどうかを決定します。

<stmt> の前の UNIQUE キーワードは、実行中または実行の「保留中」の同一ステートメントがないときにだけ、ステートメントが実行されることを定義します。ステートメントは、単純なストリング比較で比較されます。そのため、例えば「call foo(1)」は「call foo(2)」と異なります。レプリカも比較で考慮されます。つまり、UNIQUE を使用しても、サーバーは、異なるレプリカで同じトリガー呼び出しを実行しなくなるわけではありません。「ユニーク」とは、ステートメントの実行のオーバーラップをブロックするだけであることに注意してください。現行の呼び出しの実行を終了した後で、同じステートメントが再度呼び出された場合、その実行ができなくなることはありません。

NONUNIQUE は、重複するステートメントをバックグラウンドで同時に実行できることを意味します。

例: 以下のステートメントは、すべて異なると見なされます。そのため、それぞれに UNIQUE キーワードが含まれていますが、実行されます。(name は、レプリカのユニークなプロパティーです。)

```
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;
```

しかし、以下のステートメントが、上記のステートメントと同じトランザクションで実行され、レプリカ R1、R2、R3 のいずれかにプロパティー「color='blue'」が含まれている場合、そのレプリカでは再び呼び出しは実行されません。

```
START AFTER COMMIT FOR EACH REPLICA WHERE color='blue'
UNIQUE call myproc;
```

また、一意性は、「自動」実行が「手動」実行とオーバーラップすることを防止しないことに注意してください。例えば、特定のパブリケーションからリフレッシュするコマンドを手動で実行し、同じパブリケーションからリフレッシュするリモート・ストアード・プロシージャーをマスターも呼び出した場合、手動リフレッシュが既に実行されているので、マスターは呼び出しを「スキップ」しません。一意性は、START AFTER COMMIT で開始されたステートメントにだけ適用されます。

START AFTER COMMIT ステートメントは、ストアード・プロシージャー内でも使用できます。例えば、トランザクションが正常に完了したときにだけイベントを通知するとします。この場合、トランザクションがコミットされたときにイベントを通知する (ただし、ロールバックされた場合は通知しない) START AFTER COMMIT ステートメントを実行するストアード・プロシージャーを作成できます。このコードは、以下のようになります。

このサンプルには、イベント・パラメーターを「受け取って」使用する例も含まれています。スクリプト 1 の「*wait_on_event_e*」というストアード・プロシージャーを参照してください。

```
-- このデモを正常に実行するには、2 つのユーザー/接続が必要です。
-- このデモには、5 つの別個の「スクリプト」が含まれていて、
-- 以下に示す順序で実行する必要があります。
--         ユーザー 1 が、最初のスクリプトを実行します。
--         ユーザー 2 が、2 番目のスクリプトを実行します。
--         ユーザー 1 が、3 番目のスクリプトを実行します。
--         ユーザー 2 が、4 番目のスクリプトを実行します。
--         ユーザー 1 が、5 番目のスクリプトを実行します。
-- 意外な位置に、いくつかの COMMIT WORK ステートメントが
-- あります。これらは、他のユーザーによる最新の変更を
-- 各ユーザーが参照できるようにするためのものです。COMMIT WORK
-- ステートメントがないと、場合によっては、一方のユーザーが
-- データベースの古い「スナップショット」を参照することになります。
--
-- 両方のユーザー/接続で、自動コミットはオフに設定してください。



-------------------- スクリプト 1 (ユーザー 1) --------------------
CREATE EVENT e (i int);
CREATE TABLE table1 (a int);

-- ここで、table1 に行を挿入します。挿入される値は、
-- パラメーターからプロシージャーにコピーされます。
"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";

-- ここで、「e」というイベントが通知されます。
"CREATE PROCEDURE post_event(i integer)
BEGIN
POST EVENT e(i);
END";

-- ここでは、ストアード・プロシージャー内で START AFTER COMMIT を
-- 使用する方法を示します。このプロシージャーと
-- COMMIT WORK を呼び出した後、サーバーはイベントを通知します。
"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";

-- ユーザー 2 がこのプロシージャーを呼び出すと、プロシージャーは
-- 「e」というイベントが通知されるまで待機してから
-- table1 にレコードを挿入するストアード・プロシージャーを呼び出します。
"CREATE PROCEDURE wait_on_event_e
BEGIN
-- イベント・パラメーターの保持に使用する変数を宣言します。
-- パラメーターはイベントが作成されたときに宣言されていますが、
-- それを変数として、イベントを受け取るプロシージャーの中で
-- 宣言する必要があります。
DECLARE i INT;
WAIT EVENT
    WHEN e (i) BEGIN
        -- イベントを受け取った後、表に行を挿入します。
        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;


-------------------- スクリプト 2 (ユーザー 2) --------------------
-- ユーザー 2 が、ユーザー 1 によって行われた変更を参照できるようにします。
COMMIT WORK;

-- ユーザー 1 がイベントを通知するまで待機します。
CALL wait_on_event_e;
-- 再度コミットは (まだ) しません。


-------------------- スクリプト 3 (ユーザー 1) --------------------
COMMIT WORK;

-- ユーザー 2 はイベント e を待機しています。
   ここで sac_demo というストアード・プロシージャーを
-- 実行して、 作業をコミットした後、
   ユーザー 2 がイベントを参照する必要があります。
-- START AFTER COMMIT ステートメントが非同期に実行されるため、
-- COMMIT WORK とそれに関連付けられている POST EVENT の間に
-- 少し遅延が生じることがあります。
CALL sac_demo;
COMMIT WORK;

-------------------- スクリプト 4 (ユーザー 2) --------------------
-- イベントを受け取った後で、inserter() を呼び出したときに既に行った
-- INSERT をコミットします。
COMMIT WORK;

-------------------- スクリプト 5 (ユーザー 1) --------------------
-- ユーザー 2 が挿入したデータが表示されるようにします。
COMMIT WORK;
```

```
-- ユーザー 2 が挿入したレコードを表示します。
SELECT * FROM table1;

COMMIT WORK;
```

START AFTER COMMIT について、知っておくべき重要事項がいくつかあります。

- 据え置きプロシージャー呼び出し (START AFTER COMMIT) の本体が実行されるときは、バックグラウンドで非同期に実行されます。これによって、サーバーは、据え置きプロシージャー呼び出しステートメントの終了を待たずに、プログラムの次にある SQL コマンドの実行をすぐに開始できます。また、サーバーを切断する前に完了を待つ必要がありません。これはほとんどの場合、利点となります。逆に、状況によってはこれが欠点になることが多少あります。例えば、据え置きプロシージャー呼び出しの本体が、プログラム中の後続の SQL コマンドに必要なレコードをロックする場合、据え置きプロシージャー呼び出しの本体をバックグラウンドで実行し、その間、次の SQL コマンドをフォアグラウンドで実行して、同じレコードにアクセスするために待機することは望ましくない場合があります。
- 実行するステートメントは、トランザクションが ROLLBACK ではなく、COMMIT で完了した場合にだけ実行されます。トランザクション全体が明示的にロールバックされた場合、またはトランザクションが異常終了したために暗黙的にロールバックされた場合 (接続の失敗など)、START AFTER COMMIT の本体は実行されません。
- 据え置きプロシージャー呼び出しが発生するトランザクションはロールバックできますが (この場合、据え置きプロシージャー呼び出しの本体は実行されません)、据え置きプロシージャー呼び出しが実行された場合、本体自身はロールバックできません。バックグラウンドで非同期に実行されるため、実行が開始された場合、本体を取り消すメカニズムまたはロールバックするメカニズムがありません。
- 据え置きプロシージャー呼び出しのステートメントが完了まで実行される、または「アトミック」トランザクションとして実行されるという保証はありません。例えば、サーバーが異常終了した場合、次にサーバーが始動したときに、ステートメントの実行は再開されません。また、サーバーが異常終了する前に完了したアクションが保持されることがあります。このような場合のデータの不整合を防ぐには、慎重にプログラムを作成し、データ保全性を保証する参照制約などの機能を適切に使用する必要があります。
- 自動コミット・モードで START AFTER COMMIT ステートメントを実行した場合、START AFTER COMMIT の本体は「すぐに」(START AFTER COMMIT が実行され、自動的にコミットされるとすぐに) 実行されます。これは一見、START AFTER COMMIT の本体を直接実行すればよいだけで、無用に思われます。しかし、これらには、小さな違いがあります。まず、my_proc の直接呼び出しは、同期的です。ストアード・プロシージャーが実行を終了するまで、サーバーは制御を返しません。my_proc を START AFTER COMMIT の本体として呼び出す場合、呼び出しは非同期的です。サーバーは my_proc の終了を待たずに、次の SQL ステートメントを実行できます。さらに、START AFTER COMMIT ステートメントは、本当に「すぐに」(トランザクションのコミット時に) 実行されるのではなく、サーバーがビジーであればむしろ遅れることがあるため、実際には、my_proc の実行が開始される前に、次の SQL ステートメントの実行が開始

されることもあれば開始されないこともあります。これが望ましい動作であることはまれです。ただし、どうしてもプログラムを続行している間にバックグラウンドで実行される非同期ストアード・プロシージャーを起動したい場合は、START AFTER COMMIT を自動コミット・モードで実行することは有効な方法です。

- 同じトランザクションで複数の据え置きプロシージャー呼び出しを実行した場合、すべての START AFTER COMMIT ステートメントの本体は非同期に実行されます。つまり、トランザクション内でこの START AFTER COMMIT ステートメントを実行した場合と同じ順序で実行されるとは限りません。
- START AFTER COMMIT の本体に含めることができる SQL ステートメントは 1 つだけです。ただし、この 1 つのステートメントをプロシージャー呼び出しにでき、プロシージャーには、他のプロシージャー呼び出しを含む複数の SQL ステートメントを含めることができます。
- START AFTER COMMIT ステートメントは、それが定義されたトランザクションにだけ適用されます。現行トランザクションで START AFTER COMMIT を実行する場合、据え置きプロシージャー呼び出しの本体は、現行トランザクションがコミットされたときにだけ実行されます。後続のトランザクションや、別の接続で行われたトランザクションでは実行されません。START AFTER COMMIT ステートメントは、「永続的な」動作を作成しません。複数のトランザクションの最後に同じ本体を呼び出すには、各トランザクションで、「START AFTER COMMIT ... CALL my_proc」ステートメントを実行する必要があります。
- 据え置きプロシージャー呼び出し (START AFTER COMMIT) ステートメントの本体の実行「結果」は、据え置きプロシージャー呼び出しを実行した接続に返されません。例えば、据え置きプロシージャー呼び出しの本体が、エラーが発生したかどうかを示す値を返す場合、その値は廃棄されます。
- ほとんどすべての SQL ステートメントを START AFTER COMMIT ステートメントの本体として使用できます。ストアード・プロシージャー呼び出しが標準ですが、UPDATE、CREATE TABLE など、ほとんどすべてを使用できます。(ただし、START AFTER COMMIT 内に START AFTER COMMIT ステートメントを配置することは推奨しません。) SELECT のようなステートメントは、実行しても結果が返されないため、通常は据え置きプロシージャー呼び出し内で使用する意味がありません。
- START AFTER COMMIT ステートメントがトランザクション内で実行された時点では、本体は実行されません。そのため、据え置きプロシージャー呼び出し自身または本体に、構文エラーまたは本体を実際に実行せずに検出できるその他のエラーが含まれている場合を除き、START AFTER COMMIT ステートメントはほとんど失敗しません。

据え置きプロシージャー呼び出しステートメントの実行が終了するまで、プログラムで次に実行される SQL ステートメントを実行したくない場合は、以下の回避策があります。

1. 据え置きプロシージャー呼び出しステートメントの最後 (据え置きプロシージャー呼び出しステートメントで呼び出されたストアード・プロシージャーの最後など) で、イベントを通知します。 (イベントについて詳しくは、 「*IBM solidDB プログラマー・ガイド*」を参照してください。)
2. 据え置きプロシージャー呼び出しを指定したトランザクションをコミットした直後に、イベントを待機するストアード・プロシージャーを呼び出します。

3. (イベントを待機する) ストアード・プロシージャー呼び出しの後に、プログラムで実行する次の SQL ステートメントを配置します。

例えば、以下のようなプログラムになります。

```
...
   START AFTER COMMIT ... CALL myproc;
   ...
   COMMIT WORK;
   CALL wait_for_sac_completion;
   UPDATE ...;
```

ストアード・プロシージャー wait_for_sac_completion は、myproc が通知するイベントを待機します。そのため、UPDATE ステートメントは、据え置きプロシージャー呼び出しステートメントが終了するまで実行されません。

この回避策は、やや危険であることに注意してください。据え置きプロシージャー呼び出しステートメントは、完了まで実行される保証がありません。そのため、ストアード・プロシージャー wait_for_sac_completion が、待機しているイベントを受け取らなくなる可能性があります。

完了まで実行したりしなかったりするコマンドを設計した理由は、何でしょうか。それは、START AFTER COMMIT 機能の主な目的が「プル同期通知」のサポートであるためです。プル同期通知機能を使用して、マスター・サーバーは、そのレプリカ (複数可) に対して、データが更新されたこと、およびレプリカが新しいデータを取得するリフレッシュを要求できることを通知できます。この通知プロセスが何らかの理由で失敗しても、データ破壊は発生しません。レプリカがデータをリフレッシュするまでの時間が長くなるだけです。レプリカは、常に、最後の正常なリフレッシュ操作以降のすべてのデータを取得するので、データの受信が遅れても、レプリカがデータを永久に失うことはありません。詳しくは、「*IBM solidDB 拡張レプリケーション・ユーザー・ガイド*」の『*プル同期通知の概要*』のセクションを参照してください。

注: START AFTER COMMIT の本体内のステートメントには、SELECT を含む、すべてのステートメントを使用できます。ただし、START AFTER COMMIT の本体は結果を返しません。そのため、SELECT ステートメントを START AFTER COMMIT 内で使用しても、通常は意味がないことを覚えておいてください。

注: 自動コミット・モードで START AFTER COMMIT... を実行した場合、指定されたステートメントはバックグラウンドですぐに開始されます。サーバーが実行できるときに非同期で実行されるため、ここで言う「すぐに」とは、「できるだけ早く」という意味になります。

プル同期通知 (「プッシュ同期」) の例

プル同期通知 (つまり、レプリカがリフレッシュを要求できる新しいデータが存在することを、マスターが関係するすべてのレプリカに通知する処理) を実装するには、前のトピックで説明したように START ステートメントと CALL ステートメントを使用します。ここで示す例では、トリガーも使用します。

マスター M1 およびレプリカ R1 と R2 を使用するシナリオを考えてみましょう。

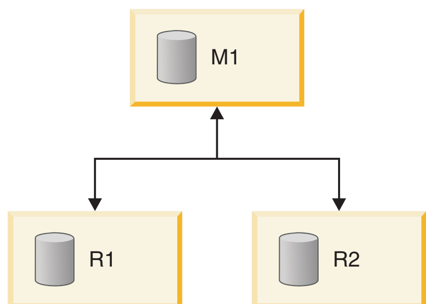


図 1. プル同期通知

プル同期通知を実行するには、以下の手順に従います。

1. マスター M1 でプロシージャー Pm1 を定義します。プロシージャー Pm1 に以下のステートメントを組み込みます。

   ```
   EXECDIRECT CALL Pr1 AT R1;
   EXECDIRECT CALL Pr1 AT R2;
   ```

   (対象となるレプリカごとに呼び出しを 1 回行います。レプリカ名は変わりますが、一般にプロシージャーの名前は各レプリカで同じであることに注意してください。)
2. レプリカ R1 でプロシージャー Pr1 を定義します。マスターが複数のレプリカで Pr1 を呼び出す場合は、対象となるすべてのレプリカに対して Pr1 を定義する必要があります。レプリカ・プロシージャーの例については、この後のセクションを参照してください。
3. 以下のような関連するすべての DML 操作に対してトリガーを定義します。
   - INSERT
   - UPDATE
   - DELETE
4. 各トリガー本体に、以下のステートメントを埋め込みます。

   ```
   EXECDIRECT START [UNIQUE] CALL Pm1;
   ```
5. 各レプリカで、適切なユーザーに EXECUTE 権限を付与します (事前にレプリカ上のユーザー Ur1 がマスター上の対応するユーザー Um1 にマップされている必要があります。ユーザー Um1 は、以下のステートメントを実行する必要があります。

   ```
   EXECDIRECT START [UNIQUE] CALL Pm1;
   ```

   Um1 がプロシージャーをリモートで呼び出した場合、その呼び出しがレプリカで実行されるときには Ur1 の特権が使用されます)。

スライスされたレプリカ

ある販売アプリケーションに CUSTOMER という表があり、この表に SALESMAN という列があります。マスター・データベースには、すべての営業担当者に関する情報が格納されています。各営業担当者には専用のレプリカ・データベースが用意され、そのレプリカにはマスターのデータの「スライス」のみが格納されます。つまり、各営業担当者のレプリカには、その営業担当者のデータ・スライスが格納されます。例えば、営業担当者 Smith のレプリカには、Smith のデータのみが格納されます。特定の顧客に割り当てられている営業担当者を変更する場合は、正しいレ

プリカに通知する必要があります。XYZ 社の営業担当者を Smith から Jones に再割り当てする場合は、XYZ 社に関連するデータを Jones のレプリカ・データベースに追加する一方で、Smith のレプリカ・データベースから削除する必要があります。両方のレプリカ・データベースを更新するコードを以下に示します。

```
-- 顧客に営業担当者を再割り当てする場合は、
-- 新旧両方の営業担当者に通知する必要があります。
-- 注: このサンプルには「UPDATE」トリガーのみを示していますが、
-- 実際には INSERT トリガーと DELETE トリガーも定義する必要があります。
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;
```

このアプリケーションで、ユーザーが販売地域「CA」の顧客すべてを営業担当者 Mike に割り当てるとします。

```
UPDATE CUSTOMER SET SALESMAN='Mike' WHERE SALES_AREA='CA';
COMMIT WORK;
```

マスター・サーバーには以下のプロシージャーがあります。

```
CREATE PROCEDURE CUST(salesman VARCHAR)
BEGIN
EXEC SQL EXECDIRECT CALL CUST(salesman) AT salesman;
COMMIT WORK;
END
```

各レプリカには以下のプロシージャーがあります。

```
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
```

プロシージャー CUST() では、営業担当者のレプリカをマスターのデータで強制的にリフレッシュします。このプロシージャー CUST() は、すべてのレプリカで定義されています。顧客の再割り当て先であるレプリカと再割り当て元であるレプリカの両方でこのプロシージャーを呼び出すと、プロシージャーによって両方のレプリカが更新されます。この操作では、実質的に、この顧客の担当を外れたレプリカから古いデータが削除され、この顧客を担当することになったレプリカにデータが挿入されます。パブリケーションとそのパラメーターが適切に定義されていれば、想定される各操作 (顧客を担当する営業担当者の変更など) を処理する詳細なロジックを追加で記述する必要はありません。各レプリカに最新のデータからリフレッシュするように通知するだけで済みます。

注:

プル同期通知はトリガーを使用しなくても実装できます。アプリケーションで適切なプロシージャーを呼び出すことで、プル同期を実装できます。 トリガーは、プル同期通知を START AFTER COMMIT ステートメントおよびリモート・プロシージャー・コールと連動させるための 1 つの手段です。

プル同期通知プロセスでは、レプリカが 1 往復分の余分なメッセージ交換を不必要に実行しなければならない場合があります。この状況は、マスターに変更を送信したレプリカがマスターの「ホット・データ」を変更することになる場合に、マスターが呼び出したプロシージャーがそのレプリカに対してメッセージを送信しようとすると発生する可能性があります。ただし、この状況は START AFTER COMMIT ステートメントを慎重に使用することで回避できます。マスターが更新されるたびにレプリカが即座に更新され、それによってマスターが即座に更新されるという「無限ループ」が生じないように注意する必要があります。これを回避するには、更新データを「即座に」マスターに送信するトリガー (それによってレプリカに再リフレッシュが「即座に」指示されます) をレプリカで作成する際に注意するのが最善の方法です。

バックグラウンド・ジョブの実行のトレース

START AFTER COMMIT ステートメントは、1 つの INTEGER 列を含んだ結果セットを返します。この整数はユニークな「ジョブ」ID です。この ID を使用して、なんらかの理由 (SQL ステートメントが無効、アクセス権限がない、レプリカを使用できないなど) で開始できなかったステートメントの状況を照会できます。

非コミット据え置きプロシージャー呼び出しステートメントの最大数に達すると、据え置きプロシージャー呼び出しが発行されるときにエラーが返されます。この最大数は、`solid.ini` で構成可能です。*IBM solidDB 管理者ガイド*を参照してください。

ステートメントを開始できない場合は、その理由がシステム表 SYS_BACKGROUNDJOB_INFO にログとして記録されます。

```
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)
);
```

この表には、失敗した START AFTER COMMIT ステートメントのみがログとして記録されます。ステートメント (プロシージャー呼び出しなど) が正常に開始された場合は、システム表に情報が格納されません。

ユーザーは、SQL SELECT 照会を使用するか、システム・プロシージャー SYS_GETBACKGROUNDJOB_INFO を呼び出すことで、表 SYS_BACKGROUNDJOB_INFO から情報をリトリーブできます。入力パラメーターはジョブ ID です。戻り値は、ID INTEGER、STMT WVARCHAR、USER_ID INTEGER、ERROR_CODE INTEGER、ERROR_TEXT INTEGER です。

また、ステートメントの開始に失敗すると、イベント SYS_EVENT_SACFAILED が通知されます。

```
CREATE EVENT SYS_EVENT_SACFAILED (ENAME WVARCHAR,
POSTSRVTIME TIMESTAMP,
UID INTEGER,
NUMDATAINFO INTEGER,
TEXTDATA WVARCHAR);
```

NUMDATAINFO フィールドにはジョブ ID が格納されています。アプリケーションはこのイベントを待機し、ジョブ ID を使用してシステム表 SYS_BACKGROUNDJOB_INFO から原因をリトリーブできます。

システム表 SYS_BACKGROUNDJOB_INFO を空にするには、ADMIN COMMAND cleanbgjobinfo を使用します。このコマンドを実行するには DBA 特権が必要です。つまり、表から行を削除できるのは DBA だけです。

バックグラウンド・タスクの制御

バックグラウンド・タスクは、SSC API および ADMIN COMMAND を使用して制御できます (SSC API について詳しくは、「*IBM solidDB 共有メモリー・アクセスおよびリンク・ライブラリー・アクセス・ユーザー・ガイド*」を参照してください)。サーバーは、START AFTER COMMIT で始まるステートメントを実行するタスクに、タスク・タイプ SSC_TASK_BACKGROUND を使用します。これらのタスクは複数存在する場合がありますが、個々に制御できない点に注意してください。

シーケンスの使用

シーケンス・オブジェクトは、効率的な方法でシーケンス番号を取得するために使用します。構文は以下のとおりです。

```
CREATE [DENSE] SEQUENCE sequence_name
```

シーケンスの作成方法に応じて、シーケンスにホールがある場合とない場合があります (シーケンスは疎にも密にもできます)。密シーケンスでは、シーケンス番号にホールがないことが保証されます。シーケンス番号の割り振りは、現行トランザクションにバインドされます。トランザクションがロールバックされると、シーケンス番号の割り振りもロールバックされます。密シーケンスの欠点は、現行のトランザクションが終了するまで、シーケンスが他のトランザクションからロックアウトされる点です。

密シーケンスの必要がない場合は、疎シーケンスを使用できます。疎シーケンスは、戻り値が一意であることを保証しますが、現行トランザクションにバインドされません。トランザクションが疎シーケンス番号を割り振り、後でロールバックした場合、シーケンス番号は単純に失われます。

シーケンス・オブジェクトは、例えば、主キー番号の生成などに使用できます。シーケンス表でなくシーケンス・オブジェクトを使用する利点は、シーケンス・オブジェクトは高速実行用に特に微調整でき、通常の更新ステートメントよりオーバーヘッドが少なくて済むことです。

密と疎のどちらのシーケンス番号も、1 から始まります。

CREATE SEQUENCE ステートメントでシーケンスを作成後、SQL ステートメント内で以下の構文を使用することにより、シーケンス・オブジェクトの値にアクセスできます。

- *sequencename*.CURRVAL は、シーケンスの現行値を返します。
- *sequencename*.NEXTVAL は、シーケンスを 1 だけインクリメントし、次の値を返します。

以下に、表のユニーク ID を自動的に作成する例を示します。

```
INSERT INTO ORDERS (id, ...) VALUES (order_seq.NEXTVAL, ...);
```

シーケンスは、ストアード・プロシージャーの内部でも使用できます。現行シーケンスの値は、以下のステートメントを使用してリトリーブすることができます。

```
EXEC SEQUENCE sequence_name.CURRENT INTO variable;
```

新しいシーケンスの値は、以下の構文を使用してリトリーブすることができます。

```
EXEC SEQUENCE sequence_name.NEXT INTO variable;
```

以下の構文を使用して、シーケンスの現行値を事前定義値に設定することもできます。

```
EXEC SEQUENCE sequence_name SET VALUE USING variable;
```

以下に、ストアード・プロシージャーを使用して新規シーケンス番号をリトリーブする例を示します。

```
"CREATE PROCEDURE get_my_seq
RETURNS (val INTEGER)
BEGIN
EXEC SEQUENCE my_sequence.NEXT INTO (val);
END";
```

イベントの使用

イベント・アラートは、solidDB データベース内の特殊なオブジェクトです。イベントは、主にタイミングの調整に使用されますが、少量の情報の送信にも使用できます。1 つの接続は、別の接続がイベントを「通知」するまで、そのイベントを「待ち」ます。

複数の接続が同じイベントを待つ場合もあります。複数の接続が同じイベントを待つ場合、イベントが通知されると、待っているすべての接続がその通知を受けます。1 つの接続が複数のイベントを待つこともでき、その場合、接続はそれらのいずれかのイベントが通知されたときに通知を受けます。

一般に、イベントがコンシュームするリソースの量は、ポーリングでコンシュームされる量よりずっと少量です。

ユーザーは独自のイベントを作成できます。サーバーにも、いくつかのシステム・イベントが組み込まれています。

サーバーはユーザー定義イベントを自動的には通知しません。それらはストアード・プロシージャーによって通知する必要があります。同様に、イベントはストアード・プロシージャー内で受信され (待たれ) ます。(ADMIN EVENT コマンドを使用して、ストアード・プロシージャーの外部でイベントを待つこともできます。)ア

プリケーションで、特定のイベントの発生を待つストアード・プロシージャーを呼び出す場合、そのイベントが通知されて受信されるまで、アプリケーションはブロックされます。マルチスレッド環境では、イベント待ちの間、別のスレッドおよび接続を使用してデータベースにアクセスできます。

イベントには、そのイベントを識別する名前と一連のパラメーターがあります。名前は、ユーザー指定の任意の英数字ストリングとすることができます。イベント・オブジェクトを作成するには、以下の SQL ステートメントを使用します。

```
CREATE EVENT event_name
   [(parameter_name datatype
      [parameter_name datatype...])]
```

パラメーター・リストは、パラメーター名とパラメーター・タイプを指定します。パラメーター・タイプは、通常の SQL タイプです。イベントをドロップするには、以下の SQL ステートメントを使用します。

```
DROP EVENT event_name
```

イベントは、常にストアード・プロシージャーの内部で通知されます。イベントは通常、ストアード・プロシージャーの内部で受信されます。イベントの通知と受信には、特殊なストアード・プロシージャー・ステートメントが使用されます。

イベントを通知するには、以下のストアード・プロシージャー・ステートメントを使用します。

```
post_statement ::= POST EVENT event_name [(parameters)]
```

イベント・パラメーターは、イベントがトリガーされたストアード・プロシージャー内のローカル変数、またはパラメーターでなければなりません。通知されたイベントを待っているすべてのクライアントは、そのイベントを受信します。

それぞれの接続は、独自のイベント・キューを備えています。イベント・キュー内に収集されるイベントを指定するには、以下のストアード・プロシージャー・ステートメントを使用します。

```
wait_register-statement ::=
REGISTER EVENT event_name
```

イベントをイベント・キューから除去するには、以下のストアード・プロシージャー・ステートメントを使用します。

```
UNREGISTER EVENT event_name
```

イベント・パラメーターは、イベントがトリガーされたストアード・プロシージャー内のローカル変数、またはパラメーターでなければなりません。

プロシージャーにイベントの発生を待たせるには、ストアード・プロシージャーの中で、以下のような WAIT EVENT 構文を使用します。

```
wait_event_statement::=
 WAIT EVENT
  [event_specification...]
 END WAIT
event_specification::=
 WHEN event_name [(parameters)] BEGIN
  statements
 END EVENT
```

ADMIN EVENT コマンドを使用して、イベントを待つこともできます。これは、例えば solsql コマンド行などで使用できます。以下は、ADMIN EVENT コマンドを使用してイベントの登録と待ちを行うために必要なコードの例です。

```
ADMIN EVENT 'register sys_event_hsbstateswitch';
ADMIN EVENT 'wait';
```

システム定義イベントを待つことも、ユーザー定義イベントを待つこともできます。ADMIN EVENT を使用してイベントを通知できないことに注意してください。ADMIN EVENT について詳しくは、186 ページの『ADMIN EVENT』を参照してください。

イベント例 1

このセクションには、イベントを使用する 2 つの例が含まれています。例 1 は、一緒に使用するとイベントの使用方法が分かる 1 対の SQL スクリプトです。例 2 は、一緒に使用すると、複数のイベントを待つことができる 1 対の SQL スクリプトで、ストアード・プロシージャーを含んでいます。

このイベントを使用する最初の例では、2 つのスクリプトがあります。一方のスクリプトはイベントを待ち、もう一方のスクリプトはそのイベントを通知します。イベントが通知された後、待っている側のイベントは待ちを終了し、次のコマンドに移ります。

このコード例を実行するには、WaitOnEvent.sql スクリプトを開始できるようにした後、WaitOnEvent.sql が待っている間に PostEvent.sql スクリプトを実行できるよう、2 つのコンソールが必要になります。

この特定の例では、待つストアード・プロシージャーはイベントが通知された後、実際には何もしません。スクリプトは単に待ちを終了し、呼び出し元に戻ります。その後、呼び出し元は任意の操作に進むことができ、この事例では、それは待っている間に挿入されたレコードをSELECT することです。

この例では、「record_was_inserted」という単一のイベントだけを待ちます。この章の後半では、単一の「WAIT」を使用して複数のイベントを待つ別のスクリプトを示します。

```
============================= 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;
```

SCRIPT 1 (Wait0nEvent.sql) でのトランザクションのコミットに関するガイドライン

可能な場合はいつでも、イベントを待つ前にすべての現行トランザクションを完了してください。トランザクションの内部で WAIT を実行すると、トランザクションはイベントが発生して次の COMMIT または ROLLBACK が実行されるまで、開かれたまま保持されます。これは、待ちの間、サーバーがロックを保持し、それによって Bonsai ツリーが過度に大きくなる可能性があることを意味します。Bonsai ツリーの詳細およびその増大の防止については、「*solidDB 管理者ガイド*」の『トランザクションのコミットによる Bonsai ツリーのサイズ縮小』のセクションを参照してください。

この例では、COMMIT WORK をプロシージャー内部の WAIT の直前に置いています。しかし、通常これはよい解決策ではありません。COMMIT または ROLLBACK を「待ち」プロシージャーの内部に置くことは、そのプロシージャーが別のトランザクションの一部として呼び出された場合、COMMIT または ROLLBACK はそれらが入っているトランザクションを打ち切り、新しいトランザクションを開始することを意味します。これは、多くの場合、ユーザーが望んでいることではありません。例えば、参照制約がある「子」表にデータを入力していた場合、参照されるデータが「親」表に入力されるのを待っているときに、トランザクションが 2 つのトランザクションに分裂すると、親への挿入がまだ済んでいないために、単純に「子」レコードの挿入が失敗します。

最良の方法は、トランザクションの内部で WAIT を行う必要がないようにプログラムを設計することです。代わりに、可能であれば、「待ち」プロシージャーをトランザクションとトランザクションの間で呼び出してください。イベント/待ちを使用することにより、作業の実行順序をある程度制御でき、これを利用して、実際にすべてを単一のトランザクション内に収めなくても、従属関係が満たされるようにすることができます。例えば、「非同期」の状況において、子と親の両方のレコードが挿入されるのを待つ場合、データベース・サーバーが「イベント」機能を備えていなければ、参照整合性を保証するためには、両方のレコードを同じトランザクションに挿入する必要があります。

イベント/待ちを使用することにより、親の挿入が必ず最初に実行されるようにできます。そうすれば、子が挿入されるときに常に親が存在することを保証できるので、2 番目のトランザクション内に子レコードの挿入を配置できます。(厳密に言えば、子が挿入されるときに、親が存在することをほとんどの場合、保証できます。挿入を 2 つの異なるトランザクションに分割した場合は、たとえ子の前に親が挿入されることを保証できても、プログラムが子レコードの挿入を試みる前に親が削除されるわずかな可能性が存在します。)

```
============================= 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;
```

イベント例 2

前の例では、単一のイベントを待つ方法を示しました。次の例では、複数のイベントを待ち、それらのイベントのいずれか 1 つが通知されたときに待ちを終了するストアード・プロシージャーの作成方法を示します。

```
============================= 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');
```

イベント例 3

この例は、REGISTER EVENT コマンドと UNREGISTER EVENT コマンドの非常に単純な使用法を示しています。お気付きかもしれませんが、前のスクリプトでは REGISTER EVENT を使用していませんでした。それでも、それらの WAIT コマンドは成功しました。その理由は、イベントを待つとき、まだそのイベントについて明示的に登録していない場合は、暗黙に登録されるためです。このため、明示的にイベントを登録する必要があるのは、それらのイベントのキューをその時点で開始したいが、それらのイベントの WAIT は後刻まで開始したくないという場合だけです。

```
CREATE EVENT e0;
CREATE EVENT e1 (param1 int);
COMMIT WORK;

-- イベントが発生したときに、そのイベントを登録するプロシージャーを作成します。
-- それらのイベントは、この接続のイベント・キューに置かれます。
"CREATE PROCEDURE eeregister
 BEGIN
 REGISTER event e0;
 REGISTER EVENT e1;
END";

CALL eeregister;
COMMIT WORK;

-- イベントを通知するプロシージャーを作成します。
"CREATE PROCEDURE eepost
 BEGIN
 DECLARE x int;
 x := 1;
 POST EVENT e0;
 POST EVENT e1(x);
 END";

COMMIT WORK;

-- イベントを通知します。まだイベントを待ってはいませんが、
-- イベントを登録してあるので、イベントはキューに保管されます。
CALL eepost;
COMMIT WORK;

-- この時点で、イベントを待つプロシージャーを作成します。
"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;

-- 既に 2 つのイベントについて登録してあり、既に
-- 2 つのイベントを通知してあるので、eewait プロシージャーを 2 回呼び出すと、
-- プロシージャーは待つことなく、即時に戻ります。
CALL eewait;
CALL eewait;
COMMIT WORK;


-- イベントについての登録を抹消します。
"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;

-- イベントが発生したときに、そのイベントを登録するプロシージャーを作成します。
-- それらのイベントは、この接続のイベント・キューに置かれます。
"CREATE PROCEDURE eeregister
 BEGIN
 REGISTER event e0;
 REGISTER EVENT e1;
END";

CALL eeregister;
COMMIT WORK;

-- イベントを通知するプロシージャーを作成します。
"CREATE PROCEDURE eepost
 BEGIN
 DECLARE x int;
 x := 1;
 POST EVENT e0;
 POST EVENT e1(x);
 END";

COMMIT WORK;

-- イベントを通知します。まだイベントを待ってはいませんが、
-- イベントを登録してあるので、イベントはキューに保管されます。
CALL eepost;
COMMIT WORK;


-- この時点で、イベントを待つプロシージャーを作成します。
"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;

-- 既に 2 つのイベントについて登録してあり、既に
-- 2 つのイベントを通知してあるので、eewait プロシージャーを 2 回呼び出すと、
-- プロシージャーは待つことなく、即時に戻ります。
CALL eewait;
CALL eewait;
COMMIT WORK;


-- イベントについての登録を抹消します。
"CREATE PROCEDURE eeunregister
 BEGIN
 UNREGISTER event e0;
```

```
 UNREGISTER EVENT e1;
END";

CALL eeunregister;
COMMIT WORK;
```

4 データベース管理のための solidDB SQL の使用

solidDB データベースと、そのユーザーおよびスキーマは、solidDB SQL ステートメントを使用して管理します。この章では、solidDB SQL を使用して行う管理タスクについて説明します。それらのタスクには、ロールと特権、表、索引、トランザクション、カタログ、およびスキーマの管理が含まれます。

solidDB SQL 構文の使用

SQL 構文は、ANSI X3H2-1989 (SQL-89) レベル 2 規格 (重要な SQL-92 および SQL-99 拡張を含む) に基づいています。構文の正式な定義について詳しくは 177 ページの『付録 B. solidDB SQL 構文』を参照してください。

SQL ステートメントは、solidDB SQL エディターを使用する場合に限り、セミコロン (;) で終了する必要があります。それ以外の場合、セミコロンで終わる SQL ステートメントは構文エラーになります。

SQL ステートメントを実行するには、solidDB SQL エディター (またはサード・パーティーの ODBC または JDBC 準拠ツール) を使用できます。タスクを自動化するために、SQL ステートメントをファイルに保存するのもよいでしょう。それらのファイルは、作成した SQL ステートメントを後で再実行するために使用するか、ユーザー、表、および索引のドキュメントとして使用できます。

solidDB SQL データ型

solidDB SQL は、SQL-92 規格の基本レベルの仕様、および重要な中間レベルの拡張をサポートしています。サポートされているデータ型について詳しくは、 169 ページの『付録 A. データ型』を参照してください。

長さ、位取り、および精度のパラメーターを任意に指定してデータ型を各自で定義することもできます。その場合、対応するデータ型のデフォルト・プロパティーは使用されません。

solidDB ADMIN COMMAND

solidDB SQL は、基本的な管理用タスク (バックアップ、パフォーマンス・モニター、シャットダウンなど) を実行するための拡張機能 ADMIN COMMAND ’*command* [*command_args*]’ を備えています。

ADMIN COMMAND のコマンド・オプションを実行するには、solidDB SQL エディター (テレタイプ) を使用します。使用可能な ADMIN COMMAND の簡略説明を参照するには、ADMIN COMMAND ’help’ を実行します。これらのステートメントの構文の正式な定義については、このガイドの 177 ページの『付録 B. solidDB SQL 構文』を参照してください。

注:

ADMIN COMMAND タスクは、solidDB リモート制御 (テレタイプ) で管理用コマンドとして使用することもできます。詳しくは、「*solidDB 管理者ガイド*」の『solidDB リモート制御 (テレタイプ)』という表題のセクションを参照してください。

solidDB には、データ同期機能を実装する SQL 拡張機能も用意されています。

関数の使用

solidDB 独自のスカラー関数はいずれも標準的な方法で使用できます。以下に例を示します。

```
select substring(line, 1,4) from test;
```

一方、名前が予約語と一致する関数は、エスケープ文字とともに使用する必要があります。以下に例を示します。

```
select "left"(line,4) from test;
```

または

```
select {fn left(line,4)} from test;
```

2 番目の例は ODBC 実装に依存しない構文に該当します。この構文はすべての API インターフェースおよび GUI インターフェースで使用できます。

ユーザー特権およびロールの管理

solidDB のテレタイプ・ツール、および多数の ODBC に準拠した SQL ツールを使用して、ユーザー特権を変更できます。ユーザーおよびロールの作成と削除には、SQL ステートメントまたはコマンドを使用します。いくつかの SQL ステートメントからなるファイルは、SQL スクリプトと呼ばれます。

`samples/sql` ディレクトリーに、ユーザーとロールの作成の例を示す SQL スクリプト `sample.sql` があります。このスクリプトは、solsql を使用して実行できます。独自のユーザーとロールを作成するために、ユーザー環境を記述した独自のスクリプトを作成できます。

ユーザー特権

solidDB データベースをマルチユーザー環境で使用する場合は、一部のユーザーに対して特定の表を隠蔽するために、ユーザー特権を提供できます。例えば、従業員の給与がリストされている表を従業員に見せたくない場合、または他のユーザーにテスト表を変更されたくない場合があります。

種類が異なる 5 つのユーザー特権を適用できます。ユーザーは、表またはビューに入っている情報の表示、削除、挿入、更新、または参照を行うことができます。これらの特権を任意に組み合わせて適用することもできます。表に対して、これらのどの特権も持っていないユーザーは、その表をまったく使用できません。

注: ユーザー特権は、付与された後、その特権を付与されたユーザーがデータベースにログオンした時点で有効になります。特権が付与されたとき、ユーザーが既にデータベースにログオンしていた場合は、そのユーザーが以下を行った場合に特権が有効になります。

- 特権が設定されている表またはオブジェクトに初めてアクセスしたとき、または、
- データベースへの接続をいったん切断し、再接続したとき

ユーザー・ロール

特権は、ロールと呼ばれるエンティティーに付与することもできます。ロールは、1 つの単位として複数のユーザーに付与できる特権のグループです。ロールを作成して、特定のロールにユーザーを割り当てることができます。単一のユーザーを複数のロールに割り当てることができ、単一のロールを複数のユーザーに割り当てることができます。

注:

1. 同じストリングをユーザー名とロール名の両方に使用することはできません。
2. ユーザー・ロールは、付与された後、そのロールを付与されたユーザーがデータベースにログオンした時点で有効になります。ロールが付与されたとき、ユーザーが既にデータベースにログオンしていた場合は、そのユーザーがデータベースへの接続をいったん切断して再接続した時点で、ロールが有効になります。

以下のユーザー名とユーザー・ロールは予約済みです。

表 15. 予約済みのユーザー名およびユーザー・ロール

予約名	説明
PUBLIC	このロールは、すべてのユーザーに特権を付与します。ある表に対するユーザー特権がロール *PUBLIC* に付与された場合、現在および将来のすべてのユーザーは、その表に対し、指定されたユーザー特権を持ちます。このロールは、すべてのユーザーに自動的に付与されます。
SYS_ADMIN_ROLE	これは、データベース管理者のデフォルトのロールです。このロールは、solidDB リモート制御だけでなく、すべての表、索引、およびユーザーに対する管理特権を持ちます。これは、データベース作成者ロールでもあります。
_SYSTEM	これは、すべてのシステム表およびビューのスキーマ名です。
SYS_CONSOLE_ROLE	このロールは solidDB リモート制御を使用する権限を持ちますが、その他の管理特権は持ちません。
SYS_SYNC_ADMIN_ROLE	これは、データ同期機能用の管理者ロールです。
SYS_SYNC_REGISTER_ROLE	このロールは、レプリカ・データベースをマスターに登録および登録抹消するためだけのものです。

SQL ステートメントの例

ユーザー、ロール、およびユーザー特権を管理する SQL ステートメントの例を以下に示します。

ユーザーの作成

```
CREATE USER username IDENTIFIED BY password;
```

このステートメントを実行する特権を持っているのは管理者だけです。以下の例は、CALVIN という名前でパスワードが HOBBES の新しいユーザーを作成します。

```
CREATE USER CALVIN IDENTIFIED BY HOBBES;
```

ユーザーの削除

```
DROP USER username;
```

このステートメントを実行する特権を持っているのは管理者だけです。以下の例では、CALVIN という名前のユーザーを削除します。

```
DROP USER CALVIN;
```

パスワードの変更

```
ALTER USER username IDENTIFIED BY new password;
```

ユーザー username および管理者は、このコマンドを実行する特権を持ちます。以下の例では、CALVIN さんのパスワードを GUBBES に変更します。

```
ALTER USER CALVIN IDENTIFIED BY GUBBES;
```

ロールの作成

```
CREATE ROLE rolename;
```

以下の例は、GUEST_USERS という新しいユーザー・ロールを作成します。

```
CREATE ROLE GUEST_USERS;
```

ロールの削除

```
DROP ROLE role_name;
```

以下の例では、GUEST_USERS という名前のユーザー・ロールを削除します。

```
DROP ROLE GUEST_USERS;
```

ユーザーまたはロールへの特権の付与

```
GRANT user_privilege ON table_name TO username or role_name ;
```

表に対するユーザー特権には、
SELECT、INSERT、DELETE、UPDATE、REFERENCES、および ALL があります。ALL では、前述の 5 つの特権すべてがユーザーまたはロールに付与されます。新しいユーザーには、付与されるまで特権がありません。

以下の例では、TEST_TABLE という名前の表に対する INSERT 特権と DELETE 特権を GUEST_USERS ロールに付与します。

```
GRANT INSERT, DELETE ON TEST_TABLE TO GUEST_USERS;
```

EXECUTE 特権は、ユーザーにストアード・プロシージャーを実行する権限を与えます。

```
GRANT EXECUTE ON procedure_name TO username or role_name ;
```

以下の例では、SP_TEST というストアード・プロシージャーに対する EXECUTE 特権を、ユーザー CALVIN に付与します。

```
GRANT EXECUTE ON SP_TEST TO CALVIN;
```

ユーザーにロールを与えることによるユーザーへの特権の付与

GRANT *role_name* TO *username* ;

以下の例では、GUEST_USERS ロールに定義されている特権をユーザー CALVIN に付与します。

```
GRANT GUEST_USERS TO CALVIN;
```

ユーザーまたはロールからの特権の取り消し

REVOKE *user_privilege* ON *table_name* FROM *username* または *role_name* ;

以下の例は、TEST_TABLE という表の INSERT 特権を GUEST_USERS ロールから取り消します。

```
REVOKE INSERT ON TEST_TABLE FROM GUEST_USERS;
```

ユーザーのロールの取り消しによる特権の取り消し

REVOKE *role_name* FROM *username* ;

以下の例は、GUEST_USERS ロールに定義されている特権を CALVIN から取り消します。

```
REVOKE GUEST_USERS FROM CALVIN;
```

ユーザーへの管理者特権の付与

GRANT SYS_ADMIN_ROLE TO *username* ;

以下の例では、CALVIN に管理者特権を付与します。これでこのユーザーはすべての表に対するすべての特権を持つことになります。

```
GRANT SYS_ADMIN_ROLE TO CALVIN;
```

ユーザーにデータ同期操作を実行する権限を付与することもできます。そのためには、以下のコマンドを実行します。

```
GRANT SYS_SYNC_ADMIN_ROLE TO HOBBES
```

注:

自動コミット・モードがオフに設定されている場合は、手動で作業をコミットする必要があります。作業をコミットするには、SQL ステートメント COMMIT WORK を使用します。自動コミット・モードがオンに設定されている場合は、トランザクションが自動的にコミットされます。

表の管理

solidDB には、オンラインで表の作成、削除、および変更ができる動的データ・ディクショナリーがあります。solidDB のデータベース表は、SQL コマンドを使用して管理されます。

solidDB ディレクトリーに、表の管理の例を示す `sample.sql` という名前の SQL スクリプトがあります。このスクリプトは、solsql を使用して実行できます。

以下に、表を管理するための SQL ステートメントの例をいくつか示します。solidDB SQL ステートメントの正式な定義については、 177 ページの『付録 B. solidDB SQL 構文』を参照してください。

データベース内のすべての表の名前を表示したい場合は、SQL ステートメント SELECT * FROM TABLES を発行します。(「TABLES」は、システム定義のビューです。) 表の名前は、列 TABLE_NAME に入っています。

システム表へのアクセス

solidDB システム表には、solidDB サーバー情報が、ユーザー情報も含めて格納されています。特定のシステム表にアクセスできるかどうかは、ユーザーのロールおよびアクセス権限に依存します。例えば DBA は、すべてのストアード・プロシージャーに関するすべての情報を、プロシージャー定義テキスト (つまり、CREATE PROCEDURE ステートメント) も含めて表示できます。通常のユーザーは、自分が作成したプロシージャーのプロシージャー定義テキストも含めて、ストアード・プロシージャーを表示できます。ストアード・プロシージャーに対する実行権限を持っていても、そのストアード・プロシージャーの作成者でない通常のユーザーは、そのストアード・プロシージャーに関する一部の情報を見ることができますが、プロシージャー定義テキストを表示することはできません。システム表のリストについては、 363 ページの『付録 D. データベース・システム表とシステム・ビュー』を参照してください。

下記の表は、特定のシステム表とそのデータに関する表示アクセス特権やオブジェクト付与特権をユーザー・ロールとユーザーのアクセス権限別に示したものです。

この表で、「アクセス権限を持つユーザー」とは、INSERT、UPDATE、DELETE、SELECT のいずれかのアクセス権限を持つ通常のユーザーを指していることに注意してください。*

表 16. 表の表示とアクセス権限の付与

タスク	DBA	所有者	アクセス権限を持つユーザー*	アクセス権限を持たないユーザー
SYS_TABLES の表示	すべて (制限なし)	すべて (制限なし)	すべて (制限なし)	すべて (制限なし)
SYS_TABLES 内のユーザー表の表示	すべて (制限なし)	所有者の表だけに制限されます。	ユーザーが INSERT、UPDATE、DELETE、SELECT、REFERENCES のいずれかのアクセス権限を持っているすべての表。	表を表示できません。
SYS_COLUMNS の表示	すべて (制限なし)	所有者の表内の列	ユーザーが INSERT、UPDATE、DELETE、SELECT、REFERENCES のいずれかのアクセス権限を持っている表内の列。	列を表示できません。

表 16. 表の表示とアクセス権限の付与 (続き)

タスク	DBA	所有者	アクセス権限を持つユーザー*	アクセス権限を持たないユーザー
SYS_PROCEDURES の表示 (プロシージャー定義テキスト、つまり CREATE PROCEDURE ステートメントのテキストは除く)	すべて (制限なし)	そのユーザー (所有者) が作成したプロシージャー。	そのユーザーが実行権限を持つプロシージャー。	プロシージャーを表示できません。
SYS_PROCEDURES 内のプロシージャー定義テキストの表示	すべて (制限なし)	そのユーザー (所有者) が作成したプロシージャー	実行権限があるユーザーでもプロシージャー定義テキストを表示できないことに注意してください。	プロシージャーまたはプロシージャー定義テキストを表示できません。
プロシージャーに対するアクセス権限を付与する能力	可	可	不可	不可
SYS_TRIGGERS の表示	すべて (制限なし)	そのユーザー (所有者) が作成したトリガー	なし	トリガーを表示できません。
SYS_TRIGGERS 内のトリガー定義テキストの表示	すべて (制限なし)	そのユーザー (所有者) が作成したトリガー	なし	トリガーを表示できません。

SQL ステートメントの例

表を管理する SQL ステートメントの例を以下に示します。

表の作成

```
CREATE TABLE table_name (column_name column_type
    [, column_name column_type]...);
```

すべてのユーザーが、表の作成権を持ちます。

以下の例は、TEST という名前の新しい表を作成します。この表には、列タイプが INTEGER の列 I と、列タイプが VARCHAR の列 TEXT が含まれます。

```
CREATE TABLE TEST (I INTEGER, TEXT VARCHAR);
```

表の削除

```
DROP TABLE table_name;
```

特定の表の作成者または SYS_ADMIN_ROLE を持つユーザーだけが、表の削除を行う特権を持ちます。

以下の例では、TEST という表を削除します。

```
DROP TABLE TEST;
```

注:

カタログおよびスキーマの場合。SQL の ANSI 規格で、キーワード RESTRICT および CASCADE が定義されています。カタログまたはスキーマをドロップするときにキーワード RESTRICT を使用した場合、他のデータベース・オブジェクト (表など) を含むカタログまたはスキーマはドロップできません。キーワード CASCADE を使用すると、データベース・オブジェクトを含むカタログまたはスキーマをドロップできます。含まれるデータベース・オブジェクトは、自動的にドロップされます。デフォルトの動作 (RESTRICT も CASCADE も指定されていない場合) は、RESTRICT です。

カタログおよびスキーマ以外のデータベース・オブジェクトの場合。solidDB SQL のほとんどの DROP ステートメントで、キーワード RESTRICT および CASCADE はその一部として受け入れられません。さらに、これらのデータベース・オブジェクトでは、単純な「純粋 CASCADE」または「純粋 RESTRICT」の動作よりもルールが複雑ですが、通常、オブジェクトは、ドロップ動作 RESTRICT でドロップされます。例えば、表 1 をドロップしようとしたとき、表 2 が表 1 と外部キーの従属関係を持っているか、表 1 を参照するパブリケーションがある場合、先に従属表またはパブリケーションをドロップしなければ、表 1 をドロップできません。ただし、サーバーは、すべての可能なタイプの従属関係に RESTRICT 動作を使用するわけではありません。例えば、ビューまたはストアード・プロシージャーが表を参照する場合、参照先の表はドロップできます。ビューまたはストアード・プロシージャーは、次にその表を参照しようとするときに失敗します。また、表に対応する同期履歴表がある場合、その同期履歴表は自動的にドロップされます。同期履歴表について詳しくは、「*solidDB 拡張レプリケーション・ユーザー・ガイド*」を参照してください。

表への列の追加

```
ALTER TABLE table_name ADD COLUMN column_name column_type;
```

特定の表の作成者、または SYS_ADMIN_ROLE を持つユーザーだけが、表内の列の追加または削除を行う特権を持ちます。

以下の例では、列タイプ CHAR(1) の列 C を表 TEST に追加します。

```
ALTER TABLE TEST ADD COLUMN C CHAR(1);
```

表からの列の削除

```
ALTER TABLE table_name DROP COLUMN column_name;
```

ユニーク制約または主キーの一部となっている列はドロップできません。主キーについて詳しくは、115 ページの『索引の管理』を参照してください。

以下に示すステートメントの例では、表 TEST から列 C を削除します。

```
ALTER TABLE TEST DROP COLUMN C;
```

注:

自動コミット・モードがオフに設定されている場合は、先に作業をコミットしてからでないと、変更した表のデータを変更できません。表の変更後に作業をコミットするには、以下の SQL ステートメントを使用します。

```
COMMIT WORK;
```

自動コミット・モードがオンに設定されている場合は、DDL (データ定義言語) ステートメントをはじめとするすべてのステートメントが自動的にコミットされます。

索引の管理

索引は、表へのアクセスを高速にするために使用されます。データベース・エンジンは索引を使用して、表の中の行に直接アクセスします。索引がない場合、エンジンは表の内容全体を検索して、求める行を見つける必要があります。索引は、1 つの表にいくつでも必要なだけ作成できます。ただし、索引を追加すると、その表に対する挿入、削除、更新など、書き込み操作の速度が低下します。パフォーマンスを向上させるための索引の作成について詳しくは、161 ページの『索引を使用した照会パフォーマンスの向上』を参照してください。

索引には、非ユニーク索引とユニーク索引の 2 種類があります。ユニーク索引は、すべてのキー値が固有である索引です。ユニーク索引は、索引の作成時に UNIQUE 制約が使用された場合は、常に作成されます。

以下の SQL ステートメントを使用して、索引の作成と削除ができます。これらのステートメントの構文の正式な定義については、131 ページの『並行性制御とロック方式』を参照してください。

SQL ステートメントの例

索引を管理する SQL コマンドの例を以下に示します。

表の索引の作成

```
CREATE [UNIQUE] INDEX index_name ON base_table_name
 column_identifier [ASC | DESC]
 [, column_identifier [ASC | DESC]] ...
```

特定の表の作成者または SYS_ADMIN_ROLE を持つユーザーだけが、索引の作成またはドロップを行う特権を持ちます。

以下の例は、表 TEST の列 I に X_TEST という索引を作成します。

```
CREATE INDEX X_TEST ON TEST (I);
```

表のユニーク索引の作成

```
CREATE UNIQUE INDEX index_name ON table_name (column_name);
```

以下の例は、表 TEST の列 I に UX_TEST というユニーク索引を作成します。

```
CREATE UNIQUE INDEX UX_TEST ON TEST (I);
```

索引の削除

```
DROP INDEX index_name;
```

以下の例では、X_TEST という名前の索引を削除します。

```
DROP INDEX X_TEST;
```

注:

索引を作成またはドロップした場合は、その索引の表のデータを変更する前に、作業をコミットまたはロールバックする必要があります。

主キー索引

表から特定のレコードを 1 つだけリトリーブするには、そのレコードを一意的に識別できなければなりません。solidDB は「主キー」を使用して、個々の表の個々のレコードを一意的に識別します。主キーは 1 つの列または複数の列の組み合わせであり、固有値または値の組み合わせを格納しています。それぞれの表ごとに、明示的または暗黙的な主キーが存在します。

solidDB は「主キー索引」を、その主キーのフィールド (単数または複数) に基づいて自動的に作成します。主キー索引は、すべての索引と同様に、表内にあるデータへのアクセスを高速にします。しかし、他の索引とは異なり、主キー索引はレコードをデータベースに保管する順序も制御します。(これを「クラスタリング」と呼びます。) 各レコードは、主キーの値に基づいて昇順で保管されます。

表の作成者が主キーを指定しなかった場合は、solidDB が自動的に表の主キーを作成します。その主キー内の一意性を確保するために、サーバーは非表示の内部行 ID を使用します。その行 ID の値は、「ROWID」というシンボリック疑似列名を使用してリトリーブし、照会の中で使用できます。

注:

solidDB では、表を作成した後に明示的な主キーを追加することはできません。ユーザーが主キーを指定しなかった場合は、その表に最も効率的な照会方式を (ROWID を使用しなければ) 使用できません。また、そのような表を参照整合性制約の中で参照表として使用することもできません。これらの理由から、表の作成時に必ず主キーを定義することを強く推奨します。

主キーが (表の作成者またはサーバーによって) 定義された後、サーバーは重複する主キー値を持つ行が表に挿入されるのを防止します。

副次キー索引

索引は検索速度を向上させるため、表に、検索で頻繁に使用される属性ごと (または属性の組み合わせごと) に 1 つの索引を作成すると有用です。1 次索引以外のすべての索引を「副次索引」と呼びます。

すべての索引が、列、列の順序、値の順序 (昇順、降順) の固有の組み合わせであれば、表に作成できる索引の数に上限はありません。例えば、以下のコードで、3 番目の索引は最初の索引の重複になるため、エラー・メッセージが生成されるか、重複した情報によってディスク・スペースが浪費されます。

```
CREATE INDEX i1 ON TABLE t1 (col1, col2);
-- 以下は、索引 i1 と列は同じですが、列の順序が異なるため、
-- 適切です。
CREATE INDEX i2 ON TABLE t1 (col2, col1);
-- 索引 i3 は、索引 i1 と完全に同一なので、
-- 正しくありません。
CREATE INDEX i3 ON TABLE t1 (col1, col2);   -- エラー。
-- 以下は、列と列の順序は同じですが、
-- 索引値の順序 (昇順と降順) が
-- 異なるため、適切です。
CREATE INDEX i3b ON TABLE t1 (col1, col2) DESC;
```

索引が、別の索引の「先導サブセット」(索引 2 の N 列すべての列、列の順序、値の順序が、索引 1 の先頭 N 列と完全に同じ) である場合は、スーパーセットである索引だけを作成する必要があります。例えば、DEPARTMENT + OFFICE + EMP_NAME の組み合わせで索引を作成したとします。この索引は、department、office、emp_name を一緒に検索するときだけではなく、department だけ、または department と office だけを一緒に検索するときにも使用できます。そのため、department だけ、または department と office だけの索引を別に作成する必要はありません。同じことが ORDER BY 演算子にも言えます。ORDER BY 基準が既存の索引のサブセットと一致する場合、サーバーはその索引を使用できます。

主キーまたはユニーク制約を定義した場合、そのキーまたは制約は、索引として実装されることに留意してください。そのため、主キーまたは既存のユニーク制約の「先導サブセット」になる索引を作成する必要はありません。このような索引は冗長です。

副次索引を使用して検索するとき、サーバーが要求されたすべてのデータを索引キーで検索した場合は、表の行全体をルックアップする必要はありません。(これは、「読み取り」操作、すなわち SELECT ステートメントにのみ適用されます。ユーザーが表の値を更新する場合は、表のデータ行も索引の値と同様に更新する必要があります。)

重複索引に対する保護

solidDB には、重複索引に対する保護が含まれています。元の索引が重複索引になるような他の索引が作成された場合、索引の再作成 (DROP/CREATE) は失敗することがあります。重複索引について理解するには、以下の例を参照してください。

A、B、C、D、E という 5 つの列を持つ表を作成したとします。以下の索引が表に作成されています。

- A
- AB
- BCE
- ABC

索引 B は、列 B の検索またはフィルタリングに使用します。索引 BCE は、列 B で始まります。したがって、列 B を見つけるために索引を使用する照会は、索引 BCE を使用できます。索引 AB と索引 ABC の場合も同様です。よって、索引 B と索引 AB は重複索引です。

重複索引は、以下のような悪影響を及ぼします。

- 必要なストレージ・スペースが増える
- 更新のパフォーマンスが低下する
- バックアップ時間が増える

重複索引を作成しようとすると、索引作成が失敗し、solidDB は以下のエラーを発行します。

```
SOLID Table Error 13199: Duplicate index definition
```

詳しくは、「*IBM solidDB 管理者ガイド*」の付録『エラー・コード』を参照してください。

参照整合性

参照整合性は、データベース表の間の関係を整合性のある状態のままに保つための概念です。つまり、データへの参照は有効である必要があります。

2 つのデータベース表 (参照先の表と参照元の表と言います) の間の関係は、外部キーを使用して作成されます。外部キーは、参照先の表の主キー列 (または、類似したその他のユニーク列) に一致する参照元の表のフィールドです。つまり、外部キーを使用して、「部署に所属する従業員」など 1 対 n の概念的な関係を表すことができます。参照元の表に参照先の表への外部キーがある場合は、参照整合性の概念によって、参照先の表に対応するレコードがなければ参照元の表 (外部キーを含む表) にレコードを追加できません。

上で説明したように、外部キーを使用すると、参照整合性が強制されます。外部キーは、参照制約定義で保守されます。制約は、制約違反が発生したときに solidDB が実行する参照アクションも指定します。これは、例えば、参照される主キーを持つ行が参照先の表から削除されたときに発生します。外部キーと制約については、以下の章で詳しく説明します。

主キーと候補キー

表を参照制約に参照表として参加させるためには、主キー (こちらが望ましい) または候補キーを定義する必要があります。主キーは、CREATE TABLE ステートメント内の主キー制約構文で定義します。以下に例を示します。

```
CREATE TABLE customers (
 cust_id INTEGER PRIMARY KEY,
 name CHAR(24),
 city CHAR(40));
```

もう 1 つの可能な方法は、列または列のグループにユニーク索引を定義し、それらの列に NOT NULL 制約を設定することです。これは事実上、「候補キー」を生成します。結合を派生させている間のパフォーマンスのゲインのために、明示的な主キーを使用する方が望ましい方法です。

外部キー

外部キーは、表内で別の表の固有値を参照する (あるいは「関係付ける」) 列 (または列のグループ) です。外部キー列の各値と一致する値が、別の表に存在している必要があります。

参照元の表の各レコードが参照先の表のレコードを 1 つだけ参照するようにするには、参照先の表の参照先の列に主キー制約、またはユニーク制約と非 NULL 制約の両方を設定する必要があります (ユニーク索引だけでは不十分なので注意してください)。

例えば、ある銀行に顧客情報を格納する表と口座情報を格納する表があるとします。各口座は顧客に関連付ける必要があり、ユニークな customer_id を保持することになります。この customer_id は、顧客表の主キーとなります。また各口座には、その口座を所有する顧客の customer_id のコピーも必要です。これにより、口

座情報を基に顧客情報を参照できるようになります。口座表にある customer_id のコピーは外部キーです。このキーは、顧客表の主キーで一致する値を参照します。

以下に例を示します。この例では、Customers 表の CUST_ID 列が参照先の表の主キー、Accounts 表の CUST_ID 列が Customers 表を参照する外部キーとなっています。この図からわかるように、各口座は対応する顧客に関連付けられています。複数の口座を所有する顧客もいます。

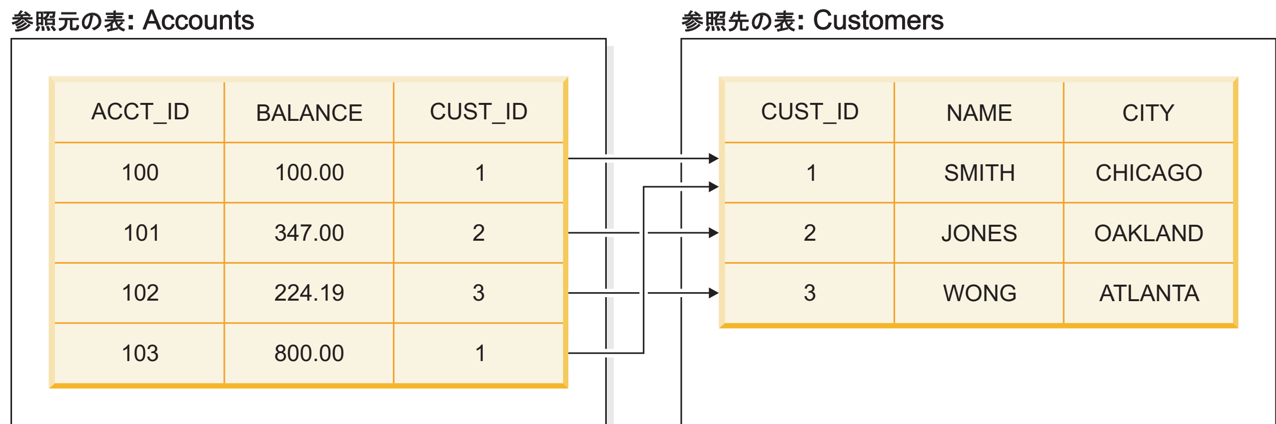


図 2. 参照制約

参照先の表「accounts」は、以下のステートメントで作成されます。

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

REFERENCES 節では参照先の表のみが指定され、参照先の列は指定されていません。デフォルトでは、主キーが参照先と想定されます。この方法は、参照先の列を指定した場合に発生する可能性があるエラーを回避できる望ましい方法です。

上の例では、主キーと外部キーが 1 つの列を使用しています。ただし、主キーと外部キーが複数の列で構成される場合もあります。各外部キー値は対応する主キー値と正確に一致している必要があるので、外部キーを構成する列の数およびデータ型は主キーと同じでなければならず、キー列の順序も同じであることが必要です。外部キーの列名が主キーと異なっていてもかまいませんが、そのようなことはまれです。(外部キーおよび主キーのデフォルト値も異なっていてかまいません。ただし、参照先の表の値が固有でなければならないため、デフォルト値はあまり使用されず、主キーの一部である列に使用されることはほとんどありません。外部キー列にもデフォルト値はあまり使用されません。)

主キーの値は固有でなければなりませんが、外部キーの値は固有である必要はありません。例えば、1 つの銀行で 1 人の顧客が複数の口座を所有している場合があります。顧客表の主キー列に含まれる顧客 ID は固有であることが必要ですが、口座表の外部キー列には同じ CUST_ID が複数回出現する可能性があります。上の図では、顧客 SMITH が複数の口座を所有しているため、その CUST_ID が Accounts 表の外部キー列に複数回出現しています。

まれに、ある表の外部キーが同じ表の主キーを参照する場合があります。つまり、参照先の表と参照元の表が同じ表である場合です。例えば、従業員の表で、各従業員レコードにその従業員の管理者の ID を格納するフィールドがあるとします。管理者も同じ表に格納されています。したがって、この表の管理者 ID は、同じ表の従業員 ID を参照する外部キーとなります。この例を以下の図で示します。

自己参照表

EMP_ID	MGR_ID	EMP_NAME
1	NULL	ANNAN
10	1	WONG
20	1	SMITH
147	10	JONES
162	20	RAMA

図 3. 自己参照制約

この例では、Rama の管理者が Smith (Rama の MGR_ID は 20、Smith の EMP_ID は 20) です。Smith は Annan の直属です (Smith の MGR_ID は 1、Annan の EMP_ID は 1)。 Jones の管理者は Wong、Wong の管理者は Annan です。Annan が社長である場合、Annan には管理者がいないことになり、外部キーの値は NULL になります。

主キーが複数の列で構成される場合は、列を定義した後で主キーを定義する必要があります。以下に例を示します。

```
CREATE TABLE DEPT (
    DIVNO INTEGER,
    DEPTNO INTEGER,
    DNAME VARCHAR,
    PRIMARY KEY (DIVNO, DEPTNO));
```

外部キーにも同様の構文を使用できます。ただし、外部キーは、制約名も使用する CONSTRAINT 構文で定義することを強く推奨します。この方法により、表を作成した後に ALTER TABLE ステートメントで外部キーを動的に削除できるようになります。以下に例を示します。

```
CREATE TABLE EMP (
    EMPNO INTEGER PRIMARY KEY,
    DIVNO INTEGER,
    DEPTNO INTEGER,
    ENAME VARCHAR,
    CONSTRAINT emp_fk1 FOREIGN KEY (DIVNO, DEPTNO) REFERENCES DEPT);
```

注:

他の整合性制約と同様に、参照整合性制約 (外部キー) の指定およびその操作 (ドロップまたは追加) を ALTER TABLE ステートメントで動的に実行できます。詳しくは、『 122 ページの『制約の動的な管理』』を参照してください。

CREATE TABLE および ALTER TABLE の構文について詳しくは、『 177 ページの『付録 B. solidDB SQL 構文』』を参照してください。

すべての表に外部キーを作成できるわけではありません。表がマスター/レプリカの同期に関与し、かつレプリカ・サーバーにある場合、その表には外部キー制約を設定できません。この制限は、レプリカにあり、かつパブリッシュ/サブスクライブ (リフレッシュ) アクティビティーに関与している表にのみ適用されます。レプリカにある表でも、リフレッシュ・アクティビティーに関与していなければ、外部キーを設定できます。マスター・データベース内の表には、パブリッシュ/リフレッシュ・アクティビティーに関与している場合でも外部キーを設定できます。

この制限は主キーには適用されません。どの表にも主キーを設定できます (同期表のように主キーを必要とする表もあります)。

外部キーを定義すると、必ず外部キー列に索引が作成されます。参照されるレコードが更新または削除されるたびに、参照がない状態で残される参照元レコードがないことがサーバーによって検査されます。各外部キーに索引があることで、外部キー検査のパフォーマンスが向上します。

参照アクション

参照整合性は、参照制約違反が発生した場合に特定のアクションを実行することによって、システムで保守されます。参照制約違反には、例えば、以下の場合があります。

- 無効な外部キー値を含む行が参照元の表に挿入された
- 参照元の表の外部キーが無効な値に更新された
- 参照される主キーを持つ行が参照先の表から削除された
- 参照される主キーが参照先の表で更新された

制約違反が発生した場合、以下のアクションが可能です。

- *No action*。このオプションは、参照整合性制約に違反する操作を制限またはロールバックします。
- *Cascade*。参照先の表での実行操作の場合、参照先の表でのその操作を参照元の表にカスケードします。すべての参照元行の削除 (カスケード削除) や、すべての参照元外部キー値の更新 (カスケード更新) などがあります。
- *Set default*。参照先の表での実行操作の場合、参照元の列を事前設定済みのデフォルト値に設定します。
- *Set null*。参照先の表での実行操作の場合、参照元の列を NULL に設定します。
- *Restrict*。参照整合性アクションは、一時的に参照制約に違反した表の変更を許可することがあります。No action が、このような違反を許可します。表の状態を、一時的にでも決して制約に違反しないようにする必要がある場合は、Restrict 参照アクションを使用します。

アクションが指定されていない場合、デフォルトの「No action」であると見なされます。

参照アクションのカスケードでは、循環は許可されません。カスケード・アクションを含む外部キーで構成された循環を作成しようとすると、エラーが発生します。

注: 任意の 2 つの表の間で定義できる CASCADE UPDATE パスは、最大 1 つです。この制限は、CASCADE DELETE には適用されません。

制約の動的な管理

制約は、ALTER TABLE 節で動的に管理できます。使用できるサブ節は以下のとおりです。

- ADD CONSTRAINT。この節は名前付きの制約を表に追加します。
- DROP CONSTRAINT。この節は名前付きの制約を表から削除します。

注:

solidDB では、キーワード CONSTRAINT を使用する際に制約名を指定する必要があります。

- CHECK。この制約では、表または表の列に対するルールを指定できます。それぞれのルールは条件であり、そのルールが定義されている表のすべての行について false でないことが必要です。そうでない場合は表を更新できません。

  このルールはブール式です。例えば、このルールで値の範囲や公正さを検査したり、単純な比較を行ったりできます。1 つのステートメントで複数の検査を実行できます。使用可能な式と演算子は以下のとおりです。

表 17. 式および演算子

式	説明
<	より小さい
>	より大きい
=	等しい
<=	より小か等しい
>=	より大か等しい
<>	等しくない
AND	論理積
ANY	後続のリスト内または指定された表内
BETWEEN	範囲内
IN	後続のリスト内または指定された表内
MAX	最大値

表 17. 式および演算子 (続き)

式	説明
MIN	最小値
NOT	否定
OR	論理和
XOR	排他的論理和

- UNIQUE。UNIQUE 制約は、所定の列または列のリストに同じ値を含んでいる行が表内に複数存在しないことを必要とします。ユニーク制約は、表レベルまたは列レベルで作成できます。主キーにはユニーク制約が設定されることに注意してください。
- FOREIGN KEY。FOREIGN KEY 制約は、外部キー列の各値に一致する値が参照先の表に存在することを必要とします。

注:

solidDB では、名前のない制約に対して自動的に名前が生成されます。名前を表示する場合は、コマンド soldd -x hiddennames を使用します。

制約の構文情報と例については、 177 ページの『付録 B. solidDB SQL 構文』の CREATE TABLE および ALTER TABLE のセクションを参照してください。

データベース・オブジェクトの管理

概要

solidDB では、カタログとスキーマを使用してデータを編成することができます。(カタログには他用途もありますが、それについては後述します。) solidDB でのスキーマの用法は SQL 標準に準拠しています。一方、solidDB でのカタログの用法は SQL 標準に対する拡張機能です。

カタログとスキーマを使用することで、データベース・オブジェクト (表やシーケンスなど) を階層的にグループ化できます。これにより、関連する項目を同じグループに入れることができます。例えば、会計システムに関連するすべての表を 1 つのグループ (カタログなど) にまとめ、人事システムに関連するすべての表を別のグループにまとめることができます。また、データベース・オブジェクトをユーザー別にグループ化することもできます。例えば、Jane Smith が使用するすべての表を 1 つのスキーマにまとめることができます。

カタログは階層の最も高い (最も広い) レベルです。スキーマ名は中間レベルです。表などの特定のデータベース・オブジェクトは、階層の最も低い (最も狭い) レベルです。したがって、1 つのカタログが複数のスキーマで構成され、各スキーマが複数の表で構成されている可能性があります。

オブジェクト名はグループ内で固有でなければなりませんが、グループ間で固有である必要はありません。したがって、例えば Jane Smith のスキーマと Robin

Trower のスキーマに「bills」という同じ名前の表が含まれていてもかまいません。この 2 つの表の間には関連がありません。表の名前は同じでも、構造やデータがそれぞれに異なっている可能性があります。同様に、カタログ「accounting_catalog」および「human_resources_catalog」にはそれぞれ「david_jones」という名前のスキーマを含めることが可能です。この 2 つのスキーマは名前は同じですが、互いに関連しているわけではありません。

当然のことながら、特定の表を指定する場合に、その表名がデータベース内で固有でないときは、以下のようにカタログ、スキーマ、および表の名前を指定することでその表を識別できます。

```
accounting_catalog.david_jones.bills
```

構文の詳細については後述します。

完全な名前を指定しない場合 (つまりスキーマまたはスキーマとカタログを省略した場合) は、サーバーで現在/デフォルトのカタログ名およびスキーマ名を使用して使用する表が決定されます。

一般に、カタログは論理的なデータベースと考えることができます。スキーマは通常は 1 人のユーザーに対応します。これについては、以下で詳しく説明します。

カタログ

物理データベース・ファイルには、複数の論理データベースが含まれている場合があります。それぞれの論理データベースは、表、索引、プロシージャー、トリガーなど、独立した完全なデータベース・オブジェクト・グループです。それぞれの論理データベースは、1 つのカタログです。solidDB カタログは、索引 (項目の完全な内容を含まずに項目を見つけることができる、従来のライブラリー・カード・カタログの意味) だけに限られたものでないことに注意してください。

カタログを使用すると、データベースを論理的に区分することができるため、以下のことが可能になります。

- データをビジネス、ユーザー、およびアプリケーションの必要に合わせて編成する。
- 複数のマスターまたはレプリカ・データベースを、同期化のために 1 つの物理データベース・サーバー内に (論理データベースを使用して) 指定する。マルチマスター環境での同期の実装について詳しくは、 「*IBM solidDB 拡張レプリケーション・ユーザー・ガイド*」の『マルチマスター同期モデル』をお読みください。

スキーマ

カタログには、1 つ以上のスキーマを含めることができます。スキーマは、データベースの一部またはすべての定義を提供する永続的なデータベース・オブジェクトです。特定のスキーマ名に関連付けられているデータベース・オブジェクトの集合を表します。これらのオブジェクトには、表、ビュー、索引、ストアード・プロシージャー、トリガー、およびシーケンスが含まれます。スキーマを使用して、同じ論理データベース (すなわち、単一カタログ) で、ユーザーごとに独自のデータベース・オブジェクト (表など) を提供できます。データベース・オブジェクトでスキーマが指定されていない場合、デフォルト・スキーマは、オブジェクトを作成したユーザーのユーザー ID です。

カタログおよびスキーマ内でオブジェクトを一意的に識別する

スキーマは、2 人のユーザーが同じ物理データベースあるいは同じ論理データベースに同じ名前の表を作成することを可能にします。例えば、1 つの物理データベースに employee_catalog と inventory_catalog という 2 つのカタログがあるとします。それぞれのカタログに smith と jones という 2 つのスキーマが含まれており、1 人の Smith が両方の「smith」スキーマを所有し、1 人の Jones が両方の「jones」スキーマを所有しています。Smith と Jones が自身の各スキーマに books という表を作成すると、「books」という名前の表が合計で 4 つ作成されます。これらの表は以下の名前でアクセスされます。

```
employee_catalog.smith.books
employee_catalog.jones.books
inventory_catalog.smith.books
inventory_catalog.jones.books
```

このように、カタログ名とスキーマ名を使用して、表などのデータベース・オブジェクトの名前を「修飾」(一意的に識別) することができます。オブジェクト名は、すべての DML ステートメントで以下の構文を使用して修飾できます。

catalog_name.schema_name.database_object

または

catalog_name.user_id.database_object

例:

```
SELECT cust_name FROM accounting_dept.smith.overdue_bills;
```

カタログ名を指定するかどうかにかかわらず、1 つ以上のデータベース・オブジェクトをスキーマ名で修飾できます。構文は以下のとおりです。

schema_name.database_object_name

または

user_id.database_object_name

例:

```
SELECT SUM(sales_tax) FROM jones.invoices;
```

データベース・オブジェクトでスキーマ名を使用するには、スキーマを事前に作成しておく必要があります。

デフォルトでは、スキーマ名なしで作成されたデータベース・オブジェクトはそのオブジェクトの作成者のユーザー ID で修飾されます。以下に例を示します。

user_id.table_name

カタログ・コンテキストとスキーマ・コンテキストは、SET CATALOG ステートメントまたは SET SCHEMA ステートメントを使用して設定されます。

SET CATALOG でカタログ・コンテキストが設定されていない場合は、すべてのデータベース・オブジェクト名がデフォルトのカタログ名を使用して解決されます。

注: 新しいデータベースを作成するとき、または古いデータベースを新しいフォーマットに変換するときに、ユーザーはデータベース・システム・カタログのデフォ

ルトのカタログ名を指定するように要求されます。ユーザーはこの指定されたデフォルトのカタログ名を知らなくてもデフォルトのカタログ名にアクセスできます。例えば、以下の構文を指定することでシステム・カタログにアクセスできます。

```
""._SYSTEM.table
```

カタログ名として指定した空ストリング ("") が、solidDB によってデフォルトのカタログ名に変換されます。また、ユーザーがカタログ名を指定しない場合でも、solidDBによって _SYSTEM スキーマがシステム・カタログに自動的に解決されます。

カタログとスキーマを作成する SQL ステートメントの例を次で示します。solidDB SQL ステートメントの正式な定義については、 177 ページの『付録 B. solidDB SQL 構文』を参照してください。

SQL ステートメントの例

データベース・オブジェクトを管理する SQL ステートメントの例を以下に示します。

カタログの作成

```
CREATE CATALOG catalog_name
```

データベースの作成者または SYS_ADMIN_ROLE を持つユーザーだけが、カタログの作成またはドロップを行う特権を持ちます。

以下の例では、C というカタログを作成し、ユーザー ID は SMITH であるとします。

```
CREATE CATALOG C;
SET CATALOG C;
CREATE TABLE T (i INTEGER);
SELECT * FROM T;
-- 名前 T は C.SMITH.T に解決されます。
```

カタログ・コンテキストおよびスキーマ・コンテキストの設定

以下の例では、カタログ・コンテキストを C に設定し、スキーマ・コンテキストを S に設定します。

```
SET CATALOG C;
SET SCHEMA S;
CREATE TABLE T (i INTEGER);
SELECT * FROM T;
-- 名前 T は C.S.T に解決されます。
```

カタログの削除

```
DROP CATALOG catalog_name
```

以下の例では、C という名前のカタログを削除します。

```
DROP CATALOG C;
```

スキーマの作成

```
CREATE SCHEMA schema_name
```

すべてのデータベース・ユーザーがスキーマを作成できます。ただし、ユーザーには、そのスキーマに関するオブジェクトを作成する権限が必要です (CREATE PROCEDURE、CREATE TABLE など)。

スキーマを作成しても、その新しいスキーマが暗黙的に現行スキーマまたはデフォルト・スキーマになるわけではないことに注意してください。新しいスキーマを現行スキーマにするには、そのスキーマを SET SCHEMA ステートメントで明示的に設定する必要があります。

以下の例では、FINANCE というスキーマを作成し、ユーザー ID は SMITH であるとします。

```
CREATE SCHEMA FINANCE;
CREATE TABLE EMPLOYEE (EMP_ID INTEGER);
-- 注: employee 表は、FINANCE.EMPLOYEE ではなく、SMITH.EMPLOYEE と
-- 修飾されます。スキーマを作成しても、その新しいスキーマが暗黙的に
-- 現行スキーマまたはデフォルト・スキーマになるというわけではありません。
SET SCHEMA FINANCE;
CREATE TABLE EMPLOYEE (ID INTEGER);
SELECT ID FROM EMPLOYEE;
-- この場合、表は FINANCE.EMPLOYEE に修飾されます。
```

スキーマの削除

DROP SCHEMA *schema_name*

以下の例では、FINANCE という名前のスキーマを削除します。

```
DROP SCHEMA FINANCE;
```

5 トランザクションの管理

この章では、トランザクション管理に絞って説明します。具体的には、トランザクションの管理方法、並行性制御とロック方式、および持続性レベルの選択方法を説明します。

トランザクションの管理

トランザクションは、単一の作業単位として扱われる SQL ステートメントのグループです。したがって、すべてのステートメントが 1 つのグループとして実行されるか、どれも実行されないかのどちらかになります。このセクションでは、読者が標準 SQL ステートメントを使用してトランザクションを作成する基本的な事項について、知識があることを前提としています。ここでは、solidDB SQL を使用してトランザクションの動作、並行性制御、および分離レベルを処理する方法を説明します。

読み取り専用トランザクションおよび読み取り/書き込みトランザクションの定義

トランザクションを読み取り専用または読み取り/書き込みに定義するには、以下の SQL コマンドを使用します。

```
SET TRANSACTION { READ ONLY | READ WRITE }
```

このコマンドでは、以下のオプションが使用可能です。

- READ ONLY

  このオプションは、読み取り専用トランザクションに使用します。

- READ WRITE

  このオプションは、読み取り/書き込みトランザクションに使用します。このオプションはデフォルトです。

注: トランザクション間の競合を検出するには、標準 ANSI SQL コマンドの SET TRANSACTION ISOLATION LEVEL を使用して、REPEATABLE READ または SERIALIZABLE 分離レベルでトランザクションを定義します。詳しくは、「*IBM solidDB 管理者ガイド*」の『トランザクション分離レベルの選択』の章を参照してください。

トランザクションは、自動コミットを使用する場合を除き、COMMIT WORK コマンドまたは ROLLBACK WORK コマンドで終了する必要があります。

並行性制御の設定

並行性制御 (ロック方式) の背後にある理論については既に説明しました。このセクションでは、使用する並行性制御のタイプを指定するコマンドについて説明します。

ペシミスティック並行性制御および混合並行性制御の設定

solidDB では、デフォルトでオプティミスティック並行性制御が使用されます。必要に応じて、ペシミスティック (行レベル・ロック方式) 並行性制御または混合並行性制御も使用できます。例えば、頻繁に更新される小さな表を含んだアプリケーションでは、ペシミスティック並行性制御の方が適しています。このようなホット・スポット と呼ばれるケースでは、競合が高い確率で発生するため、オプティミスティック並行性制御で競合するトランザクションをロールバックしても無駄になります。

個々の表にオプティミスティックまたはペシミスティックを設定して、混合並行性制御を行うこともできます。混合並行性制御は、行レベルのペシミスティック・ロック方式とオプティミスティック並行性制御を組み合わせたものです。行レベルのロック方式を表単位でオンにすることで、1 つのトランザクションで両方の並行性制御方式が同時に使用されるように指定できます。この機能は、読み取り専用トランザクションと読み取り/書き込みトランザクションの両方に使用できます。

注:

同期される表では、共有モードでの表レベルのペシミスティック・ロックを使用できます。この機能により、ユーザーはオプティミスティック表でもペシミスティック・モードで同期のための操作を実行することができます。例えば、レプリカでペシミスティック・モードで REFRESH を実行した場合、solidDB はすべての表を共有モードでロックします。サーバーは、後から必要に応じてこのロックを排他的な表ロックへと「プロモート」させることができます。オプションのキーワード PESSIMISTIC を指定すると、この操作が数個の同期ステートメントで実行されます。読み取り操作ではロックが使用されないので注意してください。

オプティミスティックまたはペシミスティックの並行性を表単位で設定するには、以下の SQL コマンドを使用します。

```
ALTER TABLE base_table_name SET {OPTIMISTIC | PESSIMISTIC}
```

デフォルトではすべての表がオプティミスティックに設定されます。

データベース全体のデフォルトを設定することもできます。そのためには、構成ファイルの[General] セクションで以下のパラメーターを指定します。

Pessimistic = yes

ペシミスティック並行性制御を指定すると、ユーザーが照会または更新を行にサブミットしたときに、サーバーは行にロックを設定して整合性と並行性のレベルを制御します。

ロック・タイムアウトの設定

ロック・タイムアウトの設定は、ロックが解放されるまでエンジンが待機する時間 (秒単位) です。デフォルトでロック・タイムアウトは 30 秒に設定されます。タイムアウトの時間に達すると、solidDB はタイムアウトになったステートメントを終了します。例えば、あるユーザーが表の特定の行を照会している場合に別のユーザーが同じ行のデータを更新しようとすると、最初のユーザーの照会が完了するまで (またはタイムアウトになるまで) 更新は行われません。最初のユーザーの照会が完了した時点で 2 番目のユーザーの照会がまだタイムアウトになっていない場合は、

2 番目のユーザーの更新トランザクションに対してロックが発行されます。最初のユーザーの操作が完了する前に 2 番目のユーザーがタイムアウトになった場合は、2 番目のユーザーのステートメントをサーバーが強制終了します。

ロック・タイムアウトを設定するには、以下の SQL コマンドを使用します。

```
SET LOCK TIMEOUT timeout_in_seconds
```

デフォルトの細分度は秒です。値の後に「MS」を付加することで、ロック・タイムアウトをミリ秒の細分度で設定できます。以下に例を示します。

```
SET LOCK TIMEOUT 10MS;
```

「MS」を指定しなければ、ロック・タイムアウトは秒単位となります。

タイムアウトの最大値は 1000 秒 (15 分強) です。サーバーはそれより長い時間を受け付けません。

オプティミスティック表に対するロック・タイムアウトの設定

SELECT FOR UPDATE を使用すると、表のロック方式が「オプティミスティック」に設定されている場合でも、選択した行がロックされます。更新を正常に実行するには、これらの行をロックする必要があります。デフォルトでは、この状況のロック・タイムアウトは 0 秒です。つまり、ユーザーはロックを即座に取得するか、またはエラー・メッセージを受け取ります。サーバーがロックの取得を中止せずに待機して再試行するようにするには、以下の SQL コマンドを使用してオプティミスティック表に対して個別にロック・タイムアウトを設定します。

```
SET OPTIMISTIC LOCK TIMEOUT seconds
```

並行性制御とロック方式

複数のユーザーが同じデータの更新を同時に試みる可能性があるシステムでは、並行アクセスをシステムが制限する必要があります。言い換えれば、システムは一度に 1 人のユーザーだけがデータを変更できるようにする必要があります。

並行性制御は、良質のデータベース・システムなら基本的な機能ですが、このトピックは驚くほど複雑になる可能性があり、いくつかの微妙な点もあります。本書のこのセクションでは、並行性制御とロック方式の動作について、ユーザーの視点から説明します。(ここでは、サーバーが並行性制御を実装する内部メカニズムの大部分については、説明しません。) このセクションでは、以下の点について説明します。

1. 並行性制御の目的
2. 排他ロックおよび共有ロック
3. ペシミスティック並行性制御およびオプティミスティック並行性制御
4. 表ロック
5. ロック期間
6. トランザクション分離レベル
7. 各種情報

ロックとトランザクションについて詳しくは、 129 ページの『5 章 トランザクションの管理』も参照してください。

並行性制御の目的

並行性制御の目的は、2 人のユーザー (または同じユーザーによる 2 つの接続) が同じデータを同時に更新できないようにすることです。また、並行性制御によって、あるユーザーがデータを更新している間に別のユーザーに古いデータが表示されないようにすることができます。並行性制御が必要である理由を、以下の簡単な例で説明します。

ある顧客の当座預金口座に $1,000 があるとします。顧客は今日、この口座に $300 を預金し、口座から $200 を使います。したがって、今日の終わりには口座の残高が $1,100 になっているはずです。しかし、口座の更新が順番にではなく「同時に」実行された場合、一方の更新が別の更新で上書きされる可能性があります。

例えば、銀行の出納係 1 が午前 11:00 に口座を調べて $1,000 があることを確認します。この出納係は $200 を差し引きますが、更新された口座残高 ($800) を即座に保存することができません。午前 11:01 に出納係 2 が口座を調べ、まだ $1,000 の残高があることを確認します。出納係 2 は $300 の預金を加算し、新たな口座残高 $1,300 を保存します。午前 11:09 に出納係 1 が自分の端末に戻って、計算した更新金額 ($800) を入力して保存します。この $800 の金額で $1300 は上書きされます。この日が終わった時点で、$1,100 ($1000 + 300 - 200) であるはずの口座残高は $800 となってしまいます。

2 人のユーザーが「同時に」データを更新できないように (さらに互いの更新を上書きできないように) するために、データベース・ソフトウェアは並行性制御メカニズムを使用します。solidDB は、2 種類の並行性制御メカニズムを提供しています。1 つは「ペシミスティック並行性制御」(通常は単に「ロック」と呼ばれます)、もう 1 つは「オプティミスティック並行性制御」と呼ばれます (これらの用語の根拠については、後述します)。

この例では、簡略化のために、システムの並行性制御メカニズムとしてロックが使用されることを前提としています。

ロックは、データに対する他のユーザーのアクセスを制限するメカニズムです。あるユーザーがレコードでロックを取得すると、そのロックによって他のユーザーはそのレコードを変更 (場合によっては読み取りも) できなくなります。

出納係 1 が口座で作業を開始すると、その口座に「ロック」が設定されます。出納係 1 が口座を更新している間は、出納係 2 が口座の読み取りまたは更新を試みてもアクセスを拒否され、通常はエラー・メッセージが表示されます。ほとんどのデータベース・サーバーでは、データベースの個々のレコードに対してロックが設定されます (表レベルのロックについては後で説明します)。ここで挙げた銀行の例では、出納係が当座預金口座の残高を含んでいるレコードでロックを取得します。普通預金口座の残高がロックされたり、他のユーザーの口座のレコードがロックされたりすることはありません。

ロックを使用すると、並行性を犠牲にして安全性を高めることができます。データの保全性は確保されますが、特定のデータを同時に複数のユーザーが操作することはできなくなります。

排他ロックと共有ロック

排他ロックでは、1 ユーザー/接続のみに特定のデータへのアクセス (読み取りまたは更新) が許可されます。共有ロックでは、複数のユーザーにデータの読み取りが許可されますが、データの更新は許可されません。

ユーザーがデータを更新する際に (ここで使用する銀行の例のように) ペシミスティック並行性制御 (つまりロック方式) を使用する場合、そのユーザーは「排他」ロックを獲得する必要があります。排他ロックが保持されている間は、他のユーザーがそのデータ (銀行口座レコードなど) を読み取ったり更新したりすることはできません。また、ペシミスティック並行性制御を使用している場合、他のユーザーは排他的にロックされているレコードを表示することもできません。これにより、例えば更新されたデータとまだ更新されていないデータが一緒に表示されないようにします。特定のデータに排他ロックを取得できるユーザーは常に 1 人だけです。

2 人のユーザーがデータを単に読み取る (変更を行わない) 場合は、それぞれのユーザーが「共有」ロックを使用できます。例えば、あるユーザーがレコードを更新せずに読み取っている場合は、同時に別のユーザーがそのレコードを表示できます。同じ項目 (レコード、表など) で同時に複数のユーザーが共有ロックを取得する場合があります。例えば、本人、配偶者、銀行担当者、および信用格付け機関の全員が、同時に本人の当座預金口座残高を参照できます (そのうちの誰かが同じタイミングで残高を変更しようとしない限り)。

共有ロックと排他ロックを混用することはできません。あるレコードで排他ロックを取得した場合、同じレコードで共有ロック (または排他ロック) を取得することはできません。

ペシミスティック並行性制御およびオプティミスティック並行性制御

前に述べたように、solidDB は 2 つのタイプの並行性制御メカニズムを備えており、それらは「ペシミスティック」と「オプティミスティック」として知られています。以下では、その両方の方式について説明します。デフォルトでは、solidDB は「オプティミスティック」並行性制御を使用します。

ペシミスティック並行性制御は、単に「ロック」とも呼ばれます。ロックを使用すると、すべてのユーザーが同時に異なるデータを更新している限り、複数のユーザーが 1 つのデータベースを安全に共有できます。例えば、あるユーザーがスミスさんのレコードを更新している間、別のユーザーはクマーさんのレコードを更新します。(さしあたって、ここでは更新操作と排他ロックだけに絞って説明を単純化します。読み取り専用操作や選択操作、および共有ロックについては触れません。)

ロックを使用した場合、行の一部分でも更新されると、直ちにロックが掛けられます。これにより、2 人のユーザーが同時に 1 つの行を更新することが不可能になります。ユーザーがロックを取得すると同時に、他のユーザーは、誰もその行を処理できなくなります。これは、安全で、概念的に単純な手法です。欠点は、実際に複数のユーザーが同じレコードにアクセスを試みているかどうかに関係なく、すべての操作でオーバーヘッドが必要になることです。このオーバーヘッドはわずかなものですが、更新されるすべての行にロックが必要となるので、積もり積もってゆき

ます。さらに、ユーザーがある行にアクセスを試みるたびに、システムは、要求された行が既に別のユーザー (または接続) によってロックされているかどうかを検査する必要もあります。

前に挙げた銀行の出納係の例を拡張すると、出納係 #1 がロックを取得したとき、出納係 #1 と同時に出納係 #2 が同じレコードに対して作業を行う可能性がほとんどない場合でも、出納係 #2 はロックの有無を検査する必要があります。使用するすべてのレコードを検査するには、それなりの時間がかかります。さらに重要な点は、その検査の間、他の出納係は出納係 #2 と同じ検査を試みないことです (そうしないと、両者が 10:59:59 の時点でレコード X が使用されていないことを確認し、両者が 11:00:00 にロックを試みる可能性があります)。このように、ロックの検査自体が、その時点で 2 人のユーザーがロックを変更しないよう、別のロックを必要とする場合さえあります。

ペシミスティック並行性制御 (つまり、ロック) が「ペシミスティック」と呼ばれる理由は、システムが最悪の状態を想定しているからです。つまり、2 人のユーザーが同時に同じレコードを更新しようと考える場合を想定して、実際に競合が発生する可能性がほとんどなくても、レコードをロックすることによって、その可能性を防止します。

ロックに代わる手法は、「オプティミスティック並行性制御」と呼ばれます。オプティミスティック並行性制御は、競合の可能性はあっても、非常にまれであるという想定に立っています。使用されるすべてのレコードを毎回ロックする代わりに、ソフトウェアは、単に 2 人のユーザーが実際に同じレコードを同時に更新しようとした形跡だけを探します。その形跡が見つかった場合は、1 人のユーザーの更新が廃棄されます (また、そのユーザーに通知が出されます)。

競合が発生した後に (競合を発生前に防止するのでなく)、サーバーがその競合を検出できる 1 つの方法を、以下に説明します。説明を簡単にするために、以下の一連のアクションによって更新が行われるとします。

1. データをディスク・ドライブからメモリーに読み取ります。
2. メモリー内のデータを更新します。
3. 更新したデータをディスク・ドライブに書き戻します。

(原則は、更新されたデータがディスク・ドライブ以外のデバイスに書き込まれる場合でも同じです。)

オプティミスティック・ロック方式を使用した場合、サーバーは更新しようとするレコードを読み取るたびに、そのレコードの「バージョン番号」のコピーを作成し、そのコピーを後で参照するために保管します。更新したデータをディスク・ドライブに書き戻すとき、サーバーは当初に読み取ったバージョン番号を、その時点でディスク・ドライブに入っているバージョン番号と比較します。それらのバージョン番号が同じなら、ほかに誰もそのレコードを変更していないことになるので、更新した値を書き込むことができます。しかし、当初に読み取った値とディスク上の現行値が同じでない場合は、データを読み取った後に誰かがデータを変更したことになり、こちら側で実行した操作は、すべて古くなっている可能性があります。このため、こちら側のデータのバージョンを廃棄し、ユーザーにエラー・メッセージを与えます。当然ながら、レコードを更新するたびに、バージョン番号も更新されます。

オプティミスティック・ロック方式を使用した場合は、更新したデータを書き込む直前まで競合の存在が分かりません。ペシミスティック・ロック方式では、データを読み取ろうとすると、直ちに競合の存在が分かります。再び銀行との類推を使用すると、ペシミスティック・ロック方式は、銀行の入り口に警備員がいて、中に入ろうとするユーザーの口座番号を確認するようなものです。既に他の誰か (配偶者か、小切手を切った相手) が銀行内にいて、こちらの銀行口座にアクセスしている場合は、その別の人間が取引を終了して出てくるまで、中に入ることはできません。一方、オプティミスティック・ロック方式では、いつでも銀行内に入って行き、取引を試みることができます。しかし、銀行を出ようとしたときに警備員から、取引が他の誰かと競合したために、戻って取引を再度行う必要があると告げられるリスクがあります。

オプティミスティックとペシミスティックの並行性制御には、競合の検出とエラー・メッセージの発行の時期以外に、もう 1 つの重要な違いがあります。ペシミスティック・ロック方式では、あるユーザーが別のユーザーによる同じレコードの更新をブロックできるだけでなく、そのレコードの読み取りさえもブロックできます。ペシミスティック・ロック方式を使用して排他ロックを取得した場合、他のユーザーはそのレコードを読み取ることさえできなくなります。しかし、ペシミスティック・ロック方式では、更新したデータをディスクに書き込むとき以外、競合の有無が検査されません。ユーザー 1 がレコードを更新し、ユーザー 2 がそのレコードを読みたいだけの場合、ユーザー 2 は単純にディスク上にあるデータを読み取って先へ進むだけで、データがロックされているかどうかを検査しません。ユーザー 1 がデータを読み取って更新しても、まだそのトランザクションを「コミット」していない場合、ユーザー 2 はわずかに古くなった情報を見る可能性があります。

solidDB は、実際にはそれよりもっと洗練された方法でオプティミスティック並行性制御を実装します。各ユーザーに「ディスク上のデータが読み取られた瞬間におけるデータのバージョン」を与えるのでなく、solidDB は各データ行の複数のバージョンを一時的に保管できます。各ユーザーのトランザクションから見えるデータベースは、トランザクションの開始時点におけるデータベースです。このため、それぞれのユーザーに見えるデータは、そのトランザクション中は一貫性を持っており、複数のユーザーが並行してデータベースにアクセスできます。ロックが使用されないため、データは常に使用可能であり、デッドロックがもはや適用されないため、アクセスが向上します。 (ただし、この場合でも、他のユーザーの変更と競合する変更を行ったときは、その変更が廃棄されるリスクがあります。) マルチバージョン管理がどのように行われるのかについて詳しくは、「*IBM solidDB 管理者ガイド*」の『*solidDB Bonsai ツリーのマルチバージョン管理と並行性制御*』という表題のセクションを参照してください。

上記のオプティミスティックおよびペシミスティック並行性制御に関する説明は、少し単純化されています。特定の条件下では、表にペシミスティック・ロック方式が使用されている場合でも、またその表内のレコードに排他ロックが掛かっていても、ロックされたレコードに対して別のユーザーが読み取り操作を実行できる場合があります。読み取り側ユーザーがトランザクションを読み取り専用トランザクションとして明示的に設定した場合、そのユーザーはロック方式でなくバージョン管理方式を使用できます。これができるのは、ユーザーが以下のコマンドを使用して、トランザクションを読み取り専用として明示的に宣言した場合だけです。

```
SET TRANSACTION READ ONLY;
```

これにより、例えばユーザー 1 は、あるレコードに排他ロックを掛け、それを更新できます。レコードが更新された時点で、そのバージョン番号は変更されます。ユーザー 2 は、読み取り専用トランザクションを使用している場合、レコードに排他ロックが掛かっていても、前のバージョンのレコードを読み取ることができます。

ペシミスティック・ロック方式では、オプティミスティック・ロック方式が提供しないオプションを使用できることに注意してください。前に述べたように、ペシミスティック・ロックは「即時に」失敗します。つまり、あるレコードについての排他ロックを取得しようとした場合、既に別のユーザーがそのレコードに対するロック (共有または排他) を保有していると、ロックを取得できないことを告げられます。実際に、solidDB では、即時に失敗するか、それとも特定の秒数だけ待ってから失敗するかを選択できます。例えば、30 秒間の待ちを指定した場合、最初にロックの取得を試みて取得できなければ、サーバーはロックを取得できるか 30 秒が経過するまで、ロックを取得しようとし続けます。多くの場合、特にトランザクションが非常に短時間で終わる傾向がある場合は、短い待ちを設定することにより、そうしなかった場合にロックによってブロックされたはずのアクティビティーを続行できることがあります。

この待ちメカニズムは、ペシミスティック・ロック方式にのみ適用され、オプティミスティック並行性制御には適用されません。「オプティミスティック・ロックを待つ」というようなことは存在しません。データを読み取った後に、他の誰かがそのデータを変更した場合、いくら待っても、既に発生した競合を防止することはできません。実際に、オプティミスティック並行制御方式ではロックが掛けられないので、待つべき「オプティミスティック・ロック」は事実上存在しません。

注: SELECT FOR UPDATE を実行した場合、サーバーは更新モード・ロックを使用するため、他のユーザーはその行の読み取りや更新ができなくなり、現行ユーザーは確実に行を更新できます。詳しくは、 139 ページの『共有ロック、排他ロック、および更新ロック』、 129 ページの『並行性制御の設定』、および 131 ページの『オプティミスティック表に対するロック・タイムアウトの設定』を参照してください。

ペシミスティックとオプティミスティックのどちらの並行性制御にも、「正しい」とか「誤り」とかはありません。正しく実装されれば、どちらの手法でもデータが正しく更新されることが保証されます。ほとんどのシナリオでは、オプティミスティック並行性制御の方が効率が良く、パフォーマンスも優れていますが、一部のシナリオにはペシミスティック・ロック方式の方が適しています。多数の更新があり、複数のユーザーが同時にデータの更新を試みる機会が比較的に多い状況では、ペシミスティック・ロック方式が好ましいでしょう。競合が起きる確率が非常に低い場合 (多数のレコードと比較的に少数のユーザー、または、非常に少数の更新と大部分が「読み取り」の操作) には、通常、オプティミスティック並行性制御が最良の選択肢です。この決定には、1 人のユーザーが一度に更新できるレコードの数も影響を及ぼします。銀行の例では、通常の場合、一度に 1 つの口座/レコードだけを更新します。しかし、一部のアプリケーションでは、それぞれの操作が一度に多数のレコードを更新する場合があり (例えば、銀行は各月末に、すべての口座に利息を追加する場合があります)、そのような 2 つのアプリケーションが同時に実行されれば、ほぼ確実に競合が発生します。

オプティミスティック・ロック方式をオーバーライドし、代わりにペシミスティック・ロック方式を指定できます。これは、個々の表のレベルで行うことができます。1 つの表がオプティミスティック・ロック方式の規則に従い、別の表がペシミスティック・ロック方式のルールに従っていてもかまいません。両方の表を同じトランザクション内で使用でき、同じステートメントの中でさえ使用できます。細部の処理は、solidDB がユーザーに代わって実行してくれます。オプティミスティックとペシミスティックの指定方法について詳しくは、『並行性 (ロック方式) モードをオプティミスティックまたはペシミスティックに設定する』を参照してください。

読者は「オプティミスティック・ロック方式」が果たして本当にロック・スキームなのかと疑問を感じるかもしれません。オプティミスティック・ロック方式を使用した場合、実際には何もロックを掛けません。したがって、「オプティミスティック・ロック方式」という名前は誤解を招くおそれがあります。しかし、オプティミスティック・ロック方式はペシミスティック・ロック方式と同じ目的 (重複する更新の防止) に役立つものです。したがって、基礎となっているメカニズムは真のロックではありませんが、「ロック方式」という名前が付いています。

重要: デフォルトでは、solidDB サーバーはディスク・ベースの表にオプティミスティック・ロック方式を使用します。オプティミスティック・ロック方式を使用すると、高速のパフォーマンスと高度な並行性 (複数のユーザーによるアクセス) を得ることができますが、その代償として、当初は受け入れられたデータ書き込みが、最後の瞬間に別のユーザーによる変更と競合していることが判明して、「拒否」されることがあります。

一方、インメモリー表では、ペシミスティック並行性制御のみが使用可能です。それは、メモリーの保存に良い効果があるためです。

並行性 (ロック方式) モードをオプティミスティックまたはペシミスティックに設定する

インメモリー表を使用するトランザクションの分離レベルが READ COMMITTED より高い場合、サーバーはその表に対してペシミスティック並行性制御を使用します。

それ以外の表には以下のルールが使用されます (優先順位の高い順に示しています)。

1. ALTER TABLE コマンドを使用して特定の表に並行性モードを設定できます。以下に例を示します。

   ```
   ALTER TABLE MyTable SET PESSIMISTIC;
   ALTER TABLE MyTable SET OPTIMISTIC;
   ```

2. `solid.ini` 構成パラメーター **General.Pessimistic** を設定することで、すべての表にデフォルトの並行性モードを設定できます。以下に例を示します。

   ```
   [General]
   Pessimistic=yes
   ```

   このパラメーターは、サーバーを始動した時点で有効となるので注意してください。`solid.ini` ファイルを手動で編集した場合は、サーバーを再始動するまで変更が反映されません。

4.0 以前のバージョンでは、ADMIN COMMAND でこのパラメーターを設定できないことにも注意してください。

3. 上記の方法で並行性モードを指定しなかった場合は、デフォルトでオプティミスティック並行性制御方式が設定されます。

General.Pessimistic の値は変更可能であるため、表に対する並行性制御も変化する可能性があります。表が、サーバーの 1 回の「実行」中にオプティミスティック並行性制御を使用し、別の実行中にはペシミスティック並行性制御を使用する可能性が十分にあります。

表の設定が **General.Pessimistic** パラメーターに基づいている場合、その表では表が作成されたときの値ではなく、**General.Pessimistic** パラメーターの現行値が使用されます。

並行性モードの読み取り

READ COMMITTED よりも高い分離レベルのトランザクションでインメモリー表を使用する場合、サーバーはペシミスティック並行性制御を使用するので、以下のルールは無視する必要があります。

その他のすべての表の場合、表の並行性モードを読み取る単一の方式はありません。以下のステップに従って、目的の表の並行性モードを判別する必要があります。

1. 表の並行性モードが明示的に ALTER TABLE コマンドで設定されている場合、その表の並行性モードはシステム表 SYS_TABLEMODES に記録されています。以下のコマンドを実行して、値を読み取ることができます。

   ```
   SELECT SYS_TABLEMODES.ID, table_name,
   FROM SYS_TABLES, SYS_TABLEMODES
   WHERE SYS_TABLEMODES.ID = SYS_TABLES.ID;
   ```

   これは、表の並行性モードを ALTER TABLE コマンドで明示的に設定した場合にのみ機能することに注意してください。

2. 表の並行性モードが ALTER TABLE コマンドで設定されていない場合、サーバーを始動した時点で `solid.ini` ファイルによって指定された並行性制御モードを確認します。このレベルは、以下のコマンドを実行して読み取ることができます。

   ADMIN COMMAND ’describe parameter general.pessimistic’;

   `solid.ini` ファイルの値がサーバー始動時から変更されておらず、ADMIN COMMAND でオーバーライドされていない場合は、`solid.ini` ファイルを見て値を判別できます。

   (注: バージョン 4.00.0031 より前では、サーバーは **General.Pessimistic** 変数の値を表示する ADMIN COMMAND を正しく認識していませんでした。つまり、古いバージョンのサーバーでは、`solid.ini` ファイルの値を調べる必要があります。`solid.ini` ファイルの値が、サーバー始動後に変更された場合、正しい値はわかりません。)

3. 上記のどれにも当てはまらない場合、すべての表に対し、サーバーはデフォルトのオプティミスティック並行性制御を用います。

共有ロック、排他ロック、および更新ロック

以下のロック・モードは、ペシミスティック・ロック方式を使用する表内の行にのみ使用されます。

- SHARED

  複数のユーザーが同じ行に同時に共有ロックを保持できます。共有ロックは、読み取り専用操作または SELECT 操作で使用されます。共有ロックでは、複数のユーザーにデータの読み取りが許可されますが、データの変更はどのユーザーにも許可されません。

- EXCLUSIVE

  あるユーザーが行で排他ロックを取得すると、その行には他のどのタイプのロックも設定できなくなります。したがって、排他ロックを取得したユーザーはその行に排他的にアクセスできます。排他ロックは、挿入、更新、削除の各操作で使用されます。

- UPDATE

  ユーザーが SELECT... FOR UPDATE ステートメントで行にアクセスすると、その行は更新モード・ロックでロックされます。つまり、他のユーザーはその行を読み取ったり更新したりできなくなり、現在のユーザーが後で確実にその行を更新できます。更新ロックは排他ロックと似ています。両ロックの最大の違いは、他のユーザーが共有ロックを取得しているレコードで更新ロックを獲得できる点です。これにより更新ロックの保有者は、他のユーザーを排除することなくデータを読み取ることができます。ただし、更新ロックの保有者がデータを変更すると、更新ロックは排他ロックに変換されます。更新ロックの特徴は、共有ロックに関して非対称的であるという点です。ユーザーは、共有ロックが設定されているレコードで更新ロックを獲得できますが、更新ロックが設定されているレコードで共有ロックを獲得することはできません。更新ロックを設定するとそれ以降の読み取りロックが設定されなくなることから、更新ロックは容易に排他ロックに変換できます。

表ロック

ここまでは、主に表内の個々の行 (当座預金残高を格納する銀行口座情報など) のロックについて説明しました。サーバーでは、行レベルのロックだけでなく表レベルのロックも設定できます。個々のレコードのロックに適用される原則の多くは、表のロックにも適用されます。

なぜ表のロックが必要なのでしょうか。例えば、表に新しい列を追加するとします。他のユーザーが同時に同じ名前の列を追加しないようにする必要があります。

このため、ALTER TABLE 操作を実行するときに、対象の表で共有ロックを取得します。これにより、他のユーザーは表からデータを読み取ることは引き続き可能ですが、表に変更を加えることはできなくなります。別のユーザーが同時に同じ表で DDL 操作 (ALTER TABLE など) を実行しようとすると、そのユーザーは待たされるか、またはエラー・メッセージを受け取ります。

このように、基本的な表ロック方式の目的とメカニズムは、レコード・ロック方式と同じです。ただし、表ロック方式が使用される状況はほかにもあります。つまり、あるユーザーが表の構造を更新しようとする場合だけではありません。

表内のレコードを更新する場合を考えてみます。例えば、顧客の自宅電話番号を更新するとします。一方で、別のユーザーが電話番号列をドロップし、E メール・アドレス列を追加して表を変更しようとしています。別のユーザーが電話番号列をドロップした後に、存在しなくなったその列に更新した電話番号を書き込もうとすると、間違いなくデータが破損します。このため、ユーザーが表内のレコードに対して共有ロックまたは排他ロックを取得するときは、暗黙的に表全体に対してもロック (通常は共有ロック) を取得することになります。これにより、ユーザーが表の一部を使用しているときにその表の構造が変更されることを防ぎます。

表レベルのロックは常に「ペシミスティック」です。つまり、サーバーは単にバージョン管理情報を参照するのではなく、その表に実際にロックを設定します。これは、表のロック方式がオプティミスティックに設定されている場合も同様です (この説明はわかりづらいかもしれません。表にロック・モードを設定するときは、実際には表そのものではなく表内の行にロック・モードを設定していることを念頭に置いてください。つまり、表レベルのロックではなく行レベルのロックを設定しています)。

表を変更している場合を除き、表に対するロックは通常は共有ロックです。この表ロックの通常のタイムアウトは 0 秒です。つまり、ロックを直ちに取得できなかった場合、サーバーは待機せずにエラー・メッセージを表示します。

表全体をロックする 3 番目の理由があります。表内のすべてのレコードを 1 つのトランザクションで変更する場合を考えてみましょう。例えば、1 月 1 日の午前 12:01 にすべての普通預金口座に前年の利息を入金するとします。表内の各レコードで個別に排他ロックを取得することもできますが、効率的ではありません。表全体で排他ロックを取得することが必要です。表のすべてのレコードで可能性のあるロックを検査するよりも、この 1 つのロックを検査する方が効率的です。当然ながら、他のユーザーがその表でロックを取得している場合 (表内のレコードをロックすることで共有表ロックを獲得した場合など)、その表では排他ロックを取得できません。排他/共有ロックに関するルールは、表もレコードも同じです。つまり、共有ロックは必要な数だけ取得できますが、排他ロックは同時に 1 つしか設定できず、排他ロックと共有ロックを組み合わせて使用することもできません。

サーバーは、特定の操作 (WHERE 節のない UPDATE ステートメントなど) が表内のすべてのレコードに影響すると認識した場合に、表全体のロックが最も効率的であり、かつ対象の表に競合するロックが存在しないことを条件として表全体をロックする可能性があります。

このように、表ロックは少なくとも 3 つの目的で使用されます。

1. 同時に 2 人のユーザーが表を変更することを防ぐ。
2. 表内のレコードが変更されているときに表が変更されることを防ぐ。
3. 一括更新を行う操作の効率を高める。

表レベルのロックのほとんどは暗黙的に設定されます。つまり、サーバー自体が必要に応じてそれらのロックを設定します。ただし、LOCK TABLE コマンドを使用

して表レベルのロックを明示的に設定することもできます。これは、保守モード機能セットを使用する場合に便利です。詳しくは、「*IBM solidDB 拡張レプリケーション・ユーザー・ガイド*」の『分散システムのスキーマの更新および保守』という章を参照してください。

表レベル・ロック方式

EXCLUSIVE および SHARED ロック・モード (139 ページの『共有ロック、排他ロック、および更新ロック』を参照) は、ペシミスティック表とオプティミスティック表の両方に使用されます。

注:

デフォルトでは、オプティミスティック表とペシミスティック表は常に共有モードでロックされます。さらに、オプションで PESSIMISTIC キーワードを指定できる一部の solidDB ステートメントは、表がオプティミスティックの場合でも EXCLUSIVE 表レベル・ロックを使用します。

ロック期間

トランザクション (まとめてコミットまたはロールバックされる一連のステートメント) の目的は、内部的なデータの整合性を確保することにあります。そのためには、トランザクションが終了するまでロックが保持される必要があります。

まずトランザクションの対象を確認しましょう。例えば新しい自転車を購入し、小切手で支払ったとします。銀行は、自転車の代金を購入者の口座から出金し、自転車販売店の口座にその代金を入金する必要があります。この 2 つの操作は「一緒に」実行する必要があります。そうしないと、代金の出金先や入金元がわからなくなるおそれがあります。例えば、購入者の口座から代金を出金してトランザクションをコミットした後に、自転車販売店の口座の更新が失敗することが考えられます (販売店の口座を更新する直前に電源障害が発生した場合など)。この場合、購入者の残高は減りますが、販売店の残高は増えないことになります。つまり代金が消失したように見えます (購入者が既に支払った代金を再び怒った販売店から請求されることになります)。

2 つの操作 (購入者の口座からの出金と販売店の口座への入金) を同じトランザクションにまとめれば、代金が消失することはありません。電源障害などの理由でトランザクションが中断されてロールバックされた場合、同じ操作を後で再実行しても購入者が二重請求される (あるいは販売店に支払が行われない) リスクは生じません。

一般に、更新ロックは獲得された時点からトランザクションがコミットまたはロールバックによって完了する時点まで保持されます。トランザクション終了時までロックが保持されないと、ロールバックが失敗する可能性があります。(レコードを更新してからトランザクションが終了するまでの間に他のユーザーがそのレコードを更新するとどうなるかを考えてみてください。なんらかの理由でロールバックが必要となった場合、他のユーザーが自分のトランザクションを続行してコミットしても、サーバーはそのユーザーによる変更をロールバックするかどうかを判断する必要があります。あるいは単純にその変更を無効にする可能性があります。)

solidDB では、共有ロック (「読み取りロック」) もトランザクション終了時まで保持されます。この点で、solidDB サーバーは他の一部のサーバーと異なります。一部のサーバーでは、トランザクション分離レベルが低い場合に、トランザクションの終了前に共有ロックが解放されます。

共有ロックが常にトランザクション終了時まで保持される場合、共有ロックに関するサーバーの動作にトランザクション分離レベルが影響するのかどうかという疑問が生じます。ロックがトランザクション終了時まで保持される場合でも、分離レベル間にはある程度の違いがあります。例えば、SERIALIZABLE 分離レベルでは追加の検査が行われます。また、トランザクションで生成されるはずだった結果セットに新たな行が追加されていないことも検査されます。つまり、トランザクション内にある結果セットに対する資格を持った他のユーザーによって行が挿入されないようにします。例えば、以下の更新コマンドを使用する SERIALIZABLE トランザクションがあるとします。

```
UPDATE customers SET x = y WHERE area_code = 415;
```

SERIALIZABLE トランザクションでは、トランザクションがコミットされるまで他のユーザーが area_code=415 を指定してレコードを入力することがサーバーによって禁止されます。

トランザクション分離について詳しくは、次のセクションを参照してください。

トランザクション分離レベル

「単純な」世界では、データの参照が終了すると直ちに共有ロックが解放されます (前のセクションで説明したように、更新ロックはトランザクションが終了するまで保持されます)。

しかし現実はそれほど単純ではありません。場合によっては、ユーザーが 1 つのトランザクションで同じレコードを複数回参照することがあります。例えば、スクロール・カーソルを使用するプログラムでは、ユーザーが一連のレコードを前後にスクロールして同じレコードを何度も表示する可能性があります。同じトランザクション内でユーザーがそのレコードを参照するたびにレコードの値が変化すると、ユーザーは混乱します。このため、多くのデータベース・サーバー (特に SQL 言語に関する ANSI 規格および ISO 規格に準拠しているサーバー) では、読み取り/共有ロックの期間を延長できるようになっています。これは、ユーザーがデータを表示するたびに (1 つのトランザクション内で) 同じデータが表示されるようにすることを意図しています。レコードを読み取ると、そのレコードに共有ロックが設定され、トランザクションが終了するまで保持されます。

(これはトランザクション分離レベルに関係する要素の 1 つに過ぎません。トランザクション分離レベルは、レコードがロックされる時間だけでなく、表示される内容にも影響します。例えば、solidDB とは違って「READ UNCOMMITTED」(「ダーティー読み取り」と呼ばれることもあります) と「READ COMMITTED」の両方を許可するシステムでは、一部のレコードをロックしているために、分離レベルが他のユーザーが表示できる内容だけでなく自分の表示内容にも影響します。)

solidDB では、分離レベルを、構成パラメーターでグローバルに設定するか、セッション単位およびトランザクション単位で設定できます。詳しくは、*solidDB 管理者ガイド*」の『トランザクション分離レベルの選択』という章を参照してください。

各種のロック情報

特定のカテゴリー内のすべてのロック (共有ロックなど) は、同等です。誰がロックを掛けたかは、問題にはなりません。DBA によって掛けられたロックは、他のユーザーによって掛けられたロックより強いわけでも弱いわけでもありません。ロックが対話式に入力されたステートメントの一部として実行されたのか、コンパイルされたリモート・アプリケーションから呼び出されたのか、それとも共有メモリー・アクセスまたはリンク・ライブラリー・アクセスを使用したときにローカル・アプリケーションの中から呼び出されたのかは、問題になりません。また、ロックがストアード・プロシージャーまたはトリガーの内部にあるステートメントの結果として掛けられたのかも問題になりません。

ペシミスティック・ロック方式では、ロックを最初に要求したユーザーがロックを取得します。ロックを取得した後、他のユーザーまたは接続がそのロックを無効にすることはできません。solidDB では、ロックはトランザクションの終わりまで継続するか、長い表ロックの場合は、ユーザーが明示的に解除するまで継続します。

一部のロックはエスカレートする場合があることに注意してください。例えば、スクロール・カーソルを使用しており、レコードに対する共有ロックを獲得した後、同じトランザクション内でそのレコードを更新した場合、獲得した共有ロックが排他ロックにアップグレードされる場合があります。排他ロックを取得することは、その表に対して他のロック (共有ロックまたは排他ロック) が存在しない場合にのみ可能です。ユーザーが同じレコードに対して別のユーザーと一緒に共有ロックを持っている場合、サーバーは、別のユーザーが自身の共有ロックをドロップするまで、ユーザーの共有ロックを排他ロックにアップグレードできません。

表ロック

表ロックは通常、「保守モード」操作で使用されますが、これら 2 つの機能は独立したものです。表ロック機能は、保守モード機能と一緒に使用しても、一緒に使用しなくてもかまいません。

レプリカでは、PESSIMISTIC キーワードでリフレッシュする場合、パブリケーション表に対して排他ロックが暗黙に発行されます。

solidDB は、すべての DDL および DML 操作で暗黙の表共有ロックを発行します。これにより、あるユーザーが表内のデータを更新しているときに、別のユーザーがその表をドロップすることが防止されます。

情報ロックの要約

ロックは、2 人のユーザーによって競合する操作が同時に実行されないようにする機能です。操作が「競合」するのは、少なくともいずれかの操作がデータの更新 (UPDATE、DELETE、INSERT、ALTER TABLE など) に関係している場合です。すべての操作が読み取り専用操作 (SELECT など) であれば、競合は発生しません。現行バージョンの solidDB では、行レベルのロックをユーザーが明示的に指定することはできません。「LOCK RECORD」コマンドは存在しません。サーバーが常に行レベルでロックを行います。サーバーは表レベルでもロックを行います。表レベルのロックを明示的に設定する必要がある場合は、ユーザーが LOCK TABLE コマンドを使用して設定します。

トランザクション持続性の選択

わずかなら最新のデータを失ってもかまわない場合、およびパフォーマンスが重要である場合は、リラックス持続性を使用するとよいでしょう。リラックス持続性は、個々のトランザクションに重大な意味がない場合に適しています。例えば、システム・パフォーマンスをモニターしており、応答時間についてのデータを保管したい場合は、データのわずかな部分が欠落しても大きな影響がない平均応答時間だけに、関心を向けることができます。実際に、パフォーマンスの測定自体が (CPU 時間と入出力帯域幅などのリソースを使用し尽くすことにより) パフォーマンスに影響を及ぼすため、多くの場合、パフォーマンス追跡操作自体には、高い精度よりも高いパフォーマンス (低いコスト) が期待されます。リラックス持続性は、そのような状況に適しています。

一方、請求書の支払いなどの財務データを追跡している場合は、コミットしたデータの 100% が保管されてリカバリー可能であることを保証したいと考えるでしょう。そのような状況では、ストリクト持続性が必要になります。

リラックス持続性は、少数の最新のトランザクションが失われてもかまわない場合にだけ、使用してください。それ以外の場合は、ストリクト持続性を使用してください。ストリクト持続性とリラックス持続性のどちらが適切かがよく分からない場合は、ストリクト持続性を使用してください。

トランザクション持続性レベルの設定

トランザクション持続性レベルを設定するには 4 とおりの方法があります。以下に、これらの方法を優先順位の高いものから降順に示します。

1. SET TRANSACTION DURABILITY

   ```
   SET TRANSACTION DURABILITY { RELAXED | STRICT }
   ```

   例

   ```
   SET TRANSACTION DURABILITY RELAXED;
   SET TRANSACTION DURABILITY STRICT;
   ```

   SET TRANSACTION DURABILITY コマンドを使用すると、トランザクションの持続性がトランザクション単位で設定されます。このコマンドは現行トランザクションにのみ作用します。

2. SET DURABILITY

   ```
   SET DURABILITY { RELAXED | STRICT }
   ```

   例

   ```
   SET DURABILITY RELAXED;
   SET DURABILITY STRICT;
   ```

   SET DURABILITY コマンドを使用すると、トランザクションの持続性がセッション単位で設定されます。セッション とは、サーバーに接続してから切断するまでの時間です。セッションはユーザーごとに存在します。これはセッションの時間がオーバーラップしていても同様です。1 人のユーザーが複数のセッションを確立することもあります (例えば、solsql の複数のコピーを実行する場合や、同じサーバーとの間に複数の接続を作成するプログラムを作成した場合など)。SET DURABILITY ステートメントを使用してトランザクション持続性レベルを

指定すると、そのコマンドを発行したセッションに対してのみその持続性レベルが設定されます。この選択は、他のユーザー、現在使用しているセッション以外のオープン・セッション、今後使用するセッションのいずれにも影響しません。各ユーザー・セッションで、それぞれのデータを損失しないことの重要性に基づいて、独自にトランザクション持続性レベルを設定できます。

このステートメントの効果は、セッションが終了するまで、または別の SET DURABILITY コマンドが発行されるまで持続します。

3. `solid.ini` 構成ファイル内の **DurabilityLevel** パラメーターを設定します。

```
[Logging]
DurabilityLevel=3
```

「*IBM solidDB 拡張レプリケーション・ユーザー・ガイド*」の『*DurabilityLevel*』という章を参照してください。

この設定はすべてのユーザーに作用します。

このパラメーターは動的に変更できます。サーバーの実行中にデフォルトの設定を変更する場合は、以下のコマンドを使用します。

```
ADMIN COMMAND 'parameter Logging.DurabilityLevel={1 | 2 | 3}'
```

このコマンドは、実行すると直ちに有効となります。

4. 上記の方法でトランザクション持続性レベルを設定しなかった場合、サーバーはデフォルトでリラックス持続性を使用します (**Logging.DurabilityLevel=1**)。

ストリクト持続性を使用する場合は、追加の構成パラメーター (**LogWriteMode**) を設定することもできますが、これもパフォーマンスに影響を与えます。**LogWriteMode** について詳しくは、「*IBM solidDB 管理者ガイド*」のこのパラメーターの説明を参照してください。

6 診断およびトラブルシューティング

この章では、以下の solidDB 診断ツールについて説明します。

- SQL 情報機能および EXPLAIN PLAN FOR ステートメント。アプリケーションをチューニングし、アプリケーション内の非効率的な SQL ステートメントを特定する場合に使用します。
- ストアード・プロシージャーおよびトリガーのトレース機能

これらの機能を使用して、パフォーマンスの監視、問題のトラブルシューティング、および高品質の問題報告書の作成を行うことができます。これらのレポートでは、問題の原因が製品カテゴリー (solidDB ODBC API、solidDB ODBC ドライバー、solidDB JDBC ドライバーなど) ごとに分けられているため、原因を特定しやすくなっています。

パフォーマンスの監視

SQL 情報機能を使用して、ある SQL ステートメントおよび SQL ステートメント EXPLAIN PLAN FOR に関する情報を提供し、指定された SQL ステートメントについて SQL オプティマイザーが選択した実行グラフを表示できます。一般に、IBM の技術支援担当窓口に連絡する必要がある場合は、SQL ステートメント、EXPLAIN PLAN 出力、および、さらに詳細なトレース出力を得るために情報レベル 8 で実行した EXPLAIN PLAN からの SQL 情報出力を提供するよう要請されます。

SQL 情報機能

アプリケーションを実行するときは、SQL 情報機能を有効にしてください。SQL 情報機能は、solidDB で処理された各 SQL ステートメントの情報を生成します。

[SQL] セクションの Info パラメーターに、SQL パーサーおよびオプティマイザーでの追跡レベルを 0 (トレースなし) から 8 (フェッチされたすべての行から solidDB 情報を取得) の整数で指定します。トレース情報は、solidDB ディレクトリーの `soltrace.out` というファイルに出力されます。

例:

```
[SQL]
info = 1
```

表 18. SQL Info のレベル

Info の値	情報
0	出力なし
1	SQL 形式での表、索引、およびビューの情報
2	SQL 実行グラフ (IBM が技術支援に使用)
3	SQL 見積もり情報の一部、solidDB で選択されたキー名

表 18. *SQL Info のレベル (続き)*

Info の値	情報
4	すべての SQL 見積もり情報、solidDB で選択されたキー情報
5	破棄されたキーからの solidDB 情報も含む
6	solidDB の表レベルの情報
7	フェッチされたすべての行からの SQL 情報
8	フェッチされたすべての行からの solidDB 情報

SQL 情報機能は、以下の SQL ステートメントでオンにすることもできます (この場合、ステートメントを実行するクライアントのみに対して SQL Info がオンに設定されます)。

```
SET SQL INFO ON LEVEL info_value FILE file_name
```

オフにする場合は以下の SQL ステートメントを使用します。

```
SET SQL INFO OFF
```

例:

```
SET SQL INFO ON LEVEL 1 FILE 'my_query.txt'
```

EXPLAIN PLAN FOR ステートメント

EXPLAIN PLAN FOR ステートメントの構文は以下のとおりです。

```
EXPLAIN PLAN FOR sql_statement
```

EXPLAIN PLAN FOR ステートメントは、指定した SQL ステートメントに対して SQL オプティマイザーが選択した実行プランを表示するために使用します。実行プランとは、solidDB がステートメントを実行するために実行する一連のプリミティブ操作とその順序です。実行プランに含まれる各操作はユニットと呼ばれます。

表 19. *EXPLAIN PLAN FOR のユニット*

ユニット	説明
JOIN UNIT*	結合ユニットは、複数の表を結合します。結合は、ループ結合またはマージ結合を使用して実行できます。
TABLE UNIT	表ユニットは、表または索引からデータ行をフェッチするために使用されます。
ORDER UNIT	順序ユニットは、グループ化または ORDER BY に対応して行を順序付けるために使用されます。順序付けは、メモリー内で、または外部ディスク・ソーターを使用して実行できます。
GROUP UNIT	グループ・ユニットは、グループ化および集約計算 (SUM、MIN など) を行うために使用されます。
UNION UNIT*	和ユニットは、UNION 操作を実行します。このユニットは、ループ結合またはマージ結合を使用して実行できます。

表 19. EXPLAIN PLAN FOR のユニット (続き)

ユニット	説明
INTERSECT UNIT*	積ユニットは、INTERSECT 操作を実行します。このユニットは、ループ結合またはマージ結合を使用して実行できます。
EXCEPT UNIT*	差ユニットは、EXCEPT 操作を実行します。このユニットは、ループ結合またはマージ結合を使用して実行できます。

*このユニットは、1 つの表のみを参照する照会に対しても生成されます。その場合、ユニットでは結合が実行されず、行が操作されることなく渡されます。

EXPLAIN PLAN FOR ステートメントから返される表は、以下の列で構成されます。

表 20. EXPLAIN PLAN 表の列

列名	説明
ID	出力行番号。行がユニークであることを保証するためにのみ使用されます。
UNIT_ID	SQL インタープリターが内部で使用するユニット ID。ユニットごとに ID は異なります。ユニット ID は疎番号シーケンスです。これは、SQL インタープリターが最適化フェーズで削除されるユニットにもユニット ID を生成するためです。同じユニット ID を持つ行が複数ある場合、それらの行は同じユニットに属しています。フォーマット上の理由から、1 つのユニットの情報が複数の行に分割されることもあります。
PAR_ID	ユニットの親ユニット ID。親 ID 番号は、UNIT_ID 列の ID を参照します。
JOIN_PATH	結合、和、積、差の各ユニットには、ユニットで結合される表および表の結合順序を指定する結合パスが存在します。結合パス番号は、UNIT_ID 列のユニット ID を参照します。つまり、ユニットへの入力はそのユニットから送られます。表が結合される順序は、結合パスがリストされている順序です。最初にリストされるのはループ結合の最外部の表です。
UNIT_TYPE	ユニット・タイプは、実行グラフ・ユニット・タイプです。
INFO	INFO 列は、追加の情報用に予約されています。例えば、索引の使用率、データベース表の名前、solidDB で行を選択するために使用される制約などが格納されます。ここに格納された制約が、SQL ステートメントに指定された制約と一致しない場合があるので注意してください。

INFO 列には、ユニットのタイプごとに以下のテキストが格納されている可能性があります。

表 21. ユニットの INFO 列内のテキスト

ユニット・タイプ	INFO 列のテキスト	説明
TABLE UNIT	*tablename*	表ユニットが表 *tablename* を参照しています。
TABLE UNIT	*constraints*	データベース・エンジンに渡される制約がリストされます。結合で制約値が事前にわからない場合などは、制約値が NULL と表示されます。
TABLE UNIT	SCAN TABLE	行の検索に全表スキャンが使用されます。
TABLE UNIT	SCAN *indexname*	行の検索に索引 *indexname* が使用されます。選択されたすべての列が索引から見つかる場合は、表全体をスキャンするよりも索引をスキャンする方が処理が速い場合があります。これは、索引の方がディスク・ブロック数が少ないためです。
TABLE UNIT	PRIMARY KEY	行の検索に主キーが使用されます。主キーの属性には制限的な制約があるために表全体がスキャンされないという点で、これは SCAN と異なります。
TABLE UNIT	INDEX *indexname*	行の検索に索引 indexname が使用されます。一致する索引行ごとに、実際のデータ行が個別にフェッチされます。
TABLE UNIT	INDEX ONLY *indexname*	行の検索に索引 *indexname* が使用されます。選択された列はすべて索引内にあるため、実際のデータ行が表から読み取る方法で個別にフェッチされることはありません。
JOIN UNIT	MERGE JOIN	マージ結合を使用して表が結合されます。
JOIN UNIT	3-MERGE JOIN	3 マージ結合を使用して表が結合されます。
JOIN UNIT	LOOP JOIN	ループ結合を使用して表が結合されます。
ORDER UNIT	NO ORDERING REQUIRED	順序付けが不要です。solidDB から正しい順序で行がリトリーブされます。
ORDER UNIT	EXTERNAL SORT	外部ソーターを使用して行がソートされます。外部ソーターを使用可能にするには、構成ファイルの Sorter セクションに一時ディレクトリー名を指定する必要があります。

表 21. ユニットの INFO 列内のテキスト (続き)

ユニット・タイプ	INFO 列のテキスト	説明
ORDER UNIT	FIELD *n* USED AS PARTIAL ORDER	別個の結果セットを得るために、内部ソーター (インメモリー・ソーター) がソートに使用され、solidDB からリトリーブした行が列番号 *n* で部分的にソートされます。部分的に順序を付けることで、内部ソーターはデータを何度も通過する必要がなくなります。
ORDER UNIT	*n* FIELDS USED FOR PARTIAL SORT	内部ソーター (インメモリー・ソーター) がソートに使用され、solidDB からリトリーブした行が *n* 個のフィールドで部分的にソートされます。部分的に順序を付けることで、内部ソーターはデータを何度も通過する必要がなくなります。
ORDER UNIT	NO PARTIAL SORT	内部ソーターがソートに使用されます。行は solidDB からランダムな順序でリトリーブされてソーターに渡されます。
UNION UNIT	MERGE JOIN	マージ結合を使用して表が結合されます。
UNION UNIT	3-MERGE JOIN	3 マージ結合を使用して表が結合されます。
UNION UNIT	LOOP JOIN	ループ結合を使用して表が結合されます。
INTERSECT UNIT	MERGE JOIN	マージ結合を使用して表が結合されます。
INTERSECT UNIT	3-MERGE JOIN	3 マージ結合を使用して表が結合されます。
INTERSECT UNIT	LOOP JOIN	ループ結合を使用して表が結合されます。
EXCEPT UNIT	MERGE JOIN	マージ結合を使用して表が結合されます。
EXCEPT UNIT	3-MERGE JOIN	3 マージ結合を使用して表が結合されます。
EXCEPT UNIT	LOOP JOIN	ループ結合を使用して表が結合されます。

例 1

```
EXPLAIN PLAN FOR SELECT * FROM TENKTUP1 WHERE
UNIQUE2_NI BETWEEN 0 AND 99;
```

表 22. EXPLAIN PLAN FOR の例 1

ID	UNIT_ID	PAR_ID	JOIN_PATH	UNIT_TYPE	INFO
1	2	1	3	JOIN UNIT	
2	3	2	0	TABLE UNIT	TENKTUP1

表 22. EXPLAIN PLAN FOR の例 1 (続き)

ID	UNIT_ID	PAR_ID	JOIN_PATH	UNIT_TYPE	INFO
3	3	2	0		FULL SCAN
4	3	2	0		UNIQUE2_NI <= 99
5	3	2	0		UNIQUE2_NI >= 0
6	3	2	0		

実行グラフ

JOIN UNIT 2 は TABLE UNIT 3 から入力を取得します。

表 TENKTUP1 の TABLE UNIT 3 は、制約 UNIQUE2_NI <= 99 および UNIQUE2_NI >= 0 を使用して全表スキャンを実行します。

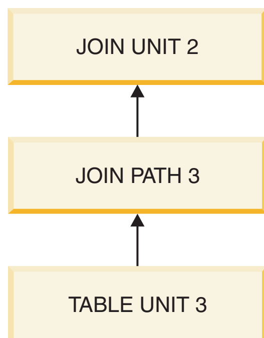


図 4. 実行グラフ 1

例 2

```
EXPLAIN PLAN FOR SELECT * FROM TENKTUP1, TENKTUP2
WHERE TENKTUP1.UNIQUE2 > 4000 AND TENKTUP1.UNIQUE2 < 4500
AND TENKTUP1.UNIQUE2 = TENKTUP2.UNIQUE2;
```

表 23. EXPLAIN PLAN FOR の例 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 REQUIRED
4	8	9	0	TABLE UNIT	TENKTUP2
5	8	9	0		PRIMARY KEY

表 23. *EXPLAIN PLAN FOR の例 2 (続き)*

ID	UNIT_ID	PAR_ID	JOIN_PATH	UNIT_TYPE	INFO
6	8	9	0		UNIQUE2 < 4500
7	8	9	0		UNIQUE2 > 4000
8	8	9	0		
9	10	6	0	ORDER UNIT	NO ORDERING REQUIRED
10	7	10	0	TABLE UNIT	TENKTUP1
11	7	10	0		PRIMARY KEY
12	7	10	0		UNIQUE2 < 4500
13	7	10	0		UNIQUE2 > 4000
14	7	10	0		

実行グラフ

JOIN UNIT 6 では、ORDER UNIT 9 および 10 からの入力が、マージ結合アルゴリズムを使用して結合されます。

ORDER UNIT 9 は TABLE UNIT 8 からの入力に順序を付けます。データは正しい順序でリトリーブされるため、実際には順序付けは必要ありません。

ORDER UNIT 10 は TABLE UNIT 7 からの入力に順序を付けます。データは正しい順序でリトリーブされるため、実際には順序付けは必要ありません。

TABLE UNIT 8 では、表 TENKTUP2 から主キーを使用して行がフェッチされます。行の選択には、制約 UNIQUE2 < 4500 および UNIQUE2 > 4000 が使用されます。

TABLE UNIT 7 では、表 TENKTUP1 から主キーを使用して行がフェッチされます。行の選択には、制約 UNIQUE2 < 4500 および UNIQUE2 > 4000 が使用されます。

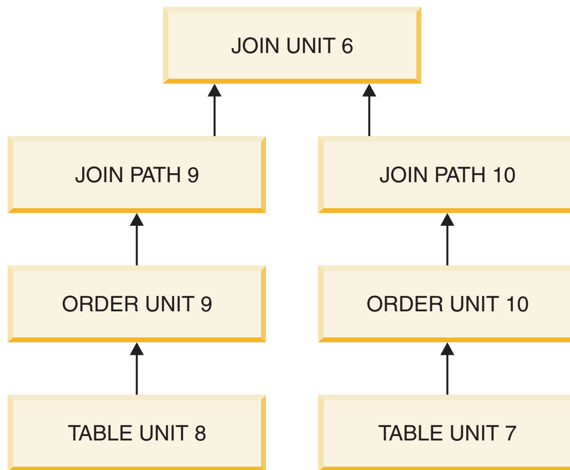


図 5. 実行グラフ 2

ストアード・プロシージャーおよびトリガーのトレース機能

ストアード・プロシージャーまたはトリガーをデバッグする際に、「トレース」コマンドを追加して、コードのどの部分が実行中であるかを確認できます。あるいは、プロシージャーまたはトリガー内のすべてのステートメントをトレースすることができます。以下の 2 つのセクションでは、その実行方法について説明します。

ユーザー定義可能な、プロシージャー・コードからのトレース出力

以下のコマンドを使用して、ストアード・プロシージャーまたはトリガーの内部から、「トレース」出力を `soltrace.out` ファイルへ送信できます。

```
WRITETRACE (entry VARCHAR)
```

以下のコマンドを使用して、出力をオンまたはオフにすることができます。

```
ADMIN COMMAND 'usertrace { on | off }
user username { procedure | trigger | table } entity_name'
```

「entity_name」は、トレースをオンまたはオフにするプロシージャー、トリガー、または表の名前です。キーワード「table」を指定した場合、その表のすべてのトリガーがトレースされます。

指定したプロシージャー、指定したトリガー、または指定した表のすべてのトリガーについて、トレースをオン (またはオフ) にすることができます。

トレースは、指定されたユーザーがプロシージャーまたはトリガーを呼び出したときにのみアクティブ化されます。これは、例えば、拡張レプリケーション・マスターで伝搬されたプロシージャー呼び出しをトレースするときに役に立ちます。

トレースをオンにすると、トレースをオンに切り替えた接続からの呼び出しだけでなく、そのユーザーによるすべてのプロシージャー/トリガー呼び出しでトレースがオンになります。同じユーザー名を使用している複数の接続がある場合は、それら

すべての接続におけるすべての呼び出しがトレースされます。さらに、レプリカで実行された呼び出しだけでなく、マスターに伝搬された (マスターで実行された) 呼び出しでもトレースが行われます。

プロシージャー実行トレース

ストアード・プロシージャーまたはトリガーのすべてのステートメントをトレースする必要がある場合、すべての SQL ステートメントに WRITETRACE ステートメントを書き込む必要はありません。指定されたストアード・プロシージャーまたはトリガー内のすべてのステートメントをトレースする「PROCTRACE」をオンにするだけです。USERTRACE の場合と同様に、指定されたプロシージャー、指定されたトリガー、または特定の表に関連付けられているすべてのトリガーに対して proctrace をオンにできます。構文は以下のとおりです。

```
ADMIN COMMAND 'proctrace { on | off }
user username { procedure | trigger | table } entity_name'
```

「entity_name」は、トレースをオンまたはオフにするプロシージャー、トリガー、または表の名前です。

トレースは、指定されたユーザーがプロシージャー/トリガーを呼び出したときにのみアクティブになります。これは、例えば、拡張レプリケーション・マスターで伝搬されたプロシージャー呼び出しをトレースするときに役に立ちます。

トレースをオンにすると、トレースをオンに切り替えた接続からの呼び出しだけでなく、そのユーザーによるすべてのプロシージャー/トリガー呼び出しでトレースがオンになります。同じユーザー名を使用している複数の接続がある場合は、それらすべての接続におけるすべての呼び出しがトレースされます。さらに、レプリカで実行された呼び出しだけでなく、マスターに伝搬された (マスターで実行された) 呼び出しでもトレースが行われます。

キーワード「table」を指定した場合、その表のすべてのトリガーがトレースされます。

例:

```
"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);
```

この例の出力は以下のようになります。

```
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 ----
```

START AFTER COMMIT ステートメントのパフォーマンスの測定および向上

START AFTER COMMIT ステートメントのパフォーマンスのチューニング

バックグラウンド・タスクは、SSC-API および ADMIN COMMAND を使用して制御できます (詳しくは、「*IBM solidDB 共有メモリー・アクセスおよびリンク・ライブラリー・アクセス・ユーザー・ガイド*」を参照してください)。タスク・タイプ SSC_TASK_BACKGROUND は、START AFTER COMMIT で開始されたステートメントを実行するタスクに使用されます。このタスク・タイプは、優先順位を変更したり、中断したりすることができます。

このタイプのタスクは複数存在する可能性がありますが、個別に制御できないので注意してください。つまり、SSC_TASK_BACKGROUND に対して SSCSuspendTaskClass を呼び出すと、すべてのバックグラウンド・タスクが中断されます。

START AFTER COMMIT ステートメントでの障害の分析

同時に存在できる非コミット START AFTER COMMIT ステートメントの数には、制限があります。(「非コミット」とは、START AFTER COMMIT ステートメントが実行されたトランザクションが、まだコミットされていないことを意味します。この時点で、START AFTER COMMIT ステートメントの本体、つまりプロシージャー呼び出しは、実行を開始さえしていません。) 最大値に到達した場合、次の START AFTER COMMIT が発行された時点でエラーが返されます。最大数は、MaxStartStatements という名前のパラメーターを使用して、solid.ini 内で構成できます (詳しくは、「*IBM solidDB 管理者ガイド*」で、このパラメーターの説明を参照してください)。

ステートメントを開始できない場合、その理由はシステム表 SYS_BACKGROUNDJOB_INFO にログとして記録されます。この表には、失敗した START AFTER COMMIT ステートメントのみがログとして記録されます。この表について詳しくは、365 ページの『SYS_BACKGROUNDJOB_INFO』を参照してください。

ユーザーは、SQL SELECT ステートメントを使用するか、システム・プロシージャー SYS_GETBACKGROUNDJOB_INFO を呼び出すことによって、表 SYS_BACKGROUNDJOB_INFO から情報をリトリーブすることができます。ストアード・プロシージャー SYS_GETBACKGROUNDJOB_INFO は、START AFTER COMMIT ステートメントで指定されたジョブ ID に一致する行を返します。SYS_GETBACKGROUNDJOB_INFO について詳しくは、420 ページの『SYS_GETBACKGROUNDJOB_INFO』を参照してください。

ステートメントを開始できなかったときに通知を受けたい場合は、システム・イベント SYS_EVENT_SACFAILED を待つことができます。このイベントについて詳しくは、 424 ページの『各種イベント』の説明を参照してください。アプリケーションはこのイベントを待ち、ジョブ ID を使用して、システム表 SYS_BACKGROUNDJOB_INFO からエラー・メッセージをリトリーブできます。

7 パフォーマンスのチューニング

この章では、solidDB のパフォーマンスを向上させるために使用する手法を説明します。この章で説明するトピックは、以下のとおりです。

- SQL ステートメントとアプリケーションのチューニング
- 単一表 SQL 照会の最適化
- 索引を使用した照会パフォーマンスの向上
- イベント待ち
- バッチ挿入および更新の最適化
- パフォーマンスに関するオプティマイザーのヒントの使用
- パフォーマンス低下の診断

拡張レプリケーションのデータ同期の最適化に関するヒントについては、「*IBM solidDB 拡張レプリケーション・ユーザー・ガイド*」を参照してください。

SQL ステートメントとアプリケーションのチューニング

一般に、SQL ステートメントをチューニングすることは、特に複雑な照会が関係するアプリケーションで、データベースのパフォーマンスを高める最も効率の良い手段です。

アプリケーションのチューニングは、必ず RDBMS をチューニングする*前に* 行ってください。理由は以下のとおりです。

- アプリケーション設計時に、SQL ステートメントと処理対象のデータを制御できます。
- これから使用する RDBMS の内部の仕組みに詳しくなくてもパフォーマンスを高めることができます。
- アプリケーションが適切にチューニングされていないと、RDBMS が適切にチューニングされていてもそこでアプリケーションが適切に実行されません。

アプリケーションで処理されるデータ、使用される SQL ステートメント、およびアプリケーションがデータに対して実行する操作を把握しておく必要があります。例えば、不要な節や述部を使用しない単純な SELECT ステートメントにすると、照会のパフォーマンスが向上します。

アプリケーション・パフォーマンスの評価

アプリケーションでパフォーマンスが不足している領域を切り分けるために、solidDB では以下の診断ツールでデータベースのパフォーマンスを監視できます。

- SQL 情報機能
- EXPLAIN PLAN FOR ステートメント

これらのツールは、アプリケーションのチューニングおよびアプリケーション内の非効率的な SQL ステートメントの特定に役立ちます。ツールの使い方について詳しくは、 147 ページの『6 章 診断およびトラブルシューティング』を参照してください。

また、以下のコマンドを使用することでパフォーマンスの評価に役立つ情報を得られます。

- ADMIN COMMAND 'status'

  このコマンドは、サーバーから統計情報を返します。詳しくは、「*IBM solidDB 管理者ガイド*」のこのコマンドに関する説明を参照してください。

- ADMIN COMMAND 'perfmon'

  このコマンドは、サーバーから詳細なパフォーマンス上の統計を返します。詳しくは、「*IBM solidDB 管理者ガイド*」の perfmon に関する説明および『*DBMS モニター (Perfmon) の詳細*』を参照してください。

- ADMIN COMMAND 'trace'

  このコマンドは、SQL ステートメントおよびネットワーク通信のトレースをオンに切り替えます。完全な構文については、 177 ページの『ADMIN COMMAND』のトレース・オプションの構文を参照してください。

ストアード・プロシージャー言語の使用

ストアード・プロシージャーを使用すると、一部の操作を 2 つの方法で高速化できます。

- ストアード・プロシージャー内のステートメントは構文解析され、1 回だけコンパイルされてからコンパイル済み形式で保管されます。ストアード・プロシージャーの外部にあるステートメントは、実行されるたびに再構文解析され、コンパイルされます。このため、ステートメントをストアード・プロシージャーの中に置くと、ステートメントが複数回実行される場合、オーバーヘッド (構文解析とコンパイル) が少なくなります。
- 単一のストアード・プロシージャーの内部に複数のステートメントがある場合、ストアード・プロシージャーを 1 回だけ呼び出すことは、各ステートメントを個別にクライアントからサーバーへ渡すより、ネットワーク上の「行程」が少なくて済みます。

単一表 SQL 照会の最適化

solidDB は、特定のタイプの単一表 SQL 照会でパフォーマンスを向上させる単純 SQL 最適化機能を備えています。パフォーマンスの向上は、SELECT、DELETE、および UPDATE ステートメントで見られます。この機能は、INSERT ステートメントには適用されません。

単純 SQL 最適化は、`solid.ini` ファイルの [SQL] セクションで、**SimpleSQLOpt** パラメーターによって使用可能/使用不可にします。デフォルトでは、この機能はオンにされ、**SimpleSQLOpt** パラメーターは `solid.ini` ファイル内に現れません。この機能を使用不可にするには、以下の行を `solid.ini` ファイルに追加する必要があります。

```
[SQL]
SimpleSQLOpt=No
```

これらの行をファイルに追加した後は、**SimpleSQLOpt=Yes** を指定するか、上記のパラメーターを [SQL] セクションから除去することによって、いつでもこの機能を使用可能にすることができます。いつものように、solid.ini ファイルに加えた変更は、サーバーを再始動するまで有効にならないことに注意してください。

単純 SQL 最適化をオンにした場合、solidDB は以下の条件を満たす単一表 SQL 照会を自動的に最適化します。

- ステートメントが単一表だけにアクセスする。
- ステートメントにビュー、副照会、UNION、INTERSECT などが含まれていない。
- ステートメントが ROWNUM を使用しない。
- ステートメントが、シーケンス番号をリトリーブするために使用される solidDB シーケンス・オブジェクトを使用していない。

他の最適化の技法と同様に、単純 SQL 最適化機能はほとんどの照会を高速化しますが、少数のタイプの照会ではパフォーマンスが低下します。単純 SQL 最適化を使用しているときに特定の照会で処理速度の低下が見られる場合は、この機能をオフにできます。

索引を使用した照会パフォーマンスの向上

索引を使用して、照会のパフォーマンスを向上させることができます。 WHERE 節の中で索引付きの列を参照する照会では、索引を使用できます。照会で索引付きの列だけを選択する場合は、照会で索引付きの列を、表からでなく、索引から直接読み取ることができます。

照会の SELECT リスト内にあるすべてのフィールドが 1 つの索引に入っている場合、solidDB オプティマイザーは完全なレコードを読み取るために追加の参照を行わず、単にその索引を使用することができます。同様に、WHERE 節のすべてのフィールドが 1 つの索引内にある場合、オプティマイザーはその索引を使用できます。その索引内の情報が、レコードを WHERE 節に適格でないことを証明するのに十分なものである場合、オプティマイザーは完全なレコードの参照を回避できます。

例えば、次のような 2 つ以上の列を参照する WHERE 節があるとします。

```
WHERE col1 = x AND col2 >= a AND col2 <=b
```

さらに、col1 と col2 の両方を含んでいる索引があり、その索引が col1 か col2 をそのキーの先行列として持っているとします。例えば、col2 + col3 + col1 に索引がある場合、この索引は両方の列を含んでおり、そのうちの 1 つ (col2) がキーの先行列になっています。ユーザーが次のような照会を行ったとします。

```
SELECT col1, col4
    FROM table1
    WHERE col1 = x AND col2 >= a AND col2 <=b;
```

この場合、検索基準が満たされた場合を除き、完全なレコードを参照する必要はありません。結局、検索基準が満たされなかった場合は、col4 の値に関心がないので、完全なレコードを参照する必要はありません。

表に主キーが存在する場合、solidDB は主キーの値の順序で、ディスク上の行を順序付けます。行は物理的に主キーの順序になっているため、主キー自体が索引として機能し、索引に適用される最適化のヒントは主キーにも当てはまります。

表にユーザー指定の主キーがない場合、行は ROWID を使用して順序付けられます。ROWID は各行が挿入されたときに割り当てられ、各レコードはその前に挿入されたレコードより大きい ROWID を取得します。このため、ユーザー指定の主キーがない表では、レコードはそれらの行が挿入された順序で保管されます。主キーについて詳しくは、 116 ページの『主キー索引』をお読みください。

行値コンストラクター制約を持つ検索は、索引が使用可能な場合、索引を使用するよう最適化されます。効率を良くするため、solidDB は索引を使用して、(A, B, C) >= (1, 2, 3) という形式の行値コンストラクター制約を解決します。ただし、演算子は <、<=、>=、および > のいずれかとすることができます。(サーバーは、演算子 =、!=、または <> を含んでいる行値コンストラクター制約の解決に索引を使用しません。サーバーは索引を使用して、=、!=、または <> を使用するその他のタイプの制約を解決できます。) 行値コンストラクターについて詳しくは、 22 ページの『行値コンストラクター』を参照してください。

索引を使用すると、表から選択する行のパーセンテージが小さい照会のパフォーマンスが向上します。選択する行が表全体の 15% 未満の照会では、索引の使用を考慮してください。

全表スキャン

照会で索引を使用できない場合、solidDB は全表スキャンを実施してその照会を実行する必要があります。この処理では、表の全行が順次読み取られます。行ごとにその行が照会の WHERE 節の基準を満たしているかどうかが検査されます。1 つの行を検索する場合は、全表スキャンよりも索引照会を使用する方がはるかに高速です。一方、照会で選択される表の行数が 15% を超える場合は、索引照会よりも全表スキャンの方が処理が速い場合があります。

すべての照会を EXPLAIN PLAN ステートメントで確認する必要があります。(これを行う場合は実際のデータを使用してください。最適なプランは実際のデータ量とそのデータの特性によって決まるからです。) EXPLAIN PLAN ステートメントの出力から索引が実際に使用されるかどうかがわかり、必要に応じて照会または索引を再実行できます。全表スキャンでは、SELECT 照会の応答が遅くなり、ディスク・アクティビティーが過剰になることがよくあります。パフォーマンス低下の問題を診断するには、「*IBM solidDB 管理者ガイド*」に説明されているように `ADMIN COMMAND 'perfmon'` を使用してファイル操作に関する統計を要求できます。(『DBMS モニター (Perfmon) の詳細』という表題のセクションを参照してください。)

全表スキャンでは、表内のすべてのブロックが読み取られます。ブロックごとに、ブロックに格納されているすべての行が読み取られます。索引照会では、行がその格納先のブロックとは関係なく索引での順序に従って読み取られます。1 つのブロ

ックに複数の選択行が含まれている場合は、そのブロックが複数回読み取られる場合があります。したがって結果セットが比較的大きい場合は、索引照会よりも全表スキャンの方が入出力の回数が少なくなることがあります。

連結索引

1 つの索引を複数の列から形成できます。そのような索引を連結索引と呼びます。可能な場合は、連結索引を使用することを推奨します。

SQL ステートメントで連結索引が使用されているかどうかは、SQL ステートメントの WHERE 節に含まれている列によって判別されます。照会は、WHERE 節内で索引の先行位置を参照する場合、連結索引を使用できます。索引の先行位置とは、CREATE INDEX ステートメントで指定された最初の列 (単数または複数) を指しています。

例:

```
CREATE INDEX job_sal_deptno ON emp(job, sal, deptno);
```

この索引は、以下の照会で使用できます。

```
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';
```

以下の照会には、WHERE 節に索引の最初の列が含まれていないため、索引を使用できません。

```
SELECT * FROM emp WHERE sal = 6000;
```

索引付けする列の選択

以下のリストは、索引付けする列を選択する際のガイドラインを示しています。

- 索引は、WHERE 節の中で使用することが多い列について作成してください。
- 索引は、表を結合するために使用することが多い列について作成してください。
- 索引は、ORDER BY 節の中で使用することが多い列について作成してください。
- 索引は、表の中に同じ値または固有値がほとんどない列について作成してください。
- 小さな表 (少数のブロックしか使用しない表) には索引を作成しないでください。表全体をスキャンした方が、索引付きの照会よりも高速な場合があるからです。
- できれば、行を最も適切な順序で並べる主キーを選択してください。
- 連結索引の中の 1 つの列だけが WHERE 節の中で頻繁に使用される場合は、その列を CREATE INDEX ステートメントの最初に配置してください。
- 連結索引の中の複数の列が WHERE 節の中で頻繁に使用される場合は、最も選択効率がよい列を CREATE INDEX ステートメントの最初に配置してください。

イベント待ち

多くのプログラムで、タスクを実行する前に、特定の条件が発生するのを待たなければならない場合があります。場合によっては、while ループを使用して、条件が発生したかどうかを検査できます。solidDB が提供するイベントを使用すると、条件を待つためにループ内を回って CPU 時間を浪費するのを避けられる場合もあります。

あるイベントを 1 つ (以上) のクライアントまたはスレッドで待ち、別のクライアントまたはスレッドでそのイベントを通知できます。例えば、いくつかのスレッドで、あるセンサーが新しいデータの断片を取得するのを待つこともできます。別の (そのセンサーを処理する) スレッドでは、データが使用可能であることを示すイベントを通知できます。イベントについて詳しくは、 97 ページの『イベントの使用』、および 177 ページの『付録 B. solidDB SQL 構文』の 201 ページの『CREATE EVENT』を含むさまざまなセクションを参照してください。

バッチ挿入および更新の最適化

バッチ挿入を主キー順で実行できるデータベース・スキーマを設計するよう、強く推奨します。データベース・ファイル内のデータは、表の主キーによって定義された順序で物理的に保管されます。主キーが定義されていない場合、データはデータベースに書き込まれた順序でデータベース・ファイルに保管されます。データベース操作 (つまり、読み取りと書き込み) は、常にページ・レベルでデータにアクセスします。データベースのデフォルトのページ・サイズは 8 KB です。

バッチ書き込み操作が、主キーをサポートする順序で行われた場合、サーバーのキャッシュ・アルゴリズムは、データベース・ファイルの書き込み操作をグループにまとめることができます。これにより、多数の行が 1 回の物理的なディスク I/O 操作でディスクに書き込まれます。最悪のケースでは、挿入順序が主キーの順序と異なっている場合、1 回の挿入または削除操作ごとに、1 行しか変更されていないデータベース・ページを再書き込みする必要があります。

これらの理由から、バッチ書き込み操作の表に、バッチ書き込み操作のアクセス順序に一致する主キーを設けることには意味があります。このタイプのデータベース・スキーマにより、操作のパフォーマンスに大きな差が出ることがあります。

例えば、以下の種類の表があるとします。

```
CREATE TABLE USAGE_EVENT (
 EVENT_ID INTEGER NOT NULL PRIMARY KEY,
 DEVICE_ID INTEGER NOT NULL,
 EVENT_DATA VARCHAR NOT NULL);
```

この表で、EVENT_ID はシーケンス番号です。挿入操作と削除操作は EVENT_ID 列によって指定された順序で行われ、最大の効率が得られます。

この同じ表に対するバッチ書き込み操作のパフォーマンスは、主キーの最初の列が DEVICE_ID で、データが EVENT_ID の順序でデータベースに書き込まれるとすると、大幅に低下することに注意してください。そのようなシナリオでは、バッチ書き込み操作を完了するために必要なファイル入出力操作の数は、表のサイズが大きくなると増大します。

バッチ挿入および更新の高速化

solidDB に対する大規模なバッチ挿入およびバッチ更新の速度を最適化することができます。高速化のためのガイドラインを以下に示します。

1. AUTOCOMMIT モードがオフに設定された状態でアプリケーションを実行していることを確認します。

   solidDB ODBC ドライバーのデフォルトの設定は AUTOCOMMIT です。これは ODBC の仕様に従った標準の設定です。アプリケーションで AUTOCOMMIT をオフに設定するには、以下の例のように SQLSetConnectOption 関数を呼び出します。

   ```
   rc = SQLSetConnectOption
   (hdbc, SQL_AUTOCOMMIT, SQL_AUTOCOMMIT_OFF);
   ```
2. 大きなトランザクションは使用しないでください。初期トランザクション・サイズには 500 行が推奨されています。トランザクション・サイズの最適値はアプリケーションによって異なるため、実際に試すことが必要です。
3. バッチ挿入を高速化するために、ロギングをオフにすることができます。ただし、システム障害発生時のデータ損失のリスクが高くなります。環境によっては、このトレードオフが許容されることもあります。

上記ガイドラインの 1 と 2 は、バッチ挿入を高速化するための最も重要なアクションです。実際の挿入速度は、ハードウェア、1 行あたりのデータ量、および表の既存の索引にも左右されます。

オプティマイザーのヒントの使用

データ、ユーザー照会、およびデータベースの状態はさまざまであるため、SQL オプティマイザーは可能な最良の実行プランを常に選択できるとは限りません。例えば、ユーザーは (オプティマイザーとは違って) データが既にソートされていることが分かっている場合、効率を上げるためにマージ結合を強制できます。

あるいは、照会内の特定の述部が原因で、オプティマイザーでは解消できないパフォーマンス上の問題が起きるときがあります。ユーザーには、オプティマイザーが使用している索引が最適でないことが分かる場合もあります。そのような場合は、より高速な結果を生成する索引を使用するよう、オプティマイザーに強制することもできます。

オプティマイザーのヒントを使用すると、ユーザーのパフォーマンス上のニーズに合わせて、応答時間に関する制御を改善できます。ユーザーは照会の中で、オプティマイザーに対するディレクティブまたはヒントを指定でき、オプティマイザーはそれを使用して、照会実行プランを決定します。ヒントは、SQL-92 からの疑似コメント構文によって検出されます。

ヒントは、以下のものに使用可能です。

- マージ結合またはネストしたループ結合の選択
- from リストで指定された固定の結合順序の使用
- 内部または外部ソートの選択
- 特定の索引の選択
- 索引スキャンでなく表スキャンの選択

- グループ化の前または後でのソートの選択

ヒント (複数可) は、SQL ステートメント内に静的ストリングとして、SELECT、UPDATE、または DELETE キーワードの直後に配置できます。ヒントを INSERT キーワードの後に置くことは許されません。

オプティマイザーのヒントでの表名の解決は、SQL ステートメント内のすべての表名の場合と同じです。したがって、照会内に表の別名がある場合は、オプティマイザーのヒントの中で、表名でなく別名を使用する必要があります。以下に例を示します。

```
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;
```

別名を指定する必要があるときに表名を指定した場合は、以下のエラー・メッセージを受け取ります。

```
102: Unused optimizer hint.
```

別名を使用しておらず、別のスキーマや別のカタログの表を使用している場合は、必ず、ヒントの中で表名の前にスキーマ名やカタログ名を付けてください。以下に例を示します。

```
SELECT
 --(* vendor(SOLID), product(Engine), option(hint)
 -- FULL SCAN sally_schema.employee_table *)--
 emp_id, employee_name
FROM sally_schema.employee_table;
```

ヒントの指定にエラーがある場合は、その SQL ステートメント全体がエラー・メッセージと共に失敗します。

ヒントを使用可能または使用不可にするには、solid.ini 内で以下の構成パラメーターを使用します。

```
[Hints]
EnableHints=YES | NO
```

デフォルトは YES に設定されます。

オプティマイザーのヒントについて詳しくは、可能なヒントと例の説明も含め、261 ページの『HINT』を参照してください。

パフォーマンス低下の診断

solidDB には、パフォーマンスの低下を招くさまざまな領域があります。パフォーマンス上の問題を解決するには、根本原因を特定する必要があります。以下の表では、一般的なパフォーマンス低下の症状とその考えられる原因を挙げ、解決策が記載されているこの章のセクションを示します。

表 24. パフォーマンス低下の診断

症状	診断	解決策
1 回の照会で応答時間が遅い。データベースへの他の並行アクセスが影響を受けている。ディスクがビジーの可能性がある。	• 照会で索引が効率よく使用されていません。 • オプティマイザーによる決定が最適ではありません。 • 外部ソートが定義されておらず、大規模な内部ソートによってディスクへの過剰なスワッピングが発生しています。	索引定義が不足している場合は、新しい索引を作成するか、遅い照会に対する索引付けの要件に合わせて既存の索引を変更します。詳しくは、 161 ページの『索引を使用した照会パフォーマンスの向上』を参照してください。 遅い照会に対して EXPLAIN PLAN FOR ステートメントを実行し、照会オプティマイザーが索引を使用しているかどうかを検証します。詳しくは、 148 ページの『EXPLAIN PLAN FOR ステートメント』を参照してください。 オプティマイザーが最適な照会実行プランを選択していない場合は、オプティマイザー・ヒントを使用してオプティマイザーの決定をオーバーライドします。詳しくは、 165 ページの『オプティマイザーのヒントの使用』を参照してください。 外部ソーターを使用できるように、 Sorter.TmpDir 構成パラメーターを定義します。詳しくは、「*IBM solidDB 管理者ガイド*」の『TmpDir_[1...N]』の説明を参照してください。
すべての照会で応答時間が遅い。同時ユーザー数が増えると、パフォーマンスが線形より大きく低下する。全ユーザーの接続を解除して再接続してもパフォーマンスが向上しない。	キャッシュ・サイズが十分ではありません。	キャッシュ・サイズを増やしてください。キャッシュを少なくとも同時ユーザーごとに 0.5 MB ずつ割り振るか、またはデータベース・サイズの 2% から 5% 割り振ってください。詳しくは、「*IBM solidDB 管理者ガイド*」の『データベース・キャッシュ・サイズの定義』というセクションを参照してください。
すべての照会と書き込み操作で応答時間が遅い。全ユーザーの接続を解除して再接続しても、一時的にしかパフォーマンスが向上しない。ディスクが非常にビジーである。	Bonsai ツリーが大きすぎてキャッシュに収まりません。	意図した時間より長く実行されているトランザクションがないかを確認します。すべてのトランザクション (読み取り専用トランザクションも含めて) が適時にコミットされていることを検証します。詳しくは、「*IBM solidDB 管理者ガイド*」の『トランザクションのコミットによる Bonsai ツリーのサイズ縮小』を参照してください。

表 24. パフォーマンス低下の診断 (続き)

症状	診断	解決策
データベース・サイズが増大するにつれてバッチ書き込み操作のパフォーマンスが低下する。ディスク I/O の量が多すぎる。	• データをデータベースにコミットするバッチの単位が小さすぎます。 • 表の主キーがサポートしていない順序でデータがディスクに書き込まれています。	自動コミットがオフに切り替えられていることを確認し、書き込み操作がトランザクションあたり 100 行以上のバッチ単位でコミットされるようにします。 書き込み操作が主キーの順序で行われるように、主キーまたはバッチ書き込みプロセスを変更します。詳しくは、 164 ページの『バッチ挿入および更新の最適化』を参照してください。
サーバー・プロセスのフットプリントが大きくなりすぎてオペレーティング・システムでスワップが発生する。ディスクが非常にビジーである。ADMIN COMMAND 'report' の出力に示される、現在アクティブなステートメントのリストが長い。	SQL ステートメントが使用後に閉じておらず、ドロップもされていません。	クライアント・アプリケーションで使用されなくなったステートメントが適時に閉じられ、ドロップされるようにします。

付録 A. データ型

この付録の各表では、サポートされているデータ型をカテゴリー別にリストします。それぞれの表で以下の略語が使用されます。

表 25. サポートされているデータ型

略語	説明
DEFLEN	定義されている列の長さ。例えば CHAR(24) の場合は精度と長さが 24 です。
DEFPREC	定義されている精度。例えば NUMERIC(10,3) の場合は 10 です。
DEFSCALE	定義されている位取り。例えば NUMERIC(10,3)の場合は 3 です。
MAXLEN	列の最大長
N/A	該当なし

文字データ型

表 26. 文字データ型

データ型	サイズ	精度	位取り	長さ	表示サイズ
CHARACTER CHAR	2 G - 1* (2147483647)	DEFLEN	N/A	DEFLEN	DEFLEN
WCHAR NATIONAL CHARACTER NATIONAL CHAR NCHAR	2 G - 1* (2147483647)	DEFLEN	N/A	DEFLEN	DEFLEN
VARCHAR CHARACTER VARYING CHAR VARYING	2 G - 1** (2147483647)	DEFLEN	N/A	DEFLEN	DEFLEN

表 26. 文字データ型 (続き)

データ型	サイズ	精度	位取り	長さ	表示サイズ
WVARCHAR NATIONAL VARCHAR NCHAR VARYING NVARCHAR	2 G - 1** (2147483647)	DEFLEN	N/A	DEFLEN	DEFLEN
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
* デフォルトは 1 です。 ** デフォルトは 254 です。					

数値データ型

表 27. 数値データ型

データ型	サイズ	精度	位取り	長さ	表示サイズ
TINYINT	[-128, 255]	3	0	1 (バイト)	4 (符号付き) 3 (符号なし)
SMALLINT	[-32768, 65535]	5	0	2 (バイト)	6 (符号付き) 5 (符号なし)
INTEGER INT	$[-2^{31}, 2^{31}-1]$	10	0	4 (バイト)	11 (符号付き) 10 (符号なし)
BIGINT	$[-2^{63}, 2^{63}-1]$	19	0	8 (バイト)	20 (符号付き)

表 27. 数値データ型 (続き)

データ型	サイズ	精度	位取り	長さ	表示サイズ
REAL	正数: 1.175494351e-38 から 1.7014117e+38 負数: -1.7014117e+38 から -1.175494351e-38 値ゼロ (0) も、このデータ型で使用できます。	7	N/A	4 (バイト)	13
FLOAT	正数: 2.2250738585072014e-308 から 8.98846567431157854e+307 負数: -8.98846567431157854e+307 から -2.2250738585072014e-308 値ゼロ (0) も、このデータ型で使用できます。	15	N/A	8 (バイト)	22
DOUBLE PRECISION	正数: 2.2250738585072014e-308 から 8.98846567431157854e+307 負数: -8.98846567431157854e+307 から -2.2250738585072014e-308 値ゼロ (0) も、このデータ型で使用できます。	15	N/A	8 (バイト)	22
DECIMAL*	±1.0e254	DEFPREC 最大 52 デフォルト 52	DEFSCALE デフォルト 0	2 から 27 (バイト)	可変

表 27. 数値データ型 (続き)

データ型	サイズ	精度	位取り	長さ	表示サイズ
NUMERIC	±1.0e254	DEFPREC 最大 52 デフォルト 52	DEFSCALE デフォルト 0	2 から 27 (バイト)	可変
* DECIMAL に精度も位取りも指定しなかった場合、値は精度が 52 で範囲が ±1.0e254 の (厳密な) 10 進浮動小数点数として表されます。					

注:

整数データ型 (TINYINT、SMALLINT、INT、および BIGINT) は、クライアント・プログラムによって符号付きまたは符号なしとして解釈される場合がありますが、solidDB では符号付き整数として保管され、順序付けられます。サーバーに、整数データ型を符号なしデータとして順序付けるよう指示する方法はありません。

注意:

BIGINT の有効数字は、約 19 桁です。これは、BIGINT を非整数データ型に格納したときに、最下位桁が失われる場合があることを示しています。それらの整数データ型は、FLOAT (有効数字は約 15 桁)、SMALLFLOAT (有効数字は約 7 桁)、DECIMAL (有効数字は約 16 桁) などです。

バイナリー・データ型

表 28. バイナリー・データ型

データ型	サイズ	精度	位取り	長さ	表示サイズ
BINARY	2 G*	DEFLEN	N/A	DEFLEN	DEFLEN x 2
VARBINARY	2 G**	DEFLEN	N/A	DEFLEN	DEFLEN x 2
LONG VARBINARY BLOB	2 G	MAXLEN	N/A	MAXLEN	MAXLEN x 2
* デフォルトは 1 です。 ** デフォルトは 254 です。					

ヒント:

BINARY、VARBINARY、および LONG VARBINARY フィールドに値を挿入するには、値を 16 進数として表現し、CAST 演算子を使用できます。以下に例を示します。

```
INSERT INTO table1 VALUES (CAST('FF00AA55' AS VARBINARY));
```

同様に、CAST() 式を WHERE 節の中で使用できます。

```
CREATE TABLE t1 (x VARBINARY);
INSERT INTO t1 (x) VALUES (CAST('000000A512' AS VARBINARY));
INSERT INTO t1 (x) VALUES (CAST('000000FF12' AS VARBINARY));

-- LIKE を使用して VARBINARY 値を比較するには
-- VARBINARY を VARCHAR にキャストします。
SELECT * FROM t1 WHERE CAST(x AS VARCHAR) LIKE '000000A5%';
SELECT * FROM t1 WHERE CAST(x AS VARCHAR) LIKE '000000A5__';

-- 注: 「LIKE」でなく「=」を使用したい場合は、
-- どちらかのオペランドをキャストできます。
SELECT * FROM t1 WHERE CAST(x AS VARCHAR) = '000000A512';
SELECT * FROM t1 WHERE x = CAST('000000A512' AS VARBINARY);
```

警告: この種の照会は LIKE 述部の索引付き検索を使用できず、多く場合、照会のパフォーマンスも貧弱な結果に終わります。

日付データ型

表 29. 日付データ型

データ型	サイズ	精度	位取り	長さ	表示サイズ
DATE	N/A	10*	N/A	6**	10*
* yyyy-mm-dd フォーマットの文字数 ** DATE_STRUCT 構造体のサイズ					

TIME データ型

表 30. TIME データ型

データ型	サイズ	精度	位取り	長さ	表示サイズ
TIME	N/A	8*	N/A	6**	8*
* hh:mm:ss 形式での文字の数 ** TIME_STRUCT 構造体のサイズ					

TIMESTAMP データ型

表 31. TIMESTAMP データ型

データ型	サイズ	精度	位取り	長さ	表示サイズ
TIMESTAMP	N/A	19*	9	16**	19/29***
* 「yyyy-mm-dd hh:mm:ss.fffffffff」形式での文字の数 ** TIMESTAMP_STRUCT 構造体のサイズ *** サイズは小数部を含めて 29					

最小限の非ゼロ数値

表 32. 最小限の非ゼロ数値

データ型	値
DOUBLE PRECISION	2.2250738585072014e-308
REAL	1.175494351e-38

BLOB および CLOB

solidDB では、最大 2147483647 (2G - 1) バイトの長さのバイナリー・データまたは文字データを保管できます。そのようなデータが特定の長さを超えた場合、そのデータは情報を保管するデータ型に応じて、BLOB (バイナリー・ラージ・オブジェクト) または CLOB (文字ラージ・オブジェクト) と呼ばれます。CLOB は「プレーン・テキスト」のみを格納し、以下のどのデータ型にも保管できます。

- CHAR、WCHAR
- VARCHAR、WVARCHAR
- LONG VARCHAR (標準の CLOB 型にマップされる)
- LONG WVARCHAR (標準の NCLOB 型にマップされる)

BLOB には、例えば、デジタル化した画像、ビデオ、オーディオ、定様式テキスト・ドキュメントなど、バイト列で表現できる任意のデータ型を保管できます。プレーン・テキストも保管できますが、プレーン・テキストは CLOB に保管した方が柔軟性が増します。

BLOB は、以下のどのデータ型にも保管できます。

- BINARY
- VARBINARY
- LONG VARBINARY (標準の BLOB 型にマップされる)

文字データはバイト列であるため、CHAR フィールドだけでなく、BINARY フィールドにも保管できます。CLOB は BLOB のサブセットと見なすことができます。

ヒント: *BLOB* という用語は、CLOB と BLOB の両方を指すものとして使用します。

ほとんどの非 BLOB データ型 (例えば、整数、浮動小数点数、日付など) には、そのデータ型に対して実行できる有効な操作の豊富なセットが存在します。例えば、FLOAT 値では、加算、減算、乗算、除算、およびその他の演算を行うことができます。BLOB はバイト列であり、データベース・サーバーにはそのバイト列の「意味」が分からないため (つまり、それらのバイトがムービーであるのか、歌であるのか、あるいはスペース・シャトルの設計であるのかが分からないため)、BLOB に対して行うことができる操作は、非常に限られています。

solidDB では、CLOB に対して、いくつかのストリング操作を行うことができます。例えば、LOCATE() 関数を使用して、CLOB の内部にある特定のサブストリング (例えば、人名など) を検索できます。そのような操作は、大量のサーバーのリソ

ース (メモリーか CPU 時間、またはその両方) を必要とするので、solidDB では、処理する CLOB のバイト数を制限できます。例えば、ストリング検索を行うとき、各 CLOB の最初の 1 メガバイトだけを検索するよう指定できます。詳しくは、「*IBM solidDB 管理者ガイド*」で **MaxBlobExpressionSize** 構成パラメーターの説明を参照してください。

理論的には、BLOB 全体を標準的な表の「内部」に保管することが可能ですが、BLOB が大きい場合は、通常、BLOB の大部分または全部が表に保管されていない場合の方がサーバーのパフォーマンスが良くなります。solidDB では、BLOB の長さが N バイト以下の場合、BLOB は表に保管されます。BLOB が N バイトより長い場合は、最初の N バイトが表に保管され、BLOB の残りの部分は、物理データベース・ファイル内のディスク・ブロックとして表の外部に保管されます。「N」の正確な値は、ある程度、表の構造やデータベース作成時に指定したディスク・ページ・サイズなどに依存しますが、最小値は常に 256 です。256 バイト以下のデータは、常に表内に保管されます。

データ行サイズがデータベース・ファイルのディスク・ブロック・サイズの 3 分の 1 より大きい場合は、それを部分的に BLOB として保管する必要があります。

SYS_BLOBS システム表は、物理データベース・ファイル内のすべての BLOB データのディレクトリーとして使用されます。1 つの SYS_BLOB 項目には、50 個の BLOB パーツを収容できます。BLOB サイズが 50 パーツを超える場合は、1 つの BLOB に複数の SYS_BLOB 項目が必要になります。

以下の照会は、データベース内にある BLOB の合計サイズの見積もりを返します。

```
select sum(totalsize) from sys_blobs
```

この見積もりは、情報がチェックポイントでのみ保守されているので、正確ではありません。2 つの空のチェックポイントの後、この照会は正確な応答を返します。

表内のさまざまな列値の説明

数値列の範囲とは、列に格納できる最小値と最大値を指します。文字列のサイズとは、文字データ型の列に格納できるデータの最大長を指します。

数値列の精度とは、列のデータ型で使用される最大桁数を指します。非数値列の精度とは、列に定義されている長さを指します。

数値列の位取りとは、小数点の右側の最大桁数を指します。近似浮動小数点数列の場合は、小数点の右側の桁数が固定でないため、位取りが定義されないことに注意してください。

列の長さは、データがそのデフォルトの C タイプに転送されるときにアプリケーションに返される最大バイト数です。文字データの場合は、長さに NULL 終端バイトが含まれません。列の長さと、データ・ソースでデータを格納するために必要なバイト数が異なる場合があります。

列の表示サイズは、データを文字形式で表示するために必要な最大バイト数です。

付録 B. solidDB SQL 構文

この付録では、SQL ステートメントについて簡単に説明するとともに、いくつかの例を示します。

このマニュアルの以前のバージョンでは、同期に関連する SQL コマンドが別の章に記載されていました。このバージョンでは、すべての SQL コマンドをこの付録にまとめています。

solidDB SQL の構文は、ANSI X3H2-1989 レベル 2 規格に基づいています。この規格には重要な ANSI X3H2-1992 (SQL-92) 拡張機能が含まれています。以前の規格にはないユーザーとロールの管理サービスは、ANSI SQL-99 ドラフトに基づいています。

ここに挙げるコマンドのほとんどは、solidDB のディスク・ベース・エンジンおよび solidDB のメイン・メモリー・エンジンで使用できます。拡張レプリケーションのライセンス交付を受けていない場合は、拡張レプリケーション同期に関連する一部のコマンドを使用できません。

ADMIN COMMAND

```
ADMIN COMMAND 'command_name'

command_name ::= ABORT | ASSERTEXIT | BACKUP |
BACKGROUNDJOB | BACKUPLIST | CHECKPOINTING | CLEANBGJOBINFO |
CLOSE | DESCRIBE | ERRORCODE | ERROREXIT | FILESPEC |
HELP | HOTSTANDBY | INDEXUSAGE | INFO | MAKECP | MEMORY | MESSAGES |
MONITOR | NETBACKUP | NETBACKUPLIST | NETSTAT | NOTIFY |
OPEN | PASSTHROUGH STATUS | PARAMETER | PERFMON | PID | PROCTRACE |
PROTOCOLS | REPORT | RUNMERGE | SAVE | SHUTDOWN |
SOLCONNECTOR PROPAGATOR SHUTDOWN | SQLLIST | STARTMERGE |
STATUS | THROWOUT | TID | TRACE | USERID | USERLIST |
USERTRACE | VERSION
```

使用法

ADMIN COMMAND は、管理コマンドを実行する SQL 拡張機能です。

solidDB SQL エディター (solsql) による ADMIN COMMAND の使用

solidDB SQL エディター (solsql) を使用する場合、構文内の *command_name* は solsql のコマンド・ストリングとなります。コマンド名は引用符で囲んで指定する必要があります。以下に例を示します。

```
ADMIN COMMAND 'backup'
```

solidDB リモート制御 (solcon) による ADMIN COMMAND の使用

solidDB リモート制御 (solcon) を使用する場合、構文には引用符なしの *command_name* のみが含まれます。以下に例を示します。

```
backup
```

省略形

ADMIN COMMAND に省略形を使用することもできます。以下に例を示します。

```
ADMIN COMMAND 'bak'
```

省略したコマンドのリストにアクセスするには、以下を実行します。

```
ADMIN COMMAND 'help'
```

結果セットには、 RC と TEXT という 2 つの列があります。

- RC (戻りコード) 列はコマンドの戻りコードです。コマンドの実行が成功した場合、値 0 が返されます。
- TEXT 列はコマンド応答です。

使用上の重要な注意

- **ADMIN COMMAND の一部のオプションはトランザクション用ではないため、ロールバックできません。**
- **ADMIN COMMAND およびトランザクション開始**

  ADMIN COMMAND はトランザクションに関するものではありませんが、まだトランザクションが開かれていない場合は、新規トランザクションを開始します。(これらのコマンドは、オープン・トランザクションのコミットやロールバックを行いません。) これによる影響は、通常では大きなものではありません。しかし、トランザクションの「開始時刻」に影響を及ぼし、それによって予想外の結果が生じる場合も考えられます。solidDB の並行性制御はバージョン管理システムに基づいており、データベースをトランザクション開始時の状態で認識します。

  例えば別のコミットをせずに ADMIN COMMAND を発行し、1 時間留守にすると、戻ってきたときに、次に実行する SQL コマンドは、データベースを 1 時間前、つまり ADMIN COMMAND でトランザクションを最初に開始したときの状態で認識することになります。
- **エラー・コード**

  ADMIN COMMAND のエラー・コードは、コマンド構文やパラメーター値が不正な場合にのみ、エラーを返します。要求された操作のみが開始できる場合には、コマンドは SQLSUCCESS (0) を返します。操作自体の結果は、結果セットに書き込まれます。結果セットには、RC と TEXT という 2 つの列があります。RC (戻りコード) の列には操作の戻りコードが示され、「0」は成功を、その他の数値はエラーを表します。そのため、ADMIN COMMAND ステートメントのコードと操作のコードの両方を確認することが必要です。

以下は、それぞれの ADMIN COMMAND コマンド・オプションの構文に関する説明です。

表 33. ADMIN COMMAND 構文とオプション

オプションの構文	説明
ADMIN COMMAND 'abort [backup | netbackup]'	アクティブなローカルまたはネットワークのバックアップ処理を中止します。バックアップ操作はアトミックである保証がないので、操作をキャンセルすると、次のバックアップが行われるまで、バックアップ・ディレクトリーに不完全なバックアップ・ファイルが作成されます。 オプションを入力しないと、デフォルトで ADMIN COMMAND 'abort backup' コマンドと同じ動作になります。
ADMIN COMMAND 'assertexit' 省略形: asex	正常なシャットダウンをせずに、ただちにサーバーを終了します。
ADMIN COMMAND 'backgroundjob' [LIST [-l] [*user*]] | [ABORT {*jobid* | *user* | ALL }] | [DELETE ERRORINFO {*jobid* | *user* | ALL }]' *user* ::= USER {*username*|*userid*} 省略形: bgjob	実行中のバックグラウンド・ジョブ、つまり START AFTER COMMIT (SAC) ステートメントを使用して開始された SQL ステートメントをリストし、場合によって中止します。 LIST オプションは、実行中のすべてのユーザー・ジョブまたは指定されたユーザーのジョブのみのいずれかをリストします。 -l オプションは、長いリスト (AC 'userlist -l' など) を参照します。 ABORT オプションは、ジョブ識別番号によりジョブを中止、またはユーザー識別番号により全ジョブを中止します。引数なしで ABORT を入力すると、全ユーザーの全ジョブが中止されます。 DELETE ERRORINFO オプションは、バックグラウンド・ジョブで発生したエラーを格納してある SYS_BACKGROUNDJOB_INFO システム表からエラー情報を削除します。このオプションは、推奨されない ADMIN COMMAND 'CLEANBGJOBINFO' コマンドと同じ操作を行います。
ADMIN COMMAND 'backup [-s] [*backup_directory*]' 省略形: bak	データベースのバックアップを作成します。この操作は、同期または非同期 (デフォルト) で行うことができます。同期操作の指定には、オプションの -s パラメーターを使用します。 デフォルトのバックアップ・ディレクトリーは、構成パラメーター BackupDirectory の [General] セクションに定義されたものです。引数としてバックアップ・ディレクトリーを指定することもできます。例えば、backup abc と指定すると、バックアップをディレクトリー abc に作成します。すべてのディレクトリー定義が、solidDB 作業ディレクトリーからの相対的な位置です。
ADMIN COMMAND 'backuplist' 省略形: bls	前回のローカル・バックアップの状況のリストを表示します。
ADMIN COMMAND 'cleanbgjobinfo' 省略形: cleanbgi	**注:** このコマンドは、推奨されません。詳しくは、backgroundjob コマンドを参照してください。 バックグラウンド・プロシージャーの状況データを格納した SYS_BACKGROUNDJOB_INFO 表を消去します。
ADMIN COMMAND 'checkpointing' 省略形: cp	チェックポイントをオンまたはオフにします。
ADMIN COMMAND 'close' 省略形: clo	新しい接続に対してサーバーを閉じます。新しい接続は許可されません。
ADMIN COMMAND 'describe parameter *param*' 省略形: des	指定されたパラメーターの説明を返します。 param は、**section_name.param_name** の形式で指定する必要があります。セクション名およびパラメーター名では、大/小文字を区別しません。 以下の例では、パラメーター **Com.Trace = y/n** についての説明が示されます。 ADMIN COMMAND 'des parameter com.trace'
ADMIN COMMAND 'errorcode {all | *SOLID_error_code*}' 省略形: ec	特定のエラー・コード (またはすべてのコード) の説明を表示します。errorcode 10033 のように、引数としてコード番号を指定してください。
ADMIN COMMAND 'errorexit <number>' 省略形: erex	指定された処理終了コードですぐにサーバーの処理を強制的に終了させます。
ADMIN COMMAND 'filespec' 省略形: fs	データベース・ファイルの指定、現行の充てん率、および現行のファイル・サイズを表示します。
ADMIN COMMAND 'help' 省略形: ?	使用可能コマンドを表示します。
ADMIN COMMAND 'hotstandby [option]' 省略形: hsb	HotStandby コマンドです。 オプションのリストについては、「*IBM solidDB 高可用性ユーザー・ガイド*」を参照してください。 オプションのリストについては、「*IBM solidDB 高可用性ユーザー・ガイド*」の HotStandby ADMIN COMMANDを参照してください。

表 33. ADMIN COMMAND 構文とオプション (続き)

オプションの構文	説明
ADMIN COMMAND 'indexusage' 省略形: ixu	索引と、各索引が使用された回数を表示します。

ADMIN COMMAND 'info *options*'
省略形: info
	サーバー情報を返します。サーバー情報は、25 行のデータで構成されています。表示された情報は、値の意味を説明するものではありません。しかし、以下のリストを使用して、各値の意味を調べることができます。 25 の値は、上から順に以下のとおりです。
	• numusers: 現行ユーザーの数
	• maxusers: ユーザーの最大数
	• sernum: サーバー通し番号
	• dbsize: データベースのサイズ
	• logsize: ログ・ファイルのサイズ
	• uptime: サーバーの稼働時間
	• bcktime: 正常に完了した前回のローカル・バックアップのタイム・スタンプ
	• cptime: 前回正常に完了したチェックポイントのタイム・スタンプ
	• tracestate: 現行のトレース状態
	• monitorstate: 現行のモニター状態。これは、現在 SQL モニターを有効にしているユーザーの数を示します。すべてのユーザーが SQL モニターを有効にしている場合、この値は -1 になります。SQL モニターを有効にするには、`ADMIN COMMAND 'monitor {on | off} [user {username | userid}]'` (後述) を使用することに注意してください。
	• openstate: 現在のオープンまたはクローズ状態。つまり、データベース・サーバーが新しい接続を受け入れるかどうかを示します。「open」は、データベース・サーバーが新しい接続を受け入れることを意味します。
	• nummerges: マージの数
	• numlocks: ロックの数
	• numcursors: オープン・カーソルの数
	• numtransactions: オープン・トランザクションの数
	• memtotal: 割り振られたメモリーの総バイト数
	• dbfreesize: データベースのフリー・スペースの残量
	• dbpagesize: データベースのページ・サイズ
	• imdbsize: インメモリー表 (テンポラリー表およびトランジエント表を含む) およびそれらの表の索引により使用されているスペースの量。戻り値はキロバイト (KB) で示され、VARCHAR の形式になります。
	• name: サーバー名を出力します。
	• primarystarttime: 1 次ロールの開始時間
	• secondarystarttime: 2 次ロールの開始時間
	• dbconfigsize: 構成されたデータベースのサイズ
	• dbcreatetime: このオプションは、データベース作成のタイム・スタンプを出力します。省略形の dbcreationtime も使用することができます。
	• processsize: このオプションは、システム・レベルの仮想プロセス・サイズをキロバイトで出力します。省略形の psize も使用することができます。
	
	1 つのコマンドにつき複数のオプションを使用することができます。値は、1 行に値が 1 つずつ要求時と同じ順序で返されます。
	
	コマンド例:
	
	`ADMIN COMMAND 'info dbsize logsize'`
	
	出力の例:
	
	`RC TEXT`
	`0    851968`
	`0    573440`

ADMIN COMMAND 'makecp [-s]' 省略形: mcp	チェックポイントを作成します。SYS_ADMIN_ROLE 特権が必要です。 デフォルトでは、チェックポイントは非同期です。-s オプションを指定すると、コマンドはチェックポイント完了後にのみ戻ります。
ADMIN COMMAND 'memory' 省略形: mem	サーバー処理のメモリー・サイズを返します。報告される処理のメモリー・サイズは、オペレーティング・システムから報告されるものと異なる場合があります。

表 33. ADMIN COMMAND 構文とオプション (続き)

オプションの構文	説明
ADMIN COMMAND 'messages [{ warnings | errors}] [*count*]' 省略形: mes	サーバーのメッセージを表示します。オプションで重大度およびメッセージ番号も定義することができます。例えば、以下のように指定します。 ADMIN COMMAND 'messages warnings 100' で、最新の 100 件の警告が表示されます。
ADMIN COMMAND 'monitor {on | off} [user {*username* | *userid*}]' 省略形: mon	サーバーのモニターをオンまたはオフに設定します。モニターでは、ユーザー・アクティビティーおよび soltrace.out ファイルに対するSQL 呼び出しをログに記録します。
ADMIN COMMAND 'netbackup [*options*] [DELETE_LOGS | KEEP_LOGS] [connect *connect str*] [dir *backup dir*]' 省略形: nbak	データベースのネットワーク・バックアップを作成します。この操作は、同期または非同期 (デフォルト) で行うことができます。同期操作の指定には、オプションの -s パラメーターを使用します。 DELETE_LOGS パラメーターを使用すると、ソース・サーバーのバックアップされたログ・ファイルが削除されます。これをフル・バックアップと呼ぶ場合もあります。これはデフォルト値です。一方、KEEP_LOGS パラメーターを使用すると、バックアップされたログ・ファイルはソース・サーバーに残ります。これをコピー・バックアップと呼ぶ場合もあります。キーワード KEEP_LOGS を使用することは、General パラメーターの NetbackupDeleteLog を「no」に設定することと同じです。 デフォルトの接続ストリングおよびデフォルトのネットバックアップ・ディレクトリーは、構成ファイルの [General] セクションの NetBackupConnect パラメーターおよび NetBackupDirectory パラメーターで定義されています。 netbackup コマンドで入力されたオプションは、構成ファイルで指定された値をオーバーライドします。ディレクトリー定義は、solidDB 作業ディレクトリーからの相対的な位置です。
ADMIN COMMAND 'netbackuplist' 省略形: nbls	データベース・サーバーの最近作成されたネットワーク・バックアップの状況リストを表示します。
ADMIN COMMAND 'netstat'	サーバーの設定およびネットワークの状況を表示します。
ADMIN COMMAND 'notify user {*username* | *user id* | ALL } *message*' 省略形: not	このコマンドは、イベント ID の NOTIFY により指定されたユーザーにイベントを送信します。この ID は、ステートメントのタイムアウトが短くて切断ができない場合にイベント待ちのスレッドをキャンセルする、またはイベント登録を変更するのに使用します。 以下の例では、ユーザー ID が 5 のユーザーに通知メッセージを送信した後、イベントはメッセージ・パラメーターの値を受け取ります。 ADMIN COMMAND 'notify user 5 Canceled by admin'
ADMIN COMMAND 'open' 省略形: ope	新しい接続に対してサーバーを開きます。新しい接続が許可されます。
ADMIN COMMAND 'parameter [option][name[=[*|value][temporary]]' 省略形: par	サーバーのパラメーター値を表示および設定します。値を指定せずにコマンドを実行すると、パラメーターは開始値に設定されます。アスタリスク (*) をパラメーター値に割り当てると、そのパラメーターはファクトリー値に設定されます。「name」にできるのはセクション名のみ、またはセクション名とパラメーター名の間にピリオドを付けた名前 (「com.trace」など) です。以下に例を示します。 • parameter を単独で使用すると、すべてのパラメーターを表示します。 • parameter general は、[General] セクションのすべてのパラメーターを表示します。 • parameter general.readonly は、[General] セクションの readonly という名前の単一パラメーターを表示します。セクション名 ([General]) とパラメーター名 (readonly) の間にピリオドを入力する必要があります。 • parameter com.trace=yes は、通信トレースをオンに設定します。セクション名 ([Com] など) とパラメーター名 (trace など) の間にピリオドを入力する必要があります。 等号の前後にブランクを入力しないでください。 • parameter com.trace= は、通信トレースを開始値に設定します。 • parameter com.trace=* は、通信トレースをファクトリー値に設定します。 出力には、以下のように 3 つの値が含まれる場合があります。 0 Logging DurabilityLevel 1 2 3 上記 3 つの値は、以下を示します。 • 1 は現行値 (動的に設定可能) です。 • 2 は INI ファイルの値 (開始値) です。 • 3 はファクトリー値です。 -r オプションを使用すると、現行のパラメーター値のみが返されます。

表 33. ADMIN COMMAND 構文とオプション (続き)

オプションの構文	説明
ADMIN COMMAND 'passthrough status' 省略形: pt	SQL パススルー接続に関する以下の状況情報を提供します。 • NO REMOTE SERVER - リモート・サーバー・オブジェクトが定義されていません。 • NOT CONNECTED - 接続されていません。エラーはありません。 • CONNECTED - 接続されています。 • LOGIN FAILED - ログインに失敗しました。 • CONNECTION BROKEN - 接続に失敗しました。
ADMIN COMMAND 'perfmon [- c | - r] [options] [*name_prefix_list*]' 省略形: pmon	サーバーのパフォーマンス・カウンターを返します。以下のオプションが使用できます。 • -c: 実際のカウンター値を出力します。このオプションが指定されない場合、出力数値は 1 秒当たりの操作数となります (該当する場合)。 • -r: ロー・モードで出力します。このモードでは、フォーマット設定のない最新のカウンター値のみが含まれます。オプション名や追加情報は出力されません。このオプションは、カウンター値をサーバーからリトリーブする別の外部プログラムを使用して実際のモニターを行う場合に便利です。 • -xtime: 時間を秒単位で出力します。 • -xtimediff: 前回の pmon 呼び出しに対する差分をミリ秒単位で出力します。 • -xnames: 出力の列名を出力します。 • -xdiff: 絶対値の代わりに、前回実行した perfmon に対する差分を示します。 • *name_prefix_list*: 出力を特定のカウンター名に制限します。例えば、ファイル関連のカウンターをすべて出力するには、*name_prefix_list* を file とします。複数の接頭部を指定することもできます。 以下の例では、すべての情報が返されます。 ADMIN COMMAND 'perfmon' 以下の例では、名前が file および cache という接頭部で始まるすべての値がカウンター値として返されます。 ADMIN COMMAND 'perfmon-c file cache' file および cache という接頭部は、perfmon 出力内に含まれるカウンター名と一致することに注意してください。
ADMIN COMMAND 'perfmon [diff [start | stop] [*filename interval*]' 省略形: pmon diff	サーバーのパフォーマンス・カウンターを返します。以下のオプションが使用できます。 • diff: すべての perfmon カウンターを、指定された間隔でファイルに出力するサーバーのタスクを開始します。間隔は、ミリ秒で指定する必要があります。出力ファイルは、1 行目にカウンター名を示した、「コンマ区切り値」で記述されます。ファイルはそのままで Excel などのスプレッドシート・プログラムで処理することが可能です。 • filename: デフォルト値は pmondiff.out です。 • interval: デフォルト値は 1000 ミリ秒です。 以下のコマンド例では、1000 ミリ秒間隔で myd.csv ファイルに書き込みを行う diff タスクが開始されます。 ADMIN COMMAND 'pmon diff start myd.csv 1000'
ADMIN COMMAND 'pid' 省略形: pid	サーバーのプロセス ID を返します。
ADMIN COMMAND 'proctrace { on | off } user *username* { procedure | trigger | table } *entity_name*' 省略形: ptrc	ストアード・プロシージャーおよびトリガーのトレースをオンにします。 「username」は、トレースしたいプロシージャー呼び出し (またはトリガー) のユーザー名です。複数の接続が同じユーザー名を使用している場合、それらの接続の呼び出しがすべてトレースされます。さらに、拡張レプリケーションを使用している場合、レプリカの呼び出しだけではなく、マスターに伝搬された後でマスターで実行された呼び出しもトレースされます。 「entity_name」は、トレースをオンまたはオフにしたいプロシージャー、トリガー、または表の名前です。プロシージャー名またはトリガー名を指定した場合、指定されたプロシージャーまたはトリガー内のステートメントごとに出力を生成します。表名を指定した場合、その表でのすべてのトリガーに対して出力を生成します。トレースは、指定されたユーザー名によりプロシージャーまたはトリガーが呼び出された場合にのみアクティブ化されます。 proctrace について詳しくは、「*IBM solidDB SQL ガイド*」のストアード・プロシージャーおよびトリガーのトレース機能のセクションを参照してください。 ADMIN COMMAND 'usertrace' も参照してください。

表 33. ADMIN COMMAND 構文とオプション (続き)

オプションの構文	説明
ADMIN COMMAND 'protocols' 省略形: prot	1 行に 1 プロトコルずつ、使用可能な通信プロトコルのリストを返します。 例: `ADMIN COMMAND 'protocols'`
ADMIN COMMAND 'report *filename*' 省略形: rep	引数で指定されたファイルにサーバー情報のレポートを生成します。
ADMIN COMMAND 'runmerge' 省略形: rm	索引マージを実行します。
ADMIN COMMAND 'save parameters [*filename*]' 省略形: save	現行の構成パラメーターの一連の値をファイルに保存します。ファイル名が指定されていない場合、デフォルトの solid.ini ファイルに再書き込みされます。この操作は、各チェックポイントにおいて暗黙的に行われます。
ADMIN COMMAND 'shutdown [force]' 省略形: sd	solidDB を停止します。 force オプションが使用されると、アクティブなトランザクションは中止され、ユーザーは強制的に切断されます。
ADMIN COMMAND 'sqllist top *number_of_statements*'	このコマンドは、現在実行中のステートメントの中で最も実行時間が長い SQL ステートメントのリストを出力します。このリストには、選択された数のステートメントが含まれます。
ADMIN COMMAND 'status' 省略形: sta	サーバーの統計情報を表示します。
ADMIN COMMAND 'status backup | netbackup' 省略形: sta backup | netbackup	前回開始されたローカル・バックアップまたはネットワーク・バックアップの状況を表示します。該当する状況は、以下のうち 1 つです。 • 前回のバックアップが成功している、またはバックアップが要求されていない場合は、「0 SUCCESS」が出力されます。 • バックアップが途中である (例えば、開始されたが、まだ準備ができていない) 場合、「14003 ACTIVE」が出力されます。 • バックアップの終了処理中は、「14003 STOPPING」が出力されます。 • 前回のバックアップが失敗している場合、「*errorcode* ERROR」と出力されます。ここで *errorcode* は失敗の理由を示します。
ADMIN COMMAND 'startmerge' 省略形: sm	マージを開始し、その完了を待ちます。
ADMIN COMMAND 'throwout {*username* | *userid* | all}' 省略形: to	ユーザーを solidDB から退去させます。指定したユーザーを退去させる場合、引数にユーザー ID を指定します。すべてのユーザーを退去させる場合、引数にキーワードの ALL を使用します。
ADMIN COMMAND 'tid' 省略形: tid	このコマンドは、(サーバー内の) 現行ユーザー・スレッドの ID (4 桁のコード) を返します。

表 33. ADMIN COMMAND 構文とオプション (続き)

オプションの構文	説明
ADMIN COMMAND 'trace { on │ off} sql │ rpc │ sync │ info <level> │ flowplans │ logreader │ passthrough │ all' 省略形: tra	サーバーのトレースをオンまたはオフに設定します。以下のトレース・オプションが使用できます。 • sql : SQL メッセージ • rpc : ネットワーク通信 • sync : 同期メッセージ • info <level> : SQL 実行トレース (レベルは 0 から 8) • flowplans: Flow SQL ステートメントのプラン • ログ・リーダー: 以下の情報のログをトレース・ファイル soltrace.out に記録します。 – ログ・リーダーが読み取りを開始。 – ログ・リーダー・カーソル内のエラーが開始。合計 14 の異なるエラー状態が出力されます。 – ログ・リーダーが読み取りを停止。 – 特定のシステム変更後に、読み取りが異常停止。 – 返されるログ・レコード数と読み取りの進行に関する高水準の情報。 各情報はユーザー ID でタグ付けされており、異なるユーザーからの操作を区別できます。 • passthrough - SQL パススルー接続と ODBC ドライバーのロードに関するトレース情報が、以下のように提供されます。 – ODBC ドライバーのロード: ドライバー名とロード状況 – バックエンドへの接続状況: 接続/再接続/切断/失敗 オプションが 1 つも指定されないか、またはすべてのオプションが指定された場合、SQL メッセージおよびネットワーク通信メッセージの両方がトレース・ファイルに書き込まれます。デフォルトのトレース・ファイル名は、`soltrace.out` です。
ADMIN COMMAND 'userid' 省略形: uid	現行接続のユーザー識別番号を返します。 例: `ADMIN COMMAND 'userid'`

表 33. ADMIN COMMAND 構文とオプション (続き)

オプションの構文	説明
ADMIN COMMAND 'userlist [-l] [*name* | *id*]' 省略形: ul	このコマンドは、現在データベースにログインしているユーザーのリストを、1 次属性の数とともに表示します。この属性は、User name、User Id、Type、Machine Id、Login time、および Appinfo (オプション) です。属性の説明については、以下の詳細な出力の説明を参照してください。 オプション -l (long) は、さらに詳細な出力を表示します。long 出力のフィールドは、以下のとおりです。 • *Id*: データベース内のユーザー・セッション識別番号。ID の存続時間は、ユーザー・セッションと同じです。ユーザーがログアウトした後、その番号を再利用できます。 • *Type*: クライアント・タイプ。以下の値が可能です。 – *Java*。JDBC を使用しているクライアントを指します。 – *ODBC*。ODBC を使用しているクライアントを指します。 – *SQL*。solidDB SQL エディター (solsql) を指します。 • *Machine*: クライアントのコンピューター名 (ホスト名) および (使用可能であれば) その IP アドレス。 • *Login tile*: クライアント・コンピューターのログイン・タイム・スタンプ。 • *Appinfo*: クライアントが ODBC を使用している場合のクライアント・コンピューターの環境変数 SOLAPPINFO の値。JDBC の場合、この値を出力中に表示するには、Java ユーティリティーのプロパティー solid_appinfo をこの値に設定する必要があります。代わりに、以下の Java コマンド行を使用して、環境変数の値をドライバーに渡すこともできます。 java -Dsolid_appinfo=%SOLAPPINFO% *java program name* 注: SOLAPPINFO の値にブランクを含めないでください。 • *Last activity*: クライアントが前回サーバーに要求を送信した時刻。 • *Autocommit*: 自動コミット・モードがオフ (値が 0) に切り替わっている場合、現行トランザクションは、COMMIT または ROLLBACK のステートメントが発行されるまで開いています。その後、新しいステートメントにより新しいトランザクションが開始されます。 自動コミット・モードがオン (値が 1) に切り替わっている場合、各ステートメントが自動的にコミットされます。 • *RPC compression*: データ伝送圧縮がオンまたはオフのいずれになっているかを示します。 • *Transparent failover*: このフィールドは、透過的フェイルオーバー (TF) が使用されているかどうかを示します。 透過的フェイルオーバーは、HotStandby 構成の 1 つの特性です。それは、サーバーのロール変更をユーザーから隠します。solidDB ツールは TF に対応していないので、このフィールドには「no」の値のみが示されます。 • *Transparent cluster*: 透過クラスターは、この接続に対して (HSB 内の) 負荷のバランシング機能が有効となっているかどうかを示します。 • *Transaction active*: このフィールドは、非コミットのオープン・トランザクションが接続上にある (値が 1) またはない (値が 0) ことを示します。接続が自動コミットに設定されているとき、ほとんどの場合この値は 0 になります。 • *Transaction duration*: このフィールドは、現行のオープン・トランザクションの期間を示します。COMMIT または ROLLBACK の後、この値は 0 になります。 • *Transaction isolation*: このフィールドは、トランザクションのトランザクション分離レベルを示します。分離レベルにより、実行中のトランザクションの一部であるデータをどのように他のトランザクションに見せるかが決定されます。 • *Transaction durability*: このフィールドは、現行のオープン・トランザクションの持続性を示します。デフォルトでは、solidDB はアダプティブ 持続性を使用します。 • *Transaction safeness*: このフィールドは、現行のオープン・トランザクションの安全性を示します。安全性は、 SafenessLevel パラメーターを使用して設定します。 デフォルトでは、solidDB は *2safe* のトランザクション安全性を使用します。 • *Transaction autocommit*: このフィールドは、現行のオープン・トランザクションが自動的にコミットされるかどうかを示します。トランザクションの自動コミットが現行トランザクションに対してオフ (値が 0) に切り替わっている場合、現行トランザクションは、COMMIT または ROLLBACK のステートメントが発行されるまで開いています。その後、新しいステートメントにより新しいトランザクションが開始されます。 自動コミット・モードが現行トランザクションに対してオン (値が 1) に切り替わっている場合、各ステートメントが自動的にコミットされます。

表 33. ADMIN COMMAND 構文とオプション (続き)

オプションの構文	説明
(続き) ADMIN COMMAND 'userlist [-l] [*name* | *id*]' 省略形: ul	• *Current® schema*: 現行のスキーマ名を示します。 • *Current catalog*: 現行のカタログ名を示します。 • *Sortgroubby*: 結果グループの数が明らかでない場合、どのように GROUP BY ステートメントを実行するかを示します。可能な値は 2 つあります。 – ADAPTIVE: 結果グループの実数が GROUP BY 用の中央メモリーの配列に収まる行数を超えた場合、GROUP BY の入力が事前にソートされます。 – STATIC: GROUP BY リストに少なくとも 2 つの項目がある場合、GROUP BY の入力が常に事前にソートされます。そうでない場合、GROUP BY の入力は事前にソートされません。 • *Simple optimizer rules*: solid.ini の SQL パラメーター SimpleOptimizerRules の値を示します。可能な値は、Yes/No/Default です。 • *Statement max time*: 接続固有のステートメント最大実行時間を秒数で示します。この設定は、最大時間が新たに設定されるまで有効です。時間がゼロの場合、最大時間がないことを示します。これがデフォルト値です。 • *Lock timeout*: SET LOCK TIMEOUT ステートメントを使用して設定したタイムアウトを示します。 • *Optimistic lock timeout*: SET OPTIMISTIC LOCK TIMEOUT ステートメントを使用して設定したタイムアウトを示します。 • *Idle timeout*: SET IDLE TIMEOUT ステートメントを使用して設定したタイムアウトを示します。 • *Join Path Span*: SET SQL JOINPATHSPAN ステートメントを使用して設定したパス結合スパン値を示します。 • *RPC seqno*: 内部プロトコル・メッセージのシーケンス番号。 • *SQL sortarray*: ユーザー固有の内部ソート配列のサイズ。 • *SQL unionsfromors*: この値は、(最大で) いくつの OR 演算子を和集合 (UNION) に変換できるかを示します。和集合の方が実行速度は早いですが、実行するのに必要なメモリーが多くなります。 • *EVENT QUEUE LENGTH*: イベント・キュー中の通知済みイベントの数を示します。 • *Stmt id*: 現行ステートメントの識別番号。番号はセッション固有であり、それぞれ異なるステートメントに割り当てられます。 • *Stmt state*: 内部ステートメント実行状態。 • *Stmt rowcount*: 現行ステートメントでリトリーブまたは挿入された行数。 • *Stmt starttime*: 現行ステートメントの開始日時。 • *Stmt duration*: 秒単位の内部ステートメント期間。注: この値は、外部に表示されるステートメント待ち時間とは関係ありません。通常、ステートメント期間は待ち時間よりかなり長くなります。 • *Stmt SQL str*: 現行ステートメントのストリング。
ADMIN COMMAND 'usertrace { on | off } user *username* { procedure | trigger | table } *entity_name*' 省略形: utrc	ストアード・プロシージャーおよびトリガーにおいてユーザー・トレースをオンにします。このコマンドは、指定されたプロシージャーまたはトリガー内の WRITETRACE ステートメントごとに出力を生成します。 「username」は、トレースしたいプロシージャー呼び出し (またはトリガー) のユーザー名です。複数の接続が同じユーザー名を使用している場合、それらの接続の呼び出しがすべてトレースされます。さらに、拡張レプリケーションを使用している場合、レプリカの呼び出しだけではなく、マスターに伝搬された後でマスターで実行された呼び出しもトレースされます。 「entity_name」は、トレースをオンまたはオフにしたいプロシージャー、トリガー、または表の名前です。表名を指定した場合、その表でのすべてのトリガーに対して出力を生成します。トレースは、指定されたユーザーによりプロシージャーまたはトリガーが呼び出された場合にのみアクティブ化されます。 usertrace について詳しくは、「*IBM solidDB SQL ガイド*」のストアード・プロシージャーおよびトリガーのトレース機能のセクションを参照してください。 ADMIN COMMAND 'proctrace' も参照してください。
ADMIN COMMAND 'version' 省略形: ver	サーバーのバージョン情報およびご使用の solidDB ソフトウェアのライセンスに関連した情報を表示します。

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
```

使用法

これは solidDB 固有の SQL 拡張機能です。これを使用すると、ストアード・プロシージャーの作成や呼び出しをしなくても、システム生成イベントについて登録し、それらのイベントを待つことができます。

明示的にイベントについて登録し、イベントを待つ必要があります。例えば、以下のとおりです。

```
ADMIN EVENT 'register sys_event_hsbstateswitch';
ADMIN EVENT 'wait';
```

イベントがシステムによって通知された後、以下のようなものが表示されます。

```
ENAME                    POSTSRVTIME         UID NUMDATAINFO TEXTDATA
-----                    -----------         --- ----------- --------
SYS_EVENT_HSBSTATESWITCH 2003-10-28 18:10:14 -1  NULL        PRIMARY ACTIVE

1 rows fetched.
```

イベントを待つ前に、そのイベントについて登録を行う必要があります。 (これは、ストアード・プロシージャーでの WAIT の機能と異なります。ストアード・プロシージャーでは、明示的な登録はオプションです。)

注:

このコマンドで (「SYNC_」で始まる) 同期イベントに対して登録を行うことはできません。その目的には、プロシージャー言語コマンド WAIT EVENT を使用できます。

接続はイベント待ちを開始した後、イベントが通知されるまで、他のことを何もできなくなります。

複数のイベントについて、登録することができます。待つ場合、どのタイプのイベントを待つかを指定することはできません。待ちは、登録してあるイベントのいずれかを受信するまで続きます。

ADMIN EVENT を使用して、ユーザー・イベントでなくシステム・イベントのみを待つこともできます。ユーザー・イベントを待つ場合は、ストアード・プロシージャーを作成して呼び出す必要があります。

ADMIN EVENT コマンドは、イベントを通知するオプションを備えていません。

ADMIN EVENT を使用するには、DBA 特権を持っているか、ロール SYS_ADMIN_ROLE を付与されている必要があります。

例

```
ADMIN EVENT 'register sys_event_hsbstateswitch';
ADMIN EVENT 'wait';
ADMIN EVENT 'unregister sys_event_hsbstateswitch';
```

ALTER REMOTE SERVER

```
ALTER REMOTE SERVER SET USERNAME | PASSWORD <password>
```

使用法

ALTER REMOTE SERVER ステートメントは、SYS_SERVER システム表のバックエンド・ログイン情報を変更します。ログイン・データは、Universal Cache での SQL パススルーのために使用されます。

通常の用途では、このステートメントをエンド・ユーザーが使用することはありません。

例

TBD

関連項目

CREATE [OR REPLACE] REMOTE SERVER

DROP REMOTE SERVER

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}
```

使用法

表の構造は、ALTER TABLE ステートメントによって変更できます。列の追加、除去、変更、または名前変更を行うことができます。表にオプティミスティック並行性制御とペシミスティック並行性制御のどちらを使用するかを変更できます。表をメモリーとディスクのどちらに格納するかを変更できます。表が属するスキーマを変更できます。

サーバーでは、ユーザーは ALTER TABLE コマンドを使用して、列の幅を変更できます。列幅は、随時 (つまり表が空 (行がない) か空でないかにかかわらず) 大きくすることができます。ただし、ALTER TABLE コマンドでは、表が空でないときに列幅を小さくすることは許可されません。列幅を小さくするには、表が空でなければなりません。

ユニーク・キーまたは主キーの一部になっている列は除去できないことに注意してください。

表の所有者を変更するには、ALTER TABLE *base_table_name* MODIFY SCHEMA *schema_name* ステートメントを使用します。このステートメントは、作成者権限を含むすべての権限を新しい所有者に付与します。表に対する旧所有者のアクセス権限は、作成者権限を除いて保存されます。

SET HISTORY COLUMNS 節については、『ALTER TABLE ... SET HISTORY COLUMNS』を参照してください。

SET [NO]SYNCHISTORY 節については、191 ページの『ALTER TABLE ... SET SYNCHISTORY』を参照してください。

ステートメント ALTER TABLE *base_table_name* SET {OPTIMISTIC | PESSIMISTIC} を使用して、個々の表をオプティミスティックまたはペシミスティックに設定できます。デフォルトでは、すべての表がオプティミスティックになります。データベース全体のデフォルトは、構成ファイルの General セクションで、パラメーター Pessimistic = yes によって設定できます。

表は、ディスク・ベースからインメモリーに変更でき、その逆の変更もできます。(これは、solidDB メインメモリー・エンジンでのみ許可されます。) これは、表が空の場合にのみ行うことができます。表を、既に使用しているのと同じストレージ・モードに変更しようとした場合 (例えば、インメモリー・ストレージを使用するためにインメモリー表を変更しようとした場合)、コマンドは効果がなく、エラー・メッセージも発行されません。

例

```
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_tabl ADD CONSTRAINT div_check CHECK(division_id < 12);
ALTER TABLE dept_tabl DROP CONSTRAINT div_check;
```

ALTER TABLE ... SET HISTORY COLUMNS

```
ALTER TABLE table_name SET HISTORY COLUMNS ( col1, col2, colN ...)
```

使用法

同期履歴プロセスをさらに最適化するため、同期履歴用の表を設定した後、SET HISTORY COLUMNS ステートメントを使用して、マスター内およびそれに対応する同期表内のどの列の更新項目を履歴表に入れるかを指定できます。このステートメントを使用して特定の列を指定しなかった場合、マスター・データベース内での (すべての列に対する) すべての更新操作は、対応する同期表が更新されると、履歴表に新規項目を生成します。一般に、検索基準または結合に使用される列には、ALTER TABLE ... SET HISTORY COLUMNS を使用することを推奨します。

マスターでの使用

SET SYNCHISTORY および SET HISTORY COLUMNS をマスターで使用して、表のインクリメンタル・パブリケーションを使用可能にします。

レプリカでの使用

SET SYNCHISTORY および SET HISTORY COLUMNS をレプリカで使用して、表のインクリメンタル REFRESH を使用可能にします。

注:

ALTER TABLE ... SET HISTORY COLUMNS を成功させるには、最初にステートメント ALTER TABLE ... SET SYNCHISTORY を実行しておく必要があります。ALTER TABLE ... SET NOSYNCHISTORY を実行すると、ALTER TABLE ... SET HISTORY COLUMNS の効果もなくなります。

例

```
ALTER TABLE myLargeTable SET HISTORY COLUMNS (accountid);
```

戻り値

各エラー・コードについて詳しくは、「*solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。

表 34. ALTER TABLE SET HISTORY COLUMNS の戻り値

エラー・コード	**説明**
13047	操作する特権がありません
13100	正しくない表モードの組み合わせ
13134	表が基本表ではありません
25038	表がパブリケーション *publication_name* で参照されていますが、ドロップ操作または変更操作は許可されません
25039	表がパブリケーション *publication_name* に対するサブスクリプションで参照されていますが、ドロップ操作または変更操作は許可されません

関連項目

ALTER TABLE ... SET SYNCHISTORY

ALTER TABLE ... SET SYNCHISTORY

```
ALTER TABLE table_name SET {SYNCHISTORY | NOSYNCHISTORY}
```

使用法

SET [NO]SYNCHISTORY

「SET SYNCHISTORY / NOSYNCHISTORY」節はサーバーに、この表に solidDB アーキテクチャーのインクリメンタル・パブリケーション・メカニズムを使用するよう指示します。デフォルトでは、SYNCHISTORY はオンではありません。指定された表について、このステートメントが SYNCHISTORY に設定された場合、メイン表の古いバージョンの更新または削除された列を保管するために、シャドー表が自動的に作成されます。シャドー表は、「同期履歴表」または単に「履歴表」と呼ばれます。

履歴表内のデータは、レプリカがマスター内のパブリケーションからインクリメンタル REFRESH を取得するときに参照されます。例えば、スミスさんの電話料金請求書のレコードがメイン表から削除されるとします。このレコードのコピーは同期履歴表内に格納されています。レプリカがリフレッシュされると、マスターは履歴表を検査して、レプリカにスミスさんのレコードが削除されたことを知らせます。これにより、レプリカもそのレコードを削除できます。削除または変更されたレコードのパーセンテージがかなり低い場合は、インクリメンタル更新の方が、表全体をマスターからダウンロードするより高速です。(ユーザーがインクリメンタル REFRESH でなく完全 REFRESH を行う場合、履歴表は使用されません。マスター上の表内のデータが、単にレプリカにコピーされます。)

バージョン管理されたデータは、REFRESH 要求を満たすためにそのデータを必要とするレプリカが 1 つもなくなった時点で、データベースから自動的に削除されます。

表をマスター/レプリカ同期に参加させるには、事前にこのコマンドを使用して、同期履歴をオンにする必要があります。このコマンドは、中にデータが現在存在する表に対しても実行できます。しかし、ALTER TABLE SET SYNCHISTORY は、指定された表が既存のパブリケーションによって参照されていない場合にのみ使用できます。

SET SYNCHISTORY は、マスターとレプリカの両方のデータベース表で指定する必要があります。

ある表について、SYNCHISTORY がオンであるかどうかを SYS_TABLEMODES システム表から検査できます。MODE 列に、SYNCHISTORY 情報が含まれています。

例えば、以下の照会を使用できます。

```
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 は、モードが明示的に設定された表のモードのみを表示します。言い換えれば、SYS_TABLEMODES はデフォルト・モードのままにされた表のモードを表示しません。表に SYNCHISTORY (または NOSYNCHISTORY) を設定しなかった場合、照会は空の結果セットを返します。

マスターでの使用

SET SYNCHISTORY をマスターで使用して、表のインクリメンタル・パブリケーションを使用可能にします。

レプリカでの使用

SET SYNCHISTORY をレプリカで使用して、表のインクリメンタル REFRESHES を使用可能にします。

注:

レプリカが読み取り専用の場合 (パブリケーションの複製された部分に変更が加えられていない場合)、ステートメント ALTER TABLE ... SET SYNCHISTORY は必要ありません。同時に、以下の Flow Replica 常駐パラメーターを設定する必要があります。

```
set sync parameter SYS_SYNC_KEEPLOCALCHANGES 'Yes';
```

例

```
ALTER TABLE myLargeTable SET SYNCHISTORY;
ALTER TABLE myVerySmallTable SET NOSYNCHISTORY;
```

戻り値

各エラー・コードについて詳しくは、「*solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。

表 35. ALTER TABLE SET SYNCHISTORY の戻り値

エラー・コード	**説明**
13047	操作する特権がありません
13100	正しくない表モードの組み合わせ
13134	表が基本表ではありません
25038	表がパブリケーション publication_name で参照されていますが、ドロップ操作または変更操作は許されません
25039	表がパブリケーション publication_name に対するサブスクリプションで参照されていますが、ドロップ操作または変更操作は許されません

関連項目

ALTER TABLE ... SET HISTORY COLUMNS

ALTER TRIGGER

```
ALTER TRIGGER trigger_name_attr SET {ENABLED | DISABLED}
trigger_name_attr ::= [ catalog_name.[ schema_name. ]] trigger_name
```

使用法

ALTER TRIGGER ステートメントを使用して、トリガー属性を変更できます。有効な属性は、ENABLED および DISABLED トリガーです。

ALTER TRIGGER DISABLED ステートメントを実行すると、solidDB はアクティブ化 DML ステートメントが発行されたとき、トリガーを無視します。このコマンドを使用すると、現在アクティブでないトリガーを使用可能にするか、現在アクティブなトリガーを使用不可にすることもできます。

表に対するトリガーを変更するには、その表の所有者であるか、DBA 権限を持つユーザーであることが必要です。

例

```
ALTER TRIGGER trig_on_employee SET ENABLED;
```

ALTER USER

```
ALTER USER user_name IDENTIFIED BY password
```

使用法

ユーザーのパスワードは、ALTER USER ステートメントによって変更できます。

例

```
ALTER USER MANAGER IDENTIFIED BY O2CPTG;
```

ALTER USER (レプリカ)

```
ALTER USER replica_user SET MASTER master_name USER user_specification
```

ここで、

```
user_specification ::= { master_user IDENTIFIED BY master_password | NONE}
```

```
ALTER USER user_name SET {PUBLIC | PRIVATE}
```

使用法

以下のステートメントは、レプリカ・ユーザー ID を指定されたマスター・ユーザー ID にマップするために使用されます。

```
ALTER USER replica_user SET MASTER master_name USER user_specification
```

ユーザー ID のマッピングは、マルチマスターまたは複数層同期環境にセキュリティーを実装するために使用されます。そのような環境では、同じユーザー名とパスワードを地理的に分散した別々のデータベースで維持することは困難です。そのため、マッピングが有効です。

DBA 権限または SYS_SYNC_ADMIN_ROLE を持つユーザーのみがユーザーをマップできます。マッピングを実装するには、管理者はマスター・ユーザー名とパスワードを知っている必要があります。マスター・ユーザー ID にマップされるのは、常にレプリカ・ユーザー ID であることに注意してください。NONE を指定すると、マッピングは除去されます。

すべてのレプリカ・データベースは、ユーザー情報を更新するために、SYNC_CONFIG システム・パブリケーションにサブスクライブすることに責任を負います。このプロセスのとき、パブリック・マスター・ユーザー名およびパスワードは、MESSAGE APPEND SYNC_CONFIG コマンドを使用してレプリカ・データベースにダウンロードされます。レプリカ・ユーザー ID とマスター・ユーザー ID のマッピングにより、システムはレプリカ・データベースへログとして記録されたローカル・ユーザー ID に基づいて、現在アクティブなユーザーを判別します。SYNC_CONFIG ロード時にシステムがマッピングを検出しなかった場合、システムはユーザー ID およびパスワードをマスターとレプリカの中で突き合わせることにより、現在アクティブなマスター・ユーザーを判別します。

セキュリティーへのマッピングの使用について詳しくは、「*solidDB 拡張レプリケーション・ユーザー・ガイド*」の『アクセス権限およびロールによるセキュリティーの実装』をお読みください。

SYNC_CONFIG ロード時にレプリカへダウンロードされるマスター・ユーザーを、制限することもできます。これは、以下のコマンドでユーザーをプライベートまたはパブリックとして変更することによって行われます。

```
ALTER USER user_name SET PRIVATE | PUBLIC
```

デフォルトは PUBLIC であることに注意してください。ユーザーに PRIVATE オプションが設定されている場合、そのユーザーの情報は SYNC_CONFIG サブスクリプションに含まれません。これは、たとえ SYNC_CONFIG 要求の中でそれらが指定されている場合でも変わりません。DBA 権限または SYS_SYNC_ADMIN_ROLE を持つユーザーのみが、ユーザーの状況を変更できます。

これにより管理者は、管理権限を持つユーザー ID がレプリカへ送信されないようにすることができます。セキュリティー上の理由から、管理者は、例えば DBA パスワードが決してパブリックにならないようにすることもできます。

マスターでの使用

マスター・データベース内でユーザー ID を PUBLIC または PRIVATE に設定します。

レプリカでの使用

レプリカ・ユーザー ID をレプリカ・データベース内のマスター・ユーザー ID にマップします。

例

以下の例では、レプリカ・ユーザー ID *smith_1* を、dba のパスワードを持つマスター・ユーザー ID *dba* にマップします。

```
ALTER USER SMITH_1 SET MASTER MASTER_1 USER DBA IDENTIFIED BY DBA
```

以下の例は、ユーザーを PRIVATE および PUBLIC に設定する方法を示しています。

```
-- このマスター・ユーザーをレプリカにダウンロードしないでください
ALTER USER dba SET PRIVATE;

-- このマスター・ユーザーを、すべてのレプリカにダウンロードしてください。
ALTER USER salesman SET PUBLIC;
```

戻り値

各エラー・コードについて詳しくは、「*solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。

表 36. ALTER USER の戻り値

エラー・コード	説明
13047	操作する特権がありません
13060	ユーザー名 *xxx* が見つかりません。
25020	データベースがマスター・データベースではありません
25062	ユーザー *user_id* は、マスター *user_id* にマップされていません。
25063	ユーザー *user_id* は、既にマスター *user_id* にマップされています。

CALL

```
CALL procedure_name [( parameter [, parameter ...])] [AT node-def]
node-def ::= DEFAULT | <replica name> | <master name>
```

サポート条件

solidDB ディスク・ベース・エンジン、solidDB (リモート・プロシージャー・コールは、拡張レプリケーション・コンポーネントを備えた solidDB でのみ許可されることに注意してください。)

使用法

ストアード・プロシージャーは、ステートメント CALL で呼び出されます。

AT node_ref 節を使用することにより、別のノード上のストアード・プロシージャーを呼び出すことができます。これは、呼び出しがマスター・ノードからそのレプリカ・ノードの 1 つに対して行われるか、またはその逆の場合にのみ有効です。

DEFAULT は「現行レプリカ・コンテキスト」を使用することを意味します。「現行レプリカ・コンテキスト」は、プロシージャー呼び出しが START AFTER COMMIT ステートメントと FOR EACH REPLICA オプションを使用してバックグラウンドで開始されるときにのみ定義されます。デフォルトが設定されていない場合は、エラー「Default node not defined」が返されます。DEFAULT は、ストアード・プロシージャーの内部、および START AFTER COMMIT で始まるステートメント内で使用できます。

リモート・ストアード・プロシージャーは、結果セットを返すことができず、エラー・コードのみを返すことができます。

単一の呼び出しステートメントは、単一のノード上にある単一のプロシージャーのみを呼び出すことができます。単一のノード上にある複数のプロシージャーを呼び出したい場合は、複数の CALL ステートメントを実行する必要があります。複数のノード上で同じプロシージャー (つまり、同じプロシージャー名) を実行したい場合は、以下のいずれかを行う必要があります。

1) 次のコマンドを使用する

```
START AFTER COMMIT FOR EACH REPLICA.
```

例:

```
START AFTER COMMIT FOR EACH REPLICA WHERE NAME LIKE 'REPLICA%'
UNIQUE CALL MYPROC AT DEFAULT.
```

2) 呼び出しを複数回実行する

プロシージャー呼び出しは同期式に実行され、呼び出しが実行された後に戻りが行われます。

注: プロシージャー呼び出しは、START AFTER COMMIT (例: START AFTER COMMIT UNIQUE CALL FOO AT REPLICA1) を使用して実行された場合、バックグラウンドで非同期式に実行されます。これは START AFTER COMMIT コマンドの性質によるもので、プロシージャー呼び出しの性質によるものではありません。

トランザクション

リモート・プロシージャー・コールは、(START AFTER COMMIT によって開始されたかどうかにかかわらず)、呼び出し元だったトランザクションとは別のトランザクションで実行されます。呼び出し元は、リモート・プロシージャー・コールのロールバックまたはコミットを行うことはできません。呼び出されたノードで実行されているプロシージャーは、それ自身のコミットまたはロールバック・ステートメントを発行する責任があります。

リモート・プロシージャーからの戻り値

リモート・ストアード・プロシージャーを呼び出した場合、完全な結果セットを返させることはできません。取得できるのは、ストアード・プロシージャーの戻り値 (単一の値) かエラー・コードがすべてです。

注:

リモート・プロシージャーが (START AFTER COMMIT を使用して) バックグラウンドで実行された場合、ユーザーに返される戻り値はありません。エラー・コードさえ返されません。

リモート・ストアード・プロシージャー呼び出しのアクセス権限

ストアード・プロシージャーがリモート側で呼び出される場合は、アクセス権限、つまり、呼び出し元がリモート・サーバー上でそのプロシージャーを実行する権限を持っているかどうかを考慮に入れる必要があります。

ケース 1. sync ユーザーがコマンド SET SYNC USER で設定されている場合:

呼び出し元は「sync ユーザー」のユーザー名とパスワードをリモート・サーバーへ送信し、リモート・サーバーはそのユーザー名とパスワードを使用して、プロシージャーの実行を試みます。この場合、ユーザー名とパスワードがリモート・サーバー (そこでストアード・プロシージャーが実行されるサーバー) 内に存在する必要があり、そのユーザーはデータベースおよび呼び出されるプロシージャーに対して、適切なアクセス権限を持っている必要があります。

ケース 2. sync ユーザーが設定されていない場合:

呼び出し元は、リモート・プロシージャーを呼び出すとき、以下の情報をリモート・サーバーに送信します。

呼び出し元がマスターであり、リモート・サーバーがレプリカである場合 (M → R):

- マスターの名前 (SYS_SYNC_REPLICAS.MASTER_NAME)。
- レプリカ ID (SYS_SYNC_REPLICAS.ID)。
- 呼び出し元のユーザー名。
- 呼び出し元のユーザー ID。

呼び出し元がレプリカであり、リモート・プロシージャーがマスターである場合 (R → M):

- マスターの名前 (SYS_SYNC_MASTERS.NAME)。
- レプリカ ID (SYS_SYNC_MASTERS.REPLICA_ID)。
- マスター・ユーザー ID (同じユーザー ID がレプリカがデータをリフレッシュするときに使用されます。SYS_SYNC_USERS 表にローカル・レプリカ・ユーザーからマスター・ユーザーへのマッピングが存在する必要があります。)

以下のアクションは、呼び出されたノードで実行されます。

リモート・ノードがレプリカである場合 (M → R):

- 呼び出し元から受信したマスター名に従って、表 SYS_SYNC_MASTERS からマスター ID を取得します (マスター自体はレプリカ内のその ID を知りません)。表 SYS_SYNC_USERMAPS から、マスター・ユーザー名およびマスター ID に従ってレプリカ・ユーザー ID を取得します。プロシージャーへのアクセス権限を持つ最初のユーザーを選択します。

- SYS_SYNC_USERMAPS 内に一致する行がない場合は、呼び出し元から受信したマスター ID とマスター・ユーザー名に従って、表 SYS_SYNC_USERS から NAME と PASSWD を取得し、それらを使用してプロシージャーの実行を試みます。

リモート・ノードがマスターである場合 (R → M):

- レプリカから受信したユーザー ID を使用してプロシージャーの実行を試みます。

レプリカは、すべてのマスターからの呼び出しを許可する場合、solid.ini ファイルの中で独自の接続ストリング情報を定義する必要があります。以下に例を示します。

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

レプリカは、マスターにどのようなメッセージを転送するときでも、レプリカの接続ストリングを自動的にマスターへ送信します。マスターはレプリカから接続ストリングを受信すると、以前の値を置き換えます (値が異なっている場合)。

マスターは、以下のステートメントを使用して、レプリカへの接続ストリングを設定できます (レプリカがまだどのようなメッセージングも実行したことがなく、マスターがレプリカを呼び出して、接続ストリングが変更されたことを知る必要がある場合)。

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

持続性

リモート・プロシージャー・コールには持続性がありません。サーバーがリモート・プロシージャー・コールの発行直後にダウンすると、コールは失われます。そのコールはリカバリー・フェーズで実行されません。

例

```
CALL proctest;
CALL proctest('some string', 14);
CALL remote_proc AT replica2;
CALL RemoteProc(?,?) AT MyReplica1;
```

COMMIT WORK

```
COMMIT WORK
```

使用法

データベース内で行われた変更は、COMMIT ステートメントによって永続的な変更にされます。これによって、トランザクションが終了します。変更を廃棄するには、ROLLBACK コマンドを使用します。トランザクションを明示的に COMMIT しなかった場合、およびプログラム (例えば solsql) がユーザーに代わって COMMIT しなかった場合、そのトランザクションはロールバックされます。

例

```
COMMIT WORK;
```

関連項目

ROLLBACK WORK

CREATE CATALOG

```
CREATE CATALOG catalog_name
```

使用法

カタログを使用すると、データベースを論理的に区分することができるため、ビジネスやアプリケーションの必要に合わせてデータを編成できます。solidDB でのカタログの使用は、SQL 標準を拡張したものです。

solidDB 物理データベース・ファイルには、複数の*論理*データベース が含まれている場合があります。 それぞれの論理データベースは、表、索引、トリガー、ストアード・プロシージャーなど、独立した完全なデータベース・オブジェクト・グループです。それぞれの論理データベースは、データベース・カタログとして実装されます。このため、solidDB は 1 つ以上のカタログを持つことができます。

新規データベースを作成するとき、または古いデータベースを新しいフォーマットに変換するときは、デフォルトのカタログ名を入力するプロンプトが出ます。このデフォルト・カタログ名には、バージョン 3.x より前の solidDB データベースの後方互換性が考慮されています。

カタログは、ゼロ個以上の schema_names を持つことができます。デフォルトのスキーマ名は、そのカタログを作成するユーザーのユーザー ID です。

スキーマは、ゼロ個以上のデータベース・オブジェクト名を持つことができます。データベース・オブジェクトは、スキーマまたはユーザー ID で修飾できます。

カタログ名は、データベース・オブジェクト名を修飾するために使用されます。

注意:

カタログ名にはスペースを入れないようにしてください。

データベース・オブジェクト名はすべての DML ステートメント内で、以下のように修飾することができます。

```
catalog_name.schema_name.database_object
```

または

```
catalog_name.user_id.database_object
```

カタログ名を使用する場合は、スキーマ名も使用する必要があることに注意してください。その逆は真ではありません。スキーマ名は、カタログ名を使用しなくても使用できます (デフォルト・カタログを指定するために、適切な SET CATALOG ステートメントを既に実行してある場合)。

```
catalog_name.database_object -- 正しくない
schema_name.database_object -- 正しい
```

DBA 権限 (SYS_ADMIN_ROLE) を持つユーザーのみが、データベースのカタログを作成できます。

カタログを作成しても、そのカタログが自動的に現行のデフォルト・カタログになるわけではないことに注意してください。新規カタログを作成し、そのカタログ内で後続のコマンドを実行したい場合は、SET CATALOG ステートメントも実行する必要があります。以下に例を示します。

```
CREATE CATALOG MyCatalog;
CREATE SCHEMA smith; -- MyCatalog 内ではない
SET CATALOG MyCatalog;
CREATE SCHEMA jones; -- MyCatalog 内
```

SET CATALOG について詳しくは、 310 ページの『SET』でコマンド「SET」の説明を参照してください。

スキーマを使用するには、データベース・オブジェクト名を作成する前に、スキーマ名を作成する必要があります。ただし、データベース・オブジェクト名は、スキーマ名がなくても作成できます。そのような場合、データベース・オブジェクトは user_id だけを使用して修飾されます。スキーマの作成について詳しくは、 221 ページの『CREATE SCHEMA』をお読みください。

プログラム内でカタログ・コンテキストを設定するには、以下を使用します。

SET CATALOG *catalog_name*

カタログをデータベースからドロップするには、以下を使用します。

DROP CATALOG *catalog_name*

カタログ名をドロップするときは、事前にそのカタログ名に関連するすべてのオブジェクトをドロップしておく必要があります。

カタログ名を解決するためのルールを以下に示します。

- 完全修飾名 (*catalog_name.schema_name.database_object_name*)は、ネーム解決を何も必要としませんが、検証されます。
- SET CATALOG を使用してカタログ・コンテキストが設定されなかった場合、すべてのデータベース・オブジェクト名は、デフォルトのカタログ名をカタログ名として使用して解決されます。データベース・オブジェクト名は、スキーマ名の解決ルールを使用して解決されます。これらのルールについて詳しくは、 221 ページの『CREATE SCHEMA』をお読みください。
- カタログ・コンテキストが設定されており、そのコンテキスト内で *catalog_name* を使用してカタログ名を解決できない場合、*database_object_name* 解決は失敗します。
- データベース・システム・カタログにアクセスするために、ユーザーがシステム・カタログ名を知っている必要はありません。ユーザーは、″″._SYSTEM.table″ を指定できます。solidDB はカタログ名として使用された空ストリング ″ を、デフォルトのカタログ名に変換します。また、solidDB は、カタログ名が提供されない場合でも、_SYSTEM スキーマのシステム・カタログへの自動解決を行います。

例

```
CREATE CATALOG C;
SET CATALOG C;
CREATE SCHEMA S;
SET SCHEMA S;
CREATE TABLE T (i INTEGER);
SELECT * FROM T;
-- 名前 T は C.S.T へ解決されます。

-- ユーザー ID は SMITH であると想定します。
CREATE CATALOG C;
SET CATALOG C;
CREATE TABLE T (i INTEGER);
SELECT * FROM T;
-- 名前 T は C.SMITH.T へ解決されます。

-- 設定されているカタログ・コンテキストはないと想定します。
-- デフォルトのカタログ名は BASE であるか、
-- ベース・カタログの設定であることを意味します。
CREATE SCHEMA S;
SET SCHEMA S;
CREATE TABLE T (i INTEGER);
SELECT * FROM T;
-- 名前 T は <BASE>.S.T へ解決されます。

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;
-- この select は、エラーになります。
-- T1 を解決できないためです。
```

CREATE EVENT

```
CREATE EVENT event_name [( parameter_definition
    [,parameter_definition ...])]
```

使用法

イベント・アラートは、データベース内のイベントをシグナルで通知するために使用されます。イベントは、名前を持つ単純なオブジェクトです。アプリケーションは、より多くのリソースを使用するポーリングの代わりに、イベント・アラートを使用できます。

イベント・オブジェクトを作成するには、以下の SQL ステートメントを使用します。

```
CREATE EVENT event_name [parameter_list]
```

名前は、ユーザー指定の任意の英数字ストリングとすることができます。パラメーター・リストは、パラメーター名とパラメーター・タイプを指定します。パラメーター・タイプは、通常の SQL タイプです。

イベントをドロップするには、以下の SQL ステートメントを使用します。

```
DROP EVENT event_name
```

イベントは、ストアード・プロシージャーの内部で送受信されます。イベントの送受信には、特殊なストアード・プロシージャー・ステートメントが使用されます。

イベントを送信するには、以下のストアード・プロシージャー・ステートメントを使用します。

```
post_statement ::= POST EVENT event_name
    [( parameters) ] [UNIQUE | DATA UNIQUE]
```

イベント・パラメーターは、イベントの送信元ストアード・プロシージャー内のローカル変数、定数値、または、パラメーターでなければなりません。

キーワード UNIQUE は、各ユーザーおよび各イベントの最後の通知だけをイベント・キューに保持することを意味します。例えば、POST EVENT EV(1) および POST EVENT EV(2) の後では、EV(2) だけがイベント・キュー内に存在します (EV(2) が通知される前に EV(1) が処理されない場合)。イベント EV(1) は廃棄されます。キーワード DATA UNIQUE は、イベント・パラメーターも固有でなければならないことを意味します。したがって、呼び出し POST EVENT EV(1)、POST EVENT EV(2)、および POST EVENT EV(2) の後、イベント EV(1) と EV(2) がイベント・キュー内に保持されます。最初の EV(2) は廃棄されます。

通知されたイベントを待っているすべてのクライアントは、そのイベントを受信します。それぞれの接続は、独自のイベント・キューを備えています。イベント・キュー内に収集されるイベントを指定するには、以下のストアード・プロシージャー・ステートメントを使用します。

```
wait_register_statement ::= REGISTER EVENT event_name
```

イベントをイベント・キューから除去するには、以下のストアード・プロシージャー・ステートメントを使用します。

```
wait_register_statement ::= UNREGISTER EVENT event_name
```

イベントを待つには、事前にそのイベントについて登録しなくてもよいことに注意してください。イベントを待つ場合、まだそのイベントについて明示的に登録していない場合は、暗黙に登録されます。このため、明示的にイベントを登録する必要があるのは、それらのイベントのキューをその時点で開始したいが、それらのイベントの WAIT は後刻まで開始したくないという場合だけです。

プロシージャーにイベントの発生を待たせるには、ストアード・プロシージャーの中で、以下のような WAIT EVENT 構文を使用します。

```
wait_event_statement ::=
   WAIT EVENT
      [event_specification ...]
   END WAIT
```

```
event_specification ::=
   WHEN event_name [(parameters)] BEGIN
      statements
   END EVENT
```

それぞれの接続は、独自のイベント・キューを備えています。イベント・キュー内に収集するイベントを指定するには、コマンド REGISTER EVENT *event_name* を指定します。イベントをイベント・キューから除去するには、コマンド UNREGISTER EVENT *event_name* を使用します。

```
"CREATE PROCEDURE register_event
begin
        register event test_event
end";

"CREATE PROCEDURE unregister_event
begin
        unregister event test_event
end";
```

イベントの作成者またはデータベース管理者は、そのイベントに対するアクセス権限を付与および取り消すことができます。アクセス権限は、ユーザーおよびロールに付与できます。イベントに対する「SELECT」アクセス権限を持つユーザーは、そのイベントを待つ権限を持ちます。イベントに対する「INSERT」アクセス権限を持つユーザーは、そのイベントを通知できます。

ストアード・プロシージャーがイベントを待つのを停止させたい場合は、クライアント・アプリケーション内の別のスレッドから呼び出した ODBC 関数 SQLCancel() を使用できます。この関数は、ステートメントの実行を取り消します。あるいは、具体的なユーザー・イベントを作成し、それを送信することもできます。待つ側のストアード・プロシージャーを変更して、その追加イベントを待つようにする必要があります。クライアント・アプリケーションはそのイベントを認識し、待ちのループから出ます。

イベントの使用の詳細な例については、 97 ページの『イベントの使用』のセクションを参照してください。この例には、2 つの SQL スクリプトが含まれており、それらを一緒に使用すると、複数のイベントを通知し、待つことができます。

例

```
CREATE EVENT ALERT1(I INTEGER, C CHAR(4));
```

関連項目

CREATE PROCEDURE

CREATE INDEX

```
CREATE [UNIQUE] INDEX index_name
    ON base_table_name
      (column_identifier [ASC | DESC]
     [, column_identifier [ASC | DESC]] ...)
```

使用法

与えられた列に基づいて、表の索引を作成します。

キーワード UNIQUE は、索引を付ける列 (単数または複数) に固有値が含まれている必要があることを指定します。複数の列を指定する場合は、列の組み合わせが固有値を持っている必要がありますが、個々の列が固有値を持っている必要はありません。例えば、LAST_NAME と FIRST_NAME の組み合わせに対して索引を作成する場合、以下のデータ値が受け入れられます。なぜなら、重複する姓と重複する名がありますが、姓と名の両方が同じ値を持つ 2 つの行は存在しないからです。

```
SMITH, PATTI
SMITH, DAVID
JONES, DAVID
```

キーワード ASC および DESC は、与えられた列に昇順と降順のどちらで索引を付けるかを指定します。ASC と DESC をどちらも指定しなかった場合は、昇順が使用されます。

例

```
CREATE UNIQUE INDEX UX_TEST ON TEST (I);
CREATE INDEX X_TEST ON TEST (I DESC, J DESC);
```

関連項目

216 ページの『CREATE [OR REPLACE] PUBLICATION』.

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
```

使用法

ストアード・プロシージャーは、サーバー内で実行される単純なプログラム、つまりプロシージャーです。ユーザーは複数の SQL ステートメントまたはトランザクション全体が入ったプロシージャーを作成し、それを単一の呼び出しステートメントで実行できます。ストアード・プロシージャーを使用すると、ネットワーク・トラフィックを減らし、アクセス権限およびデータベース操作に対して、より厳格な制御を行うことができます。

プロシージャーを作成するには、以下のステートメントを使用します。

```
CREATE PROCEDURE name body
```

また、ステートメントをドロップするには、以下のステートメントを使用します。

```
DROP PROCEDURE name
```

プロシージャーを呼び出すには、以下のステートメントを使用します。

```
CALL name [parameter ...]
```

すべての SQL ストアード・プロシージャーは、プロシージャー宣言の中で SQL 標準文節 *SQL Data Access Indication* によって読み取り専用プロシージャーとして指定された場合を除き、1 次サーバーで実行されます。

```
<SQL-data-access-indication> ::=
     NO SQL |
     READS SQL DATA |
     CONTAINS SQL |
     MODIFIES SQL DATA
```

読み取り専用プロシージャーおよび関数の不必要な引き渡しを回避するために、以下のいずれかの値を宣言できます。

- NO SQL
- READS SQL DATA
- CONTAINS SQL

MODIFIES SQL DATA (これはデフォルトです) のみがトランザクションの引き渡しを発生させます。

この文節は、(オプションの) RETURNS 文節とプロシージャー本体の間に置かれます。以下に例を示します。

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

ストアード・プロシージャーには、入力パラメーター、出力パラメーター、および入出力パラメーターという 3 つのパラメーター・モードがあります。以下で、各パラメーター・モードについて説明します。

1. 入力パラメーターは、呼び出し側プログラムからストアード・プロシージャーに渡されます。*parameter_mode* 値は IN です。これはデフォルトの動作です。
2. 出力パラメーターは、ストアード・プロシージャーから呼び出し側プログラムへ返されます。*parameter_mode* 値は OUT です。
3. 入出力パラメーターはプロシージャーに値を渡し、呼び出し側プロシージャーに値を返します。*parameter_mode* は INOUT です。

パラメーター・モードの比較については、以下の表を参照してください。

表 37. パラメーター・モードの比較

機能	IN	OUT	INOUT
デフォルト/指定	デフォルト。	指定する必要がある。	指定する必要がある。
操作	サブプログラムに値を渡す。	呼び出し元に値を返す。	サブプログラムに初期値を渡し、更新された値を呼び出し元に返す。
アクション	仮パラメーター。定数のように機能する。	仮パラメーター。初期化されていない変数のように機能する。	仮パラメーター。初期化された変数のように機能する。

表 37. パラメーター・モードの比較 (続き)

機能	IN	OUT	INOUT
値の割り当て	仮パラメーター。値を割り当てることができない。	仮パラメーター。式の中で使用できない。値を割り当てる必要がある。	仮パラメーター。値を割り当てる必要がある。
パラメーター・タイプ	実パラメーター。定数、初期化された変数、リテラル、式のいずれかにすることができる。	実パラメーター。変数でなければならない。	実パラメーター。変数でなければならない。

プログラミング・インターフェースでは、出力パラメーターは以下のように変数にバインドされます。

JDBC では、メソッド CallableStatement.registerOutParameter() を使用します。

ODBC では、関数 SQLBindParameter() を使用します。ここで、第 3 引数 InputOutputType は、以下のいずれかのタイプにすることができます。

SQL_PARAM_INPUT

SQL_PARAM_OUTPUT

SQL_PARAM_INPUT_OUTPUT

パラメーターの変数へのバインドについて詳しくは、「*solidDB プログラマー・ガイド*」を参照してください。

本体が空のストアード・プロシージャーを作成することは、有用ではありませんが、構文上では有効であることに注意してください。

プロシージャーは、プロシージャーの作成者によって所有されます。指定されたアクセス権限を、他のユーザーに付与できます。プロシージャーが実行される場合、そのプロシージャーはデータベース・オブジェクトに対して、作成者のアクセス権限を持ちます。

ストアード・プロシージャー構文は、SQL-99 仕様および動的 SQL をモデルとして作成された専有の構文です。プロシージャーには、制御ステートメントと SQL ステートメントが含まれます。

プロシージャー内では、以下の制御ステートメントを使用できます。

表 38. 制御ステートメント

制御ステートメント	説明
`set variable = expression`	変数に値を代入します。値は、リテラル値 (例えば、10 または「テキスト」) か別の変数にすることができます。パラメーターは、通常の変数と見なされます。
`variable ::= expression`	変数に値を代入するための代替構文。

表 38. 制御ステートメント (続き)

制御ステートメント	説明

一方
    expr
`loop`
    statement-list
`end loop`
	while 式が真である間、ループします。

`leave`	最も内側にあるループを出て、キーワード end loop の後にある次のステートメントからプロシージャーの実行を継続します。

`if`
    expr
`then`
    statement-list1
`else`
    statement-list2
`end if`
	式 *expr* が真であれば *statements-list1* を実行し、真でなければ *statement-list2* を実行します。

`if`
    expr1
`then`
    statement-list1
`elseif`
    expr2
`then`
    statement-list2
`end if`
	expr1 が真であれば、*statement-list1* を実行します。*expr2* が真であれば、*statement-list2* を実行します。このステートメントには、オプションとして複数の elseif ステートメントと 1 つの else ステートメントを含めることもできます。

`return`	出力パラメーターの現行値を返し、プロシージャーを終了します。プロシージャーに *return row* ステートメントが存在する場合、return は *return norow* のように動作します。
`return sqlerror of` *cursor-name*	カーソルに関連した sqlerror を返し、プロシージャーを終了します。
`return row`	出力パラメーターの現行値を返し、プロシージャーの実行を継続します。return row はプロシージャーを終了せず、呼び出し元に制御を返します。
`return norow`	セットの終わりを返し、プロシージャーを終了します。

すべての SQL DML および DDL ステートメントは、プロシージャー内で使用できます。このため、例えば、プロシージャーで表を作成したり、トランザクションをコミットしたりすることができます。プロシージャー内の各 SQL ステートメントはアトミックです。

「自動コミット」機能は、ストアード・プロシージャーの内部にあるステートメントの場合と、ストアード・プロシージャーの外部にあるステートメントの場合で機能が異なります。ストアード・プロシージャーの外部にある SQL ステートメントでは、自動コミットがオンの場合、個々のステートメントの直後に、暗黙の COMMIT WORK 操作が実行されます。しかし、ストアード・プロシージャーの場合、暗黙の COMMIT WORK はストアード・プロシージャーが呼び出し元に戻った後に実行されます。これは、ストアード・プロシージャーが「アトミック」でないことを意味する点に注意してください。上記のように、ストアード・プロシージャーは、独自の COMMIT コマンドおよび ROLLBACK コマンドを含んでいる場合が

あります。プロシージャーの戻りの後に実行される暗黙の COMMIT WORK は、下記以降に実行されたストアード・プロシージャー・ステートメントの部分だけをコミットします。

- プロシージャー内部の最後の COMMIT WORK
- プロシージャー内部の最後の ROLLBACK WORK
- プロシージャーの開始 (COMMIT または ROLLBACK コマンドがプロシージャーで実行されなかった場合)

ストアード・プロシージャーが別のストアード・プロシージャーの内部から呼び出された場合、最も外側のプロシージャー呼び出しが終わった後にのみ、暗黙の COMMIT WORK が実行されることに注意してください。「ネスト」されたプロシージャー呼び出しの後に実行される暗黙の COMMIT WORK はありません。

例えば、以下のスクリプトでは、暗黙の COMMIT WORK は CALL outer_proc(); ステートメントの後にのみ実行されます。

```
"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();
```

SQL ステートメントの準備

SQL ステートメントは、最初に、以下のステートメントで準備されます。

EXEC SQL PREPARE *cursor sql_statement*

cursor の指定はカーソル名で、これは必ず指定する必要があります。そのトランザクションの内部で固有のカーソル名であれば、何でもかまいません。プロシージャーが完全なトランザクションでない場合は、プロシージャーの外部にある別のオープン・カーソルが、競合するカーソル名を持っている可能性があることに注意してください。

準備済み SQL ステートメントの実行

SQL ステートメント を実行するには、以下のステートメントを使用します。

EXEC SQL EXECUTE *cursor* [*opt_using*] [*opt_into*]

オプションの *opt-using* を指定する構文は、以下のとおりです。

USING (*variable_list*)

ここで、*variable_list* には、プロシージャーの変数またはパラメーターをコンマで区切ったリストが入っています。それらの変数は SQL ステートメントの入力パラメーターです。SQL 入力パラメーターは、prepare ステートメント内で標準的な疑問符 (?) 構文を使用してマーク付けされます。SQL ステートメントに入力パラメーターがない場合、USING の指定は無視されます。

オプションの *opt_into* を指定する構文は、以下のとおりです。

```
INTO (variable_list)
```

ここで、*variable_list* には、SQL SELECT ステートメントの列値の格納先となる変数が入っています。INTO の指定は、SQL SELECT ステートメントにのみ効果があります。

UPDATE、INSERT、および DELETE ステートメントの実行後、追加変数を使用して、ステートメントの結果を検査できます。変数 SQLROWCOUNT には、最後のステートメントの影響を受けた行の数が格納されています。

結果のフェッチ

行をフェッチするには、以下のステートメントを使用します。

```
EXEC SQL FETCH cursor_name
```

フェッチが正常に完了した場合、列値は EXECUTE または EXECDIRECT ステートメントの *opt_into* の指定で定義された変数に格納されます。

カーソルのクローズとドロップ

カーソルの使用が終了したら、カーソルを CLOSE し、カーソルを DROP してください。これを行わなかった場合、カーソルに割り当てられているリソース (メモリーなど) を再利用のために解放できない場合があります。

エラーの検査

プロシージャー本体の内部で実行された各 EXEC SQL ステートメントの結果は、変数 SQLSUCCESS の中に保管されます。この変数は、すべてのプロシージャーで自動的に生成されます。前の SQL ステートメントが成功した場合は、値 1 が SQLSUCCESS に格納されます。失敗した SQL ステートメントの後では、値ゼロが SQLSUCCESS に格納されます。

```
EXEC SQL WHENEVER SQLERROR {ABORT | [ROLLBACK [WORK], ABORT}
```

このステートメントは、プロシージャー内の SQL ステートメントを実行した後に、毎回 IF NOT SQLSUCCESS THEN テストを行わなくても済むようにするために使用されます。このステートメントをストアード・プロシージャーに組み込んだ場合、実行されたステートメントのすべての戻り値は、エラーがないかどうか検査されます。ステートメントの実行でエラーが返されると、プロシージャーは自動的に打ち切られます。オプションとして、トランザクションをロールバックできます。

失敗した最新の SQL ステートメントのエラー・ストリングは、変数 SQLERRSTR に格納されます。

トランザクションの使用

```
EXEC SQL {COMMIT | ROLLBACK} WORK
```

は、トランザクションを終了するために使用します。

```
EXEC SQL SET TRANSACTION {READ ONLY | READ WRITE}
```

は、トランザクションのタイプを制御するために使用します。

シーケンサー・オブジェクトおよびイベント・アラートの使用

CREATE SEQUENCE ステートメントおよび CREATE EVENT ステートメントの使用法を参照してください。

writetrace

writetrace() 関数を使用して、soltrace.out トレース・ファイルにストリングを送信できます。これは、ストアード・プロシージャーの問題をデバッグするときに役に立ちます。

出力は、トレースをオンにしているときにだけ書き込まれます。

writetrace について、およびトレースをオンにする方法について詳しくは、154 ページの『ストアード・プロシージャーおよびトリガーのトレース機能』を参照してください。

プロシージャー・スタック関数

以下の関数は、プロシージャー・スタックの現在の内容を分析するために使用できます。 PROC_COUNT()、 PROC_NAME(N)、 PROC_SCHEMA(N)。

PROC_COUNT() は、プロシージャー・スタック内のプロシージャーの数を返します。これには、現行プロシージャーも含まれます。

PROC_NAME(N) は、スタック内の N 番目のプロシージャー名を返します。最初のプロシージャー位置はゼロです。

PROC_SCHEMA(N) は、プロシージャー・スタック内の N 番目のプロシージャーのスキーマ名を返します。

動的カーソル名

```
CURSORNAME(
    prefix -- VARCHAR
)
```

CURSORNAME() 関数を使用すると、カーソル名をハードコーディングせずに、動的に生成できます。

注:

厳密に言うと、CURSORNAME() は (構文はよく似ていますが) 関数ではありません。CURSORNAME(arg) は、実際には何も返しません。その代わりに、現行ステートメントのカーソルの名前を、指定された引数に基づいて設定します。しかし、ここでは便宜上、関数と呼ぶことにします。

カーソル名は、1 つの接続内で固有であることが必要です。このため再帰的ストアード・プロシージャーでは、それぞれの呼び出しが同じカーソル名を使用するので、問題が発生します。再帰的プロシージャーがそれ自体を呼び出した場合、2 番目の呼び出しは、2 番目の呼び出しで使用したい名前と同じ名前のカーソルが、既に最初の呼び出しで作成されていることを発見します。

この問題を回避するためには、カーソルを宣言して使用するときに、固有のカーソル名を動的に生成して使用できなければなりません。固有の名前を生成し、カーソルとして使用できるようにするために、ここでは 2 つの関数を使用します。

- GET_UNIQUE_STRING
- CURSORNAME

GET_UNIQUE_STRING 関数は、その名前が示すとおり、固有のストリングを生成します。CURSORNAME 関数 (実際には疑似関数) を使用すると、動的に生成されたストリングをカーソル名の一部として使用できます。

GET_UNIQUE_STRING は、たとえ入力が同じであっても、呼び出されるたびに異なる出力を返すことに注意してください。一方、CURSORNAME は、入力が毎回同じであれば、毎回同じ出力を返します。

以下に、GET_UNIQUE_STRING と CURSORNAME を使用してカーソル名を生成する例を示します。動的に生成されたカーソル名はプレースホルダー「cname」に代入され、その後、このプレースホルダーが PREPARE の後に各ステートメント内で使用されます。

```
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() は、PREPARE ステートメントと EXECDIRECT ステートメントの中でのみ使用されます。EXECUTE、FETCH、CLOSE、DROP などの中で使用することはできません。

CURSORNAME() フィーチャーと GET_UNIQUE_STRING() 関数を使用することにより、再帰的ストアード・プロシージャーの中で固有のカーソル名を生成できます。プロシージャーがそれ自体を呼び出す場合、ストアード・プロシージャー内でこの関数が呼び出されるたびに、この関数は固有のストリングを返し、そのストリングは PREPARE ステートメント内でカーソル名として使用できます。ストアード・プロシージャーの内部で使用できるコードのいくつかの例については、以下を参照してください。

入力 (autoname) が変わらない限り、CURSORNAME(autoname) を呼び出すたびに、同じ値 (つまり同じカーソル名) が返されることに注意してください。

EXECDIRECT

EXECDIRECT ステートメントを使用すると、ストアード・プロシージャーの内部で、事前に「準備」していないステートメントを実行できます。これにより、必要なコードの量が少なくなります。ステートメントがカーソルの場合でも、クローズとドロップが必要なことに注意してください。PREPARE ステートメントだけをスキップできます。

以下を使用する場合

```
EXEC SQL [USING(var_list)] [CURSORNAME(variable)]
EXECDIRECT <statement>
```

または

```
EXEC SQL <cursor_name> [USING(var_list)] [INTO (var_list)]
[CURSORNAME(variable)] EXECDIRECT <statement>
```

以下のルールに留意してください。

- ステートメントでカーソル名を指定した場合は、そのカーソルを EXEC SQL DROP ステートメントでドロップする必要があります。
- カーソル名を指定しなかった場合は、ステートメントをドロップする必要はありません。
- ステートメントがフェッチ・カーソルの場合は、INTO... 節を指定する必要があります。
- INTO 節を指定する場合は、cursor_name を指定する必要があります。指定しなかった場合、FETCH ステートメントで、どのカーソル名から行をフェッチするのかを指定できなくなります。(一度に複数のカーソルをオープンしている場合があります。)

以下に、いくつかの CREATE PROCEDURE ステートメントの例を示します。PREPARE コマンドと EXECUTE コマンドを使用するものと、EXECDIRECT を使用するものがあります。

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";
```

明示的な RETURN ステートメントの使用

この例では、明示的な RETURN ステートメントを使用して、複数の行を一度に 1 つずつ返します。

```
"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";
```

EXECDIRECT の使用

```
-- この例では、execdirect の使用方法を示します。
"CREATE PROCEDURE p
BEGIN
    DECLARE host_x INT;
    DECLARE host_y INT;

    -- カーソルがない execdirect の例。ここでは、表を作成し、
    -- その表に行を挿入します。
    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;

    -- カーソル名がある execdirect の例。
    -- この例では、c1 はカーソル名です。host_x は
    -- 「?」の代わりに使用される値を持つ変数です。;
    -- host_y も変数で、この中に
    -- 列 y の値を (フェッチしたときに) 保管します。
    -- 注: prepare ステートメントは必要ありませんが、
    -- close/drop は必要です。
    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";
```

CURSORNAME の使用

この例では、CURSORNAME() 疑似関数の使用法を示します。これは完全なストアード・プロシージャーでなく、ストアード・プロシージャーの本体部分だけを示しています。

```
-- カーソル名として使用できる固有のストリングを保持する
-- 変数を宣言します。
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;
```

GET_UNIQUE_STRING と CURSORNAME の使用

ここでは、実際に再帰的ストアード・プロシージャーの中で GET_UNIQUE_STRING 関数と CURSORNAME 関数を使用する、より完全な例を示します。

下記のストアード・プロシージャーは、これら 2 つの関数を再帰的プロシージャーの中で使用する実例です。カーソル名「curs1」はハードコーディングされているように見えますが、実際には動的に生成された名前にマップされていることに注意してください。

```
-- GET_UNIQUE_STRING 関数と CURSORNAME 関数を
-- 再帰的ストアード・プロシージャーの中で使用するデモ。
-- 数値 N が 1 以上である場合、このプロシージャーは
-- 数値 1 から N までの合計を返します。(これをループ内でも
-- 実行できますが、この例の目的は、再帰的プロシージャーの中での
-- CURSORNAME 関数の使用を示すことです。)
"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
        Autoname := GET_UNIQUE_STRING('CURSOR_NAME_PREFIX_') ;
        EXEC SQL PREPARE curs1 CURSORNAME(autoname) 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";
```

例 6

CREATE PROCEDURE での EXECDIRECT の使用

```
CREATE TABLE table1 (x INT, y INT);
INSERT INTO table1 (x, y) VALUES (1, 2);

"CREATE PROCEDURE FOO
RETURNS (r INT)
BEGIN
DECLARE autoname VARCHAR;
Autoname := GET_UNIQUE_STRING('CUR_');
EXEC SQL curs_name INTO(r) CURSORNAME(autoname) 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;
```

同期メッセージ用の固有の名前の作成

同期メッセージ用の固有の名前の作成:

```
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;
```

GET_UNIQUE_STRING の使用

```
-- これは GET_UNIQUE_STRING() 関数を使用して、
-- 再帰的ストアード・プロシージャーの中から固有のメッセージ名を
-- 生成する方法を示す実例です。

CREATE TABLE table1 (i int, beginMsg VARCHAR, endMsg VARCHAR);

-- これは、再帰の簡単な例です。
```

```
-- ここで作成するメッセージは実用でないことに注意してください。これは
-- 同期化の真の例ではありません。単に、固有のメッセージ名を生成する
-- 例にすぎません。「count」パラメーターは、
-- この関数がそれ自体を呼び出す回数です
-- (初回の呼び出しは含まれません)。
"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;

-- SQL ステートメントをストリングとして作成した後、
-- それを以下の 2 つの方法のいずれかで実行できます。
-- 1) EXECDIRECT フィーチャーを使用する。または、
-- 2) SQL ステートメントを準備し、実行する。
-- この例では、EXECDIRECT を使用します。
EXEC SQL EXECDIRECT MsgBeginStr;
EXEC SQL EXECDIRECT MsgEndStr;
-- ここで、何か実用的なことを実行します。

-- 関数の再帰的な部分。
IF (count > 1) THEN
    SET count = count - 1;
    -- 固有の名前をカーソル名としても使用できることに注意してください。
    -- 以下に示すとおりです。
    EXEC SQL Autoname USING (count) EXECDIRECT CALL repeater(?);
    EXEC SQL CLOSE Autoname;
    EXEC SQL DROP Autoname;
END IF

RETURN;
END";

CALL repeater(3);

-- 作成したメッセージ名を示します。
SELECT * FROM table1;
```

この SELECT ステートメントからの出力は、以下のようになります。

```
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
```

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
```

注: *Search_condition* は、*parameter_definitions* や、前の (より高い) レベルで定義されたレプリカ表の列を参照できます。

使用法

パブリケーションは、マスターからレプリカ・データベースにリフレッシュできるデータのセットを定義します。パブリケーションは、常に、トランザクションに整合性があります。つまり、1 つのトランザクションでマスター・データベースからデータを読み取って、1 つのトランザクションでレプリカ・データベースにそのデータを書き込みます。

注意:

マスターが READ COMMITTED 分離レベルを使用している場合を除き、パブリケーションから読み取られたデータは、内部的に整合性があります。

SELECT 節の検索条件には、パブリケーションの入力引数をパラメーターとして含めることができます。パラメーター名には、接頭部としてコロンが必要です。

パブリケーションには、複数の表のデータを含めることができます。パブリケーションの表は独立していることも、表の間に関係を持つこともできます。表の間に関係がある場合、結果セットはネストする必要があります。パブリケーションの内側の結果セットに対する SELECT ステートメントの WHERE 節は、外側の結果セットの表の列を参照する必要があります。

パブリケーションの外側と内側の結果セットの関係が、N-1 の関係である場合は、結果セット定義でキーワード DISTINCT を使用する必要があります。

replica_table_name は、 *master_table_name* と異なっていてもかまいません。パブリケーション定義は、マスター表とレプリカ表のマッピングを提供します。(マスターで使用される名前と異なる名前を使用する場合でも、複数のレプリカがある場合は、すべてのレプリカが同じ名前を使用する必要があります。) マスター表とレプリカ表の列名は、同じである必要があります。

初期ダウンロードは常に、フル・パブリケーション であることに注意してください。つまり、パブリケーションに含まれているすべてのデータがレプリカ・データ

ベースに送信されます。同じパブリケーションの後続のダウンロード (リフレッシュ) は、前のリフレッシュ以降に変更されたデータだけが含まれるインクリメンタル・パブリケーション にできます。インクリメンタル・パブリケーションの使用を有効にするには、マスター・データベースとレプリカ・データベースの両方で、パブリケーションに含まれる表の SYNCHISTORY を ON に設定する必要があります。詳しくは、 191 ページの『ALTER TABLE ... SET SYNCHISTORY』および 244 ページの『DROP PUBLICATION REGISTRATION』を参照してください。

オプション・キーワードの「OR REPLACE」を使用した場合に、パブリケーションが既に存在すると、それは新しい定義に置き換えられます。パブリケーションをドロップして再作成したわけではないので、レプリカを再登録する必要はありません。パブリケーションに対して行われた変更に厳密に応じて、パブリケーションからの後続のリフレッシュをフルではなくインクリメンタルにできます。

パブリケーションを更新している間に、レプリカがそのパブリケーションからリフレッシュすることを防ぐために、カタログの同期モードを一時的に保守モードに設定できます。ただし、パブリケーションを置き換えるときに、必ずしも保守モードを使用する必要はありません。

既存のパブリケーションを置き換えると、次にレプリカがリフレッシュを要求したときに、パブリケーションの新しい定義が各レプリカに送信されます。レプリカは、明示的にパブリケーションに再登録する必要はありません。

既存のパブリケーションを新しい定義で置き換えるときに、結果セット定義を変更できます。パブリケーションのパラメーターは変更できません。パラメーターを変更するには、パブリケーションをドロップして、新しいパブリケーションを作成する必要があります。つまり、レプリカを再登録する必要があり、次にリフレッシュを要求するときに、インクリメンタル・リフレッシュではなくフル・リフレッシュを取得します。

既存のパブリケーションを置き換えるときに、そのパブリケーションに関連する特権は変更されないままです。(再作成する必要はありません。)

CREATE PUBLICATION コマンドを実行できる状態であれば、CREATE OR REPLACE PUBLICATION コマンドを実行できます。

注意:

CREATE OR REPLACE PUBLICATION を使用して既存の拡張レプリケーション・パブリケーションの内容を変更する場合は、レプリカから無効な行を削除する必要があります。

マスターでの使用

マスター・データベースでパブリケーションを定義し、レプリカがこのパブリケーションからリフレッシュを取得できるようにします。

レプリカでの使用

レプリカでパブリケーションを定義する必要はありません。パブリケーション・サブスクリプション機能は、マスター・データベースの定義だけに依存します。この

コマンドをレプリカで実行した場合、パブリケーション定義はレプリカに保管されますが、このパブリケーション定義は何にも使われません。

注: データベースが (その上位のマスターに対する) レプリカと (その下位のレプリカに対する) マスターの両方である場合は、そのデータベースでパブリケーション定義を作成できます。

例

以下のサンプル・パブリケーションは、顧客の市外局番を検索基準に使用して、顧客表からデータをリトリーブします。顧客ごとに、顧客のオーダーと請求書 (1-N の関係)、および専門の販売員 (1-1 の関係) がリトリーブされます。

```
"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";
```

注:

:IN_AREA_CODE のコロン (:) は、同じ名前のパブリケーション・パラメーターの参照であることを示すために使用されています。

例 2:

開発者は、パブリケーション P で参照されている新しい列 C を表 T に追加することに決定しました。マスター・データベースとすべてのレプリカ・データベースを変更する必要があります。

マスター・データベースで実行するタスクは、以下のとおりです。

```
-- 他のユーザーが、このカタログに並行して同期操作を
-- 実行しないようにします。
SET SYNC MAINTENANCE MODE ON;
ALTER TABLE T ADD COLUMN C INTEGER;
COMMIT WORK;
CREATE OR REPLACE PUBLICATION P ...
(列 C もパブリケーションに追加されます)
COMMIT WORK;
SET SYNC MAINTENANCE MODE OFF;
```

すべてのレプリカ・データベースで実行するタスクは、以下のとおりです。

```
-- 他のユーザーが、このカタログに並行して同期操作を
-- 実行しないようにします。
SET SYNC MAINTENANCE MODE ON;
ALTER TABLE T ADD COLUMN C INTEGER;
COMMIT WORK;
SET SYNC MAINTENANCE MODE OFF;
```

戻り値

各エラー・コードについて詳しくは、「*solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。

表 39. CREATE PUBLICATION の戻り値

エラー・コード	説明
13047	操作する特権がありません。このパブリケーションのドロップまたはパブリケーションの作成に必要な特権がありません
13120	名前が、パブリケーションの名前として長すぎます
25015	構文エラー: *error_message*、行 *line_number*
25021	データベースがマスター・データベースまたはレプリカ・データベースではありません。パブリケーションは、マスター・データベースまたはレプリカ・データベースにだけ作成できます (実際には、マスター・データベースにだけ作成する必要があります)
25033	パブリケーション *publication_name* は、既に存在します
25049	参照される表 *table_name* が、サブスクリプション階層で見つかりません
25061	表 *table_name* の条件が、パブリケーションの外部表を参照する必要があります

CREATE [OR REPLACE] REMOTE SERVER

```
CREATE [OR REPLACE] REMOTE SERVER [USERNAME <username> PASSWORD <password>]
```

使用法

CREATE REMOTE SERVER ステートメントは、Universal Cache での SQL パススルーのために、SYS_SERVER システム表でバックエンド・ログイン情報を作成します。

CREATE OR REPLACE REMOTE SERVER ステートメントは、SYS_SERVER システム表のログイン情報を変更します。

solidDB とバックエンドの間の最初のサブスクリプションが作成されるときに、CDC for solidDB インスタンスは、バックエンド CDC インスタンスからバックエンド・データ・サーバーのログイン・データをリトリーブしてから、CREATE REMOTE SERVER ステートメントを使用して、これを solidDB システム表 SYS_SERVER に保管します。

通常の用途では、これらのステートメントをエンド・ユーザーが使用することはありません。

例

```
CREATE REMOTE SERVER [USERNAME dba PASSWORD dba]
```

関連項目

ALTER REMOTE SERVER

DROP REMOTE SERVER

CREATE ROLE

```
CREATE ROLE role_name
```

使用法

新しいユーザー・ロールを作成します。

例

```
CREATE ROLE GUEST_USERS;
```

CREATE SCHEMA

```
CREATE SCHEMA schema_name
```

使用法

スキーマは、データベース・ユーザー用のデータベース・オブジェクト (表、ビュー、索引、イベント、トリガー、シーケンス、ストアード・プロシージャーなど) の集合です。デフォルト・スキーマ名は、ユーザー ID です。各ユーザーに 1 つのデフォルト・スキーマがあることに注意してください。solidDB のスキーマ使用法は、SQL 標準に準拠します。

スキーマ名は、データベース・オブジェクト名を修飾するために使用されます。データベース・オブジェクト名は、すべての DML ステートメントで以下のように修飾されます。

catalog_name.schema_name.database_object_name

または

user_id.database_object_name

データベースを論理的にパーティション化するために、ユーザーは、スキーマを作成する前にカタログを作成できます。カタログの作成について詳しくは、『199 ページの『CREATE CATALOG』』を参照してください。新しいデータベースを作成するとき、または古いデータベースを新しいフォーマットに変換するときに、デフォルト・カタログ名の入力を促すプロンプトが出されることに注意してください。

スキーマを使用するには、データベース・オブジェクト名 (表名、プロシージャー名など) を作成する前に、スキーマ名を作成する必要があります。ただし、データ

ベース・オブジェクト名は、スキーマ名がなくても作成できます。そのような場合、データベース・オブジェクトは user_id だけを使用して修飾されます。

データベース・オブジェクト名は、DML ステートメントで、完全に修飾して明示的に指定することも、以下を使用してスキーマ名コンテキストを設定し、暗黙的に指定することもできます。

```
SET SCHEMA schema_name
```

スキーマを作成しても、自動的にそのスキーマが現行のデフォルト・スキーマになるわけではありません。新しいスキーマを作成し、後続のコマンドをそのスキーマで実行する場合は、SET SCHEMA ステートメントも実行する必要があります。以下に例を示します。

```
CREATE SCHEMA MySchema;
CREATE TABLE t1; -- MySchema ではありません。
SET SCHEMA MySchema;
CREATE TABLE t2; -- MySchema です。
```

SET SCHEMA について詳しくは、 310 ページの『SET』の SET SCHEMA コマンドの説明を参照してください。

スキーマは、以下を使用して、データベースからドロップできます。

```
DROP SCHEMA schema_name
```

スキーマ名をドロップするときは、スキーマをドロップする前に、そのスキーマ名に関連付けられているすべてのオブジェクトをドロップする必要があります。

スキーマ・コンテキストは、以下を使用して、削除できます。

```
SET SCHEMA USER
```

以下は、スキーマ名解決のルールです。

- 完全修飾名 (*schema_name.database_object_name*) は、ネーム解決の必要はありませんが、検証されます。
- SET SCHEMA でスキーマ・コンテキストが設定されていない場合、 すべてのデータベース・オブジェクト名が、常にスキーマ名としてユーザー ID を使用して解決されます。
- スキーマ名からデータベース・オブジェクト名を解決できない場合、 データベース・オブジェクト名はすべての既存のスキーマ名から解決されます。
- ネーム解決で、一致するデータベース・オブジェクト名がゼロ個または複数検出された場合、solidDB サーバーは、ネーム解決競合エラーを発行します。

例

```
-- ユーザー ID は SMITH であると想定します。
CREATE SCHEMA FINANCE;
CREATE TABLE EMPLOYEE (EMP_ID INTEGER);
SET SCHEMA FINANCE;
CREATE TABLE EMPLOYEE (ID INTEGER);
SELECT ID FROM EMPLOYEE;
-- この場合、表は FINANCE.EMPLOYEE に修飾されます。
SELECT EMP_ID FROM EMPLOYEE;
-- コンテキストに FINANCE があり、表が FINANCE.EMPLOYEE に
-- 解決されるため、エラーが発生します。
```

```
-- 以下は、有効なスキーマ・ステートメントです。一方には
-- スキーマ・コンテキストがあり、もう一方にはありません。
SELECT ID FROM FINANCE.EMPLOYEE;
SELECT EMP_ID FROM SMITH.EMPLOYEE
-- 以下のステートメントは、スキーマ・コンテキストなしの
-- スキーマ SMITH に解決されます。
SELECT EMP_ID FROM EMPLOYEE;
```

CREATE SEQUENCE

CREATE [DENSE] SEQUENCE *sequence_name*

使用法

シーケンサー・オブジェクトは、シーケンス番号の取得に使用されるオブジェクトです。

密シーケンスを使用すると、シーケンス番号のホールを防ぐことができます。シーケンス番号の割り振りは、現行トランザクションにバインドされます。トランザクションがロールバックされると、シーケンス番号割り振りもロールバックされます。密シーケンスの欠点は、現行のトランザクションが終了するまで、シーケンスが他のトランザクションからロックアウトされる点です。

疎シーケンスを使用すると、戻り値が一意であることを保証しますが、現行トランザクションにはバインドされません。トランザクションが疎シーケンス番号を割り振り、後でロールバックした場合、シーケンス番号は単純に失われます。

シーケンス番号は、8 バイトの値です。シーケンス値は、BIGINT、INT、または BINARY データ型で保管できます。BIGINT が推奨です。INT 変数にシーケンス値を保管すると、8 バイトのシーケンス番号は 4 バイトの INT に入りきらないため、情報が失われます。8 バイトの BINARY 値はシーケンス番号を完全に保管できますが、BINARY 値は整数データ型ほど、常に操作が簡単ではありません。

注:

シーケンス番号は 8 バイトの数値なので、(ストアード・プロシージャーまたはアプリケーション・プログラムの) 4 バイト整数は、高位 4 バイトを省略します。そのため、シーケンス番号が 2^31 - 1 (=2147483647) を超えると、不要な動作が生じます。以下は、この動作を示すいくつかのサンプル・コードと出力です。

```
CREATE SEQUENCE seq1;

-- シーケンス番号を 2^31 - 1 に設定し、
-- この値と「次の」値 (2^31) を返します。
"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();
```

この呼び出しの戻り値は、以下のとおりです。

```
      x            y
2147483647      -2147483648
```

x の値は正確ですが、y の値は、正しい正数ではなく負の数値になっています。

個別の表ではなくシーケンサー・オブジェクトを使用する利点は、シーケンサー・オブジェクトは特に高速実行用に調整されていて、通常の更新ステートメントよりもオーバーヘッドが少ない点にあります。

シーケンス値は、SQL ステートメントでインクリメントおよび使用できます。以下の構文を SQL で使用できます。

```
sequence_name.CURRVAL
sequence_name.NEXTVAL
```

シーケンスは、ストアード・プロシージャーの内部でも使用できます。以下のストアード・プロシージャー・ステートメントを使用して、現行シーケンス値をリトリーブできます。

```
EXEC SEQUENCE sequence_name.CURRENT INTO variable
```

以下のストアード・プロシージャー・ステートメントを使用して、新しいシーケンス値をリトリーブできます。

```
EXEC SEQUENCE sequence_name.NEXT INTO variable
```

以下のストアード・プロシージャー・ステートメントで、シーケンス値を設定できます。

```
EXEC SEQUENCE sequence_name SET VALUE USING variable
```

現行シーケンス値をリトリーブするには、選択アクセス権限が必要です。新しいシーケンス値を割り振るには、更新アクセス権限が必要です。これらのアクセス権限は、表アクセス権限と同じ方法で付与または取り消されます。

例

```
CREATE DENSE SEQUENCE SEQ1;
INSERT INTO ORDER (id) VALUES (SEQ1.NEXTVAL);
```

CREATE SYNC BOOKMARK

```
CREATE SYNC BOOKMARK bookmark_name
```

サポート条件

これには、solidDB 拡張レプリケーションが必要です。

使用法

このステートメントは、マスター・データベースでブックマークを作成します。ブックマークは、データベースのユーザー定義のバージョンを表します。solidDB データベースの永続的スナップショットで、特定の同期タスクを実行するための参照を提供します。ブックマークは、通常、レプリカにインポートするデータを EXPORT SUBSCRIPTION コマンドを使用してマスターからエクスポートするため

に使用します。 2 GB よりも大きなデータベースがある場合、データのエクスポートとインポートを使用すると、より効率的にマスターからレプリカを作成できます。

ブックマークを作成するには、管理 DBA 特権または SYS_SYNC_ADMIN_ROLE が必要です。データベースに作成できるブックマークの数に制限はありません。ブックマークは、マスター・データベースにだけ作成します。レプリカ・データベースにブックマークを作成しようとすると、システムがエラーを発行します。

ALTER TABLE SET SYNCHISTORY コマンドで、表が同期履歴用にセットアップされている場合、ブックマークが表の履歴情報を保持します。このため、必要がなくなったときには、DROP SYNC BOOKMARK ステートメントを使用してブックマークをドロップします。そうしない場合、余分な履歴データによって、ディスク・スペースの使用量が増えます。

新しいブックマークを作成すると、システムがその他の属性 (ブックマークの作成者、作成日時、ユニークなブックマーク ID など) を関連付けます。このメタデータは、システム表 SYS_SYNC_BOOKMARKS で保守されます。この表について詳しくは、 386 ページの『SYS_SYNC_BOOKMARKS』を参照してください。

マスターでの使用

CREATE SYNC BOOKMARK ステートメントを使用して、マスター・データベースでブックマークを作成します。

レプリカでの使用

CREATE SYNC BOOKMARK ステートメントは、レプリカ・データベースでは使用できません。

例

```
CREATE SYNC BOOKMARK BOOKMARK_AFTER_DATALOAD;
```

戻り値

各エラー・コードについて詳しくは、「*solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。

表 40. CREATE SYNC BOOKMARK の戻り値

エラー・コード	説明
25066	ブックマークは既に存在しています
13047	操作する特権がありません

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}
```

使用法

表は、CREATE TABLE ステートメントで作成します。CREATE TABLE ステートメントには、作成する列、データ型、各列の値のサイズ (該当する場合) のリストと、その他のオプション (主キーの作成など) が必要です。

重要:

表を作成するときは、常に主キーを定義します。主キーを定義しない場合、solidDB が自動的に作成します。これによって、データがディスク上で予期しない順序になり、性能低下の原因になることがあります。適切な主キーによって、主キーを使用する照会の速度が速くなります。

列レベルと表レベルの両方で、制約定義が使用可能です。列レベルでは、NOT NULL で定義される制約は、列挿入でNULL 以外の値が必要であることを指定します。UNIQUE は、2 つの行で同じ値を持つことができないことを指定します。PRIMARY KEY では、主キーである列が 2 つの行で同じ値を持つことが許可されず、NULL 値も許可されません。すなわち、PRIMARY KEY は、UNIQUE と NOT NULL の組み合わせと同等です。 FOREIGN KEY を含む REFERENCES 節は、参照整合性制約の対象になる表名と列のリストを指定します。これは、この表にデータが挿入または更新されるときに、そのデータが、参照される表および列の値と一致する必要があることを意味します。

CHECK キーワードは、列に挿入できる値を制約します (例えば、値を特定の整数範囲に制約します)。定義すると、チェック制約は、その列に挿入または更新されるすべてのデータの妥当性検査を実行します。データが制約に違反する場合、その変更は禁止されます。以下に例を示します。

```
CREATE TABLE table1 (salary DECIMAL CHECK (salary >= 0.0));
```

check_condition は、列のチェック制約を指定するブール式です。チェック制約は、述部 >、<、=、<>、<=、>=、およびキーワード BETWEEN、IN、LIKE (ワイルド

カード文字を含めることができます)、IS [NOT] NULL で定義されます。式 (WHERE 節の構文に似ています) は、キーワード AND および OR で修飾できます。以下に例を示します。

```
...CHECK (col1 = 'Y' OR col1 = 'N')...
...CHECK (last_name IS NOT NULL)...
```

UNIQUE 制約と PRIMARY KEY 制約は、列レベルまたは表レベルで定義できることに注意してください。また、指定された列に自動的にユニーク索引を作成します。

外部キーは、自分自身の値を通じて別の表を参照する、または別の表に関連する、表の列または列グループです。FOREIGN KEY を使用して、リストされた列が、この表の外部キーであることを指定します。ステートメントの REFERENCES キーワードは、外部キーの参照である表および表の列を指定します。列レベル制約で REFERENCES 節を使用できますが、FOREIGN KEY ... REFERENCES 節は、表レベル制約でのみ使用できることに注意してください。

FOREIGN KEY で REFERENCES 制約を使用するには、常に、参照される表の行を一意的に識別するために十分な列が外部キーの定義に含まれている必要があります。外部キーには、参照される表の主キーと同じ数および型 (データ型) の列が同じ順序で含まれている必要があります。ただし、外部キーの列名とデフォルト値は、主キーと異なっていてもかまいません。

制約に関する以下のルールに注意してください。

- *check_condition* に、副照会、集約関数、ホスト変数、パラメーターを含めることはできません。
- 列のチェック制約は、制約が定義されている列だけを参照できます。
- 表のチェック制約は、表のすべての列がそのステートメントで既に定義されていれば、表のすべての列を参照できます。
- 表に含めることができる主キー制約は 1 つだけですが、ユニーク制約は複数含めることができます。
- CREATE TABLE ステートメントの UNIQUE 制約と PRIMARY KEY 制約を使用して索引を作成できます。ただし、ALTER TABLE ステートメントを使用する場合は、ユニーク・キーまたは主キーの一部になっている列のドロップはできないので注意してください。索引に名前が付き、そのドロップが可能であるため、索引の作成に代わりに CREATE INDEX ステートメントを使用する場合があります。 CREATE INDEX ステートメントには、非ユニーク索引を作成できる機能、索引を昇順と降順のどちらでソートするかを指定できる機能など、いくつかの追加機能もあります。
- パーシスタント表タイプ、トランジエント表タイプ、テンポラリー表タイプの参照整合性ルールは、異なります。
  - テンポラリー表は、別のテンポラリー表を参照できますが、他のタイプの表 (パーシスタントまたはトランジエント) は参照できません。他のタイプの表は、テンポラリー表を参照できません。
  - トランジエント表は、他のトランジエント表とパーシスタント表を参照できます。テンポラリー表は参照できません。テンポラリー表またはパーシスタント表は、トランジエント表を参照できません。

ディスク・ベース表では、行の最大サイズ (BLOB 以外) はページ・サイズの約 1/3 です。インメモリー表では、行の最大サイズ (BLOB を含む) はページ・サイズとほぼ同じです。(少量のオーバーヘッドがディスク・ベース・ページとインメモリー・ページのどちらでも使用されるため、そのページ全体が完全にユーザー・データに使用できるわけではありません。) デフォルトのページ・サイズは 8 KB です。ページ・サイズについて詳しくは、「*IBM solidDB 管理者ガイド*」の solid.ini 構成パラメーター **BlockSize** の説明を参照してください。

サーバーは、BLOB ストレージを判別するために単純なルールを使用するのではなく、原則として、各 BLOB が、行の存在するページの 256 バイトを占有し、BLOB の残りが別の BLOB ページに保管されます。BLOB が 256 バイトよりも短い場合は、BLOB ページではなくメイン・ディスク・ページに完全に保管されます。

各行は、1000 列までに制限されます。

STORE 節は、表がメモリーとディスクのどちらに保管されるかを示します。(この節は、solidDB メイン・メモリー・エンジンでのみ使用可能です。) STORE 節について詳しくは、 「*IBM solidDB インメモリー・データベース・ユーザー・ガイド*」を参照してください。

インメモリー表にできるのは、パーシスタント (通常の) 表、テンポラリー表、トランジエント表です。テンポラリー表およびトランジエント表について詳しくは、「*IBM solidDB インメモリー・データベース・ユーザー・ガイド*」を参照してください。

すべてのテンポラリー表およびトランジエント表は、インメモリー表にする必要があります。「STORE MEMORY」節を指定する必要はありません。 STORE 節を省略すると、テンポラリー表およびトランジエント表は、自動的にインメモリー表として作成されます。(テンポラリー表およびトランジエント表の場合、solid.ini 構成パラメーター DefaultStoreIsMemory は無視されます。) 以下のようなコマンドを実行して、テンポラリー表またはトランジエント表を明示的にディスク・ベース表として作成しようとすると、エラーが発生します。

```
CREATE TEMPORARY TABLE t1 (i INT) STORE DISK; -- 間違い
```

キーワード「GLOBAL」は、テンポラリー表で、SQL:1999 標準に準拠するために含めます。solidDB では、GLOBAL キーワードを使用するかどうかにかかわらず、すべてのテンポラリー表がグローバルです。

構成パラメーターとの相互作用

CREATE TABLE ステートメントのストレージ・ロケーション (ディスクまたはメモリー) は、solid.ini 構成ファイルの DefaultStoreIsMemory パラメーターで指定されているストレージ・ロケーションよりも優先されます。

例

```
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);
```

CREATE TRIGGER

```
CREATE TRIGGER trigger_name ON table_name time_of_operation
    triggering_event [REFERENCING column_reference]
    BEGIN trigger_body END
```

ここで、

```
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
```

注:

この付録は、solidDB SQL コマンドの使用方法に関するクイック・リファレンスとして提供されています。トリガーを使用する状況と方法について詳しくは、 65 ページの『トリガーおよびプロシージャー』を参照してください。

使用法

トリガーは、特定のアクション (INSERT、UPDATE、または DELETE) が発生したときに一連の SQL ステートメントを実行するメカニズムを提供します。トリガーの「本体」には、ユーザーが実行する SQL ステートメントが含まれます。トリガーの本体は、ストアード・プロシージャー言語で作成されます (ストアード・プロシージャー言語については、CREATE PROCEDURE ステートメントに関するセクションで詳しく説明します)。

表に 1 つ以上のトリガーを作成できます。各トリガーは、特定の INSERT、UPDATE、または DELETE コマンドでアクティブ化するように定義されます。ユーザーが表のデータを変更すると、そのコマンドに対応するトリガーがアクティブ化されます。

トリガーでは、インライン SQL またはストアード・プロシージャーだけを使用できます。ストアード・プロシージャーをトリガーで使用する場合は、プロシージャーを CREATE PROCEDURE コマンドで作成する必要があります。トリガー本体から呼び出されるプロシージャーは、別のトリガーを呼び出すことができます。

トリガーを作成するには、DBA、またはトリガーを定義している表の所有者である必要があります。

トリガーは、以下のステートメントで作成されます。

```
CREATE TRIGGER name body
```

また、以下のステートメントでシステム・カタログからドロップされます。

```
DROP TRIGGER name
```

トリガーを使用不可にするには、以下のステートメントを使用します。

```
ALTER TRIGGER name
```

表に定義したトリガーを使用不可にすると、solidDB サーバーは、アクティブ化する DML ステートメントが発行されたときにトリガーを無視します。このコマンドを使用して、現在非アクティブであるトリガーを使用可能にすることもできます。

注:

以下は、CREATE TRIGGER コマンドで使用されるキーワードと節の要約です。使用方法について詳しくは、27 ページの『3 章 ストアード・プロシージャー、イベント、トリガー、およびシーケンス』を参照してください。

トリガー名

trigger_name は、トリガーを識別するもので、254 文字まで含めることができます。

BEFORE | AFTER 節

BEFORE | AFTER 節は、DML ステートメントを呼び出す前にトリガーを実行するか、呼び出した後で実行するかを指定します。状況によっては、BEFORE 節と AFTER 節は交換可能です。ただし、一方の節がもう一方の節より望ましい場合もあります。

- ドメイン制約および参照整合性の検査など、データの妥当性検査を行う場合は、BEFORE 節を使用した方が効率的です。
- AFTER 節を使用すると、DML ステートメントの呼び出しによって使用可能になった、表の行が処理されます。逆に、AFTER 節は DELETE ステートメントを呼び出した後に、データ削除の確認も行います。

表ごとに最大 6 つのトリガーを定義できます。アクション (INSERT、UPDATE、DELETE) と時間 (BEFORE および AFTER) の組み合わせごとに 1 つのトリガーで以下の 6 つがあります。

- BEFORE INSERT
- BEFORE UPDATE
- BEFORE DELETE

- AFTER INSERT
- AFTER UPDATE
- AFTER DELETE

以下の例では、table1 に BEFORE INSERT でトリガー trig01 が定義されています。

```
"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"
```

以下に、CREATE TRIGGER コマンドの BEFORE および AFTER 節をそれぞれの DML 操作に使用した例を (意味と利点も含めて) 示します。

- UPDATE 操作

  BEFORE 節は、UPDATE を処理する前に、変更されたデータが整合性制約ルールに従っているかどうかを検査できます。REFERENCING NEW AS *new_column_identifier clause* を BEFORE UPDATE 節で使用した場合、更新された値は、トリガー SQL ステートメントで使用可能です。トリガー内で、UPDATE の実行前にデフォルトの列値または派生した列値を設定できます。

  AFTER 節は、新規に変更されたデータに対して操作を行うことができます。例えば、支社のアドレスを更新後に、その支社の売上高を計算できます。

  REFERENCING OLD AS *old_column_identifier* 節を AFTER UPDATE 節で使用した場合、トリガー SQL ステートメントから、更新の呼び出し前に存在していた値にアクセスできます。

- INSERT 操作

  BEFORE 節は、INSERT を実行する前に、新しいデータが整合性制約ルールに従っているかどうかを検査できます。パラメーターとして引き渡された列値は、トリガー SQL ステートメントで可視ですが、挿入された行は可視ではありません。トリガー内で、INSERT の実行前にデフォルトの列値または派生した列値を設定できます。

AFTER 節は、新規に挿入されたデータに対して操作を行うことができます。例えば、販売注文の挿入後に、注文の合計を計算して、顧客が割引きの対象になるかどうかを調べることができます。

列値はパラメーターとして引き渡され、挿入された行は、トリガー SQL ステートメントで可視です。

- DELETE 操作

  BEFORE 節は、削除されようとするデータに対して操作を行うことができます。パラメーターとして引き渡された列値と、削除される挿入された行は、トリガー SQL ステートメントで可視です。

  AFTER 節を使用して、データの削除を確認できます。パラメーターとして引き渡された列値は、トリガー SQL ステートメントで可視です。削除された行がトリガー SQL ステートメントで可視であることに注意してください。

INSERT | UPDATE | DELETE 節

INSERT | UPDATE | DELETE 節は、ユーザー・アクション (INSERT、UPDATE、 DELETE) が試行されたときのトリガー・アクションを示します。

トリガーの処理に関連するステートメントは、まず表での呼び出し側 DML ステートメント (INSERT、UPDATE、DELETE) からのコミットおよび自動コミットの前に発生します。トリガー本体またはトリガー本体の中で呼び出されたプロシージャーが COMMIT または ROLLBACK を実行しようとすると、solidDB サーバーは対応するランタイム・エラーを返します。

INSERT では、表での INSERT によってトリガーがアクティブ化されるように指定されます。n 行のデータをロードすることは、n 回の挿入と見なされます。

注:

トリガーを使用可能にしてデータをロードしようとすると、パフォーマンスに影響が生じることがあります。ビジネス・ニーズに応じて、ロードの前はトリガーを使用不可にし、ロードの後にトリガーを使用可能にすることができます。詳しくは、193 ページの『ALTER TRIGGER』を参照してください。

DELETE では、表での DELETE によってトリガーがアクティブ化されるように指定されます。

UPDATE では、表での UPDATE によってトリガーがアクティブ化されるように指定されます。UPDATE 節を使用するときは、以下のルールに注意してください。

- トリガーの REFERENCES 節の中で列を参照できる (列に別名を付けることができる) のは、BEFORE サブ節で 1 回、AFTER サブ節で 1 回までです。また、列が BEFORE と AFTER の両方のサブ節で参照される場合は、各サブ節で列の別名が異なっていなければなりません。

-

solidDB サーバーでは、同じ表に対する再帰的更新が可能で、同じ行に対する再帰的更新が禁止されません。

solidDB サーバーでは、複数の異なるトリガーのアクションによって同じデータが更新される状況は検出されません。例えば、表 Table1 の異なる列、Col1 と Col2 に、2 つの更新トリガー (1 つは BEFORE トリガーで、もう 1 つは AFTER トリガー) があるとします。Table1 のすべての列で更新を試行すると、2 つのトリガーがアクティブ化されます。どちらのトリガーも、2 番目の表 (Table2) の同じ列 (Col3) を更新するストアード・プロシージャーを呼び出します。最初のトリガーが、Table2.Col3 を 10 に更新し、2 番目のトリガーが Table2.Col3 を 20 に更新します。

同様に、solidDB サーバーでは、トリガーをアクティブ化する UPDATE の結果がトリガー自体のアクションと競合する状況は検出されません。例えば、以下の SQL ステートメントがあるとします。

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

この UPDATE でトリガーがアクティブ化され、以下の SQL ステートメントを含むプロシージャーを呼び出す場合、プロシージャーは、トリガーをアクティブ化した UPDATE の結果を上書きします。

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

注:

上の例は再帰的なトリガーの実行につながる可能性があります。これは回避する必要があります。

table_name

table_name は、トリガーが作成される表の名前です。solidDB サーバーでは、従属トリガーが定義されている表をドロップすることができます。表をドロップすると、トリガーを含むすべての従属オブジェクトがドロップされます。それでもランタイム・エラーが発生することがあるので注意してください。例えば、A と B の 2 つの表を作成したとします。プロシージャー SP-B が表 A にデータを挿入した後に表 A がドロップされた場合、表 B に SP-B を呼び出すトリガーがあるとランタイム・エラーが発生します。

trigger_body

trigger_body には、トリガーが起動されたときに実行されるステートメントが格納されています。*trigger_body* の定義は、ストアード・プロシージャーの定義と同じです。ストアード・プロシージャー本体の作成について詳しくは、204 ページの『CREATE PROCEDURE』を参照してください。

本体が空のトリガーを作成することは、無意味ですが、構文としては有効であることに注意してください。

トリガー本体が、solidDB サーバーに登録されているプロシージャーを呼び出すこともできます。solidDB プロシージャー呼び出しルールは、標準のプロシージャー呼び出し方法に従います。

ビジネス・ロジックのエラーは明示的に検査し、エラーを発生させる必要があります。

REFERENCING 節

この節は、INSERT/UPDATE/DELETE 操作にトリガーを作成するときのオプションです。INSERT 操作および DELETE 操作の場合、現行の列 ID を参照できるようにします。 UPDATE 操作の場合、操作が発生する列に別名を割り当てることで、古い列 ID と新しい更新された列 ID の両方を参照できるようにします。

アクセスするときは、OLD または NEW *column_identifier* を指定する必要があります。REFERENCING サブ節を使用して定義しない場合、solidDB サーバーは *column_identifier* へのアクセスを提供しません。

{OLD | NEW} column_name AS col_identifier

この REFERENCING 節のサブ節を使用して、UPDATE 操作の前と後の両方で、列の値を参照できます。これは、ストアード・プロシージャーに渡すことができる新旧の列値セットを生成します。渡されたストアード・プロシージャーには、それらのパラメーター値を判別するためのロジック (例えば、ドメイン制約の検査など) が含まれています。

OLD AS 節は、UPDATE の前に存在する表の古い ID に別名を割り当てるときに使用します。NEW AS 節は、UPDATE の後に存在する表の新しい ID に別名を割り当てるときに使用します。

同じ列の古い値と新しい値の両方を参照する場合は、異なる *column_identifiers* を使用する必要があります。

NEW または OLD として参照される各列は、別個の REFERENCING サブ節を持っている必要があります。

トリガー内のステートメント・アトミシティは、トリガーで実行される操作が、トリガー内の後続の SQL ステートメントで可視になる単位です。例えば、トリガー内で INSERT ステートメントを実行し、その後、同じトリガー内で選択を実行する場合、挿入された行は可視です。

AFTER トリガーの場合、挿入された行または更新された行は AFTER 挿入トリガーの中で可視ですが、削除された行は、そのトリガー内で実行される選択には見ることができません。BEFORE トリガーの場合、挿入された行または更新された行はトリガー内で可視でなく、削除された行は可視です。UPDATE の場合、更新前の値を BEFORE トリガーの中で使用できます。

以下の表で、トリガーのステートメント・アトミシティの要約を示し、トリガー本体の SELECT ステートメントで行が可視かどうかを示します。

表 41. トリガーのステートメント・アトミシティ

操作	**BEFORE トリガー**	**AFTER トリガー**
INSERT	行は不可視	行は可視

表 41. トリガーのステートメント・アトミシティ (続き)

操作	BEFORE トリガー	AFTER トリガー
UPDATE	前の値は可視	新しい値は可視
DELETE	行は可視	行は不可視

トリガーのコメントおよび制限事項

- トリガーが呼び出すストアード・プロシージャーを使用するには、トリガーが定義されている表のカタログ、スキーマ/所有者、および名前を指定し、表でトリガーを使用可能にするか使用不可にするかを指定します。ストアード・プロシージャーについて詳しくは、 27 ページの『3 章 ストアード・プロシージャー、イベント、トリガー、およびシーケンス』を参照してください。
- 表にトリガーを作成するには、DBA 権限があるか、トリガーを定義している表の所有者である必要があります。
- デフォルトでは、表、アクション (INSERT、UPDATE、DELETE)、および時間 (BEFORE および AFTER) の組み合わせごとに最大 1 つのトリガーを定義できます。つまり、表ごとに最大 6 つのトリガーを作成できます。

注:

トリガーは、各行に適用されます。つまり、10 回の挿入があると、トリガーが 10 回実行されます。

- ビューにはトリガーを定義できません (ビューが単一表に基づいている場合でも同様です)。
- トリガーが定義されている表は、従属列が影響を受ける場合、変更できません。
- システム表にはトリガーを作成できません。
- ドロップまたは変更されたオブジェクトを参照するトリガーは実行できません。このエラーを防ぐには、以下のようにします。
  - ドロップした参照先オブジェクトを再作成する。
  -

変更したすべての参照先オブジェクトを元の状態 (トリガーが認識する状態) にリストアする。

•

トリガー・ステートメントでは、二重引用符で囲むことで予約語を使用できます。例えば、以下の CREATE TRIGGER ステートメントは、予約語である「data」という名前の付いた列を参照します。

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

ネストしたトリガーの最大数の設定

トリガーは、別のトリガーを呼び出すことができます。また、自分自身を呼び出すこともできます (再帰的トリガー)。トリガーは、16 レベルの深さまでネストできます。ネストしたトリガーの最大数は、`solid.ini` 構成ファイルの SQL セクションの MaxNestedTriggers パラメーターで設定します。

```
[SQL]
MaxNestedTriggers=n
```

n はネストされたトリガーの最大数です。

デフォルトは、16 トリガーです。

トリガー・キャッシュの設定

トリガーは、solidDB サーバーの個別のキャッシュに入れられます。トリガーが実行されるとき、トリガー・プロシージャー・ロジックがトリガー・キャッシュに入れられ、トリガーが再実行されるときに再開されます。

キャッシュ・サイズは、`solid.ini` 構成ファイルの SQL セクションの TriggerCache パラメーターで設定します。

```
[SQL]
TriggerCache=n
```

ここで、*n* はキャッシュに予約されるトリガーの数です。

エラーの検査

トリガーの実行中に、エラーを受け取ることがあります。エラーは、SQL ステートメントまたはビジネス・ロジックの実行が原因の場合があります。トリガーがエラーを返した場合、その呼び出し側の DML コマンドは失敗します。DML ステートメントの実行中に、自動的にエラーを返すには、WHENEVER SQLERROR ABORT ステートメントをトリガー本体で使用する必要があります。そうしない場合は、各プロシージャー呼び出しまたは SQL ステートメントの後、トリガー本体でエラーを明示的に検査する必要があります。

トリガー本体の一部であるユーザー作成のビジネス・ロジックで発生したエラーの場合は、以下の SQL ステートメントを使用して、プロシージャー変数でエラーを受け取ることができます。

```
RETURN SQLERROR error_string
```

または

```
RETURN SQLERROR char_variable
```

エラーは、以下のフォーマットで返されます。

```
User error: error_string
```

ユーザーが RETURN SQLERROR ステートメントをトリガー本体で指定しない場合、トラップされたすべての SQL エラーは、システムが決定するデフォルトの error_string で発生します。詳しくは、「*solidDB 管理者ガイド* 」の付録『エラー・コード』を参照してください。

注:

トリガー SQL ステートメントは、呼び出し側トランザクションの一部です。トリガーが原因で、またはトリガーの外部で生成されたその他のエラーが原因で呼び出し側 DML ステートメントが失敗した場合、トリガーのすべての SQL ステートメントと失敗した呼び出し側 DML コマンドがロールバックされます。

以下は、呼び出したストアード・プロシージャーの中で、トリガーがエラーを確実にキャッチするように、WHENEVER SQLERROR ABORT を使用する例です。

```
-- ストアード・プロシージャーから SQLERROR を返すと、
-- エラーが表示されます。ただし、ストアード・プロシージャーがトリガーの中から
-- 呼び出された場合は、SQL ステートメント WHENEVER SQLERROR ABORT
-- を使用しない限り、エラーは表示されません。

CREATE TABLE table1 (x INT);
CREATE TABLE table2 (x INT);

"CREATE PROCEDURE stproc1
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 stproc1();
END";
COMMIT WORK;

"CREATE TRIGGER does_not_display_error ON table2 BEFORE INSERT
BEGIN
    EXEC SQL EXECDIRECT CALL stproc1();
END";
COMMIT WORK;

-- ストアード・プロシージャーを実行した場合にエラーが返されたことを示します。
CALL stproc1();

-- トリガーに WHENEVER SQL ERROR ABORT があるため、エラーを表示します。
INSERT INTO table1 (x) values (1);
-- エラーを表示しません。
INSERT INTO table2 (x) values (1);
```

トリガー・スタック関数

以下の関数を使用して、トリガー・スタックの現在の内容を分析できます。

TRIG_COUNT() は、トリガー・スタック内のトリガー数を返します。この数には、現行のトリガーが含まれます。戻り値は整数です。

TRIG_NAME(n) は、トリガー・スタック内の n 番目のトリガー名を返します。最初のトリガー位置またはオフセットはゼロです。

TRIG_SCHEMA(n) は、トリガー・スタック内の n 番目のトリガー・スキーマ名を返します。最初のトリガー位置またはオフセットはゼロです。戻り値はストリングです。

例

```
"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";
```

CREATE USER

```
CREATE USER user_name IDENTIFIED BY password
```

使用法

指定されたパスワードで新しいユーザーを作成します。

例

```
CREATE USER HOBBES IDENTIFIED BY CALVIN;
```

CREATE VIEW

```
CREATE VIEW viewed_table_name [( column_identifier
    [,column_identifier ]... )]
AS query-specification
```

使用法

ビューは、仮想表として表示できます。つまり、この表は、物理的に存在する表ではなく、1 つ以上の表に対する照会仕様によって形成されるものです。

例

```
CREATE VIEW TEST_VIEW
    (VIEW_I, VIEW_C, VIEW_ID)
    AS SELECT I, C, ID FROM TEST;
```

DELETE

```
DELETE FROM table_name [WHERE search_condition]
```

使用法

検索条件に従って、指定された行が所定の表から削除されます。

例

```
DELETE FROM TEST WHERE ID = 5;
DELETE FROM TEST;
```

DELETE (位置付け)

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

使用法

位置付け DELETE ステートメントは、カーソルの現在行を削除します。

例

```
DELETE FROM TEST WHERE CURRENT OF MY_CURSOR;
```

DESCRIBE

```
DESC[RIBE] [type option] {[context_specifier.]object} [format option]
```

上記の詳細は以下のとおりです。

```
type option := CATALOG | SCHEMA | TABLE | VIEW | PROCEDURE | TRIGGER
context_specifier := catalog | [catalog.]schema | [[catalog.]schema.]table
object := catalog | schema | table | view | procedure | trigger
format option :=  [NICE (default) | RAW ]
```

使用法

DESCRIBE ステートメントでは、カタログ、スキーマ、表、ビュー、プロシージャー、または関数などのデータベース・オブジェクトを記述します。このステートメントは、指定された表またはビューの列定義か、指定された関数またはプロシージャーの仕様を出力します。

DESCRIBE の結果は、出力形式 (NICE または RAW) によって異なります。

- NICE 形式は、簡潔で簡素化されています。データベースのメタデータについて素早くアドホック照会する場合に使用できます。
- RAW 形式の方が、より具体的で詳細です。通常であれば solidDB データ・ディクショナリー (soldd) または複雑な SQL 照会を使用して取得されるような情報を出力します。

NICE および RAW の両方の形式が、すべての型で使用可能なわけではありません。

- DESCRIBE CATALOG my_catalog は、指定されたカタログの名前、作成者、所有者、およびスキーマ・リストを NICE 形式で出力します。
- DESCRIBE SCHEMA my_schema は、スキーマ名、それが属するカタログ名、作成者、および所有者を、NICE 形式で出力します。
- DESCRIBE TABLE my_table は、表名、使用されるストレージ・タイプ、親表のリスト、子表のリスト、列、列タイプ、列の精度、列のヌル可能性、列が主キーで

あるか、および列が副次キーであるかを、NICE 形式で出力します。RAW 形式では、create ステートメントが、関連する索引および制約の create ステートメントと共に出力されます。

- DESCRIBE VIEW my_view は、ビューの列を NICE 形式で出力し、create ステートメントを RAW 形式で出力します。
- DESCRIBE PROCEDURE my_procedure は、create ステートメントを RAW 形式で出力します。
- DESCRIBE TRIGGER my_trigger は、create ステートメントを RAW 形式で出力します。

注: 多くの場合、単一の出力形式のみが使用可能です。その場合、ユーザーが指定した出力形式に関わらず、その使用可能な出力のみが生成されます。

例

以下の例では、PERSON という名前の表に関する情報が NICE 形式で出力されています。

```
DESCRIBE PERSON;
  Table name : PERSON
  Table type : In-memory

  Parent tables : CITY
  Child tables  : OWNER, IN_RELATION

  COLUMN      TYPE         PRECISION    NULLABLE   PRIMARY KEY   SECONDARY KEY
  ----------------------------------------------------------------------------
  FIRSTNAME   VARCHAR      30           YES        NO            NULL
  LASTNAME    VARCHAR      60           NO         YES           NULL
  HOME_CITY   INT          NULL         YES        NO            'CITY'

1 rows fetched.
```

以下の例では、PERSON という名前の表に関する情報が RAW 形式で出力されています。

```
DESCRIBE PERSON RAW;

  CREATE TABLE "DBA"."DBA"."PERSON" (
        "FIRSTNAME" VARCHAR(30),
        "LASTNAME" VARCHAR(60) NOT NULL,
        "HOME_CITY" INTEGER,
        FOREIGN KEY("HOME_CITY")
                REFERENCES "DBA"."DBA"."CITY"("CITY"),
        PRIMARY KEY ("LASTNAME")
  );

1 rows fetched.
```

関連項目

LIST

DROP CATALOG

```
DROP CATALOG catalog_name [CASCADE | RESTRICT]
```

使用法

DROP CATALOG ステートメントは、指定されたカタログをデータベースからドロップします。

RESTRICT キーワードを使用する場合、または RESTRICT も CASCADE も指定しない場合は、カタログ自体をドロップする前に、カタログ内のすべてのデータベース・オブジェクトをドロップする必要があります。

CASCADE キーワードを使用すると、カタログにデータベース・オブジェクト (表など) が含まれている場合、自動的にドロップされます。CASCADE キーワードを使用する場合、ドロップするカタログ内のオブジェクトを他のカタログ内のオブジェクトが参照しているときは、参照しているオブジェクトをドロップするか、更新して参照を除去することで、参照が自動的に解決されます。

カタログを作成またはドロップする特権があるのは、データベースの作成者または SYS_ADMIN_ROLE を持つユーザー (つまり DBA) だけです。カタログの作成者であっても、SYS_ADMIN_ROLE 特権を失うとそのカタログをドロップできなくなります。

例

```
DROP CATALOG C1;
DROP CATALOG C2 CASCADE;
DROP CATALOG C3 RESTRICT;
```

DROP EVENT

```
DROP EVENT event_name
DROP EVENT [[catalog_name.]schema_name.]event_name
```

使用法

DROP EVENT ステートメントは、指定されたイベントをデータベースから削除します。

例

```
DROP EVENT EVENT_TEST;
-- カタログ、スキーマ、およびイベントの名前を使用。
DROP EVENT
HR_database.smith_schema.event1;
```

DROP INDEX

```
DROP INDEX index_name
DROP INDEX[[catalog_name.]schema_name.]index_name
```

使用法

DROP INDEX ステートメントは、指定された索引をデータベースから削除します。

例

```
DROP INDEX test_index;
-- カタログ、スキーマ、および索引の名前を使用。
DROP INDEX bank_accounts.bankteller.first_name_index;
```

DROP MASTER

```
DROP MASTER master_name
```

使用法

このステートメントは、マスター・データベースの定義をレプリカ・データベースからドロップします。この操作を行うと、レプリカをマスター・データベースに同期できなくなります。

注:

1. マスター・データベースの使用を終了する場合は、レプリカの登録を抹消する方法が推奨されます。DROP MASTER ステートメントは、MESSAGE APPEND UNREGISTER REPLICA ステートメントを実行できない場合にのみ使用します。レプリカの登録抹消について詳しくは、 275 ページの『MESSAGE APPEND』を参照してください。
2. solidDB で DROP MASTER コマンドを使用する場合は、自動コミットをオフに設定する必要があります。
3. *master_name* が予約語である場合は、それを二重引用符で囲む必要があります。

マスターでの使用

このステートメントはマスターでは使用できません。

レプリカでの使用

このステートメントは、レプリカからマスターをドロップするために、レプリカで使用されます。

例

```
DROP MASTER "MASTER";
DROP MASTER MY_MASTER;
```

戻り値

各エラー・コードについて詳しくは、「*solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。

表 42. DROP MASTER の戻り値

エラー・コード	説明
13047	操作する特権がありません
25007	マスター *master_name* が見つかりません
25019	データベースがレプリカ・データベースではありません

表 42. *DROP MASTER の戻り値 (続き)*

エラー・コード	説明
25056	自動コミットは許可されません
25065	マスター *master_name* に未完了のメッセージ *message_name* が見つかりました

DROP PROCEDURE

```
DROP PROCEDURE procedure_name
DROP PROCEDURE [[catalog_name.]schema_name.]procedure_name
```

使用法

DROP PROCEDURE ステートメントは、指定されたプロシージャーをデータベースから削除します。

例

```
DROP PROCEDURE PROCTEST;
-- カタログ、スキーマ、およびプロシージャーの名前を使用。
DROP PROCEDURE telecomm_database.technician1.add_new_IP_address;
```

DROP PUBLICATION

```
DROP PUBLICATION publication_name
```

使用法

このステートメントは、マスター・データベースでパブリケーション定義をドロップします。ドロップされたパブリケーションに対するすべてのサブスクリプションも自動的にドロップされます。

マスターでの使用

マスターからパブリケーションをドロップするとそのパブリケーションが削除され、レプリカはそのパブリケーションからリフレッシュできなくなります。

レプリカでの使用

このステートメントをレプリカで使用すると、レプリカでパブリケーションを定義している場合にレプリカからパブリケーション定義がドロップされます (ただし、レプリカ・データベースでパブリケーションを定義する必要はなく、定義して有用でないため、レプリカで CREATE PUBLICATION または DROP PUBLICATION を使用する必要はないはずです)。

例

```
DROP PUBLICATION customers_by_area;
```

戻り値

各エラー・コードについて詳しくは、「*solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。

表 43. DROP PUBLICATION の戻り値

エラー・コード	説明
25010	パブリケーション *publication_name* が見つかりません
13111	エンティティー名が未確定です

DROP PUBLICATION REGISTRATION

```
DROP PUBLICATION publication_name REGISTRATION
```

サポート条件

これには、solidDB 拡張レプリケーションが必要です。

使用法

このステートメントは、レプリカ・データベースでパブリケーションの登録をドロップします。パブリケーション定義はマスター・データベースに残されますが、ユーザーはパブリケーションからリフレッシュできなくなります。登録されたパブリケーションに対するサブスクリプションもすべて自動的にドロップされます。

マスターでの使用

このステートメントは、マスター・データベースでは使用されません。

レプリカでの使用

このステートメントをレプリカで使用すると、レプリカでのパブリケーションの登録がドロップされます。このパブリケーションに対するすべてのサブスクリプションとそのデータも、自動的にドロップされます。

例

```
DROP PUBLICATION customers_by_area REGISTRATION;
```

戻り値

各エラー・コードについて詳しくは、「*solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。

表 44. DROP PUBLICATION REGISTRATION の戻り値

エラー・コード	説明
13047	操作する特権がありません
25019	データベースがレプリカ・データベースではありません

表 44. DROP PUBLICATION REGISTRATION の戻り値 (続き)

エラー・コード	説明
25025	ノード名が定義されていません
25071	パブリケーション *publication_name* には登録していません

DROP REMOTE SERVER

```
DROP REMOTE SERVER
```

使用法

DROP REMOTE SERVER ステートメントは、SYS_SERVER システム表のバックエンド・ログイン情報を削除します。ログイン・データは、Universal Cache での SQL パススルーのために使用されます。

通常の用途では、このステートメントをエンド・ユーザーが使用することはありません。

例

```
TBD
```

関連項目

CREATE [OR REPLACE] REMOTE SERVER

ALTER REMOTE SERVER

DROP REPLICA

```
DROP REPLICA replica_name
```

サポート条件

これには、solidDB 拡張レプリケーションが必要です。

使用法

このステートメントは、マスター・データベースからレプリカ・データベースをドロップします。この操作を行うと、ドロップされたレプリカをマスター・データベースに同期できなくなります。

注:

1. レプリカ・データベースの使用を終了する場合は、レプリカの登録を抹消する方法が推奨されます。DROP REPLICA ステートメントは、MESSAGE APPEND UNREGISTER REPLICA ステートメントを実行できない場合にのみ使用します。レプリカの登録抹消について詳しくは、 275 ページの『MESSAGE APPEND』 を参照してください。
2. solidDB で DROP REPLICA ステートメントを使用する場合は、自動コミットをオフに設定する必要があります。

3. *replica_name* が予約語である場合は、それを二重引用符で囲む必要があります。

マスターでの使用

このステートメントは、マスターからレプリカをドロップするためにマスターで使用されます。

レプリカでの使用

このステートメントはレプリカでは使用できません。

例

```
DROP REPLICA salesman_smith ;
DROP REPLICA "REPLICA";
```

戻り値

各エラー・コードについて詳しくは、「*solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。

表 45. DROP REPLICA の戻り値

エラー・コード	説明
13047	操作する特権がありません
25009	レプリカ *replica_name* が見つかりません
25020	データベースがマスター・データベースではありません
25056	自動コミットは許可されません
25064	レプリカ *replica_name* に未完了のメッセージ *message_name* が見つかりました

DROP ROLE

```
DROP ROLE role_name
```

使用法

DROP ROLE ステートメントは、指定されたロールをデータベースから削除します。

例

```
DROP ROLE GUEST_USERS;
```

DROP SCHEMA

```
DROP SCHEMA schema_name [CASCADE | RESTRICT]
DROP SCHEMA [catalog_name.] schema_name [CASCADE | RESTRICT]
```

使用法

DROP SCHEMA ステートメントは、指定されたスキーマをデータベースからドロップします。RESTRICT キーワードを使用する場合、または RESTRICT も CASCADE も指定しない場合は、このステートメントを使用する前に、指定した *schema_name* に関連付けられているすべてのオブジェクトをドロップする必要があります。CASCADE キーワードを使用すると、指定したスキーマ内のすべてのデータベース・オブジェクト (表など) が自動的にドロップされます。

CASCADE キーワードを使用する場合、ドロップするスキーマ内のオブジェクトを他のスキーマ内のオブジェクトが参照しているときは、参照しているオブジェクトをドロップするか、更新して参照を除去することで、参照が自動的に解決されます。

例

```
DROP SCHEMA finance;
DROP SCHEMA finance CASCADE;
DROP SCHEMA finance RESTRICT;
DROP SCHEMA forecasting_db.securities_schema CASCADE;
```

DROP SEQUENCE

```
DROP SEQUENCE sequence_name
DROP SEQUENCE [[catalog_name.]schema_name.]sequence_name
```

使用法

DROP SEQUENCE ステートメントは、指定されたシーケンスをデータベースから削除します。

例

```
DROP SEQUENCE SEQ1;
-- カタログ、スキーマ、およびシーケンスの名前を使用。
DROP SEQUENCE bank_db.checking_acct_schema.account_num_seq;
```

DROP SUBSCRIPTION

レプリカ:

```
DROP SUBSCRIPTION publication_name [{( parameter_list ) | ALL}]
    [COMMITBLOCK number_of_rows ] [OPTIMISTIC | PESSIMISTIC]
```

マスター:

```
DROP SUBSCRIPTION publication_name [{( parameter_list ) | ALL}]
    REPLICA replica_name
```

サポート条件

このコマンドには、solidDB 拡張レプリケーションが必要です。

使用法

レプリカ・データベースで不要となったデータをそのデータベースから削除するには、マスター・データベースからそのデータをリトリーブするために使用されていたサブスクリプションをドロップします。

注:

サブスクリプションをドロップする場合は、solidDB で自動コミットをオフに設定する必要があります。

デフォルトでは、サブスクリプションのデータは 1 つのトランザクションで削除されます。データ量が多い場合 (数万行など) は、COMMITBLOCK を定義することを推奨します。 COMMITBLOCK オプションを使用すると、データが複数のトランザクションで削除されます。これにより、この操作で望ましいパフォーマンスが得られるようになります。

レプリカでは、DROP SUBSCRIPTION ステートメントを最初に実行するときに表レベルのペシミスティック・ロック方式が使用されるようにこのステートメントを定義できます。ペシミスティック・モードを指定すると、影響を受ける表への他のすべての並行アクセスが、ドロップが完了するまでブロックされます。オプティミスティック・モードを使用すると、並行性の競合が原因で DROP SUBSCRIPTION が失敗する場合があります。

サブスクリプションはマスター・データベースからドロップすることもできます。その場合は、コマンドにレプリカ名を指定します。マスター・データベースに登録されているすべてのレプリカの名前は、SYS_SYNC_REPLICAS 表で確認できます。この操作では、このレプリカのサブスクリプションに関する内部情報のみが削除されます。レプリカの実際のデータは保持されます。

マスターからのサブスクリプションのドロップは、レプリカがもうサブスクリプションを使用しなくなったときに、そのサブスクリプション自体をレプリカがドロップしていない場合に便利です。古いサブスクリプションをドロップすると、古い履歴データがデータベースから解放されます。この履歴データは、サブスクリプションをドロップした後にマスター・データベースから自動的に削除されます。

レプリカのサブスクリプションがマスター・データベースでドロップされた場合、レプリカは次回のリフレッシュで全データを受信します。

この場合にサブスクリプションがドロップされると、DROP SUBSCRIPTION は、パブリケーションに対する最後のサブスクリプションがドロップされた場合にパブリケーションの登録もドロップします。それ以外の場合は、DROP PUBLICATION REGISTRATION ステートメントまたは MESSAGE APPEND UNREGISTER PUBLICATION を使用して、登録を明示的にドロップする必要があります。

DROP SUBSCRIPTION ステートメントを最初に実行するときに表レベルのペシミスティック・ロック方式が使用されるようにこのステートメントを定義できます。ペシミスティック・モードを指定すると、影響を受ける表への他のすべての並行アクセスが、インポートが完了するまでブロックされます。オプティミスティック・モードを使用すると、並行性の競合が原因で DROP SUBSCRIPTION が失敗する場合があります。

トランザクションが表に対する排他ロックを獲得した場合は、solid.ini 構成ファイルの [General] セクションの TableLockWaitTimeout パラメーター設定によって、排他ロックまたは共有ロックが解放されるまでのトランザクションの待ち期間が決まります。詳しくは、「*solidDB 管理者ガイド*」のこのパラメーターの説明を参照してください。

マスターでの使用

このステートメントを使用して、指定したレプリカのサブスクリプションをドロップします。

レプリカでの使用

このステートメントを使用して、対象のレプリカからサブスクリプションをドロップします。

例

サブスクリプションをマスター・データベースからドロップします。

```
DROP SUBSCRIPTION customers_by_area('south')
    FROM REPLICA salesman_joe
```

戻り値

各エラー・コードについて詳しくは、「*IBM solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。

表 46. DROP SUBSCRIPTION の戻り値

エラー・コード	説明
13047	操作する特権がありません
25004	動的パラメーターはサポートされません
25009	レプリカ *replica_name* が見つかりません
25010	パブリケーション *publication_name* が見つかりません
25019	データベースがレプリカ・データベースではありません
25020	データベースがマスター・データベースではありません
25041	パブリケーション *publication_name* へのサブスクリプションが見つかりません
25056	自動コミットは許可されません

DROP SYNC BOOKMARK

```
DROP SYNC BOOKMARK bookmark_name
```

サポート条件

このコマンドには、solidDB 拡張レプリケーションが必要です。

使用法

このステートメントは、マスター・データベースで定義されているブックマークをドロップします。ブックマークをドロップするには、管理特権 DBA または SYS_SYNC_ADMIN_ROLE が必要です。一般に、ブックマークはデータをファイルにエクスポートする際に使用されます。ファイルがマスター・データベースからレプリカに正常にインポートされた後に、データをファイルにエクスポートするために使用したブックマークをドロップすることを推奨します。

ブックマークが残っていると、マスター上のデータに対する削除や更新などのそれ以降のすべての変更がマスター・データベース上でブックマークごとに追跡されるので、増分リフレッシュが容易になります。

ブックマークをドロップしないと、マスター・データベースに登録されているブックマークごとに、履歴情報がディスク・スペースを占有し、不必要なディスク I/O が発生します。このためにパフォーマンスが低下することがあります。

注意:
ブックマークのドロップは、エクスポートされたデータが目的のレプリカすべてにインポートされ、すべてのレプリカが少なくとも 1 回同期されてから実行する必要があります。ブックマークのドロップは、インポート対象のレプリカがなくなり、インポート後にそれらのレプリカがパブリケーションから 1 回リフレッシュを実行した時点で行ってください。

solidDB では、ブックマークをドロップする際に、以下のルールを使用して履歴レコードが削除されます。

- 対象の表のいずれかのレプリカに送信された最も古い REFRESH が検索されます。
- 最も古いブックマークが検索されます。
- 検出した REFRESH とブックマークのどちらが古いかが判別されます。
- 古いと判別された方 (最も古い REFRESH か最も古いブックマーク) の時点までのすべての行が履歴から削除されます。

マスターでの使用

DROP SYNC BOOKMARK ステートメントは、マスター・データベースでブックマークをドロップするために使用します。

レプリカでの使用

DROP SYNC BOOKMARK ステートメントは、レプリカ・データベースでは使用できません。

例

```
DROP SYNC BOOKMARK new_database;
DROP SYNC BOOKMARK database_after_dataload;
```

戻り値

各エラー・コードについて詳しくは、「*IBM solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。

表 47. *DROP SYNC BOOKMARK の戻り値*

エラー・コード	説明
25067	シンクロナイザー・ブックマーク *bookmark_name* が見つかりません
13047	操作する特権がありません

DROP TABLE

```
DROP TABLE base_table_name [CASCADE [CONSTRAINTS]]
DROP TABLE [[catalog_name.]schema_name.]table_name [CASCADE
[CONSTRAINTS]]
```

注:

オブジェクトは、通常はドロップ動作 RESTRICT でドロップされます。ただし、以下の場合は例外です。

1. 表に同期履歴表がある場合は、その同期履歴表が自動的にドロップされます (solidDB 3.7 以降)。
2. 表に索引がある場合、索引を先にドロップする必要はありません。表をドロップすると索引が自動的にドロップされます。

使用法

DROP TABLE ステートメントは、指定された表をデータベースから削除します。

例

```
DROP TABLE table1;
-- カタログ、スキーマ、および表の名前を使用。
DROP TABLE domains_db.demand_schema.bad_address_table;
-- 参照元の表で外部キー制約を削除。
DROP TABLE table2 CASCADE CONSTRAINTS;
```

DROP TRIGGER

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

使用法

表で定義されているトリガーを、システム・カタログからドロップ (または削除) します。

表からトリガーを削除するには、表の所有者であるか、DBA 権限を持っている必要があります。

例

```
DROP TRIGGER update_acct_balance;
-- スキーマおよびトリガーの名前を使用。
DROP TRIGGER savings_accounts.update_acct_balance;
-- カタログ、スキーマ、およびトリガーの名前を使用。
DROP TRIGGER accounts.savings_accounts.update_acct_balance;
```

DROP USER

```
DROP USER user_name
```

使用法

DROP USER ステートメントは、指定されたユーザーをデータベースから削除します。このステートメントを使用する前に、指定した *user_name* に関連付けられているすべてのオブジェクトをドロップする必要があります。DROP USER ステートメントはカスケード操作ではありません。

例

```
DROP USER HOBBES;
```

DROP VIEW

```
DROP VIEW view_name
DROP VIEW [[catalog_name.]schema_name.]view_name
```

使用法

DROP VIEW ステートメントは、指定されたビューをデータベースから削除します。

例

```
DROP VIEW sum_of_acct_balances;
-- スキーマおよびビューの名前を使用。
DROP VIEW acct_manager_schema.sum_of_acct_balances;
-- カタログ、スキーマ、およびビューの名前を使用。
DROP VIEW account_db.acct_manager_schema.sum_of_acct_balances;
```

EXPLAIN PLAN FOR

```
EXPLAIN PLAN FOR sql_statement
```

使用法

EXPLAIN PLAN FOR ステートメントは、指定された SQL ステートメントに対して選択されている検索プランを表示します。

例

```
EXPLAIN PLAN FOR select * from tables;
```

EXPORT SUBSCRIPTION

```
EXPORT SUBSCRIPTION publication_name [(publication_parameters)]
    TO 'filename'
    USING BOOKMARK bookmark_name;
    [WITH [NO] DATA];
```

サポート条件

このコマンドには、solidDB 拡張レプリケーションが必要です。

使用法

EXPORT SUBSCRIPTION ステートメントでは、あるバージョンのデータをマスター・データベースからファイルにエクスポートすることができます。その後に IMPORT ステートメントを使用してファイル内のデータをレプリカ・データベースにインポートできます。

EXPORT SUBSCRIPTION にはいくつかの用途があります。その一部を以下に示します。

- 既存のマスターから大規模なレプリカ・データベース (2 GB 超) を作成する。

  この手順では、データのあるまたはデータのないサブスクリプションをまずファイルにエクスポートし、その後にサブスクリプションをレプリカにインポートする必要があります。詳しくは、「*IBM solidDB 拡張レプリケーション・ユーザー・ガイド*」の『データのあるサブスクリプションのエクスポートによるレプリカの作成』および『データのないサブスクリプションのエクスポートによるレプリカの作成』を参照してください。
- 特定のバージョンのデータをレプリカにエクスポートする。

  パフォーマンス上の理由から、MESSAGE APPEND REFRESH を使用してデータをレプリカに送信する代わりに、データの「エクスポート」を選択できます。
- 実際の行データを含まないメタデータ情報をエクスポートする。

  既存のデータを含み、パブリケーションに関連付けられているスキーマおよびバージョン情報のみを必要とするレプリカを作成することができます。

レプリカから REFRESH が要求される MESSAGE APPEND REFRESH ステートメントとは異なり、エクスポートはマスター・データベースから直接要求します。エクスポートの出力は、solidDB メッセージではなく、ユーザー指定のファイルに保存されます。

キーワードおよび節

publication_name および *bookmark_name* は、データベースに存在していなければならない ID です。パブリケーションの作成について詳しくは、『216 ページの『CREATE [OR REPLACE] PUBLICATION』』を参照してください。ブックマークの作成について詳しくは、『224 ページの『CREATE SYNC BOOKMARK』』を参照してください。ファイル名は、単一引用符で囲まれたリテラル値を表します。同じファイル名を指定することで、単一のファイルに複数のパブリケーションをエクスポートできます。

パブリケーション・データは、マスター・データベースから一連の入力パラメーター値 (パブリケーションで使用される場合) とともに REFRESH としてエクスポートされます。

EXPORT SUBSCRIPTION ステートメントは、指定されたブックマークを基準に実行されます。つまり、エクスポート・データの整合性がこのブックマークまで確保されます。データをエクスポートすると、EXPORT SUBSCRIPTION ステートメントはフル・パブリケーションと同様にブックマークまでのすべての行を取り込みます。ただし、エクスポートは指定のブックマークを基準に実行されるため、それ以降の REFRESH はインクリメンタルです。

データをエクスポートおよびインポートする目的でマスターにブックマークを作成する場合は、以下の時点でそのブックマークが存在している必要があります。

- EXPORT SUBSCRIPTION ステートメントがマスター・データベースで実行されるとき

  この時点でブックマークが存在しない場合は、ブックマークが見つからないことを通知するエラー・メッセージ 25067 が生成されます。
- IMPORT ステートメントが対象のレプリカすべてで実行され、それらのレプリカが最初のデータ・セット (REFRESH) を受け取るとき

  ファイルをインポートするときは、マスター・データベースに接続する必要がなく、ブックマークの存在も検査されません。ただし、レプリカがその最初の REFRESH を受け取る時点でブックマークが存在していないと、エラー・メッセージ 25067 で REFRESH が失敗し、インポート・データは使用できません。これに対処するには、マスターで新しいブックマークを作成し、データを再度エクスポートし、データを再度インポートします。

エクスポート・ファイルには複数のパブリケーションを含めることができます。サブスクリプションは、WITH DATA オプションまたは WITH NO DATA オプションを使用してエクスポートできます。

- データのエクスポート先である既存のデータベースが、マスター・データを含んでおらず、データのサブセットを必要としている場合は、WITH DATA オプションを使用してレプリカを作成します。詳しくは、「*IBM solidDB 拡張レプリケーション・ユーザー・ガイド*」の『データのあるサブスクリプションのエクスポートによるレプリカの作成』を参照してください。
- サブスクリプションのインポート先であるデータベースに既にデータがある場合 (例えば既存のマスターのバックアップ・コピーを使用した場合) は、WITH NO DATA オプションを使用してレプリカを作成します。詳しくは、「*IBM solidDB 拡張レプリケーション・ユーザー・ガイド*」の『データのないサブスクリプションのエクスポートによるレプリカの作成』を参照してください。

デフォルトでは、WITH DATA オプションを使用してエクスポート・ファイルが作成され、このファイルにすべての行が取り込まれます。複数のパブリケーションが指定された場合は、エクスポートされるファイルに「WITH DATA」と「WITH NO DATA」の組み合わせを使用することができます。

使用ルール

EXPORT SUBSCRIPTION ステートメントを使用する際には、以下のルールに注意してください。

- 1 つのサブスクリプションにつき 1 つのファイルのみをエクスポートできます。同じファイル名を使用して、1 つのファイルに複数のサブスクリプションを含めることができます。
- エクスポート・ファイルのサイズは、基礎となるオペレーティング・システムによって異なります。使用するプラットフォーム (SUN や HP など) で 2 GB を超えるサイズが許可されていれば、2 GB を超えるファイルを書き込むことができます。この場合、レプリカ (受信側) でも互換性のあるプラットフォームとファイル・システムが使用されている必要があります。そうでないと、レプリカはエクスポート・ファイルを受け入れることができません。マスターとレプリカのオペレーティング・システムがともに 2 GB を超えるファイル・サイズをサポートしていれば、2 GB を超えるエクスポート・ファイルを使用できます。
- エクスポート・ファイルには複数のサブスクリプションを含めることができます。サブスクリプションは WITH DATA または WITH NO DATA のいずれかのオプションを使用してエクスポートできます。複数のサブスクリプションを含むエクスポート・ファイルには、WITH DATA と WITH NO DATA の両方を含めることが可能です。
- WITH NO DATA オプションを使用してサブスクリプションをファイルにエクスポートすると、メタデータ (つまり対象のパブリケーションに対応するスキーマおよびバージョン情報) のみがファイルにエクスポートされます。
- solidDB で EXPORT SUBSCRIPTION ステートメントを使用するときは、自動コミットをオフに設定する必要があります。

マスターでの使用

このステートメントは、ファイルにエクスポートするマスター・データを要求するために使用します。

レプリカでの使用

このステートメントは、レプリカ・データベースでは使用できません。

例

```
EXPORT SUBSCRIPTION FINANCE_PUBLICATION(2004) TO 'FINANCE.EXP'
USING BOOKMARK BOOKMARK_FOR_FINANCE_DB WITH NO DATA;
```

戻り値

各エラー・コードについて詳しくは、「*solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。

表 48. EXPORT SUBSCRIPTION の戻り値

エラー・コード	説明
25056	自動コミットは許可されません
25067	ブックマークが見つかりません

表 48. EXPORT SUBSCRIPTION の戻り値 (続き)

エラー・コード	説明
25068	エクスポート・ファイル *file_name* を開く操作に失敗しました
25010	パブリケーション *name* が見つかりません

EXPORT SUBSCRIPTION TO REPLICA

```
EXPORT SUBSCRIPTION publication_name [(publication_parameters)]
    TO REPLICA replica_node_name
    USING BOOKMARK bookmark_name
    [COMMITBLOCK number_of_rows]
```

サポート条件

このコマンドには、solidDB 拡張レプリケーションが必要です。

使用法

EXPORT SUBSCRIPTION TO REPLICA ステートメントでは、パブリケーションによって指定された大量のデータをマスター・データベースからレプリカ・データベースに送信することができます。エクスポート操作が完了すると、レプリカで MESSAGE APPEND REFRESH ステートメントを使用してサブスクリプションのデータをインクリメンタル方式でリフレッシュできます。

EXPORT SUBSCRIPTION TO REPLICA ステートメントでは、マスターからレプリカにデータを送信する際にディスク・ベースの拡張レプリケーション・メッセージが使用されません。このため、操作中のディスク使用量が最小限に抑えられるので、大量のデータをはるかに効率よくマスターからレプリカに送信できます。

キーワードおよび節

publication_name および *bookmark_name* は、データベースに存在していなければならない ID です。パブリケーションの作成について詳しくは、 216 ページの『CREATE [OR REPLACE] PUBLICATION』を参照してください。ブックマークの作成について詳しくは、 224 ページの『CREATE SYNC BOOKMARK』を参照してください。

パブリケーション・データは、マスター・データベースから一連の入力パラメーター値 (パブリケーションで使用される場合) とともに REFRESH としてエクスポートされます。

EXPORT SUBSCRIPTION TO REPLICA ステートメントは、指定されたブックマークを基準に実行されます。つまり、エクスポート・データの整合性がこのブックマークまで確保されます。データをエクスポートすると、EXPORT SUBSCRIPTION ステートメントはフル・パブリケーションと同様にブックマークまでのすべての行を取り込みます。ただし、エクスポートは指定のブックマークを基準に実行されるため、それ以降の REFRESH はインクリメンタルです。

データをエクスポートする目的でマスターにブックマークを作成する場合は、マスター・データベースで EXPORT SUBSCRIPTION ステートメントが実行されるとき

にそのブックマークが存在している必要があります。この時点でブックマークが存在しない場合は、ブックマークが見つからないことを通知するエラー・メッセージ 25067 が生成されます。

COMMIT BLOCK キーワードでは、エクスポートされるデータのうちレプリカ・データベースにおいて 1 つのトランザクションでコミットされる行数を指定します。大量の行をエクスポートする場合にコミット・ブロックを指定すると、操作のパフォーマンスが向上します。ただし、コミット・ブロックを指定したエクスポート操作がアクティブであるときは、レプリカ・データベースを他のアプリケーションが使用しないようにすることを推奨します。

マスターでの使用

このステートメントは、レプリカ・データベースにエクスポートするマスター・データを要求するために使用されます。

レプリカでの使用

このステートメントは、レプリカ・データベースでは使用できません。

例

```
EXPORT SUBSCRIPTION FINANCE_PUBLICATION(2004) TO REPLICA replica_1
USING BOOKMARK BOOKMARK_FOR_FINANCE_DB COMMITBLOCK 10000 ;
```

戻り値

各エラー・コードについて詳しくは、「*solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。

表 49. EXPORT SUBSCRIPTION TO REPLICA の戻り値

エラー・コード	説明
25056	自動コミットは許可されません
25067	ブックマークが見つかりません
25010	パブリケーション *name* が見つかりません

GET_PARAM()

```
get_param('param_name')
```

サポート条件

このコマンドには、solidDB 拡張レプリケーションが必要です。

使用法

get_param() 関数は、PUT_PARAM() 関数または SAVE PROPERTY、SAVE DEFAULT PROPERTY、および SET SYNC PARAMETER の各コマンドを使用してトランザクション掲示板に入れられたパラメーターをリトリーブします。リトリーブされたパラメーターはカタログ固有であり、各カタログにはそれぞれに異なるパ

ラメーター・セットがあります。この関数は、パラメーターの VARCHAR 値または NULL (掲示板にパラメーターがない場合) を返します。

get_param() は SQL 関数であるため、プロシージャーまたは SELECT ステートメントでのみ使用できます。

パラメーター名は単一引用符で囲む必要があります。

マスターでの使用

get_param() 関数は、マスターでパラメーター値をリトリーブするために使用します。

レプリカでの使用

get_param() 関数は、レプリカでパラメーター値をリトリーブするために使用します。

solidDB システム・パラメーター

solidDB のシステム・パラメーターは、以下のカテゴリーに分類されます。

- *読み取り専用の*システム・パラメーター。solidDB によって保守され、GET_PARAM(*parameter_name*) 構文でのみ読み取ることができます。

  このカテゴリーのパラメーターのライフ・サイクルは 1 つのトランザクションです。つまり、このパラメーターの値はトランザクション開始時に必ず初期化されます。

- *更新可能な*システム・パラメーター。ユーザーが PUT_PARAM(*parameter_name, value*) を使用して設定および更新できます。更新可能なシステム・パラメーターは、solidDB によって使用されます。

  上記のカテゴリーと同様に、このパラメーターのライフ・サイクルも 1 つのトランザクションです。

- データベース・カタログ・レベルのシステム・パラメーター。SET SYNC PARAMETER *parameter_name value* 構文を使用して設定されます。

  このカテゴリーのパラメーターは、変更または削除されるまで有効な、データベース・カタログ・レベルのパラメーターです。このパラメーターは、掲示板パラメーターとして指定されます。

GET_PARAM()、PUT_PARAM()、および SET SYNC PARAMETER の各関数の完全な構文と使用例については、この章の前半のセクションを参照してください。

特定の掲示板パラメーターについて詳しくは、「*IBM solidDB 拡張レプリケーション・ユーザー・ガイド*」を参照してください。

例

```
SELECT put_param('myparam', '123abc');
SELECT get_param('myparam');
```

戻り値

各エラー・コードについて詳しくは、「*solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。

表 50. GET_PARAM の戻り値

エラー・コード	説明
13086	パラメーターのデータ型が無効です

get_param() が正常に実行されると、割り当てられたパラメーターの値が返されます。

関連項目

PUT_PARAM

SAVE PROPERTY

SET SYNC PARAMETER

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]... }
```

使用法

GRANT ステートメントは以下の目的で使用されます。

1. 指定したユーザーまたはロールに特権を付与する。
2. 指定したユーザーに、指定したロールの特権を与えることで特権を付与する。

ユーザーに付与できるロールは、自分で作成したロールまたは SYS_SYNC_ADMIN_ROLE や SYS_ADMIN_ROLE などのシステム定義ロールです。

ロール SYS_SYNC_ADMIN_ROLE を付与すると、以下をはじめとするデータ同期管理操作を実行する特権が指定したユーザーに与えられます。

- 停止した同期メッセージのドロップまたは再実行
- マスター・データベースからのレプリカ・データベースのドロップ
- ブックマークの作成

ロール SYS_ADMIN_ROLE は、データベースの作成者に与えられるロールです。このロールには、すべての表、索引、およびユーザーに対する特権、および solidDB リモート制御 (テレタイプ) を使用する権限があります。

オプションの WITH GRANT OPTION を使用すると、この特権を与えられたユーザーが他のユーザーに特権を付与できるようになります。

例

```
GRANT GUEST_USERS TO CALVIN;
GRANT INSERT, DELETE ON TEST TO GUEST_USERS;
```

関連項目

ユーザー特権について詳しくは、以下も参照してください。

- 303 ページの『REVOKE (ロールまたはユーザーから特権を)』
- 108 ページの『ユーザー特権およびロールの管理』.

定義済みのロールについて詳しくは、「*IBM solidDB 管理者ガイド*」の『データベース管理のための特殊なロール』という章を参照してください。

GRANT PASSTHROUGH

```
GRANT PASSTHROUGH {READ | WRITE}
TO {PUBLIC | user_name [, user_name]... | role_name [, role_name]... }
[WITH GRANT OPTION]
```

サポート条件

このコマンドでは、SQL パススルー機能を使用する必要があります。

使用法

GRANT PASSTHROUGH ステートメントは、SQL パススルーのアクセス権限を付与します。

例

```
GRANT PASSTHROUGH READ TO cdcuser1, cdcuser2

GRANT PASSTHROUGH WRITE TO cdcuser1, cdcuser2
```

関連項目

303 ページの『REVOKE PASSTHROUGH』

GRANT REFRESH

```
GRANT { REFRESH | SUBSCRIBE } ON publication_name TO {PUBLIC |
user_name,
    [ user_name ] ... | role_name , [ role_name ] ...}
```

サポート条件

このコマンドには、solidDB 拡張レプリケーションが必要です。

使用法

このステートメントは、マスター・データベースで定義されているユーザーまたはロールに対してパブリケーションでのアクセス権限を付与します。

注:

キーワード「SUBSCRIBE」および「REFRESH」は同等です。ただし、GRANT ステートメントでは「SUBSCRIBE」キーワードは推奨されません。

マスターでの使用

このステートメントは、ユーザーまたはロールにパブリケーションへのアクセス権限を付与するために使用します。

レプリカでの使用

このステートメントは、レプリカ・データベースでは使用できません。

例

```
GRANT REFRESH ON customers_by_area TO salesman_jones;
GRANT REFRESH ON customers_by_area TO all_salesmen;
```

戻り値

各エラー・コードについて詳しくは、「*solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。

表 51. GRANT REFRESH の戻り値

エラー・コード	**説明**
13137	付与/取り消しモードが正しくありません
13048	付与オプションの特権がありません
25010	パブリケーション *name* が見つかりません

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]
```

構文で使用するキーワードと節については以下を参照してください。

疑似コメント ID

疑似コメントの接頭部には、識別情報が付加されます。vendor には SOLID、product には Engine、および疑似コメントのクラス名である option には hint を指定する必要があります。

注:

疑似コメントの接頭部 --(* および *)-- では、括弧とアスタリスクの間にスペースを入れないでください。

ヒント

ヒントは、常にその適用対象である SELECT、UPDATE、または DELETE キーワードに続いて置きます。

注:

ヒントを INSERT キーワードの後に置くことは許されません。

注意:

ヒントを使用する場合に、照会をストリングとして構成し、そのストリングを ODBC または JDBC を使用してサブミットするときは、コメントの終わりを表す適切な改行文字がストリングに埋め込まれていることを確認する必要があります。これが埋め込まれていないと、構文エラーになります。改行を埋め込まないと、最初のコメントの開始位置以降のすべてのステートメントがコメントのように見えてしまいます。例えば、コードが以下のようになっているとします。

```
strcpy(s, "SELECT --(* hint... *)-- col_name FROM table;");
```

最初の「--」以降がすべてコメントのように見えるため、ステートメントが不完全に見えます。以下のように記述する必要があります。

```
strcpy(s, "SELECT --(* hint... *)-- ¥n col_name FROM table;");
```

埋め込まれた改行文字「¥n」でコメントを終了させます。デバッグの際に、ストリングを印刷して表記が正しいことを確認するために便利な手法です。以下のような表記になっている必要があります。

```
SELECT --(* hint ... *)--
column_name FROM table_name...;
```

または

```
SELECT --(* hint ... *)--
column_name FROM table_name...;
```

各副選択にはそれぞれにヒントを指定する必要があります。有効なヒント構文の例を以下に示します。

```
INSERT INTO ... SELECT hint FROM ...
UPDATE hint TABLE ... WHERE column = (SELECT hint ... FROM ...)
DELETE hint TABLE ... WHERE column = (SELECT hint ... FROM ...)
```

1 つの疑似コメントに複数のヒントを指定する場合は、以下の例に示すようにヒントをコンマで区切ってください。

例 1

```
SELECT
--(* vendor(SOLID), product(Engine), option(hint)
--MERGE JOIN
--JOIN ORDER FIXED *)--
*
FROM TAB1 A, TAB2 B;
WHERE A.INTF = B.INTF;
```

例 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;
```

ヒントは、特定の動作に対応する特定のセマンティクスです。指定可能なヒントの一覧を以下に示します。

表 52. ヒント

ヒント	定義
MERGE JOIN	選択照会で FROM 節にリストされたすべての表に対してマージ結合アクセス・プランを選択するように、オプティマイザーに指示します。MERGE JOIN オプションは、2 つの表のサイズがほぼ同じで、かつデータが等しく分散している場合に使用します。同じ量の行が結合される場合は、LOOP JOIN よりも高速です。MERGE JOIN では、データを結合する場合に最大 3 つの表がサポートされます。結合表は、列を結合し、それらの列の結果を組み合わせることで順に並べられます。 このヒントは、データが結合キーでソートされ、かつネスト・ループ結合のパフォーマンスが十分でない場合に使用できます。オプティマイザーは、表間に等しい述部がある場合にのみマージ結合を選択します。そうでない場合は、MERGE JOIN ヒントが指定されていても、オプティマイザーは LOOP JOIN を選択します。 マージ操作を実行する前にデータがソートされていない場合は、solidDB の照会実行プログラムによってデータがソートされることに注意してください。 このヒントの使用を検討する際には、ソートを伴うマージ結合の方がソートを伴わないマージ結合よりも多くのリソースを必要とすることに留意してください。

表 52. ヒント (続き)

ヒント	定義
LOOP JOIN	選択照会で FROM 節にリストされたすべての表に対してネスト・ループ結合を選択するように、オプティマイザーに指示します。デフォルトでは、オプティマイザーはネスト・ループ結合を選択しません。表が小さく、メモリーに収まる場合にループ結合を使用すると、他の結合アルゴリズムを使用した場合よりも効率がよくなることがあります。 LOOP JOIN では、内部表と外部表の両方がループされます。この方法は、少量の行と大量の行を結合するときに使用します。この場合、内部表の列と外部表の列で一致するものが検索されます。パフォーマンスを高めるために、結合する列には索引を付ける必要があります。 ループ結合は、表が小さく、メモリー内に収まる場合に使用できます。
JOIN ORDER FIXED	結合の際に照会の FROM 節でリストされている順に表を使用するように、オプティマイザーに指示します。つまり、オプティマイザーは結合の順序を変更せず、結合の実行を完了するために代わりのアクセス・パスを検索することもありません。 このヒントを使用する前に、必ず EXPLAIN PLAN を実行して関連するプランを参照してください。これにより、この結合順序で照会を実行する際に使用されるアクセス・プランを把握できます。
INTERNAL SORT	照会実行プログラムで内部ソーターを使用するように指定します。このヒントは、予想される結果セットが小さい場合 (数千行ではなく数百行規模) に使用します。例えば、数個の集計の実行、結果セットが小さい ORDER BY や GROUP BY などを実行する場合が考えられます。 このヒントを指定することで、コストのかかる外部ソーターの使用を回避できます。

EXTERNAL SORT	照会実行プログラムで外部ソーターを使用するように指定します。このヒントは、予想される結果セットが大きく (例えば数千行規模)、メモリーに収まらない場合に使用します。
	
	また、外部ソート・ヒントを使用する前に solid.ini ファイルで SORT 作業ディレクトリーを指定します。作業ディレクトリーが指定されていないと、ランタイム・エラーが発生します。作業ディレクトリーは、solid.ini 構成ファイルの [sorter] セクションで指定します。以下に例を示します。
	
	`[sorter]`
	`TmpDir_1=c:¥soldb¥temp1`

表 52. ヒント (続き)

ヒント	定義
INDEX [REVERSE] *table_name.index_name*	指定の表に対して指定の索引スキャンの実行を強制します。この場合オプティマイザーは、アクセス・プランの作成に使用できる索引が他にないか、あるいは表スキャンの方が指定された照会に適していないかを評価する処理に進みません。 このヒントを使用する前に、EXPLAIN PLAN 出力を実行してヒントを「テスト」し、生成されたプランが指定された照会に最適であることを確認することを推奨します。 オプションのキーワード REVERSE を指定すると、行が逆の順序で返されます。この場合、照会実行プログラムは索引の最後のページから処理を開始し、索引のキーの降順 (逆の順序) に行を返します。 *tablename.indexname* の tablename は、*catalogname* と *schemaname* を含む完全に修飾された表名であることに注意してください。
PRIMARY KEY [REVERSE] *tablename*	指定の表に対して主キー・スキャンの実行を強制します。 オプションのキーワード REVERSE を指定すると、行が逆の順序で返されます。 指定の表に主キーがない場合は、ランタイム・エラーが返されます。
FULL SCAN *table_name*	指定の表に対して表スキャンの実行を強制します。この場合オプティマイザーは、アクセス・プランの作成に使用できる索引が他にないか、あるいは表スキャンの方が指定された照会に適していないかを評価する処理に進みません。 このヒントを使用する前に、EXPLAIN PLAN 出力を実行してヒントを「テスト」し、生成されたプランが指定された照会に最適であることを確認することを推奨します。
[NO] SORT BEFORE GROUP BY	結果セットが GROUP BY 列でグループ化される前に SORT 操作を実行するかどうかを指定します。 グループ化される項目が少数 (数百行) である場合は、NO SORT BEFORE を使用します。一方、グループ化される項目が大量 (数千行) である場合は、SORT BEFORE を使用してください。

使用法

データ、ユーザー照会、およびデータベースの状態はさまざまであるため、SQL オプティマイザーは可能な最良の実行プランを常に選択できるとは限りません。データが既にソート済みであれば、オプティマイザーに頼らずに、マージ結合の実行を強制して効率を高めることができます。

あるいは、照会内の特定の述部が原因で、オプティマイザーでは解消できないパフォーマンス上の問題が起きるときがあります。ユーザーには、オプティマイザーが

使用している索引が最適でないことが分かる場合もあります。そのような場合は、より高速な結果を生成する索引を使用するよう、オプティマイザーに強制することもできます。

オプティマイザー・ヒントを使用すると、応答時間を細かく制御してパフォーマンス上のニーズを満たすことができます。ユーザーは照会の中で、オプティマイザーに対するディレクティブまたはヒントを指定でき、オプティマイザーはそれを使用して、照会実行プランを決定します。ヒントは、SQL-92 からの疑似コメント構文によって検出されます。

ヒントは、SQL ステートメント内に静的ストリングとして SELECT、INSERT、UPDATE、または DELETE キーワードの直後に記述できます。ヒントは、常にその適用対象である SQL ステートメントに続けて記述します。

オプティマイザーのヒントでの表名の解決は、SQL ステートメント内のすべての表名の場合と同じです。ヒントの指定にエラーがある場合は、その SQL ステートメント全体がエラー・メッセージと共に失敗します。

ヒントを使用可能または使用不可にするには、`solid.ini` で以下の構成パラメーターを使用します。

```
[Hints]
EnableHints = YES | NO
```

デフォルトは YES です。

例

```
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
```

IMPORT

```
IMPORT 'file_name' [COMMITBLOCK number_of_rows]
[{OPTIMISTIC | PESSIMISTIC}]
```

使用法

IMPORT コマンドでは、EXPORT SUBSCRIPTION コマンドで作成されたデータ・ファイルからレプリカにデータをインポートすることができます。

file_name は、単一引用符で囲まれたリテラル値を表します。インポート・コマンドは 1 つのファイル名のみを受け入れます。したがって、レプリカにインポートするすべてのデータが 1 つのファイルに格納されている必要があります。

COMMITBLOCK オプションは、データがコミットされるまでに処理される行数を示します。*number_of_rows* は、オプションの COMMITBLOCK 節でコミット・ブロックのサイズを指定するために使用される整数値です。COMMITBLOCK オプションを使用すると、インポートのパフォーマンスが向上し、内部トランザクション・リソースが頻繁に解放されるようになります。

COMMITBLOCK サイズの最適な値はサーバーにある各種リソースによって異なります。例えば、10,000 行に適した COMMITBLOCK のサイズは 1000 です。COMMITBLOCK オプションを指定しない場合は、IMPORT コマンドでパブリケーション内の全行が 1 つのトランザクションとして使用されます。これは、行数が少ない場合は有効ですが、行数が膨大である場合は問題となります。

インポートを最初に実行するときに表レベルのペシミスティック・ロック方式が使用されるようにインポートを定義できます。ペシミスティック・モードを指定すると、影響を受ける表への他のすべての並行アクセスが、インポートが完了するまでブロックされます。オプティミスティック・モードを使用すると、並行性の競合が原因で IMPORT が失敗する場合があります。

トランザクションが表に対する排他ロックを獲得した場合は、solid.ini 構成ファイルの [General] セクションの **TableLockWaitTimeout** パラメーター設定によって、排他ロックまたは共有ロックが解放されるまでのトランザクションの待ち期間が決まります。詳しくは、「*IBM solidDB 管理者ガイド*」のこのパラメーターの説明を参照してください。

インポートされたデータは、インポート後に 1 回リフレッシュされるまでレプリカで有効となりません。レプリカでその最初の REFRESH が作成されるときに、ファイルのエクスポートに使用されたブックマークがマスター・データベースに存在している必要があります。ブックマークが存在しない場合は REFRESH が失敗します。つまり、マスター・データベース上に新しいブックマークを作成し、データを再度エクスポートし、レプリカにデータを再度インポートする必要があります。

使用ルール

IMPORT コマンドを使用する際には、以下のルールに注意してください。

- 1 つのサブスクリプションにつき 1 つのファイルのみをインポートできます。
- エクスポート・ファイルのサイズは、基礎となるオペレーティング・システムによって異なります。使用するプラットフォーム (SUN や HP など) で 2 GB を超えるサイズが許可されていれば、2 GB を超えるファイルを書き込むことができます。この場合、レプリカ (受信側) でも互換性のあるプラットフォームとファイル・システムが使用されている必要があります。そうでないと、レプリカはエクスポート・ファイルを受け入れることができません。マスターとレプリカのオペレーティング・システムがともに 2 GB を超えるファイル・サイズをサポートしていれば、2 GB を超えるエクスポート・ファイルを使用できます。
- IMPORT コマンドを使用する前にレプリカ・データベースをバックアップします。COMMITBLOCK オプションを使用して処理が失敗した場合は、インポートされたデータの一部のみがコミットされます。この場合は、バックアップ・ファイルを使用してレプリカをリストアする必要があります。
- solidDB で IMPORT コマンドを使用する場合は、自動コミットをオフに設定する必要があります。

マスターでの使用

このステートメントは、マスター・データベースでは使用できません。

レプリカでの使用

このステートメントは、レプリカで、マスター・データベースで EXPORT SUBSCRIPTION ステートメントによって作成されたデータ・ファイルからデータをインポートするために使用します。

例

```
IMPORT 'FINANCE.EXP';
```

戻り値

各エラー・コードについて詳しくは、「*IBM solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。

表 53. IMPORT の戻り値

エラー・コード	説明
25007	マスター *master_name* が見つかりません
25019	データベースがレプリカ・データベースではありません
25069	インポート・ファイル *file_name* を開く操作に失敗しました
13XXX	表レベル・エラー

表 53. IMPORT の戻り値 (続き)

エラー・コード	説明
13124	ユーザー ID *num* が見つかりません このメッセージは、例えばユーザーがドロップされた場合に、生成されます。
10006	並行性競合 (他の操作と同時)
13047	操作する特権がありません
13056	疑似列に挿入は許可されません
21XXX	通信エラー
25024	マスターが定義されていません
25026	有効なマスター・ユーザーではありません
25031	トランザクションがアクティブで、操作が失敗しました
25036	パブリケーション *publication_name* が見つからないか、パブリケーションのバージョンが一致しません
25040	ユーザー ID *user_id* が見つかりません メッセージ応答を実行中に、マスター・ユーザーをローカル・レプリカ ID にマップしようとして失敗しました。
25041	パブリケーション *publication_name* へのサブスクリプションが見つかりません
25048	パブリケーション *publication_name* 要求情報が見つかりません
25054	表 *table_name* が同期履歴に設定されていません
25056	自動コミットは許可されません
25060	列 column_name は、表 *table_name* 内のパブリケーション *publication_name* 結果セット上に存在しません

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

使用法

INSERT ステートメントにはいくつかの種類があります。最も単純な例では、表が定義 (または変更) されたときに指定された順序で、新しい行の各列に値が挿入されます。望ましい形式の INSERT ステートメントでは、列がステートメントの一部として指定され、列リストの順序が値リストの順序と一致している限り特定の順序で列を並べる必要がありません。

<insert_value> には、リテラル、スカラー関数、または変数 (プロシージャー内) を指定できます。

例

```
INSERT INTO TEST (C, ID) VALUES (0.22, 5);
INSERT INTO TEST VALUES (0.35, 9);
```

複数行の挿入を実行することもできます。例えば、3 行を 1 つのステートメントで挿入するには、以下のコマンドを使用します。

```
INSERT INTO employees VALUES
(10021, 'Peter', 'Humlaut'),
(10543, 'John', 'Wilson'),
(10556, 'Bunba', 'Olo');
```

以下の 2 番目の例のように、DEFAULT VALUES ステートメントを使用することでデフォルト値を挿入できます。等価の形式は「INSERT INTO TEST()VALUES()」です。ある列に特定の値を割り当て、別の列にデフォルト値を使用することもできます。それぞれの方法は以下の例で示すとおりです。

```
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;
```

LIST

```
LIST <statement>
```

上記の詳細は以下のとおりです。

```
statement :=

CATALOGS |
USERS |
ROLES |

[catalog.]SCHEMAS |
[catalog.]MASTERS |
[catalog.]REPLICAS |
[catalog.]SUBSCRIPTIONS |
[catalog.]PUBLICATIONS |
[[catalog.]schema.]TABLES |
```

```
[[catalog.]schema.]VIEWS |
[[catalog.]schema.]PROCEDURES |
[[catalog.]schema.]EVENTS |
[[catalog.]schema.]SEQUENCES |
[[[catalog.]schema.]table.]INDEXES |
[[[catalog.]schema.]table.]TRIGGERS
```

使用法

LIST ステートメントを使用して、solidDB データベース内の既存のデータベース・オブジェクトを表示できます。LIST ステートメントは、特定のデータベース・オブジェクト・タイプのリストを出力します。

オブジェクト・タイプは、引数として指定されます。任意のオブジェクト名を、ワイルドカード式 (例えば foo%b_r など) に置き換えることができます。ワイルドカード「%」は、その特定のレベルにあるすべての可能なオブジェクトに範囲を拡張します。

LIST ステートメントは、以下のように DESCRIBE コマンドと共に使用できます。

```
LIST my_catalog.my_schema.PROCEDURES
```

```
DESCRIBE my_catalog.my_schema.certain_procedure
```

例

```
LIST my_schema.TABLES;

Catalog: my_catalog
  Schema: my_schema
    TABLES:
    -------
    CITY
    PERSON

1 rows fetched.
```

```
LIST my_catalog.%.TABLES;

Catalog: my_catalog
  Schema: my_schema
    TABLES:
    -------
    PERSON
    IN_RELATION
    CITY
    CAR
  Schema: another_schema
    TABLES:
    -------
    HORSENAME


1 rows fetched.
```

関連項目

DESCRIBE

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: ロックする表の名前。表名を修飾することで、表のカタログとスキーマを指定することもできます。ロックできるのは表だけです。ビューはロックできません。

SHARED: 共有モードでは、他のユーザーに対象の表での読み取り操作と書き込み操作の実行が許可されます。DDL 操作も許可されます。共有モードでは、他のユーザーが同じ表に排他ロックを発行することは禁止されます。

EXCLUSIVE: 表でペシミスティック・ロック方式が使用されている場合に排他ロックを取得すると、他のユーザーは表にいっさいアクセス (データの読み取りやロックの獲得など) できなくなります。表でオプティミスティック・ロック方式が使用されている場合に排他ロックを取得すると、他のユーザーはその表で SELECT を実行できますが、それ以外のアクティビティー (共有ロックの獲得など) は禁止されます。

LONG: デフォルトでは、トランザクションの終了時にロックが解放されます。LONG オプションを指定すると、ロック元のトランザクションがコミットされてもロックは解放されません。(ロック元のトランザクションが異常終了するかロールバックされた場合は、LONG ロックを含めたすべてのロックが解放されます。) ユーザーは、UNLOCK コマンドを使用して LONG ロックを明示的に解除する必要があります。このコマンドについては、この資料で後述します。LONG ロックは、排他モードでのみ使用できます。LONG 共有ロックはサポートされていません。

NOWAIT: 指定した表が別のユーザーによってロックされている場合でも、直ちに自分に制御が戻されるように指定します。要求したロックが許可されない場合は、エラーが返されます。

WAIT: 要求したロックを取得するまでシステムが待機する時間を秒単位で指定します。要求したロックが指定した時間内に許可されなかった場合は、エラーが返されます。

使用法

LOCK コマンドと UNLOCK コマンドでは、表を手動でロックまたはロックを解除できます。表 (またはその他のオブジェクト) にロックを設定すると、そのオブジェクトへのアクセスが制限されます。LOCK TABLE コマンドには、手動の排他ロックの期間を現行のトランザクションが終了した後も延長するオプションがあります。つまり、連続する複数のトランザクションにわたって表の排他ロックを維持できます。

手動ロック方式が必要になることはあまりありません。通常はサーバーによる自動ロック方式で十分です。ロック方式全般、特にサーバーの自動ロック方式について詳しくは、131 ページの『並行性制御とロック方式』を参照してください。

表を明示的にロックする主な目的は、データベース管理者が他のユーザーに妨害されることなく保守操作を実行できるようにすることです。(保守モードについて詳しくは、「*solidDB 拡張レプリケーション・ユーザー・ガイド*」の『分散システムのスキーマの更新および保守』という表題の章を参照してください。) ただし、「保守モード」を使用していない場合でも表を手動でロックできます。

表ロックには共有と排他のいずれかを指定できます。

表の排他ロックは、他のユーザーや接続による表または表内のレコードの変更を禁止します。表で排他ロックを取得すると、その排他ロックを解放するまで他のユーザー/接続はその表に対して以下のすべての操作を実行できなくなります。

- INSERT、UPDATE、DELETE
- ALTER TABLE
- DROP TABLE
- LOCK TABLE (共有モードまたは排他モード)

さらに、表でペシミスティック・ロック方式が使用されている場合は、排他ロックによって他のユーザー/接続が以下の操作も実行できなくなります。

- SELECT

表でペシミスティック・ロック方式が使用されている場合にその表で排他ロックを取得すると、他のすべてのユーザーがその表で SELECT を実行できなくなります。ただし、表でオプティミスティック・ロック方式が使用されている場合は、排他ロックを取得しても他のユーザーは SELECT でその表からレコードを選択できます。(市場で販売されているデータベース・サーバーのほとんどはこのように動作しません。つまり、排他的にロックされている表では SELECT を実行できません。これは、それらのデータベース・サーバーがペシミスティック・ロック方式のみを使用するためです。)

共有ロックは排他ロックほど制限的ではありません。表で共有ロックを取得すると、そのロックを解放するまで他のユーザー/接続は以下の操作を実行できなくなります。

- ALTER TABLE
- DROP TABLE
- LOCK TABLE (排他モード)

表で共有ロックを取得した場合、他のユーザー/接続はその表で挿入、更新、削除、および選択を実行できます。

表での共有ロックは、レコードでの共有ロックとやや異なることに注意してください。レコードで共有ロックを取得した場合、他のユーザーはレコードのデータを変更できません。一方、表で共有ロックを取得した場合は、他のユーザーもその表のデータを変更できます。

一度に複数のユーザーが同じ表で共有ロックを取得できます。したがって、表で共有ロックを取得した場合は、他のユーザーもその表で共有ロックを取得する可能性があります。ただし、あるユーザーが表で共有ロック (または排他ロック) を取得した場合は、どのユーザーもその表で排他ロックを取得できません。

LOCK コマンドは実行された時点で有効となります。LONG オプションを使用しなかった場合は、トランザクション終了時にロックが解放されます。LONG オプションを使用した場合は、表を明示的にロック解除するまで表ロックが維持されます。(表ロックは、ロックを設定したトランザクションをロールバックした場合にも解放されます。つまり、LONG ロックは、そのロックを設定したトランザクションをコミットした場合にのみ複数のトランザクション間で維持されます。)

LOCK/UNLOCK TABLE コマンドは表のみに適用されます。表内のレコードを手動でロックまたはロックを解除するコマンドはありません。

特権の必要性: LOCK TABLE コマンドを使用して表でロックを発行するには、その表で挿入、削除、または更新を行うための特権が必要です。他のユーザーに表での LOCK 特権と UNLOCK 特権を付与する GRANT コマンドはないので注意してください。

1 回の LOCK コマンドで複数の表をロックし、別々のモードを指定できます。LOCK コマンドが失敗した場合は、どの表もロックされません。LOCK コマンドが成功した場合は、要求したすべてのロックが許可されます。

ユーザーが待機オプション (NOWAIT または WAIT の秒数) を指定しなかった場合は、デフォルトの待機時間が使用されます。これは、デッドロック検出タイムアウトと同じです。

例

```
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;
```

戻り値

各エラー・コードについて詳しくは、「*solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。

表 54. LOCK TABLE の戻り値

エラー・コード	**説明**
10014	リソースがロックされています
13047	操作する特権がありません
13011	表 <tablename> が見つかりません

関連項目

UNLOCK TABLE

SET SYNC MODE { MAINTENANCE | NORMAL }

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')]
```

サポート条件

このコマンドには、solidDB 拡張レプリケーションが必要です。

使用法

MESSAGE BEGIN コマンドでレプリカ・データベース内にメッセージを作成した後、以下のタスクをそのメッセージに付加できます。

- トランザクションをマスター・データベースに伝搬する
- パブリケーションをマスター・データベースからリフレッシュする
- レプリカ・サブスクリプションのパブリケーションを登録または登録抹消する
- レプリカ・データベースをマスターに登録または登録抹消する
- マスター・ユーザー情報 (ユーザー名とパスワードのリスト) をマスター・データベースからダウンロードする

PROPAGATE TRANSACTIONS タスクには WHERE 節が含まれている場合があり、この WHERE 節は、SAVE PROPERTY ステートメントで定義されたトランザクション・プロパティーが特定の基準を満たすトランザクションのみを伝搬するために使用されます。キーワード ALL を使用すると、以前にこのステートメントで設定されたデフォルトの伝搬条件が上書きされます。

```
SAVE DEFAULT PROPAGATE PROPERTY
WHERE property_name {=|<|<=|>|>=|<>} 'value'.
```

これを使用すると、何のプロパティーも含んでいないトランザクションを伝搬できます。

REGISTER REPLICA タスクは、マスター・データベース内のレプリカのリストに新しいレプリカ・データベースを追加します。事前にレプリカがマスター・データベースに登録されていないと、レプリカ・データベース内でその他の同期機能を実行することができません。

マルチマスター環境でそれぞれのマスター・データベースをレプリカと同期させるには、カタログをセットアップして、それぞれのマスター・データベースにレプリカを登録する必要があります。1 つのレプリカ・カタログは、1 つのマスター・カタログにのみ登録できます。このステートメントは同期環境内にカタログが作成さ

れた後に、実際の登録を行います。レプリカへの同期には、マスター・データベースごとに新しいカタログが必要です。カタログについて詳しくは、「*IBM solidDB 拡張レプリケーション・ユーザー・ガイド*」で『マルチマスター・トポロジーのガイドライン』という表題のセクションを参照してください。

注:

単一マスター環境では、カタログを使用する必要はありません。デフォルトでは、カタログが使用されない場合、レプリカの登録はマスター・ベース・カタログへマップされているベース・カタログを自動的に使用して行われ、そのマスター・ベース・カタログの名前はデータベースの作成時に指定されます。

注:

1 つのレプリカ・ノード は複数のマスターを持つことができますが、それは、そのノードが各マスター・カタログごとに別々のレプリカ・カタログ を持っている場合に限られます。1 つのレプリカ・カタログ が、複数のマスターを持つことはできません。

UNREGISTER REPLICA オプションは、マスター・データベース内のレプリカ・リストから、既存のレプリカ・データベースを除去します。

REFRESH タスクは、パブリケーションに対する引数を含んでいる場合があります (パブリケーションで使用された場合)。それらのパラメーターはリテラルでなければならず、例えば、ストアード・プロシージャー変数を使用したりすることはできません。キーワード FULL を REFRESH と一緒に使用すると、完全なデータがレプリカへ強制的にフェッチされます。要求されたパブリケーションは、登録されている必要があります。キーワード REFRESH と SUBSCRIBE は同義語であることに注意してください。ただし、SUBSCRIBE は MESSAGE APPEND ステートメントでは推奨されません。

REGISTER PUBLICATION タスクは、レプリカをパブリケーションからリフレッシュできるよう、パブリケーションをレプリカ内に登録します。ユーザーは、登録したパブリケーションからのみ、リフレッシュすることができます。これにより、パブリケーション・パラメーターが検証され、ユーザーが誤って、希望しないサブスクリプションにサブスクライブしたり、随時サブスクリプションを要求したりすることが防止されます。登録済みパブリケーションが参照するすべての表は、レプリカ内に存在する必要があります。

UNREGISTER PUBLICATION オプションは、既存の登録済みパブリケーションを、マスター・データベース内の登録済みパブリケーションのリストから除去します。

SYNC_CONFIG タスクの入力引数は、マスター・データベースからレプリカへ返されるユーザー名の検索パターンを定義します。LIKE キーワードの規則に従った SQL ワイルドカード (シンボル「%」など) は、この引数内で、文字ストリングである *match_string* と一緒に使用されます。LIKE キーワードの使用について詳しくは、 346 ページの『ワイルドカード文字』を参照してください。

マスターでの使用

MESSAGE APPEND ステートメントをマスター・データベース内で使用することはできません。

レプリカでの使用

MESSAGE APPEND は、MESSAGE BEGIN で作成されたメッセージにタスクを付加するために、レプリカ内で使用します。

例

```
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;
```

戻り値

各エラー・コードについて詳しくは、「*solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。

表 55. MESSAGE APPEND の戻り値

エラー・コード	説明
13133	この製品に有効なライセンスではありません
25004	動的パラメーターはサポートされません
25005	メッセージ *message_name* が既にアクティブです
25006	メッセージ *message_name* はアクティブではありません
25015	構文エラー: *error_message*、行 *line_number*
25018	正しくないメッセージ状態 レプリカ内の付加メッセージは、MESSAGE BEGIN ステートメントと MESSAGE END ステートメントの間に置く必要があります。
25024	マスターが定義されていません
25025	ノード名が定義されていません
25026	有効なマスター・ユーザーではありません
25028	メッセージ *message_name* には、システム・サブスクリプションを 1 つだけ組み込むことができます
25035	メッセージ *message_name* は使用中です ユーザーは現在、このメッセージを作成中または転送中です
25044	SYNC_CONFIG システム・パブリケーションは文字引数だけを受け入れます

表 55. MESSAGE APPEND の戻り値 (続き)

エラー・コード	説明
25056	自動コミットは許可されません
25071	パブリケーション *publication_name* には登録していません
25072	パブリケーション *publication_name* には、既に登録済みです

MESSAGE BEGIN

```
MESSAGE unique_message_name BEGIN [TO master_node_name]
```

サポート条件

このコマンドには、solidDB 拡張レプリケーションが必要です。

使用法

レプリカからマスター・データベースへ送信される各メッセージは、明示的に MESSAGE BEGIN ステートメントで開始されている必要があります。

各メッセージには、レプリカ内で固有の名前を持っている必要があります。固有のメッセージ名を構成するには、339 ページの『ストリング関数』で説明されている GET_UNIQUE_STRING() 関数を使用できます。メッセージの処理が完了した後、そのメッセージの名前を再利用できます。ただし、何らかの理由でメッセージが失敗した場合、マスターは失敗したメッセージのコピーを保持するため、失敗したメッセージを削除せずにメッセージ名を再利用しようとすると、その名前は固有でなくなります。既存の名前を再利用できる状態でも、新しいメッセージ名を使用した方がよいでしょう。同じマスターの 2 つのレプリカが同じメッセージ名を持つことは可能である点に注意してください。

レプリカをマスター・システム・カタログ以外のマスター・カタログに登録する場合は、MESSAGE BEGIN コマンド内でマスター・ノード名を指定する必要があります。マスター・ノード名は、マスター・データベース側で正しいカタログを解決するために使用されます。マスター・ノード名を指定するのは、REGISTER REPLICA ステートメントを使用するときだけであることに注意してください。その後のメッセージは、自動的に正しいマスター・ノードへ送信されます。

オプションの「TO *master_node_name*」節を使用する場合は、*master_node_name* を二重引用符で囲む必要があります。

注:

メッセージを処理するときは、必ず自動コミット・モードをオフに切り替えてください。

マスターでの使用

MESSAGE BEGIN ステートメントをマスター・データベース内で使用することはできません。

レプリカでの使用

MESSAGE BEGIN は、レプリカ内で新規メッセージの構築を開始するために使用します。

例

```
MESSAGE MyMsg0001 BEGIN ;
MESSAGE MyMsg0002 BEGIN TO "BerkeleyMaster";
```

レプリカからの戻り値

各エラー・コードについて詳しくは、「*solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。

表 56. レプリカからの MESSAGE BEGIN の戻り値

エラー・コード	**説明**
25005	メッセージ *message_name* が既にアクティブです 指定された名前のメッセージは既に作成されており、まだアクティブのように見えます。そのメッセージは、メッセージの応答がレプリカ内で正常に実行されると、自動的に削除されます。
25035	メッセージ *message_name* は使用中です ユーザーは現在、このメッセージを作成中または転送中です。
25056	自動コミットは許可されません

マスターからの戻り値

各エラー・コードについて詳しくは、「*solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。

表 57. マスターからの MESSAGE BEGIN の戻り値

エラー・コード	**説明**
25019	データベースがレプリカ・データベースではありません
25025	ノード名が定義されていません
25056	自動コミットは許可されません

MESSAGE DELETE

```
MESSAGE message_name [FROM REPLICA replica_name] DELETE
```

サポート条件

このコマンドには、solidDB 拡張レプリケーションが必要です。

使用法

メッセージの実行がエラーのために終了した場合、このコマンドを使用して明示的にメッセージをデータベースから削除し、エラーからリカバリーできます。メッセージを削除した場合、そのメッセージ内でマスターへ伝搬された現行トランザクションと後続のすべてのトランザクションが、永遠に失われることに注意してください。このステートメントを使用するには、SYS_SYNC_ADMIN_ROLE アクセス権限を持っている必要があります。

注:

代替の方法として、MESSAGE DELETE CURRENT TRANSACTION コマンドの方がリカバリー手段として優れています。問題を起こしているトランザクションだけを削除できるからです。

メッセージをマスター・データベースから削除する必要がある場合は、そのメッセージを転送したレプリカ・データベースのノード名も指定する必要があります。

メッセージを削除するときは、必ず自動コミット・モードをオフに切り替えてください。

マスターでの使用

このステートメントは、失敗したメッセージを削除するためにマスター内で使用します。必ず、「FROM REPLICA *replica_name*」構文でレプリカを指定してください。

レプリカでの使用

このステートメントは、メッセージを削除するためにレプリカ内で使用します。

例

```
MESSAGE MyMsg0000 DELETE ;
MESSAGE MyMsg0001 FROM REPLICA bills_laptop DELETE ;
```

レプリカからの戻り値

各エラー・コードについて詳しくは、「*solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。

表 58. レプリカからの MESSAGE DELETE の戻り値

エラー・コード	**説明**
25005	メッセージ *message_name* が既にアクティブです
25013	メッセージ *message_name* が見つかりません
25035	メッセージ *message_name* は使用中です ユーザーは現在、このメッセージを作成中または転送中です。
25056	自動コミットは許可されません

各エラー・コードについて詳しくは、「*solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。

表 59. マスターからの MESSAGE DELETE の戻り値

エラー・コード	説明
13047	操作する特権がありません
25009	レプリカ *replica_name* が見つかりません
25013	メッセージ *message_name* が見つかりません
25020	データベースがマスター・データベースではありません
25035	メッセージ *message_name* は使用中です ユーザーが現在このメッセージを実行中です。
25056	自動コミットは許可されません

MESSAGE DELETE CURRENT TRANSACTION

```
MESSAGE message_name FROM REPLICA replica_name
DELETE CURRENT TRANSACTION
```

サポート条件

このコマンドには、solidDB 拡張レプリケーションが必要です。

使用法

このステートメントは、マスター・データベース内の指定されたメッセージから現行トランザクションを削除します。このステートメントを使用するには、SYS_SYNC_ADMIN_ROLE 特権が必要です。

実行時に重複挿入などの DBMS レベル・エラーが発生した場合は、メッセージの実行が停止します。この種のエラーは、問題を起こしたトランザクションをメッセージから削除すれば解決できます。MESSAGE FROM REPLICA DELETE CURRENT TRANSACTION を使用して削除した後、管理者は同期プロセスに進むことができます。

現行トランザクションを削除するときは、必ず自動コミット・モードをオフに切り替えてください。

このステートメントは、メッセージがエラー状態にあるときにのみ使用します。それ以外で使用すると、エラー・メッセージが返されます。このステートメントはトランザクション操作であり、コミットしてからでないとメッセージの実行を続行できません。削除をコミットした後にメッセージを再始動するには、以下のステートメントを使用します。

```
MESSAGE msgname FROM REPLICA replicaname EXECUTE
```

削除は、MESSAGE FROM REPLICA EXECUTE ステートメントが実行される前に完了することに注意してください。つまり、このステートメントはレプリカからメッセージを開始しますが、実際にメッセージを実行する前に、アクティブ・ステートメントが完了するまで待ちます。このように、このステートメントは非同期式のメッセージ実行を行います。

注意:

トランザクションの削除は、最後の手段としてのみ行ってください。通常、トランザクションはマスター・データベース内の未解決の競合を防止するように書かれています。MESSAGE FROM REPLICA DELETE CURRENT TRANSACTION は、未解決の競合が起きる頻度がより多い開発段階での使用を意図したものです。

トランザクションの削除は、注意して行ってください。後続のトランザクションが、削除されたトランザクションの結果に依存している場合もあるので、トランザクション・エラーが増える危険があります。

マスターでの使用

このステートメントは、失敗したトランザクションを削除するためにマスター内で使用します。

レプリカでの使用

このステートメントは、レプリカ内では使用できません。

例

```
MESSAGE somefailures FROM REPLICA laptop1 DELETE
CURRENT TRANSACTION;
COMMIT WORK;
MESSAGE somefailures FROM REPLICA laptop1 EXECUTE;
COMMIT WORK;
```

戻り値

各エラー・コードについて詳しくは、「*solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。

表 60. MESSAGE DELETE CURRENT TRANSACTION の戻り値

エラー・コード	説明
13047	操作する特権がありません
25009	レプリカ *replica_name* が見つかりません
25013	メッセージ名 *message_name* が見つかりません
25018	正しくないメッセージ状態 エラーがないメッセージからトランザクションを削除しようとしました。
25056	自動コミットは許可されません

MESSAGE END

```
MESSAGE unique_message_name END
```

サポート条件

このコマンドには、solidDB 拡張レプリケーションが必要です。

使用法

メッセージをマスター・データベースへ送信するには、事前にメッセージを「仕上げ」て、永続的なものにしておく必要があります。メッセージを MESSAGE END コマンドで終了すると、そのメッセージはクローズされます。つまり、それ以上、メッセージに何も付加できなくなります。トランザクションをコミットすると、メッセージは永続的なものになります。

注:

メッセージを処理するときは、必ず自動コミット・モードをオフに切り替えてください。

マスターでの使用

MESSAGE END ステートメントをマスター・データベース内で使用することはできません。

レプリカでの使用

MESSAGE END ステートメントは、メッセージを終了するためにレプリカ内で使用します。

例

```
MESSAGE MyMsg001 END ;
COMMIT WORK ;
```

以下の例は、トランザクションを伝搬し、パブリケーション PUB_CUSTOMERS_BY_AREA からリフレッシュする完全なメッセージを示しています。

```
MESSAGE MyMsg001 BEGIN ;
MESSAGE MyMsg001 APPEND PROPAGATE TRANSACTIONS;
MESSAGE MyMsg001 APPEND REFRESH PUB_CUSTOMERS_BY_AREA("~SOUTH');
MESSAGE MyMsg001 END ;
COMMIT WORK ;
```

レプリカからの戻り値

各エラー・コードについて詳しくは、「*solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。

表 61. レプリカからの MESSAGE END の戻り値

エラー・コード	説明
13133	この製品に有効なライセンスではありません

表 61. レプリカからの MESSAGE END の戻り値 (続き)

エラー・コード	説明
25005	メッセージ *message_name* が既にアクティブです
25013	メッセージ *message_name* が見つかりません
25018	正しくないメッセージ状態 トランザクションを開始するには MESSAGE BEGIN ステートメントが存在する必要があり、MESSAGE END ステートメントは 1 つのメッセージにつき 1 回だけ実行できます
25026	有効なマスター・ユーザーではありません
25035	メッセージ *message_name* は使用中です ユーザーは現在、このメッセージを作成中または転送中です。
25056	自動コミットは許可されません

マスターからの戻り値

各エラー・コードについて詳しくは、「*solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。

表 62. マスターからの MESSAGE END の戻り値

エラー・コード	説明
25019	データベースがレプリカ・データベースではありません
25056	自動コミットは許可されません

MESSAGE EXECUTE

```
MESSAGE message_name EXECUTE [{OPTIMISTIC | PESSIMISTIC}]
```

サポート条件

このコマンドには、solidDB 拡張レプリケーションが必要です。

使用法

このステートメントを使用すると、レプリカ内で応答メッセージの実行が失敗した場合に、メッセージを再実行できます。そのような失敗が起きる可能性があるのは、例えば、データベース・サーバーが REFRESH と進行中のユーザー・トランザクションの間で並行性競合を検出した場合などです。

並行性競合が頻繁に起き、メッセージの再実行が並行性競合のために失敗すると予想される場合は、表レベル・ロック方式用に PESSIMISTIC オプションを使用して、メッセージを実行できます。これにより、メッセージの実行が必ず成功します。

このモードでは、影響を受ける表に対する他のすべての並行アクセスは、同期メッセージが完了するまでブロックされます。そのようにせず、オプティミスティック・モードを使用した場合は、MESSAGE EXECUTE ステートメントが並行性競合のために失敗する可能性があります。

トランザクションが表に対する排他ロックを獲得した場合は、solid.ini 構成ファイルの General セクションの TableLockWaitTimeout パラメーター設定によって、排他ロックまたは共有ロックが解放されるまでのトランザクションの待ち期間が決まります。詳しくは、「*solidDB 管理者ガイド*」のこのパラメーターの説明を参照してください。

注:

メッセージを処理するときは、必ず自動コミット・モードをオフに切り替えてください。

マスターでの使用

このステートメントは、マスター内では使用できません。 290 ページの『MESSAGE FROM REPLICA EXECUTE』を参照してください。

レプリカでの使用

このステートメントは、失敗したメッセージの実行を再実行するために、レプリカ内で使用します。

結果セット

MESSAGE EXECUTE は結果セットを返します。返される結果セットは、コマンド MESSAGE GET REPLY の場合と同じものです。

例

```
MESSAGE MyMsg0002 EXECUTE;
```

戻り値

各エラー・コードについて詳しくは、「*solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。

表 63. MESSAGE EXECUTE の戻り値

エラー・コード	説明
13XXX	表レベル・エラー
10006	並行性競合 (他の操作と同時)
13047	操作する特権がありません
13056	疑似列に挿入は許可されません
25005	メッセージ *message_name* が既にアクティブです
25013	メッセージ名 *message_name* が見つかりません

表 63. MESSAGE EXECUTE の戻り値 (続き)

エラー・コード	説明
25018	正しくないメッセージ状態
25024	マスターが定義されていません
25026	有効なマスター・ユーザーではありません
25031	トランザクションがアクティブで、操作が失敗しました。
25035	メッセージ *message_name* は使用中です ユーザーは現在、このメッセージを作成中または転送中です。
25040	ユーザー ID *user_id* が見つかりません メッセージ応答を実行中に、マスター・ユーザーを ローカル・レプリカ ID にマップしようとして失敗しました。
25041	パブリケーション *publication_name* へのサブスクリプションが見つかりません
25048	パブリケーション *publication_name* 要求情報が見つかりません
25056	自動コミットは許可されません

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}]
```

サポート条件

このコマンドには、solidDB 拡張レプリケーションが必要です。

使用法

メッセージを完成し、MESSAGE END ステートメントで永続化した後、MESSAGE FORWARD ステートメントを使用して、そのメッセージをマスター・データベースに送信できます。

メッセージの受信側をキーワード TO で指定する必要があるのは、新しいレプリカをマスター・データベースに登録する場合、つまり、レプリカからマスター・サーバーへ最初のメッセージを送信する場合だけです。

connect_string は、以下のような有効な接続ストリングです。

```
tcp [host_computer_name] server_port_number
```

接続ストリングについて詳しくは、「*solidDB 管理者ガイド*」の『通信プロトコル』という表題のセクションを参照してください。

MESSAGE FORWARD コマンドのコンテキストでは、接続ストリングを単一引用符で区切る必要があります。

node_name (引用符なし) は、予約語以外の有効な英数字シーケンスです。 *"node_name"* (二重引用符付き) は、ノード名が予約語である場合に使用します。その場合、二重引用符によって、ノード名が区切り ID として扱われることが保証されます。例えば、ワード master は予約語であるため、ノード名として使用する場合は、以下のように二重引用符で囲みます。

```
--マスター上
SET SYNC NODE "master";
--レプリカ上
MESSAGE refresh_severe_bugs2 FORWARD TO "master" TIMEOUT FOREVER;
```

送信されたメッセージごとに応答メッセージがあります。TIMEOUT プロパティーは、レプリカ・サーバーが応答メッセージを待つ時間の長さを定義します。

TIMEOUT が定義されていない場合、メッセージはマスターへ転送され、レプリカは応答をフェッチしません。その場合、応答を別の MESSAGE GET REPLY 呼び出しでリトリーブすることができます。

送信されたメッセージの応答に大きなパブリケーションの REFRESH が含まれている場合、REFRESH のコミット・ブロックのサイズ (1 つのトランザクションでコミットされる行の数) を、COMMITBLOCK プロパティーを使用して定義できます。これは、レプリカ・データベースのパフォーマンスに良い影響を与えます。COMMITBLOCK プロパティーを使用するときは、データベースにアクセスするオンライン・ユーザーがいないようにすることを推奨します。

応答メッセージがレプリカ内で初期実行されるとき、MESSAGE FORWARD 操作の一部として、表レベルのペシミスティック・ロック方式を指定できます。PESSIMISTIC モードを指定した場合、影響を受ける表に対する他のすべての並行アクセスは、同期メッセージが完了するまでブロックされます。そのようにせず、オプティミスティック・モードを使用した場合は、MESSAGE FORWARD 操作が並行性競合のために失敗する可能性があります。

トランザクションが表に対する排他ロックを獲得した場合は、`solid.ini` 構成ファイルの General セクションの TableLockWaitTimeout パラメーター設定によって、排他ロックまたは共有ロックが解放されるまでのトランザクションの待ち期間が決まります。詳しくは、「*solidDB 管理者ガイド*」のこのパラメーターの説明を参照してください。

転送されたメッセージの配信が通信エラーのために失敗した場合は、MESSAGE FORWARD を明示的に使用して、メッセージを再送信する必要があります。MESSAGE FORWARD は、再送信された後、メッセージを再実行します。

注:

メッセージを処理するときは、必ず自動コミット・モードをオフに切り替えてください。

例

メッセージを転送し、応答を 60 秒間待ちます。

```
MESSAGE MyMsg001 FORWARD TIMEOUT 60 ;
```

「mastermachine.acme.com」マシン上で稼働するマスター・サーバーにメッセージを転送します。応答メッセージを待ちません。

```
MESSAGE MyRegistrationMsg FORWARD TO
'tcp mastermachine.acme.com 1313';
```

メッセージを転送し、応答を 5 分 (300 秒) 間待ち、リフレッシュしたパブリケーションのデータを、最大 1000 行のトランザクション単位でレプリカ・データベースへコミットします。

```
MESSAGE MyMsg001 FORWARD TIMEOUT 300 COMMITBLOCK 1000 ;
```

レプリカからの戻り値

各エラー・コードについて詳しくは、「*solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。

表 64. レプリカからの MESSAGE FORWARD の戻り値

エラー・コード	**説明**
13XXX	表レベル・エラー
21XXX	通信エラー
10006	並行性競合 (他の操作と同時)
13047	操作する特権がありません
13056	疑似列に挿入は許可されません
25005	メッセージ *message_name* が既にアクティブです
25013	メッセージ名 *message_name* が見つかりません
25018	正しくないメッセージ状態 レプリカ内では、メッセージが終了し、終了トランザクションがコミットされた場合、MESSAGE FORWARD ステートメントを使用してのみメッセージを実行できます。

表 64. レプリカからの MESSAGE FORWARD の戻り値 (続き)

<table>
<tr><th>エラー・コード</th><th>説明</th></tr>
<tr><td>25024</td><td>マスターが定義されていません

このメッセージは、MESSAGE FORWARD ステートメントで connect_string が単一引用符ではなく二重引用符で囲まれていた場合に作成されます。

例えば、マスター・ノードのノード名が "master" (これは予約語なので、二重引用符で区切る必要があります) で、ノードの接続ストリングが以下のとおりであるとします。

tcp localhost 1315

この場合、以下に示す MESSAGE ステートメントは正しいものです。
<pre>--レプリカ上
...
--二重引用符
MESSAGE msg1 BEGIN TO "master";
...
--単一引用符
MESSAGE msg2 FORWARD TO 'tcp localhost 1315';</pre>MESSAGE BEGIN ステートメントは、マスターのノード名が何であるかを (レプリカ・サーバー内で) 定義することに注意してください。MESSAGE FORWARD ステートメントには、サーバーへの接続ストリングを含めることができます。</td></tr>
<tr><td>25026</td><td>有効なマスター・ユーザーではありません</td></tr>
<tr><td>25031</td><td>トランザクションがアクティブで、操作が失敗しました</td></tr>
<tr><td>25035</td><td>メッセージ message_name は使用中です

ユーザーは現在、このメッセージを作成中または転送中です。</td></tr>
<tr><td>25040</td><td>ユーザー ID user_id が見つかりません

メッセージ応答を実行中に、マスター・ユーザーを ローカル・レプリカ ID にマップしようとして失敗しました。</td></tr>
<tr><td>25041</td><td>パブリケーション publication_name へのサブスクリプションが見つかりません</td></tr>
<tr><td>25048</td><td>パブリケーション publication_name 要求情報が見つかりません</td></tr>
<tr><td>25052</td><td>ノード名を node_name に設定できませんでした</td></tr>
<tr><td>25054</td><td>表 table_name が同期履歴に設定されていません</td></tr>
<tr><td>25055</td><td>接続情報は、登録されていない場合にのみ許可されます

MESSAGE message_name FORWARD TO connect_info options 内の接続情報は、レプリカがマスター・データベースにまだ登録されていない場合にのみ許可されます。</td></tr>
<tr><td>25056</td><td>自動コミットは許可されません</td></tr>
</table>

表 64. レプリカからの MESSAGE FORWARD の戻り値 (続き)

エラー・コード	説明
25057	レプリカ・データベースは、既にマスター・データベースに登録されています
25060	列 *column_name* は、表 *table_name* 内のパブリケーション *publication_name* 結果セット上に存在しません

マスターからの戻り値

各エラー・コードについて詳しくは、「*solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。

表 65. マスターからの MESSAGE FORWARD の戻り値

エラー・コード	説明
13XXX	表レベル・エラー
13124	ユーザー ID *num* が見つかりません このメッセージは、例えばユーザーがドロップされた場合に、生成されます。
25016	メッセージが見つかりません。レプリカ ID *replica_id*、メッセージ ID *message_id*
25056	自動コミットは許可されません

結果セット

MESSAGE FORWARD で応答もリトリーブする場合、このステートメントは結果セットを返します。返される結果セットは、ステートメント MESSAGE GET REPLY で返されるものと同じです。 292 ページの『MESSAGE GET REPLY』を参照してください。

MESSAGE FROM REPLICA DELETE

```
MESSAGE msgid FROM REPLICA replicaname DELETE;
MESSAGE msgid FROM REPLICA replicaname DELETE CURRENT TRANSACTION;
```

このコマンドは、マスター上でのみ実行できます。

MESSAGE FROM REPLICA EXECUTE

```
MESSAGE message_name FROM REPLICA replica_name EXECUTE
```

サポート条件

このコマンドには、solidDB 拡張レプリケーションが必要です。

使用法

実行時に重複挿入などの DBMS レベル・エラーが発生した場合、または SYS_ROLLBACK パラメーターをトランザクション掲示板に配置することによってプロシージャーからエラーが発生した場合は、メッセージの実行が停止します。この種のエラーは、例えば、データベースから重複する行を除去するなど、エラーの原因を修正してからメッセージを実行することにより、リカバリーが可能です。

エラーのあるトランザクションを MESSAGE DELETE CURRENT TRANSACTION で削除する場合、削除は、MESSAGE FROM REPLICA EXECUTE コマンドが実行される前に完了することに注意してください。つまり、このステートメントはレプリカからメッセージを開始しますが、実際にメッセージを実行する前に、アクティブ・ステートメントが完了するまで待ちます。このように、このコマンドは非同期式のメッセージ実行を行います。

注:

メッセージを処理するときは、必ず自動コミット・モードをオフに切り替えてください。

マスターでの使用

このコマンドは、失敗したメッセージを実行するためにマスター内で使用します。

レプリカでの使用

このコマンドは、レプリカ内では使用できません。代替の方法については、284 ページの『MESSAGE EXECUTE』を参照してください。

例

```
MESSAGE MyMsg0002 FROM REPLICA bills_laptop EXECUTE;
```

戻り値

各エラー・コードについて詳しくは、「*solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。

表 66. MESSAGE FROM REPLICA EXECUTE の戻り値

エラー・コード	説明
13047	操作する特権がありません
25009	レプリカ *replica_name* が見つかりません
25013	メッセージ名 *message_name* が見つかりません
25018	正しくないメッセージ状態 エラーがないメッセージからトランザクションを削除しようとしました。
25056	自動コミットは許可されません

MESSAGE FROM REPLICA RESTART

```
MESSAGE msgid FROM REPLICA replicaname RESTART <err-options>;
```

ここで、*<err-options>* は IGNORE_ERRORS、LOG_ERRORS、FAIL_ERRORS のいずれかにすることができます。

このコマンドは、マスター上でのみ実行できます。

このコマンドを使用すると、失敗してシステム表に保管され、SYNC_FAILED_MESSAGES ビューを使用してリトリーブすることができるトランザクションを再実行できます。

MESSAGE GET REPLY

```
MESSAGE unique_message_name GET REPLY
[TIMEOUT {FOREVER | seconds}]
[COMMITBLOCK block_size_in_rows]
[NO EXECUTE]
[{OPTIMISTIC | PESSIMISTIC}]
```

サポート条件

このコマンドには、solidDB 拡張レプリケーションが必要です。

使用法

送信されたメッセージへの応答が MESSAGE FORWARD ステートメントによって受信されなかった場合、レプリカ・データベース内の MESSAGE GET REPLY ステートメントを使用して、マスター・データベースとは別に応答を要求できます。

応答メッセージに大きなパブリケーションの REFRESH が含まれている場合、REFRESH のコミット・ブロックのサイズ (1 つのトランザクションでコミットされる行の数) を、COMMITBLOCK プロパティーを使用して制限できます。これは、レプリカ・データベースのパフォーマンスに良い影響を与えます。COMMITBLOCK プロパティーの使用中は、データベース内にオンライン・ユーザーがいないことが推奨されます。

COMMITBLOCK プロパティーを備えた応答メッセージの実行が、レプリカ・データベース内で失敗した場合、再実行することはできません。失敗したメッセージをレプリカ・データベースから削除し、マスター・データベースからリフレッシュする必要があります。

マスター側で応答メッセージが使用可能なときに NO EXECUTE を指定した場合、そのメッセージに対しては、後で実行するために読み取りと保管だけが行われます。それ以外の場合は、応答メッセージがマスターからダウンロードされ、同じステートメント内で実行されます。NO EXECUTE を使用すると、応答メッセージを後で別のトランザクション内で実行できるので、通信回線のボトルネックが減少します。

応答メッセージを初期に実行するとき、表レベルのペシミスティック・ロック方式を使用するよう、応答メッセージを定義できます。PESSIMISTIC モードを指定した場合、影響を受ける表に対する他のすべての並行アクセスは、同期メッセージが完

了するまでブロックされます。そのようにせず、オプティミスティック・モードを使用した場合は、MESSAGE GET REPLY 操作が並行性競合のために失敗する可能性があります。

トランザクションが表に対する排他ロックを獲得した場合は、solid.ini 構成ファイルの General セクションの TableLockWaitTimeout パラメーター設定によって、排他ロックまたは共有ロックが解放されるまでのトランザクションの待ち期間が決まります。詳しくは、「*solidDB 管理者ガイド*」のこのパラメーターの説明を参照してください。

応答メッセージの配信が通信エラーのために失敗した場合は (COMMITBLOCK を使用していない場合)、MESSAGE GET REPLY を明示的に使用して、メッセージを再送信する必要があります。MESSAGE GET REPLY は再送信された後、メッセージを再実行します。

注:

メッセージを処理するときは、必ず自動コミット・モードをオフに切り替えてください。

マスターでの使用

MESSAGE GET REPLY をマスター内で使用することはできません。

レプリカでの使用

MESSAGE GET REPLY をレプリカ内で使用して、マスターからのメッセージの応答をフェッチします。

例

```
MESSAGE MyMessage001 GET REPLY TIMEOUT 120
MESSAGE MyMessage001 GET REPLY TIMEOUT 300 COMMITBLOCK 1000
```

レプリカからの戻り値

トランザクションの伝搬での致命的エラーはメッセージを異常終了し、レプリカにエラー・コードを返します。異常終了したメッセージを伝搬するには、致命的エラーを訂正し、コマンド MESSAGE FROM REPLICA EXECUTE でメッセージを再始動する必要があります。

REFRESH がマスター内で失敗すると、失敗した REFRESH に関するエラー・メッセージが結果セットに追加されます。メッセージのそれ以外の部分は、通常どおり実行されます。失敗した REFRESH は、別の同期メッセージ内でマスターから REFRESH する必要があります。

REFRESH (つまり、応答メッセージの実行) がレプリカ内で失敗した場合、メッセージは引き続きレプリカ・データベース内で使用可能であり、MESSAGE *msg_name* EXECUTE コマンドで再始動できます。

各エラー・コードについて詳しくは、「*solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。

表 67. レプリカからの MESSAGE GET REPLY の戻り値

エラー・コード	説明
13XXX	表レベル・エラー
13124	ユーザー ID *num* が見つかりません このメッセージは、例えばユーザーがドロップされた場合に、生成されます。
10006	並行性競合 (他の操作と同時)
13047	操作する特権がありません
13056	疑似列に挿入は許可されません
21XXX	通信エラー
25005	メッセージ *message_name* が既にアクティブです
25013	メッセージ名 *message_name* が見つかりません
25018	正しくないメッセージ状態 レプリカ内では、メッセージがマスターへ転送される場合、MESSAGE GET REPLY ステートメントを使用してのみ、メッセージを実行できます。
25024	マスターが定義されていません
25026	有効なマスター・ユーザーではありません
25031	トランザクションがアクティブで、操作が失敗しました。
25035	メッセージ *message_name* は使用中です。ユーザーは現在、このメッセージを作成中または転送中です。
25036	パブリケーション *publication_name* が見つからないか、パブリケーションのバージョンが一致しません
25040	ユーザー ID *user_id* が見つかりません メッセージ応答を実行中に、マスター・ユーザーをローカル・レプリカ ID にマップしようとして失敗しました。
25041	パブリケーション *publication_name* へのサブスクリプションが見つかりません
25048	パブリケーション *publication_name* 要求情報が見つかりません
25054	表 *table_name* が同期履歴に設定されていません
25056	自動コミットは許可されません
25057	既にマスター *master_name* に登録済みです

表 67. レプリカからの MESSAGE GET REPLY の戻り値 (続き)

エラー・コード	説明
25060	列 *column_name* は、表 *table_name* 内のパブリケーション *publication_name* 結果セット上に存在しません

マスターからの戻り値

各エラー・コードについて詳しくは、「*solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。

表 68. マスターからの MESSAGE GET REPLY の戻り値

エラー・コード	説明
13XXX	表レベル・エラー
13124	ユーザー ID *num* が見つかりません このメッセージは、例えばユーザーがドロップされた場合に、生成されます。
25012	メッセージ応答がタイムアウトになりました
25016	メッセージが見つかりません。レプリカ ID *replica-id*、メッセージ ID *message-id*
25043	応答メッセージが長すぎます (*size_of_messages* バイト)。最大値は *max_message_size* バイトに設定されています。
25056	自動コミットは許可されません

結果セット

MESSAGE GET REPLY は結果セット表を返します。結果セットの列は、以下のとおりです。

表 69. MESSAGE GET REPLY 結果セット表

列名	説明
Partno	メッセージ・パーツ・ナンバー

表 69. MESSAGE GET REPLY 結果セット表 (続き)

列名	説明
Type	結果セット行のタイプ。以下のタイプがあります。 0: メッセージ・パーツの開始 1: このタイプは使用されていません 2: メッセージは、かつて伝搬メッセージだったものであり、その操作の状況は戻りメッセージに保管されています 3: タスク 4: サブスクリプション・タスク 5: リフレッシュのタイプ (FULL または INCREMENTAL) 6: MESSAGE DELETE 状況
Masterid	マスター ID
Msgid	メッセージ ID
Errcode	メッセージ・エラー・コード。成功した場合はゼロ
Errstr	メッセージ・エラー・ストリング。NULL は成功
Insertcount	レプリカへ挿入された行の数 Type=3: 挿入の合計数 Type=4: レプリカ・ヒストリーからレプリカ基本表へリストアされた行挿入 Type=5: マスターから受信した挿入操作
Deletecount	Type = 3: 削除の合計数 Type = 4: レプリカ基本表から復元された行削除 Type = 5: マスターから受信した削除操作
Bytecount	メッセージのサイズ (バイト単位)。コマンド MESSAGE END から受信した結果の中で示されます。それ以外の場合は 0
Info	現行タスクの情報 Type = 0: メッセージ名 Type = 3: パブリケーション名 Type = 4: 表名 Type = 5: FULL/INCREMENTAL

POST EVENT

POST EVENT コマンドは、ストアード・プロシージャーの内部でのみ許可されます。詳しくは、 204 ページの『CREATE PROCEDURE』を参照してください。

PUT_PARAM()

```
put_param(param_name, param_value)
```

サポート条件

このコマンドには、solidDB 拡張レプリケーションが必要です。

使用法

solidDB インテリジェント・トランザクションでは、パラメーター掲示板を使用して相互にパラメーターを受け渡すことで、トランザクションの SQL ステートメントまたはプロシージャーが相互に通信できます。掲示板は、トランザクションのすべてのステートメントで可視であるパラメーターのストレージです。

パラメーターは、カタログに固有です。異なるレプリカ・カタログおよびマスター・カタログが、それぞれ、相互に可視ではない掲示板パラメーターのセットを所有します。

put_param() 関数を使用して、掲示板にパラメーターを入れます。既にそのパラメーターが存在する場合は、新しい値で古い値が上書きされます。

これらのパラメーターは、マスターに伝搬されません。レプリカからマスターにプロパティーを伝搬するには、SAVE PROPERTY ステートメントを使用します。詳しくは、 307 ページの『SAVE PROPERTY』を参照してください。

put_param() は SQL 関数なので、プロシージャーまたは SQL ステートメントの中でのみ使用できます。

パラメーター名とパラメーター値は、どちらも VARCHAR 型です。

マスターでの使用

put_param() 関数は、現行トランザクションのパラメーター掲示板にパラメーターを設定するために、マスターで使用できます。

レプリカでの使用

put_param() 関数は、現行トランザクションのパラメーター掲示板にパラメーターを設定するために、レプリカで使用できます。

「PUT_PARAM()」と「SAVE PROPERTY property_name VALUE property_value;」の違い

プロシージャー間でパラメーターを受け渡すには、通常、(実行中の) トランザクション内で put_param を使用します。トランザクションが終了 (コミットまたはロールバック) すると、これらのパラメーター値は掲示板から消えます。

トランザクション全体のプロパティーを設定するには、通常、レプリカで SAVE PROPERTY ステートメントを使用します。これらのプロパティーは、PROPAGATE TRANSACTIONS ステートメントの WHERE 節で使用できます。トランザクションがマスターで実行されると、トランザクションの開始時に、トランザクションのプロパティーがトランザクションのパラメーター掲示板に入れられます。そのため、トランザクションのすべてのプロシージャーが GET_PARAM(param_name) 関数を使用してそれらのパラメーターにアクセスできます。

例

```
Select put_param('myparam', '123abc');
```

戻り値

各エラー・コードについて詳しくは、「*solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。

表 70. PUT_PARAM() の戻り値

エラー・コード	説明
13086	パラメーターのデータ型が無効です

正常に実行された場合、put_param() は、割り当てられたパラメーターの新しい値を返します。

関連項目

GET_PARAM

SAVE PROPERTY

SET SYNC PARAMETER

REFRESH

```
REFRESH publication [parameters] [FULL]
[OPTIMISTIC|PESSIMISTIC]
[COMMITBLOCK number_of_rows]
[TIMEOUT {DEFAULT | FOREVER | timeout_ms} ]
```

使用法

REFRESH ステートメントは、ストレージなしのリフレッシュ・コマンドです。関連付けられているデータをストリーミングすることで、メモリーを節約します。また、メッセージをディスクに書き込まないため、I/O 帯域幅も節約します。各コマンドは、正常に実行されるまでブロックされます。

オプション・プロパティー OPTIMISTIC|PESSIMISTIC で、レプリカ表をロックする方法が定義されます。

- OPTIMISTIC モード (デフォルト値) は、並行性制御方式が表のタイプと分離レベルに依存することを定義します。OPTIMISTIC モードのディスク・ベース表では、REFRESH は常に成功します。インメモリー表全般と PESSIMISTIC モードのディ

スク・ベース表では、行レベル・ロック方式が使用されます。ロックがかけられない場合、PESSIMISTIC は失敗し、エラーが返されます。

- PESSIMISTIC は、選択されている表のタイプおよび分離レベルにかかわらず、リフレッシュ時に表が排他的にロックされることを定義します。ロックがかけられない場合、リフレッシュ要求は失敗し、エラーが返されます。

REFRESH 要求への応答に、大きなパブリケーションの REFRESH が含まれている場合、COMMITBLOCK プロパティーを使用して、REFRESH のコミット・ブロックのサイズ (1 つのトランザクションでコミットされる行数) を定義できます。これは、レプリカ・データベースのパフォーマンスに良い影響を与えます。COMMITBLOCK プロパティーを使用するときは、データベースにアクセスするオンライン・ユーザーがいないようにすることを推奨します。

COMMITBLOCK を使用しない場合、REFRESH の実行は、現行トランザクションの一部になります。ROLLBACK コマンドを発行することによって、REFRESH の効果を取り消すことができます。REFRESH の効果を永続的にするには、COMMIT WORK を発行する必要があります。REFRESH は、ロールバックおよびコミットを越えて繰り返し発行でき、データベースの休止状態では常に効果が同じであるという意味で、べき等性 (数学用語) があります。つまり、何回繰り返してもその結果の性質に変わりはありません。

COMMITBLOCK 節を使用した場合、(指定されたサイズの) 各転送部分がレプリカで暗黙的にコミットされます。ROLLBACK ステートメントは、最後の転送部分の効果だけを除去します。COMMIT WORK は、最後の転送部分をコミットします。

TIMEOUT プロパティーは、レプリカ・サーバーが応答メッセージを待つ時間の長さを定義します。TIMEOUT を定義しない場合、FOREVER が使用されます。

例

以下は同期、メッセージレス・リフレッシュです。

```
REFRESH publ_states;
PESSIMISTIC;
COMMITBLOCK 1000;
COMMIT WORK;
```

戻り値

各エラー・コードについて詳しくは、「*solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。

表 71. REFRESH の戻り値

エラー・コード	説明
13133	この製品に有効なライセンスではありません
25004	動的パラメーターはサポートされません
25015	構文エラー: *error_message*、行 *line_number*
25024	マスターが定義されていません

表 71. REFRESH の戻り値 (続き)

エラー・コード	説明
25025	ノード名が定義されていません
25026	有効なマスター・ユーザーではありません
25044	SYNC_CONFIG システム・パブリケーションは文字引数だけを受け入れます
25056	自動コミットは許可されません
25071	パブリケーション *publication_name* には登録していません
25072	パブリケーション *publication_name* には、既に登録済みです
13XXX	表レベル・エラー
21XXX	通信エラー
10006	並行性競合 (他の操作と同時)
13047	操作する特権がありません
13056	疑似列に挿入は許可されません
25005	メッセージ *message_name* が既にアクティブです
25018	正しくないメッセージ状態 レプリカ内では、メッセージが終了し、終了トランザクションがコミットされた場合、MESSAGE FORWARD ステートメントを使用してのみメッセージを実行できます

表 71. *REFRESH の戻り値 (続き)*

エラー・コード	説明
25024	マスターが定義されていません このメッセージは、MESSAGE FORWARD ステートメントで *connect_string* が単一引用符ではなく二重引用符で囲まれていた場合に作成されます 例えば、マスター・ノードのノード名が ″master″ (これは予約語なので、二重引用符で区切る必要があります) で、ノードの接続ストリングが以下のとおりであるとします tcp localhost 1315 この場合、以下に示す MESSAGE ステートメントは正しいものです --レプリカ上 ... -- 二重引用符 MESSAGE msg1 BEGIN TO "master"; ... -- 単一引用符 MESSAGE msg2 FORWARD TO 'tcp localhost 1315'; MESSAGE BEGIN ステートメントは、マスターのノード名が何であるかを (レプリカ・サーバー内で) 定義することに注意してください。MESSAGE FORWARD ステートメントには、サーバーへの接続ストリングを含めることができます
25026	有効なマスター・ユーザーではありません
25031	トランザクションがアクティブで、操作が失敗しました。
25035	メッセージ *message_name* は使用中です。 ユーザーは現在、このメッセージを作成中または転送中です。
25040	ユーザー ID *user_id* が見つかりません メッセージ応答を実行中に、マスター・ユーザーを ローカル・レプリカ ID にマップしようとして失敗しました
25041	パブリケーション *publication_name* へのサブスクリプションが見つかりません。
25048	パブリケーション *publication_name* 要求情報が見つかりません
25052	ノード名を *node_name* に設定できませんでした。
25054	表 *table_name* が同期履歴に設定されていません
25055	接続情報は、登録されていない場合にのみ許可されます MESSAGE *message_name* FORWARD TO *connect_info options* 内の接続情報は、レプリカがマスター・データベースにまだ登録されていない場合にのみ許可されます
25056	自動コミットは許可されません

表 71. *REFRESH の戻り値 (続き)*

エラー・コード	説明
25057	レプリカ・データベースは、既にマスター・データベースに登録されています
25060	列 *column_name* は、表 *table_name* 内のパブリケーション *publication_name* 結果セット上に存在しません
13XXX	表レベル・エラー
13124	ユーザー ID *num* が見つかりません このメッセージは、例えばユーザーがドロップされた場合に、生成されます。
25056	自動コミットは許可されません

REGISTER EVENT

イベントの登録によって、このイベントが将来発生した場合に、イベントを待機していなくてもすべて通知するようにサーバーに指示します。「登録」コマンドと「待機」コマンドを分離することで、イベントのキューイングはすぐに開始するが、実際に処理を開始するのは後にすることができます。

イベントを待つには、事前にそのイベントについて登録しなくてもよいことに注意してください。イベントを待つ場合、まだそのイベントについて明示的に登録していない場合は、暗黙に登録されます。このため、明示的にイベントを登録する必要があるのは、それらのイベントのキューをその時点で開始したいが、それらのイベントの WAIT は後刻まで開始したくないという場合だけです。

同期イベントは登録できません。これは、ADMIN EVENT 'wait' コマンドは変数結果セットを返すことができないためです。代わりに、同期イベントを処理するストアード・プロシージャーを使用する必要があります。

REGISTER EVENT コマンドは、ストアード・プロシージャー内でのみ使用できます。詳しくは、CREATE PROCEDURE ステートメントおよび CREATE EVENT ステートメントを参照してください。

REVOKE (ユーザーからロールを)

```
REVOKE { role_name [, role_name ]... }
    FROM {PUBLIC | user_name [, user_name ]... }
```

使用法

REVOKE ステートメントは、ユーザーからロールを除去するために使用します。

例

```
REVOKE GUEST_USERS FROM HOBBES;
```

REVOKE (ロールまたはユーザーから特権を)

```
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]... }
```

注:

solidDB は、REVOKE ステートメントでキーワード CASCADE および RESTRICT をサポートしません。

使用法

REVOKE ステートメントは、ユーザーおよびロールから特権を取り除くために使用します。

例

```
REVOKE INSERT ON TEST FROM GUEST_USERS;
```

関連項目

ユーザー特権について詳しくは、以下も参照してください。

- 259 ページの『GRANT』、および
- 108 ページの『ユーザー特権およびロールの管理』

REVOKE PASSTHROUGH

```
REVOKE PASSTHROUGH
FROM {PUBLIC | user_name [, user_name]... | role_name [, role_name]... }
```

サポート条件

このコマンドでは、SQL パススルー機能を使用する必要があります。

使用法

REVOKE PASSTHROUGH ステートメントは、SQL パススルーのアクセス権限を取り消します。

例

```
REVOKE PASSTHROUGH READ FROM cdcuser1, cdcuser2
```

```
REVOKE PASSTHROUGH WRITE FROM cdcuser1, cdcuser2
```

関連項目

260 ページの『GRANT PASSTHROUGH』

REVOKE REFRESH

```
REVOKE { REFRESH | SUBSCRIBE} ON publication_name FROM {PUBLIC |
     user_name, [ user_name ] ... |
     role_name , [ role_name ] ...}
```

サポート条件

このコマンドには、solidDB 拡張レプリケーションが必要です。

使用法

このステートメントは、マスター・データベースで定義されているユーザーまたはロールから、パブリケーションへのアクセス権限を取り消します。

注:

キーワード「REFRESH」と「SUBSCRIBE」は同義です。ただし、REVOKE ステートメントでは、「SUBSCRIBE」は推奨されません。

マスターでの使用

このステートメントを使用して、ユーザーまたはロールから、パブリケーションへのアクセス権限を取り消します。

レプリカでの使用

このステートメントは、レプリカ・データベースでは使用できません。

例

```
REVOKE REFRESH ON customers_by_area FROM joe_smith;
REVOKE REFRESH ON customers_by_area FROM all_salesmen;
```

戻り値

表 72. REVOKE REFRESH の戻り値

エラー・コード	説明
13137	付与/取り消しモードが正しくありません
13048	付与オプションの特権がありません
25010	パブリケーション *name* が見つかりません

ROLLBACK WORK

```
ROLLBACK WORK
```

使用法

現行トランザクションで行ったデータベースの変更は、ROLLBACK WORK ステートメントで廃棄されます。これによって、トランザクションが終了します。

例

```
ROLLBACK WORK;
```

SAVE

```
SAVE [NO CHECK] [ { IGNORE_ERRORS | LOG_ERRORS | FAIL_ERRORS } ]
[ { AUTOSAVE | AUTOSAVEONLY } ] sql_statement
```

サポート条件

このコマンドには、solidDB 拡張レプリケーションが必要です。

使用法

マスター・データベースに伝搬する必要があるトランザクションのステートメントは、明示的に、レプリカ・データベースのトランザクション・キューに保存する必要があります。そのためには、トランザクション・ステートメントの前に SAVE ステートメントを追加します。

マスター・ユーザーだけがステートメントを保存できます。これは、保存されたステートメントがマスターで実行されるとき、マスターのユーザーの適切なアクセス権限を使用して実行される必要があるためです。保存されたステートメントは、ステートメントが保存されたときにレプリカでアクティブだったマスター・ユーザーのアクセス権限を使用して、マスター・データベースで実行されます。レプリカのユーザーがマスターのユーザーにマップされた場合、SAVE ステートメントは、マスターのユーザーのアクセス権限を使用します。

トランザクション伝搬のエラー処理のデフォルト動作では、失敗したトランザクションはメッセージの実行を停止します。これによって、現在実行中のトランザクションは中止され、同じメッセージにある後続のトランザクションの実行が妨げられます。ただし、別のエラー処理動作を選択することもできます。

SAVE コマンドのオプションは、以下のとおりです。

NO CHECK: このオプションは、そのステートメントをレプリカで準備しないことを意味します。このオプションは、そのコマンドがレプリカでは無意味であるときに有用です。例えば、SQL コマンドが、マスターには存在するがレプリカには存在しないストアード・プロシージャーを呼び出す場合、レプリカではステートメントを準備しないようにできます。このオプションを使用する場合、ステートメントはパラメーター・マーカーを使用できません。

IGNORE_ERRORS: このオプションは、マスターでの実行中にステートメントが失敗した場合、失敗したステートメントを無視し、トランザクションを中止することを意味します。ただし、中止されるのはトランザクションだけで、メッセージ全体ではありません。マスターは、メッセージの実行を続行し、失敗したトランザクションの後の最初のトランザクションから再開します。

LOG_ERRORS: マスターでの実行中にステートメントが失敗した場合、失敗したステートメントを無視し、現行トランザクションを中止することを意味します。失敗したトランザクションのステートメントは、後で実行または調査できるように、SYS_SYNC_RECEIVED_STMTS システム表に保存されます。失敗したトランザクションは、SYNC_FAILED_MESSAGES システム・ビューを使用して確認できます。また、MESSAGE <msg_id> FROM REPLICA <replica_name> RESTART ステートメントを使用して、そこから再実行できます。

IGNORE_ERROR オプションと同様に、トランザクションは中止されますが、メッセージ全体は中止されないことに注意してください。マスターは、メッセージの実行を続行し、失敗したトランザクションの後の最初のトランザクションから再開します。

FAIL_ERRORS: このオプションは、ステートメントが失敗した場合、マスターがメッセージの実行を停止することを意味します。これはデフォルトの動作です。

AUTOSAVE: このオプションは、マスターが別のマスターのレプリカでもある場合に (中間層ノード)、ステートメントがマスターで実行され、後で伝搬するために自動的に保存されることを意味します。

AUTOSAVEONLY: このオプションは、マスターが別のマスターのレプリカでもある場合に (中間層ノード)、ステートメントがマスターで実行されず、後で伝搬するために自動的に保存されることを意味します。

マスターでの使用

このステートメントは、マスターでは使用できません。

レプリカでの使用

このステートメントは、レプリカで、マスターに伝搬するためのステートメントの保存に使用します。

例

```
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)
```

戻り値

各エラー・コードについて詳しくは、「*solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。

表 73. SAVE の戻り値

エラー・コード	説明
25001	内部エラー マスター・データベースが、ステートメントの保存に必要なデータベース・サイズ制限を超えました

表 73. SAVE の戻り値 (続き)

エラー・コード	説明
25003	SAVE ステートメントを保存できません
25070	ステートメントは、トランザクションで 1 つのマスターにだけ保存できます

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
```

サポート条件

このコマンドには、solidDB 拡張レプリケーションが必要です。

使用法

以下のコマンドで、現行のアクティブ・トランザクションにプロパティーを割り当てることができます。

```
SAVE PROPERTY property_name VALUE 'value_string'
```

マスター・データベースのトランザクションのステートメントは、GET_PARAM() 関数を呼び出すことで、これらのプロパティーにアクセスできます。プロパティーは、以下のコマンドを適用するレプリカ・データベースでのみ使用可能です。

```
MESSAGE APPEND unique_message_name PROPAGATE TRANSACTIONS
WHERE property > 'value_string'
```

トランザクションがマスター・データベースで実行されると、保存されたプロパティーがトランザクションのパラメーター掲示板に入れられます。保存されたプロパティーが既に存在する場合は、新しい値で古い値が上書きされます。

現在の接続のすべてのトランザクションで保存されるデフォルト・プロパティーを定義することもできます。このステートメントは、以下のとおりです。

```
SAVE DEFAULT PROPERTY property_name VALUE 'value_string'
```

SAVE DEFAULT PROPAGATE PROPERTY WHERE ステートメントを使用して、デフォルトのトランザクション伝搬基準を保存できます。これは、例えば、現在の接続で作成されたトランザクションの伝搬優先順位を設定するために使用できます。

SAVE DEFAULT PROPAGATE PROPERTY WHERE *property* > '*value*' を接続レベルで使用すると、すべての MESSAGE *unique_message_name* APPEND PROPAGATE TRANSACTIONS ステートメントで、デフォルトの WHERE ステートメントが追加されるようにできます。WHERE ステートメントが PROPAGATE ステートメントにも入力されている場合は、DEFAULT PROPAGATE PROPERTY で設定されたステートメントをオーバーライドします。

プロパティーまたはデフォルト・プロパティーは、値をストリング NONE にしてプロパティーを保存しなおすことで、削除できます。

マスターでの使用

このステートメントは、マスター・データベースでは使用できません。

レプリカでの使用

このステートメントをレプリカで使用して、マスターに伝搬するために保存されるトランザクションのプロパティーを設定できます。プロパティーの値は、マスター・データベースで読み取ることができます。

「PUT_PARAM()」と「SAVE PROPERTY property_name VALUE property_value;」の違い

「SAVE PROPERTY」と「PUT_PARAM()」の違いについては、PUT_PARAM() 関数の説明を参照してください。

例

```
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'
```

戻り値

各エラー・コードについて詳しくは、「*solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。

表 74. SAVE PROPERTY の戻り値

エラー・コード	説明
13086	パラメーターのデータ型が無効です

結果セット

SAVE PROPERTY は、結果セットを返しません。

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
```

使用法

SELECT ステートメントを使用して、0 個以上のレコードを 1 つ以上の表から選択できます。

非標準節 LIMIT *row_count* OFFSET *skipped_rows* を使用して、サイズが *row_count* で位置が *skipped_rows* + 1 行のスライディング・ウィンドウで、結果セットの一部をマスクできます。*skipped_rows* が負の値の場合、エラーが発生しますが、row_count が負の値の場合は、作成された結果セット全体になります。2 つの書式を使用できます。例えば、`LIMIT 24 OFFSET 10` は、 `LIMIT 10, 24` と同じです。

表に階層データが含まれている場合、階層型照会節を使用して、階層の順に行を選択できます。階層型照会節で、START WITH は階層のルート行を指定し、CONNECT BY は階層の親行と子行の関係を指定します。CONNECT BY 条件に副照会を含めることはできません。

LEVEL は、階層型照会のコンテキストでのみ有効な疑似列です。相互参照されている行のツリーとして結果セットが表示される場合、LEVEL 列にツリー・レベル番号が作成されます。ツリー・レベル番号は、最上位の行に「1」が割り当てられます。

ORDER SIBLINGS BY 節を使用すると、各レベルの行がそれに応じた順序で並びます。

階層型照会では、親行を参照する PRIOR 演算子で、条件の 1 つの式を修飾する必要があります。PRIOR は単項演算子で、単項算術演算子 + および - と同じ優先順位です。階層型照会で、直後の式について、現在行である親行を評価します。PRIOR は、等価演算子で列値を比較するときによく使用されます。PRIOR キーワードは、演算子のどちらの側にでも置くことができます。

例

```
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
```

START WITH の例

```
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
```

LEVEL と ORDER SIBLINGS BY の例

```
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
```

SET

使用法

SET コマンドは、そのコマンドが実行されるユーザー・セッション (接続) に適用されます。他のユーザー・セッションには影響しません。

SET ステートメントはいつでも発行できますが、そのすべてが直ちに有効となるわけではありません。直ちに有効となるステートメントは以下のとおりです。

- SET CATALOG
- SET IDLE TIMEOUT
- SET SCHEMA
- SET STATEMENT MAXTIME

以下のステートメントは、次の COMMIT WORK の後で有効となります。

- SET DURABILITY
- SET OPTIMISTIC LOCK TIMEOUT
- SET LOCK TIMEOUT
- SET ISOLATION LEVEL
- SET { READ ONLY | READ WRITE | WRITE}

SET ステートメントはロールバックされません。つまり、ステートメントを発行したトランザクションが異常終了するかロールバックされても、ステートメントが無効になることはありません。トランザクションでは、SET ステートメントを DDL/DML SQL ステートメントの前に発行することを推奨します。

この設定は、セッション (接続) が終了するまで、または別の SET コマンドによって設定が変更されるまで有効です。場合によっては、より優先度が高いコマンド (SET TRANSACTION など) が実行されるまで有効となることもあります。

SET および SET TRANSACTION の違い

solidDB SQL には、トランザクションの分離レベル、読み取りレベル、および持続性レベルを設定する 2 種類のコマンドがあります。このセクションでは SET コマンドについて説明します。

```
SET { READ ONLY | READ WRITE | WRITE};
SET ISOLATION LEVEL {READ COMMITTED...};
SET DURABILITY ...;
```

もう 1 つのコマンドは SET TRANSACTION です。これについては 324 ページの『SET TRANSACTION』で説明します。

```
SET TRANSACTION { READ ONLY | READ WRITE | WRITE};
SET TRANSACTION ISOLATION LEVEL {READ COMMITTED ...};
SET TRANSACTION DURABILITY ...;
```

この 2 つのコマンドの違いについては、325 ページの『SET および SET TRANSACTION の違い』を参照してください。

SET の例

```
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;
```

SET (読み取り/書き込みレベル)

```
SET {READ ONLY | READ WRITE | WRITE}
```

SET {READ ONLY | READ WRITE | WRITE} を使用して、接続を読み取り専用にするか、読み取り書き込み可能にするか、書き込み専用にするかを指定できます。

312 ページの『SET ISOLATION LEVEL』も参照してください。

SET CATALOG

```
SET CATALOG catalog_name
```

SET CATALOG は、接続で現行カタログ・コンテキストを設定します。

SET DELETE CAPTURE

```
SET DELETE CAPTURE {NONE | CHANGES}
```

使用法

SET DELETE CAPTURE ステートメントは、solidDB Universal Cache でデータ・エージング・モードを設定します。データ・エージングでは、表の行はフロントエンドでは削除されますが、バックエンドでは保持されます。

- NONE: データ・エージングは有効です。
- CHANGES: データ・エージングは無効です。

SET DELETE CAPTURE ステートメントは、次の SQL ステートメントから即座に有効になり、類似ステートメントまたは SET TRANSACTION DELETE CAPTURE によって戻されるまで有効です。

例

関連項目

SET TRANSACTION DELETE CAPTURE

SET DURABILITY

```
SET DURABILITY { RELAXED | STRICT | DEFAULT}
```

SET DURABILITY は、トランザクションの持続性レベルを設定します。可能な設定について詳しくは、「*solidDB 管理者ガイド*」の『ロギングおよびトランザクション持続性』の説明を参照してください。

SET ISOLATION LEVEL

```
SET ISOLATION LEVEL {
    READ COMMITTED |
    REPEATABLE READ |
    SERIALIZABLE }
```

SET ISOLATION LEVEL を使用して、分離レベルを指定できます。分離レベルについて詳しくは、 142 ページの『トランザクション分離レベル』を参照してください。

割り当てられているワークロード・サーバーが 2 次サーバーの場合、プログラムで 1 次サーバーに変更できます。セッション・レベルで、以下のステートメントによってワークロード接続サーバーが 1 次サーバーに変更されます。

```
SET WRITE (nonstandard)
SET ISOLATION LEVEL REPEATABLE READ
SET ISOLATION LEVEL SERIALIZABLE
```

トランザクションの最初のステートメントの場合、ステートメントはすぐに有効になります。そうでない場合、次のトランザクションから有効になります。

上記のステートメントを適用できない場合は、SQL_SUCCESS が返され、アクションは何も実行されません。例えば、SET WRITE がスタンドアロン・サーバーに適用された場合です。この場合、SET WRITE の意味は、SET READ WRITE と同じです。

SET WRITE ステートメントの効果は、ステートメント SET READ WRITE または ... READ ONLY で戻すことができます (SQL:1999)。以下の分離レベル・ステートメントにも同じ効果があります。

```
SET ISOLATION LEVEL READ COMMITTED
```

SET PASSTHROUGH

```
SET PASSTHROUGH {READ  <passthrough level> [WRITE <passthrough level>]}
| {WRITE <passthrough level> | [READ  <passthrough level>]}
| <passthrough level>
```

上記の詳細は以下のとおりです。

```
passthrough level ::=  NONE | CONDITIONAL | FORCE | DEFAULT
```

使用法

SET PASSTHROUGH ステートメントは、solidDB Universal Cache で SQL パススルー・モードを設定します。

- NONE: SQL パススルーは使用されません。コマンドは、フロントエンドからバックエンドに渡されません。
- CONDITIONAL: SQL パススルーは、表欠落エラーまたは構文エラーによってアクティブ化されます。
- FORCE: すべてのステートメントをフロントエンドからバックエンドに渡すために SQL パススルーが使用されます。
- DEFAULT: SQL パススルーの現行セッションのデフォルトが使用されます。

SET PASSTHROUGH ステートメントは、次の SQL ステートメントから即座に有効になり、類似ステートメントまたは SET TRANSACTION PASSTHROUGH によって戻されるまで有効です。

例

関連項目

SET TRANSACTION PASSTHROUGH

SET SAFENESS

```
SET SAFENESS {1SAFE | 2SAFE | DEFAULT}
```

SET SAFENESS は、レプリケーション・プロトコルを同期 (2-safe) または非同期 (1-safe) に決定します。

- 1-safe: トランザクションは、まず 1 次サーバーでコミットされ、次に 2 次サーバーに転送されます。
- 2-safe: トランザクションは、2 次サーバーで確認されるまでコミットされません (デフォルト)。

SET SAFENESS は、現行セッションの安全性レベルを設定します。

SET SCHEMA

```
SET SCHEMA {'schema_name' | USER | 'user_name'}
```

使用法

solidDB は、SQL89 スタイルのスキーマをサポートします。スキーマは、データベース内のエンティティー (表、ビューなど) を一意的に識別するために使用します。スキーマを使用して、各ユーザーは、自分の名前が他のユーザー/スキーマで選択された名前とオーバーラップするという心配なしに、エンティティーを作成できます。

エンティティー (表など) を一意的に識別するには、カタログ名とスキーマ名を指定して「修飾」します。以下は、完全修飾された表名の例です。

```
FinanceCatalog.AccounstReceivableSchema.CustomersTable
```

ANSI SQL-92 規格に従い、user_name または schema_name は単一引用符で囲むことができます。

デフォルト・スキーマは、SET SCHEMA ステートメントで変更できます。 SET SCHEMA USER ステートメントを使用すると、スキーマを現在のユーザー名に変更できます。または、データベースの有効なユーザー名である「*user_name*」にスキーマを設定できます。

エンティティー名 *[schema_name.]table_identifier* を解決するアルゴリズムは、以下のとおりです。

1. *schema_name* が指定されている場合、そのスキーマでのみ *table_identifier* が検索されます。
2. *schema_name* が指定されていない場合、
   a. まず、デフォルト・スキーマで *table_identifier* が検索されます。デフォルト・スキーマは、最初はユーザー名と同じですが、SET SCHEMA ステートメントで変更できます。
   b. 次に、データベースのすべてのスキーマで、*table_identifier* が検索されます。*identifier* およびタイプ (表、ストアード・プロシージャーなど) が同じであるエンティティーが複数検出された場合、新しいエラー・コード 13110 (未確定のエンティティー名 *table_identifier*) が返されます。

SET SCHEMA ステートメントは、デフォルトのエンティティー名解決にのみ影響し、データベース・エンティティーへのアクセス権限は変更しません。 EXECDIRECT ステートメントまたは準備ステートメントによって現行セッションで準備されたステートメントにある修飾されていない名前にデフォルト・スキーマ名を設定します。

例

```
SET SCHEMA 'CUSTOMERS';
```

関連項目

カタログも、表およびその他のデータベース・エンティティーの名前を修飾する (一意的に識別する) ために使用されます。そのため、SET CATALOG コマンドについても参照してください。

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 { | DEFAULT}
SET SQL CONVERTORSTOUNIONS
    {YES [COUNT ] | NO | DEFAULT}
```

使用法

すべての設定は、ユーザー・セッションごとに読み取られます (solid.ini ファイルの設定が、solidDB が始動するたびに自動的に読み取られるのとは異なります)。

SET SQL INFO: SET SQL INFO コマンドを使用して、問題のデバッグや照会のチューニングを行うためのトレース情報をオンにできます。SQL INFO で使用するデフォルト・ファイルは、すべてのユーザーで共有するグローバルな soltrace.out です。ファイル名が指定された場合、新しいファイルが設定されるまで、以後のすべての INFO ON 設定でそのファイルが使用されます。ファイル名は単一引用符で囲んで指定することを推奨します。囲まないと、ファイル名が大文字に変換されます。出力される情報はファイルに追加され、ファイルは決して切り捨てられません。そのため、情報ファイルが不要になった後、ユーザーが手動でファイルを削除する必要があります。ファイル・オープンが失敗した場合、出力される情報はエラーなしで廃棄されます。

デフォルトの SQL INFO LEVEL は 4 です。有用な出力情報を生成するには、新しいファイル名で情報をオンに設定し、EXPLAIN PLAN FOR 構文を使用して SQL ステートメントを実行します。この方式を使用すると、すべての必要なエスティメーター情報が生成され、(巨大な出力ファイルが生成される原因になる) フェッチからの出力は生成されません。

SET SQL SORTARRAYSIZE: このコマンドは、照会の結果セットを順序付けするときに SQL が使用する配列のサイズを設定します。単位は「行」です。例えば、値を 1000 に指定すると、サーバーは 1000 行をソートするのに十分な大きさの配列を作成します。

SET SQL JOINPATHSPAN: このコマンドは廃止されました。構文は受け入れられますが、コマンドは無効です。

SET SQL CONVERTORSTOUNIONS を使用して、「OR」演算子を含む照会を「UNION」演算子を使用する等価な照会に変換できます。以下の操作は、論理的に同等です。

```
select ... where x = 1 OR y = 1;
select ... where x = 1 UNION select... where y = 1;
```

CONVERTORSTOUNIONS を設定することで、データの量および分散に基づいて UNION の方が効率的であると思われた場合、OR 演算子の代わりに等価な UNION 演算子を使用するようにオプティマイザーに指示します。SQL

CONVERTORSTOUNIONS (「Convert ORs to UNIONs」) の COUNT パラメーターは、UNION 演算子に変換できる OR 演算子の最大数を指定します。solid.ini 構成パラメーター ConvertORsToUNIONs を使用して、CONVERTORSTOUNIONS を指定することもできます (詳しくは、「*solidDB 管理者ガイド*」の、このパラメーターの説明を参照してください)。デフォルト値は 100 で、ほぼすべての場合を満たします。

例

```
SET SQL INFO ON FILE 'sqlinfo.txt' LEVEL 5
```

SET STATEMENT MAXTIME

SET STATEMENT MAXTIME *minutes*

SET STATEMENT MAXTIME は、接続固有の最大実行時間を分単位で設定します。設定は、新しい最大時間が設定されるまで有効です。ゼロ時間は、最大時間を設定しないことを意味し、これがデフォルトです。

SET SYNC

以下の章で、さまざまな SET SYNC コマンドについて説明します。

SET SYNC master_or_replica

SET SYNC *master_or_replica yes_or_no*

ここで、

master_or_replica ::= MASTER | REPLICA
yes_or_no ::= YES | NO

サポート条件: このコマンドには、solidDB 拡張レプリケーションが必要です。

使用法: データベース・カタログを作成し、同期に使用するように構成するときに、このコマンドを使用して、データベースがマスターか、レプリカか、両方かを指定する必要があります。DBA、または SYS_SYNC_ADMIN_ROLE を持つユーザーだけが、データベース・ロールを設定できます。

このデータベースからのパブリケーションからリフレッシュを行うレプリカ、または、このデータベースにトランザクションを伝搬するレプリカがドメインにある場合、そのデータベース・カタログはマスター・データベースです。マスター・データベースにあるパブリケーションからリフレッシュできる場合、そのデータベース・カタログはレプリカ・カタログです。複数層同期では、中間レベルのデータベースが、マスター・データベースとレプリカ・データベースの両方として、2 つの役割を果たします。

このコマンドを使用するには、SET SYNC NODE コマンドを使用して、マスターまたはレプリカのノード名を既に設定している必要があることに注意してください。詳しくは、 320 ページの『SET SYNC NODE』を参照してください。

データベースを 2 つの役割に設定するときは、ステートメントを 1 回使用することも、2 回使用することもできます。以下に例を示します。

```
SET SYNC MASTER YES;
SET SYNC REPLICA YES;
```

データベースを 2 つの役割に設定する場合、SET SYNC REPLICA YES が SET SYNC MASTER YES をオーバーライドしないことに注意してください。以下の明示的ステートメントだけが、マスター・データベースの状況をオーバーライドできます。

```
SET SYNC MASTER NO;
```

オーバーライドされると、現行のデータベースはレプリカのみとして設定されます。

例:

```
-- レプリカとして構成
SET SYNC REPLICA YES;
-- マスターとして構成
SET SYNC MASTER YES;
```

戻り値: 各エラー・コードについて詳しくは、「*solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。

表 75. SET SYNC の戻り値

エラー・コード	説明
13047	操作する特権がありません
13107	セット演算が正しくありません
13133	この製品に有効なライセンスではありません
25051	未完了のメッセージが見つかりました

SET SYNC CONNECT

```
SET SYNC CONNECT 'connect_string [,connect_string]' TO MASTER
master_name
SET SYNC CONNECT 'connect_string' TO REPLICA replica_name
```

サポート条件: このコマンドには、solidDB 拡張レプリケーションが必要です。

使用法: このステートメントは、データベース名に関連付けられているネットワーク名を変更します。レプリカ (またはマスター) が接続するデータベースのネットワーク名を変更したときは、必ず、このステートメントをレプリカ (またはマスター) で使用します。ネットワーク名は、 `solid.ini` 構成ファイルの Listen パラメーターで定義されます。

SET SYNC CONNECT ... TO MASTER の 2 番目の接続ストリングは、1 次マスター・サーバーに障害が発生したときに、スタンバイ・マスター・サーバーにレプリカ・サーバーの透過的フェイルオーバーを行うために役に立ちます。接続ストリングの順序は重要ではありません。接続は、自動的に、現在アクティブな 1 次サーバーで維持されます。

マスターでの使用: このステートメントは、レプリカのネットワーク名を変更するために、マスターで使用します。

レプリカでの使用: このステートメントは、マスターのネットワーク名を変更するために、レプリカで使用します。

例:

```
SET SYNC CONNECT 'tcp server.company.com 1313' TO MASTER hq_master;
```

戻り値: 各エラー・コードについて詳しくは、「*solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。

表 76. SET SYNC CONNECT の戻り値

エラー・コード	**説明**
13047	操作する特権がありません
13107	セット演算が正しくありません
21300	ネットワーク・プロトコルが正しくありません
25007	マスター *master_name* が見つかりません
25019	データベースがレプリカ・データベースではありません

SET SYNC MODE

```
SET SYNC MODE { MAINTENANCE | NORMAL }
```

サポート条件: このコマンドには、solidDB 拡張レプリケーションが必要です。

使用法: このコマンドは、現行のカタログの同期モードを保守モードまたは通常モードに設定します。

このコマンドは、同期に関係するカタログ (つまり、「マスター」カタログまたは「レプリカ」カタログ、または 3 レベル以上の階層で、マスターとレプリカの両方であるカタログ) にのみ適用されます。

このコマンドは、現行のカタログにのみ適用されます。複数のカタログの同期モードを保守に設定するには、(SET CATALOG コマンドを使用して) 各カタログに切り替え、そのカタログに SET SYNC MODE MAINTENANCE コマンドを発行する必要があります。

カタログの同期モードが保守の間、以下のルールが適用されます。

- カタログは同期メッセージの送受信を行わないため、同期アクティビティー (リフレッシュ、リフレッシュ要求への応答など) に関わりません。
- DDL コマンド (ALTER TABLE など) は、パブリケーションが参照する表で許可されます。
- 同期モードが変更されると、サーバーはシステム・イベント SYNC_MAINTENANCEMODE_BEGIN または SYNC_MAINTENANCEMODE_END を送信します。
- マスター・カタログのパブリケーションが REPLACE オプションで変更 (ドロップまたは再作成) されると、パブリケーションのメタデータ (内部パブリケーション定義データ) は、変更されたパブリケーションから次にレプリカがリフレッシ

ュするときに、自動的に各レプリカにリフレッシュされます。(これは、パブリケーションが置き換えられたときに、データベースが保守同期モードかどうかに関わらず、真です。)

- カタログごとに、パラメーター掲示板に読み取り専用の SYNC_MODE パラメーターがあり、アプリケーションはカタログのモードを検査できます。このパラメーターの値は、カタログが保守同期モードの場合の「MAINTENANCE」、保守同期モードでない場合の「NORMAL」のいずれかです。カタログがマスターでもレプリカでもない場合、値は NULL です。
- ユーザーは、同期モードを保守または通常に設定するには、ユーザーに DBA または同期管理特権が必要です。
- ユーザーは、保守同期モードのカタログを同時に複数持つことができます。
- モードをオンに設定したセッションが切断されると、モードはオフに設定されます。
- 通常の同期履歴操作は使用不可です。例えば、同期履歴がオンになっている表で削除または更新操作を実行した場合、同期履歴表は「元の」行 (削除または更新される前の行) を保管しません。ただし、この削除および更新は、同期履歴表に適用されます。すなわち、

```
DELETE * FROM T WHERE c = 5
```

この操作は、基本表からと同様に、履歴表からも行を削除します。以下の表で、同期モードが保守に設定されているときに、各操作がマスターおよびレプリカの同期履歴表にどのように適用されるかを示します。

表 77. 同期履歴表への各種操作の適用方法

操作	マスター	レプリカ
INSERT	基本表に行が挿入されます。	基本表に行が挿入され、正式のマークが付けられます。
UPDATE	基本表と履歴の両方が更新されます。	基本表と履歴の両方が更新されます。一時的/正式の状況は更新されないため、一時的な行は一時的なままで、正式な行は正式なままです。
DELETE	基本表と履歴から行が削除されます。	基本表と履歴から行が削除されます。
列の追加、変更、ドロップ	同じ操作が履歴に対しても実行されます。	同じ操作が履歴に対しても実行されます。
表モードの変更	履歴モードは変更されません。	履歴モードは変更されません。
索引の作成	同じ索引が履歴にも作成されます。	同じ索引が履歴にも作成されます。
トリガーの作成	トリガーは、履歴では作成されません。	トリガーは、履歴では作成されません。

例:

```
SET SYNC MODE MAINTENANCE SET SYNC MODE NORMAL
```

戻り値: 各エラー・コードについて詳しくは、「*IBM solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。

表 78. SET SYNC MODE の戻り値

エラー・コード	説明
13047	操作する特権がありません
13133	この製品に有効なライセンスではありません。
25021	データベースがマスター・データベースまたはレプリカ・データベースではありません。この操作は、マスター・データベースおよびレプリカ・データベースにのみ適用されます
25088	カタログは既に保守モードです。既にモードがオンに設定されています
25089	保守モードをオフに設定できません。別のユーザーがモードをオンに設定したため、オフに設定できません
25090	カタログは既に保守モードです。別のユーザーがモードをオンに設定したため、オンに設定できません
25091	カタログは保守モードではありません。モードをオフに設定しようとしましたが、現在、オンではありません

SET SYNC NODE

```
SET SYNC NODE {unique_node_name | NONE}
```

サポート条件: このコマンドには、solidDB 拡張レプリケーションが必要です。

使用法: ノード名の割り当ては、レプリカ・データベースの登録プロセスの一部です。solidDB 環境の各カタログには、ドメイン内で固有のノード名が必要です。1 つのカタログが持つことができるノード名は、1 つだけです。2 つのカタログが同じノード名を持つことはできません。

以下の条件を満たす場合、SET SYNC NODE *unique_node_name* オプションを使用して、ノード名を名前変更できます。

- ノードがレプリカ・データベースで、マスターに登録されていない場合。

かつ/または

- ノードがマスター・データベースで、このマスター・データベースに登録されているレプリカがない場合。

以下に、ノード名を名前変更する例を示します。

```
SET SYNC NODE A; -- ここで、ノード名は A になります。
SET SYNC NODE B; -- ここで、ノード名は B になります。
COMMIT WORK;
SET SYNC NODE C; -- ここで、ノード名は C になります。
ROLLBACK WORK; -- ここで、ノード名は B にロールバックされます。
SET SYNC NODE NONE; -- ここで、ノード名はなくなります。
COMMIT WORK;
```

unique_node_name は、データベースのその他のオブジェクト (表など) で使用される命名ルールに従う必要があります。ノード名は、単一引用符で囲まないでください。

NONE を指定すると、このコマンドは、現行のノード名を削除します。

「NONE」などの予約語をノード名として使用する場合は、キーワードを二重引用符で囲み、区切り ID として扱われるようにします。以下に例を示します。

```
SET SYNC NODE "NONE"; -- ここで、ノード名は「NONE」になります。
```

ノード名割り当ては、以下のステートメントで確認できます。

```
SELECT GET_PARAM('SYNC NODE')
```

SET SYNC NODE NONE オプションは、現行カタログからノード名を削除します。このオプションは、同期されたデータベースをドロップし、登録を削除するときに使用します。

注:

SET SYNC NODE NONE オプションを使用するときは、ノード名に関連付けられているカタログがマスター、レプリカ、またはその両方として定義されていないことを確認します。ノード名を削除するには、カタログを SET SYNC MASTER NO または SET SYNC REPLICA NO、またはその両方として定義する必要があります。マスター・カタログまたはレプリカ・カタログまたはその両方でノード名を NONE に設定しようとすると、solidDB は、エラー・メッセージ 25082 を返します。

マスターでの使用: このステートメントは、現行カタログにノード名を設定する、または現行カタログからノード名を削除するときに、マスターで使用します。

レプリカでの使用: このステートメントは、現行カタログにノード名を設定する、または現行カタログからノード名を削除するときに、レプリカで使用します。

例:

```
SET SYNC NODE SalesmanJones;
```

戻り値: 各エラー・コードについて詳しくは、「*solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。

表 79. SET SYNC NODE の戻り値

エラー・コード	説明
13047	操作する特権がありません
13107	セット演算が正しくありません
25059	登録後にノード名を変更することはできません
25082	ノードがマスターまたはレプリカの場合、ノード名は削除できません

SET SYNC PARAMETER

```
SET SYNC PARAMETER parameter_name 'value_as_string';
SET SYNC PARAMETER parameter_name NONE;
```

サポート条件: このコマンドには、solidDB 拡張レプリケーションが必要です。

使用法: このステートメントは、パラメーター掲示板を通じて、カタログで実行されるすべてのトランザクションで可視になる、永続的なカタログ・レベル・パラメーターを定義します。各カタログには、異なるパラメーターのセットがあります。

既にそのパラメーターが存在する場合は、新しい値で古い値が上書きされます。既存のパラメーターは、その値を NONE に設定することで、削除できます。すべてのパラメーターは、 SYS_BULLETIN_BOARD システム表に保管されます。

これらのパラメーターは、マスターに伝搬されません。

システム固有のパラメーターに加えて、同期機能を構成する多数のシステム・パラメーターもシステム表に保管できます。使用可能なシステム・パラメーターのリストについては、SQL リファレンスの巻末を参照してください。

マスターでの使用: SET SYNC PARAMETER は、マスターで、データベース・パラメーターの設定に使用されます。

レプリカでの使用: SET SYNC PARAMETER は、レプリカで、データベース・パラメーターの設定に使用されます。

例:

```
SET SYNC PARAMETER db_type 'REPLICA'
SET SYNC PARAMETER db_type NONE
```

戻り値: 各エラー・コードについて詳しくは、「*solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。

表 80. SET SYNC PARAMETER の戻り値

エラー・コード	説明
13086	パラメーターのデータ型が無効です

関連項目: GET_PARAM

PUT_PARAM

SET SYNC PROPERTY

マスターでの構文は以下のとおりです。

```
SET SYNC PROPERTY <propertyname> = { 'value' | NONE } FOR REPLICA
<replicaname>
```

レプリカでの構文は以下のとおりです。

```
SAVE SET SYNC PROPERTY <propertyname> = {'value' | NONE }
```

サポート条件: このコマンドには、solidDB 拡張レプリケーションが必要です。

使用法: このコマンドを使用して、レプリカのプロパティー名と値を指定できます。プロパティーを持つレプリカはグループ化でき、グループは、START AFTER COMMIT ステートメントを使用するときに指定できます。例えば、自転車産業に関連したいくつかのレプリカと、サーフボード業界に関連したいくつかのレプリカがあり、それぞれのレプリカ・グループを別々に更新するとします。この場合、プロ

パティー名を使用して、これらのレプリカをグループ化できます。グループのすべてのメンバーが同じプロパティーを持ち、そのプロパティーに同じ値を持ちます。

詳しくは、「*solidDB 拡張レプリケーション・ユーザー・ガイド*」の『レプリカ・プロパティー名』という表題のセクションを参照してください。

例: マスター:

```
SET SYNC PROPERTY color = 'red' FOR REPLICA replica1;
SET SYNC PROPERTY color = NONE FOR REPLICA replica1;
```

レプリカ:

```
SAVE SET SYNC PROPERTY color = 'red';
SAVE SET SYNC PROPERTY color = NONE;
```

SET SYNC USER

```
SET SYNC USER master_username IDENTIFIED BY password
SET SYNC USER NONE
```

サポート条件: このコマンドには、solidDB 拡張レプリケーションが必要です。

使用法: このステートメントは、レプリカ・データベースをマスター・データベースに登録するときに使用する登録処理用のユーザー名とパスワードを定義するために使用します。このコマンドを使用するには、SYS_SYNC_ADMIN_ROLE アクセス権限が必要です。

注:

SET SYNC USER ステートメントは、レプリカの登録にのみ使用します。登録以外のすべての同期操作では、レプリカ・データベースにある有効なマスター・ユーザー ID が必要です。異なるレプリカ用マスター・ユーザーを指定するには、レプリカ・データベースのレプリカ ID とマスター・データベースのマスター ID をマップする必要があります。詳しくは、「*solidDB 拡張レプリケーション・ユーザー・ガイド*」の『レプリカ・ユーザー ID のマスター・ユーザー ID へのマッピング』という表題のセクションを参照してください。

登録ユーザー名は、マスター・データベースで定義します。指定した名前には、レプリカ登録タスクを実行できる権限が必要です。マスター・データベースのマスター・ユーザーに登録権限を付与するには、GRANT *rolename* TO *user* ステートメントを使用して、SYS_SYNC_REGISTER_ROLE または SYS_SYNC_ADMIN_ROLE をユーザーに指定します。

登録が正常に完了した後、同期ユーザーを NONE にリセットする必要があります。リセットしない場合、マスター・ユーザーがステートメントの保存、メッセージの伝搬、パブリケーションからのリフレッシュ、パブリケーションへの登録を行ったときに、以下のエラー・メッセージが返されます。

```
User definition not allowed for this operation.
```

マスターでの使用: このステートメントは、マスター・データベースでは使用不可です。

レプリカでの使用: このステートメントは、レプリカで、ユーザー名の設定に使用します。

例:

```
SET SYNC USER homer IDENTIFIED BY marge;
SET SYNC USER NONE;
```

SET TIMEOUT

```
SET IDLE TIMEOUT { timeout_in_seconds |
    timout_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 は、接続固有の最大タイムアウトを秒単位で設定します。この設定は新しいタイムアウトが設定されるまで有効です。タイムアウトを DEFAULT に設定すると、最大時間は設定されません。

SET LOCK TIMEOUT は、ロックが解放されるまでエンジンが待機する秒数を設定します。デフォルトでは、ロックのタイムアウトが 30 秒に設定されます。ロック・タイムアウトの最大値は 1000 秒です。1000 秒を超える値を指定すると、SET LOCK TIMEOUT は失敗します。

デフォルトの細分度は秒です。値の後に「MS」を付加することで、ロック・タイムアウトをミリ秒の細分度で設定できます。以下に例を示します。

```
SET LOCK TIMEOUT 500MS;
SET LOCK TIMEOUT 1500 MS;
```

「MS」のスペーシングは重要ではなく、大文字も小文字も使用できます。「MS」を指定しなければ、ロック・タイムアウトは秒単位となります。タイムアウトの時間に達すると、solidDB はタイムアウトになったステートメントを終了します。詳しくは、 130 ページの『ロック・タイムアウトの設定』を参照してください。

SET TRANSACTION

使用法

この設定は、現行トランザクションにのみ適用されます。

トランザクションのロギングおよび持続性に関する背景情報

サーバーは、トランザクション・ロギングを使用して、異常シャットダウンが発生した場合にデータをリカバリーできるようにします。「ストリクト」持続性は、トランザクションがコミットされるとすぐにサーバーがトランザクション・ログ・ファイルに情報を書き込むことを意味します。「リラックス」持続性は、トランザクションがコミットされてもサーバーがすぐに情報を書き込まないことを意味します。代わりにサーバーは、例えばビジー状態が緩和されるまで、あるいは複数のトランザクションを 1 回の書き込み操作で書き込めるようになるまで待機します。リラックス持続性を使用すると、サーバーが異常シャットダウンした場合に、最新のトランザクションが数件失われる可能性があります。持続性について詳しくは、「*solidDB インメモリー・データベース・ユーザー・ガイド*」を参照してください。

セッションに対して既に設定されている持続性レベルと SET TRANSACTION DURABILITY ステートメントが一致する場合、このステートメントの効果はなく、状況「SUCCESS」が返されます。

SET および SET TRANSACTION の違い

solidDB SQL には、トランザクションの分離レベル、読み取りレベル、および持続性レベルを設定する 2 種類のコマンドがあります。このセクションでは SET TRANSACTION コマンドについて説明します。

```
SET TRANSACTION { READ ONLY | READ WRITE | WRITE}
SET TRANSACTION ISOLATION LEVEL {READ COMMITTED ...}
SET TRANSACTION DURABILITY ...;
```

もう 1 つのコマンドは SET です。これについては 310 ページの『SET』で説明します。

```
SET { READ ONLY | READ WRITE | WRITE}
SET ISOLATION LEVEL {READ COMMITTED ...}
SET DURABILITY ...;
```

「TRANSACTION」キーワードを含んだコマンドはトランザクション・レベルのコマンドと呼ばれます。一方、「TRANSACTION」キーワードを含んでいないコマンドはセッション・レベルのコマンドと呼ばれることがあります。

トランザクション・レベルのコマンドは、セッション・レベルのコマンドと異なるルールに従います。以下にその相違点を示します。

- トランザクション・レベルのコマンドはそのコマンドを発行したトランザクションで有効となりますが、セッション・レベルのコマンドは次のトランザクションで (次の COMMIT WORK の後に) 有効となります。
- トランザクション・レベルのコマンドは現行トランザクションのみに適用されますが、セッション・レベルのコマンドは後続のすべてのトランザクションに適用されます。つまり、セッション (接続) が終了するまで、または別の SET コマンドでトランザクションが変更されるまで適用されます。
- トランザクション・レベルのコマンドはトランザクションの始め (つまり DML ステートメントまたは DDL ステートメントの前) に実行する必要があります。(ただし、他の SET ステートメントの後に実行することは可能です。)このルールに違反するとエラーが返されます。セッション・レベルのコマンドは、トランザクションのどの時点でも実行できます。
- トランザクション・レベルのコマンドはセッション・レベルのコマンドよりも優先されます。ただし、トランザクション・レベルのコマンドは現行トランザクションにのみ適用されます。現行トランザクションが終了すると、その直前の SET コマンド (存在する場合) で設定された値に設定が戻されます。以下に例を示します。

```
COMMIT WORK; -- 直前のトランザクションが終了します。
SET ISOLATION LEVEL SERIALIZABLE;
COMMIT WORK;
-- 分離レベルが SERIALIZABLE になります。
...
COMMIT WORK;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
-- 分離レベルが REPEATABLE READ になります。これは
-- トランザクション・レベルの設定がセッション・
-- レベルの設定より優先されるためです。
COMMIT WORK;
-- 分離レベルが再び SERIALIZABLE になります。
-- これはトランザクション・レベルの設定がそのトランザクション
-- だけに適用されるためです。
```

分離レベルおよび読み取りレベルの設定の全体的な優先順位を以下に示します。リストの上位にあるほど優先順位は高くなります。

1. SET TRANSACTION... (トランザクション・レベルの設定)
2. SET ... (セッション・レベルの設定)
3. サーバー・レベルの設定。これは solid.ini 構成パラメーターの値で指定されます (**IsolationLevel** や **DurabilityLevel** など (READ ONLY/READ WRITE の設定に対応する solid.ini パラメーターはありません))。この設定は、solid.ini ファイルを編集するか、以下のようなコマンドを発行することで変更できます。

   ```
   ADMIN COMMAND 'parameter Logging.DurabilityLevel = 2';
   ```

   solid.ini パラメーターを変更する場合は、次にサーバーを始動するまで新しい設定が有効とならないことに注意してください。
4. サーバーのデフォルト設定。「*IBM solidDB 管理者ガイド*」の*付録* セクション、『*サーバー・サイド構成パラメーター*』を参照してください。

持続性に関する注意事項

- サーバーが予期せずシャットダウンした場合に一部のトランザクションが失われても支障がないユーザー以外は、ストリクト持続性を使用する必要があります。
- 値を **DurabilityLevel** パラメーターで指定された値に設定するための「DEFAULT」オプションはありません。また、現行セッションに適用される持続性レベルを読み取ることもできません。このため、一度 SET DURABILITY ステートメントを実行して明示的に持続性を設定すると、**DurabilityLevel** パラメーターで指定された「デフォルト」持続性レベルに戻すことはできません。持続性を RELAXED と STRICT の間で切り替えることはいつでもできますが、デフォルト・レベルがわからないまま変更を「取り消し」てデフォルト・レベルに戻すことはできません。

SET TRANSACTION コマンドは ANSI SQL に基づいています。ただし、solidDB の実装には ANSI 定義と異なる点がいくつかあります。ANSI 定義では、ANSI が定義する 2 つの「節」(分離レベルと読み取りレベル) を組み合わせることができます。以下に例を示します。

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE, READ WRITE;
```

solidDB ではこの構文がサポートされていません。ただし、 solidDB では、1 つのトランザクションで複数の SET ステートメントを実行できます。以下に例を示します。

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SET TRANSACTION READ WRITE;
```

SET TRANSACTION の例

```
SET TRANSACTION DURABILITY RELAXED;
SET TRANSACTION ISOLATION REPEATABLE READ;
SET TRANSACTION READ WRITE;
```

関連項目

310 ページの『SET』.

142 ページの『トランザクション分離レベル』.

「*solidDB 管理者ガイド*」の『ロギングおよびトランザクション持続性』

SET TRANSACTION (読み取り/書き込みレベル)

```
SET TRANSACTION {READ ONLY | READ WRITE | WRITE}
```

コマンド SET TRANSACTION { READ ONLY | READ WRITE | WRITE} は ANSI SQL に基づいています。このコマンドでは、トランザクションでデータを変更できるかどうかを指定できます。

SET TRANSACTION DELETE CAPTURE

```
SET TRANSACTION DELETE CAPTURE {NONE | CHANGES | DEFAULT}
```

使用法

SET TRANSACTION DELETE CAPTURE ステートメントは、次のトランザクション用に、solidDB Universal Cache でデータ・エージング・モードを設定します。データ・エージングでは、表の行はフロントエンドでは削除されますが、バックエンドでは保持されます。

- NONE: データ・エージングは有効です。
- CHANGES: データ・エージングは無効です。
- DEFAULT: データ収集モードは、前の SET DELETE CAPTURE ステートメントで設定されたモードか、サーバーのデフォルトに戻ります。

SET TRANSACTION DELETE CAPTURE ステートメントは、トランザクションの開始時に有効になり、トランザクションがコミットまたは異常終了するまで影響を与えます。このステートメントが、トランザクションの中間で発行された場合、エラーが返されます。

例

関連項目

SET DELETE CAPTURE

SET TRANSACTION DURABILITY

```
SET TRANSACTION DURABILITY {RELAXED | STRICT}
```

コマンド SET TRANSACTION DURABILITY { RELAXED | STRICT } は、サーバーがトランザクション・ロギングに「ストリクト」持続性と「リラックス」持続性のどちらを使用するかを指定します。このコマンドは、SQL に対する solidDB の拡張機能であり、ANSI 規格には含まれていません。

この選択は、他のユーザー、現在使用しているセッション以外のオープン・セッション、今後使用するセッションのいずれにも影響しません。各ユーザー・セッションで、それぞれのデータを損失しないことの重要性に基づいて、独自に持続性レベルを設定できます。

新しいトランザクション持続性の設定を STRICT にすると、それ以前のまだディスクに書き込まれていないすべてのトランザクションが、現行トランザクションがコミットされた時点で書き込まれることに注意してください。(トランザクション持続

性レベルを STRICT に変更した時点ですぐに以前のトランザクションがディスクに書き込まれるのではなく、現行トランザクションがコミットされて初めて書き込みが行われます。)

割り当てられたワークロード・サーバーが 2 次サーバーである場合、あるトランザクションが実行される間それをプログラムで 1 次サーバーに変更できます。トランザクション・レベルで以下のステートメントを使用すると、1 つのトランザクションが実行される間ワークロード接続サーバーが 1 次サーバーに変更されます。

```
SET TRANSACTION WRITE (nonstandard)
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
```

対象となるトランザクションは、ステートメントで開始されるトランザクションか、それ以外の場合は次のトランザクションです。このトランザクションが 1 次サーバーで実行されると、そのセッションではワークロード接続サーバーがデフォルトのサーバーに戻されます。

上記のステートメントを適用できない場合は、SQL_SUCCESS が返され、アクションは何も実行されません。例えば、SET TRANSACTION WRITE をスタンドアロン・サーバーに適用した場合がこのケースに該当します。この場合、SET TRANSACTION WRITE は SET TRANSACTION READ WRITE と同じセマンティクスになります。

SET TRANSACTION WRITE ステートメントの効果は、SET TRANSACTION READ WRITE または ... READ ONLY (SQL:1999) ステートメントによって無効にできます。以下の分離レベル・ステートメントにも同じ効果があります。

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
```

SET TRANSACTION ISOLATION LEVEL

```
SET TRANSACTION ISOLATION LEVEL {
    READ COMMITTED |
    REPEATABLE READ |
    SERIALIZABLE}
```

コマンド SET TRANSACTION ISOLATION は、ANSI SQL に基づいています。このコマンドは、トランザクションの分離レベル (READ COMMITTED、REPEATABLE READ、または SERIALIZABLE) および読み取りレベル (READ ONLY または READ WRITE) を設定します。分離レベルについて詳しくは、 142 ページの『トランザクション分離レベル』を参照してください。

SET TRANSACTION PASSTHROUGH

```
SET TRANSACTION PASSTHROUGH {READ  <passthrough level> [WRITE <passthrough level>]}
| {WRITE <passthrough level> | [READ  <passthrough level>]}
| <passthrough level>
```

上記の詳細は以下のとおりです。

```
passthrough level ::=  NONE | CONDITIONAL | FORCE | DEFAULT
```

使用法

SET TRANSACTION PASSTHROUGH ステートメントは、solidDB Universal Cache で次のトランザクションの SQL パススルー・モードを設定します。

- NONE: SQL パススルーは使用されません。コマンドは、フロントエンドからバックエンドに渡されません。
- CONDITIONAL: SQL パススルーは、表欠落エラーまたは構文エラーによってアクティブ化されます。
- FORCE: すべてのステートメントをフロントエンドからバックエンドに渡すために SQL パススルーが使用されます。
- DEFAULT: SQL パススルーの現行セッションのデフォルトが使用されます。

SET TRANSACTION PASSTHROUGH ステートメントは、トランザクションの開始時に有効になり、トランザクションがコミットまたは異常終了するまで影響を与えます。このステートメントが、トランザクションの中間で発行された場合、エラーが返されます。

例

関連項目

SET PASSTHROUGH

SET TRANSACTION SAFENESS

```
SET TRANSACTION SAFENESS {1SAFE | 2SAFE | DEFAULT}
```

SET TRANSACTION SAFENESS では、レプリケーション・プロトコルが同期 (2-safe) か非同期 (1-safe) かを指定します。

- 1-safe: トランザクションは、まず 1 次サーバーでコミットされ、次に 2 次サーバーに転送されます。
- 2-safe: トランザクションは、2 次サーバーで確認されるまでコミットされません (デフォルト)。

SET TRANSACTION SAFENESS によって、現行トランザクションの安全性レベルが設定されます。

START AFTER COMMIT

```
START AFTER COMMIT
 [FOR EACH REPLICA WHERE search_condition [RETRY retry_spec]]
{UNIQUE | NONUNIQUE} stmt;

stmt ::= 任意の SQL ステートメント
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 ::= レプリカ・プロパティーの名前
like_test ::= property_name [NOT] LIKE value [ESCAPE value]
value ::= literal
retry_spec ::= seconds,count
```

使用法

START AFTER COMMIT ステートメントでは、現行トランザクションがコミットされたときに実行される SQL ステートメント (ストアード・プロシージャーの呼び出しなど) を指定します (このトランザクションがロールバックされると、指定した SQL ステートメントは実行されません)。

START AFTER COMMIT は、1 つの INTEGER 列で構成される結果セットを返します。この整数はユニークな「ジョブ」ID です。この整数を使用して、無効な SQL ステートメント、不十分なアクセス権限、使用不可能なレプリカなどが原因で開始できなかったステートメントの状況を照会できます。

<stmt> の前に UNIQUE キーワードを使用すると、同一のステートメントが実行されていない、または「保留中」である場合にのみステートメントが実行されます。ステートメントは単純なストリング比較を使用して比較されます。例えば、「call foo(1)」と「call foo(2)」は異なります。サーバーでは、既に実行されているステートメント (または実行を保留されているステートメント) が同じレプリカと別のレプリカのどちらにあるかも考慮されます。同じレプリカ上にある同一のステートメントのみが破棄されます。

重要:

UNIQUE キーワードを使用して重複するステートメントを破棄する場合、最新のステートメントが破棄され、最も古いステートメントが引き続き実行されることに注意してください。例えば、複数の更新を実行するために複数の START AFTER COMMIT 操作を起動すると、最も古い更新のみが実行されて、最後に更新されたデータがレプリカに直ちに送信されない状況が発生する可能性が高くなります。

NONUNIQUE は、重複するステートメントをバックグラウンドで同時に実行できることを意味します。

FOR EACH REPLICA では、WHERE 節の search_condition で指定されたプロパティー条件を満たす各レプリカでステートメントが実行されるように指定します。ステートメントを実行する前に、レプリカとの接続が確立されます。プロシージャー呼び出しが開始される場合は、キーワード「DEFAULT」を使用してそのプロシージャーで「現在の」レプリカ名を取得できます。

RETRY を指定すると、最初の試行でレプリカにアクセスできなかった場合に、N 秒後 (retry_spec の seconds で指定) に操作が再実行されます。count では、再試行される回数を指定します。

START AFTER COMMIT コマンドについて詳しくは、27 ページの『3 章 ストアード・プロシージャー、イベント、トリガー、およびシーケンス』を参照してください。

トランザクション

START AFTER COMMIT を使用してバックグラウンドで開始されたステートメントは、別のトランザクションで実行されます。このトランザクションは自動コミット・モードで実行されます。つまり、開始後にロールバックすることはできません。

バックグラウンド・ステートメントのコンテキスト

バックグラウンドで開始されたステートメントは、START AFTER COMMIT ステートメントを発行したユーザーのコンテキストで実行され、START AFTER COMMIT ステートメントが実行されたカタログとスキーマで実行されます。

以下の例では、「CALL FOO」がカタログ「katmandu」およびスキーマ「steinbeck」で実行されます。

```
SET CATALOG katmandu;
SET SCHEMA steinbeck;
START AFTER COMMIT UNIQUE CALL FOO;
COMMIT WORK;
SET CATALOG irrelevant_catalog;
SET SCHEMA irrelevant_schema
```

持続性

バックグラウンド・ステートメントには持続性がありません。つまり、START AFTER COMMIT で開始されたステートメントの実行は保証されません。

ロールバック

バックグラウンド・ステートメントは、開始後にロールバックすることができません。したがって、START AFTER COMMIT で開始されたステートメントが正常に実行された後は、そのステートメントをロールバックできません。

START AFTER COMMIT ステートメント自体はロールバック可能です。これにより、指定したステートメントは実行されなくなります。以下に例を示します。

```
START AFTER COMMIT UNIQUE INSERT INTO MyTable VALUES (1);
ROLLBACK;
```

この例では、トランザクションがロールバックされるため、「INSERT INTO MyTable VALUES (1)」は実行されません。

実行の順序

バックグラウンド・ステートメントは非同期で実行されるため、トランザクション内であってもその順序は保証されません。

例

ローカル・プロシージャーをバックグラウンドで開始します。

```
START AFTER COMMIT NONUNIQUE CALL myproc;
```

「CALL myproc」がまだバックグラウンドで実行されていない場合に、呼び出しを開始します。

```
START AFTER COMMIT UNIQUE call myproc;
```

プロパティー「color」が「blue」に設定されているレプリカを使用して、プロシージャーをバックグラウンドで開始します。

```
START AFTER COMMIT FOR EACH REPLICA WHERE color='blue' UNIQUE CALL myproc;
```

以下のステートメントはいずれも異なっていると見なされるため、キーワード UNIQUE の指定に関係なく各ステートメントが実行されます (「name」は各レプリカのユニーク・プロパティーです)。

```
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;
```

ただし、以下のステートメントが上記のステートメントと同じトランザクションで実行される場合に、条件「color='blue'」がレプリカ R1、R2、または R3 で一致しているときは、該当するレプリカに対して呼び出しがもう一度実行されることはありません。

```
START AFTER COMMIT FOR EACH REPLICA WHERE color='blue' UNIQUE call myproc;
```

その他の例については、27 ページの『3 章 ストアード・プロシージャー、イベント、トリガー、およびシーケンス』を参照してください。

TRUNCATE TABLE

```
TRUNCATE TABLE tablename
```

使用法

このステートメントは、呼び出し元の視点では、意味的に `DELETE FROM tablename` と同等です。ただし、分離が緩やかであるために効率ははるかに高くなります。

制限事項

このステートメントの実行中は、定義されている分離レベルが並行トランザクションで維持されません。行を削除すると、すべての並行トランザクションでその効果が直ちに反映されます。したがって、このステートメントは保守の目的でのみ使用することを推奨します。

切り捨てられた表を参照整合性制約に参照先の表として参加させると、参照元の表が空でない場合、定義された参照アクション (`RESTRICT`、`CASCADE` など) に関わらず、`TRUNCATE` ステートメントは失敗します。この制限事項は、参照元の表が参照先の表 (ツリー構造の表) と同じ場合は適用されません。

相互に関連する一連の表の切り捨てを行う場合、`TRUNCATE` ステートメントは、参照元としてのみ使用される表で始めて、参照のチェーンに従って各表を指定してから、参照先としてのみ使用される表で終わるような順序で発行する必要があります。

UNLOCK TABLE

```
UNLOCK TABLE { ALL | tablename [,tablename]}
tablename ::= アンロックする表の名前
```

キーワード ALL を指定すると、すべての表のすべての表レベルのロックが解放されます。

表名を修飾することで、表のカタログとスキーマを指定することもできます。

使用法

このコマンドでは、LOCK TABLE コマンドで LONG オプションを指定して手動でロックした表をアンロックすることができます。LONG オプションを指定すると、ロックが設定されたトランザクションが終了した後もロックが保持されます。ロックが自然に終了するポイントはトランザクション終了時以外にないため、LONG ロックは UNLOCK コマンドを使用して明示的に解放する必要があります。

UNLOCK TABLE コマンドは、サーバーの自動ロック、または LONG オプションでロックされていない手動ロックには適用されません。ロックが自動である場合や、手動であっても LONG でない場合は、ロックが設定されたトランザクションの終了時にサーバーが自動的にロックを解放します。したがって、これらのロックは手動でアンロックする必要がありません。

UNLOCK TABLE コマンドを使用してもすぐには作用しません。ロックは、現行トランザクションがコミットされたときに解放されます。

注意:

現行トランザクション (UNLOCK TABLE コマンドが実行されたトランザクション) がコミットされなかった場合 (ロールバックされた場合など)、表はアンロックされず、別の UNLOCK TABLE コマンドが正常に実行されてコミットされるまでロックされたままとなります。

LOCK/UNLOCK コマンドは表のみに適用されます。個々のレコードを手動でロックまたはアンロックするコマンドはありません。

「ALL」という名前の表がある場合は、区切り ID 機能を使用して表名を指定する必要があります (このセクションの最後に示す例を参照してください)。

LOCK および UNLOCK の使用例

```
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;

-- 現行トランザクション終了後も持続する排他ロックを取得します。
-- 排他ロックを直ちに取得できない場合は
-- 取得できるまで最大 60 秒間待機します。
LOCK TABLE emp, dept IN LONG EXCLUSIVE MODE WAIT 60;
-- スキーマに変更を加えます (または排他ロックを必要とする
-- 任意の操作を実行します)。
CALL DO_SCHEMA_CHANGES_1;
COMMIT WORK;
CALL DO_SCHEMA_CHANGES_2;
UNLOCK TABLE ALL; -- このトランザクションの終了時にロックを解放します。
...
COMMIT WORK;
...
UNLOCK TABLE "ALL";  -- 「ALL」という名前の表をアンロックします。
```

戻り値

各エラー・コードについて詳しくは、「*solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。

表 81. LOCK TABLE の戻り値

エラー・コード	説明
10083	表 <table_name> がロックされていません
13011	表 <tablename> が見つかりません

関連項目

LOCK TABLE

SET SYNC MODE { MAINTENANCE | NORMAL }

UNREGISTER EVENT

UNREGISTER EVENT コマンドは、ストアード・プロシージャー内でのみ使用できます。詳しくは、CREATE PROCEDURE ステートメントおよび CREATE EVENT ステートメントを参照してください。

UPDATE (位置付け)

```
UPDATE table_name
    SET [table_name.]column_identifier = {expression | NULL}
    [, [table_name.]column_identifier = {expression | NULL}]...
    WHERE CURRENT OF cursor_name
```

使用法

位置付け UPDATE ステートメントは、カーソルの現在行を更新します。カーソルの名前は、ODBC API 関数 SQLSetCursorName を使用して定義されます。

例

```
UPDATE TEST SET C = 0.33
WHERE CURRENT OF MYCURSOR
```

UPDATE (検索付き)

```
UPDATE table-name
    SET [table_name.]column_identifier = {expression | NULL}
    [, [table_name.]column_identifier = {expression | NULL}]...
    [WHERE search_condition]
```

使用法

UPDATE ステートメントは、検索条件に従って 1 つ以上の行で 1 つ以上の列の値を変更するために使用されます。

例

```
UPDATE TEST SET C = 0.44
WHERE ID = 5
```

WAIT EVENT

WAIT EVENT コマンドは、ストアード・プロシージャーの内部でのみ許可されます。詳しくは、CREATE PROCEDURE ステートメントおよび CREATE EVENT ステートメントを参照してください。

table_reference

表 82. *table_reference*

table_reference	
table_reference_list	::= *table_reference* [, *table-reference* ...]
table_reference	::= *table_name* [[AS] *correlation_name*] | *derived_table* [[AS] *correlation_name* [(*derived_column_list*)]] | *joined_table*
table_name	::= *table_identifier* | *schema_name.table_identifier*
derived_table	::= *subquery*
derived_column_list	::= *column_name_list*
joined_table	::= *cross_join* | *qualified_join* | *(joined_table)*
cross_join	::= *table_reference* CROSS JOIN *table_reference*
qualified_join	::= *table_reference* [NATURAL] [*join_type*] JOIN *table_reference* [*join_specification*]
join_type	::= INNER | *outer_join_type* [OUTER] | UNION
outer_join_type	::= LEFT | RIGHT | FULL
join_specification	::= *join_condition* | *named_columns_join*
join_condition	::= ON *search_condition*
named_columns_join	::= USING (*column_name_list*)
column_name_list	::= *column_identifier* [{ , *column_identifier* } ...]

query_specification

表 83. *query_specification*

query_specification	
query_specification	::= SELECT [DISTINCT | ALL] *select_list* *table_expression*

表 83. query_specification (続き)

query_specification	
select_list	::= * | *select_sublist* [{, *select_sublist* } ...]
select_sublist	::= *derived_column* | [*table_name* | *table_identifier*].*
derived_column	::= *expression* [[AS] *column_alias*]]
table_expression	::= 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*]

search_condition

表 84. search_condition

search_condition	
search_condition	::= *search_item* | *search_item* { AND | OR } *search_item*
search_item	::= [NOT] { *search_test* | (*search_condition*)}
search_test	::= *comparison_test* | *between_test* | *like_test* | *null_test* | *set_test* | *quantified_test* | *existence_test*
comparison_test	::= *expression* { = | <> | < | <= | > | >= } { *expression* | *subquery* } 注: 演算子の両側のスペースはオプションです。
between_test	::= *column_identifier* [NOT] BETWEEN *expression* AND *expression*
like_test	::= *column_identifier* [NOT] LIKE *value* [ESCAPE *value*]
null_test	::= *column_identifier* IS [NOT] NULL
set_test	::= *expression* [NOT] IN ({ *value* [, *value*]... | *subquery* })
quantified_test	::= *expression* { = | <> | < | <= | > | >= } [ALL | ANY | SOME] *subquery*

表 84. search_condition (続き)

search_condition	
existence_test	::= EXISTS *subquery*

Check_condition

表 85. Check_condition

Check_condition	
check_condition	::= *check_item* | check_item { AND | OR } *check_item*
check_item	::= [NOT] { *check_test* | (*check_condition*) }
check_test	::= *comparison_test* | *between_test* | *like_test* | *null_test* | *list_test*
comparison_test	::= *expression* { = | <> | < | <= | > | >= } { *expression* | *subquery* }
between_test	::= *column_identifier* [NOT] BETWEEN *expression* AND *expression*
like_test	::= *column_identifier* [NOT] LIKE *value* [ESCAPE *value*]
null_test	::= *column_identifier* IS [NOT] NULL
list_test	::= *expression* [NOT] IN ({ *value* [, *value*]...})

式

表 86. 式

式	
expression	::= *expression_item* | *expression_item* { + | - | * | / } *expression_item* 注: 演算子の両側のスペースはオプションです。
expression_item	::= [+ | -] { *value* | *column_identifier* | *function* | *case_expression* | *cast_expression* | (*expression*) }
value	::= *literal* | USER | *variable*

表 86. 式 (続き)

式	
function	::= *set_function* | *null_function* | *string_function* | *numeric_function* | *datetime_function* | *system_function* | *datatypeconversion_function* 注: string、numeric、datetime、datatypeconversion の各関数はスカラー関数です。この関数では、関数名で表される演算の後に 1 対の括弧で囲まれたゼロ個以上の引数が指定されます。各スカラー関数は、1 つの値を返します。
set_function	::= COUNT (*) | { AVG | MAX | MIN | SUM | COUNT } ({ ALL | DISTINCT } *expression*)
null_function	::= { NULLVAL_CHAR( ) | NULLVAL_INT( ) }
datatypeconversion_function	::= 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*) 注: 上記の関数は、ODBC で定義されている {fn CONVERT(*value, odbc_typename*)} エスケープ節を実装するために使用されます。ただし、推奨されているのは CAST(*value* AS *sql_typename*) を使用する方法です。これは SQL-92 で定義されており、solidDB で完全にサポートされています。詳しくは、「*solidDB プログラマー・ガイド*」の付録 F を参照してください。
case_expression	::= *case_abbreviation* | *case_specification*
case_abbreviation	::= NULLIF(*value_exp*, *value_exp*) | COALESCE(*value_exp* {, *value_exp* }...) NULLIF 関数は、1 番目のパラメーターが 2 番目のパラメーターと等しい場合に NULL を返し、そうでない場合は 1 番目のパラメーターを返します。この関数は、IF (p1 = p2) THEN RETURN NULL ELSE RETURN p1; と同等です。NULL を示すフラグとして機能する特殊値がある場合は、NULLIF 関数が便利です。NULLIF を使用することで、その特殊値を NULL に変換できます。つまり、IF (p1 = NullFlag) THEN RETURN NULL ELSE RETURN p1; と同じように動作します。 COALESCE は、NULL 値でない最初の引数を返します。引数のリストの長さにはほぼ制限がありません。すべての引数が同じ (または互換性のある) データ型であることが必要です。

表 86. 式 (続き)

式	
case_specification	::= CASE [*value_exp*] WHEN *value_exp* THEN {*value_exp* } [WHEN *value_exp* THEN { *value_exp* } ...] [ELSE { *value_exp* }] END
cast_expression	::= CAST (*value_exp* AS *-data-type*)
row value constructor expression	行値コンストラクター (RVC) は、順番に並べて括弧で区切られた一連の値です。以下に例を示します。 (1, 4, 9) ('Smith', 'Lisa') これは、表の行が一連のフィールドで構成されるのと同様に、一連の要素/値を基に行を構成する処理と見なすことができます。 行値コンストラクターについて詳しくは、 22 ページの『行値コンストラクター』を参照してください。

ストリング関数

表 87. ストリング関数

関数	目的
ASCII(*str*)	ストリング *str* と等価な整数を返します。
CHAR(*code*)	*code* と等価な文字を返します。
CONCAT(*str1*, *str2*)	str2 を str1 に連結します。
str1 { + | || } str2	*str2* を *str1* に連結します。 例: SELECT str1 + str2, col1 ... SELECT str1 || str2, col1 ...
GET_UNIQUE_STRING(*str*)	「接頭部」(任意の入力ストリング) とシーケンス番号 (内部で作成されて使用されるもの) に基づいて固有のストリングを生成します。入力が NULL の場合でも、固有のシーケンス番号に基づいてストリングを返します。
INSERT(*str1*, *start*, *length*, *str2*)	length の長さの文字を *str1* から削除し *str2* を挿入することで、ストリングをマージします。
LCASE(*str*)	ストリング *str* を小文字に変換します。

表 87. ストリング関数 (続き)

関数	目的
LEFT(*str*, *count*)	ストリング *str* の左端から count の個数の文字を返します。
LENGTH(*str*)	*str* の文字数を返します。
LOCATE(*str1*, *str2* [, *start*])	*str2* における *str1* の開始位置を返します。オプションの引数 *start* を指定すると、start の値で示される文字の位置から検索が開始されます。string_exp2 に string_exp1 が見つからない場合、この関数は 0 を返します。戻り値も入力パラメーター *start* も、ストリングの位置は 0 ではなく 1 から数えられます。
LTRIM(*str*)	*str* の先行スペースを除去します。
POSITION (*str1* IN *str2*)	*str2* における *str1* の開始位置を返します。
REPEAT(*str*, *count*)	*count* の回数だけ繰り返された *str* の文字を返します。
REPLACE(*str1*, *str2*, *str3*)	*str1* に出現する *str2* を *str3* で置換します。
RIGHT(*str*, *count*)	ストリング *str* の右端から count の個数の文字を返します。
RTRIM(*str*)	*str* の末尾のスペースを除去します。
SOUNDEX(*str*)	4 文字の soundex (音声) コードを計算します。
SPACE(*count*)	count の個数のスペースで構成されるストリングを返します。
SUBSTRING(*str*, *start*, *length*)	*str* から start を始点とする長さ *length* バイトのサブストリングを派生させます。例えば、str が「First Second Third」の場合に SUBSTRING(str, 7, 6) を実行すると、「Second」が返されます。 ストリングの位置は 0 ではなく 1 から数えられることに注意してください。
TRIM(*str*)	*str* から先行スペースと末尾のスペースを除去します。
UCASE(*str*)	*str* を大文字に変換します。

ストリング操作でワイルドカード文字を使用する場合は、 346 ページの『ワイルドカード文字』も参照してください。

数字関数

表 88. 数字関数

関数	目的
ABS(*numeric*)	*numeric* の絶対値
ACOS(*float*)	*float* のアークコサイン。ここで、*float* はラジアン単位で表します。

表 88. 数字関数 (続き)

関数	目的
ASIN(*float*)	*float* のアークサイン。ここで、*float* はラジアン単位で表します。
ATAN(*float*)	*float* のアークタンジェント。ここで、*float* はラジアン単位で表します。
ATAN2(*float1*, *float2*)	*float1* および *float2* によって、ラジアン単位の角度としてそれぞれ指定された、*x* と *y* の座標のアークタンジェント
CEILING(*numeric*)	*numeric* 以上で最小の整数
COS(*float*)	*float* のコサイン。ここで、*float* はラジアン単位で表します。
COT(*float*)	*float* のコタンジェント。ここで、*float* はラジアン単位で表します。
DEGREES(*numeric*)	*numeric* ラジアンを度に変換します。
DIFFERENCE(*str1*, *str2*)	音声の相違の値 (0 から 4) を返します。
EXP(*float*)	*float* の指数値
FLOOR(*numeric*)	*numeric* 以下で最大の整数
LOG(*float*)	*float* の自然対数
LOG10(*float*)	*float* の 10 を底とする対数
MOD(*integer1*, *integer2*)	*integer1* を *integer2* で除算したモジュラス
PI()	浮動小数点数のパイ
POWER(*numeric*, *integer*)	*numeric* を *integer* 乗した値
RADIANS(*numeric*)	*numeric* 度をラジアンに変換します。
ROUND(*numeric*, *integer*)	*numeric* を *integer* まで丸めた値
SIGN(*numeric*)	*numeric* の符号
SIN(*float*)	*float* のサイン。ここで、*float* はラジアン単位で表します。
SQRT(*float*)	*float* の平方根
TAN(*float*)	*float* のタンジェント。ここで、*float* はラジアン単位で表します。
TRUNCATE(*numeric*, *integer*)	*numeric* を *integer* まで切り捨てた値

日時関数

表 89. 日時関数

関数	目的
CURDATE()	現在日付を返します
CURTIME()	現在時刻を返します
DAYNAME(*date*)	曜日のストリングを返します
DAYOFMONTH(*date*)	1 から 31 の整数として、月の何日目かを返します
DAYOFWEEK(*date*)	1 から 7 の整数として、曜日を返します (1 が日曜日を表します)
DAYOFYEAR(*date*)	1 から 366 の整数として、年の何日目かを返します
EXTRACT (*date field* FROM *date_exp*)	日時またはインターバルの単一フィールドを分離して、数値に変換します
HOUR(*time_exp*)	0 から 23 の整数として、時間を返します
MINUTE(*time_exp*)	0 から 59 の整数として、分を返します
MONTH(*date*)	1 から 12 の整数として、月を返します
MONTHNAME(*date*)	月の名前をストリングとして返します
NOW()	現在の日時をタイム・スタンプとして返します
QUARTER(*date*)	1 から 4 の整数として、四半期を返します
SECOND(*time_exp*)	0 から 59 の整数として、秒を返します
TIMESTAMPADD(interval, *integer_exp, timestamp_exp*)	interval タイプの *integer_exp* インターバルを *timestamp_exp* に加算して、タイム・スタンプを計算します 有効な TIMESTAMPADD の interval 値を表すキーワードは、以下のとおりです 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

表 89. 日時関数 (続き)

関数	目的
TIMESTAMPDIFF(interval, *timestamp-exp1*, *timestamp-exp2*)	*timestamp-exp2* が *timestamp-exp1* よりもどのくらい大きいかを表すインターバルの整数を返します 有効な TIMESTAMPDIFF の interval 値を表すキーワードは、以下のとおりです 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(*date*)	1 から 52 の整数として、年の何週目かを返します
YEAR(*date*)	年を整数として返します

システム関数

システム関数は、solidDB データベースに関する特別な情報を返します。

表 90. システム関数

関数	目的
UIC()	接続に関連付けられている接続 ID を返します。
CURRENT_USERID ()	現在のユーザー ID を返します。
LOGIN_USERID ()	ログイン・ユーザー ID を返します。
CURRENT_CATALOG ()	現在のカタログを返します。
LOGIN_CATALOG ()	ログイン・カタログを返します。
CURRENT_SCHEMA ()	現在のスキーマを返します。
LOGIN_SCHEMA ()	ログイン・スキーマを返します。

各種関数

表 91. 各種関数

関数	目的
BIT_AND(*integer1*, *integer2*)	ビット単位の AND 演算の結果を返します。
DATABASE_ENCRYPTION_LEVEL()	データベースの暗号化レベルを返します。 • 0 - 暗号化されていない • 1 - 暗号化されているが、鍵は保護されていない (パスワードが空) • 2 - 暗号化されており、鍵が個別の開始パスワードによって保護されている • 3 - 暗号化されており、カスタムの暗号化方式が使用されている
IFNULL(*exp*, *value*)	*exp* がヌルの場合は *value* を返し、そうでない場合は *exp* を返します。 (*value* が返された場合、それは *exp* のタイプに変換されます。)
SLEEP(*milliseconds*)	これは、ストアード・プロシージャーまたはトリガーからのみ呼び出すことができます。これにより、ストアード・プロシージャーまたはトリガーは、指定されたミリ秒間、「スリープ」(一時的に活動を中断) 状態になります。解決の精度は、約 1 秒 (つまり、1000 ミリ秒) です。正確なスリープの長さは、コンピューターが他のプロセスまたはスレッドでどの程度ビジーであるかによっても異なります。値は変数や式でなく、リテラルでなければなりません。

data_type

表 92. data_type

変数名	データ型
data_type	::= {BIGINT | BINARY | BLOB | CHAR [*length*] | CHARACTER LARGE OBJECT | CHAR LARGE OBJECT | CLOB | DATE | DECIMAL [(*precision* [, *scale*])] | DOUBLE PRECISION | FLOAT [(*precision*)] | INTEGER | LONG NATIONAL VARCHAR | LONG VARBINARY | LONG VARCHAR | LONG WVARCHAR | NCHAR LARGE OBJECT | NUMERIC [(*precision* [, *scale*])] | NATIONAL CHAR | NATIONAL CHARACTER | NATIONAL VARCHAR | NCHAR | NCHAR VARYING | NCLOB | NVARCHAR | REAL | SMALLINT | TIME | TIMESTAMP [(*timestamp precision*)] | TINYINT | VARBINARY | VARCHAR [(*length*)] } | WCHAR | WVARCHAR [length]

日時リテラル

表 93. 日時リテラル

日時リテラル	
date_literal	’YYYY-MM-DD’
time_literal	’HH:MM:SS’
timestamp_literal	’YYYY-MM-DD HH:MM:SS’

疑似列

SELECT ステートメントの選択リストでは、以下の疑似列も使用できます。

表 94. 疑似列

疑似列	タイプ	説明
ROWVER	VARBINARY(10)	表の行のバージョン。
ROWID	VARBINARY(254)	表の行の永続 ID。
ROWNUM	DECIMAL(16,2)	表または結合された行のセットから選択された行のシーケンスを示す行番号。選択された最初の行の ROWNUM は 1 で、2 番目の行は 2 です。 ROWNUM は、order by 節が評価される前に行に与えられるため、ソートされた行の識別には使用できません。 ROWNUM は、主に照会から返される行の数の制限に役に立ちます。例えば、WHERE ROWNUM < 10) のようにします。

注:

ROWID と ROWVER は単一行を参照するため、単一表から行を返す照会でのみ使用できます。

ワイルドカード文字

以下の文字は、LIKE ’<string>’ などの特定の式の中で、ワイルドカード文字として使用できます。

表 95. ワイルドカード文字

文字	説明
_ (下線)	下線文字は、任意の 1 文字に一致します。例えば、’J_NE’ は ’JANE’ と ’JUNE’ に一致します。
% (パーセント記号)	% 記号は、0 個以上の文字からなる任意のグループに一致します。例えば、’ED%’ は ’EDWARD’ および ’EDITOR’ に一致します。別の例として、’%ED%’ は ’EDWARD’、’TEDDY’、および ’FRED’ に一致します。

SQL ワイルドカードの使用

完全一致突き合わせ検索は、以下のようにリテラルを指定することによって行います。

```
SELECT * FROM table1 WHERE name = 'SMITH';
```

ストリング ’SMITH’ はリテラル値です。

同様な突き合わせ検索は、別の文字ストリングによく似た文字ストリングを表す SQL ワイルドカードを指定することによって行います。論理式 (WHERE 節および

CHECK 制約で使用されるようなもの) で「ワイルドカード」とキーワード LIKE を使用して、類似のストリングを突き合わせることができます。

下線文字 (_) は、任意の 1 文字に一致するワイルドカード文字です。例えば、以下のような照会を実行したとします。

```
SELECT * FROM table1 WHERE first_name LIKE 'J_NE';
```

これは、JANE と JUNE の両方を (それ以外にも、最初の文字が J で最後の 2 文字が NE であるすべての 4 文字の名前を) 返します。

パーセント記号 (%) は、0 文字以上の任意のオカレンスに一致するワイルドカード文字です。例えば、以下のような照会を実行したとします。

```
SELECT * FROM table1 WHERE first_name LIKE 'JOHN%';
```

これは、JOHN、JOHNNY、JOHNATHAN などを返します。

% ワイルドカードは、ストリングの末尾に使用される場合が最も多いのですが、任意の場所に使用できます。例えば、以下の検索パターンを使用したとします。

```
LIKE '%JO%'
```

これは、名前のどこかに JO を含んでいるすべての人を返します。例えば、以下のような人ですが、これだけに限りません。

JOANNE、BILLY JO、および LONG JOHN SILVER

1 つのストリングの中で複数のワイルドカードを使用することもできます。例えば、ストリング J_V_ は JAVA と JIVE のほか、J で始まり 3 文字目が V であるすべての 4 文字のワードまたは名前に一致します。下線 (_) は正確に 1 文字にしか一致しないので、ストリング J_V_ は、4 文字を超える JOVIAL に一致しないことに注意してください。

リテラルとしてのワイルドカード文字

ワイルドカード文字をストリングの 1 つの部分で使用する一方、リテラル文字の % (パーセント) または下線 (_) を同じストリングの別の部分で使用できます。 ワイルドカード文字をリテラルとして使用するには、ワイルドカード文字の前にエスケープ文字を付けます。エスケープ文字自体は、照会の一部として指定する必要があります。例えば、以下の式は円記号 (¥) をエスケープ文字として使用しています。

```
LIKE 'MY¥_EXPRESSION_' ESCAPE '¥';
```

これは、以下に一致します。

```
MY_EXPRESSION1 MY_EXPRESSIONA MY_EXPRESSION_
```

しかし、以下には一致しません。

```
MY#EXPRESSION1
```

ANSI 規格の SQL は、文字ストリングを単一引用符で区切る必要があると指定しています。以下に例を示します。

```
...LIKE ' J_N_'; -- 正
...LIKE "J_N_"; --誤
```

二重引用符は、データでなく ID の区切りに使用されます。 C および Java のプログラマーは、これで混乱する場合があります。なぜなら、C 言語では二重引用符を *"C-language string"* のようにストリングを区切るために使用し、単一引用符を *'C'* のように単一の文字を区切るために使用するからです。

付録 C. 予約語

この付録では、いくつかの SQL 規格 (ODBC 3.0、X/Open および SQL Access Group SQL CAE 仕様、データベース言語 - SQL: ANSI X3H2 (SQL-92)) に含まれている予約語を示します。いくつかの単語は、solidDB SQL で使用されています。アプリケーションで、これらのキーワードを別の目的に使用することは避けてください。以下の表には、潜在的な予約語も含まれています。このマーキングは、括弧で囲んで示します。

この付録の一部の予約語は、二重引用符 ("") で囲むことで、ID (表名、列名など) として使用できます。二重引用符内の ID は、区切り ID とも呼ばれ、SQL の ANSI 規格に準拠しています。以下の SQL ステートメントの例では、予約語「NULL」が表名 ID として使用されています。

```
CREATE TABLE "NULL" (column_1 INTEGER);
```

注: solidDB SQL では、一部の予約語は、二重引用符で囲まなくても ID として使用できます。ただし、予約語を ID として使用する場合は、二重引用符で囲むことを強く推奨します。これによって、ポータビリティーが向上します。

表 96. 予約語リスト

予約語	ODBC	X/Open SQL	ANSI SQL-92	Solid SQL
ABSOLUTE	•		•	
ACTION	•		•	
ADA	•			
ADD	•	•	•	•
ADMIN				•
AFTER			(•)	•
ALIAS			(•)	
ALL	•	•	•	•
ALLOCATE	•	•	•	
ALTER	•	•	•	•
AND	•	•	•	•
ANY	•	•	•	•
APPEND				•
ARE	•		•	

表 96. 予約語リスト (続き)

予約語	ODBC	X/Open SQL	ANSI SQL-92	Solid SQL
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	•	•	•	•

表 96. 予約語リスト (続き)

予約語	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	•	•	•	•

表 96. 予約語リスト (続き)

予約語	ODBC	X/Open SQL	ANSI SQL-92	Solid SQL
CYCLE			(•)	
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			(•)	•

表 96. 予約語リスト (続き)

予約語	ODBC	X/Open SQL	ANSI SQL-92	Solid SQL
ENABLE				•
END	•	•	•	•
END-EXEC	•		•	
EQUALS			(•)	
ESCAPE	•		•	•
EVENT				•
EXCEPT	•		•	•
EXCEPTION				
EXEC	•	•	•	•
EXECUTE	•	•	•	•
EXISTS	•	•	•	•
EXPLAIN				•
EXPORT				•
EXTERNAL	•		•	•
EXTRACT	•		•	•
FALSE	•		•	
FETCH	•	•	•	•
FIRST	•		•	
FIXED				•
FLOAT	•	•	•	•
FOR	•	•	•	•
FOREIGN	•	•	•	•
FOREVER				•
FORTRAN	•			
FORWARD				•
FOUND	•	•	•	
FROM	•	•	•	•

表 96. 予約語リスト (続き)

予約語	ODBC	X/Open SQL	ANSI SQL-92	Solid SQL
FROMFIXED				•
FULL	•		•	•
GENERAL			(•)	
GET	•	•	•	•
GLOBAL	•		•	
GO	•		•	
GOTO	•	•	•	
GRANT	•	•	•	•
GROUP	•	•	•	•
HAVING	•	•	•	•
HINT				•
HOUR	•		•	
IDENTIFIED				•
IDENTITY	•		•	
IF			(•)	•
IGNORE	•		(•)	
IMMEDIATE	•	•	•	
IMPORT				•
IN	•	•	•	•
INCLUDE	•	•		
INDEX	•	•		•
INDICATOR	•		•	
INITIALLY	•		•	
INNER	•		•	•
INPUT	•		•	
INSENSITIVE	•		•	
INSERT	•	•	•	•

表 96. 予約語リスト (続き)

予約語	ODBC	X/Open SQL	ANSI SQL-92	Solid SQL
INT	•	•	•	•
INTEGER	•	•	•	•
INTERNAL				•
INTERSECT	•		•	•
INTERVAL	•		•	
INTO	•	•	•	•
IS	•	•	•	•
ISOLATION	•		•	•
JAVA				•
JOIN	•		•	•
KEY	•	•	•	•
LANGUAGE	•		•	
LAST	•		•	
LEADING	•		•	
LEAVE			(•)	•
LEFT	•		•	•
LESS			(•)	
LEVEL	•		•	•
LIKE	•	•	•	•
LIMIT			(•)	
LOCAL	•		•	•
LOCK				•
LONG				•
LOOP			(•)	•
LOWER	•		•	
MAINMEMORY				•
MASTER				•

表 96. 予約語リスト (続き)

予約語	ODBC	X/Open SQL	ANSI SQL-92	Solid SQL
MATCH	•		•	
MAX	•	•	•	
MERGE				•
MESSAGE				•
MIN	•	•	•	
MINUTE	•		•	
MODIFY			(•)	•
MODULE	•		•	
MONTH	•		•	
NAMES	•		•	
NATIONAL	•		•	
NATURAL	•		•	•
NCHAR	•		•	
NEW			(•)	•
NEXT	•		•	•
NO	•		•	•
NONE	•		(•)	
NOT	•	•	•	•
NULL	•	•	•	•
NULLIF	•		•	•
NUMERIC	•	•	•	•
OBJECT			(•)	
OCTET_LENGTH	•		•	
OF	•	•	•	•
OFF				
OID			(•)	
OLD			(•)	•

表 96. 予約語リスト (続き)

予約語	ODBC	X/Open SQL	ANSI SQL-92	Solid SQL
ON	•	•	•	•
ONLY	•		•	•
OPEN	•	•	•	
OPERATION			(•)	
OPERATORS			(•)	
OPTIMISTIC				•
OPTION				
OR				•
ORDER				
OTHERS				
OUTER				•
OUTPUT	•		•	
OVERLAPS	•		•	
PARAMETERS			(•)	
PARTIAL	•		•	
PASCAL	•			
PENDANT			(•)	
PESSIMISTIC				•
PLAN				•
PLI	•			
POSITION	•		•	
POST				•
PRECISION	•	•	•	•
PREORDER			(•)	
PREPARE				
PRESERVE				
PRIMARY	•	•	•	•

表 96. 予約語リスト (続き)

予約語	ODBC	X/Open SQL	ANSI SQL-92	Solid SQL
PRIOR	•		•	
PRIVATE			(•)	
PRIVILEGES	•		•	•
PROCEDURE	•		•	•
PROPAGATE				•
PROTECTED			(•)	
PUBLIC	•	•	•	•
PUBLICATION				•
READ			•	•
REAL		•	•	•
RECURSIVE			(•)	
REF			(•)	
REFERENCES	•	•	•	•
REFERENCING			(•)	•
REFRESH				•
REGISTER				•
RELATIVE	•		•	
RENAME				•
REPEATABLE				•
REPLACE			(•)	
REPLICA				•
REPLY				•
RESIGNAL			(•)	
RESTART				•
RESTRICT	•	•	•	•
RESULT				•
RETURN			(•)	•

表 96. 予約語リスト (続き)

予約語	ODBC	X/Open SQL	ANSI SQL-92	Solid SQL
RETURNS			(•)	•
REVERSE				•
REVOKE	•	•	•	•
RIGHT	•		•	•
ROLE			(•)	•
ROLLBACK	•	•	•	•
ROUTINE			(•)	
ROW			(•)	
ROWID				•
ROWNUM				•
ROWSPERMESSAGE				•
ROWVER				•
ROWS	•		•	
SAVEPOINT			(•)	•
SCAN				•
SCHEMA	•		•	•
SCROLL	•		•	
SEARCH			(•)	
SECOND	•		•	
SECTION	•	•	•	
SELECT	•	•	•	•
SENSITIVE			(•)	
SEQUENCE			(•)	•
SERIALIZABLE				•
SESSION	•		•	
SESSION_USER	•		•	
SET	•	•	•	•

表 96. 予約語リスト (続き)

予約語	ODBC	X/Open SQL	ANSI SQL-92	Solid SQL
SIGNAL			(•)	
SIMILAR			(•)	
SIZE	•		•	
SMALLINT	•	•	•	•
SOME	•		•	•
SORT				•
SPACE	•			
SQL	•	•	•	•
SQLCA	•	•		
SQLCODE	•		•	
SQLERROR	•	•	•	•
SQLEXCEPTION			(•)	
SQLSTATE				
SQLWARNING	•		(•)	
START				•
STRUCTURE			(•)	
SUBSCRIBE				•
SUBSCRIPTION				•
SUBSTRING	•		•	
SUM	•	•	•	
SYNC_CONFIG				•
SYSTEM	•			
SYSTEM_USER			•	
TABLE	•	•	•	•
TEMPORARY	•		•	
TEST			(•)	
THEN	•		•	•

表 96. 予約語リスト (続き)

予約語	ODBC	X/Open SQL	ANSI SQL-92	Solid SQL
THERE			(•)	
TIME	•		•	•
TIMEOUT				•
TIMESTAMP	•		•	•
TIMEZONE_HOUR	•		•	
TIMEZONE_MINUTE	•		•	
TINYINT				•
TO	•	•	•	•
TRAILING			•	
TRANSACTION	•		•	•
TRANSACTIONS				•
TRANSLATE	•		•	
TRANSLATION	•		•	
TRIGGER			(•)	•
TRIM	•		•	
TRUE	•		•	
TRUNCATE				•
TYPE			(•)	
UNDER			(•)	
UNION	•	•	•	•
UNIQUE	•	•	•	•
UNKNOWN	•		•	
UNREGISTER				•
UPDATE	•	•	•	•
UPPER	•		•	
USAGE	•		•	
USER	•	•	•	•

表 96. 予約語リスト (続き)

予約語	ODBC	X/Open SQL	ANSI SQL-92	Solid SQL
USING	•	•	•	•
VALUE	•	•	•	•
VALUES	•	•	•	•
VARBINARY				•
VARCHAR	•	•	•	•
VARIABLE			(•)	
VARWCHAR				•
VARYING	•	•	•	
VIEW	•	•	•	•
VIRTUAL			(•)	
VISIBLE			(•)	
WAIT			(•)	•
WCHAR				•
WHEN	•		•	•
WHENEVER	•	•	•	
WHERE	•	•	•	•
WHILE			(•)	•
WITH	•	•	•	•
WITHOUT			(•)	
WORK	•	•	•	•
WRITE			•	•
WVARCHAR				•
YEAR	•		•	
ZONE			•	

注:

CASCADED: 単語 CASCADED は、solidDB で予約されていますが、現在、solidDB SQL ステートメントでは使用されていません。

付録 D. データベース・システム表とシステム・ビュー

システム表

SQL_LANGUAGES

SQL_LANGUAGES システム表には、サポートされている SQL 規格および SQL ダイアレクトがリストされています。

表 97. SQL_LANGUAGES

列名	データ型	説明
SOURCE	WVARCHAR	この特定の SQL バージョンを定義した組織。
SOURCE_YEAR	WVARCHAR	関連する規格が承認された年。
CONFORMANCE	WVARCHAR	関連する規格への適合レベル。
INTEGRITY	WVARCHAR	Integrity Enhancement Feature がサポートされているかどうか。
IMPLEMENTATION	WVARCHAR	ベンダーの SQL 言語を一意的に識別します (SOURCE が「ISO」の場合は NULL)。
BINDING_STYLE	WVARCHAR	バインディング・スタイル「DIRECT」、「EMBED」、または「MODULE」。
PROGRAMMING_LANG	WVARCHAR	使用されているホスト言語。

SYS_ATTAUTH

表 98. SYS_ATTAUTH

列名	データ型	説明
REL_ID	INTEGER	表 ID。
UR_ID	INTEGER	ユーザーまたはロールの ID。
ATTR_ID	INTEGER	列 ID。
PRIV	INTEGER	特権情報。
GRANT_ID	INTEGER	付与者 ID。
GRANT_TIM	TIMESTAMP	付与時刻。

SYS_AUDIT_TRAIL

SYS_AUDIT_TRAIL システム表には、監査情報が含まれています。監査を有効にする (**Sql.AuditTrailEnabled=yes**) と、データベース・アクティビティーに関する情報が SYS_AUDIT_TRAIL システム表に書き込まれます。管理者権限を持つユーザーは、通常の SQL 構文で SYS_AUDIT_TRAIL システム表を照会できます。

表 99. SYS_AUDIT_TRAIL 表の定義

列名	データ型	説明
CREATIME	TIMESTAMP	監査対象ステートメントがコミットされたタイム・スタンプ。
LOGIN_USER	WVARCHAR	監査対象データを実行およびコミットするためにシステムにログインしたユーザー。
MACHINE_ID	WVARCHAR	監査対象ステートメントが実行された場所を示します。MACHINE_ID が設定されていない場合、値は NULL です。
APP_INFO	WVARCHAR	どのアプリケーションがステートメントを実行したかを示します。APP_INFO が設定されていない場合、値は NULL です。
CURRENT_CATALOG	WVARCHAR	トランザクションがコミットされたときに、どのカタログが使用されていたかを示します。
CURRENT_SCHEMA	WVARCHAR	トランザクションがコミットされたときに、どのスキーマが使用されていたかを示します。
TYPE	VARCHAR	監査対象となる変更が、どのような種類の操作であったかを示します。有効な値を以下に示します。
SQLSTR	LONG WVARCHAR	監査された実行済み SQL ストリングを示します。

TYPE の有効な値は以下のとおりです。

```
STATUS
ALTER TABLE
ALTER USER (SQL ステートメント内のパスワードは記録されない)
CREATE CATALOG
CREATE DOMAIN
CREATE EVENT
CREATE INDEX
CREATE PUBLICATION
CREATE ROLE
CREATE SCHEMA
CREATE SEQUENCE
CREATE TABLE
CREATE USER (SQL ステートメント内のパスワードは記録されない)
CREATE VIEW
DELETE (DELETE ステートメントが SYS_AUDIT_TRAIL システム表に影響を与えた場合に記録される)
DROP CATALOG
DROP DOMAIN
DROP EVENT
DROP INDEX
DROP PUBLICATION
DROP ROLE
DROP SCHEMA
DROP SEQUENCE
DROP TABLE
DROP USER
DROP VIEW
GRANT
GRANT ROLE
REVOKE
REVOKE ROLE
CREATE PROCEDURE
```

```
DROP PROCEDURE
CREATE TRIGGER
ALTER TRIGGER
DROP TRIGGER
```

SYS_BACKGROUNDJOB_INFO

START AFTER COMMIT ステートメントの本体を開始できない場合は、その理由がシステム表 SYS_BACKGROUNDJOB_INFO にログとして記録されます。この表には、失敗した START AFTER COMMIT ステートメントのみがログとして記録されます。ステートメント (プロシージャー呼び出しなど) が正常に開始された場合、このシステム表には情報が格納されません。正常に開始して実行が完了しなかったステートメントも、このシステム表には格納されません。

ユーザーは、SQL SELECT ステートメントを使用するか、システム・プロシージャー SYS_GETBACKGROUNDJOB_INFO を呼び出すことで、表 SYS_BACKGROUNDJOB_INFO から情報をリトリーブできます。詳しくは、『SYS_BACKGROUNDJOB_INFO』を参照してください。

また、START AFTER COMMIT ステートメントの開始に失敗すると、システム定義イベント SYS_EVENT_SACFAILED が通知されます。このイベントについて詳しくは、424 ページの『各種イベント』を参照してください。アプリケーションはこのイベントを待ち、ジョブ ID を使用してシステム表 SYS_BACKGROUNDJOB_INFO からエラー・メッセージをリトリーブできます。

システム表 SYS_BACKGROUNDJOB_INFO を空にするには、ADMIN COMMAND を使用します。

ADMIN COMMAND ’cleanbgjobinfo’;

’cleanbgjobinfo’ コマンドを実行できるのは DBA だけです。

表 100. SYS_BACKGROUNDJOB_INFO

列名	データ型	説明
ID	INTEGER	ジョブ ID。
STMT	WVARCHAR	実行できなかったステートメント。
USER_ID	INTEGER	ユーザーまたはロールの ID。
ERROR_CODE	INTEGER	ステートメントを実行しようとしたときに発生したエラー。
ERROR_TEXT	WVARCHAR	エラーの説明。

SYS_BLOBS

この表には、データベースに格納される BLOB に関する情報が取り込まれます。さらに、この表によって、論理的には複数回保存される BLOB でも、物理的にはディスクに 1 回だけ保存されるようになります。

表 101. SYS_BLOBS

列名	データ型	説明
ID	BIGINT	BLOB ID。
STARTPOS	BIGINT	BLOB の先頭からのバイト・オフセット。つまりページの開始位置。
ENDSIZE	BIGINT	最後のページの末尾のバイト・オフセット + 1。
TOTALSIZE	BIGINT	BLOB の合計サイズ。
REFCOUNT	INTEGER	リファレンスの数。つまり、同じ BLOB の既存のインスタンスの数。
COMPLETE	INTEGER	BLOB への書き込みが可能であるかどうか。
STARTCPNUM	INTEGER	BLOB の書き込みが開始されたチェックポイント・レベル。
NUMPAGES	INTEGER	BLOB を構成するページの数。
P01_ADDR	INTEGER	BLOB の先頭からの最初のページのバイト・オフセット。
P01_ENDSIZE	BIGINT	最初のページの最後のバイト + 1。
P[02...50]_ADDR	INTEGER	BLOB の先頭からのページ [2...50] のバイト・オフセット。
P[02...50]_ENDSIZE	BIGINT	ページ [2...50] の最後のバイト + 1。

SYS_CARDINAL

この表のデータは、チェックポイントごとにリフレッシュされ、それ以外のタイミングではリフレッシュされません。

表 102. SYS_CARDINAL

列名	データ型	説明
REL_ID	INTEGER	SYS_TABLES にあるリレーション ID。
CARDIN	INTEGER	表内の行数。
SIZE	INTEGER	表内のデータのサイズ。
LAST_UPD	TIMESTAMP	表での最後の更新のタイム・スタンプ。

SYS_CATALOGS

SYS_CATALOGS には、使用可能なカタログがリストされています。

表 103. SYS_CATALOGS

列名	データ型	説明
ID	INTEGER	カタログ ID。
NAME	WVARCHAR	カタログ名。
CREATIME	TIMESTAMP	作成日時。
CREATOR	WVARCHAR	作成者名。

SYS_CHECKSTRINGS

SYS_CHECKSTRINGS には、表の CHECK 制約がリストされています。

表 104. SYS_CHECKSTRINGS

列名	データ型	説明
ID	INTEGER	SYS_TABLES を参照する表 ID。
CONSTRAINT_NAME	WVARCHAR	CHECK 制約の名前 (表ごとにユニーク) または名前のない制約を表す空ストリング (名前のないすべての CHECK 制約に 1 つのストリング。これらの制約は AND で連結されます)。
CONSTRAINT	WVARCHAR	制約ストリング自体。指定の表に挿入/更新を実行する際に SQL インタープリターによって検査されます。

SYS_COLUMNS

この表には、システム表のすべての列がリストされています。

システム列は、所有者やユーザーによる表示に制限がありません。つまり、所有者はこの表にある、自身が作成した列以外の列を表示できます。また、アクセス権限のないユーザーまたは特定のアクセス権限を持つユーザーも、この表のすべてのシステム列を表示できます。

表 105. SYS_COLUMNS

列名	データ型	説明
ID	INTEGER	ユニークな列 ID。
REL_ID	INTEGER	SYS_TABLES にあるリレーション ID。
COLUMN_NAME	WVARCHAR	列の名前。
COLUMN_NUMBER	INTEGER	表での列の番号 (作成順)。
DATA_TYPE	WVARCHAR	列のデータ型。

表 105. SYS_COLUMNS (続き)

列名	データ型	説明
SQL_DATA_TYPE_NUM	SMALLINT	ODBC 準拠のデータ型番号。
DATA_TYPE_NUMBER	INTEGER	内部データ型番号。
CHAR_MAX_LENGTH	INTEGER	CHAR フィールドの最大長。
NUMERIC_PRECISION	INTEGER	数値の精度。
NUMERIC_PREC_RADIX	SMALLINT	数値の精度の基数。
NUMERIC_SCALE	SMALLINT	数値の位取り。
NULLABLE	CHAR	NULL 値が許可されるかどうか (Yes、No)。
NULLABLE_ODBC	SMALLINT	(ODBC) NULL 値が許可されるかどうか (1、0)。
FORMAT	WVARCHAR	将来の使用のために予約済み。
DEFAULT_VAL	WVARCHAR	現在のデフォルト値 (設定されている場合)。
ATTR_TYPE	INTEGER	ユーザー定義 (0) または内部 (>0)。
REMARKS	LONG WVARCHAR	将来の使用のために予約済み。

SYS_COLUMNS_AUX

行が存在する表にデフォルト値の列を挿入した場合、既存の行には列のデフォルト値が追加されません。代わりに、列を挿入するステートメントで定義されたデフォルト値が SYS_COLUMNS_AUX 表に書き込まれます。列の前に表に挿入されていた行が SQL 照会の対象である場合は、その行の新しい列値が挿入後に変更されていない限り、列値は SYS_COLUMNS_AUX 表から読み取られます。SYS_COLUMNS_AUX 表には元のデフォルト値のみが保存されます。

表 106. SYS_COLUMNS_AUX

列名	データ型	説明
ID	INTEGER	表 ID。
ORIGINAL_DEFAULT	WVARCHAR	元のデフォルト値。

SYS_DL_REPLICA_CONFIG

この表には、マスターでのディスクレス構成が格納されています。この表は、soldlsetup コマンドによる更新のみに使用されます。ユーザーはこの表を直接変更しないようにする必要があります。直接変更すると、悪影響が生じるおそれがあります。

表 107. SYS_DL_REPLICA_CONFIG

列名	データ型	説明
CFG_NAME	WVARCHAR (254) PRIMARY KEY NOT NULL	ディスクレス・レプリカ構成の名前。
INI_FILE	LONG WVARCHAR	レプリカ構成ファイルの名前。この列には、solid.ini ファイルの内容が BLOB として挿入されます。
LIC_FILE	LONG WVARCHAR	レプリカ・ライセンス・ファイルの名前。この列には、solid.lic ファイルの内容が BLOB として挿入されます。
SCHEMA_FILE	LONG WVARCHAR	レプリカ・スキーマの名前。この列には、スキーマ・ファイルの内容が BLOB として挿入されます。

SYS_DL_REPLICA_DEFAULT

この表には、マスターでのディスクレスのデフォルト構成が格納されています。この表は、soldlsetup コマンドによる更新のみに使用されます。ユーザーはこの表を直接変更しないようにする必要があります。直接変更すると、悪影響が生じるおそれがあります。

表 108. SYS_DL_REPLICA_DEFAULT

列名	データ型	説明
REPLICA_NAME	VARCHAR(254) NOT NULL PRIMARY KEY	レプリカの名前。
INI_CFG	VARCHAR(254) REFERENCE SYS_DL_REPLICA_CONFIG(CFG_NAME)	レプリカ構成ファイルの名前。
LIC_CFG	VARCHAR(254) REFERENCE SYS_DL_REPLICA_CONFIG(CFG_NAME)	レプリカ・ライセンス・ファイルの名前。
SCHEMA_CFG	VARCHAR(254) REFERENCE SYS_DL_REPLICA_CONFIG(CFG_NAME)	レプリカ・スキーマの名前。

SYS_EVENTS

表 109. SYS_EVENTS

列名	データ型	説明
ID	INTEGER	ユニークなイベント ID。
EVENT_NAME	WVARCHAR	イベントの名前。

表 109. SYS_EVENTS (続き)

列名	データ型	説明
EVENT_PARAMCOUNT	INTEGER	パラメーターの数。
EVENT_PARAMTYPES	LONG VARBINARY	パラメーターのタイプ。
EVENT_TEXT	WVARCHAR	イベントの本体。
EVENT_SCHEMA	WVARCHAR	イベントの所有者。
EVENT_CATALOG	WVARCHAR	イベントの所有者。
CREATIME	TIMESTAMP	作成時刻。
TYPE	INTEGER	将来の使用のために予約済み。

SYS_FEDT_DB_PARTITION

SYS_FEDT_DB_PARTITION 表に、使用可能なトランザクション・ログ・リーダー・パーティションが記述されています。

表 110. SYS_FEDT_DB_PARTITION 表の定義

列名	データ型	説明
PARTITION_ID	CHAR(30)	この列には、パーティション名が含まれています。
DESCRIPTION	VARCHAR(255)	使用しない
LOADACTIVE	INTEGER	使用しない

SYS_FEDT_TABLE_PARTITION

SYS_FEDT_TABLE_PARTITION 表では、表とログ・リーダー・パーティションの間のマッピングを提供します。

表 111. SYS_FEDT_TABLE_PARTITION 表の定義

列名	データ型	説明
TABLE_CATALOG	VARCHAR(255)	この列では、カタログ名を示します。 それぞれの表に、関連付けられたカタログ名があります。カタログは、データベースが作成されたときに作成されるか、後から CREATE CATALOG ステートメントを使用して作成されます。カタログ名にはファクトリー値はありません。

表 111. SYS_FEDT_TABLE_PARTITION 表の定義 (続き)

列名	データ型	説明
TABLE_SCHEMA	VARCHAR(255)	この列では、スキーマ名を示します。 それぞれの表に、関連付けられたスキーマ名があります。デフォルトでは、スキーマ名はユーザー ID (ユーザー名) です。スキーマは、CREATE SCHEMA ステートメントを使用して作成することもできます。
PARTITION_ID	CHAR(30)	この列では、パーティション名を示します。 この列は、FEDT_TABLE_PARTITION 表への外部キーです。
BE_TABLE_DATABASE	VARCHAR(255)	使用しない
BE_TABLE_SCHEMA	VARCHAR(255)	使用しない
BE_TABLE_NAME	CHAR(30)	使用しない
RANGE_COL_NAME	CHAR(30)	使用しない
LOWER_LIMIT	VARCHAR(255)	使用しない
UPPER_LIMIT	VARCHAR(255)	使用しない
SELECT_LIST	VARCHAR(255)	使用しない
LOADORDER	INTEGER	使用しない

SYS_FORKEYPARTS

表 112. SYS_FORKEYPARTS

列名	データ型	説明
KEY_CATALOG	INTEGER	キーの作成者名または所有者。
ID	INTEGER	外部キー ID。
KEYP_NO	INTEGER	キー・パート番号。
ATTR_NO	INTEGER	列番号。
ATTR_ID	INTEGER	列 ID。
ATTR_TYPE	INTEGER	列タイプ。
CONST_VALUE	VARBINARY	内部定数値。ない場合は NULL。

SYS_FORKEYS

表 113. SYS_FORKEYS

列名	データ型	説明
ID	INTEGER	外部キー ID。
REF_REL_ID	INTEGER	参照先の表 ID。
CREATE_REL_ID	INTEGER	作成者の表 ID。
REF_KEY_ID	INTEGER	参照されるキー ID。
REF_TYPE	INTEGER	参照タイプ。
KEY_SCHEMA	WVARCHAR	作成者の名前。
KEY_CATALOG	WVARCHAR	キーの作成者名または所有者。
KEY_NREF	INTEGER	参照されるキー・パートの数。

SYS_HOTSTANDBY

この表の使用は推奨されません。4.0 より前のバージョンに関係する表です。

SYS_INFO

表 114. SYS_INFO

列名	データ型	説明
PROPERTY	WVARCHAR	プロパティーの名前。
VALUE_STR	WVARCHAR	ストリングとしての値。
VALUE_INT	INTEGER	整数としての値。

SYS_KEYPARTS

表 115. SYS_KEYPARTS

列名	データ型	説明
ID	INTEGER	この列は sys_keys.id への外部キー参照です。これにより、各キー・パートがどのキーの一部であるかを特定できます。
REL_ID	INTEGER	SYS_TABLES にあるリレーション ID。
KEYP_NO	INTEGER	キー・パート ID。
ATTR_ID	INTEGER	列 ID。
ATTR_NO	INTEGER	表での列の番号 (作成順)。

表 115. SYS_KEYPARTS (続き)

列名	データ型	説明
ATTR_TYPE	INTEGER	列のタイプ。
CONST_VALUE	VARBINARY	定数値または NULL。
ASCENDING	CHAR	キーが昇順 (Yes) か降順 (No) か。

SYS_KEYS

すべてのデータベース表には 1 つのクラスタリング・キーが必要です。このキーは、データの物理的なソート順を定義します。このキーは容量には影響しません。主キーが定義されている場合は、主キーがクラスタリング・キーとして使用されます。主キーが定義されていない場合は、SYS_KEYS に key_name が「$CLUSTKEY_xxxxx」のエントリーが自動的に作成されます。

表に主キーの定義がある場合は、key_name が「$PRIMARYKEY_xxxx」のエントリーが SYS_KEYS に作成されます。key_primary 列と key_clustering 列の値は YES になります。

表に主キーの定義がない場合は、key_name が「$CLUSTKEY_xxxxx」のエントリーが SYS_KEYS に作成されます。key_primary 列の値は NO、key_clustering 列の値は YES になります。

表 116. SYS_KEYS

列名	データ型	説明
ID	INTEGER	ユニーク・キー ID。
REL_ID	INTEGER	SYS_TABLES にあるリレーション ID。
KEY_NAME	WVARCHAR	キーの名前。
KEY_UNIQUE	CHAR	キーがユニークかどうか (Yes、No)。
KEY_NONUNIQUE_ODBC	SMALLINT	(ODBC) キーがユニークでないかどうか (1、0)。
KEY_CLUSTERING	CHAR	キーがクラスタリング・キーかどうか (Yes、No)。
KEY_PRIMARY	CHAR	キーが主キーかどうか (Yes、No)。
KEY_PREJOINED	CHAR	将来の使用のために予約済み。
KEY_SCHEMA	WVARCHAR	キーの所有者。

表 116. SYS_KEYS (続き)

列名	データ型	説明
KEY_NREF	INTEGER	サーバーは、主キーを作成する際に、ユーザーが N 個のフィールドを指定した場合でも、表のすべてのフィールドを使用します (ユーザーが指定した N 個のフィールドは、キーにおける最初の N 個のフィールドになります)。KEY_NREF の値は N、つまりユーザーが指定したフィールドの数です。

SYS_LOGPOS

SYS_LOGPOS 表には、トランザクション・ログ・リーダーがキャッチアップのために必要とする、ログ間隔に関する情報が含まれています。

各行は、**LogReader.SaveLogposInterval** パラメーターによって定義されたおおよその細分度で記録されるログ間隔を表します。現在の間隔のサイズが **SaveLogposInterval** の値に達すると、新規行が追加されます。ログ間隔サイズの合計は、**LogReader.MaxLogSize** パラメーターの値とほぼ同じです。最後に記録されたログ間隔の先頭から、示されているログ位置までの距離が **LogReader.MaxLogSize** よりも大きくなると、ログ間隔の行は自動的に削除されます。

SYS_LOGPOS 表は、SYS_LOG 表照会の実行中に、ログ・リーダーによって内部のみで使用されます。この表には、ログ・リーダーを使用するアプリケーションの設計時に使用される情報は含まれていません。

照会で、SYS_LOGPOS が表すログ間隔を超えるログ・レコードが要求された場合、照会は「ログ・オーバーフロー」というエラーを返します。

表 117. SYS_LOGPOS 表の定義

列名	データ型	説明
ID	INTEGER	論理ログ間隔 ID (ログ間隔の ID)
LOG_POS	VARCHAR(255)	ログ間隔の先頭の物理的位置
SIZE	INTEGER	直前のログ間隔の先頭位置からのバイト単位の距離 (**LogReader.SaveLogposInterval** パラメーターで定義した値に近似)

SYS_PROCEDURES

このシステム表には、プロシージャーがリストされています。

特定のユーザーがプロシージャーを表示できないように制限があります。所有者は自身の作成したプロシージャーの表示に制限されます。ユーザーは、プロシージャー定義を参照するための実行権限を持っているプロシージャーの表示のみができます。アクセス権限がないユーザーは、すべてのプロシージャーが表示できないように制限されます。実行権限があるユーザーでもプロシージャー定義を表示できないことに注意してください。DBA には制限が適用されません。

表 118. SYS_PROCEDURES

列名	データ型	説明
ID	INTEGER	ユニークなプロシージャー ID。
PROCEDURE_NAME	WVARCHAR	プロシージャー名。
PROCEDURE_TEXT	LONG WVARCHAR	プロシージャー本体。
PROCEDURE_BIN	LONG VARBINARY	コンパイルされた形式のプロシージャー。
PROCEDURE_SCHEMA	WVARCHAR	PROCEDURE_NAME を含んでいるスキーマの名前。
PROCEDURE_CATALOG	WVARCHAR	PROCEDURE_NAME を含んでいるカタログの名前。
CREATIME	TIMESTAMP	作成時刻。
TYPE	INTEGER	将来の使用のために予約済み。

SYS_PROCEDURE_COLUMNS

SYS_PROCEDURE_COLUMNS は、入力パラメーターと結果セットの列を定義します。

表 119. SYS_PROCEDURE_COLUMNS

列名	データ型	説明
PROCEDURE_ID	INTEGER	プロシージャー ID。
COLUMN_NAME	WVARCHAR	プロシージャー列の名前。
COLUMN_TYPE	SMALLINT	プロシージャー列のタイプ (SQL_PARAM_INPUT または SQL_RESULT_COL)。
DATA_TYPE	SMALLINT	列の SQL データ型。
TYPE_NAME	WVARCHAR	列の SQL データ型名。
COLUMN_SIZE	INTEGER	プロシージャー列のサイズ。
BUFFER_LENGTH	INTEGER	列サイズ (バイト単位)。
DECIMAL_DIGITS	SMALLINT	プロシージャー列の小数桁数。
NUM_PREC_RADIX	SMALLINT	数値データ型の基数 (2、10、または該当しない場合は NULL)。
NULLABLE	SMALLINT	プロシージャー列で NULL 値が許可されるかどうか。
REMARKS	WVARCHAR	プロシージャー列の説明。

表 119. SYS_PROCEDURE_COLUMNS (続き)

列名	データ型	説明
COLUMN_DEF	WVARCHAR	列のデフォルト値。常に NULL です。つまりデフォルト値は指定されません。
SQL_DATA_TYPE	SMALLINT	SQL データ型。
SQL_DATETIME_SUB	SMALLINT	日時のサブタイプ・コード。常に NULL。
CHAR_OCTET_LENGTH	INTEGER	文字またはバイナリー・データ型列の最大長 (バイト単位)。
ORDINAL_POSITION	INTEGER	列の序数位置。
IS_NULLABLE	WVARCHAR	常に「YES」。

SYS_PROPERTIES

この表は HSB が内部で使用します。

表 120. SYS_PROPERTIES

列名	データ型	説明
KEY	WVARCHAR	プロパティー ID。
VALUE	WVARCHAR	プロパティーの値。
MODTIME	TIMESTAMP	プロパティーの作成時刻。

SYS_RELAUTH

この表には、表名とユーザー名の各組み合わせに対して発行された GRANT 特権が格納されています。GRANT ステートメントを実行せずにデータベースを作成すると、この表は空になります。

表 121. SYS_RELAUTH

列名	説明
REL_ID	表またはオブジェクトの ID。
UR_ID	ユーザーまたはロールの ID。
PRIV	ユーザーまたはロールの特権に関する情報。各特権は、それを付与した他のユーザー (GRANT_ID) に関連します。
GRANT_ID	付与者 ID。
GRANT_TIM	付与時刻。
GRANT_OPT	「Yes」に設定されている場合、特権を与えられたユーザーは他のユーザーに特権を付与できます。設定される値は「Yes」または「No」です。

SYS_SCHEMAS

SYS_SCHEMAS には、使用可能なスキーマがリストされています。

表 122. SYS_SCHEMAS

列名	データ型	説明
ID	INTEGER	スキーマ ID。
NAME	WVARCHAR	スキーマ名。
OWNER	WVARCHAR	スキーマの所有者名。
CREATIME	TIMESTAMP	作成日時。
SCHEMA_CATALOG	WVARCHAR	スキーマ・カタログ。

SYS_SEQUENCES

表 123. SYS_SEQUENCES

列名	データ型	説明
SEQUENCE_NAME	WVARCHAR	シーケンス名。
ID	INTEGER	ユニーク ID。
DENSE	CHAR	シーケンスが密であるか疎であるか。
SEQUENCE_SCHEMA	WVARCHAR	SEQUENCE_NAME を含んでいるスキーマの名前。
SEQUENCE_CATALOG	WVARCHAR	SEQUENCE_NAME を含んでいるカタログの名前。
CREATIME	TIMESTAMP	作成時刻。

SYS_SERVER

SYS_SERVER 表には、solidDB Universal Cache で SQL パススルーを使用するためのバックエンド・サーバーのログイン・データが含まれています。

solidDB とバックエンドの間の最初のサブスクリプションが作成されるときに、CDC for solidDB インスタンスは、バックエンド CDC インスタンスからバックエンド・データ・サーバーのログイン・データをリトリーブしてから、`CREATE REMOTE SERVER` ステートメントを使用して、これを solidDB システム表 SYS_SERVER に保管します。

表 124. SYS_SERVER 表の定義

列名	データ型	説明
NAME	VARCHAR	バックエンド・サーバーの名前

表 124. SYS_SERVER 表の定義 (続き)

列名	データ型	説明
DRIVER	VARCHAR	使用しない
CONNECT	VARCHAR	使用しない
UID	VARCHAR	バックエンド・サーバーへの接続時に使用するユーザー名
PWD	VARCHAR	バックエンド・サーバーへの接続時に使用するパスワード **注:** パスワード・データは隠されます。

SYS_SYNC_REPLICA_PROPERTIES

表 125. SYS_SYNC_REPLICA_PROPERTIES

列名	データ型	説明
ID	INTEGER	レプリカ ID。
NAME	VARCHAR	プロパティー名。
VALUE	VARCHAR	プロパティー値。

主キーは ID フィールドと NAME フィールドにあります。

SYS_SYNONYM

表 126. SYS_SYNONYM

列名	データ型	説明
TARGET_ID	INTEGER	将来の使用のために予約済み。
SYNON	INTEGER	将来の使用のために予約済み。

SYS_TABLEMODES

表 127. SYS_TABLEMODES

列名	データ型	説明
ID	INTEGER	関係 ID。
MODE	WVARCHAR	並行性制御モード (許容値: OPTIMISTIC、PESSIMISTIC、MAINMEMORY、または MAINMEMORY PESSIMISTIC)。
MODIFY_TIME	TIMESTAMP	最終変更時刻。
MODIFY_USER	WVARCHAR	最後に変更したユーザー。

SYS_TABLEMODES では、モードが明示的に設定された表のモードのみが示されます。デフォルト・モードのままの表のモードは SYS_TABLEMODES で表示されません (デフォルト・モードは、solid.ini の構成パラメーター Pessimistic=Yes を設定しない限り「オプティミスティック」です)。

明示的にオプティミスティックまたはペシミスティックに設定された表の名前とモードをリストするには、以下のコマンドを実行します。

```
SELECT SYS_TABLEMODES.ID, table_name, mode
FROM SYS_TABLES, SYS_TABLEMODES
WHERE SYS_TABLEMODES.ID = SYS_TABLES.ID;
```

出力は以下のようになります。

```
   ID TABLE_NAME   MODE
   -- ----------   ----
10054 TABLE2       OPTIMISTIC
10056 TABLE3       PESSIMISTIC
```

並行性制御モードの設定について詳しくは、 137 ページの『並行性 (ロック方式) モードをオプティミスティックまたはペシミスティックに設定する』を参照してください。

SYS_TABLES

この表には、すべてのシステム表がリストされています。

システム表の表示には制限がありません。つまり、アクセス権限のないユーザーでもシステム表を表示できます。ただし、ユーザー表の情報については、特定のユーザーに対して表示が制限されます。所有者は自身の作成したユーザー表のみを表示でき、ユーザーは INSERT、UPDATE、DELETE、または SELECT の各アクセス権限を持つ表のみを表示できます。アクセス権限のないユーザー表をユーザーが表示することはできません。DBA には制限が適用されません。

表 128. SYS_TABLES

列名	**データ型**	**説明**
ID	INTEGER	ユニークな表 ID。
TABLE_NAME	WVARCHAR	表の名前。
TABLE_TYPE	WVARCHAR	表のタイプ (BASE TABLE または VIEW)。
TABLE_SCHEMA	WVARCHAR	TABLE_NAME が入っているスキーマの名前。
TABLE_CATALOG	WVARCHAR	TABLE_NAME が入っているカタログの名前。
CREATIME	TIMESTAMP	表の作成時刻。
CHECKSTRING	LONG WVARCHAR	表に定義されているチェック・オプション。
REMARKS	LONG WVARCHAR	将来の使用のために予約済み。

SYS_TRIGGERS

このシステム表には、トリガーがリストされています。

トリガーの表示は、特定のユーザーに対して制限されます。所有者は自身の作成したトリガーのみを表示できます。通常のユーザーはトリガーを表示できません。DBA には制限が適用されません。

表 129. SYS_TRIGGERS

列名	データ型	説明
ID	INTEGER	ユニークな表 ID。
TRIGGER_NAME	WVARCHAR	トリガー名。
TRIGGER_TEXT	LONG WVARCHAR	トリガー本体。
TRIGGER_BIN	LONG VARBINARY	コンパイルされた形式のトリガー。
TRIGGER_SCHEMA	WVARCHAR	TRIGGER_NAME を含んでいるスキーマの名前。
TRIGGER_CATALOG	WVARCHAR	TRIGGER_NAME を含んでいるカタログの名前。
TRIGGER_ENABLED	CHAR	トリガーが使用可能である場合は「YES」、そうでない場合は「NO」。
CREATIME	TIMESTAMP	トリガーの作成時刻。
TYPE	INTEGER	将来の使用のために予約済み。
REL_ID	INTEGER	関係 ID。

SYS_TYPES

表 130. SYS_TYPES

列名	データ型	説明
TYPE_NAME	WVARCHAR	データ型の名前。
DATA_TYPE	SMALLINT	(ODBC) データ型番号。
PRECISION	INTEGER	(ODBC) データ型の精度。
LITERAL_PREFIX	WVARCHAR	(ODBC) リテラル値の接頭部。
LITERAL_SUFFIX	WVARCHAR	(ODBC) リテラル値の接尾部。
CREATE_PARAMS	WVARCHAR	(ODBC) このデータ型の列を作成するために必要なパラメーター。

表 130. SYS_TYPES (続き)

列名	データ型	説明
NULLABLE	SMALLINT	(ODBC) データ型が NULL 値を許容するか。
CASE_SENSITIVE	SMALLINT	(ODBC) データ型が大/小文字を区別するか。
SEARCHABLE	SMALLINT	(ODBC) サポートされている検索操作。
UNSIGNED_ATTRIBUTE	SMALLINT	(ODBC) データ型が符号なしか。
MONEY	SMALLINT	(ODBC) データが通貨データ型かどうか。
AUTO_INCREMENT	SMALLINT	(ODBC) データ型が自動インクリメントを行うかどうか。
LOCAL_TYPE_NAME	WVARCHAR	(ODBC) データ型に別の実装で定義された名前があるか。
MINIMUM_SCALE	SMALLINT	(ODBC) データ型の最小の位取り。
MAXIMUM_SCALE	SMALLINT	(ODBC) データ型の最大の位取り。

SYS_UROLE

SYS_UROLE には、ユーザーとロールのマッピングが格納されています。

表 131. SYS_UROLE

列名	データ型	説明
U_ID	INTEGER	ユーザー ID。
R_ID	INTEGER	ロール ID。

SYS_USERS

SYS_USERS には、ユーザーとロールに関する情報が格納されています。

表 132. SYS_USERS

列名	データ型	説明
ID	INTEGER	ユーザーまたはロールの ID。
NAME	WVARCHAR	ユーザーまたはロールの名前。
TYPE	WVARCHAR	ユーザー・タイプ (USER または ROLE)。
PRIV	INTEGER	特権情報。
PASSW	VARBINARY	暗号化された形式のパスワード。

表 132. SYS_USERS (続き)

列名	データ型	説明
PRIORITY	INTEGER	将来の使用のために予約済み。
PRIVATE	INTEGER	ユーザーがプライベートかパブリックか。
LOGIN_CATALOG	WVARCHAR	将来の使用のために予約済み。

SYS_VIEWS

表 133. SYS_VIEWS

列名	データ型	説明
V_ID	INTEGER	このビューのユニーク ID。
TEXT	LONG WVARCHAR	ビュー定義。
CHECKSTRING	LONG WVARCHAR	ビューに定義されているチェック・オプション。
REMARKS	LONG WVARCHAR	将来の使用のために予約済み。

データ同期に使用されるシステム表

solidDB には、同期機能を実装するための数々のシステム表が用意されています。一般に、これらの表は内部でのみ使用されます。ただし、新しいアプリケーションを開発し、トラブルシューティングする際には、これらの表の内容を把握しておく必要があります。

表はアルファベット順に並べてあります。

SYS_BULLETIN_BOARD

この表には、対象のデータベース・カタログでトランザクションが実行されるときにパラメーター掲示板に常に提供されるパーシスタント・パラメーターが格納されています。

表 134. SYS_BULLETIN_BOARD

列名	説明
PARAM_NAME	パーシスタント・パラメーターの名前。
PARAM_VALUE	パラメーターの値。
PARAM_CATALOG	マスター/レプリカ・カタログを定義します。

SYS_PUBLICATION_ARGS

この表には、対象のマスター・データベースのパブリケーション入力引数が格納されています。

表 135. SYS_PUBLICATION_ARGS

列名	説明
PUBL_ID	パブリケーションの内部 ID。
ARG_NUMBER	引数のシーケンス番号。
NAME	引数の名前。
TYPE	引数のタイプ。
LENGTH_OR_PRECISION	引数の長さまたは精度。
SCALE	引数の位取り。

SYS_PUBLICATION_REPLICA_ARGS

この表には、レプリカ・データベース内のパブリケーション引数の定義が格納されています。

表 136. SYS_PUBLICATION_REPLICA_ARGS

列名	説明
MASTER_ID	データのリフレッシュ元であるマスターの内部 ID。
PUBL_ID	パブリケーションの内部 ID。
ARG_NUMBER	引数のシーケンス番号。
NAME	引数の名前。
LENGTH_OR_PRECISION	引数の長さまたは精度。
SCALE	引数の位取り。

SYS_PUBLICATION_REPLICA_STMTARGS

この表には、レプリカ内のパブリケーション引数とステートメントのマッピングが格納されています。

表 137. SYS_PUBLICATION_REPLICA_STMTARGS

列名	説明
MASTER_ID	データのリフレッシュ元であるマスターの内部 ID。
PUBL_ID	パブリケーションの内部 ID。
STMT_NUMBER	ステートメントのシーケンス番号。

表 137. SYS_PUBLICATION_REPLICA_STMTARGS (続き)

列名	説明
STMT_ARG_NUMBER	ステートメント引数のシーケンス番号。
PUBL_ARG_NUMBER	パブリケーション引数のシーケンス番号。

SYS_PUBLICATION_REPLICA_STMTS

この表には、レプリカ・データベース内のパブリケーション・ステートメントの定義が格納されています。

表 138. SYS_PUBLICATION_REPLICA_STMTS

列名	説明
MASTER_ID	データのリフレッシュ元であるマスターの内部 ID。
PUBL_ID	パブリケーションの内部 ID。
STMT_NUMBER	ステートメントのシーケンス番号。
REPLICA_CATALOG	レプリカ・データベースでのターゲット・カタログの名前。
REPLICA_SCHEMA	レプリカ・データベースでのターゲット・スキーマの名前。
REPLICA_TABLE	レプリカ・データベースでのターゲット表の名前。
TABLE_ALIAS	ターゲット表の別名。
REPLICA_FROM_STR	ストリングとしての SQL FROM 表。
WHERE STR	ストリングとしての SQL WHERE 引数。
LEVEL	このパブリケーション階層での対象の SQL ステートメントのレベル。

SYS_PUBLICATION_STMTARGS

この表には、マスター・データベース内のパブリケーション引数とステートメントとのマッピングが格納されています。

表 139. SYS_PUBLICATION_STMTARGS

列名	説明
PUBL_ID	パブリケーションの内部 ID。
STMT_NUMBER	ステートメントのシーケンス番号。
STMT_ARG_NUMBER	ステートメント引数のシーケンス番号。
PUBL_ARG_NUMBER	パブリケーション引数のシーケンス番号。

SYS_PUBLICATION_STMTS

この表には、マスター・データベース内のパブリケーション・ステートメントが格納されています。

表 140. SYS_PUBLICATION_STMTS

列名	説明
PUBL_ID	パブリケーションの内部 ID。
MASTER_SCHEMA	マスター・データベースでのパブリケーション・スキーマの名前。
MASTER_TABLE	マスター・データベースでの表の名前。
REPLICA_SCHEMA	レプリカ・データベースでのスキーマの名前。
REPLICA_TABLE	レプリカ・データベースでの表の名前。
TABLE_ALIAS	ターゲット表の別名。
MASTER_SELECT_STR	ストリングとしての SQL SELECT INTO 列。
REPLICA_SELECT_STR	ストリングとしての SQL SELECT INTO 列。
MASTER_FROM_STR	ストリングとしての SQL SELECT FROM 表。
REPLICA_FROM_STR	ストリングとしての SQL SELECT FROM 表。
WHERE_STR	ストリングとしての SQL WHERE 引数。
DELETEFLAG_STR	内部で使用。
LEVEL	パブリケーション階層での対象の SQL ステートメントのレベル。

SYS_PUBLICATIONS

この表には、対象のマスター・データベースで定義されているパブリケーションが格納されています。

表 141. SYS_PUBLICATIONS

列名	説明
ID	パブリケーションの内部 ID。
NAME	パブリケーションの名前。
CREATOR	パブリケーションの作成者のユーザー ID。
CREATTIME	パブリケーションが作成された日時。
ARGCOUNT	このパブリケーションの入力引数の数。
STMTCOUNT	このパブリケーションに含まれているステートメントの数。

表 141. SYS_PUBLICATIONS (続き)

列名	説明
TIMEOUT	N/A
TEXT	CREATE PUBLICATION ステートメントの内容。
PUBL_CATALOG	マスター・カタログを定義します。

SYS_PUBLICATIONS_REPLICA

この表には、対象のレプリカ・データベースで使用されているパブリケーションが格納されています。

表 142. SYS_PUBLICATIONS_REPLICA

列名	説明
MASTER_ID	データのリフレッシュ元であるマスターの内部 ID。
ID	パブリケーションの内部 ID。
NAME	パブリケーションの名前。
CREATOR	パブリケーションの作成者のユーザー ID。
ARGCOUNT	このパブリケーションの入力引数の数。
STMTCOUNT	このパブリケーションに含まれているステートメントの数。

SYS_SYNC_BOOKMARKS

この表には、マスター・データベースで使用されているブックマークが格納されています。

表 143. SYS_SYNC_BOOKMARKS

列名	説明
BM_ID	ブックマークの内部 ID。
BM_CATALOG	将来の使用のために予約済み。
BM_NAME	ブックマークの名前。
BM_VERSION	マスターでのブックマークの内部バージョン情報。
BM_CREATOR	ブックマークの作成者のユーザー ID。
BM_CREATIME	ブックマークの作成時刻。

SYS_SYNC_HISTORY_COLUMNS

表の同期履歴をオンにする場合、すべての列に対してオンにするか、列のサブセットに対してオンにすることができます。列のサブセットに対してオンにすると、SYS_SYNC_HISTORY_COLUMNS 表に同期履歴情報を保持している列が記録されます。SYS_SYNC_HISTORY_COLUMNS では、同期履歴を保持する列ごとに 1 行が使用されます。

表 144. SYS_SYNC_HISTORY_COLUMNS

列名	説明
REL_ID	同期履歴を保持する表の ID。
COLUMN_NUMBER	この表内の同期履歴を保持する対象となる列の序数 (例えば、表の 2 番目の列の同期履歴を保持する場合は、このフィールドに数値 2 が格納されます)。

SYS_SYNC_INFO

この表には、同期情報がノードごとに 1 行ずつ格納されています。

表 145. SYS_SYNC_INFO

列名	説明
NODE_NAME	マスターまたはレプリカのノード。
NODE_CATALOG	ノードが属しているカタログ。
IS_MASTER	YES の場合、このノードはマスターです。
IS_REPLICA	YES の場合、このノードはレプリカです。
CREATIME	ノードの作成日時。
CREATOR	ノード作成者のユーザー名。

SYS_SYNC_MASTER_MSGINFO

この表には、マスター・データベース内の現在アクティブなメッセージに関する情報が格納されています。

この表のデータは、レプリカ・データベースとマスター・データベースの間の同期プロセスを制御するために使用されます。この表には、トラブルシューティングに役立つ情報も含まれています。マスター・データベースでメッセージの実行がエラーのために停止した場合は、この表を照会して、問題の原因、およびエラーの原因となったトランザクションとステートメントを確認できます。

表 146. SYS_SYNC_MASTER_MSGINFO

列名	説明
STATE	メッセージの 現在 の状態。取り得る値は以下のとおりです。 • 0 = DELETED - N/A (内部の非永続的な状態) • 1 = ERROR - メッセージ処理中にエラーが発生しました。エラーの原因はこの行の error-columns に記録されています。 • 10 = RECEIVED - マスターがレプリカからメッセージを受信しました。 • 11 = SAVED - メッセージがマスター・データベースに保存され、現在処理されています。 • 12 = READY - マスターがメッセージを処理しました。 • 13 = SENT - N/A (内部の非永続的な状態)
REPLICA_ID	メッセージの送信元であるレプリカ・データベースの ID。
MASTER_ID	メッセージの送信先であるマスター・データベースの ID。
MSG_ID	メッセージの内部 ID。
MSG_NAME	ユーザーが指定したメッセージの名前。
MSG_TIME	メッセージの作成時刻。
MSG_BYTE_COUNT	メッセージのサイズ (バイト単位)。
CREATE_UID	メッセージを作成したユーザーの ID。
FORWARD_UID	メッセージを転送したユーザーの ID。
ERROR_CODE	メッセージの実行が終了する原因となったエラーのコード。TRX_ID および STMT_ID の情報から、エラーの原因となったトランザクションとステートメントを特定できます。
ERROR_TEXT	メッセージの実行が終了する原因となったエラーの説明。
TRX_ID	エラーの原因となったトランザクションのシーケンス番号。
STMT_ID	エラーの原因となったトランザクションのステートメントのシーケンス番号。
ORD_ID_COUNT	N/A (内部で使用)。
ORD_ID	N/A (内部で使用)。
FLAGS	NULL または 0 = 通常のメッセージ。 1 = 応答をレプリカに送信するときにメッセージを削除。
FAILED_MSG_ID	主キーの一部である INTEGER 列。通常のメッセージの場合、値はゼロです。LOG_ERRORS オプションが ON で、かつエラーが存在する場合、値は msg_id です。

SYS_SYNC_MASTER_RECEIVED_BLOB_REFS

受信した BLOB は、マスター側にあるこの表に格納されます。この実装によって、論理的には複数回保存される BLOB でも、物理的にはディスクに 1 回だけ保存されるようになります。

表 147. SYS_SYNC_MASTER_ RECEIVED_BLOB_REFS

列名	説明
REPLICA_ID	受信したメッセージの送信元であるレプリカ・データベースの内部 ID。
MSG_ID	メッセージの内部 ID。
BLOB_NUM	BLOB を識別する番号。
DATA	BLOB への参照。

SYS_SYNC_MASTER_RECEIVED_MSGPARTS

この表には、マスター・データベースがレプリカ・データベースから受信し、まだマスター・データベースで処理されていないメッセージの一部が格納されています。

表 148. SYS_SYNC_MASTER_RECEIVED _MSGPARTS

列名	説明
REPLICA_ID	受信したメッセージの送信元であるレプリカ・データベースの内部 ID。
MSG_ID	メッセージの内部 ID。
PART_NUMBER	メッセージ・パーツのシーケンス番号。
DATA_LENGTH	メッセージ・パーツのデータの長さ。
DATA	メッセージ・パーツのデータ。

SYS_SYNC_MASTER_RECEIVED_MSGS

この表には、マスター・データベースがレプリカ・データベースから受信し、まだマスター・データベースで処理されていないメッセージが格納されています。

表 149. SYS_SYNC_MASTER_RECEIVED_MSGS

列名	説明
REPLICA_ID	受信したメッセージの送信元であるレプリカ・データベースの内部 ID。
MSG_ID	メッセージの内部 ID。
CREATIME	メッセージの作成時刻。

表 149. SYS_SYNC_MASTER_RECEIVED_MSGS (続き)

列名	説明
CREATOR	メッセージを作成したユーザーのユーザー ID。

SYS_SYNC_MASTER_STORED_BLOB_REFS

マスター側にあるこの表には、送信される BLOB が格納されます。この実装によって、論理的には複数回保存される BLOB でも、物理的にはディスクに 1 回だけ保存されるようになります。

表 150. SYS_SYNC_MASTER_STORED_BLOB_REFS

列名	説明
REPLICA_ID	メッセージの送信先となるレプリカ・データベースの内部 ID。
MSG_ID	メッセージの内部 ID。
BLOB_NUM	BLOB を識別する番号。
DATA	BLOB への参照。

SYS_SYNC_MASTER_STORED_MSGPARTS

この表には、マスター・データベースで作成されてまだレプリカ・データベースに送信されていないメッセージ結果セットの一部が格納されます。

表 151. SYS_SYNC_MASTER_STORED_MSGPARTS

列名	説明
REPLICA_ID	メッセージの送信先となるレプリカ・データベースの内部 ID。
MSG_ID	メッセージの内部 ID。
ORDER_ID	結果セットのシーケンス番号。
RESULT_SET_ID	結果セットの内部 ID。
RESULT_SET_TYPE	結果セットのタイプ。
PART_NUMBER	結果セット内のメッセージ・パーツのシーケンス番号。
DATA_LENGTH	結果セット内のメッセージ・パーツのデータの長さ。
DATA	メッセージ・パーツのデータ。

SYS_SYNC_MASTER_STORED_MSGS

この表には、マスター・データベースで作成されてまだレプリカ・データベースに送信されていないメッセージが格納されます。

表 152. SYS_SYNC_MASTER_STORED_MSGS

列名	説明
REPLICA_ID	メッセージの送信先となるレプリカ・データベースの内部 ID。
MSG_ID	メッセージの内部 ID。
CREATIME	メッセージの作成時刻。
CREATOR	メッセージを作成したユーザーのユーザー ID。

SYS_SYNC_MASTER_SUBSC_REQ

この表には、マスターでの実行を待っている要求されたサブスクリプションがリストされています。

表 153. SYS_SYNC_MASTER_SUBSC_REQ

列名	説明
REPLICA_ID	ステートメントを送信したレプリカの内部 ID。
MSG_ID	ステートメントを受け取ったメッセージの内部 ID。
ORD_ID	サブスクリプションのシーケンス番号。
TRX_ID	サブスクリプションが属しているトランザクションの内部 ID。
STMT_ID	サブスクリプションでのステートメントの内部 ID。
REQUEST_ID	N/A
PUBL_ID	サブスクライブ/リフレッシュされるパブリケーションの内部 ID。
VERSION	マスターでのサブスクリプションの内部バージョン情報。
REPLICA_VERSION	レプリカでのサブスクリプションの内部バージョン情報。
FULLSUBSC	サブスクリプションがフルかインクリメンタルか。

SYS_SYNC_MASTER_VERSIONS

この表には、マスター・データベースからレプリカ・データベースへのサブスクリプション (サブスクライブされたもの) がリストされています。

表 154. SYS_SYNC_MASTER_VERSIONS

列名	説明
REPLICA_ID	レプリカ・データベースの内部 ID。
REQUEST_ID	サブスクリプションのシーケンス番号。
VERS_TIME	サブスクリプションの作成時刻。

表 154. SYS_SYNC_MASTER_VERSIONS (続き)

列名	説明
PUBL_ID	パブリケーションの ID。
TABNAME	パブリケーションの表の名前。
TABSCHEMA	表のスキーマの名前。
PARAM_CRC	N/A (内部で使用)。
PARAM	バイナリー・フォーマットでのパブリケーションのパラメーター。
VERSION	レプリカ・データベースから要求されたデータのバージョン。

SYS_SYNC_MASTERS

この表には、レプリカがアクセスするマスター・データベースがリストされています。

表 155. SYS_SYNC_MASTERS

列名	説明
NAME	マスター・データベースの名前。
ID	マスター・データベースの内部 ID。
REMOTE_NAME	N/A
REPLICA_NAME	レプリカ・データベースの名前。
REPLICA_ID	レプリカ・データベースのサロゲート ID。
REPLICA_CATALOG	このマスターに登録されているレプリカ・カタログ。
CONNECT	マスター・データベースの接続ストリング。
CREATOR	データベースをマスターとして設定したユーザーの ID。
ISDEFAULT	将来の使用のために予約済み。

SYS_SYNC_RECEIVED_BLOB_ARGS

この表はマスター側にあります。レプリカからのメッセージが抽出されるときに、この表に BLOB パラメーターが保存されます。この行は、メッセージ内のトランザクションが実行されるまで保持されます。

表 156. SYS_SYNC_RECEIVED_BLOB_ARGS

列名	説明
REPLICA	BLOB パラメーターを送信したレプリカの内部 ID。

表 156. SYS_SYNC_RECEIVED_BLOB_ARGS (続き)

列名	説明
MSG	メッセージの内部 ID。
ORD_ID	BLOB パートのシーケンス番号。
TRX_ID	トランザクションを識別するトランザクション ID。
ID	ユーザーの内部 ID。
ARGNO	パラメーターの番号。
ARG_VALUE	バイナリー・フォーマットでのパラメーターの値。

SYS_SYNC_RECEIVED_STMTS

この表には、マスター・データベースで受信した伝搬ステートメントが格納されています。

表 157. SYS_SYNC_RECEIVED_STMTS

列名	説明
REPLICA	ステートメントを送信したレプリカの内部 ID。
MSG	ステートメントを受け取ったメッセージの内部 ID。
ORD_ID	N/A
TXN_ID	ステートメントが属しているトランザクションの内部 ID。
ID	トランザクションでのステートメントのシーケンス番号。
CLASS	定数のタイプ。
STRING	ストリングとしての SQL ステートメント。
ARG_COUNT	ステートメントにバインドされているパラメーターの数。
ARG_TYPES	ステートメントにバインドされているパラメーターのタイプ。
ARG_VALUES	バイナリー・フォーマットでのパラメーターの値。
USER_ID	ステートメントを保存したユーザーの ID。
REQUEST_ID	N/A
FLAGS	エラー処理モード (IGNORE_ERRORS、LOG_ERRORSなど)。
ERRCODE	マスターでの実行中にステートメントが失敗した場合のエラー・コード。
ERR_STR	マスターでの実行中にステートメントが失敗した場合に発生したエラーの説明。

SYS_SYNC_REPLICA_MSGINFO

この表には、レプリカ・データベース内の現在アクティブなメッセージに関する情報が格納されています。

この表のデータは、レプリカ・データベースとマスター・データベースの間の同期プロセスを制御するために使用されます。この表には、トラブルシューティングに役立つ情報も含まれています。レプリカ・データベースでメッセージの実行がエラーのために停止した場合は、この表を照会して、問題の原因、およびエラーの原因となったトランザクションとステートメントを確認できます。

表 158. SYS_SYNC_REPLICA_MSGINFO

列名	説明
STATE	メッセージの 現在 の状態。取り得る値は以下のとおりです。 • 0 = DELETED - N/A (内部の非永続的な状態) • 1 = ERROR - メッセージ処理中に内部エラーが発生しました。エラーの原因はこの行の error-columns に記録されています。 • 20 = R_INIT - N/A (内部の非永続的な状態) • 21 = R_INITEND - N/A (内部の非永続的な状態) • 22 = R_SAVED - レプリカが出力メッセージを保存しました。 • 23 = R_SENT - レプリカがマスターにメッセージを送信しました。 • 24 = R_RECEIVED - レプリカがマスターから応答メッセージを受信しました。 • 25 = R_EXECUTE - レプリカ内の応答メッセージは実行できる状態にあります。 • 26 = R_EXECUTE_NOTIFYMASTER - レプリカが応答を受信しましたが、マスターがまだそれを確認していません。
MASTER_ID	メッセージの送信先であるマスター・データベースの ID。
MASTER_NAME	メッセージの送信先であるマスター・データベースの名前。
MSG_ID	メッセージの内部 ID。
MSG_NAME	ユーザーが指定したメッセージの名前。
MSG_TIME	メッセージの作成時刻。
MSG_BYTE_COUNT	メッセージのサイズ (バイト単位)。
CREATE_UID	メッセージを作成したユーザーの ID。
FORWARD_UID	メッセージを送信したユーザーの ID。
ERROR_CODE	メッセージの実行が終了する原因となったエラーのコード。
ERROR_TEXT	メッセージの実行が終了する原因となったエラーの説明。

表 158. SYS_SYNC_REPLICA_MSGINFO (続き)

列名	説明
FLAGS	NULL または 0 = 通常のメッセージ。 1 = マスターから応答を受信するときにメッセージを削除。 3 = メッセージは登録メッセージ。

SYS_SYNC_REPLICA_RECEIVED_BLOB_REFS

この表には受信した BLOB が格納されます。この実装によって、論理的には複数回保存される BLOB でも、物理的にはディスクに 1 回だけ保存されるようになります。

表 159. SYS_SYNC_REPLICA_RECEIVED_ BLOB_REFS

列名	説明
MASTER_ID	受信したメッセージの送信元であるマスター・データベースの内部 ID。
MSG_ID	メッセージの内部 ID。
BLOB_NUM	BLOB を識別する番号。
DATA	BLOB への参照。

SYS_SYNC_REPLICA_RECEIVED_MSGPARTS

この表には、レプリカ・データベースがマスター・データベースから受信し、まだレプリカ・データベースで処理されていない応答メッセージの一部が格納されています。

表 160. SYS_SYNC_REPLICA_RECEIVED_ MSGPARTS

列名	説明
MASTER_ID	受信したメッセージの送信元であるマスター・データベースの内部 ID。
MSG_ID	メッセージの内部 ID。
PART_NUMBER	メッセージ・パーツのシーケンス番号。
DATA_LENGTH	メッセージ・パーツのデータの長さ。
RESULT_SET_TYPE	結果セットのタイプ。
DATA	メッセージ・パーツのデータ。

SYS_SYNC_REPLICA_RECEIVED_MSGS

この表には、レプリカ・データベースがマスター・データベースから受信し、まだレプリカ・データベースで処理されていない応答メッセージが格納されています。

表 161. SYS_SYNC_REPLICA_RECEIVED_MSGS

列名	説明
MASTER_ID	受信したメッセージの送信元であるマスター・データベースの内部 ID。
MSG_ID	メッセージの内部 ID。
CREATIME	メッセージの作成時刻。
CREATOR	メッセージを作成したユーザーのユーザー ID。

SYS_SYNC_REPLICA_STORED_BLOB_REFS

この表には、フロー・メッセージ内の BLOB が格納されます。この実装によって、論理的には複数回保存される BLOB でも、物理的にはディスクに 1 回だけ保存されるようになります。

表 162. SYS_SYNC_REPLICA_STORED_BLOB_REFS

列名	説明
MASTER_ID	受信したメッセージの送信元であるマスター・データベースの内部 ID。
MSG_ID	メッセージの内部 ID。
BLOB_NUM	BLOB を識別する番号。
DATA	BLOB への参照。

SYS_SYNC_REPLICA_STORED_MSGS

この表には、レプリカ・データベースで作成されてまだマスター・データベースに送信されていないメッセージが格納されています。

表 163. SYS_SYNC_REPLICA_STORED_MSGS

列名	説明
MASTER_ID	メッセージの送信先となるマスター・データベースの内部 ID。
MSG_ID	メッセージの内部 ID。
CREATIME	メッセージの作成時刻。
CREATOR	メッセージを作成したユーザーのユーザー ID。

SYS_SYNC_REPLICA_STORED_MSGPARTS

この表には、レプリカ・データベースで作成されてまだマスター・データベースに送信されていないメッセージの一部が格納されています。

表 164. SYS_SYNC_REPLICA_STORED_MSGPARTS

列名	説明
MASTER_ID	メッセージの送信先となるマスター・データベースの内部 ID。
MSG_ID	メッセージの内部 ID。
PART_NUMBER	メッセージ・パーツのシーケンス番号。
DATA_LENGTH	メッセージ・パーツのデータの長さ。
DATA	メッセージ・パーツのデータ。

SYS_SYNC_REPLICA_VERSIONS

この表には、マスター・データベースからこのレプリカ・データベースへのサブスクリプション (サブスクライブされたもの) がリストされています。

表 165. SYS_SYNC_REPLICA_VERSIONS

列名	説明
BOOKMARK_ID	サブスクリプションでのブックマークの内部 ID。
REQUEST_ID	サブスクリプションでのパブリケーション要求の内部 ID。
VERS_TIME	サブスクリプションの作成時刻。
PUBL_ID	サブスクライブされたパブリケーションの ID。
MASTER_ID	パブリケーションのサブスクライブ元となったマスター・データベースの ID。
PARAM_CRC	内部で使用。
PARAM	サブスクリプションのパラメーター。
VERSION	マスター・データベースでのサブスクライブされたパブリケーションのバージョン番号。
LOCAL VERSION	レプリカ・データベースでのサブスクライブされたパブリケーションのバージョン番号。
PUBL_NAME	パブリケーションの名前。
REPLY_ID	パブリケーション応答の ID。

SYS_SYNC_REPLICAS

この表には、マスターに登録されているレプリカ・データベースがリストされています。

表 166. SYS_SYNC_REPLICAS

列名	説明
NAME	レプリカ・データベースの名前。
ID	レプリカ・データベースの内部 ID。
MASTER_NAME	N/A
MASTER_CATALOG	レプリカが登録されているカタログ。
CONNECT	レプリカの接続ストリング (「tcp MyWorkstation1315」など)。

SYS_SYNC_SAVED_BLOB_ARGS

ユーザーが BLOB パラメーターを伴うトランザクションをレプリカで保存すると、BLOB への参照が SYS_SYNC_SAVED_BLOB_ARGS 表に保存されます。この参照は、SYS_SYNC_REPLICA_STORED_BLOB_REFS 表を指します。この行は、送信されるメッセージの準備が完了するまで保持されます。

表 167. SYS_SYNC_SAVED_BLOB_ARGS

列名	説明
MASTER	パラメーターの送信先となるマスター・データベースの ID。
TRX_ID	トランザクションを識別するトランザクション ID。
ID	ユーザーの内部 ID。
ARGNO	パラメーターの番号。
ARG_VALUE	バイナリー・フォーマットでのパラメーターの値。

SYS_SYNC_SAVED_STMTS

この表には、後から伝搬する目的でレプリカ・データベースに保存されたステートメントが格納されています。

表 168. SYS_SYNC_SAVED_STMTS

列名	説明
MASTER	ステートメントの伝搬先となるマスター・データベースの内部 ID。
TRX_ID	ステートメントが属しているトランザクションの内部 ID。
ID	トランザクションでのステートメントのシーケンス番号。

表 168. SYS_SYNC_SAVED_STMTS (続き)

列名	説明
CLASS	定数のタイプ。
STRING	ストリングとしての SQL ステートメント。
ARG_COUNT	ステートメントにバインドされているパラメーターの数。
ARG_TYPES	ステートメントにバインドされているパラメーターのタイプ。
ARG_VALUES	バイナリー・フォーマットでのパラメーターの値。
USER_ID	ステートメントを保存したユーザーの ID。
REQUEST_ID	N/A
FLAGS	エラー処理モード (IGNORE_ERRORS、LOG_ERRORSなど)。

SYS_SYNC_TRX_PROPERTIES

トランザクションを保存するときに、それらにプロパティーを割り当てることができます。このプロパティーは、後で伝搬の対象となるトランザクションを選択するために使用できます。これらのプロパティーは、SYS_SYNC_TRX_PROPERTIES 表に保存されます。

表 169. SYS_SYNC_TRX_PROPERTIES

列名	説明
TRX_ID	トランザクションを識別するトランザクション ID。
NAME	トランザクションのプロパティーの名前 (COLOR など)。
VALUE_STR	トランザクションのプロパティーの値 (RED など)。

SYS_SYNC_USERMAPS

この表では、レプリカ・ユーザーの ID が SYS_SYNC_USERS 表のマスター・ユーザーにマップされます。

表 170. SYS_SYNC_USERMAPS

列名	説明
REPLICA_UID	マスター・ユーザーにマップされたレプリカ・ユーザーの ID。
MASTER_ID	マスター ID。
REPLICA_USERNAME	レプリカ・ユーザー名。
MASTER_USERNAME	マスター・ユーザー名。
PASSW	マスター・ユーザー名の暗号化されたパスワード。

SYS_SYNC_USERS

この表には、レプリカ・データベースの同期機能にアクセスできるユーザーがリストされています。この機能には、トランザクションの保存や同期メッセージの作成が含まれます。

レプリカでは、以下のコマンドを使用したメッセージでこの表のデータがマスターからダウンロードされます。

```
MESSAGE unique-message-name APPEND SYNC_CONFIG
['sync-config-arg']
```

表 171. SYS_SYNC_USERS

列名	説明
MASTER_ID	マスター・データベースの内部 ID。
ID	ユーザーの内部 ID。
NAME	ユーザー名。
PASSW	ユーザーの暗号化されたパスワード。

システム・ビュー

solidDB では、X/Open SQL 規格で規定されているとおりにビューがサポートされています。

COLUMNS

COLUMNS システム・ビューは、現行ユーザーがアクセスできる列を識別します。

表 172. COLUMNS

列名	データ型	説明
TABLE_CATALOG	WVARCHAR	TABLE_NAME が入っているカタログの名前。
TABLE_SCHEMA	WVARCHAR	TABLE_NAME が入っているスキーマの名前。
TABLE_NAME	WVARCHAR	表またはビューの名前。
COLUMN_NAME	WVARCHAR	指定された表またはビューの列の名前。
DATA_TYPE	WVARCHAR	列のデータ型。
SQL_DATA_TYPE_NUM	SMALLINT	ODBC 準拠のデータ型番号。
CHAR_MAX_LENGTH	INTEGER	文字データ型列の最大長。その他の場合は NULL。

表 172. COLUMNS (続き)

列名	データ型	説明
NUMERIC_PRECISION	INTEGER	DATA_TYPE が適切な数値データ型である場合は、列の小数部精度の桁数。NUMERIC_PREC_RADIX は測定単位を示します。その他の数値型の場合、列内で許可される 10 進数字の合計数が入っています。文字データ型の場合は NULL。
NUMERIC_PREC_RADIX	SMALLINT	DATA_TYPE がいずれかの概数データ型である場合は、数値精度の基数。それ以外の場合は NULL。
NUMERIC_SCALE	SMALLINT	小数点の右側の有効数字の合計数。INTEGER および SMALLINT では 0。その他の場合は NULL。
NULLABLE	CHAR	列が NULL 可能でないことが分かっている場合は「NO」、それ以外の場合は「YES」。
NULLABLE_ODBC	SMALLINT	(ODBC) 列が NULL 可能ないことが分かっている場合は「0」、それ以外の場合は「1」。
REMARKS	LONG WVARCHAR	将来の使用のために予約済み。

SERVER_INFO

SERVER_INFO システム・ビューは、現行のデータベース・システムまたはサーバーの属性を提供します。

表 173. SERVER_INFO

列名	データ型	説明
SERVER_ATTRIBUTE	WVARCHAR	サーバーの属性を識別します。
ATTRIBUTE_VALUE	WVARCHAR	属性の値。

TABLES

TABLES システム・ビューは、現在のユーザーがアクセスできる表を特定します。

表 174. TABLES

列名	データ型	説明
TABLE_CATALOG	WVARCHAR	TABLE_NAME が入っているカタログの名前。
TABLE_SCHEMA	WVARCHAR	TABLE_NAME が入っているスキーマの名前。

表 174. TABLES (続き)

列名	データ型	説明
TABLE_NAME	WVARCHAR	表またはビューの名前。
TABLE_TYPE	WVARCHAR	表のタイプ。
REMARKS	LONG WVARCHAR	将来の使用のために予約済み。

USERS

USERS システム・ビューは、ユーザーおよびロールを識別します。

表 175. USERS

列名	データ型	説明
ID	INTEGER	ユーザーまたはロールの ID。
NAME	WVARCHAR	ユーザーまたはロールの名前。
TYPE	WVARCHAR	ユーザー・タイプ (USER または ROLE)。
PRIV	INTEGER	特権情報。
PRIORITY	INTEGER	将来の使用のために予約済み。
PRIVATE	INTEGER	ユーザーがプライベートかパブリックか。

同期関連ビュー

solidDB には、マスターとレプリカの間の同期メッセージに関する情報を表示する 4 つのビューが用意されています。このうちの 2 つのビュー (SYNC_FAILED_MESSAGES and SYNC_FAILED_MASTER_MESSAGES) には失敗したメッセージが表示され、残りの 2 つのビュー (SYNC_ACTIVE_MESSAGES と SYNC_ACTIVE_MASTER_MESSAGES) にはアクティブなメッセージが表示されます。

SYNC_FAILED_MESSAGES

この表はマスター側にあり、レプリカから受信したメッセージに関する情報を保持します。1 つの単純なビューを使用して、失敗したメッセージに関する必要なすべての情報を表示できます。

```
SELECT * FROM SYNC_FAILED_MESSAGES.
```

これは以下の列を返します。

表 176. SYNC_FAILED_MESSAGES

列名	データ型	説明
REPLICA_NAME	WVARCHAR	メッセージを送信したレプリカの所定のノード名。
MESSAGE_NAME	WVARCHAR	ユーザーが指定したメッセージの名前。
TRANSACTION_ID	BINARY	失敗したレプリカ・トランザクションの内部 ID。
STATEMENT_ID	INTEGER	トランザクション内でのステートメントのシーケンス番号。
STATEMENT_STRING	WVARCHAR	ストリングとしての SQL ステートメント。
ERROR_CODE	INTEGER	メッセージの実行が終了する原因となったエラーのコード。
ERROR_MESSAGE	VARCHAR	エラーの説明。

すべてのユーザーがこのビューにアクセスできます。特権は必要ありません。

SYNC_FAILED_MASTER_MESSAGES

この表はレプリカ側にあり、マスターに送信されたメッセージに関する情報を保持します。1 つの単純なビューを使用して、失敗したメッセージに関する必要なすべての情報を表示できます。

```
SELECT * FROM SYNC_FAILED_MASTER_MESSAGES.
```

これは以下の列を返します。

表 177. SYNC_FAILED_MASTER_MESSAGES

列名	データ型	説明
MASTER_NAME	WVARCHAR	マスターの所定のノード名。
MESSAGE_NAME	WVARCHAR	ユーザーが指定したメッセージの名前。
ERROR_CODE	INTEGER	メッセージの実行が終了する原因となったエラーのコード。
ERROR_MESSAGE	VARCHAR	エラーの説明。

すべてのユーザーがこのビューにアクセスできます。特権は必要ありません。

SYNC_ACTIVE_MESSAGES

この表はマスター側にあり、レプリカから受信したメッセージに関する情報を保持します。これは以下の列を返します。

表 178. SYNC_ACTIVE_MESSAGES

列名	データ型	説明
REPLICA_NAME	WVARCHAR	レプリカの所定のノード名。
MESSAGE_NAME	WVARCHAR	ユーザーが指定したメッセージの名前。
MESSAGE STATE	VARCHAR	ストリングとしてのメッセージの現在の状態。詳しくは、システム表 SYS_SYNC_MASTER_MSGINFO を参照してください。

すべてのユーザーがこのビューにアクセスできます。特権は必要ありません。

SYNC_ACTIVE_MASTER_MESSAGES

この表はレプリカ側にあり、マスターに送信されたメッセージに関する情報を保持します。1 つの単純なビューを使用して、失敗したメッセージに関する必要なすべての情報を表示できます。

```
SELECT * FROM SYNC_FAILED_MASTER_MESSAGES
```

これは以下の列を返します。

表 179. SYNC_ACTIVE_MASTER_MESSAGES

列名	データ型	説明
MASTER_NAME	WVARCHAR	マスターの所定のノード名。
MESSAGE_NAME	WVARCHAR	ユーザーが指定したメッセージの名前。
MESSAGE STATE	VARCHAR	ストリングとしてのメッセージの現在の状態。詳しくは、システム表 SYS_SYNC_REPLICA_MSGINFO を参照してください。

すべてのユーザーがこのビューにアクセスできます。特権は必要ありません。

付録 E. データベース仮想表

SYS_LOG

SYS_LOG 表は仮想表です。ログ・リーダーが SYS_LOG 表の SQL 要求を受け取ると、適切な結果セットが内部ログ構造から動的に生成されます。

表 180. SYS_LOG 表の定義

列名	データ型	説明
RECID	INTEGER	レコード ID 詳しくは 406 ページの『レコード ID の説明』というセクションを参照してください。
RECNAME	VARCHAR	ログ・レコードの記述名。これは、RECID フィールドに対応する、人が読みやすい形式です。
TRXID	INTEGER	トランザクション ID
STMTTRXID	INTEGER	トランザクション内のステートメント ID
RELID	INTEGER	SYS_TABLES 表に保管された表 ID
FLAGS	INTEGER	フラグ・フィールド。このフィールドには以下の意味があります。 • 0: NOP 使用可能なデータはありません。この場合、その他のすべての列値は NULL です。 • ビット 0 を設定: DATA データが使用可能です。 • ビット 1 を設定: SHUTDOWN サーバーのシャットダウンが開始しました。 • ビット 6 を設定: CAPTURE_OFF DBE_LOGREADER_LOG_REC_TRX_START としてログ・レコードで設定された場合、このトランザクション内の操作は伝搬されません。
LOGADDR	VARBINARY	これは、現在のログ・レコードのログ・アドレスです。 LOGADDR は、SYS_LOG からの読み取りの開始点として使用できます。 ログ・アドレスの長さは 20 バイトです。 バイナリー比較を使用してログ・アドレスを比較すると、どのログ・アドレスが先であるかを確認できます。

表 180. SYS_LOG 表の定義 (続き)

列名	データ型	説明
DATA	VARBINARY	表内の実際のユーザー・データ 詳細な説明については、下記の『行データ用の DATA 列形式』および 408 ページの『DDL データ用の DATA 列形式』のセクションを参照してください。
TEXTDATA	LONG VARCHAR	使用しない。常に NULL。(DATA に対応する、ユーザーが読みやすいテキスト形式の提供を目的としています。)

レコード ID の説明

表 181. レコード ID の説明

RECNAME	RECID	説明
DBE_LOGREADER_LOG_REC_EMPTY	0	空のレコード。使用可能なデータはありません。
DBE_LOGREADER_LOG_REC_INSERT	1	行挿入。
DBE_LOGREADER_LOG_REC_UPDATE	3	更新用の変更後イメージ。主キーが変更された場合、それは更新として表示されません。挿入および削除として表示されます。
DBE_LOGREADER_LOG_REC_DELETE_FULL	4	すべての列値がログに格納される行削除。
DBE_LOGREADER_LOG_REC_UPDATE_BEFOREIMAGE	5	更新用の変更前イメージ。主キーが変更された場合、それは更新として表示されません。挿入および削除として表示されます。
DBE_LOGREADER_LOG_REC_TRX_START	7	トランザクションの開始。伝搬されない削除を検出する方法については、FLAGS フィールドを参照してください。
DBE_LOGREADER_LOG_REC_COMMIT	12	トランザクションのコミット。
DBE_LOGREADER_LOG_REC_DDL	13	DDL 操作。
DBE_LOGREADER_LOG_REC_SQL	6	DDL 操作の SQL ストリング。

行データ用の DATA 列形式

このセクションでは、RECNAME フィールドの値が以下のようになっている行の DATA 列形式について説明します。

- DBE_LOGREADER_LOG_REC_INSERT
- DBE_LOGREADER_LOG_REC_UPDATE
- DBE_LOGREADER_LOG_REC_UPDATE_BEFOREIMAGE
- DBE_LOGREADER_LOG_REC_DELETE_FULL

DATA 列は、表内のユーザー列ごとに <length><data> のペアがバイナリー形式で含まれるようにフォーマット設定されています。列の順序は、表で定義された順序に従います。

整数値の形式は、いわゆる MSB (最上位バイト) ファースト形式です。これは、例えば、4 バイトの 16 進整数 0x12345678 の各バイトが、12、34、56、78 の順で格納されることを意味します。

<length> フィールドは、常に、MSB ファースト形式の 4 バイトの整数です。

NULL 値が格納される場合は、<length> フィールドが -1 となります。

BLOB 値が格納される場合は、<length> フィールドが -2 となり、それに続く <data> フィールドに MSB ファースト形式の 8 バイトの整数が入ります。

その 8 バイトの整数は固有の BLOB ID です。新規 BLOB 値が格納されたか、古い値が変更された場合、その値には常に固有の BLOB ID が割り振られます。

表 182. 行データ用の DATA 型の説明

名前	説明	SQL データ型
BINARY UNICODE CHAR	生データ・バイトとして格納されます。列に格納されるものと同じ形式です。	CHAR VARCHAR LONG VARCHAR WCHAR WVARCHAR LONG WVARCHAR BINARY VARBINARY LONG VARBINARY
DOUBLE FLOAT	MSB ファースト形式の 8 バイトの IEEE 浮動小数点数として格納されます。	FLOAT REAL DOUBLE PRECISION
INTEGER	MSB ファースト形式の 4 バイトの整数として格納されます。	BIT TINYINT
BIGINT	MSB ファースト形式の 8 バイトの整数として格納されます。	BIGINT

表 182. 行データ用の DATA 型の説明 (続き)

名前	説明	SQL データ型
DATE	11 バイトの生データ形式として格納されます。それらのバイトには以下の意味があります。 • 0-1: 年 (MSB ファースト形式で 2 バイト) • 2: 月 (1 バイト) • 3: 日 (1 バイト) • 4: 時 (1 バイト) • 5: 分 (1 バイト) • 6: 秒 (1 バイト) • 7: 秒の小数部 (MSB ファースト形式で 4 バイト)	DATE TIME TIMESTAMP
DFLOAT	ストリング形式の 10 進浮動小数点数。	NUMERIC DECIMAL

DDL データ用の DATA 列形式

RECNAME フィールドの値が DBE_LOGREADER_LOG_REC_DDL である行では、DATA 列の形式は以下のようになります。

- DATA 列は、<length><logrecid><length><info> の値が含まれるようにフォーマット設定されています。

RECNAME フィールドの値が DBE_LOGREADER_LOG_REC_SQL である行では、DATA 列の形式は以下のようになります。

- DATA 列は、<length><info> の値が含まれるようにフォーマット設定されています。<info> フィールドの値は、SQL ストリングです。

整数値の形式は、いわゆる MSB (最上位バイト) ファースト形式です。これは、4 バイトの 16 進整数 0x12345678 の各バイトが、12、34、56、78 の順で格納されることを意味します。

<length> フィールドは、常に、MSB ファースト形式の 4 バイトの整数です。

使用できる <logrecid> フィールドの説明と、<info> の値を、以下の表に示します。

表 183. DDL データ用の DATA 型の説明

LOGRECID	説明	INFO
45	表を作成します。	表の完全修飾名。
17	表をドロップします。	表名。
47	表の名前を変更します。	表の完全修飾名。
22	表を変更します。	表名。
73	表を切り捨てます。	表名。
16	索引を作成します。	索引名。
18	索引をドロップします。	索引名。
46	ビューを作成します。	ビューの完全修飾名。
20	ビューをドロップします。	ビュー名。

表 183. DDL データ用の DATA 型の説明 (続き)

LOGRECID	説明	INFO
28	シーケンスを作成します。	シーケンス名。
30	シーケンスをドロップします。	シーケンス名。
27	カウンターを作成します。	カウンター名。
29	カウンターをドロップします。	カウンター名。

付録 F. システム・ストアード・プロシージャー

この章では、タスクの簡素化に役立つ solidDB 付属のストアード・プロシージャーについて説明します。これらのストアード・プロシージャーはサーバーに組み込まれており、ユーザーが使用できるライブラリーと見なすことができます。

同期関連ストアード・プロシージャー

以下に示すシステム・プロシージャーは、定常的な同期タスクを簡略化します。この利便性を維持するには、「不必要に」エラーが発生しないようにする必要があります。

同期システム・プロシージャーを実行するには、管理者または同期管理者のアクセス権限が必要です。

SYNC_SETUP_CATALOG

```
CALL SYNC_SETUP_CATALOG (
    catalog_name,   -- WVARCHAR
    node_name,      -- WVARCHAR
    is_master,      -- INTEGER
    is_replica      -- INTEGER
)
```

EXECUTES ON: マスターまたはレプリカ。

SYNC_SETUP_CATALOG() プロシージャーは、カタログを作成し、これにノード名を割り当て、カタログのロールをマスター、レプリカ、またはその両方に設定します。

catalog_name パラメーターが NULL の場合は、指定したノード名とロールが現在のカタログに割り当てられます。

is_master と *is_replica* については、値 0 が「no」を意味し、それ以外の値は「yes」を意味します。少なくともどちらかをゼロ以外にする必要があります。1 つのカタログをレプリカとマスターの両方にすることができるため、*is_master* と *is_replica* の両方をゼロ以外に設定しても問題はありません。

表 184. SYNC_SETUP_CATALOG のエラー・コード

RC	テキスト	説明
13047	操作する特権がありません	
13110	NULL は許可されません	NULL にできるのはカタログ名のみです。それ以外のパラメーターは NULL 以外にする必要があります。
13133	この製品に有効なライセンスではありません	

表 184. *SYNC_SETUP_CATALOG* のエラー・コード *(*続き*)*

RC	テキスト	説明
25031	トランザクションがアクティブで、操作が失敗しました	ユーザーの行った変更がまだコミットされていません。
25052	ノード名を *node_name* に設定できませんでした	*node_name* が無効である可能性があります。
25059	登録後にノード名を変更することはできません	カタログには既に名前と 1 つ以上のレプリカがあります。

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: レプリカ。

SYNC_REGISTER_REPLICA() システム・プロシージャーは、新しいカタログを作成し、指定されたマスターにレプリカを登録します。ユーザーには、管理者または同期管理者のアクセス権限が必要です。

master_network_name は、マスター・データベース・サーバーの接続ストリングです。

指定したカタログが存在しない場合は、自動的に作成されます。

指定したレプリカ・カタログ名が NULL の場合は、現在のカタログが使用されます。マスター・ノード名も NULL にすることができます。それ以外のパラメーターは NULL にできません。

登録が失敗すると、マスター側とレプリカ側の両方が元の状況にリセットされます。いずれかのパラメーターの値が正しくないと、エラーが返されます。

データを変更したオープン・トランザクションが存在する場合は、この関数からエラーが返されます。

このシステム・プロシージャーは、結果セットを返しません。

表 185. *SYNC_REGISTER_REPLICA* のエラー・コード

RC	テキスト	説明
13047	操作する特権がありません	

表 185. *SYNC_REGISTER_REPLICA のエラー・コード (続き)*

RC	テキスト	説明
13110	NULL は許可されません	NULL にできるのはカタログ名とマスター・ノード名のみです。それ以外のパラメーターは NULL 以外にする必要があります。
13133	この製品に有効なライセンスではありません	
21xxx	通信エラー	マスターに接続できませんでした。21xxx エラーについて詳しくは、「*solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。
25005	メッセージは既にアクティブです	
25031	トランザクションがアクティブで、操作が失敗しました	ユーザーの行った変更がまだコミットされていません。
25035	メッセージは使用中です	
25051	未完了のメッセージが見つかりました	
25052	ノード名を *node_name* に設定できませんでした	*node_name* が無効である可能性があります。
25056	自動コミットは許可されません	このストアード・プロシージャーは、自動コミットをオフにして実行する必要があります。
25057	レプリカ・データベースは、既にマスター・データベースに登録されています	
25059	登録後にノード名を変更することはできません	

SYNC_UNREGISTER_REPLICA

```
CALL SYNC_UNREGISTER_REPLICA (
    replica_catalog_name,   -- WVARCHAR
    drop_catalog,           -- INTEGER
    force                   -- INTEGER
)
```

EXECUTES ON: レプリカ。

SYNC_UNREGISTER_REPLICA() システム・プロシージャーは、指定されたレプリカ・カタログの登録をマスターから抹消し、*drop_catalog* パラメーターがゼロ以外の値であればそのレプリカ・カタログをドロップします。このレプリカにハングしているメッセージがある場合は、システムの両側でそのメッセージが削除されます。ユーザーには、管理者または同期管理者のアクセス権限が必要です。

レプリカ・カタログ名が NULL の場合は、現在のカタログが使用されます。force がゼロ以外である場合は、マスターにこのレプリカのメッセージが存在している場合でも、マスターが登録抹消を受け付けます。その場合、対象のメッセージは削除されます。

ユーザーの変更がコミットされていない (つまりオープン・トランザクションである) 場合は、呼び出しがエラーで失敗します。

このシステム・プロシージャーは、結果セットを返しません。

表 186. SYNC_UNREGISTER_REPLICA のエラー・コード

RC	**テキスト**	**説明**
13047	操作する特権がありません	
13110	NULL は許可されません	drop_catalog がゼロ以外である場合は、カタログ名をNULL にできません。
13133	この製品に有効なライセンスではありません	
21xxx	通信エラー	マスターに接続できませんでした。21xxx エラーについて詳しくは、「*solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。
25005	メッセージは既にアクティブです	
25019	データベースがレプリカ・データベースではありません	
25020	データベースがマスター・データベースではありません	
25023	レプリカが登録されていません	
25031	トランザクションがアクティブで、操作が失敗しました	ユーザーの行った変更がまだコミットされていません。
25035	メッセージは使用中です	
25051	未完了のメッセージが見つかりました	
25056	自動コミットは許可されません	このストアード・プロシージャーは、自動コミットをオフにして実行する必要があります。
25079		
25093		

SYNC_REGISTER_PUBLICATION

```
CALL SYNC_REGISTER_PUBLICATION (
    replica_catalog_name,   -- WVARCHAR
    publication_name        -- WVARCHAR
)
```

EXECUTES ON: レプリカ。

SYNC_REGISTER_PUBLICATION() システム・プロシージャーは、マスター・データベースからパブリケーションを登録します。

レプリカ・カタログ名が NULL の場合は、現在のカタログが使用されます。

ユーザーの変更がコミットされていない場合は、呼び出しがエラーで失敗します。

このシステム・プロシージャーは、結果セットを返しません。

表 187. SYNC_REGISTER_PUBLICATION のエラー・コード

RC	テキスト	説明
13047	操作する特権がありません	
13110	NULL は許可されません	NULL にできるのはカタログ名のみです。それ以外のパラメーターは NULL 以外にする必要があります。
13133	この製品に有効なライセンスではありません	
21xxx	通信エラー	マスターに接続できませんでした。21xxx エラーについて詳しくは、「*solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。
25005	メッセージは既にアクティブです	
25010	パブリケーションが見つかりません	
25019	データベースがレプリカ・データベースではありません	
25020	データベースがマスター・データベースではありません	
25023	レプリカが登録されていません	
25035	メッセージは使用中です	
25056	自動コミットは許可されません	このストアード・プロシージャーは、自動コミットをオフにして実行する必要があります。
25072	既にパブリケーションに登録されています	

SYNC_UNREGISTER_PUBLICATION

```
CALL SYNC_UNREGISTER_PUBLICATION (
    replica_catalog_name,   -- WVARCHAR
    publication_name,       -- WVARCHAR
    drop_data               -- INTEGER
)
```

EXECUTES ON: レプリカ。

SYNC_UNREGISTER_PUBLICATION() システム・プロシージャーは、パブリケーションの登録を抹消します。*drop_data* フラグをゼロ以外の値に設定すると、そのパブリケーションに対するすべてのサブスクリプションが自動的にドロップされます。

レプリカ・カタログ名が NULL の場合は、現在のカタログが使用されます。

ユーザーの変更がコミットされていない場合は、呼び出しがエラーで失敗します。

このシステム・プロシージャーは、結果セットを返しません。

表 188. SYNC_UNREGISTER_PUBLICATION のエラー・コード

RC	テキスト	説明
13047	操作する特権がありません	
13110	NULL は許可されません	NULL にできるのはカタログ名のみです。それ以外のパラメーターは NULL 以外にする必要があります。
13133	この製品に有効なライセンスではありません	
21xxx	通信エラー	マスターに接続できませんでした。21xxx エラーについて詳しくは、「*solidDB 管理者ガイド*」の『エラー・コード』という表題の付録を参照してください。
25005	メッセージは既にアクティブです	
25010	パブリケーションが見つかりません	
25019	データベースがレプリカ・データベースではありません	
25020	データベースがマスター・データベースではありません	
25023	レプリカが登録されていません	
25031	トランザクションがアクティブで、操作が失敗しました	ユーザーの行った変更がまだコミットされていません。
25035	メッセージは使用中です	
25056	自動コミットは許可されません	このストアード・プロシージャーは、自動コミットをオフにして実行する必要があります。

表 188. SYNC_UNREGISTER_PUBLICATION のエラー・コード (続き)

RC	テキスト	説明
25071	パブリケーションには登録していません	

SYNC_SHOW_SUBSCRIPTIONS

```
CREATE PROCEDURE SYNC_SHOW_SUBSCRIPTIONS (
publication_name    -- WVARCHAR
)
```

EXECUTES ON: レプリカ。

パブリケーションのサブスクリプション (ストリングで表されたパブリケーション名とパラメーター) のうち、レプリカ・データベースまたはマスター・データベースでアクティブなものをアプリケーションで認識できると便利な場合がよくあります。この機能はマスター・カタログとレプリカ・カタログの両方で使用できます。レプリカ・カタログではこの関数 (SYNC_SHOW_SUBSCRIPTIONS) を使用します。マスター・カタログでは関数 SYNC_SHOW_REPLICA_SUBSCRIPTIONS を使用します。

このプロシージャー呼び出しの結果セットを以下に示します。

表 189. CREATE PROCEDURE SYNC_SHOW_SUBSCRIPTIONS の結果セット

列名	データ型	説明
SUBSCRIPTION	WVARCHAR	ストリングとしてのパブリケーション名とパラメーター。
SUBSCRIPTION_TIME	TIMESTAMP	前回のサブスクリプションの時刻。

表 190. SYNC_SHOW_SUBSCRIPTIONS のエラー・コード

RC	テキスト	説明
13047	操作する特権がありません	
13133	この製品に有効なライセンスではありません	
25009	レプリカが見つかりません	
25010	パブリケーションが見つかりません	
25019	データベースがレプリカ・データベースではありません	
25020	データベースがマスター・データベースではありません。	
25023	レプリカが登録されていません	
25071	パブリケーションには登録していません	

関連項目

『SYNC_SHOW_REPLICA_SUBSCRIPTIONS』.

SYNC_SHOW_REPLICA_SUBSCRIPTIONS

マスターでの構文:

```
CREATE PROCEDURE SYNC_SHOW_REPLICA_SUBSCRIPTIONS (
    replica_name,           -- WVARCHAR
    publication_name        -- WVARCHAR
)
```

EXECUTES ON: マスター。

パブリケーションのサブスクリプション (ストリングで表されたパブリケーション名とパラメーター) のうち、指定されたレプリカ・データベースでアクティブなものをアプリケーションで認識できると便利な場合がよくあります。この機能は、マスター・カタログとレプリカ・カタログの両方で使用できます。

パブリケーション名が NULL の場合は、すべてのパブリケーションに対するサブスクリプションがリストされます。

このプロシージャー呼び出しの結果セットを以下に示します。

表 191. SYNC_SHOW_REPLICA_SUBSCRIPTIONS の結果セット

列名	データ型	説明
REPLICA_NAME	WVARCHAR	レプリカ名。
SUBSCRIPTION	WVARCHAR	ストリングとしてのパブリケーション名とパラメーター。
SUBSCRIPTION_TIME	TIMESTAMP	前回のサブスクリプションの時刻。

表 192. SYNC_SHOW_REPLICA_SUBSCRIPTIONS のエラー・コード

RC	テキスト	説明
13047	操作する特権がありません	
13133	この製品に有効なライセンスではありません	
25009	レプリカが見つかりません	
25010	パブリケーションが見つかりません	
25019	データベースがレプリカ・データベースではありません	
25020	データベースがマスター・データベースではありません	

表 192. SYNC_SHOW_REPLICA_SUBSCRIPTIONS のエラー・コード (続き)

RC	テキスト	説明
25023	レプリカが登録されていません	
25071	パブリケーションには登録していません	

関連項目

417 ページの『SYNC_SHOW_SUBSCRIPTIONS』.

SYNC_DELETE_MESSAGES

```
CALL SYNC_DELETE_MESSAGES (
    replica_catalog_name,   -- WVARCHAR
)
```

EXECUTES ON: レプリカ。

レプリカ・カタログ名が NULL の場合は、現在のカタログが使用されます。

レプリカ・アプリケーションが大量のメッセージを作成する一方でエラーを適切に検査/処理しない場合は、大量のメッセージがハングする可能性があります。場合によっては、リカバリーのための適切な処置として、マスター側とレプリカ側の両方でメッセージの状態に関係なくすべてのメッセージを削除することがあります。このプロシージャーは、レプリカ・データベース内のメッセージを削除します。

このプロシージャーは結果セットを返しません。

表 193. SYNC_DELETE_MESSAGES のエラー・コード

RC	テキスト	説明
13047	操作する特権がありません	
13133	この製品に有効なライセンスではありません	
25005	メッセージは既にアクティブです	
25009	レプリカが見つかりません	
25019	データベースがレプリカ・データベースではありません	
25020	データベースがマスター・データベースではありません	
25035	メッセージは使用中です	

関連項目

420 ページの『SYNC_DELETE_REPLICA_MESSAGES』.

SYNC_DELETE_REPLICA_MESSAGES

```
CALL SYNC_DELETE_REPLICA_MESSAGES(
    master_catalog_name    -- WVARCHAR,
    replica_name           -- WVARCHAR
)
```

EXECUTES ON: マスター。

レプリカ・アプリケーションが大量のメッセージを作成する一方でエラーを適切に検査/処理しない場合は、大量のメッセージがハングする可能性があります。場合によっては、リカバリーのための適切な処置として、マスター側とレプリカ側の両方でメッセージの状態に関係なくすべてのメッセージを削除することがあります。このプロシージャーは、マスター・データベースにある指定されたレプリカのメッセージを削除します。master_catalog_name パラメーターでは、指定したレプリカのメッセージが検索されるマスター・データベース内のカタログを指定します。master_catalog_name を NULL に設定すると、現在のカタログが使用されます。

このプロシージャーは結果セットを返しません。

表 194. SYNC_DELETE_REPLICA_MESSAGES のエラー・コード

RC	テキスト	説明
13047	操作する特権がありません	
13133	この製品に有効なライセンスではありません	
25005	メッセージは既にアクティブです	
25009	レプリカが見つかりません	
25019	データベースがレプリカ・データベースではありません	
25020	データベースがマスター・データベースではありません。	
25035	メッセージは使用中です	

関連項目

419 ページの『SYNC_DELETE_MESSAGES』.

各種ストアード・プロシージャー

SYS_GETBACKGROUNDJOB_INFO

```
CREATE PROCEDURE SYS_GETBACKGROUNDJOB_INFO(
    jobid INTEGER)
RETURNS(
    ID INTEGER,
```

```
STMT WVARCHAR,
USER_ID INTEGER,
ERROR_CODE INTEGER,
ERROR_TEXT INTEGER)
```

ユーザーは、SQL SELECT ステートメントを使用するか、システム・ストアード・プロシージャー SYS_GETBACKGROUNDJOB_INFO を呼び出すことで、表 SYS_BACKGROUNDJOB_INFO から情報をリトリーブできます。プロシージャー SYS_GETBACKGROUNDJOB_INFO は、指定された jobid と一致する行を返します。この jobid は、実行された START AFTER COMMIT ステートメントのジョブ ID です (このジョブ ID は、START AFTER COMMIT ステートメントが実行されたときにサーバーから返されます)。

付録 G. システム・イベント

この章ではシステム・イベントについて説明します。システム・イベントは solidDB に用意されており、これを使用することで一定のアクションが発生したときにプログラムに通知することができます。これらのイベントを使用して、マスター・データベースとレプリカ・データベース間の同期などのアクティビティーの進行をモニターできます。

システム・イベントは、他のイベントとほぼ同じルールに従います。イベントに関する全般的な情報については、以下を参照してください。

- 201 ページの『CREATE EVENT』
- 201 ページの『CREATE EVENT』では、イベントを通知する方法およびイベントを待機する方法について説明します。
- 27 ページの『3 章 ストアード・プロシージャー、イベント、トリガー、およびシーケンス』では、イベントについて広範囲にわたって説明します。

システム・イベントは事前定義されるため、ユーザーが作成することはありません。さらに、システム・イベントを通知することはできません。システム・イベントは待機することのみ可能です。

システム・イベントの多くには (すべてではありません) 5 つの同じパラメーターが設定されています。

- ename: イベント名。
- postsrvtime: サーバーがイベントを通知した時刻。
- uid: ユーザー ID (該当する場合)。
- numdatainfo: 各種数値データ。正確な意味はイベントによって異なります。例えば、イベント SYS_EVENT_BACKUP は、バックアップの開始時と完了時に通知されます。numdatainfo パラメーターの値は、どちらのケースに該当するか、つまりバックアップが開始したか完了したかを示します。数値データがない場合は、このパラメーターが NULL になることがあります。
- textdata: 各種テキスト・データ。正確な意味はイベントによって異なります。数値データがない場合は、このパラメーターが NULL になることがあります。

この付録では以下の表を掲載しています。

1. 各種イベント
2. SYS_EVENT_ERROR イベントが通知される原因となるエラー
3. SYS_EVENT_MESSAGES イベントが通知される原因となる条件または警告

注:

- HotStandby に関連するイベントについては、「*IBM solidDB 高可用性ユーザー・ガイド*」を参照してください。
- 拡張レプリケーションに関連するイベントについては、「*IBM solidDB 拡張レプリケーション・ユーザー・ガイド*」を参照してください。

各種イベント

以下のイベントは、ほとんどの場合、サーバーの内部スケジューリングおよび「ハウスキーピング」に関連しています。例えば、バックアップ、チェックポイント、およびマージに関連するイベントがあります。ユーザーはこれらのイベントを通知しませんが、多くの場合、ユーザーが間接的にイベントの原因になっている可能性があります。例えば、バックアップを要求したときや、「保守モード」をオンにしたときなどです。必要あれば、これらのイベントをモニターできます。

表 195. 各種イベント

イベント名	イベントの説明	パラメーター
SYS_EVENT_BACKUP	システムがバックアップ操作を開始したか完了しました。「状態」パラメーター (NUMDATAINFO) は、以下のことを示します。 0: バックアップが完了しました。 1: バックアップが開始されました。 サーバーはバックアップを開始または完了した時点で、2 番目のイベント (SYS_EVENT_MESSAGES) も通知することに注意してください。	ENAME WVARCHAR、 POSTSRVTIME TIMESTAMP、 UID INTEGER、 NUMDATAINFO INTEGER、 TEXTDATA WVARCHAR
SYS_EVENT_BACKUPREQ	バックアップ操作が要求されました (ただし、まだ開始されていません)。 ユーザー・アプリケーションのコールバック関数がゼロ以外を返した場合、バックアップは実行されません。 このイベントは、ユーザーが共有メモリー・アクセスまたはリンク・ライブラリー・アクセスを使用している場合のみ、ユーザーによる取得が可能です。 パラメーターはどれも使用されません。	ENAME WVARCHAR、 POSTSRVTIME TIMESTAMP、 UID INTEGER、 NUMDATAINFO INTEGER、 TEXTDATA WVARCHAR
SYS_EVENT_CHECKPOINT	システムがチェックポイント処理を開始したか完了しました。 システムがチェックポイントを開始した場合、「状態」パラメーター (NUMDATAINFO) は 1 で、メッセージ (TEXTDATA) パラメーターは「started」です。 システムがチェックポイントを完了した場合、「状態」パラメーター (NUMDATAINFO) は 0 で、メッセージ (TEXTDATA) パラメーターは「completed」です。	ENAME WVARCHAR、 POSTSRVTIME TIMESTAMP、 UID INTEGER、 NUMDATAINFO INTEGER、 TEXTDATA WVARCHAR

表 195. 各種イベント (続き)

イベント名	イベントの説明	パラメーター
SYS_EVENT_CHECKPOINTREQ	チェックポイント処理が要求されました (ただし、まだ開始されていません)。一般に、チェックポイントは特定の数のログへの書き込みが完了するたびに実行されます。 ユーザー・アプリケーションのコールバック関数がゼロ以外を返した場合、マージは実行されません。 このイベントは、ユーザーが共有メモリー・アクセスまたはリンク・ライブラリー・アクセスを使用している場合のみ、ユーザーによる取得が可能です。 パラメーターはどれも使用されません。	ENAME WVARCHAR、 POSTSRVTIME TIMESTAMP、 UID INTEGER、 NUMDATAINFO INTEGER、 TEXTDATA WVARCHAR
SYS_EVENT_ERROR	何らかのタイプのサーバー・エラーが発生しました。メッセージ・パラメーター (TEXTDATA) にエラー・テキストが入っています。このイベントを通知する原因になる可能性があるサーバー・エラーのリストについては、 431 ページの『SYS_EVENT_ERROR の原因となるエラー』を参照してください。	ENAME WVARCHAR、 POSTSRVTIME TIMESTAMP、 UID INTEGER、 NUMDATAINFO INTEGER、 TEXTDATA WVARCHAR
SYS_EVENT_IDLE	システムはアイドル状態です。(一部のタスクは「アイドル」の優先順位を持ち、システムが他のタスクを実行していないときにのみ実行されることに注意してください。非常に低い優先順位のタスクが「アイドル」状態のシステムで実行されている可能性があるので、システムは必ずしも、何もしていないという意味での真のアイドル状態にあるとは限りません。) このイベントは、ユーザーが共有メモリー・アクセスまたはリンク・ライブラリー・アクセスを使用している場合のみ、ユーザーによる取得が可能です。 パラメーターはどれも使用されません。	ENAME WVARCHAR、 POSTSRVTIME TIMESTAMP、 UID INTEGER、 NUMDATAINFO INTEGER、 TEXTDATA WVARCHAR

表 195. 各種イベント (続き)

イベント名	イベントの説明	パラメーター
SYS_EVENT_IMDB_MEMORY	システムが、インメモリー・データベース・メモリー限度に関連したイベントを検出しました。 NUMDATAINFO パラメーターは、現在のメモリー割り振りをキロバイト単位で示します。 TEXTDATA パラメーターは、以下のいずれかの値を持つことができます。 • IMDB_LIMIT_ABOVE: 使用可能な仮想メモリーの量は、**ImdbMemoryLimit** パラメーターを使用して指定された限度を超えています。 • IMDB_LIMIT_BELOW: 使用可能な仮想メモリーの量は、**ImdbMemoryLimit** パラメーターを使用して指定された限度を下回っています。 • IMDB_LOW_LEVEL_ABOVE: 使用可能な仮想メモリーの量は、**ImdbMemoryLowPercentage** パラメーターを使用して指定された限度を超えています。 • IMDB_LOW_LEVEL_BELOW: 使用可能な仮想メモリーの量は、**ImdbMemoryLowPercentage** パラメーターを使用して指定された限度を下回っています。 • IMDB_WARNING_LEVEL_ABOVE: 使用可能な仮想メモリーの量は、**ImdbMemoryLowPercentage** パラメーターを使用して指定された限度を超えています。 • IMDB_WARNING_LEVEL_BELOW: 使用可能な仮想メモリーの量は、**ImdbMemoryLowPercentage** パラメーターを使用して指定された限度を下回っています。	ENAME WVARCHAR、 POSTSRVTIME TIMESTAMP、 UID INTEGER、 NUMDATAINFO INTEGER、 TEXTDATA WVARCHAR
SYS_EVENT_ILL_LOGIN	正しくないログインの試みがありました。ユーザー名 (TEXTDATA) およびユーザー ID (NUMDATAINFO) は、ログインしようとしたユーザーを示しています。	ENAME WVARCHAR、 POSTSRVTIME TIMESTAMP、 UID INTEGER、 NUMDATAINFO INTEGER、 TEXTDATA WVARCHAR

表 195. 各種イベント (続き)

イベント名	イベントの説明	パラメーター
SYNC_MAINTENANCEMODE_BEGIN	同期モードが NORMAL から MAINTENANCE に変更されると、サーバーはこのシステム・イベントを送信します。 node_name は、保守モードが開始されたノードの名前です。(単一の solidDB サーバーが複数の「ノード」(カタログ) を持つことができる点に留意してください。) 同期モードについて詳しくは、 318 ページの『SET SYNC MODE』を参照してください。	node_name WVARCHAR.
SYNC_MAINTENANCEMODE_END	同期モードが MAINTENANCE から NORMAL に変更されると、サーバーはこのシステム・イベントを送信します。 node_name は、保守モードが開始されたノードの名前です。(単一の solidDB サーバーが複数の「ノード」(カタログ) を持つことができる点に留意してください。) 同期モードについて詳しくは、 318 ページの『SET SYNC MODE』を参照してください。	node_name WVARCHAR
SYS_EVENT_MERGE	「マージ」操作 (Bonsai ツリーから主ストレージ・ツリーへのデータのマージ) に関連したイベントが発生しました。パラメーター STATE (NUMDATAINFO) に詳細が示されます。 0: マージの停止 1: マージの開始 2: マージが進行中 3: マージが加速	ENAME WVARCHAR、 POSTSRVTIME TIMESTAMP、 UID INTEGER、 NUMDATAINFO INTEGER、 TEXTDATA WVARCHAR
SYS_EVENT_MERGEREQ	マージ操作が要求されました (ただし、まだ開始されていません)。 ユーザー・アプリケーションのコールバック関数がゼロ以外を返した場合、マージは実行されません。 このイベントは、ユーザーが共有メモリー・アクセスまたはリンク・ライブラリー・アクセスを使用している場合のみ、ユーザーによる取得が可能です。 パラメーターはどれも使用されません。	ENAME WVARCHAR、 POSTSRVTIME TIMESTAMP、 UID INTEGER、 NUMDATAINFO INTEGER、 TEXTDATA WVARCHAR

表 195. 各種イベント (続き)

イベント名	イベントの説明	パラメーター
SYS_EVENT_MESSAGES	このイベントは、サーバーが solerror.out または solmsg.out にログとして記録するメッセージ (エラー・メッセージまたは警告メッセージ) を持っているときに通知されます。その場合、TEXTDATA にメッセージ・テキストが、NUMDATAINFO にコードがそれぞれ入っています。書き込まれるメッセージがエラーである場合は、SYS_EVENT_ERROR と SYS_EVENT_MESSAGES の*両方* が通知されます。メッセージが警告にすぎない場合は、SYS_EVENT_MESSAGES だけが通知されます。SYS_EVENT_MESSAGES の原因になる可能性のある警告のリストについては、 432 ページの『SYS_EVENT_MESSAGES の原因となる状態または警告』を参照してください。	ENAME WVARCHAR、 POSTSRVTIME TIMESTAMP、 UID INTEGER、 NUMDATAINFO INTEGER、 MESSAGE WVARCHAR
SYS_EVENT_NOTIFY	ADMIN COMMAND 'notify' で送信されたイベント。	ENAME WVARCHAR、 POSTSRVTIME TIMESTAMP、 UID INTEGER、 NUMDATAINFO INTEGER、 TEXTDATA WVARCHAR
SYS_EVENT_PARAMETER	このイベントは、構成パラメーターが以下のコマンドで変更された場合に通知されます。 ADMIN COMMAND 'parameter...'; パラメーター MESSAGE (TEXTDATA) には、セクション名 (例えば、SRV) およびパラメーター名が入っています。	ENAME WVARCHAR、 POSTSRVTIME TIMESTAMP、 UID INTEGER、 NUMDATAINFO INTEGER、 TEXTDATA WVARCHAR

表 195. 各種イベント (続き)

イベント名	イベントの説明	パラメーター
SYS_EVENT_PROCESS_MEMORY	システムが、プロセス・サイズ・メモリー限度に関連したイベントを検出しました。 NUMDATAINFO パラメーターは、現在のメモリー割り振りをキロバイト単位で示します。 TEXTDATA パラメーターは、以下のいずれかの値を持つことができます。 • PROCESS_LIMIT_ABOVE: 使用可能な仮想メモリーの量は、**ProcessMemoryLimit** パラメーターを使用して指定された限度を超えています。 • PROCESS_LIMIT_BELOW: 使用可能な仮想メモリーの量は、**ProcessMemoryLimit** パラメーターを使用して指定された限度を下回っています。 • PROCESS_LOW_LEVEL_ABOVE: 使用可能な仮想メモリーの量は、**ProcessMemoryLowPercentage** パラメーターを使用して指定された限度を超えています。 • PROCESS_LOW_LEVEL_BELOW: 使用可能な仮想メモリーの量は、**ProcessMemoryLowPercentage** パラメーターを使用して指定された限度を下回っています。 • PROCESS_WARNING_LEVEL_ABOVE: 使用可能な仮想メモリーの量は、**ProcessMemoryWarningPercentage** パラメーターを使用して指定された限度を超えています。 • PROCESS_WARNING_LEVEL_BELOW: 使用可能な仮想メモリーの量は、**ProcessMemoryWarningPercentage** パラメーターを使用して指定された限度を下回っています。	ENAME WVARCHAR、 POSTSRVTIME TIMESTAMP、 UID INTEGER、 NUMDATAINFO INTEGER、 TEXTDATA WVARCHAR
SYS_EVENT_ROWS2MERGE	このイベントは、Bonsai ツリーから主ストレージ・ツリーにマージする必要がある行が存在することを示しています。行パラメーター (NUMDATAINFO) は、Bonsai ツリー内のマージされていない行の数を示しています。	ENAME WVARCHAR、 POSTSRVTIME TIMESTAMP、 UID INTEGER、 NUMDATAINFO INTEGER、 TEXTDATA WVARCHAR

表 195. 各種イベント (続き)

イベント名	イベントの説明	パラメーター
SYS_EVENT_SACFAILED	このイベントは、START AFTER COMMIT (SAC) が失敗したときに通知されます。アプリケーションはこのイベントを待ち、(NUMDATAINFO フィールドにある) ジョブ ID を使用して、システム表 SYS_BACKGROUNDJOB_INFO からエラー・メッセージをリトリーブすることができます。(NUMDATAINFO 内のジョブ ID は、START AFTER COMMIT ステートメントが実行されたときに返されたジョブ ID に一致します。)	ENAME WVARCHAR、 POSTSRVTIME TIMESTAMP、 UID INTEGER、 NUMDATAINFO INTEGER、 TEXTDATA WVARCHAR
SYS_EVENT_SHUTDOWNREQ	シャットダウン要求が受信されました。ユーザー・アプリケーションのコールバック関数がゼロ以外を返した場合、シャットダウンは実行されません。 このイベントは、ユーザーが共有メモリー・アクセスまたはリンク・ライブラリー・アクセスを使用している場合のみ、ユーザーによる取得が可能です。 パラメーターはどれも使用されません。	ENAME WVARCHAR、 POSTSRVTIME TIMESTAMP、 UID INTEGER、 NUMDATAINFO INTEGER、 TEXTDATA WVARCHAR
SYS_EVENT_STATE_MONITOR	このイベントは、モニター設定が変更されたときに通知されます。 状態 (NUMDATAINFO) は、以下のいずれかです。 0: モニターはオフ。 1: モニターはオン。 UID は、モニターがオンまたはオフにされたユーザーのユーザー ID です。	ENAME WVARCHAR、 POSTSRVTIME TIMESTAMP、 UID INTEGER、 NUMDATAINFO INTEGER、 TEXTDATA WVARCHAR
SYS_EVENT_STATE_OPEN	このイベントは、データベースの「状態」が変更されたときに通知されます。パラメーター STATE (NUMDATAINFO) は、新しい状態を示します。 0: クローズ: 新規接続は許されません。 1: オープン: 新規接続が許されます。	ENAME WVARCHAR、 POSTSRVTIME TIMESTAMP、 UID INTEGER、 NUMDATAINFO INTEGER、 TEXTDATA WVARCHAR
SYS_EVENT_STATE_SHUTDOWN	このイベントは、サーバーのシャットダウンが開始されたときに通知されます。NUMDATAINFO パラメーターおよび TEXTDATA パラメーターは有益な情報を持たないことに注意してください。	ENAME WVARCHAR、 POSTSRVTIME TIMESTAMP、 UID INTEGER、 NUMDATAINFO INTEGER、 TEXTDATA WVARCHAR

表 195. 各種イベント (続き)

イベント名	イベントの説明	パラメーター
SYS_EVENT_STATE_TRACE	サーバー・トレースをオンまたはオフにするには、以下を使用します。 ADMIN COMMAND 'trace'; パラメーター STATE (NUMDATAINFO) は、新しいトレース状態を示します。 0: トレースはオフ。 1: トレースはオン。	ENAME WVARCHAR、 POSTSRVTIME TIMESTAMP、 UID INTEGER、 NUMDATAINFO INTEGER、 TEXTDATA WVARCHAR
SYS_EVENT_TMCMD	このイベントは、「AT」コマンド (つまり、時刻指定コマンド) が実行されたときに通知されます。メッセージ・パラメーター (TEXTDATA) にコマンドが入っています。	ENAME WVARCHAR、 POSTSRVTIME TIMESTAMP、 UID INTEGER、 NUMDATAINFO INTEGER、 TEXTDATA WVARCHAR
SYS_EVENT_TRX_TIMEOUT	このイベントは現在使用されていません。	ENAME WVARCHAR、 POSTSRVTIME TIMESTAMP、 UID INTEGER、 NUMDATAINFO INTEGER、 TEXTDATA WVARCHAR
SYS_EVENT_USERS	パラメーター REASON (NUMDATAINFO) にイベントの理由が入っています。 0: ユーザーは接続されました。 1: ユーザーの接続は切断されました。 2: ユーザーの接続は異常切断されました。 4: ユーザーはタイムアウトのために接続が切断されました。	ENAME WVARCHAR、 POSTSRVTIME TIMESTAMP、 UID INTEGER、 NUMDATAINFO INTEGER、 TEXTDATA WVARCHAR

SYS_EVENT_ERROR の原因となるエラー

サーバーでイベント SYS_EVENT_ERROR が通知される原因となるエラーを以下の表に示します。

「エラー・コード」列の番号は、「*solidDB 管理者ガイド*」の付録『エラー・コード』に記載されているエラー・コード番号と対応します。この値は、NUMDATAINFO イベント・パラメーターに渡されます。

表 196. SYS_EVENT_ERROR の原因となるエラー

エラー・コード	エラーの説明
30104	シャットダウンが中止されました (ユーザー・コールバックによる拒否)。
30208	マージが開始されませんでした (ユーザー・コールバックによる拒否)。
30284	チェックポイントが開始されませんでした (ユーザー・コールバックによる拒否)。
30302	バックアップの開始に失敗しました。シャットダウンは進行中です。
30302	バックアップの開始に失敗しました。バックアップは既にアクティブです。
30303	バックアップが中止されました。
30304	バックアップが失敗しました。<エラーの説明>
30305	バックアップが開始されませんでした (ユーザー・コールバックによる拒否)。
30306	バックアップが開始されませんでした。ディスクレス・サーバーではバックアップがサポートされません。
30307	バックアップが開始されませんでした。索引検査に失敗しました。ファイル `ssdebug.log` にエラーが書き込まれました。
30360	AT コマンドが失敗しました。<理由>
30403	ログ・ファイルへの書き込みが失敗しました。
30454	構成ファイル <ファイル名>の保存に失敗しました。
30573	ネットワーク・バックアップが失敗しました。<理由>
30640	<サーバー RPC のエラー・メッセージ>

SYS_EVENT_MESSAGES の原因となる状態または警告

以下の表は、サーバーがイベント SYS_EVENT_MESSAGES を通知する原因になる場合がある警告メッセージを示しています。

表 197. SYS_EVENT_MESSAGES の原因となる警告

エラー・コード	エラーの説明
30010	ユーザー <username> が接続に失敗しました。バージョンの不一致です。クライアントのバージョンは <version> で、サーバーのバージョンは <version> です。

表 197. SYS_EVENT_MESSAGES の原因となる警告 (続き)

エラー・コード	エラーの説明
30011	ユーザー <username> が接続に失敗しました。照合バージョンの不一致です。
30012	ユーザー <username> が接続に失敗しました。接続しているクライアントの数が多すぎます。
30020	サーバーが致命的な状態にあり、新規接続は許可されません。
30020	ユーザー <username> が接続に失敗しました。データベース文字セットが utf8 ですが、これはクライアントでサポートされていません。
30282	シャットダウンが進行中のため、チェックポイントの作成は開始されませんでした。
30283	チェックポイントの作成は、使用不可に設定されているため、開始されませんでした。
30300	バックアップは正常に完了しました。 サーバーはバックアップを開始または完了した時点で、2 番目のイベント (SYS_EVENT_BACKUP) も通知することに注意してください。
30301	バックアップが <directory path> に対して開始されました。 サーバーはバックアップを開始または完了した時点で、2 番目のイベント (SYS_EVENT_BACKUP) も通知することに注意してください。
30359	サーバーは at コマンドの実行時に、時刻の不整合を検出しました。システム時刻を変更した場合は、サーバーを再始動してください。
30361	正しくない at コマンド <command> は無視されました。
30362	正しくない即時の at コマンド <command> は無視されました。
30405	メッセージ・ログ・ファイル「file name」を開くことができません。
30800	要求された <number> 個のメモリー・ブロックを外部ソーター用に予約できません。 <number> 個のメモリー・ブロックだけが使用可能でした。 SQL: <sql statement>
30801	要求された <number> 個のメモリー・ブロックを外部ソーター用に予約できません。 <number> 個のメモリー・ブロックだけが使用可能でした。

索引

日本語, 数字, 英字, 特殊文字の順に配列されています。なお, 濁音と半濁音は清音と同等に扱われています。

［ア行］

アクセス権限
　登録ユーザー　323
　パブリケーション　260, 304
アクセス権限 (リモート・ストアード・プロシージャー)　44
アプリケーション・パフォーマンスの評価　159
アンダーライン　346
イベント
　コード例　97
　使用　97
　待ち　164
　ADMIN EVENT コマンド　186
インクリメンタル・パブリケーション
　指定　191
インテリジェント・トランザクション
　パラメーター掲示板　307
　保存されているプロパティーの使用　307
インメモリー表およびインメモリー索引に使用されるメモリー量　180
エスケープ文字　347
エスケープ・シーケンス　33
エラー
　致命的エラー、同期エラー　293
　DBMS　281, 291
　SYS_EVENT_ERROR　431
エラー処理
　ストアード・プロシージャー　50
応答メッセージ
　タイムアウトの設定　286
　マスター・データベースからの要求　292
オプティマイザー・ヒント
　使用　165
オプティミスティック・ロック方式　130, 133

［カ行］

カーソル
　ストアード・プロシージャー　57
　　実行　48
　　準備　47
　　処理　47
　　閉じる　49
　　ドロップ　49
　　フェッチ　48
　ストアード・プロシージャーでのデフォルト管理　56
カーソル *(続き)*
　パラメーター・マーカー　52
外部キー　118
　制約　227
下線　346
カタログ
　削除　126
　作成　126, 199
　説明　124
関数
　各種　344
　スカラー　32
　ストアード・プロシージャーでのスタックの表示　57
　トリガー　81
　AVG　338
　COUNT　338
　MAX　338
　MIN　338
　SET_PARAM()　258
　SQL 関数　297
　SUM　338
疑似列　345
行　1, 9
行値コンストラクター　22
共有ロック　133, 139
クライアント/サーバー・アーキテクチャー
　説明　4
　マルチユーザー機能　5
クラスタリング　116
クラスタリング・キー　373
掲示板パラメーター　257
降順　116
更新ロック　139
構成
　同期　321
候補キー　118
コミット・ブロック
　リフレッシュ・サイズの定義　286, 292

［サ行］

再実行
　メッセージ　284
最適化
　バッチ挿入および更新　164
作業のコミット
　表の変更後　114
　ユーザーおよびロールの変更後　111
索引　160, 161
　外部キー　118
　管理　115

索引 *(続き)*
- 削除 115
- 作成 115
- 主キー索引 116
- 副次キー索引 116
- 複数列 163
- ユニーク索引の作成 115
- 連結 163

索引付け
- 列 163

索引を使用した照会パフォーマンスの向上 161
削除
- 失敗したメッセージ 281
- メッセージ 279

作成
- パブリケーション 216

サブスクリプション
- インポート 267
- エクスポート 253
- コミット・ブロックの定義 292
- ドロップ 247

参照アクション
- Cascade 121
- No action 121
- Restrict 121
- Set default 121
- Set null 121

参照先の表 118
参照整合性 118, 227
- 制約 121
- 制約の動的な管理 122
- トランジエント表 227

参照元の表 118
シーケンス
- 使用 96

式 337
- ストアード・プロシージャー 34

システム関数 343
システム表 363
- アクセス権限の付与 112
- 説明 112
- トリガー 81
- 表示 112

システム・パラメーター 258
システム・ビュー 400
実行
- 失敗したメッセージ 290
- メッセージ 284

自動コミット 205
集合論 11
終了
- メッセージ 283

主キー 114, 118
- 索引 116

昇順 116
数字関数 340
据え置きプロシージャー呼び出し 83
スカラー関数 32
- 説明 32, 338

スキーマ
- 削除 127
- 作成 126
- 説明 125, 221

ストアード・プロシージャー
- 位置付け更新および位置付け削除 54
- イベントの使用 97
- エラー処理 50
- カーソル 57
- カーソル内のパラメーター・マーカー 52
- 自動コミット 205
- 終了 42
- 出力パラメーター 28
- 説明 27
- デフォルト値 28
- デフォルト・カーソル管理 56
- 特権 57
- トランザクション 55
- トリガー 65
- トレース機能 154
- 入出力パラメーター 28
- 入力パラメーター 28
- パラメーターの使用 28
- プロシージャーのネスト 54
- プロシージャー本体 32
- プロシージャー・スタックの表示 57
- 変数への値の代入 32
- リモート 42
- ループ 39
- ローカル変数の宣言 31
- CREATE PROCEDURE ステートメント 27
- SQL の使用 46, 57

ストアード・プロシージャーおよびトリガーのトレース機能 154
ストアード・プロシージャーでの SQL の使用 46
ストリング
- 長さゼロ 41

ストリング関数 339
スペース 180
制御構造
- ストアード・プロシージャー内 36

制約
- 外部キー 227

接続ストリング
- マスター名への変更 317

全表スキャン 162
送信
- メッセージ 286

［タ行］

大規模なレプリカ
- 作成 253

タイムアウト
　応答メッセージについての設定　286
単純 SQL 最適化　160
チェックポイント
　SYS_EVENT_CHECKPOINT　424
　'makecp' コマンド　180
致命的エラー
　リカバリー　293
チューニング
　SQL ステートメント　159
　SQL ステートメントとアプリケーション　159
重複挿入
　修正　291
データ
　ストアード・プロシージャーでの戻り　42
　ファイルからのインポート　267
　ファイルへのエクスポート　253
データ型　11, 345
　SQL　107
データ管理
　solidDB SQL の使用　129
データベース
　行　1, 9
　表　1, 9
　フリー・スペース　180
　リレーショナル　1
　列　1, 9
データベース作成時間　180
データベース内のフリー・スペース　180
データベースの登録
　登録ユーザー　323
データベース・オブジェクト
　管理　123
伝搬
　強制終了したメッセージ　293
テンポラリー表　226
同期
　メッセージ　297
　履歴表　191
登録
　レプリカ・データベース　275
　レプリカ・ノード名の設定　320
登録抹消
　レプリカ・データベース　275
特権
　管理　108
　ストアード・プロシージャー　57
トランザクション　24
　インテリジェント・トランザクション　275, 305
　ストアード・プロシージャー　55
　説明　5, 141
　定義　129
　デフォルト・プロパティーの保存　307
　伝搬　275
　伝搬優先順位の設定　307
　伝搬用デフォルト・プロパティーの設定　307
トランザクション *(続き)*
　トランザクション・ログ　5
　トリガーの使用　68
　プロパティーの割り当て　307
　保存　305
　読み取り専用　129
　読み取り/書き込み　129
　COMMIT WORK　5
　ROLLBACK　5
トランザクション掲示板　257
トランザクション持続性レベル
　設定　144
　選択　144
　パフォーマンスの向上　144
トランザクションの伝搬　275
　デフォルト・プロパティーの設定　307
　優先順位の設定　307
　SAVE コマンド　305
トランザクション分離レベル　142
トランザクション・ログ　5
トランジエント表　226
トリガー
　エラー処理　74
　キャッシュの設定　82
　コード例　76
　コメントおよび制限事項　64, 235
　再帰的トリガー　75
　作成　59
　仕組み　58
　使用　58
　情報の入手　81
　属性の変更　80
　デフォルト列または派生列の設定　65
　特権およびセキュリティー　75
　トランザクション　68
　トレース機能　154
　ドロップ　80
　ネストしたトリガー　75
　ネストの最大数の設定　82
　パラメーターおよび変数の使用　65
　パラメーター設定　82
　プロシージャー　65
　分析とデバッグのための関数　81
ドロップ
　サブスクリプション　247
　パブリケーション　243, 244
　ブックマーク　224, 249
　マスター・データベース　242
　レプリカ・データベース　245

［ナ行］

長さゼロのストリング　41
名前　180
日時関数　342
日時リテラル　345

ヌル
　処理　40
ノード
　設定　320

［ハ行］

排他ロック　133, 139
バイナリー・データ　172
パスワード
　入力　109, 110
　変更　110
バックアップ
　SYS_EVENT_BACKUP　424
バッチ挿入および更新
　最適化　164
パフォーマンス　159
　監視　147
　索引　161
　索引の使用による向上　161
　単一表 SQL 照会　160
　問題の診断　166
パフォーマンス低下の診断
　解決策　166
　症状　166
　診断　166
パブリケーション
　アクセス権限の取り消し　304
　アクセス権限の付与　260
　作成　216
　ドロップ　243, 244
　リフレッシュ　275
パラメーター
　永続的なデータベース・レベルの定義　321
　掲示板からのリトリーブ　257
　更新可能　258
　削除　321
　データベース・レベル　258
　トリガーでの使用　65
　パラメーター掲示板へ入れる　297
　読み取り専用　258
　EnableHints　165
　GET_PARAM()　257
　get_param()　258
　MaxStartStatements　156
　PUT_PARAM()　297
　put_param()　258
　SimpleSQLOpt　160
パラメーター掲示板　257
　インテリジェント・トランザクション　307
　説明　297
　データベース・レベル・パラメーターの定義　321
パラメーター・モード　205
　出力パラメーター　206
　入出力パラメーター　206
　入力パラメーター　206
比較演算子
　説明　34
表　1, 9
　管理　111
　削除　113
　作成　113
　別名　14
　変更後の作業のコミット　114
　列の削除　114
　列の追加　114
表ロック　139, 143
副次キー
　索引　116
複数列索引　163
ブックマーク
　ドロップ　224, 249
プッシュ同期　83
　例　92
プル同期通知　83
　例　92
プロシージャー
　ストアード・プロシージャー　28
プロパティー
　デフォルトとして保存　307
　デフォルトのトランザクション伝搬基準の保存　307
　割り当て　307
並行性　132
並行性 (ロック方式) モード
　オプティミスティックまたはペシミスティック　137
並行性制御
　オプティミスティック　133
　混合　130
　設定　129, 130
　ペシミスティック　130, 133
　モード
　　表示　378
　　ペシミスティックおよびオプティミスティック　133
　　MAINMEMORY　378
　　MAINMEMORY PESSIMISTIC　378
　　OPTIMISTIC　378
　　PESSIMISTIC　378
　目的　132
　ロック方式　131
ペシミスティック・ロック方式　130, 133
変数
　ストアード・プロシージャーでの代入　32
　トリガーでの使用　65
　SQLERRNUM　50
　SQLERROR　50
　SQLERROR OF cursorname　51
　SQLERRSTR　50
　SQLROWCOUNT　50
　SQLSUCCESS　50
保守モード　318
保存
　メッセージ　283

[マ行]

マスター・データベース　275
　応答メッセージの要求　292
　トランザクションの伝搬　275
　ドロップ　242
　ネットワーク名の変更　317
　ノード名の設定　320
　パブリケーションへのアクセス権限の取り消し　304
　パブリケーションへのアクセス権限の付与　260
　パラメーター値をリトリーブ　258
　パラメーターの設定　297, 321
　プロパティー　307
　ユーザー情報　275
マスター・ユーザー
　リストのダウンロード　275
メタデータ
　エクスポート　253
メッセージ
　エラー・メッセージ、失敗したメッセージ、応答メッセージ　281, 290
　開始　278
　再実行　284
　削除　279
　実行　284
　終了　283
　送信　286
　保存　283
　マスター・データベースからの応答の要求　292
文字データ
　型　169, 170

[ヤ行]

ユーザー
　削除　110
　作成　109
ユーザーおよびロール
　変更後の作業のコミット　111
ユーザー特権　108
　管理者特権の付与　111
　取り消し　111
　付与　110
ユーザーのリスト　185, 186
ユーザー名
　予約名　109
ユーザー・ロール　109
　管理者　109, 111
　削除　110
　作成　110
　システム・コンソール・ロール　109
　特権の取り消し　111
　特権の付与　110, 111
　パスワードの変更　110
　ユーザーのロールの取り消し　111
　ユーザーへのロールの付与　111
ユーザー・ロール *(続き)*
　予約されたロール名　109
ユニーク制約　114

[ラ行]

リカバリー
　トランザクション・ロギング　5
　DBMS レベル・エラー　281, 291
リフレッシュ
　パブリケーション　275
　マスター・データベース内の障害の処理　293
　レプリカ・データベース内の障害の処理　293
リモート・ストアード・プロシージャー　42
リレーショナル・データベース　1
履歴表　191
ループ
　ストアード・プロシージャー内　39
列　1, 9
　表からの削除　114
　表への追加　114
レプリカ・データベース
　登録　275, 320, 323
　登録抹消　275
　トランザクションの保存　305
　ドロップ　245
　パブリケーションからのリフレッシュ　275
　パラメーター値をリトリーブ　258
　パラメーターの設定　297, 321
　プロパティー　307
　マスター・データベースからの応答メッセージの要求　292
　メッセージの削除　279
レプリカ・プロパティー名　83
連結索引　163
ロール
　PUBLIC　109
　SYS_ADMIN_ROLE　109
　SYS_CONSOLE_ROLE　109
　SYS_SYNC_ADMIN_ROLE　109
　SYS_SYNC_REGISTER_ROLE　109
　_SYSTEM　109
ロック
　共有　133, 139
　更新　139
　排他　133, 139
　モード
　　EXCLUSIVE　139
　　SHARED　139
　　UPDATE　139
　EXCLUSIVE LOCK　133, 139
　SHARED LOCK　133, 139
　UPDATE LOCK　139
ロック期間　141
ロック方式
　オプティミスティック　130, 133
　説明　129

ロック方式 *(続き)*
表示 378
並行性制御 131
ペシミスティック 130, 133
論理演算子
説明 34
AND 34
IS NULL 36
NOT 34, 41
OR 34
論理条件
説明 36
論理データベース 199

[ワ行]

ワイルドカード文字 346

A

ABS (関数) 340
ACOS (関数) 340
ADD CONSTRAINT 122
ADMIN COMMAND
コマンド 177
パラメーター 181
メッセージ 181
abort 179
assertexit 179
backgroundjob 179
backup 179
backuplist 179
checkpointing 179
cleanbgjobinfo 179
close 179
describe 179
errorcode 179
errorexit 179
filespec 179
help 179
hotstandby 179
indexusage 180
info 180
makecp
チェックポイント 180
memory 180
monitor 181
netbackup 181
netbackuplist 181
netstat 181
notify 181
open 181
passthrough status 182
perfmon 182
perfmon diff 182
ADMIN COMMAND *(続き)*
pid 182
proctrace 182
protocols 183
runmerge 183
save parameters 183
shutdown 183
sqllist 183
startmerge 183
status 183
throwout 183
tid 183
trace 184
userid 184
userlist 185, 186
usertrace 186
version 186
ADMIN EVENT 186
ALL (キーワード)
PROPAGATE TRANSACTIONS 275
ALTER REMOTE SERVER ステートメント 187
ALTER TABLE SET HISTORY COLUMNS 189
ALTER TABLE SET NOSYNCHISTORY
説明 191
ALTER TABLE SET SYNCHISTORY
説明 191
ALTER TABLE ステートメント 188
ALTER TRIGGER ステートメント 80, 193
ALTER USER ステートメント 193
AND (演算子) 34, 337
APPEND (キーワード) 275
AS 20
SELECT ステートメントの 20
ASCII (関数) 339
ASIN (関数) 341
ATAN (関数) 341
ATAN2 (関数) 341
AVG (関数) 338

B

bcktime 180
BEGIN 204
BIGINT データ型 170
BINARY データ型 172
CAST 関数 172
BIT_AND 関数 (ビット単位の AND 演算子) 344
BLOB および CLOB 174
BLOB (バイナリー・ラージ・オブジェクト) 17, 174
CAST 関数 172

C

CALL ステートメント 195
プロシージャーの呼び出し 27

CALL ステートメント *(続き)*
EXECDIRECT の例 215
CASCADE 113, 240, 246
REVOKE ステートメントのキーワード 303
CASCADED
予約語 349
CASE 20, 339
CAST (関数) 20, 338
バイナリー値の入力 172
CEILING (関数) 341
CHAR LARGE OBJECT データ型 170
CHAR VARYING データ型 169
CHAR (関数) 339
CHAR データ型 169
CHARACTER LARGE OBJECT データ型 170
CHARACTER VARYING データ型 169
CHARACTER データ型 169
CHECK 122
CLOB データ型 170, 174
COALESCE 338
COLUMNS システム・ビュー 400
COMMIT WORK ステートメント 5, 24, 198
COMMIT ステートメント
ストアード・プロシージャー 55
COMMITBLOCK (キーワード)
DROP SUBSCRIPTION 248
MESSAGE FORWARD 286
MESSAGE GET REPLY 292
REFRESH 298
CONCAT 339
ConnectStrForMaster 197
CONVERTORSTOUNIONS 315
CONVERT_CHAR 338
CONVERT_DATE 338
CONVERT_DECIMAL 338
CONVERT_DOUBLE 338
CONVERT_FLOAT 338
CONVERT_INTEGER 338
CONVERT_LONGVARCHAR 338
CONVERT_NUMERIC 338
CONVERT_REAL 338
CONVERT_SMALLINT 338
CONVERT_TIME 338
CONVERT_TIMESTAMP 338
CONVERT_TINYINT 338
CONVERT_VARCHAR 338
COS (関数) 341
COT (関数) 341
COUNT (関数) 338
cptime 180
CREATE CATALOG ステートメント 126, 199
CREATE EVENT ステートメント 97, 201
CREATE INDEX ステートメント 203
CREATE PROCEDURE ステートメント 204
宣言セクション 31
パラメーター・セクション 28
CREATE PUBLICATION ステートメント
説明 216
CREATE ROLE ステートメント 221
CREATE SCHEMA ステートメント 221
CREATE SEQUENCE ステートメント 96, 223
CREATE SYNC BOOKMARK ステートメント
説明 224
CREATE TABLE ステートメント 225
CREATE TRIGGER ステートメント 59, 229
CREATE USER ステートメント 238
CREATE VIEW ステートメント 238
CREATE [OR REPLACE] REMOTE SERVER ステートメント 220
CURDATE (関数) 342
CURRENT_CATALOG (システム関数) 343
CURRENT_SCHEMA (システム関数) 343
CURRENT_USERID (システム関数) 343
CURSORNAME 204, 211, 212
使用例 211, 214
CURTIME (関数) 342

D

DATE データ型 173
DAYNAME (関数) 342
DAYOFMONTH (関数) 342
DAYOFWEEK (関数) 342
DAYOFYEAR (関数) 342
dbconfigsize 180
dbcreatetime 180
dbfreesize 180
DBMS レベル・エラー
リカバリー 281, 291
dbpagesize 180
dbsize 180
DECIMAL データ型 171
DEFAULT 42
DEFAULT (START AFTER COMMIT 内) 330
DEGREES (関数) 341
DELETE (位置付け) ステートメント 239
DELETE ステートメント 238
DESCRIBE ステートメント 239
DIFFERENCE (関数) 341
DOUBLE PRECISION データ型 174
DOUBLE データ型 171
DROP BOOKMARK ステートメント 224
DROP CATALOG ステートメント 240
DROP CONSTRAINT ステートメント 122
DROP EVENT ステートメント 97, 241
DROP INDEX ステートメント 241
DROP MASTER ステートメント 242
DROP PROCEDURE ステートメント 243
DROP PUBLICATION REGISTRATION ステートメント 244
DROP PUBLICATION ステートメント 243
DROP REMOTE SERVER ステートメント 245
DROP REPLICA ステートメント 245

DROP ROLE ステートメント 246
DROP SCHEMA ステートメント 246
DROP SEQUENCE ステートメント 247
DROP SUBSCRIPTION ステートメント 247
DROP SYNC BOOKMARK ステートメント 249
DROP TABLE ステートメント 251
DROP TRIGGER ステートメント 80, 251
DROP USER ステートメント 252
DROP VIEW ステートメント 252

E

EnableHints (パラメーター) 165
END 204
END LOOP 208
EVENT
  イベントの通知 204
  イベントの登録 204
  イベントの登録解除 204
  イベントのドロップ 241
  イベントの待ち 204
EXCLUSIVE (ロック・モード) 139
EXECDIRECT 212
  使用例 215
  VARCHAR 変数内での SQL ステートメントの使用 215
EXP (関数) 341
EXPLAIN PLAN FOR ステートメント 148, 167, 252
EXPORT SUBSCRIPTION
  説明 253
EXTRACT FROM 342

F

FLOAT データ型 171
FLOOR (関数) 341
fn
  {fn func_name} での使用 32, 41
FOR EACH REPLICA 83
FOREIGN KEY (制約) 123
FULL (キーワード) 275

G

GET_PARAM()
  説明 257
GET_UNIQUE_STRING 211, 339
  使用例 211, 214, 215
GLOBAL
  CREATE TABLE コマンドのキーワード 226
GRANT EXECUTE ON ステートメント 57
GRANT PASSTHROUGH ステートメント 260
GRANT REFRESH ON
  説明 260
GRANT ステートメント 259

H

HINT ステートメント 261
HOUR (関数) 342

I

IF ステートメント
  説明 36
IFNULL (システム関数) 344
IF-THEN 構文
  説明 36
IF-THEN-ELSE 構文
  説明 36
IF-THEN-ELSEIF 構文
  説明 37
imdbsize 180
IMPORT
  説明 267
INSERT ステートメント 269
  デフォルト値の使用 270
  複数行 270
INSERT (ストリング関数) 339
INT データ型 170
INTEGER データ型 170
IS NULL (演算子)
  説明 36

L

LCASE (関数) 339
LEFT (関数) 340
LENGTH (関数) 340
LIKE 226, 336, 337, 346
  START AFTER COMMIT 内 329
LIST ステートメント 270
LOCATE (関数) 340
LOCK TABLE ステートメント 272
Lock timeout
  設定 130
  設定、オプティミスティック表の 131
LOG (関数) 341
LOG10 (関数) 341
LOGIN_CATALOG (システム機能) 343
LOGIN_SCHEMA (システム関数) 343
LOGIN_USERID (システム関数) 343
logsize 180
  'info' コマンド 180
LONG NATIONAL VARCHAR データ型 170
LONG VARBINARY
  CAST を使用した値の入力 172
LONG VARBINARY データ型 172
LONG VARCHAR データ型 170
LONG WVARCHAR データ型 170
LOOP 208
LTRIM (関数) 340

M

MAINTENANCE
  SET SYNC MODE MAINTENANCE 318
MAX (関数) 338
MaxStartStatements (パラメーター) 156
maxusers 180
memtotal 180
MESSAGE APPEND PROPAGATE TRANSACTIONS
  説明 275
MESSAGE APPEND PROPAGATE WHERE
  プロパティーの使用 307
MESSAGE APPEND REFRESH 275
  説明 275
MESSAGE APPEND REGISTER PUBLICATION
  説明 275
MESSAGE APPEND REGISTER REPLICA
  説明 275
MESSAGE APPEND SUBSCRIBE 275
MESSAGE APPEND SYNC_CONFIG
  説明 275
MESSAGE APPEND UNREGISTER PUBLICATION
  説明 275
MESSAGE APPEND UNREGISTER REPLICA
  説明 275
MESSAGE BEGIN
  説明 278
MESSAGE DELETE
  説明 279
MESSAGE END
  説明 283
MESSAGE EXECUTE
  説明 284
MESSAGE FORWARD
  説明 286
MESSAGE FROM REPLICA DELETE 290
  説明 281
MESSAGE FROM REPLICA EXECUTE
  説明 290
MESSAGE FROM REPLICA RESTART 292
MESSAGE GET REPLY
  説明 292
MIN (関数) 338
MINUTE (関数) 342
MOD (関数) 341
monitorstate 180
MONTH (関数) 342
MONTHNAME (関数) 342

N

NATIONAL CHAR データ型 169
NATIONAL CHARACTER データ型 169
NATIONAL VARCHAR データ型 170
NCHAR LARGE OBJECT データ型 170
NCHAR VARYING データ型 170
NCHAR データ型 169
NCLOB データ型 170
node-def 42
NONUNIQUE 83
NORMAL
  SET SYNC MODE NORMAL 318
NOT NULL 20
NOT (演算子) 34, 337
NOTUNIQUE 329
NOW 342
NULL 18
NULLIF 338
numcursors 180
NUMERIC データ型 172
numlocks 180
nummerges 180
numtransactions 180
numusers 180
NVARCHAR データ型 170

O

openstate 180
OR (演算子) 34, 337

P

PI (関数) 341
POSITION (関数) 340
POWER (関数) 341
PRECISION データ型 171
primarystarttime 180
processsize 180
proctrace 155
PROC_COUNT 関数
  ストアード・プロシージャー・スタック 57
PROC_NAME (N) 関数
  ストアード・プロシージャー・スタック 57
PROC_SCHEMA (N) 関数
  ストアード・プロシージャー 57
psize 180
PUT_PARAM()
  説明 297

Q

QUARTER (関数) 342

R

RADIANS (関数) 341
READ COMMITTED 310
REAL データ型 171, 174
REFERENCES (キーワード) 226, 259, 303
REFRESH ステートメント 298

REFRESH ステートメント (続き)
コミット・ブロックの定義 286
REGISTER EVENT ステートメント 302
REPEAT (関数) 340
REPEATABLE READ 310
REPLACE (関数) 340
RESTRICT キーワード 113, 240, 246, 251
REVOKE ステートメント 303
RETURN キーワード 42
REVOKE PASSTHROUGH ステートメント 303
REVOKE REFRESH 304
REVOKE REFRESH ON
説明 304
REVOKE SUBSCRIBE 304
REVOKE (ユーザーからロールを) ステートメント 302
REVOKE (ロールまたはユーザーから特権を) ステートメント 303
RIGHT (関数) 340
ROLLBACK WORK ステートメント 304
ROLLBACK (ステートメント) 5
ストアード・プロシージャー 55
ROUND (関数) 341
ROWID 161
ROWNUM 160, 346, 359
RTRIM (関数) 340
RVC (行値コンストラクター) 22

S

SAVE
説明 305
SAVE DEFAULT PROPAGATE PROPERTY WHERE
説明 307
SAVE DEFAULT PROPERTY
説明 307
SAVE PROPERTY ステートメント 307
SECOND 342
secondarystarttime 180
SELECT ステートメント 308
SERIALIZABLE 310
sernum 180
SERVER_INFO システム・ビュー 401
SET
SET および SET TRANSACTION の違い 325
SET CATALOG catalog_name 310
SET CATALOG ステートメント 125
SET DELETE CAPTURE ステートメント 312
SET DURABILITY 144, 310
SET HISTORY COLUMNS
説明 191
SET IDLE TIMEOUT 310
SET ISOLATION LEVEL 310
SET LOCK TIMEOUT 310
SET NOSYNCHISTORY
説明 191
SET OPTIMISTIC LOCK TIMEOUT 310
SET PASSTHROUGH ステートメント 313
SET READ-ONLY 310
SET READ-WRITE 310
SET SAFENESS 310
SET SCHEMA 310
SET SCHEMA USER ステートメント 314
SET SCHEMA ステートメント 125, 314
SET SQL ステートメント 315
SET STATEMENT MAXTIME 310
SET SYNC CONNECT 197
説明 317
SET SYNC MODE ステートメント 318
SET SYNC NODE
説明 320
SET SYNC PARAMETER
説明 321
SET SYNC USER IDENTIFIED BY
説明 323
SET SYNCHISTORY 189
説明 191
SET TRANSACTION
SET および SET TRANSACTION の違い 325
SET TRANSACTION DELETE CAPTURE ステートメント 327
SET TRANSACTION DURABILITY 144
SET TRANSACTION PASSTHROUGH ステートメント 328
SET TRANSACTION WRITE 324
SET TRANSACTION ステートメント 324
SET WRITE 310
SET および SET TRANSACTION の違い 325
SET ステートメント 310
ストアード・プロシージャー内 32
SHARED (ロック・モード) 139
SIGN (関数) 341
SimpleSQLOpt (パラメーター) 160
SIN 341
SLEEP 344
SMALLINT データ型 170
solidDB
データ管理 129
solidDB SQL
拡張機能 107
関数 108
データ型 107
データ管理 129
データベース管理のための使用 107
solidDB SQL 構文
準拠性 107
使用 107
soltrace.out 154
SOUNDEX 340
SPACE 340
SQL
概要 9
ストアード・プロシージャーでの使用 57
副照会 14

SQL 関数
GET_PARAM()　257, 258
PUT_PARAM()　297
SQL 情報機能　147
SQL スクリプト　108
sample.sql　111
users.sql　108
SQL ステートメント
索引を管理する例　115
使用　107
チューニング　159
データベース・オブジェクトを管理する例　126
ユーザー、ロール、およびユーザー特権を管理する例　109
例　111
SQL ワイルドカード　346
SQLERRNUM (変数)
エラー・コード　50
SQLERROR OF cursorname (変数)　51
SQLERROR (変数)
エラー・ストリング　50
SQLERRSTR (変数)
エラー・ストリング　50
SQLROWCOUNT (変数)
行数　50
SQLSUCCESS (変数)
ストアード・プロシージャー　50
SQL-92　107
SQL-99　107
SQL_LANGUAGES システム表　363
SQL_TSI_DAY　342, 343
SQL_TSI_FRAC_SECOND　342, 343
SQL_TSI_HOUR　342, 343
SQL_TSI_MINUTE　342, 343
SQL_TSI_MONTH　342, 343
SQL_TSI_QUARTER　342, 343
SQL_TSI_SECOND　342, 343
SQL_TSI_WEEK　342, 343
SQL_TSI_YEAR　342, 343
SQRT　341
SSC_TASK_BACKGROUND　156
START AFTER COMMIT ステートメント　329
障害の分析　156
パフォーマンスのチューニング　156
STORE
CREATE TABLE コマンドの STORE 節　226
SUBSTRING (関数)　340
SUM (関数)　338
SYNCHISTORY　189
SYNC_CONFIG　275
SYNC_DELETE_MESSAGES　419
SYNC_DELETE_REPLICA_MESSAGES　420
SYNC_MAINTENANCEMODE_BEGIN (イベント)　318, 427
SYNC_MAINTENANCEMODE_END (イベント)　318, 427
SYNC_REGISTER_PUBLICATION　415
SYNC_REGISTER_REPLICA　412
SYNC_SETUP_CATALOG　411
SYNC_SHOW_REPLICA_SUBSCRIPTIONS　418
SYNC_SHOW_SUBSCRIPTIONS　417
SYNC_UNREGISTER_PUBLICATION　416
SYNC_UNREGISTER_REPLICA　413
SYS_ADMIN_ROLE　259
SYS_ATTAUTH システム表　363
SYS_BACKGROUNDJOB_INFO　156
SYS_BACKGROUNDJOB_INFO システム表　365
SYS_BLOBS システム表　365
SYS_BULLETIN_BOARD システム表　382
SYS_CARDINAL システム表　366
SYS_CATALOGS システム表　366
SYS_CHECKSTRINGS システム表　367
SYS_COLUMNS システム表　367
SYS_COLUMNS_AUX システム表　368
SYS_DL_REPLICA_CONFIG システム表　368
SYS_DL_REPLICA_DEFAULT システム表　369
SYS_EVENTS システム表　369
SYS_EVENT_BACKUP　424
SYS_EVENT_BACKUPREQ　424
SYS_EVENT_CHECKPOINT (イベント)　424
SYS_EVENT_CHECKPOINTREQ　425
SYS_EVENT_ERROR　425, 431, 432
SYS_EVENT_IDLE　425
SYS_EVENT_ILL_LOGIN　426
SYS_EVENT_IMDB_MEMORY　426
SYS_EVENT_MERGE　427
SYS_EVENT_MERGEREQ　427
SYS_EVENT_MESSAGES　428
SYS_EVENT_NOTIFY　428
SYS_EVENT_PARAMETER　428
SYS_EVENT_PROCESS_MEMORY　429
SYS_EVENT_ROWS2MERGE　429
SYS_EVENT_SACFAILED　156, 430
SYS_EVENT_SHUTDOWNREQ　430
SYS_EVENT_STATE_MONITOR　430
SYS_EVENT_STATE_OPEN　430
SYS_EVENT_STATE_SHUTDOWN　430
SYS_EVENT_STATE_TRACE　431
SYS_EVENT_TMCMD　431
SYS_EVENT_TRX_TIMEOUT　431
SYS_EVENT_USERS　431
SYS_FORKEYPARTS システム表　371
SYS_FORKEYS システム表　372
SYS_GETBACKGROUNDJOB_INFO　156, 420
SYS_HOTSTANDBY システム表　372
SYS_KEYPARTS システム表　372
SYS_KEYS システム表　373
SYS_PROCEDURES システム表　374
SYS_PROCEDURE_COLUMNS システム表　375
SYS_PROPERTIES システム表　376
SYS_PUBLICATIONS システム表　385
SYS_PUBLICATIONS_REPLICA システム表　386
SYS_PUBLICATION_ARGS システム表　383
SYS_PUBLICATION_REPLICA_ARGS システム表　383
SYS_PUBLICATION_REPLICA_STMTARGS システム表　383

SYS_PUBLICATION_REPLICA_STMTS システム表 384
SYS_PUBLICATION_STMTARGS システム表 384
SYS_PUBLICATION_STMTS システム表 385
SYS_RELAUTH システム表 376
SYS_SCHEMAS システム表 377
SYS_SEQUENCES システム表 377
SYS_SYNC_ADMIN_ROLE 259
SYS_SYNC_BOOKMARKS システム表 386
SYS_SYNC_HISTORY_COLUMNS システム表 387
SYS_SYNC_INFO システム表 387
SYS_SYNC_MASTERS システム表 392
SYS_SYNC_MASTER_MSGINFO システム表 387
SYS_SYNC_MASTER_RECEIVED_BLOB_REFS システム表 389
SYS_SYNC_MASTER_RECEIVED_MSGPARTS システム表 389
SYS_SYNC_MASTER_RECEIVED_MSGS システム表 389
SYS_SYNC_MASTER_STORED_BLOB_REFS システム表 390
SYS_SYNC_MASTER_STORED_MSGPARTS システム表 390
SYS_SYNC_MASTER_STORED_MSGS システム表 390
SYS_SYNC_MASTER_SUBSC_REQ システム表 391
SYS_SYNC_MASTER_VERSIONS システム表 391
SYS_SYNC_RECEIVED_BLOB_ARGS システム表 392
SYS_SYNC_RECEIVED_STMTS システム表 393
SYS_SYNC_REPLICAS システム表 398
SYS_SYNC_REPLICA_MSGINFO システム表 394
SYS_SYNC_REPLICA_PROPERTIES システム表 378
SYS_SYNC_REPLICA_RECEIVED_BLOB_REFS システム表 395
SYS_SYNC_REPLICA_RECEIVED_MSGPARTS システム表 395
SYS_SYNC_REPLICA_RECEIVED_MSGS システム表 396
SYS_SYNC_REPLICA_STORED_BLOB_REFS システム表 396
SYS_SYNC_REPLICA_STORED_MSGPARTS システム表 397
SYS_SYNC_REPLICA_STORED_MSGS システム表 396
SYS_SYNC_REPLICA_VERSIONS システム表 397
SYS_SYNC_SAVED_BLOB_ARGS システム表 398
SYS_SYNC_SAVED_STMTS システム表 398
SYS_SYNC_TRX_PROPERTIES システム表 399
SYS_SYNC_USERMAPS システム表 399
SYS_SYNC_USERS システム表 400
SYS_SYNONYM システム表 378
SYS_TABLEMODES システム表 378
SYS_TABLES システム表 379
SYS_TRIGGERS システム表 380
SYS_TRIGGERS (システム表) 81
SYS_TYPES システム表 380
SYS_UROLE システム表 381
SYS_USERS システム表 381
SYS_VIEWS システム表 382

T

TABLES システム・ビュー 401
TAN 341
THEN
  CASE ステートメントのキーワード 339
TIME データ型 173
TIMEOUT (キーワード)
  MESSAGE FORWARD 286
  MESSAGE GET REPLY 286
TIMESTAMP データ型 173
TIMESTAMPADD 342
TIMESTAMPDIFF 343
TINYINT データ型 170
TO (キーワード)
  MESSAGE FORWARD 286
tracestate 180
TRIM (関数) 340
TRUNCATE TABLE ステートメント 332
TRUNCATE (関数) 341

U

UCASE (関数) 340
UIC (システム関数) 343
UNIQUE 83, 123, 329
UNLOCK TABLE ステートメント 332
UPDATE (位置付け) ステートメント 334
UPDATE (検索付き) ステートメント 334
UPDATE (ロック・モード) 139
uptime 180
userlist 185, 186
USERS システム・ビュー 402
usertrace 154

V

VARBINARY データ型 172
  CAST 関数 172
VARCHAR データ型 169

W

WCHAR データ型 169
WEEK (関数) 343
WHEN
  イベント指定の中のキーワード 201
  case_specification 内 339
WHERE (キーワード)
  PROPAGATE TRANSACTIONS 275
WHILE-LOOP ステートメント
  説明 39
WRITETRACE 154
WVARCHAR データ型 170

Y

YEAR (関数) 343

［特殊文字］

(greaterthan)″ status=″unchanged″>> (より大きい)　336
(notequalto)″ status=″unchanged″><> (等しくない)　336
* (アスタリスク)　337
+ (plus)　339
+ (プラス)　337
- (マイナス)　337
/ (スラッシュ)　337
= (等しい)　336
=(greaterthanorequalto)″ status=″unchanged″>>= (より大か等しい)
336
< (より小さい)　336
<= (より小か等しい)　336
|| (連結演算子)　339
%　346
% 記号　346
_ (下線)　346

特記事項

Copyright © Solid Information Technology Ltd. 1993, 2009.

All rights reserved.

Solid Information Technology Ltd. または International Business Machines Corporation の書面による明示的な許可がある場合を除き、本製品のいかなる部分も、いかなる方法においても使用することはできません。

本製品は、米国特許 6144941、 7136912、 6970876、 7139775、 6978396、 7266702、 7406489、および 7502796 により保護されています。

本製品は、米国輸出規制品目分類番号 ECCN=5D992b に指定されています。

本書は米国 IBM が提供する製品およびサービスについて作成したものです。

本書に記載の製品、サービス、または機能が日本においては提供されていない場合があります。日本で利用可能な製品、サービス、および機能については、日本 IBM の営業担当員にお尋ねください。本書で IBM 製品、プログラム、またはサービスに言及していても、その IBM 製品、プログラム、またはサービスのみが使用可能であることを意味するものではありません。これらに代えて、IBM の知的所有権を侵害することのない、機能的に同等の製品、プログラム、またはサービスを使用することができます。ただし、IBM 以外の製品とプログラムの操作またはサービスの評価および検証は、お客様の責任で行っていただきます。

IBM は、本書に記載されている内容に関して特許権 (特許出願中のものを含む) を保有している場合があります。本書の提供は、お客様にこれらの特許権について実施権を許諾することを意味するものではありません。実施権についてのお問い合わせは、書面にて下記宛先にお送りください。

〒242-8502
神奈川県大和市下鶴間1623番14号
日本アイ・ビー・エム株式会社
法務・知的財産
知的財産権ライセンス渉外

以下の保証は、国または地域の法律に沿わない場合は、適用されません。 IBM およびその直接または間接の子会社は、本書を特定物として現存するままの状態で提供し、商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任を含むすべての明示もしくは黙示の保証責任を負わないものとします。国または地域によっては、法律の強行規定により、保証責任の制限が禁じられる場合、強行規定の制限を受けるものとします。

この情報には、技術的に不適切な記述や誤植を含む場合があります。本書は定期的に見直され、必要な変更は本書の次版に組み込まれます。 IBM は予告なしに、随時、この文書に記載されている製品またはプログラムに対して、改良または変更を行うことがあります。

本書において IBM 以外の Web サイトに言及している場合がありますが、便宜のため記載しただけであり、決してそれらの Web サイトを推奨するものではありません。それらの Web サイトにある資料は、この IBM 製品の資料の一部ではありません。それらの Web サイトは、お客様の責任でご使用ください。

IBM は、お客様が提供するいかなる情報も、お客様に対してなんら義務も負うことのない、自ら適切と信ずる方法で、使用もしくは配布することができるものとします。

本プログラムのライセンス保持者で、(i) 独自に作成したプログラムとその他のプログラム (本プログラムを含む) との間での情報交換、および (ii) 交換された情報の相互利用を可能にすることを目的として、本プログラムに関する情報を必要とする方は、下記に連絡してください。

IBM Canada Limited
Office of the Lab Director
8200 Warden Avenue
Markham, Ontario
L6G 1C7
CANADA

本プログラムに関する上記の情報は、適切な使用条件の下で使用することができますが、有償の場合もあります。

本書で説明されているライセンス・プログラムまたはその他のライセンス資料は、IBM 所定のプログラム契約の契約条項、IBM プログラムのご使用条件、またはそれと同等の条項に基づいて、IBM より提供されます。

この文書に含まれるいかなるパフォーマンス・データも、管理環境下で決定されたものです。そのため、他の操作環境で得られた結果は、異なる可能性があります。一部の測定が、開発レベルのシステムで行われた可能性がありますが、その測定値が、一般に利用可能なシステムのものと同じである保証はありません。さらに、一部の測定値が、推定値である可能性があります。実際の結果は、異なる可能性があります。お客様は、お客様の特定の環境に適したデータを確かめる必要があります。

IBM 以外の製品に関する情報は、その製品の供給者、出版物、もしくはその他の公に利用可能なソースから入手したものです。IBM は、それらの製品のテストは行っておりません。したがって、他社製品に関する実行性、互換性、またはその他の要求については確証できません。IBM 以外の製品の性能に関する質問は、それらの製品の供給者にお願いします。

IBM の将来の方向または意向に関する記述については、予告なしに変更または撤回される場合があり、単に目標を示しているものです。

本書には、日常の業務処理で用いられるデータや報告書の例が含まれています。より具体性を与えるために、それらの例には、個人、企業、ブランド、あるいは製品などの名前が含まれている場合があります。これらの名称はすべて架空のものであり、名称や住所が類似する企業が実在しているとしても、それは偶然にすぎません。

著作権使用許諾:

本書には、様々なオペレーティング・プラットフォームでのプログラミング手法を例示するサンプル・アプリケーション・プログラムがソース言語で掲載されています。お客様は、サンプル・プログラムが書かれているオペレーティング・プラットフォームのアプリケーション・プログラミング・インターフェースに準拠したアプリケーション・プログラムの開発、使用、販売、配布を目的として、いかなる形式においても、IBM に対価を支払うことなくこれを複製し、改変し、配布することができます。このサンプル・プログラムは、あらゆる条件下における完全なテストを経ていません。従って IBM は、これらのサンプル・プログラムについて信頼性、利便性もしくは機能性があることをほのめかしたり、保証することはできません。

それぞれの複製物、サンプル・プログラムのいかなる部分、またはすべての派生的創作物にも、次のように、著作権表示を入れていただく必要があります。

© (お客様の会社名) (西暦年)。このコードの一部は、IBM Corp. のサンプル・プログラムから取られています。

© Copyright IBM Corp. _年を入れる_. All rights reserved.

商標

IBM、IBM ロゴ、ibm.com®、Solid、solidDB、InfoSphere™、DB2®、Informix®、および WebSphere® は、International Business Machines Corporation の米国およびその他の国における商標です。これらおよび他の IBM 商標に、この情報の最初に現れる個所で商標表示 (® または ™) が付されている場合、これらの表示は、この情報が公開された時点で、米国において、IBM が所有する登録商標またはコモン・ロー上の商標であることを示しています。このような商標は、その他の国においても登録商標またはコモン・ロー上の商標である可能性があります。現時点での IBM の商標リストについては、「Copyright and trademark information」(www.ibm.com/legal/copytrade.shtml) をご覧下さい。

Java およびすべての Java 関連の商標およびロゴは Sun Microsystems, Inc.の米国およびその他の国における商標です。

Linux® は、Linus Torvalds の米国およびその他の国における商標です。

Microsoft および Windows は、Microsoft Corporation の米国およびその他の国における商標です。

UNIX は、The Open Group の米国およびその他の国における登録商標です。

他の会社名、製品名およびサービス名等はそれぞれ各社の商標です。

Printed in Japan

SC88-8168-00

日本アイ・ビー・エム株式会社

〒103-8510 東京都中央区日本橋箱崎町19-21