ALLBASE/SQL
Reference Manual

HP 3000 MPE/iX Computer Systems

Manufacturing Part Number: 36216-90216
E0300

U.S.A. March 2000

Notice

The information contained in this document is subject to change without notice.

Hewlett-Packard makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability or fitness for a particular purpose. Hewlett-Packard shall not be liable for errors contained herein or for direct, indirect, special, incidental or consequential damages in connection with the furnishing or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

Restricted Rights Legend

Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013. Rights for non-DOD U.S. Government Departments and Agencies are as set forth in FAR 52.227-19 (c) (1,2).

Acknowledgments

Hewlett-Packard Company
3000 Hanover Street
Palo Alto, CA 94304 U.S.A.

Contents

1. Introduction

ALLBASE/SQL Components 42
Utility Programs 43
ALLBASE/SQL Databases 44
Logical Concepts 44
Physical Concepts 45
ALLBASE/SQL Data Access 48
Using Queries 49
ALLBASE/SQL Objects 50
ALLBASE/SQL Users 51
SQL Language Structure 52
Using Comments within SQL Statements 53
SQL Statement Categories 54
Error Conditions in ALLBASE/SQL 56
Severity of Errors 56
Atomicity of Error Checking 56
Additional Information about Errors 57
Native Language Support 57

2. Using ALLBASE/SQL

Creating DBEnvironments 60
Specifying a Native Language Parameter 60
Initial Privileges 61
Starting and Terminating a DBE Session 62
Sessions with Autostart 62
Sessions without Autostart 62
Terminating DBE Sessions 62
Creating Physical Storage 63
Defining How Data is Stored and Retrieved 64
Creating a Table 64
Specifying a DBEFileSet 67
Specifying Native Language Tables and Columns 67
Creating a View 67
Creating Indexes 68
Specifying Integrity Constraints 68
Creating Procedures 68
Creating Rules 69
Understanding Data Access Paths 70
Serial Access 70
Indexed Access 70
Hashed Access 72
Differences between Hashed and Indexed Access 74
When to Use a Hash Structure 74
TID Access 74
Controlling Database Access 75
Authorities 75
Obtaining Authorization 75
DBA Authority 76
Grants 76

Grantable Privileges 76
Ownership 77
Default Owner Rules 78
Ownership Privileges 78
Authorization Groups 79
Classes 80
Differences between Groups and Classes 80
Manipulating Data 81
Inserting Data 81
Updating Data 82
Deleting Data 82
Managing Transactions 83
Objectives of Transaction Management 83
Starting Transactions 85
Ending Transactions 85
Using SAVEPOINT 86
Scoping of Transaction and Session Attributes 87
Transaction Limits and Timeouts 89
Monitoring Transactions 90
Tips on Transaction Management 90
Auditing DBEnvironments 91
Partitions in Audit DBEnvironments 91
Using Wrapper DBEnvironments 92
Using SQLAudit 92
Application Programming 93
Preprocessor 93
Authorization 94
DBEnvironment Changes 94
Host Variables 94
Multiple-Row Manipulations 95
Using Multiple Connections and Transactions with Timeouts 95
Connecting to DBEnvironments 96
Setting the Current Connection 96
Setting Timeout Values 97
Setting the Transaction Mode 99
Disconnecting from DBEnvironments 103
Administering a Database 105
Understanding the System Catalog 105

3. SQL Queries

Using the SELECT Statement 110
Simple Queries 112
Complex Queries 116
UNION Queries 117
Using Character Constants with UNION 119
Subqueries 120
Special Predicates 120
Quantified Predicate 120
IN Predicate 123

EXISTS Predicate 124
Correlated Versus Noncorrelated Subqueries 126
Outer Joins 126
Using GENPLAN to Display the Access Plan 134
Generating a Plan 134
Displaying a Query Access Plan 134
Interpreting a Display 135
Updatability of Queries 136

4. Constraints, Procedures, and Rules
Using Integrity Constraints 137
Unique Constraints 137
Referential Constraints 138
Check Constraints 139
Examples of Integrity Constraints 141
Inserting Rows in Tables Having Constraints 143
How Constraints are Enforced 144
Using Procedures 145
Understanding Procedures 146
Creating Procedures 146
Executing Procedures 147
Procedures and Transaction Management 147
Using SQL Statements in Procedures 148
Queries inside Procedures 151
Using a Procedure Cursor in ISQL 153
Error Handling in Procedures Not Invoked by Rules 154
Using RAISE ERROR in Procedures 155
Recommended Coding Practices for Procedures 156
Using Rules 157
Understanding Rules 158
Creating Rules 158
Techniques for Using Procedures with Rules 159
Error Handling in Procedures Invoked by Rules 161
Using RAISE ERROR in Procedures Invoked by Rules 161
Enabling and Disabling Rules 163
Special Considerations for Procedures Invoked by Rules 163
Differences between Rules and Integrity Constraints 166

5. Concurrency Control through Locks and Isolation Levels
Defining Transactions 168
Understanding ALLBASE/SQL Data Access 169
Use of Locking by Transactions 171
Basics of Locking 171
Locks and Queries 171
Costs of Locking 172
Defining Isolation Levels between Transactions 174
Repeatable Read (RR) 174
Cursor Stability (CS) 174

Read Committed (RC) 175
Read Uncommitted (RU) 176
Details of Locking 177
Lock Granularities 177
Types of Locks. 179
Lock Compatibility 180
Weak Locks 181
What Determines Lock Types 181
Type of SQL Statement 182
Locking Structure Implicit at CREATE TABLE Time. 182
Use of the LOCK TABLE Statement. 183
Choice of a Scan Type. 183
Choice of Isolation Level 184
Updatability of Cursors or Views 187
Use of Sorting 187
Scope and Duration of Locks. 188
Examples of Obtaining and Releasing Locks. 189
Simple Example of Concurrency Control through Locking 189
Sample Transactions Using Isolation Levels 191
Resolving Conflicts among Concurrent Transactions 194
Lock Waits. 194
Deadlocks 195
Table Type and Deadlock 195
Table Size and Deadlock 196
Avoiding Deadlock 197
Undetectable Deadlock. 198
Monitoring Locking with SQLMON 199
MONITOR Authority 199
Monitoring Tasks 199

6. Names

Basic Names. 202
Native Language Object Names 203
DBEUserIDs 203
Owner Names 203
Authorization Names 204
Compound Identifiers 204
Host Variable Names. 205
Local Variable Names 205
Parameter Names 205
DBEnvironment and DBECon File Names 205
DBEFile and Log File Identifiers 206
TempSpace Names. 206
Special Names 206

7. Data Types

Type Specifications 208
Value Comparisons 211

Overflow and Truncation 212
Underflow 212
Type Conversion 213
Null Values 215
Decimal Operations 216
Date/Time Operations 217
Examples 217
Use of Date/Time Data Types in Arithmetic Expressions 218
Use of Date/Time Data Types in Predicates 219
. 220
Binary Operations 220
Long Operations 221
Defining LONG Column Data with CREATE TABLE or ALTER TABLE. 221
Defining Input and Output with the LONG Column I/O String 222
Using INSERT with LONG Column Data. 222
Using SELECT with LONG Column Data 224
Using UPDATE with LONG Column Data 224
Native Language Data 226

8. Expressions
Expression 228
Scope 228
SQL Syntax 228
Parameters 229
Description. 231
Example 233
Add Months Function. 234
Scope 234
SQL Syntax 234
Parameters 234
Description. 234
Example 235
Aggregate Functions 236
Scope 236
SQL Syntax 236
Parameters 236
Description. 237
Example 237
CAST Function 238
Scope 238
SQL Syntax 238
Parameters 238
Description. 238
Examples 241
Constant 243
Scope 243
SQL Syntax 243
Parameters 243
Current Functions 244

Scope 244
SQL Syntax. 244
Description 244
Examples. 244
Date/Time Functions 245
Scope 245
SQL Syntax—Conversion Functions. 245
Parameters—Conversion Functions 245
SQL Syntax—FormatSpecification 246
Parameters—FormatSpecification. 246
Description 247
Examples. 250
Long Column Functions 251
Scope 251
SQL Syntax. 251
Parameters 251
Description 251
Examples. 252
String Functions 253
Function Specification 253
Examples: 254
Scope 255
SQL Syntax. 255
Parameters 255
Description 256
Examples. 256
TID Function 258
Scope 258
SQL Syntax. 258
Parameters 258
Description 258
Example 260

9. Search Conditions

Search Condition 262
Scope 262
SQL Syntax. 262
Parameters 262
Description 263
BETWEEN Predicate 264
Scope 264
SQL Syntax. 264
Parameters 264
Description 264
Example 264
Comparison Predicate 265
Scope 265
SQL Syntax. 265
Parameters 265

Description. 265
Example 266
EXISTS Predicate. 267
Scope 267
SQL Syntax 267
Parameters 267
Description. 267
Example 267
IN Predicate 268
Scope 268
SQL Syntax 268
Parameters 268
Description. 270
Example 270
LIKE Predicate 272
Scope 272
SQL Syntax 272
Parameters 272
Description. 274
Example 274
NULL Predicate 275
Scope 275
SQL Syntax 275
Parameters 275
Description. 277
Example 277
Quantified Predicate 278
Scope 278
SQL Syntax 278
Parameters 278
Description. 280
Example 281

10. SQL Statements A - D

SQL Statement Summary 283
ADD DBEFILE 293
Scope 293
SQL Syntax 293
Parameters 293
Description. 293
Authorization. 293
Example 293
ADD TO GROUP 295
Scope 295
SQL Syntax 295
Parameters 295
Description. 295
Authorization. 295
Example 295

ADVANCE 297
Scope 297
SQL Syntax 297
Parameters 297
Description 297
Authorization 298
Example 298
ALTER DBEFILE 299
Scope 299
SQL Syntax 299
Parameters 299
Description 299
Authorization 299
Example 300
ALTER TABLE 301
Scope 301
SQL Syntax 301
Parameters—ALTER TABLE 301
SQL Syntax—AddColumnSpecification 301
Parameters—AddColumnSpecification 301
SQL Syntax—AddConstraintSpecification 302
Parameters—AddConstraintSpecification 302
SQL Syntax—DropConstraintSpecification 302
Parameters—DropConstraintSpecification 302
SQL Syntax—SetTypeSpecification 303
Parameters—SetTypeSpecification 303
SQL Syntax—SetPartitionSpecification 304
Parameters—SetPartitionSpecification 304
Description 304
Authorization 306
Examples 306
Assignment (=) 307
Scope 307
SQL Syntax 307
Parameters 307
Description 307
Authorization 307
Example 308
BEGIN 309
Scope 309
SQL Syntax 309
Parameters 309
Description 309
Authorization 309
Example 309
BEGIN ARCHIVE 310
Scope 310
SQL Syntax 310
Description 310

Authorization. 310
BEGIN DECLARE SECTION 311
Scope 311
SQL Syntax 311
Description. 311
Authorization. 311
Example 311
BEGIN WORK 312
Scope 312
SQL Syntax 312
Parameters 312
Description. 313
Authorization. 314
Examples 314
CHECKPOINT 316
Scope 316
SQL Syntax 316
Parameters 316
Description. 316
Authorization. 317
Example 317
CLOSE 319
Scope 319
SQL Syntax 319
Parameters 319
Description. 320
Authorization. 320
Examples 320
COMMIT ARCHIVE 322
Scope 322
SQL Syntax 322
Description. 322
Authorization. 322
COMMIT WORK 323
Scope 323
SQL Syntax 323
Parameters 323
Description. 323
Authorization. 323
Example 324
CONNECT 325
Scope 325
SQL Syntax 325
Parameters 325
Description. 326
Authorization. 326
Example 326
CREATE DBEFILE 327
Scope 327

SQL Syntax. 327
Parameters 327
Description 328
Authorization 329
Example 329
CREATE DBEFILESET 330
Scope 330
SQL Syntax. 330
Parameters 330
Description 330
Authorization 331
Example 331
CREATE GROUP 332
Scope 332
SQL Syntax. 332
Parameters 332
Description 332
Authorization 332
Example 333
CREATE INDEX 334
Scope 334
SQL Syntax. 334
Parameters 334
Description 335
Authorization 335
Example 336
CREATE PARTITION. 337
Scope 337
SQL Syntax. 337
Parameters 337
Description 337
Authorization 338
Example 338
CREATE PROCEDURE 339
Scope 339
SQL Syntax. 339
Parameters 339
SQL Syntax—ParameterDeclaration 340
Parameters—ParameterDeclaration. 341
SQL Syntax—ResultDeclaration. 341
Parameters—ResultDeclaration 341
Description 342
Authorization 344
Examples. 344
CREATE RULE 346
Scope 346
SQL Syntax. 346
Parameters 346
Description 348

Authorization. 349
Example 349
CREATE SCHEMA 351
Scope 351
SQL Syntax 351
Parameters 351
Description. 352
Authorization. 352
Example 352
CREATE TABLE 354
Scope 354
SQL Syntax—CREATE TABLE 354
Parameters—CREATE TABLE 354
SQL Syntax—Column Definition 357
Parameters—Column Definition 357
SQL Syntax—Unique Constraint (Table Level) 358
Parameters—Unique Constraint (Table Level). 358
SQL Syntax—Referential Constraint (Table Level) 358
Parameters—Referential Constraint (Table Level). 359
SQL Syntax—Check Constraint (Table Level) 359
Parameters—Check Constraint (Table Level). 359
Description. 359
Authorization. 362
Examples 362
CREATE TEMPSPACE 365
Scope 365
SQL Syntax 365
Parameters 365
Description. 365
Authorization. 366
Example 366
CREATE VIEW 367
Scope 367
SQL Syntax 367
Parameters 367
Description. 368
Authorization. 369
Examples 369
DECLARE CURSOR 371
Scope 371
SQL Syntax 371
Parameters 371
Description. 372
Authorization. 373
Examples 374
DECLARE Variable 376
Scope 376
SQL Syntax 376
Parameters 376

Description 376
Authorization 377
Example 377
DELETE 378
Scope 378
SQL Syntax 378
Parameters 378
Description 378
Authorization 380
Example 380
DELETE WHERE CURRENT 381
Scope 381
SQL Syntax 381
Parameters 381
Description 381
Authorization 382
Example 383
DESCRIBE 384
Scope 384
SQL Syntax 384
Parameters 384
Description 385
Authorization 386
Examples 386
DISABLE AUDIT LOGGING 389
Scope 389
SQL Syntax 389
Description 389
Authorization 389
Example 389
DISABLE RULES 390
Scope 390
SQL Syntax 390
Description 390
Authorization 390
Example 390
DISCONNECT 391
Scope 391
SQL Syntax 391
Parameters 391
Description 391
Authorization 392
Example 392
DROP DBEFILE 393
Scope 393
SQL Syntax 393
Parameters 393
Description 393
Authorization 393

Example . . . 393
DROP DBEFILESET . . . 395
Scope . . . 395
SQL Syntax . . . 395
Parameters . . . 395
Description . . . 395
Authorization . . . 395
Example . . . 395
DROP GROUP . . . 397
Scope . . . 397
SQL Syntax . . . 397
Parameters . . . 397
Description . . . 397
Authorization . . . 397
Example . . . 397
DROP INDEX . . . 399
Scope . . . 399
SQL Syntax . . . 399
Parameters . . . 399
Description . . . 399
Authorization . . . 399
Example . . . 400
DROP MODULE . . . 401
Scope . . . 401
SQL Syntax . . . 401
Parameters . . . 401
Description . . . 401
Authorization . . . 401
Examples . . . 402
DROP PARTITION . . . 403
Scope . . . 403
SQL Syntax . . . 403
Parameters . . . 403
Description . . . 403
Authorization . . . 403
Example . . . 403
DROP PROCEDURE . . . 404
Scope . . . 404
SQL Syntax . . . 404
Parameters . . . 404
Description . . . 404
Authorization . . . 404
Example . . . 404
DROP RULE . . . 405
Scope . . . 405
SQL Syntax . . . 405
Parameters . . . 405
Description . . . 405
Authorization . . . 405

Example 405
DROP TABLE 406
Scope 406
SQL Syntax 406
Parameters 406
Description 406
Authorization 406
Example 406
DROP TEMPSPACE 408
Scope 408
SQL Syntax 408
Parameters 408
Description 408
Authorization 408
Example 408
DROP VIEW 409
Scope 409
SQL Syntax 409
Parameters 409
Description 409
Authorization 409
Example 409

11. SQL Statements E - R
ENABLE AUDIT LOGGING 411
Scope 411
SQL Syntax 411
Description 411
Authorization 411
Example 411
ENABLE RULES 413
Scope 413
SQL Syntax 413
Description 413
Authorization 413
Example 413
END DECLARE SECTION 414
Scope 414
SQL Syntax 414
Description 414
Authorization 414
Example 414
EXECUTE 415
Scope 415
SQL Syntax 415
Parameters 415
SQL Syntax — HostVariableSpecification 416
Parameters — HostVariableSpecification 416
Description 417

Authorization. 417
Examples 418
EXECUTE IMMEDIATE 420
Scope 420
SQL Syntax 420
Parameters 420
Description. 420
Authorization. 420
Example 420
EXECUTE PROCEDURE 421
Scope 421
Syntax 421
Parameters 421
SQL Syntax—ActualParameter 421
Parameters—ParameterDeclaration 421
Description. 422
Authorization. 423
Examples 423
FETCH 424
Scope 424
SQL Syntax 424
Parameters 424
SQL Syntax — BULK HostVariableSpecification 425
Parameters — BULK HostVariableSpecification 425
SQL Syntax — non-BULK HostVariableSpecification 425
Parameters — non-BULK HostVariableSpecification. 426
Description. 426
Authorization. 426
Examples 427
GENPLAN 429
Scope 429
SQL Syntax 429
Parameters 429
Description. 429
Authorization. 433
Examples 433
GOTO 435
Scope 435
SQL Syntax 435
Parameters 435
Description. 435
Authorization. 435
Example 435
GRANT 436
Scope 436
SQL Syntax — Grant Table or View Authority. 436
Parameters — Grant Table or View Authority 436
Authorization — Grant Table or View Authority 437
SQL Syntax — Grant RUN or EXECUTE Authority 437

Parameters — Grant RUN or EXECUTE Authority 438
Authorization — Grant RUN or EXECUTE Authority 438
SQL Syntax — Grant CONNECT, DBA, INSTALL, MONITOR, or RESOURCE Authority . 438
Parameters — Grant CONNECT, DBA, INSTALL, MONITOR, or RESOURCE Authority . . 438
Description — Grant CONNECT, DBA, INSTALL, MONITOR, or RESOURCE Authority . . 439
Authorization — Grant CONNECT, DBA, INSTALL, MONITOR, or RESOURCE Authority 439
SQL Syntax — Grant DBEFileSet Authority 439
Parameters — Grant DBEFileSet Authority 439
Description 440
Authorization — Grant DBEFilesSet Authority 440
Examples. 440
IF 442
Scope 442
SQL Syntax. 442
Parameters 442
Description 442
Authorization 442
Example 443
INCLUDE 444
Scope 444
SQL Syntax. 444
Parameters 444
Description 444
Authorization 444
Example 444
INSERT 445
Scope 445
SQL Syntax - Insert Rows with Defined Values 445
Parameters - Insert Rows with Defined Values 445
SQL Syntax — SingleRowValues 446
Parameters — SingleRowValues 446
SQL Syntax — LongColumnIOString. 448
Parameters — LongColumnIOString 448
Description — LongColumnIOString 448
SQL Syntax — BulkValues 450
Parameters — BulkValues. 450
Description — Insert Rows with SingleRowValues and BulkValues 450
SQL Syntax — DynamicParameterValues 452
Parameters — DynamicParameterValues 452
Description — Insert Rows with DynamicParameterValues. 452
Authorization — Insert Rows with SingleRowValues and Bulk Values 453
SQL Syntax — INSERT Rows Defined by a SELECT Command (Type 2 Insert) 453
Parameters — INSERT Rows Defined by a SELECT Command (Type 2 Insert) 453
Description — INSERT Rows Defined by a SELECT Command (Type 2 Insert). 454
Authorization — INSERT Rows Defined by a SELECT Command (Type 2 Insert). 455

Examples 456
Labeled Statement 458
Scope 458
SQL Syntax 458
Parameters 458
Description 458
Authorization 458
Example 458
LOCK TABLE 460
Scope 460
SQL Syntax 460
Parameters 460
Description 460
Authorization 461
Examples 461
LOG COMMENT 462
Scope 462
SQL Syntax 462
Parameters 462
Description 462
Authorization 462
Example 463
OPEN 464
Scope 464
SQL Syntax 464
Parameters 464
Description 465
PREPARE 466
Scope 466
SQL Syntax 466
Parameters 466
Description 468
Authorization 468
Examples 468
PRINT 471
Scope 471
SQL Syntax 471
Parameters 471
Description 472
Authorization 472
Examples 472
RAISE ERROR 474
Scope 474
SQL Syntax 474
Parameters 474
Description 475
Authorization 475
Examples 475
REFETCH 476

Scope 476
SQL Syntax 476
Parameters 476
Description 476
Authorization 477
Example 477
RELEASE 479
Scope 479
SQL Syntax 479
Description 479
Authorization 479
Example 479
REMOVE DBEFILE 480
Scope 480
SQL Syntax 480
Parameters 480
Description 480
Authorization 480
Example 480
REMOVE FROM GROUP 482
Scope 482
SQL Syntax 482
Parameters 482
Description 482
Authorization 482
Example 483
RENAME COLUMN 484
Scope 484
SQL Syntax 484
Parameters 484
Description 484
Authorization 484
Example 484
RENAME TABLE 485
Scope 485
SQL Syntax 485
Parameters 485
Description 485
Authorization 485
Example 485
RESET 486
Scope 486
SQL Syntax 486
Parameters 486
Description 486
Authorization 486
Example 486
RETURN 487
Scope 487

SQL Syntax 487
Parameters 487
Description. 487
Example 487
REVOKE 489
Scope 489
SQL Syntax — Revoke Table or View Authority. 489
Parameters — Revoke Table or View Authority 489
Description — Revoke Table or View Authority 490
Authorization — Revoke Table or View Authority 490
SQL Syntax — Revoke RUN or EXECUTE or Authority 491
Parameters--Revoke RUN or EXECUTE Authority 491
SQL Syntax — Revoke CONNECT, DBA, INSTALL, MONITOR, or RESOURCE Authority 491
Parameters — Revoke CONNECT, DBA, INSTALL, MONITOR, or RESOURCE Authority 491
Description — Revoke CONNECT, DBA, INSTALL, MONITOR, or RESOURCE Authority 492
Authorization — Revoke CONNECT, DBA, INSTALL, MONITOR, or RESOURCE Authority 492
SQL Syntax — Revoke DBEFileSet Authority 492
Parameters — Revoke DBEFileSet Authority. 492
Description — Revoke DBEFileSet Authority. 492
Authorization — Revoke DBEFileSet Authority. 493
Examples 493
ROLLBACK WORK 495
Scope 495
SQL Syntax 495
Parameters 495
Description. 495
Authorization. 496
Example 496

12. SQL Statements S - Z

SAVEPOINT 497
Scope 497
SQL Syntax 497
Parameters 497
Description. 498
Authorization. 498
Example 498
SELECT 499
Scope 499
SQL Syntax — Select Statement Level 499
SQL Syntax — Subquery Level 499
SQL Syntax — Query Expression Level 499
SQL Syntax — Query Block Level 499
SelectList 499
HostVariableSpecification — With BULK Option. 499

HostVariableSpecification — Without BULK Option. 500
FromSpec 500
TableSpec 500
SQL Syntax — Select Statement Level. 500
Parameters — Select Statement Level 500
Description — Select Statement Level 501
SQL Syntax — Subquery Level. 501
Parameters — Subquery Level 502
Description — Subquery Level 502
SQL Syntax — Query Expression Level 503
Parameters — Query Expression Level 503
Description — Query Expression Level 503
SQL Syntax — Query Block Level. 505
Parameters — Query Block Level 505
SQL Syntax — SelectList. 507
Parameters — SelectList 507
SQL Syntax — BULK HostVariableSpecification 508
Parameters — BULK HostVariableSpecification 508
SQL Syntax — non-BULK HostVariableSpecification 508
Parameters — non-BULK HostVariableSpecification 508
SQL Syntax — FromSpec 509
Parameters — FromSpec 509
Description — Query Block Level 511
Authorization 515
Examples. 515
SET CONNECTION 519
Scope 519
SQL Syntax. 519
Parameters 519
Description 519
Authorization 519
Example 520
SET CONSTRAINTS. 521
Scope 521
SQL Syntax. 521
Parameters 521
Description 521
Authorization 522
Example 522
SET DEFAULT DBEFILESET 524
Scope 524
SQL Syntax. 524
Parameters 524
Description 524
Authorization 525
Example 525
SET DML ATOMICITY. 526
Scope 526
SQL Syntax. 526

Parameters . . . 526
Description . . . 526
Authorization . . . 527
Example . . . 527
SET MULTITRANSACTION . . . 529
Scope . . . 529
SQL Syntax . . . 529
Parameters . . . 529
Description . . . 529
Authorization . . . 529
Example . . . 530
SETOPT . . . 531
Scope . . . 531
Syntax — SETOPT . . . 531
Syntax — Scan Access . . . 531
Syntax — Join Algorithm . . . 531
Parameters . . . 531
Description . . . 532
Authorization . . . 532
Examples . . . 532
SET PRINTRULES . . . 534
Scope . . . 534
SQL Syntax . . . 534
Parameters . . . 534
Description . . . 534
Authorization . . . 535
Example . . . 535
SET SESSION . . . 536
Scope . . . 536
SQL Syntax . . . 536
Parameters . . . 536
Description . . . 538
Authorization . . . 541
Example . . . 541
SET TRANSACTION . . . 542
Scope . . . 542
SQL Syntax . . . 542
Parameters . . . 542
Description . . . 544
Authorization . . . 546
Example . . . 547
SET USER TIMEOUT . . . 548
Scope . . . 548
SQL Syntax . . . 548
Parameters . . . 548
Description . . . 548
Authorization . . . 549
Example . . . 549
SQLEXPLAIN . . . 550

Scope 550
SQL Syntax 550
Parameters 550
Description 550
Authorization 550
Example 551
START DBE 552
Scope 552
SQL Syntax 552
Parameters 552
Description 553
Authorization 554
Example 554
START DBE NEW 555
Scope 555
SQL Syntax — START DBE NEW 555
Parameters — START DBE NEW 555
SQL Syntax — DBEFile0Definition 559
Parameters — DBEFile0Definition 559
SQL Syntax — DBELogDefinition 560
Parameters — DBELogDefinition 560
Description 560
Authorization 562
Example 562
START DBE NEWLOG 563
Scope 563
SQL Syntax — START DBE NEWLOG 563
Parameters — START DBE NEWLOG 563
SQL Syntax — NewLogDefinition 567
Parameters — NewLogDefinition 567
Description 567
Authorization 569
Example 569
STOP DBE 571
Scope 571
SQL Syntax 571
Description 571
Authorization 571
Example 571
TERMINATE QUERY 572
Scope 572
SQL Syntax 572
Parameters 572
Description 572
Authorization 572
Example 572
TERMINATE TRANSACTION 573
Scope 573
SQL Syntax 573

Parameters . . . 573
Description . . . 573
Authorization . . . 573
Example . . . 573
TERMINATE USER . . . 574
Scope . . . 574
SQL Syntax . . . 574
Parameters . . . 574
Description . . . 574
Authorization . . . 574
Example . . . 575
TRANSFER OWNERSHIP . . . 576
Scope . . . 576
SQL Syntax . . . 576
Parameters . . . 576
Description . . . 576
Authorization . . . 576
Example . . . 577
TRUNCATE TABLE . . . 578
Scope . . . 578
SQL Syntax . . . 578
Parameters . . . 578
Description . . . 578
Authorization . . . 578
Example . . . 579
UPDATE . . . 580
Scope . . . 580
SQL Syntax . . . 580
Parameters . . . 580
Description . . . 580
SQL Syntax — LongColumnIOString . . . 582
Parameters — LongColumnIOString . . . 582
Description — LongColumnIOString . . . 583
Authorization . . . 584
Example . . . 584
UPDATE STATISTICS . . . 585
Scope . . . 585
SQL Syntax . . . 585
Parameters . . . 585
Description . . . 585
Authorization . . . 586
Example . . . 586
UPDATE WHERE CURRENT . . . 587
Scope . . . 587
SQL Syntax . . . 587
Parameters . . . 587
Description . . . 588
SQL Syntax — LongColumnIOString . . . 589
Parameters — LongColumnIOString . . . 589

Description — LongColumnIOString 590
Authorization 591
Example 591
VALIDATE 592
Scope 592
SQL Syntax 592
Parameters 592
Description 592
Authorization 593
Examples 594
WHENEVER 595
Scope 595
SQL Syntax 595
Parameters 595
Description 595
Authorization 596
Example 596
WHILE 597
Scope 597
SQL Syntax 597
Parameters 597
Description 597
Authorization 597
Example 597

A. SQL Syntax Summary
. 599
ADD DBEFILE 599
ADD TO GROUP 599
ADVANCE 599
ALTER DBEFILE 599
ALTER TABLE 599
Assignment (=) 600
BEGIN 600
BEGIN ARCHIVE 600
BEGIN DECLARE SECTION 600
BEGIN WORK 600
CHECKPOINT 601
CLOSE 601
COMMIT ARCHIVE 601
COMMIT WORK 601
CONNECT 601
CREATE DBEFILE 601
CREATE DBEFILESET 601
CREATE GROUP 601
CREATE INDEX 601
CREATE PARTITION 602
CREATE PROCEDURE 602
CREATE RULE 602

CREATE SCHEMA 602
CREATE TABLE 603
CREATE TEMPSPACE 603
CREATE VIEW 604
DECLARE CURSOR 604
DECLARE Variable 604
DELETE 604
DELETE WHERE CURRENT 604
DESCRIBE 604
DISABLE AUDIT LOGGING 604
DISABLE RULES 604
DISCONNECT 605
DROP DBEFILE 605
DROP DBEFILESET 605
DROP GROUP 605
DROP INDEX 605
DROP MODULE 605
DROP PARTITION 605
DROP PROCEDURE 605
DROP RULE 605
DROP TABLE 605
DROP TEMPSPACE 605
DROP VIEW 605
ENABLE AUDIT LOGGING 606
ENABLE RULES 606
END DECLARE SECTION 606
EXECUTE 606
EXECUTE IMMEDIATE 606
EXECUTE PROCEDURE 606
FETCH 606
GENPLAN 607
GOTO 607
GRANT 607
IF 608
INCLUDE 608
INSERT - 1 608
INSERT - 2 609
Labeled Statement 609
LOCK TABLE 609
LOG COMMENT 609
OPEN 609
PREPARE 610
PRINT 610
RAISE ERROR 610
REFETCH 610
RELEASE 610
REMOVE DBEFILE 610
REMOVE FROM GROUP 610
RENAME COLUMN 610

RENAME TABLE 610
RESET 610
RETURN 611
REVOKE 611
ROLLBACK WORK 611
SAVEPOINT 612
SELECT 612
SET CONNECTION 613
SET CONSTRAINTS 613
SET DEFAULT DBEFILESET 613
SET DML ATOMICITY 613
SET MULTITRANSACTION 613
SETOPT 613
SET PRINTRULES 614
SET SESSION 614
SET TRANSACTION 615
SET USER TIMEOUT 615
SQLEXPLAIN 615
START DBE 616
START DBE NEW 616
START DBE NEWLOG 617
STOP DBE 617
TERMINATE QUERY 617
TERMINATE TRANSACTION 617
TERMINATE USER 618
TRANSFER OWNERSHIP 618
TRUNCATE TABLE 618
UPDATE 618
UPDATE STATISTICS 618
UPDATE WHERE CURRENT 618
VALIDATE 619
WHENEVER 619
WHILE 619

B. ISQL Syntax Summary
. 621
CHANGE 621
DO 621
EDIT 621
END 621
ERASE 621
EXIT 621
EXTRACT 621
HELP 621
HOLD 622
INFO 622
INPUT 622
INSTALL 622
LIST FILE 622

LIST HISTORY 622
LIST INSTALL 622
LIST SET 622
LOAD 622
RECALL 623
REDO 623
RENAME 623
SELECTSTATEMENT 623
SET 623
SQLGEN 624
SQLUTIL 624
START 624
STORE 624
SYSTEM 624
UNLOAD 624

C. Sample DBEnvironment
Installing the Files for PartsDBE 626
Setting Up PartsDBE 627
Using SQLSetup 627
Creating PartsDBE 628
Using Setup 628
Listings of ISQL Command Files 629
STARTDBE Command File 630
CREATABS Command File 631
LOADTABS Command File 635
CREAINDEX Command File 638
CREASEC Command File 639
Data in the Sample DBEnvironment 645
ManufDB.SupplyBatches Table 646
ManufDB.TestData Table 647
PurchDB.Inventory Table 648
PurchDB.OrderItems Table 650
PurchDB.Orders Table 653
PurchDB.Parts Table 654
PurchDB.Reports Table 655
PurchDB.SupplyPrice Table 656
PurchDB.Vendors Table 659
RecDB.Clubs Table 661
RecDB.Events Table 662
RecDB.Members Table 663
Sample Program Files 664

D. Standards Flagging Support
Introduction 665
Non-standard Statements and Extensions 666
Non-Standard Data Types 675
Non-Standard Expression Extensions 676

Non-Standard Syntax Rules 677

Figures

Figure 1-1. . Components of ALLBASE/SQL 42
Figure 1-2. . How Tables, DBEFiles, and DBEFileSets Are Related 46
Figure 1-3. . Databases and DBEFileSets 46
Figure 1-4. . Elements of an ALLBASE/SQL DBEnvironment 47
Figure 3-1. . Range of Complex Query Types. 116
Figure 4-1. . Referential Constraints in a Set of Tables 142
Figure 5-1. . Transactions over Time 168
Figure 5-2. . Multiuser DBEnvironment 169
Figure 5-3. . Page Versus Table Level Locking 177
Figure 5-4. . Row Versus Page Level Locking 178
Figure 5-5. . Locks at Different Granularities 180
Figure 5-6. . Scope and Duration of Share Locks for Different Isolation Levels 188
Figure 5-7. . Lock Requests 1: Waiting for Exclusive Lock 190
Figure 5-8. . Lock Requests 2: Waiting for Share Locks 190
Figure 5-9. . Lock Requests 3: Share Locks Granted 191
Figure 5-10. . Deadlock 196
Figure 9-1. . Logical Operations on Predicates Containing NULL Values 263
Figure C-1. . SQLSetup Menu 627

Figures

Tables

Tables

Preface

This manual contains basic information about ALLBASE/SQL as well as in-depth information about ALLBASE/SQL data types and statements. The first three chapters are for all readers, including new users of ALLBASE/SQL. The remaining chapters are for experienced SQL users and SQL application programmers. The titles of the chapters are as follows:

- Chapter 1 , “Introduction,” presents the components of ALLBASE/SQL and introduces fundamental ALLBASE/SQL concepts and terms.
- Chapter 2 , “Using ALLBASE/SQL,” describes basic ALLBASE/SQL usage rules.
- Chapter 3 , “SQL Queries,” presents a full treatment of queries, including the use of subqueries, UNION, and special predicates.
- Chapter 4 , “Constraints, Procedures, and Rules,” presents data objects which provide a high degree of data consistency and integrity inside the DBEnvironment.
- Chapter 5 , “Concurrency Control through Locks and Isolation Levels,” describes ways of managing concurrent database transactions.
- Chapter 6 , “Names,” presents general rules for names used in ALLBASE/SQL statements.
- Chapter 7 , “Data Types,” details the data types available in ALLBASE/SQL.
- Chapter 8 , “Expressions,” describes ALLBASE/SQL expressions.
- Chapter 9 , “Search Conditions,” presents the basic syntax of ALLBASE/SQL predicates.
- Chapter 10 , “SQL Statements A - D,” contains an alphabetical reference of all the SQL statements and other elements of syntax.
- Chapter 11 , “SQL Statements E - R,” contains an alphabetical reference of all the SQL statements and other elements of syntax.
- Chapter 12 , “SQL Statements S - Z,” contains an alphabetical reference of all the SQL statements and other elements of syntax.

The appendixes contain additional reference information as follows:

- contains an alphabetical summary of all ALLBASE/SQL statements and other elements of syntax.
- contains an alphabetical summary of all ALLBASE/ISQL commands.
- describes the sample DBEnvironment, PartsDBE, which is supplied with the product. An explanation is provided of how to install, and set up a copy of PartsDBE for practice use.
- contains information about ALLBASE/SQL FIPS 127.1 compliance.

Most of the examples in this manual are based on the tables, views, and other objects in the sample DBEnvironment PartsDBE. For complete information about PartsDBE, refer to appendix C.

ALLBASE Manuals

The following is a list of the documentation titles for this MPE release of ALLBASE.

- *ALLBASE/ISQL Reference Manual*
- *ALLBASE/NET User's Guide*
- *ALLBASE/SQL Advanced Application Programming Guide*
- *ALLBASE/SQL C Application Programming Guide*
- *ALLBASE/SQL COBOL Application Programming Guide*
- *ALLBASE/SQL Database Administration Guide*
- *ALLBASE/SQL FORTRAN Application Programming Guide*
- *ALLBASE/SQL Message Manual*
- *ALLBASE/SQL Pascal Application Programming Guide*
- *ALLBASE/SQL Performance and Monitoring Guidelines*
- *ALLBASE/SQL Reference Manual*
- *HP PC API User's Guide for ALLBASE/SQL*
- *Up and Running with ALLBASE/SQL*
- *ODBCLINK/SE Reference Manual*

New Features in G1, G2 and G3

The following table highlights the new or changed functionality added in G1, G2 and G3 releases, and shows you where each feature is documented.

Ver.	Feature (Category)	Description	Documented in...
G3	String Functions (Usability)	The supported SQL syntax has been enhanced to include the following string manipulation functions: `UPPER, LOWER, POSITION, INSTR, TRIM, LTRIM, AND RTRIM`. These string functions allow you to manipulate or examine the `CHAR` and `VARCHAR` values within the SQL syntax, allowing for more sophisticated queries and manipulation commands to be formed.	In future version of the *ALLBASE/SQL Reference Manual*.
G2	Allow or disallow `SQLMON` for users. (Usability)	Grants or revokes the ability to run `SQLMON` for specific users. New attribute for `GRANT` and `REVOKE: MONITOR`.	*ALLBASE/SQL Reference Manual*, GRANT, REVOKE in "SQL Statements."
G2	Allow or disallow authority to create modules. (Usability)	Grants or revokes the ability to create modules for specific users. New attributes for `GRANT` and `REVOKE: INSTALL`.	*ALLBASE/SQL Reference Manual*, `GRANT, REVOKE` in "SQL Statements."
G2	PC ODBC 16-bit and 32-bit support (Connectivity, Client/Server)	ODBCLINK/SE allows connectivity to ALLBASE and IMAGE/SQL servers from a PC running MS Windows using ODBC.	*ODBCLINK/SE Reference Manual*
G2	Year 2000 solution (Standards)	Provides the JCW `HPSQLSPLITCENTURY` to use in setting a value between 0 and 99. This value is used to change the century part of the `DATE` and `DATETIME` functions to override the default of 19.	"Date/Time Functions" in the "Expressions" chapter of the *ALLBASE/SQL Reference Manual*.
G1	New operand to concatenate strings (Standards)	Adds an operand to concatenate character or binary strings in an expression. New operand: `||`	*ALLBASE/SQL Reference Manual*, "Expressions."

Ver.	Feature (Category)	Description	Documented in...
G1	RENAME Column or Table (Usability)	Adds capability of defining a new name for an existing table or column in a DBEnvironment. You cannot rename a table or column that has check constraints or an IMAGE/SQL table. New commands: RENAME COLUMN, RENAME TABLE.	*ALLBASE/SQL Reference Manual*, RENAME COLUMN and RENAME TABLE in "SQL Statements."
G1	CAST function added to Expression syntax (Usability)	Adds the CAST function to allow explicitly converting from one data type to another. It allows conversion between compatible data types and between normally incompatible data types such as CHAR and INTEGER. New Expression function: *CastFunction*.	*ALLBASE/SQL Reference Manual*, "Cast" in "Expressions."
G1	Syntax added to VALIDATE (Usability, Performance)	Automates execution of COMMIT WORK after each module or procedures is validated when WITH AUTOCOMMIT is used. All sections are revalidated whether valid or invalid when FORCE is used. This can reduce log space and shared memory requirements for the VALIDATE statement. New syntax for VALIDATE: FORCE, WITH AUTOCOMMIT.	*ALLBASE/SQL Reference Manual*, VALIDATE in "SQL Statements."
G1	Syntax added to DELETE (Usability, Performance)	Automates execution of COMMIT WORK at the beginning of the DELETE and after each batch of rows is deleted when WITH AUTOCOMMIT is used. Reduces log-space and shared-memory requirements. WITH AUTOCOMMIT cannot be used in some cases (see the DELETE statement). New syntax for DELETE: WITH AUTOCOMMIT.	*ALLBASE/SQL Reference Manual*, DELETE in "SQL Statements."
G1	Decimal operations (Usability)	Increases maximum precision from 18 to 27.	*ALLBASE/SQL Reference Manual*, "Decimal Operations" in "Data Types."

Ver.	Feature (Category)	Description	Documented in...
G1	Terminate a query (Usability, Performance)	Allows termination of a query for a connection or transaction. New statement: TERMINATE QUERY. New syntax for SET SESSION, SET TRANSACTION.	*ALLBASE/SQL Reference Manual*, TERMINATE QUERY, SET SESSION, SET TRANSACTION in "SQL Statements."
G1	Terminate a transaction (Usability, Performance)	Allows stopping of a given transaction. New statement: TERMINATE TRANSACTION. New syntax for SET SESSION, SET TRANSACTION.	*ALLBASE/SQL Reference Manual*, TERMINATE TRANSACTION, SET SESSION, SET TRANSACTION in "SQL Statements."
G1	Timeout enhanced to allow specifying what is rolled back or terminated (Usability, Performance)	Allows specifying the action when a timeout expires. New attributes for SET SESSION and SET TRANSACTION: TERMINATION AT LEVEL, USER TIMEOUT, ON TIMEOUT ROLLBACK.	*ALLBASE/SQL Reference Manual*, in "SQL Statements."
G1	New SQLUtil command CHKPTHLP reduces time for flushing data (Performance)	Flushes the data in parallel to the CHECKPOINT command in ISQL. New SQLUtil command: CHKPTHLP.	*ALLBASE/SQL Database Administration Guide*, CHKPTHLP in "SQLUtil"
G1	Allow or disallow SQLMON for users. (Usability)	Grants or revokes the ability to run SQLMON for specific users. New attribute for GRANT and REVOKE: MONITOR.	*ALLBASE/SQL Reference Manual*, GRANT, REVOKE in "SQL Statements."
G1	Allow or disallow authority to create modules. (Usability)	Grants or revokes the ability to create modules for specific users. New attributes for GRANT and REVOKE: INSTALL.	*ALLBASE/SQL Reference Manual*, GRANT, REVOKE in "SQL Statements."
G1	Script for migration to a new release (Usability, Tools)	Provides SQLLINSTL script for migration to a new release of ALLBASE/SQL. Read the SQLINSTL file on your system for more information.	SQLINSTL file; *Communicator 3000 MPE/iX Release 5.5 (Non-Platform Software Release C.55.00)*, "ALLBASE/SQL Enhancements"; *ALLBASE/SQL Database Administration Guide* in 'SQLINSTL" section of the "DBA Tasks and Tools" chapter.
G1	GENPLAN on a section (Usability)	Obtains an access plan of a stored static query by specifying the module and section number. Changed syntax: GENPLAN.	*ALLBASE/SQL Reference Manual*, GENPLAN in "SQL Statements."

Ver.	Feature (Category)	Description	Documented in...
G1	POSIX support (Tools)	Starting with G1, the ALLBASE/SQL preprocessor (PSQLCOB) supports preprocessing and generation of Microfocus COBOL source code under POSIX (Portable Operating system Interface).	*Communicator 3000 MPE/iX Release 5.5 (Non-Platform Software Release (C.55.00),* "ALLBASE/SQL Enhancements."
G1	Terminate a user's connections (Connectivity)	Terminates one or more connections for a user. New syntax for `TERMINATE USER`: CID *ConnectionID*.	*ALLBASE/SQL Reference Manual*, `TERMINATE USER` in "SQL Statements."
	Run Queue Control for ALLBASE/NET (Connectivity)	Allows running `HPDADVR` in D queue for an MPE/iX session or HP-UX connection or C queue for an MPE/iX job connection. New environment variable: `HPSQLJOBTYPE`.	*Communicator 3000 MPE/iX Release 5.5 (Non-Platform Software Release C.55.00),*"ALLBASE/SQL Enhancements."
	PC ODBC 16-bit and 32-bit support (Connectivity, Client/Server)	ODBCLINK/SE allows connectivity to ALLBASE and IMAGE/SQL servers from a PC running MS Windows using ODBC.	*ODBCLINK/SE Reference Manual*
	Year 2000 solution (Standards)	Provides the JCW `HPSQLSPLITCENTURY` to use in setting a value between 0 and 99. This value is used to change the century part of the `DATE` and `DATETIME` functions to override the default of 19.	"Date/Time Functions" in the "Expressions" chapter of the *ALLBASE/SQL Reference Manual*.

1 Introduction

This manual describes ALLBASE/SQL, which you use to create, maintain, and access relational database environments. **SQL** stands for **Structured Query Language**, a language for accessing a relational database.

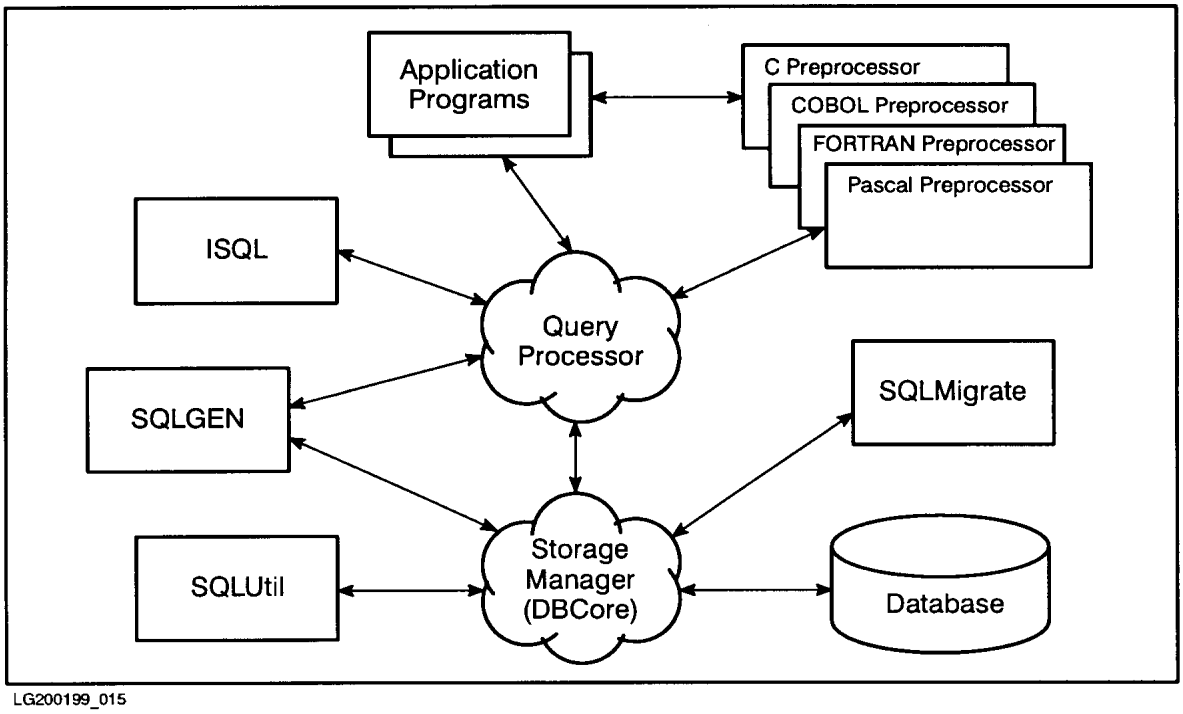
In order to define terms and provide an overview of the subject, this chapter includes the following sections:

- ALLBASE/SQL Components
- ALLBASE/SQL Databases
- ALLBASE/SQL Data Access
- Using Queries
- ALLBASE/SQL Objects
- ALLBASE/SQL Users
- Using Comments within SQL Statements
- SQL Language Structure
- SQL Statement Categories
- Error Conditions in ALLBASE/SQL
- Native Language Support

ALLBASE/SQL Components

ALLBASE/SQL consists of several distinct components, which are shown in Figure 1-1..

Figure 1-1. Components of ALLBASE/SQL



To access data with ALLBASE/SQL, you use ALLBASE/SQL statements, which conform to industry standards for SQL statements for relational databases.

You can submit SQL statements interactively or in application programs as described here:

- Interactively, you use **ISQL** (Interactive SQL) to key in statements at a terminal. ISQL is the interactive interface to ALLBASE/SQL.
- Programmatically, you embed statements in a C, COBOL, FORTRAN, or Pascal application program. Then, before compiling the program, you use an ALLBASE/SQL **preprocessor** to prepare the program for run-time database access. The preprocessor converts an embedded SQL program into a source file for input to a C, COBOL, FORTRAN, or Pascal compiler.

As SQL statements come from ISQL or from the preprocessors, they are passed along to the two following subsystems:

- **Query Processor** checks the syntax of each statement, verifies that the user has the appropriate authorization for it, and processes queries.
- **Storage Manager** performs physical file management, and transaction and logging tasks. The Storage Manager is also referred to as DBCore.

Utility Programs

In addition, these utility programs help you perform the necessary maintenance tasks:

- **SQLUtil** assists with file maintenance, backup, and recovery.
- **SQLGEN** generates statements for re-creating a given DBEnvironment.
- **SQLMigrate** lets you move DBEnvironments between releases of ALLBASE/SQL.
- **SQLCheck** checks the integrity of a DBEnvironment.
- **SQLMON** helps you monitor DBEnvironment performance.
- **SQLVer** checks the version strings of the ALLBASE/SQL files.
- **SQLAudit** organizes audit log records for analysis of operations such as `UPDATE`, `INSERT`, or `DELETE`, perhaps for security reasons.

The utility programs listed that are not included in Figure 1-1 all interact with the Storage Manager (DBCore).

ISQL is described in the *ALLBASE/ISQL Reference Manual*. The preprocessors are documented in separate ALLBASE/SQL application programming guides for each language and the release specific *ALLBASE/SQL Advanced Application Programming Guide*.

`SQLUtil`, `SQLGEN`, `SQLMigrate`, `SQLCheck`, `SQLVer`, and `SQLAudit` are documented in the *ALLBASE/SQL Database Administration Guide*. `SQLMON` is documented in the *ALLBASE/SQL Performance and Monitoring Guidelines*. The rest of this manual describes SQL, pointing out differences between interactive and programmatic usage when they exist. Most of the SQL statements can be executed through either interface.

ALLBASE/SQL Databases

The largest unit in ALLBASE/SQL is the **DBEnvironment**, which can be seen logically as a collection of database **objects** or physically as a group of files. Objects are database structures.

Logical Concepts

Logically, the DBEnvironment is a structure which contains one or more relational databases. In ALLBASE/SQL, a **database** is a set of tables, views, and other objects that have the same owner.

The data in a **relational** database is organized in tables. A **table** is a two-dimensional structure of columns and rows:

```
                        The Parts Table
---------------+-----------------------------+----------
PARTNUMBER     |PARTNAME                     |SALESPRICE
---------------+-----------------------------+---------- _
1123-P-01      |Central Processor            |    500.00  |
1133-P-01      |Communication Processor      |    200.00  |- rows
1143-P-01      |Video Processor              |    180.00 _|
      |                      |                      |
      +----------------------+----------------------+
                             |
                          columns
```

Often a table is referred to as a **relation**, and a row as a **tuple**. You can also think of a row as a record, and a column as a field in a file, or table.

A **view** is a table derived by placing a "window" over one or more tables to let users or programs view only certain data. A view derived from the Parts table shown above might look like this:

```
                The PartsID View
---------------+-----------------------------
PARTNUMBER     |PARTNAME
---------------+-----------------------------
1123-P-01      |Central Processor
1133-P-01      |Communication Processor
1143-P-01      |Video Processor
```

The **owner** of a table or view can be one of the three following entities:

1. **Individual** as identified by the DBEUserID, which is the login name. An individual who logs in as *WOLFGANG.ACCOUNTNAME* is known to ALLBASE/SQL as WOLFGANG@ACCOUNTNAME.
2. **Authorization group**, a named collection of individuals or other groups. Wolfgang might be part of a group named *Managers*. A group must be created explicitly by using the CREATE GROUP statement.
3. **Class**, a name that identifies a user-defined abstraction, such as a department or a function. Wolfgang might use tables owned by a class called *Marketing*. A class is

created implicitly when you create objects that have a class name as owner name.

Refer to Chapter 2 , “Using ALLBASE/SQL,” in this manual and to the chapter “Logical Design” in the *ALLBASE/SQL Database Administration Guide* for additional information about authorization groups and classes.

To use data in a database, you need to specify the names of the tables and views you need. You must also specify the owner name associated with the table or view unless you own it (or you have used the `ISQL SET OWNER` command). When accessing the Composers table, Wolfgang needs to specify only Composers. However, when accessing the quotas table, he needs to specify Marketing.Quotas because Marketing owns the Quotas table.

You also need the proper **authority** to access data. An authority is a privilege given to a user to perform a specific database operation, such as accessing certain tables and views and creating groups or tables. ALLBASE/SQL uses authorities to safeguard databases from access by unauthorized users. In the example above, before Wolfgang can access the Quotas table, he must be granted the authority to do so by the owner of the table.

If you have been granted the proper authorization, you access databases by first *`connecting`* to the DBEnvironment in which they reside:

```
CONNECT TO 'DBEnvironmentName'
```

Physical Concepts

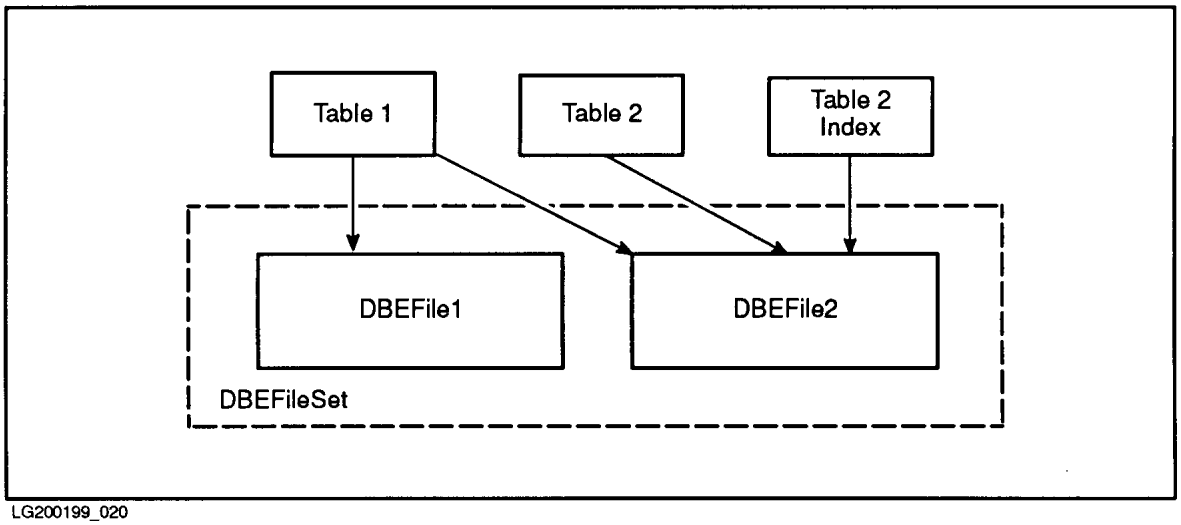
Physically, the DBEnvironment is a collection of files for one or more logical databases.

A DBEFile is an MPE XL file. Most files in a DBEnvironment are DBEFiles. Data in the tables of logical databases is stored in one or more DBEFiles. Indexes are also stored in DBEFiles; an index is a structure that ALLBASE/SQL can use to quickly find data in a table.

A **DBEFileSet** is a collection of DBEFiles. You associate physical storage with a DBEFileSet by adding DBEFiles to the DBEFileSet. Each DBEFileSet can have more than one DBEFile, but a single DBEFile cannot contain data for more than one DBEFileSet.

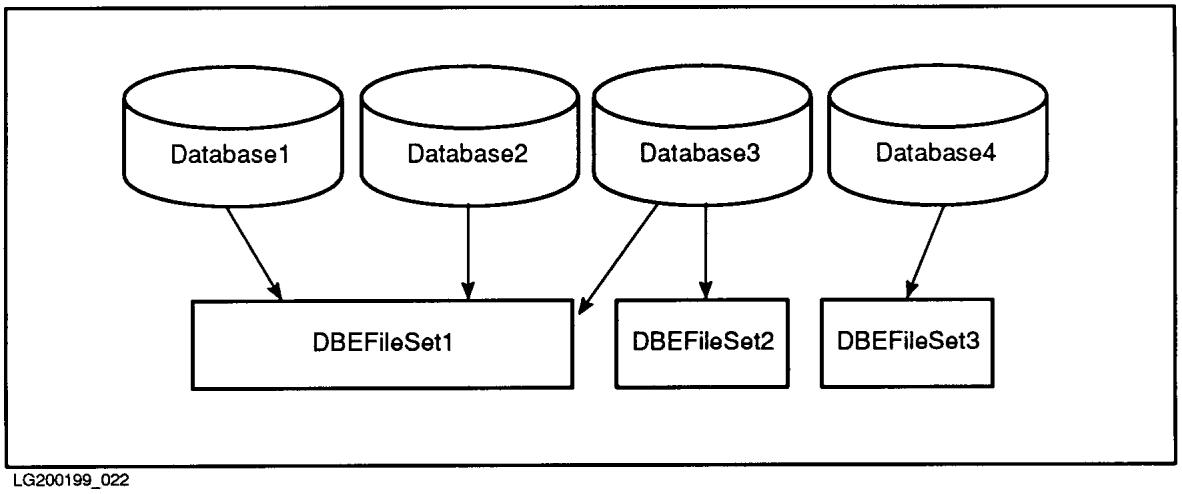
When you create a table, you can specify the DBEFileSet with which the table and its indexes will be associated. This causes physical storage space for the table and indexes to be allocated from the DBEFiles associated with the specified DBEFileSet. Figure 1-2. illustrates the relationships among tables, DBEFiles, and DBEFileSets.

Figure 1-2. How Tables, DBEFiles, and DBEFileSets Are Related



A DBEFileSet specifies the files that contain data for one or more tables associated with the DBEFileSet. These tables do not have to be in the same database. Figure 1-3. illustrates that, while a DBEFileSet can contain data for all the tables in a database, a DBEFileSet can also contain data for some of the tables in a database, or for tables in more than one database. Thus DBEFileSets offer a way to allocate data storage independently of how users think about the data.

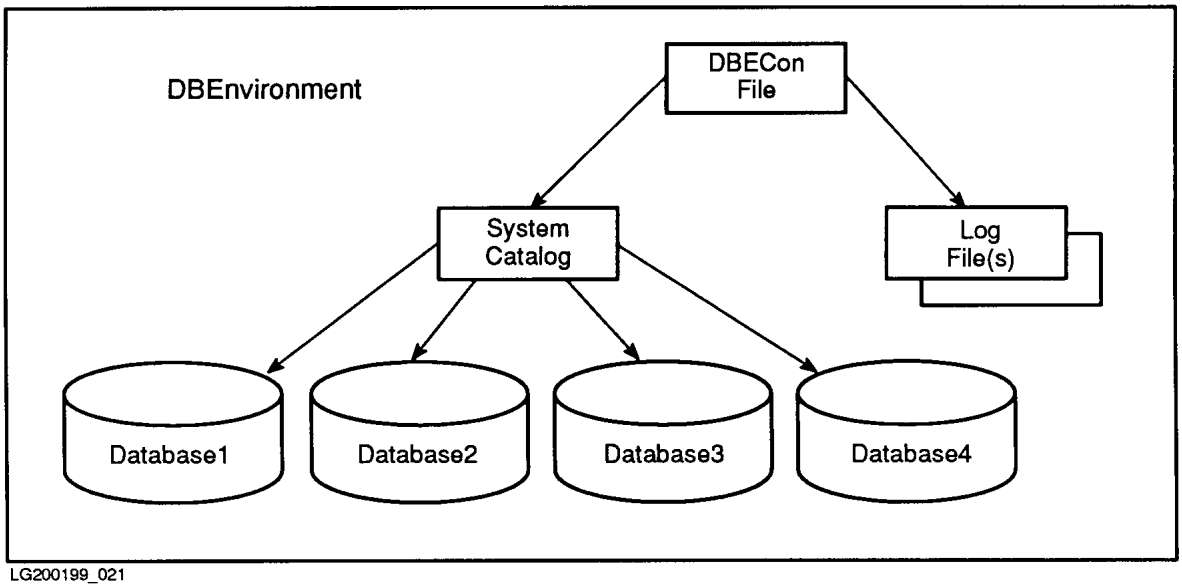
Figure 1-3. Databases and DBEFileSets



A DBEnvironment, illustrated in Figure 1-4., houses the DBEFiles for one or more ALLBASE/SQL databases, plus the following, which contain information for *all* databases in the DBEnvironment:

- A **DBECon file**. This file contains information about the DBEnvironment configuration, such as the size of various buffers and other startup parameters. The name of the DBECon file is the same as the name of the DBEnvironment.
- A **system catalog**. The system catalog is a collection of tables and views that contain data describing DBEnvironment structure and activity. The parts of the system catalog necessary for DBEnvironment startup reside in a DBEFile known by default as **DBEFile0**. All system catalog DBEFiles are associated with a DBEFileSet called **SYSTEM**.
- One or two **log files**. A log file contains a log of DBEnvironment changes. ALLBASE/SQL uses log files to undo (**roll back**) or redo (**roll forward**) changes made in the DBEnvironment. The log files are known by default as **DBELog1** and **DBELog2**.

Figure 1-4. Elements of an ALLBASE/SQL DBEnvironment



Most database users need not be concerned with the physical aspects of ALLBASE/SQL databases beyond knowing which DBEnvironment contains the databases they want to access.

ALLBASE/SQL Data Access

The DBEnvironment determines both what data can be accessed in a **transaction** and what data can be recovered. Following a failure, a transaction can be recovered, or all data can be recovered, as follows:

- A transaction is one or more SQL statements that together perform a unit of work on one or more databases in a DBEnvironment. Work done within a transaction can be made permanent (committed) or undone (rolled back).
- After a system or hardware failure, *all* data within a DBEnvironment is recovered to a consistent state. Changes performed by any transactions incomplete at failure time are rolled back. Changes performed by transactions completed before failure time are made permanent.

You can have more than one DBEnvironment on your system. When you connect to a DBEnvironment, ALLBASE/SQL establishes a **DBE session** for you. The query processor can process statements only when you are in a DBE session. You can access any DBEnvironment in either of the two following modes:

- **Single-user mode**—only one user or program can use a DBEnvironment.
- **Multiuser mode**—more than one user and/or program can use a DBEnvironment at the same time.

Using Queries

After connecting to a DBEnvironment, you use **queries** to retrieve data from database tables. A query is a statement in which you describe the data you want to retrieve. In ALLBASE/SQL, a query is performed by using the SELECT statement. For example:

```
SELECT PartName, SalesPrice
  FROM PurchDB.Parts
 WHERE PartNumber = '1123-P-01'
    OR PartNumber = '1133-P-01'
```

The result of a query is called a **query result**. In the case of the query above, which retrieves the name and selling price of two parts from the table named PurchDB.Parts, the result is a table made up of two columns and two rows:

```
----------------------------------------
PARTNAME                     |SALESPRICE
-----------------------------+----------
Central Processor            |    500.00
Communication Processor      |    200.00
```

A detailed presentation of queries and other forms of data manipulation appears in the “SQL Queries” chapter.

ALLBASE/SQL Objects

The following structures play a significant role in the use of an ALLBASE/SQL database and are known as database objects:

- Tables
- Views
- Columns (in tables and views)
- Authorization groups
- Indexes (on tables)
- Hash structures (for tables)
- Constraints
- Rules (on tables)
- Procedures
- DBEFiles
- DBEFileSets
- TempSpaces
- Modules

Many of the SQL statements let you create and then create and manipulate objects as described below:

- Data in tables and views
- Columns within tables and views
- Grant authorities to authorization groups
- Indexes for specific tables
- Hash structures for specific tables
- Constraints on specific tables, views, or columns
- Rules on specific tables
- Procedures containing SQL and control flow statements
- DBEFiles and associate them with DBEFileSets
- TempSpaces that are used for sorting
- Modules when you preprocess an application program containing SQL statements

ALLBASE/SQL Users

ALLBASE/SQL users fall into the three categories as described here. One person may do all the tasks within these categories.

- **Application programmers**. These users write application programs that access ALLBASE/SQL databases. They embed SQL statements in source code to manipulate data. Programmers then use the preprocessor that supports their programming language. The preprocessor prepares the application program for compilation and stores database access information in a **module** in the DBEnvironment; the stored module contains optimized data access paths that are used at run time. Once the program is compiled, authorized users can execute it.

  Application programmers also use ISQL throughout program development. DBEnvironments for testing and running applications can be created via ISQL. You can determine the effect of many SQL statements by using ISQL.

- **Database administrators**. These individuals, referred to as DBAs, are responsible for the creation and maintenance of ALLBASE/SQL DBEnvironments. They use SQL statements, usually via ISQL, to perform the following tasks:

  — Define DBEnvironments, grant and revoke authorities, add and drop DBEFiles, alter tables, define indexes, and define views using SQL, ISQL, or preprocessed programs.

  — Alter the configuration of a DBEnvironment, move or purge DBEFiles, and back up DBEnvironments using `SQLUtil`.

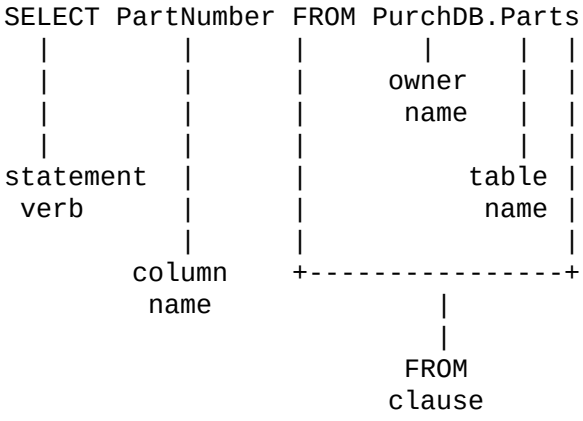
  — Access information in the system catalog to monitor DBEnvironment usage and help ensure efficient access to data.

  — Re-create all or part of a DBEnvironment on a different system by using `SQLGEN`.

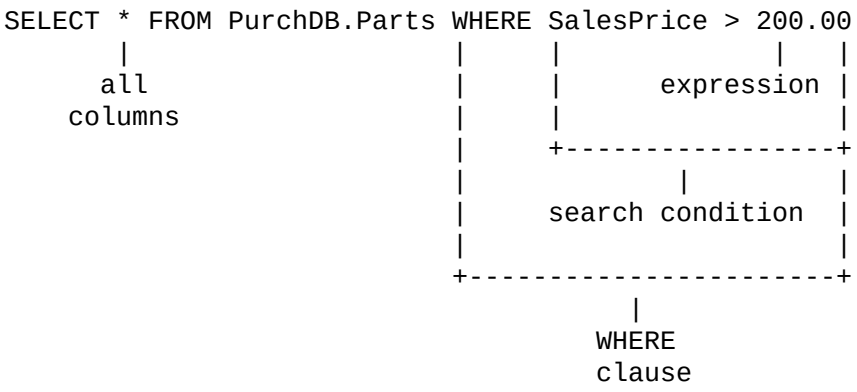
- **End users**. These users run application programs that access ALLBASE/SQL databases. They do not need to be aware of the components of ALLBASE/SQL in many cases. These users may occasionally use ISQL to issue simple SQL statements that retrieve or change data. Relational databases are particularly well-suited for data access of this nature, because you can access data without specifying specific access paths. End users only need to know table and column names.

SQL Language Structure

SQL statements begin with a verb and can include clauses or names. For example:



Statements always contain a verb, one or more words that describe the action of the statement. A statement can also contain one or more clauses. A clause is a group of names and keywords describing what the verb should operate on. A verb can operate on a *named* object, such as a table or a column. Some statements can contain expressions or search conditions. Expressions specify a value. Search conditions screen data against specific criteria:



The syntax of SQL is fully described in chapters 7-12 of this manual.

Using Comments within SQL Statements

You can initiate comments within any SQL statement or ISQL prompt either by prefixing each line of the comment with two hyphens or with the combination of slashes and asterisks at the beginning and end of the comments:

```
SELECT *
  FROM PurchDB.SupplyPrice
 WHERE PartNumber = '1723-AD-01'
   AND DeliveryDays < 30

--This statement selects values from the SupplyPrice table based on
--part number and delivery days.

SELECT *
  FROM PurchDB.SupplyPrice
 WHERE PartNumber = '1723-AD-01'
   AND DeliveryDays < 30

/*This statement selects values from the SupplyPrice table based on*/
/*part number and delivery days.*/
```

SQL Statement Categories

Writing queries is the basis of data manipulation in ALLBASE/SQL. All users employ the SELECT statement for this purpose. SQL has several other general-purpose statements, and also has statements specifically for use by application programmers or database administrators. The SQL statements are functionally summarized inTable 1-1. For the commands in each category, refer to Table 10-1., “SQL Statement Summary.”

Table 1-1. SQL Statement Categories

Group	Category	Purpose
General-purpose statements	DBEnvironment session management	Statements for obtaining and terminating database access.
	Data definition	Statements for defining tables, views, indexes, DBEFiles, DBEFileSets, TempSpace, and other SQL objects.
	Data manipulation	Statements for selecting, inserting, and changing rows.
	Transaction management	Statements for committing or rolling back work done within a single transaction. A transaction is a unit of work and may consist of one or multiple SQL statements.
	Concurrency	Statements for managing data contention in multiuser mode.
	Module Maintenance	Statements for managing modules and procedures.
Application programming statements	Single row data manipulation	Statements for manipulating a single row with each statement execution.
	Bulk data manipulation	Statements for manipulating multiple rows with a single statement execution.
	Cursor management	Statements for manipulating individual rows in a set of rows that satisfy a SELECT statement.
	Preprocessor directives	Statements for declarations in application programming.
	Dynamically preprocessed queries	Statements for handling statements preprocessed at run time.
	Status messages	A statement for retrieving an ALLBASE/SQL message describing the status of an SQL statement execution.

Table 1-1. SQL Statement Categories

Group	Category	Purpose
Database administration statements	Authorization	Statements for controlling DBEnvironment access.
	DBEnvironment configuration and use	Statements for controlling DBEnvironments.
	Space management	Statements for managing DBEFiles used for tables and indexes; statements for managing temporary space for sorting.
	Logging	Statements for managing log files.
	DBEnvironment statistics management	Statements related to the system catalog.
	Procedure control flow statements	Statements used only inside procedures.
Procedure statements	General and Control Flow Statements	Statements used only inside procedures.

If you are embedding SQL statements in an application program, refer to the ALLBASE/SQL application programming guide for the language you are using. Bulk data manipulation is not available for FORTRAN. COBOL and FORTRAN do not provide the full set of dynamic preprocessing statements.

Error Conditions in ALLBASE/SQL

When you issue an SQL statement, error messages are returned if the statement cannot be carried out as intended. In an interactive session with ISQL, the messages are displayed on your terminal. In application programs, you access the message buffer directly by using the SQLEXPLAIN statement. The effect of an error on your session depends on three factors:

- Severity of the error
- Atomicity level set within the transaction
- Constraint checking mode set within the transaction

Severity of Errors

In general, errors result in partially or completely undoing the effects of an SQL statement. If the error is very severe, the transaction is rolled back. When a transaction is rolled back, ALLBASE/SQL displays a message like the following along with other messages:

```
Your current transaction was rolled back by DBCore. (DBERR 14029)
```

If an error is less severe, the statement is undone, but the transaction is allowed to continue.

Atomicity of Error Checking

By default, error checking is done at the statement level. In other words, the entire statement either succeeds or fails. This means that for set operations, the statement succeeds for all members of the set or fails for all members of the set. For example, if there is an error on the fifteenth row of a twenty-row BULK INSERT statement, the entire statement has no effect, and no rows are inserted. Or if an UPDATE statement that affects twenty rows creates a uniqueness violation for one row, the statement will fail for all rows. This approach guarantees data integrity for the entire statement. Under special circumstances, you can choose a different atomicity level for error checking:

- Row level
- Beyond the statement level

Setting the Atomicity to the Row Level

Sometimes statement level atomicity has drawbacks which you can correct. For example, data manipulation statements involving large amounts of data require considerable overhead for logging when issued at statement level, and this can impair performance. For better performance, you can set atomicity to row level. With row level atomicity, if an error occurs on one row, earlier rows are not undone. For example, for an error on the fifteenth row of a twenty-row BULK INSERT, statement execution stops at the fifteenth row, but the first fourteen rows will be processed unless you use the ROLLBACK WORK statement. To use row level error checking, issue the following statement:

```
SET DML ATOMICITY AT ROW LEVEL
```

Only DML statements can be checked for errors at the row level of atomicity. Refer to the SET DML ATOMICITY statement in Chapter 12 , “SQL Statements S - Z,” for complete details.

Deferring Error Checking beyond the Statement Level

Sometimes statement level atomicity is too narrow for your needs. For operations involving more than one table, it may be useful to defer error checking until all tables are updated. For example, if you are loading two tables that have a referential relationship that is circular--that is, each table references a primary key element in the other table--then you must defer constraint error checking until both tables are loaded; otherwise any attempt to load a row would result in a constraint error. To defer referential constraint error checking beyond the statement level, issue the following statement:

```
SET REFERENTIAL CONSTRAINTS DEFERRED
```

After the loading of both tables is complete, issue the following statement:

```
SET REFERENTIAL CONSTRAINTS IMMEDIATE
```

This turns on constraint error checking and reports any constraint errors that now exist between the two tables. Only integrity constraint error checking can be deferred beyond the statement level. For complete details, refer to the SET CONSTRAINTS statement Chapter 12 , “SQL Statements S - Z.”

Additional Information about Errors

Refer to the “Introduction” to the *ALLBASE/SQL Message Manual* for a general description of error handling. For the coding of error handling routines in application programs, refer to the chapter “Using Data Integrity Features” in the *ALLBASE/SQL Advanced Application Programming Guide* and the “Runtime Status Checking and the SQLCA” chapter in the application programming guide for the language of your choice. For error handling in procedures, refer to Chapter 4 , “Constraints, Procedures, and Rules.” For row level error checking, see the SET DML ATOMICITY statement, and for deferred constraint checking, see the SET CONSTRAINTS statement, both in Chapter 12 , “SQL Statements S - Z.”

Native Language Support

ALLBASE/SQL lets you manipulate databases in a wide variety of native languages in addition to the default language, known as NATIVE 3000. You can use either 8-bit or 16-bit character data, as appropriate for the language you select. In addition, you can always include ASCII data in any database, because ASCII is a subset of each supported character set. The collating sequence for sorting and comparisons is that of the native language selected.

You can use native language characters in a wide variety of places, including these:

- Character literals
- Values stored in host variables for CHAR or VARCHAR data (but not as variable names)
- ALLBASE/SQL object names

If your system has the proper message files installed, ALLBASE/SQL displays prompts, messages and banners in the language you select; and it displays system dates and time according to local customs. In addition, ISQL accepts responses to its prompts in the native language selected. However, regardless of the native language used, the syntax of ISQL and SQL statements--including punctuation--remains in ASCII. Note that MPE XL does not support either native language file names or DBEnvironment names.

In order to use a native language other than the default, you must follow the steps below:

1. Make sure your I/O devices support the character set you use.
2. Set the MPE job control word NLUSERANG to the number(*LangNum*) of the native language you use. Use the following MPE XL command:

   ```
   SETJCW NLUSERLANG = LangNum
   ```

   This language then becomes the **current language**. (If NLUSERLANG is not set, the current language is NATIVE-3000.)
3. Use the LANG = *LanguageName* option of the START DBE NEW statement to specify the language of a DBEnvironment when you create it. Run the MPE XL utility program NLUTIL.PUB.SYS to determine which native languages are supported on your system Here is a list of supported languages, preceded by the *LangNum* for each:

0	NATIVE-3000	7	FRENCH	13	SWEDISH	71	HEBREW
1	AMERICAN	8	GERMAN	14	ICELANDIC	81	TURKISH
2	C-FRENCH	9	ITALIAN	41	KATAKANA	201	CHINESE-S
3	DANISH	10	NORWEGIAN	51	ARABIC	211	CHINESE-T
4	DUTCH	11	PORTUGEUSE	52	ARABICW	221	JAPANESE
5	ENGLISH	12	SPANISH	61	GREEK	231	KOREAN
6	FINNISH						

Resetting the LANG variable while you are connected to a DBEnvironment has no effect on the current DBE session.

2 Using ALLBASE/SQL

This chapter shows how to use SQL statements for the following basic tasks:

- Creating DBEnvironments
- Starting and Terminating a DBE Session
- Creating Physical Storage
- Defining How Data is Stored and Retrieved
- Understanding Data Access Paths
- Controlling Database Access
- Manipulating Data
- Managing Transactions
- Auditing DBEnvironments (including setting up partitions)
- Using Wrapper DBEnvironments
- Using SQLAudit
- Application Programming
- Using Multiple Connections and Transactions with Timeouts
- Administering a Database
- Understanding the System Catalog

The next chapters contain more detailed information about the following topics:

- SQL Queries
- Constraints, Procedures and Rules
- Concurrency Control

The examples in this chapter are not intended to show all the functionality of the statements. For detailed information on ALLBASE/SQL statements, refer to the chapters “SQL Statements” in this manual. For information about database administration, refer to the *ALLBASE/SQL Database Administration Guide.*

Creating DBEnvironments

Before you can create a database, you must first configure a DBEnvironment. You use the START DBE NEW statement, optionally specifying startup parameters to override those assigned by default. You can use parameters to specify the following information:

- Multiuser or single-user mode
- Single, dual, or audit logging
- Number of page and log buffers
- Maximum number of partitions and concurrent transactions
- Number of runtime control blocks
- Timeout parameters
- DBEFile0 characteristics
- DBELog1 and DBELog2 characteristics

The DBEnvironment name, SomeDBE for example, is specified within single quotation marks in the START DBE NEW statement:

```
START DBE 'SomeDBE' MULTI NEW
```

This statement configures a DBEnvironment named SomeDBE in your group and account. This DBEnvironment contains the following files:

- A DBECon file named SomeDBE
- A DBEFile named DBEFile0, which is associated with a DBEFileSet named SYSTEM
- DBEFile0, containing a system catalog
- A single log file named DBELog1

The startup parameter *MULTI* makes this DBEnvironment accessible in multiuser mode by default.

The DBECon file stores the startup parameters defined by the START DBE NEW statement. For more information on startup parameters, refer to START DBE NEW in Chapter 12 , “SQL Statements S - Z.”

Once a DBEnvironment exists, one or more databases can be created in it. Because databases are collections of tables and views, databases are created by defining tables and views. The definition of tables and views is discussed later in this chapter in “Defining How Data is Stored and Retrieved.”

Specifying a Native Language Parameter

You can specify a native language parameter in creating a DBEnvironment. Use the LANG = *LanguageName* option in the START DBE NEW statement to specify a native language other than NATIVE 3000, as in the following example:

```
START DBE 'SomeDBE' NEW LANG = JAPANESE;
```

If you want to specify the name of the DBEnvironment in a native language, then the native language you specify in the LANG clause must be covered by the same character set as the language designated as the current language at the operating system level. The current language can be different from that of the DBEnvironment. In that case, all processing--including comparisons and sorting--will take place in accordance with the language of the DBEnvironment, but messages will appear in the operating-system-designated language if the appropriate message catalog is available. Also, scanning of user input will be in the current language. See "Native Language Support" in Chapter 1 , "Introduction," for information about specifying a native language as the current language.

Initial Privileges

When a DBEnvironment is configured, ALLBASE/SQL grants the following initial privileges:

- **DBECreator status**. The logon name that issues the START DBE NEW statement is the DBECreator. Users with this status can use all the SQLUtil statements to maintain the DBEnvironment.
- **DBA authority**. The DBECreator is given DBA authority. When you have DBA authority, you are authorized to use all the SQL statements in a DBEnvironment.

Nobody other than the DBECreator can connect to or issue SQL statements in the DBEnvironment until the DBECreator grants the appropriate authorities.

DBA authority cannot be revoked from the DBECreator.

Starting and Terminating a DBE Session

A DBE session is the period between establishing and terminating a connection to a DBEnvironment by a user or a program. You must be in a DBE session to execute any of the SQL statements except the START DBE or CONNECT statements.

You can establish either a single-user DBE session or a multiuser DBE session for a DBEnvironment. When you have a **single-user session**, no other users can connect to the DBEnvironment for the duration of that session. When you have a **multiuser session**, others can access the DBEnvironment at the same time.

How you establish a DBE session depends on whether the DBEnvironment is configured to operate in **autostart mode**. Autostart is ON by default, but the DBA can reset it by using SQLUtil. Refer to the "DBA Tasks and Tools" chapter in the *ALLBASE/SQL Database Administration Guide* for more information about using SQLUtil.

Sessions with Autostart

When the autostart flag for a DBEnvironment has the value of ON, users with **CONNECT authority** can start a DBE session by using the CONNECT statement:

```
CONNECT TO 'PartsDBEC.SomeGrp.SomeAcct'
```

Initiate a single-user session if the DBEnvironment is configured to operate in single-user mode. Initiate a multiuser session if the DBEnvironment is configured for multiuser mode.

You can have up to 32 simultaneous DBEnvironment connections.

Sessions without Autostart

When the autostart flag has the value of OFF, a DBA must issue the START DBE statement to make a DBEnvironment accessible. For example:

```
START DBE 'PartsDBE.SomeGrp.SomeAcct'
```

The START DBE statement illustrated above initiates a single-user session for the DBEnvironment. To make multiuser access possible, the *MULTI* option is specified as follows:

```
START DBE 'PartsDBE.SomeGrp.SomeAcct
```

After a DBEnvironment has been started up with the *MULTI* option, users with CONNECT authority can initiate multiuser sessions as in the following example:

```
CONNECT TO 'PartsDBE.SomeGrp.SomeAcct'
```

The START DBE statement also lets the DBA temporarily override several of the DBECon file startup parameters.

Terminating DBE Sessions

To terminate a DBE session, you simply specify the RELEASE statement as shown below:

```
RELEASE
```

Creating Physical Storage

To create physical storage, you use data definition statements to create the following storage areas:

- DBEFileSets
- DBEFiles
- TempSpace

File space for tables and indexes is managed by adding and dropping DBEFiles from DBEFileSets. DBEFiles are units of physical storage and DBEFileSets are logical collections of DBEFiles. You use the CREATE DBEFILESET statement to define a DBEFileSet, and the CREATE DBEFILE statement to define DBEFiles. You associate physical storage with the DBEFileSet by associating DBEFiles with it, using the ADD DBEFILE statement.

```
CREATE DBEFILESET WarehFS
CREATE DBEFILE WarehD1 WITH PAGES = 50, NAME = 'WarehD1'
ADD DBEFILE WarehD1 TO DBEFILESET WarehFS
```

Once you have created DBEFileSets and added DBEFiles to them, you need to specify the name of a DBEFileSet in your table creation statements. This then defines, for that table, the physical files that will be used to store the data. For complete details about creating DBEFiles and DBEFileSets, refer to the *ALLBASE/SQL Database Administration Guide*.

TempSpace can be optionally defined and is a specific area of storage used by the system for performing sorts in the database. TempSpaces are created and dropped by using the CREATE TEMPSPACE and DROP TEMPSPACE statements. Temporary files are allocated under the available TempSpaces as they are needed for performing a sort, and deallocated once the sort is completed. TempSpace information is accessible through the system catalog view SYSTEM.TEMPSPACE. A TempSpace is referred to by a unique name. If a TempSpace is not defined, sorting is done in the current group.

Defining How Data is Stored and Retrieved

To create database objects, you use data definition statements to define the following:

- Tables
- Views
- Indexes
- Constraints
- Procedures
- Rules

Creating a Table

When you define a table, use the CREATE TABLE statement to accomplish the following tasks:

1. Establish an automatic locking mode and default access authorities.
2. Name the table.
3. Describe the columns.
4. Identify a DBEFileSet for storage of its rows.

The following example contains numbers that refer to the list of tasks shown above:

```
          1              ---2---
          |             |       |
CREATE PUBLIC TABLE PurchDB.Parts

     (PartNumber   CHAR(16)      NOT NULL, ---
      PartName     VARCHAR(30),               | --3
      SalesPrice   DECIMAL (10,2))          ---
     IN WarehFS
          |
          4
```

You can also specify native language characteristics and integrity constraints at both the table and the column level.

Choosing the Locking Mode and Default Access Authorities

ALLBASE/SQL uses one of four locking modes for controlling access to data in a table by different transactions. A transaction is one or more SQL statements that together perform a unit of work. The locking modes are as follows:

- PRIVATE mode allows only one transaction at a time to access a table for reading or updating. Locking is done at the table level. PRIVATE is the default mode.
- PUBLICREAD mode allows multiple transactions to read a table, but only one to update it. Locking is done at the table level.

- PUBLIC mode allows multiple transactions to concurrently read and update a table. Locking is done at the page level.
- PUBLICROW mode allows multiple transactions to concurrently read and update a table. Locking is done at the row level, which permits greater concurrency than PUBLIC mode.

ALLBASE/SQL automatically uses the locking mode in the table definition whenever you access a table. You can use the LOCK TABLE statement to override automatic locking. You can use the ALTER TABLE statement to permanently change the implicit locking mode.

Tables created with PUBLICREAD, PUBLIC, and PUBLICROW options also have the following initial authorities associated with them:

- A PUBLICREAD table can be read by anyone who can start a DBE session.
- A PUBLICROW or PUBLIC table can be read and updated by anyone who can start a DBE session.

A DBA or the table's owner can use the GRANT and REVOKE statements to change these authorities.

The choice of PUBLICROW rather than PUBLIC mode may result in a transaction's obtaining more locks, since each row must be locked individually. For more information about the quantity of locking in PUBLIC and PUBLICROW tables, refer to the section “Effects of Page and Row Level Locking” in the “Physical Design” chapter of the *ALLBASE/SQL Database Administration Guide*.

Naming the Table and Columns

The name you assign to a table or column can be up to 20 bytes long and is governed by the rules in Chapter 6 , “Names.”

Defining the Columns

You enclose the column definitions in parentheses, separating multiple column definitions with a comma. At least one column must be defined. Each column is defined by a name and a data type.

Specifying Data Types

Data types describe the kind of data that can be stored in a column. ALLBASE/SQL has five numeric data types, two string data types, four date/time data types, and four binary data types as follows:

- Numeric data types:

  DECIMAL

  FLOAT

  REAL

  INTEGER

  SMALLINT

- Character string data types:

  CHAR(*n*)

  VARCHAR(*n*)

- Date/time data types:

  DATE

  TIME

  DATETIME

  INTERVAL

- Binary string data types:

  BINARY(*n*)

  VARBINARY(*n*)

  LONG BINARY(*n*)

  LONG VARBINARY(*n*)

When you define a column to be of a certain data type, ALLBASE/SQL ensures that each value stored in the column is in the range for the data type. Some data types (CHAR(*n*), VARCHAR(*n*), BINARY(*n*), VARBINARY(*n*), LONG BINARY(*n*) and LONG VARBINARY(*n*)) require a column length. CHAR(*n*) has a default length of 1; VARCHAR(*n*) does not. Other data types allow the specification of a precision (DECIMAL, FLOAT) and a scale (DECIMAL). Data types also affect the operations you can perform on data. Chapter 7 , "Data Types," defines the attributes of each data type as well as how the type affects various operations.

Specifying Column Options

You can also specify a NOT NULL, DEFAULT, native language, or constraints option for each column. The native language and constraint options are discussed in separate sections below.

When you define a column as NOT NULL, ALLBASE/SQL ensures that it contains no null values. NULL is a special data type that indicates the absence of a value.

The DEFAULT option allows you to specify a default value for a column. If the DEFAULT option is defined for a column and a value is not specified when an INSERT statement is executed, ALLBASE/SQL inserts the default value. Default values are of the following types:

- Constant
- NULL
- Current date and/or time

The following example specifies column options:

```
CREATE TABLE PurchDB.Parts
            (Column 1 char(20),
            Column 2 DEFAULT NULL)
```

You cannot use the DEFAULT option for a LONG data type column.

Specifying a DBEFileSet

The table rows are stored in the DBEFiles previously associated with the DBEFileSet named in the IN clause of the CREATE TABLE statement. If you do not specify a DBEFileSet, rows for the table are stored in the SYSTEM DBEFileSet. For best performance, explicitly specify a DBEFileSet other than the SYSTEM DBEFileSet.

Specifying Native Language Tables and Columns

Use the LANG = *TableLanguageName* option in the CREATE TABLE statement to specify a language other than the DBEnvironment's language. You can only specify NATIVE 3000 or the current native language of the DBEnvironment.

```
CREATE TABLE NewTable
             LANG = "NATIVE 3000"
             (Column1 char(20),
             Column2 char(10))
```

You must use double quotes around the name “NATIVE 3000” because it contains a hyphen. Normally, native language names do not require quotes. For more information on naming rules, refer to the “Names” chapter.

Use the LANG = *ColumnLanguageName* option in the column definitions of the CREATE TABLE statement to specify a column with a language different from that of the table as a whole. For example:

```
CREATE TABLE NewTable
             (Column1 char(20) LANG = "NATIVE 3000",
             Column2 char(20))
```

Sorting and pattern matching follow the rules of the column language. In order to maintain ASCII performance as much as possible, NATIVE 3000 column sorting and matching are done in ASCII.

By default, the language of a new table is the language of the DBEnvironment, and the language of a new column is the language of the table it belongs to.

Creating a View

A view is a table derived by placing a “window” over one or more tables to let users or programs see only certain data. Views are useful for limiting data visibility; they are also useful for pulling together data from various tables for easier use. The tables from which data for the view is derived are called **base tables**.

You define a view with the CREATE VIEW statement. The following are components of a view definition:

1. Name of the view
2. Name of its columns
3. Definition of how to derive data for the view

4. Specification of `WITH CHECK OPTION`, if desired

The following example contains numbers that refer to the view components listed above:

```
                1
                |
CREATE VIEW HiPrice
            (PartNum, Price)             --2
  AS SELECT PartNumber, SalesPrice   ---
       FROM PurchDB.Parts               |--3
      WHERE SalesPrice > 1000        ---
```

View names are governed by the same rules as table names.

The columns in a view can have the same names as the columns in the table(s) they are based on, or they can have different names. You only need to include column names in a view definition if you are using multiple base tables which have duplicate column names or if you want to rename the columns. You enclose the names in parentheses, but omit data types, which depend on the types of the columns in the base tables.

The derivation of the view is a `SELECT` statement. In the previous example, the view is derived from the PurchDB.Parts table. Each row in the view contains a part number and a price; only rows for parts costing more than $1000 can be accessed through this view.

Unlike a table definition, a view definition does not require that you specify where to store rows. A view is a `SELECT` statement stored in the system catalog, not a physical copy of the data; ALLBASE/SQL extracts data from physical tables at the time you use the view. Views can be used for both retrieving and modifying data. Refer to “Updatability of Queries” in Chapter 3 , “SQL Queries,” for restrictions governing the use of a view to change data in a base table.

The `WITH CHECK OPTION` for views is described in Chapter 4 , “Constraints, Procedures, and Rules.”

Creating Indexes

You can create an index on one or more columns in a query. An index can provide quick access to the data in your tables. For information on indexes, refer to section “Understanding Data Access Paths” later in this section.

Specifying Integrity Constraints

Using integrity constraints helps to ensure that a database contains only valid data. Integrity constraints provide a way to check data within the database system rather than by coding elaborate validation checks within application programs. An integrity constraint is either a unique constraint, a referential constraint, or a check constraint. These constraints are described in Chapter 4 , “Constraints, Procedures, and Rules.”

Creating Procedures

You can define procedures to enforce relationships among database tables or to automate nearly any operation in the DBEnvironment. The following example shows creating a procedure to perform deletions from the SupplyPrice table in the sample DBEnvironment

PartsDBE:

```
CREATE PROCEDURE PurchDB.DelSupply(Part CHAR(16) NOT NULL) AS
BEGIN
   DELETE FROM PurchDB.SupplyPrice
   WHERE PartNumber = :Part;
END
```

The procedure definition includes a parameter declaration. The parameter Part accepts a value into the procedure at run time. You execute the procedure with a statement like the following example:

```
EXECUTE PROCEDURE PurchDB.DelSupply ('1123-P-01')
```

The effect of the procedure is to delete all rows in the SupplyPrice table whose part number is 1123-P-01. For detailed information about creating and using procedures, refer to Chapter 4 , “Constraints, Procedures, and Rules.”

Creating Rules

Once a table is defined, you can create a rule that will execute a procedure whenever a specific firing condition is met. For example, you can define a rule that will execute a procedure to delete rows from the SupplyPrice table whenever a specific part is dropped from the Parts table in the sample DBEnvironment PartsDBE:

```
CREATE RULE PurchDB.RemovePart AFTER DELETE FROM PurchDB.Parts
EXECUTE PROCEDURE PurchDB.DelSupply (PartNumber)
```

Once the rule exists, you activate it by performing a DELETE:

```
DELETE FROM PurchDB.Parts
WHERE PartNumber = '1123-P-01'
```

For detailed information about creating and using rules, refer to the “Constraints, Procedures, and Rules” chapter.

Understanding Data Access Paths

In creating a database, you must consider not only the arrangement of data, but also the ways in which the data will be accessed during data manipulation operations. The four following access methods are supported directly by ALLBASE/SQL:

- Serial access
- Indexed access
- Hashed access
- TID access

For indexed access, you must create a named index, or unique or referential constraint on a table. Unique and referential constraints are supported by constraint indexes, which are similar to B-tree indexes. For information on B-trees, refer to the section “Designing Indexes” in the chapter “Logical Design” of the *ALLBASE/SQL Database Administration Guide*

For hashed access, you must define a hash structure as you create the table.

By default, you do not explicitly choose an access method when you issue a query; ALLBASE/SQL does this for you in a process known as **optimization**. Optimization determines the best access path to the data for the query you have submitted. If a choice is available among the different access methods--for example, if serial, indexed, and hashed access are all possible for the same query--then the optimizer picks the best path. If no other choice is available, the optimizer chooses serial access, also known as a sequential or table scan. Serial access is always possible.

To override the access method chosen by the optimizer, use the SETOPT statement.

Serial Access

Serial access does not require the existence of any special object in addition to the table itself. If ALLBASE/SQL chooses serial access when you issue a query, it starts reading data from the first page in the table and continues to the end. Serial access is probably the best access method when you intend to read all the data in the table. For example, an application that updates every row in a table in exactly the same way would perform best using a serial scan.

Indexed Access

Indexed access requires the use of a named index defined on specific columns in the table to be accessed. Indexes can be plain, or they can be unique and/or clustering. Tables having a **unique index** cannot have duplicate data values in the key column(s). A **clustering index** causes rows with similar key values to be stored near to each other on disk when this is possible. A table that is to use a clustering index should be loaded in the key order specified by the clustering index. A clustering index can be defined on a unique or referential constraint.

Whenever you issue a query, the query processor checks to see if an index exists for one or

more of the columns in the query. If an index is available and if the optimizer decides that using the index is the fastest way to access the data, ALLBASE/SQL looks up the key values in the index first, then goes directly to the pages containing table data.

For example, in the following query, assume that PurchDB.Parts contains a large number of rows and that a unique index exists on the PartNumber column:

```
isql=> SELECT PartName, SalesPrice FROM PurchDB.Parts
> WHERE PartNumber = '1323-D-01';
```

The optimizer would probably choose this unique index for access to the single row because the alternative choice--a serial scan--would require reading each page in the table until the qualifying row is reached.

You define an index with the CREATE INDEX statement. The components of an index definition are as follows:

1. Type of the index (optional)
2. Name of the index
3. Table on which the index operates
4. Key column(s)

The following example contains numbers that refer to the index components listed above:

```
           1
           |
CREATE UNIQUE INDEX
       PartIndex       --2
    ON PurchDB.Parts   --3
       (Partno)        --4
```

ALLBASE/SQL can choose to use an index when processing the SELECT, UPDATE, or DELETE statements if the following criteria are satisfied:

- The statement contains a WHERE clause, which consists of one or more **predicates**. A predicate is a comparison of expressions that evaluates to a value of True or False. Refer to the “Search Conditions” chapter for more information on predicates.
- The statement contains explicit join syntax.
- Predicates are **optimizable**, which means that the use of an index is considered in choosing an access path for the data. The following predicates are optimizable when all the data types within them are the same; in the case of DECIMAL data, the precisions and scales of the values must be the same:

  — WHERE *Column1 ComparisonOperator Column2*, in which *ComparisonOperator* is one of the following: =, >, >=, <, or <=. An index may be used if *Column1* and *Column2* are in different tables and an index exists on either column. For example:

  ```
  WHERE PurchDB.Parts.PartNumber = PurchDB.SupplyPrice.PartNumber
  ```

  — WHERE *Column1 ComparisonOperator (Constant or HostVariable)*, in which *ComparisonOperator* is as defined above. An index may be used if one exists on *Column1*; however, an index may be used if a host variable appears in the

predicate *only* if the comparison operator is =, >, >=, <, or <= . For example:

```
WHERE SupplyPrice = :SupplyPrice
```

— WHERE *Column1* BETWEEN (*Column2* or *Constant* or *HostVariable*) AND (*Column2* or *Constant* or *HostVariable*). For example:

```
WHERE OrderNumber BETWEEN '1123-P-01' AND '1243-MU-01'
```

- Some queries which use the MIN or MAX aggregate function on an indexed column as follows are optimizable:
  - — MIN/MAX column is the first column of a nonhashed index.
  - — MIN/MAX indexed column on a single table with or without predicates.
  - — MIN/MAX indexed column on the outermost table of a nested loop join query.
  - — Single MIN/MAX within one query.
- ALLBASE/SQL does not use an index in the following types of queries:
  - — The query contains a WHERE clause using a not-equal (<>) arithmetic operator, such as, WHERE *Column1* <> (*Column2* or *Constant* or *Host Variable*). For example:

    ```
    WHERE VendorState <> :VendorState
    ```

  - — The query contains a predicate using an arithmetic expression. For example:

    ```
    WHERE Column1 > Column2*:HostVariable
    ```

  - — MIN or MAX is used with the GROUP BY, ORDER BY, or HAVING clause.
  - — A MIN or MAX indexed column exists in the inner table of a nested-loop, join query.
  - — A MIN or MAX indexed column exists on all tables of a sort-merge, join query.
  - — MIN or MAX is used with an expression.
  - — One query contains multiple MINs or MAXs.
  - — A LIKE predicate contains a host variable.

  If other predicates are used, then an index is considered in choosing an access path.

For more information about indexes, refer to the “Designing Indexes” section in the “Logical Design” chapter of the *ALLBASE/SQL Database Administration Guide*.

Hashed Access

Hashed access requires you to specify hashing when you create the table, before loading data. Because a hash structure is specified as part of the table definition, you do not assign a name to it, as you do with an index. However, you must identify specific key columns and a number of primary pages for data storage. ALLBASE/SQL determines the placement of rows based on specific unique key values. You can define one hash structure per table at table creation time; and if a hash is defined, you cannot define a clustering index on the table. You can define a multiple-column key for a hash structure; up to 16 columns are permitted in the key.

A **hash structure** is a group of designated pages in a DBEFile that are set aside for the storage of tuples according to the values in a unique **hash key**. The key enforces

uniqueness; duplicate values cannot exist in the hash key column(s). A well-chosen hash key, like a good index key, provides the optimizer with the choice of a potentially faster data access method than a serial scan.

Create a hash structure at the time you create a table. In addition to the components of a table definition, a hash structure definition includes:

1. Columns that define the hash key
2. Number of primary pages

The reference numbers in the following example refer to the table definition components listed above:

```
CREATE PUBLIC TABLE PurchDB.Vendors
                    (VendorNumber  INTEGER  NOT NULL,
                    VendorName  CHAR(30)  NOT NULL,
                    ContactName  CHAR(30),
                    PhoneNumber  CHAR(15),
                    VendorStreet  CHAR(30)  NOT NULL,
                    VendorCity  CHAR(20)  NOT NULL,
                    VendorState  CHAR(2)   NOT NULL,
                    VendorZipCode  CHAR(10)  NOT NULL,
                    VendorRemarks  VARCHAR(60) )

      UNIQUE HASH ON (VendorNumber)        -- 1
                    PAGES = 101            -- 2

                  IN PurchFS
```

Use the UNIQUE HASH clause or the HASH ON CONSTRAINT clause to specify one or more columns for a hash key. Use the PAGES= clause to define a number of **primary pages** in which to store the data in the table. This is different from ordinary data storage, which does not require a number of primary pages.

Based on the key and the number of primary pages you specify, ALLBASE/SQL calculates a page number for each row before insertion into the table. The page number depends directly on the data in the key. Because a specific number of primary pages is specified, you must create the hash structure as you create the table; you cannot modify a table from normal to hash storage at a later time.

The optimizer *can* decide to use hashed access provided the statement contains a WHERE clause with an EQUAL factor for each column in the hash key. This makes hashing especially useful for tables on which you need quick random access to a specific row.

For example, assuming you have defined a hash key on VendorNumber, the optimizer might choose hashed access for the following:

```
isql=> SELECT * FROM PurchDB.Vendors
> WHERE VendorNumber = 9002;
```

However, it would *not* consider hash access for the following:

```
isql=> SELECT * FROM PurchDB.Vendors
> WHERE VendorNumber > 9002
> ORDER BY VendorName;
```

Hash structures operate like unique indexes; that is, they enforce the uniqueness of each key in the table. If you attempt to insert a duplicate key, ALLBASE/SQL will return an error message.

Differences between Hashed and Indexed Access

Hashing may provide faster access than B-tree lookups for many types of common queries, and it does not require the overhead of additional file space required by B-tree indexes. In addition, hashing is not subject to the overhead of updating index pages when you insert or modify rows. However, updating key values in a hash table requires you to delete the row containing the key value and then insert a row containing the new value. This means that you should choose a non-volatile key for hashing whenever possible.

When to Use a Hash Structure

Hashing offers high performance when you need essentially random access to individual tuples. It is not appropriate for applications that require sorting of the query result. In cases where both random access and sorting are required at different times, you can define a B-tree index as well as a hashing structure. This allows the optimizer the choice of the most efficient method for the specific query.

The best candidates for the use of hash structures are applications in which the following occur:

- Keys are not frequently updated. Remember that you cannot use the UPDATE statement on hash key columns. This means that you must delete and then insert rows that contain changes to key values.
- Most queries contain EQUAL factors on hash key columns.
- Tuples are of fixed size, with a minimum of VARCHARS and NULL values.

You should *not* use a hash structure if your queries need to scan large areas, for instance, with BETWEEN clauses or with predicates containing <> factors.

TID Access

Each row of a table has a unique address called the **tuple identifier**, or **TID**. TID functionality provides the fastest possible data access to a single row. You can obtain the TID of any row with the SELECT statement. For more information on TID access refer to the ALLBASE/SQL application programming manual for the language you are using.

Controlling Database Access

ALLBASE/SQL uses authorities to determine who can issue which SQL statements and who can execute programs that access databases in a DBEnvironment. For complete details about security schemes refer to the *ALLBASE/SQL Database Administration Guide*.

Authorities

ALLBASE/SQL has the following several kinds of authorities:

- **Table and view authorities** are the following privileges used to access data in a specific table or through a specific view and to add columns and indexes, and create foreign keys referencing a specific table:

`SELECT`	retrieve rows
`INSERT`	insert rows
`DELETE`	delete rows
`UPDATE`	change one or more columns in a row
`ALTER`	add new columns to a table
`INDEX`	create an index for the table
`REFERENCES`	refer to one or more columns when defining a foreign key in a referencing table

- **RUN authority** is the privilege to execute a specific program module that accesses a DBEnvironment.

`EXECUTE`	execute a procedure

- **Special authorities** are the following privileges:

`CONNECT`	connect to a DBEnvironment
`RESOURCE`	create tables and authorization groups
`DBA`	issue *`all`* SQL statements and to execute any program that accesses an ALLBASE/SQL DBEnvironment

- **OWNER authority** controls specific programs, tables, views, or authorization groups.

Obtaining Authorization

You obtain authority by the following methods:

- Configuring a DBEnvironment and automatically becoming a DBA.
- Being granted one or more specific authorities.
- Owning a table, view, module, or group.

DBA Authority

When a DBEnvironment is configured, DBA authority is automatically given to the login name of the DBECreator.

A user with DBA authority (also referred to as the DBA) has extensive control over data in a DBEnvironment. The DBA can issue almost all the SQL statements and execute all the programs that access the DBEnvironment. The two SQL statements that *only* a DBECreator can issue are, START DBE NEWLOG and START DBE RECOVER. Some SQL statements *only a DBA* can issue. Most of these statements are DBEnvironment-wide in scope. For example, only DBAs can grant the special authorities (CONNECT, RESOURCE, and DBA) and define DBEFiles and DBEFileSets. In addition, only a DBA can issue statements that control objects owned by a class name; for example, only DBAs can drop or issue grants for a table owned by a class name.

Grants

All authorities except OWNER authority can be *granted* by using the GRANT statement. The GRANT statement gives authorities to individual users, to authorization groups, or to all users.

The following grants authorize a user with a logOn name of WOLFGANG@DBMS to start a DBE session and to retrieve rows from the table named Quotas. Wolfgang can also create his own database because he is also granted RESOURCE authority.

```
GRANT CONNECT TO WOLFGANG@DBMS
GRANT SELECT ON Marketing.Quotas TO WOLFGANG@DBMS
GRANT RESOURCE TO WOLFGANGg@DBMS
```

The following grants authorize the group named Managers to start a DBE session and all users to retrieve rows from the table Forecast:

```
GRANT CONNECT TO Managers
GRANT SELECT ON Marketing.Forecast TO PUBLIC
```

The REVOKE statement is used to eliminate authorities:

```
REVOKE RESOURCE FROM WOLFGANG@DBMS
```

DBAs can grant or revoke authorities. The only individuals entitled to grant and revoke authorities are users or members of groups that *own* tables, views, or modules, or those who have received grantable privileges, as described below. Individuals or members of groups that own tables, views, or modules can issue grants for objects they own.

Grantable Privileges

If a grantor specifies the WITH GRANT OPTION clause when issuing the GRANT statement on table and view authorities, the grantee receives not only the privilege, but the authority to grant that same privilege, with or without the WITH GRANT OPTION, to another user. The grantee is also entitled to revoke authorities he or she granted. This kind of privilege is called a grantable privilege. The use of grantable privileges can result in chains of grants.

A **cycle** in a chain of grants is not allowed; that is, a user cannot be granted the same authority more than once on an object. If a grant of authority causes a cycle, you will receive an error message. The WITH GRANT OPTION clause cannot be specified when the

grantee is a group. The following statement grants UPDATE authority to Amanda, who can then grant that authority to individual users or a class:

```
GRANT UPDATE ON Marketing.Forecast TO AMANDA@DBMS WITH GRANT OPTION;
```

Users with a grantable privilege can only revoke privileges they have granted and chains they have caused. To revoke the privilege given to the grantee and any subsequent grantees in a chain, the grantor must use the CASCADE option of the REVOKE statement.

Owners can revoke any privilege on their object, but to revoke a privilege that has been given to subsequent grantees, the CASCADE option must be used. The DBA does not have to use the CASCADE option to revoke a grantable privilege from a user. However, if CASCADE is not used, that privilege is removed from the specified grantee only, not from the subsequent chain of grants. Then, an orphaned privilege is created. An orphaned privilege can be given a parent by the DBA with the BY clause of the GRANT statement. For more information on orphaned privileges, refer to "Using the WITH GRANT OPTION Clause" in the chapter "Database Creation and Security" in the *ALLBASE/SQL Database Administration Guide*.

Ownership

The following six objects have owners associated with them:

- Tables
- Views
- Authorization groups
- Modules
- Procedures
- Rules

These objects can be owned by an individual, an authorization group, or a class; but an object can have only one owner at a time.

An owner becomes associated with an object in one of several ways:

- When an individual creates one of the five objects, that individual becomes its owner. The owner name is derived from the individual's login name. To create a table or group, you need DBA or RESOURCE authority. To create a module, you need DBA or CONNECT authority. To create a view, you need DBA, SELECT, or OWNER authority for the tables and views it is based on.
- A DBA or the owner of an object can *transfer ownership* of the object to another individual, a group, or a class by using the TRANSFER OWNERSHIP statement. The ownership of modules cannot be transferred. WOLFGANG@DBMS can transfer ownership of his Composers table to Wendy as follows:

  ```
  TRANSFER OWNERSHIP OF TABLE Composers TO WENDY@ROBERTS
  ```

- A DBA can create any of these objects and *name the owner* in the statement that creates the object. Other users can name any group as owner when creating an object if they are a member of that group. With the following statements, a DBA creates a group called Managers; a DBA or a member of Managers can assign ownership of the table

named Salary to that group when creating the table:

```
CREATE GROUP Managers
CREATE TABLE Managers.Salary...
```

When you refer in an SQL statement to a table, a view, a module, or an authorization group, you specify both the owner's name and the name of the object. If you own the object, however, you can omit the owner's name. When WOLFGANG@DBMS retrieves information from the Parts table, for example, he must specify the owner name. For example:

```
SELECT PartNumber FROM PurchDB.Parts
```

The system views belong to special owners named SYSTEM and CATALOG. Therefore when you refer to one of the system views, you must specify that name:

```
SELECT * FROM System.Table
or
SELECT * FROM Catalog.Table
```

Default Owner Rules

In several statements, when a name is specified, such as table name, rule name, group name, or index name, specification of the owner name is optional. The method of determining the default owner when no owner is specified is as follows:

- If the name is within a `CREATE PROCEDURE` statement (except for the procedure name itself), and it is not within a `CREATE SCHEMA` statement in that procedure, then the default owner is the procedure's owner.
- If the name is within a `CREATE SCHEMA` statement and it is not within a `CREATE PROCEDURE` statement in that schema, then the default owner name is the authorization name of that schema.
- If you have specified an owner using the `ISQL SET OWNER` command, everything you create will be owned by the owner specified in that command.
- If you use the -o option to specify an alternate DBEUserID prior to preprocessing an application containing embedded SQL statements, then the owner specified is the default owner of the module.
- If none of the above apply, then the default owner name is the current DBEUserID. The DBEUserID is the logon name concatenated with '@' and concatenated with the group name.

In `CREATE INDEX, CREATE RULE, DROP INDEX, DROP RULE`, the default owner for the index or rule name, respectively, has additional possible values which are described with those statements.

Ownership Privileges

The following summarizes the privileges that extend to users or members of groups that own objects:

- **Group owners** can add members to and remove them from their group as well as drop the group.

- **Group members** have ownership privileges over all objects owned by their group.
- **Group members** have all privileges granted to the group.
- **Table owners** can add columns to the table or drop the table.

  They can add and drop constraints.

  They can create and drop indexes for the table. They can grant and revoke authorities for the table, and transfer their ownership to another owner. They can retrieve data from the table, change the data, update statistics, lock the table, and create views on the table. Transferring ownership of a table transfers the ownership of indexes, constraints, and rules defined on the table. And grantor of privileges by owner also changes.
- **Index owners** can drop their indexes. The index owner must be the same as the owner of the table the index is defined upon. Index ownership is transferred along with the ownership of the table the index is defined upon.
- **View owners** can drop their view. They can grant and revoke authorities for the view and transfer their ownership to another owner. They can also access data through their views.
- **Module owners** can execute, validate, and drop their modules. They can grant and revoke RUN authority for their modules. Ownership of modules cannot be transferred.
- **Procedure owners** can drop their procedures. They can grant and revoke EXECUTE authority for their procedures, and they can transfer ownership to another owner.
- **Rule owners** can drop their rules. The rule owner must be the same as the owner of the table the rule is defined upon. Rule ownership is transferred along with the ownership of the table the rule is defined upon.

Authorization Groups

An authorization group is a named collection of users or other groups. The CREATE GROUP statement is used to define groups, and the ADD TO GROUP statement is used to associate individuals or other groups with the group. The GRANT statement assigns authorities to a group. All three statements are used in the following example:

```
CREATE GROUP PurchManagers
ADD MARGUERITE@RYAN, RON@HART, SHARON@MULDOON TO GROUP PurchManagers
GRANT SELECT on PurchDB.Parts TO PurchManagers
```

Any member of the group PurchManagers can select data from table PurchDB.Parts. Authorization groups have several advantages as described here:

- Groups simplify authorization. They make it possible to grant authorities to multiple users or groups with one GRANT statement. In addition, as new users need authorities, the DBA can simply add them to a group already possessing the appropriate authorization.
- Groups make control over the type of data access independent of control over who can access data. For example, the owner of a table can grant different types of access (SELECT, UPDATE, etc.) to a group; but who belongs to the group is controlled by the DBA or the group's owner, not by the table's owner.

Classes

A class is a special category of owner that is neither a conventional DBEUserID nor a group. You may wish to assign ownership of objects to a class when you do not want any individual or group to have automatic access to them. With class ownership, the DBA controls all authorities, because objects that belong to a class can be created and maintained only by the DBA. For a class to be useful, its class name must be different from the name of any existing DBEUserID or group name.

A DBA can create a class by doing one of the following:

- Creating a table or view with the class name as owner name.
- Preprocessing an application with the class name as owner name.
- Transferring ownership of an object to a class name.

For example, the sample DBEnvironment contains several tables owned by the class PurchDB. The table PurchDB.Parts was created with the following statement:

```
CREATE TABLE PurchDB.Parts
            (PartNumber  CHAR(16)  NOT NULL,
            PartName  CHAR(30),
            SalesPrice  DECIMAL(10,2))
          IN WarehFS;
```

After creating objects owned by the class, you must grant the specific authorities you wish users or groups to have. Suppose you have a group PurStaff consisting of DBEUserIDs for members of the Purchasing department. You could grant authorities to the group as follows:

```
GRANT SELECT, UPDATE ON PurchDB.Parts to PurStaff;
```

Differences between Groups and Classes

You create a group explicitly by using the CREATE GROUP statement. You create a class implicitly by creating objects that use the class name as the owner name.

A group has members, all of which have the privileges the group has. For example, if a user is a member of the group Sales, then that user can drop or alter objects owned by Sales.

A class does not have members, nor can it use any authorities, although you can grant them if you wish. This can be useful in a scenario in which you want to preassign ownership of objects to a DBEUserID which has no logon ID on your system.

Manipulating Data

Most users of ALLBASE/SQL are primarily interested in manipulating data in DBEnvironments. **Data manipulation** consists of following operations:

- Selecting data
- Inserting data into tables
- Updating rows in tables
- Deleting rows

In order to select data, you create queries, which are fully described in the next chapter. The other types of data manipulation are presented briefly in the next sections. For complete information, refer to the descriptions of the `SELECT, INSERT, UPDATE`, and `DELETE` statements in the "SQL Statements" chapter.

Inserting Data

You use the `INSERT` statement to add rows to a table, specifying the following information:

1. A table or view name
2. Column names
3. Column values

The following example contains numbers that refer to the items in the list above:

```
                          1
                          |
INSERT INTO PurchDB.Parts
            (PartNumber, PartName)              --2
     VALUES ('9999-AJ','Interface Engine')
            |                           |
            ----------------------------
                          |
                          3
```

Only a single table name or view name can be specified. Only certain views can be used to insert rows into a base table, as described under "Updatability of Queries" in Chapter 3 , "SQL Queries."

The column names can be omitted if you are going to put a value into every column in the row. Otherwise, you name the columns you want to assign values to, enclosing the column names in parentheses and separating multiple column names with commas. Columns not named are assigned their default values. If no default exists for a column, it is assigned the null value. If you define a column as `NOT NULL` when you create a table, then you *must* assign a non-null value or specify a default value to the column.

The column values are also enclosed in parentheses and separated by commas. Character data is delimited with single quotation marks. The value `NULL` can be entered into columns that permit null values.

You can copy rows from one or more tables or views into another table by using a form of the INSERT statement (often called a type 2 Insert) in which you specify the following items:

1. A table or view name
2. A SELECT statement

Note that the numbers in the next example refer to the items listed above:

```
                    1
                    |
INSERT INTO PurchDB.Drives
            SELECT * FROM PurchDB.Parts    -- 2
            WHERE PartName LIKE 'Drives%'
```

The rows in the query result produced by the SELECT statement are inserted into PurchDB.Drives. The SELECT statement cannot contain an ORDER BY clause and cannot name the target table in the FROM clause. The target table must exist prior to an INSERT operation.

Updating Data

You change data in one of more columns by using the UPDATE statement. These are the components of the UPDATE statement:

1. The name of a table or a view
2. A SET clause
3. A WHERE clause

The following example illustrates the UPDATE statement and its components; the reference numbers identify the components listed above.

```
UPDATE PurchDB.Parts          --1
  SET SalesPrice = 15.95      --2
WHERE PartNumber = '9999-AJ'  --3
```

Only a single table name or view name can be specified. Only certain views can be used to update, as described under “Updatability of Queries” in Chapter 3 , “SQL Queries.” For each column to be updated, you specify a column name and value in the SET clause. NULL is a valid value for columns that can contain null values. Unless you specify a WHERE clause, *all* rows of the named table or view are updated. A search condition in this clause describes which rows to update. The search condition in the previous example specifies that the row(s) to be updated must name PartNumber 9999-AJ.

Deleting Data

You use the DELETE statement to delete entire rows. This statement has two components as follows:

1. A table or view name
2. A WHERE clause

The following example illustrates the DELETE statement and its two components:

```
DELETE FROM PurchDB.Parts            --1
   WHERE PartNumber = '9999-AJ'      --2
```

Only a single table name or view name can be specified. Only certain views can be used to delete rows, as described under “Updatability of Queries” in Chapter 3 , “SQL Queries.”

The WHERE clause is optional. You omit it if you want to delete *all* the rows in a table or view. Otherwise, you use it to specify a search condition for which row(s) to delete.

Managing Transactions

A transaction is a logical unit of work that changes the database. All actions within this logical unit of work must succeed, or all of them must fail. When a transaction completes successfully, it is said to commit. Should a transaction fail, none of the changes it generates are recorded in the database, and the transaction aborts.

A transaction is bounded by the BEGIN WORK and COMMIT WORK statements. One or more SQL statements, and any number of programming language statements can be contained within a transaction. An example of a simple transaction is as follows:

```
BEGIN WORK

UPDATE PurchDB.Parts
   SET PartName = 'Defibrillator'
 WHERE PartNumber = '1152-DE-95683'

COMMIT WORK
```

The SQL statements used in transaction management are as follows:

BEGIN WORK	Starts the transaction.
COMMIT WORK	Terminates a successful transaction.
ROLLBACK WORK	Undoes any changes made by the current transaction.
SAVEPOINT	Permits partial rollback of a transaction.

Objectives of Transaction Management

The objectives of transaction management are related to one another. Data integrity is enforced by proper transaction management, but must be balanced by the need for high concurrency. The use of transactions facilitates the recovery of data after a crash, maintaining data integrity.

Ensuring Logical Data Integrity

The data in the database must be accurate and consistent. For example, adding a part to the warehouse inventory entails inserting a row into three tables: PurchDB.Parts, PurchDB.SupplyPrice, and PurchDB.Inventory. All three inserts must succeed, or else the

database is left in an inconsistent state. To enforce data integrity, the three inserts are contained in a single transaction. If any one insert fails, then the entire transaction fails and none of the other inserts takes effect. The following example shows how this transaction might be coded:

```
BEGIN WORK
INSERT INTO PurchDB.Parts ...
If the insert into PurchDB.Parts fails then
   ROLLBACK
else
   INSERT INTO PurchDB.SupplyPrice ...
   If the insert into PurchDB.SupplyPrice fails then
      ROLLBACK
   else
      INSERT INTO PurchDB.Inventory ...
      If the insert into PurchDB.Inventory fails then
         ROLLBACK
      else
      COMMIT WORK
      endif
   endif
endif
```

Maximizing Concurrency

Concurrency is the degree to which data can be accessed simultaneously by multiple users. For example, an application that allows one hundred users to access a table simultaneously has higher concurrency, and therefore better performance, than an application that allows only one user at a time to access the table. Locking regulates the simultaneous access of data. For example, if one user updates a row, the row is locked and other users cannot access the row until the first user is finished. Locking the row enforces data integrity, but reduces concurrency because other users are forced to wait. The isolation level specified in a `BEGIN WORK` statement affects the duration and types of locks held within a transaction. Isolation levels are fully discussed in Chapter 5 , “Concurrency Control through Locks and Isolation Levels.” Well-managed transactions balance the conflicting requirements of minimal lock contention and maximum concurrency.

Facilitating Recovery

When a soft crash occurs, incomplete transactions are automatically rolled back when the DBEnvironment is restarted. If archive logging is in effect when a hard crash occurs, committed transactions are applied to the database during rollforward recovery. In both cases, only those transactions that were uncommitted when the crash occurred need to be redone.

Starting Transactions

A transaction is initiated with either an implicit or explicit BEGIN WORK statement. An implicit BEGIN WORK statement is issued by ALLBASE/SQL when any SQL statement is executed, except for the following:

ASSIGN	BEGIN ARCHIVE	BEGIN DECLARE SECTION
BEGIN WORK	CHECKPOINT	COMMIT ARCHIVE
COMMIT WORK	CONNECT	DECLARE VARIABLE
DISABLE AUDIT LOGGING	ENABLE AUDIT LOGGING	END DECLARE SECTION
GOTO	IF	INCLUDE
PRINT	RAISE ERROR	RELEASE
RESET	RETURN	ROLLBACK TO SAVEPOINT
ROLLBACK WORK	SET SESSION	EX SET TIMEOUT
SET TRANSACTION	START DBE	STOP DBE
SQLEXPLAIN	TERMINATE USER	WHENEVER
WHILE		

Explicit BEGIN WORK statements are recommended, for the following reasons:

- Explicit BEGIN WORK statements make your code easier to read.
- You must use an explicit BEGIN WORK statement to specify a non-default isolation level or transaction priority.
- You might unintentionally lock out other users by the default isolation level of an implicit BEGIN WORK.

Since nested transactions are not allowed, an error is generated if a session with an active transaction issues a BEGIN WORK statement. The first transaction must end before another transaction can begin.

Ending Transactions

A transaction ends when either a COMMIT WORK or a ROLLBACK WORK statement is issued. All locks held by the session are released when the transaction ends, except those held by a kept cursor.

Using COMMIT WORK

Issue the COMMIT WORK statement when the transaction is successful and you want the changes made permanent. Unlike the BEGIN WORK and ROLLBACK WORK statements, the COMMIT WORK statement is never issued automatically by ALLBASE/SQL. You must issue the COMMIT WORK explicitly for each transaction. The COMMIT WORK statement causes the contents of the log buffer to be written to a log file. If rollforward recovery is needed at a later time, the transactions recorded in the log file are applied to the database.

Using ROLLBACK WORK

The ROLLBACK WORK statement ends the transaction and undoes all data modifications made since the BEGIN WORK statement, unless it references a savepoint. (See the discussion of savepoints in the following section.) The ROLLBACK WORK statement is issued automatically by ALLBASE/SQL under the following conditions:

- A non-archive log file becomes full.
- A RELEASE statement is issued before the end of the transaction.
- A system failure occurs. When the system is up again, and a START DBE statement is issued, incomplete transactions are rolled back.
- ALLBASE/SQL chooses the transaction as the victim when breaking a deadlock.
- The session is terminated by a TERMINATE USER command.

The ROLLBACK WORK statement should be issued explicitly to maintain data integrity. You may want to issue a ROLLBACK WORK in an application program when any of the following situations arise:

- The transaction contains more than one SQL statement and one of the statements generates an error. For example, if your transaction contains three INSERT statements, and the second INSERT fails, you should rollback the entire transaction.
- An INSERT, UPDATE, or DELETE statement that affects multiple rows generates an error after some of the rows have been modified. You should rollback the transaction if the partial changes will leave your database in an inconsistent state.
- The end user provides input indicating that he or she does not want to commit the transaction.

Using SAVEPOINT

The SAVEPOINT statement allows you to rollback part of a transaction. Multiple savepoints are permitted within a transaction anywhere between the BEGIN WORK and COMMIT WORK statements. Each SAVEPOINT statement places a unique marker, called a savepoint number, within the transaction. When a subsequent ROLLBACK references the savepoint number, only those database changes made after the savepoint are rolled back. Rolling back to a savepoint does not end the transaction, but it does release locks obtained after the savepoint was issued.

In the following ISQL example, the number identifying the savepoint marker is 6. The update performed after the SAVEPOINT statement is undone by the ROLLBACK statement, but any database changes made before savepoint 6 are unaffected.

```
isql=> SAVEPOINT;
```

Savepoint number is 6. Use this number to do ROLLBACK WORK to 6.

```
isql=> UPDATE PurchDBParts
> SET SalesPrice = 244.00
> WHERE PartNumber = '1243-MU-01';

isql=> ROLLBACK WORK to 6;
```

After a rollback to a savepoint has been executed, use the COMMIT WORK statement to make the changes that were not rolled back permanent. If you want to rollback the entire transaction, issue the ROLLBACK statement without a savepoint.

Savepoints are suitable for transactions that perform several operations, any of which may need to be rolled back. In the following example, a travel agency is booking tour reservations for 15 people. When the first attempt to make a hotel reservation fails, only that part of the transaction is rolled back. The car reservations are unaffected by the roll back because they were made prior to the savepoint.

```
BEGIN WORK
```

Make 15 car reservations.

```
SAVEPOINT
```

Savepoint number is 1. An attempt to make 15 hotel reservations fails because the designated hotel is full.

```
ROLLBACK WORK TO 1
SAVEPOINT
```

Savepoint number is 2. Make 15 hotel reservations at another hotel.

```
COMMIT WORK
```

Scoping of Transaction and Session Attributes

A set of attributes is associated with each transaction and user session. This section discusses the statements used to specify the following transaction and session attributes:

- priority
- isolation level
- label
- fill option
- constraint checking mode
- DML atomicity level

Each attribute can be specified in one or more of the statements listed in Table 2-1. You can issue such statements at any point in an application or ISQL session (with the exception of BEGIN WORK which cannot be issued within a transaction). However they may not take effect immediately, and the duration of their effect differs as described in the following paragraphs. Chapter 10 , “SQL Statements A - D,” and Chapters 11 and 12 contains complete syntax for each statement.

When beginning a transaction, attributes specified in a BEGIN WORK statement take effect immediately and remain in effect until the transaction ends, unless reset by a SET TRANSACTION, SET CONSTRAINTS, or SET DML ATOMICITY statement within the transaction.

Within a transaction, the attributes specified in a SET TRANSACTION, SET CONSTRAINTS, or SET DML ATOMICITY statement take effect immediately and remain in effect until the transaction ends, unless subsequently reset by such a statement. A SET SESSION

statement issued within a transaction has no effect on the present transaction, instead it takes effect for the next transaction and remains in effect for the duration of the session, unless reset by a subsequent BEGIN WORK, SET TRANSACTION, SET CONSTRAINTS, SET DML ATOMICITY, or SET SESSION statement.

Outside of a transaction, the attributes specified in a SET TRANSACTION or SET SESSION statement take effect for the next transaction, unless subsequently reset by such a statement or by a BEGIN WORK statement. The SET TRANSACTION, SET CONSTRAINTS, and SET DML ATOMICITY statements remain in effect for the duration of the transaction, unless subsequently reset. The SET SESSION statement remains in effect for the duration of the session, unless subsequently reset.

Table 2-1. shows these statements, the attributes associated with each, when each statement goes into effect after being issued and the scope of each statement's attributes if not reset by a subsequent statement:

Table 2-1. Transaction Attribute Scope

Statement	Attributes	When Effective	Duration of Attribute Setting	Begins a Transaction if None Already Begun
SET SESSION [a]	isolation level priority label constraint checking mode DML atomicity level fill option	for the next transaction	until the session ends	no
SET TRANSACTION	isolation level priority label constraint checking mode DML atomicity level	for the next or current transaction	until the transaction ends	no
SET CONSTRAINTS	constraint checking mode	for the current transaction	until the transaction ends	yes
SET DML ATOMICITY	DML atomicity level	for the current transaction	until the transaction ends	yes
BEGIN WORK	isolation level priority label fill option	when the transaction begins	until the transaction ends	yes

a. Note that SET SESSION issued within a transaction is *not* savepoint sensitive.

For example, you might write an application containing several transactions. Each transaction contains one or more SELECT statements. You want to ensure that all data

selected has been committed to the database. You know that the default isolation level for a session is RR, but RR does not provide the concurrency you need. At the beginning of the session, you set the isolation level to RC (read committed) for all transactions in the session, as follows:

```
.
.
.
SET SESSION ISOLATION LEVEL RC
.
.
.
```

Note that each transaction starts implicitly. In this example, there is no need for any `BEGIN WORK` statements. However, you might choose to include `BEGIN WORK` statements to make your code more readable or to set a different isolation level for a particular transaction.

```
SELECT * FROM PurchDB.OrderItems
     WHERE VendPartNumber = '2310'
COMMIT WORK
.
.
.
SELECT * FROM PurchDB.Vendors
     WHERE VendorNumber = 1234
COMMIT WORK
.
.
.
SELECT * FROM PurchDB.SupplyPrice
     WHERE VendorNumber = 1234 AND VendPartNumber = '2310'
COMMIT WORK
.
.
.
```

For more information on isolation levels, refer to Chapter 5 , “Concurrency Control through Locks and Isolation Levels,” in this manual.

Transaction Limits and Timeouts

The maximum number of concurrent transactions is determined by the *MaxTransactions* parameter of the DBECon file. Use either the `START DBE` statement or the `SQLUtil ALTDBE` command to set *MaxTransactions*. The `SQLUtil SHOWDBE` command displays the current setting of *MaxTransactions* in the DBECon file. If a session attempts to start a transaction, but the maximum number of concurrent transactions has already been reached, the new transaction is placed in the throttled wait queue. The transaction must wait until it reaches the head of the queue and one of the active transactions terminates. The throttled wait queue is serviced on a first in, first out basis. The transaction priority parameter of the `BEGIN WORK` statement determines which transaction is aborted to break a deadlock, not the transaction's position on the throttled wait queue.

If the transaction is still waiting when its timeout limit is reached, the transaction is

aborted. The timeout action can also be set to abort the command being processed instead of the entire transaction. Set the timeout limit for the DBEnvironment with the STARTDBE statement or the SQLUtil ALTDBE command. To specify a timeout limit for a particular session, use the SET USER TIMEOUT statement. Both SET SESSION and SET TRANSACTION have parameters to specify which action the system should take when a timer expires. The setting of timeout values is also incorporated into these commands. The SQLUtil SHOWDBE command displays the current, default, and maximum values of the timeout parameter in the DBECon file.

Monitoring Transactions

The SYSTEM.TRANSACTION pseudo-table contains the user identifier, connection-id, session identifier, transaction identifier, transaction priority, and isolation level of every current transaction. To view this information with ISQL, issue the following statement:

```
isql=> SELECT * FROM System.Transaction;
```

To identify the transactions on the throttle wait queue, query the SYSTEM.CALL pseudo-table as follows:

```
isql=> SELECT * FROM System.Call WHERE Status = 'Throttle wait';
```

For more information on transaction activity, consult Load subsystem in SQLMON, the ALLBASE/SQL on-line monitoring tool. SQLMON provides the following transaction information:

- total number of active and waiting transactions in the DBEnvironment
- total number of BEGIN WORK, COMMIT WORK, and ROLLBACK WORK statements executed in the DBEnvironment
- maximum number of transactions configured
- which sessions have active or waiting transactions
- which sessions have executed BEGIN WORK, COMMIT WORK, and ROLLBACK WORK statements

See the *ALLBASE/SQL Performance and Monitoring Guidelines* for more information on SQLMON.

Tips on Transaction Management

Keep transactions short. As the length of a transaction increases, so does the chance that other transactions are forced to wait for the locks it holds. In addition to increasing concurrency, short transactions minimize the amount of data that must be re-entered after a system crash. When archive logging is in effect, changes made to the database are written to the log file whenever a COMMIT WORK is issued. If the system crashes during a long transaction, a large number of uncommitted changes will be rolled back.

To shorten a transaction, place program statements not essential to the logical unit of work outside of the transaction. Retrieve all user input before the start of a transaction, to ensure that locks are not held if the user walks away from the terminal. Because terminal writes can also be time consuming, they should not be performed within a transaction.

Careful use of savepoints can decrease the amount of time locks are held, and reduces the need to resubmit transactions because part of a transaction was unsuccessful.

Set the maximum number of transactions (*MaxTransactions*) and timeout limit parameters correctly. If *MaxTransactions* is too low, transactions will wait for no reason. However, the overall throughput of the DBEnvironment may be reduced if *MaxTransactions* is too high. If the timeout limit is too low, transactions will abort, but if set too high, the session might wait indefinitely for a transaction slot.

Auditing DBEnvironments

Audit DBEnvironments are created with SQL statements that allow you to generate audit log records. Audit log records contain information that allows you to group log records for analysis with SQLAudit. The database operations you might analyze are UPDATE, INSERT, or DELETE operations, perhaps for security reasons.

Audit log records contain identifiers such as table names in contrast to non-audit database log records which contain identifiers such as page references and data. Audit log records are generated in addition to non-audit database log records.

A unique audit name specifies an audit DBEnvironment. Audit elements indicate which ALLBASE/SQL statement types generate audit log records. By default, statements that change *data* generate audit log records (INSERT, UPDATE, and DELETE statements); this default can also be specified explicitly by the *DATA AUDIT ELEMENTS* parameter. You can also optionally specify that log comment, data definition, authorization, or section statements (creation and deletion of sections) generate audit log records.

The Audit Tool, SQLAudit, is introduced below. SQLAudit is fully described in the *ALLBASE/SQL Database Administration Guide*. The *ALLBASE/SQL Database Administration Guide* describes how to create audit DBEnvironments and how to select records for audit. Chapter 10 , “SQL Statements A - D,” and Chapters 11 and 12 of this manual contain the detailed syntax to create audit DBEnvironments and partitions.

Partitions in Audit DBEnvironments

Partitions are divisions of DBEnvironments that contain one or more tables processed by SQLAudit as a unit. Partitions are specified in CREATE PARTITION, CREATE TABLE, and ALTER TABLE statements. In addition, default partition and comment partition numbers can optionally be specified.

Using Wrapper DBEnvironments

A wrapper DBEnvironment is a DBEnvironment created to wrap around the log files orphaned after a hard crash of a DBEnvironment. Wrapping log files means associating the files with a wrapper DBEnvironment. After a DBEnvironment becomes inaccessible, its log files are not associated with any DBEnvironment. These orphaned log files are then also inaccessible.

Wrapper DBEnvironments are usually used with inaccessible *audit* DBEnvironments, but they can be used to retrieve the log files of any inaccessible DBEnvironment.

After you wrap the log files, you can then try to extract audit information from the audit log records in the wrapped log files with SQLAudit by partition number.

Access to wrapped log files avoids having a gap in the ongoing record of audit information. The use of archive logging facilitates wrapper DBEnvironment use, but nonarchive logging does not prevent use of wrapper DBEnvironments.

To wrap log files, the orphaned log files marked Usable are first displayed and selected. Then, it must be ensured that each log file is inactive. A DBEnvironment is then created with the `START DBE NEW` statement and the new DBEnvironment is converted to a wrapper DBEnvironment with the `SQLUtil WRAPDBE` command.

NOTE Recovery of the database itself is a separate operation. It is recommended that the log files be wrapped before recovery operations.

For detailed information on database recovery and wrapper DBEnvironments, refer to the *ALLBASE/SQL Database Administration Guide.*

Using SQLAudit

SQLAudit is an ALLBASE/SQL utility program that can be used in conjunction with audit DBEnvironments to view the changes that have been made to the DBEnvironment. You use SQLAudit to audit only committed transactions. For security reasons, you need DBA authorization to use SQLAudit.

Refer to the “DBA Tasks and Tools” chapter of the *ALLBASE/SQL Database Administration Guide* for a full description of SQLAudit.

Application Programming

To use SQL statements in an application program, you embed the statements in source code, then use the ALLBASE/SQL preprocessor that supports the source language.

Preprocessor

The ALLBASE/SQL preprocessor performs the following tasks:

- Checks the syntax of SQL statements embedded in an application program.
- Translates embedded SQL statements into compilable C, FORTRAN, COBOL, or Pascal constructs that call ALLBASE/SQL external procedures at run time.
- Stores a module in the DBEnvironment.

A module contains a group of sections. A **section** consists of ALLBASE/SQL instructions for executing an SQL statement at run time. ALLBASE/SQL ensures that any objects referenced in the section exist and that current authorization criteria are satisfied. The optimal data access path is determined at preprocessing time rather than at run time which enhances runtime performance.

When an application program becomes obsolete, you can use the `DROP MODULE` statement to delete its module from the DBEnvironment and thus ensure the program can no longer operate on the databases in the DBEnvironment. For example:

```
DROP MODULE MyProgram
```

ALLBASE/SQL has the following statements that create modules when the information for an SQL statement cannot be completely defined in advance. These **dynamic preprocessing** statements are used in both programmatic and interactive environments:

```
PREPARE
EXECUTE
EXECUTE IMMEDIATE
```

In addition to the above statements, ALLBASE/SQL includes the following statements which *cannot* be used interactively:

```
BEGIN DECLARE SECTION     CLOSE CURSOR          DECLARE CURSOR
DELETE WHERE CURRENT      DESCRIBE              END DECLARE SECTION
FETCH                     INCLUDE               OPEN
REFETCH                   SQLEXPLAIN            UPDATE WHERE CURRENT
WHENEVER
```

Preprocessed programs receive messages from ALLBASE/SQL through the **SQL Communication Area**, called the SQLCA. Information is sent to ALLBASE/SQL through the **SQL Description Area**, called the SQLDA. These structures and the above statements are explained in detail along with examples in the ALLBASE/SQL application programming guides.

Authorization

ALLBASE/SQL authorization governs who can preprocess and execute a program that accesses a DBEnvironment as described here:

- To *preprocess* a program, you need DBA or CONNECT authority and the authorities needed to execute all activities against the database that are executed by the program. The module stored for the program is owned by the login name of the individual who invokes the preprocessor. A DBA, however, can associate the module with a different owner at preprocessing time. Other users can assign a group name as the module owner if they belong to the group.
- To *run* a program, you need either RUN authority or OWNER authority for the stored module. You also need the authority to start the DBE session as it is started in the program.

DBEnvironment Changes

Certain DBEnvironment changes can affect preprocessed programs. For example, one of the tables used by the program can be dropped from a database, or the authorities held by the module's owner can change. When you run a preprocessed program, ALLBASE/SQL automatically determines whether changes such as these have occurred. If any have, ALLBASE/SQL attempts to revalidate the affected sections. The only SQL statements that are executed at run time are those that operate on existing objects and those which the module's owner is authorized to execute.

Some changes do not affect successful execution of the program, but others can. If, for example, the owner of the program had SELECT and UPDATE authority for a table updated by the program and the UPDATE authority is later revoked, the program is no longer able to update that table. But if SELECT authority is revoked instead, the UPDATE statements for the table can still execute successfully.

Host Variables

Data is passed back and forth between a program and ALLBASE/SQL in host variables. SQL statements use both input and output host variables. Input host variables are used to transfer data into ALLBASE/SQL from the application. Output host variables move information from ALLBASE/SQL into the application.

An **indicator variable** is a special type of host variable. In the SELECT, FETCH, UPDATE, UPDATE WHERE CURRENT, and INSERT statements, the indicator variable is an input host variable whose value depends on whether an associated host variable contains a null value. If the indicator variable contains a negative number, then the associated host variable is null. If it contains a zero or positive number, the value in the host variable is not null.

In the SELECT and FETCH statements the indicator variable can be an output host variable and indicate that a value in the associated host variable is null or a column value is truncated. Host variable names are prefixed with a colon (:) when embedded in an SQL statement.

```
:PartNumber
:PartName
```

```
:PartNameInd
```

When host variables are used in an application outside of an embedded SQL statement, the host variable name is *not* prefixed by a colon.

Multiple-Row Manipulations

Programmatic SELECTs and INSERTs can operate only on a row at a time unless you use a cursor or the BULK option of the SELECT, INSERT, or FETCH statement.

A **cursor** is a pointer that you advance one row at a time. The BULK option is used to manipulate multiple rows with a *single* execution of the SELECT, INSERT, or FETCH statements. When you do bulk manipulations, input and output host variables must be arrays.

Using Multiple Connections and Transactions with Timeouts

A maximum of 32 simultaneous database environment connections can be established by means of an application program or ISQL. When accessing more than one DBEnvironment, there is no need to release one before connecting to another. Performance is greatly improved using this method rather than connecting to and releasing each DBEnvironment sequentially.

This multi-connect functionality is available in either of two modes. Single-transaction mode (the default) is standards compliant and allows one transaction at a time to be active across the currently connected set of DBEnvironments. Multi-transaction mode can be set to allow multiple, simultaneous transactions across the currently connected set of DBEnvironments.

Both local and remote DBEnvironments are accessible via multi-connect functionality. Remote connections require the installation of ALLBASE/NET on the client and on each related server.

The following sections discuss how to use multi-connect features:

- Connecting to DBEnvironments
- Setting the Current Connection
- Setting Timeout Values
- Setting the Transaction Mode
- Disconnecting from DBEnvironments

The sample DBEnvironment, PartsDBE, and three hypothetical DBEnvironments, SalesDBE, AccountingDBE, and BankDBE are used to provide examples in this section.

The *ALLBASE/SQL Advanced Application Programming Guide* contains further application programming information regarding multi-connect functionality.)

Connecting to DBEnvironments

With multi-connect functionality, you can specify a connection name each time you connect to a DBEnvironment by means of one of the following statements:

- CONNECT
- START DBE
- START DBE NEW
- START DBE NEWLOG

For example, in ISQL, the following CONNECT statement establishes a connection to PartsDBE and assigns a connection name for this connection:

```
isql=> CONNECT TO 'PartsDBE' AS 'Parts1';
```

In an application program, you can use either a string or, as in the following example, a host variable:

```
CONNECT TO 'PartsDBE' AS :Parts1
```

The connection name is used when setting the current connection, as described in the next section. It must be unique within an application and be assigned by means of either a character host variable or a string literal.

Which of the above statements you choose for assigning the connection name depends on the needs of your application. See Chapter 10 , “SQL Statements A - D,” and Chapters 11 and 12 for the complete syntax of each statement.

Setting the Current Connection

Within an application or ISQL, the current connection is set by the most recent statement that connects to or sets the connection to a DBEnvironment. In order for a multi-connect transaction to execute, the current connection must be set to the DBEnvironment in which the transaction will execute.

To change the current connection within a set of connected DBEnvironments, use a SET CONNECTION statement to specify the applicable connection name, as in the following example for ISQL:

```
isql=> SET CONNECTION 'Parts1';
```

In an application program, you can use either a string literal or, as in the following example, a host variable:

```
SET CONNECTION :Parts1
```

Remember, any SQL statement issued applies to the current connection.

NOTE Following a RELEASE or DISCONNECT CURRENT command, there is no current connection until a SET CONNECTION command is used to set the current connection to another existing connection, or a new connection is established by using the CONNECT, START DBE, START DBE NEW, or START DBE NEW LOG commands.

Setting Timeout Values

Be sure to set a timeout value when using multiple connections to avoid undetected deadlocks and undetected wait conditions. An undetected deadlock is possible when multi-transaction mode is used in conjunction with more than one DBEnvironment with multiple applications accessing the same DBEnvironments at the same time. An undetected wait condition is possible when multi-transaction mode is used with multiple connections to the same DBEnvironment within a single ISQL session or application.

A timeout value can be set with any of the following:

- `START DBE`
- `START DBE NEW`
- `START DBE NEWLOG`
- `SQLUtil ALTDBE`
- `SET USER TIMEOUT`
- `SET SESSION USER TIMEOUT`
- `SET TRANSACTION USER TIMEOUT`

The first four methods provide a means of setting timeout values at the DBEnvironment level. The `SET USER TIMEOUT` statement provides a way of setting transaction, session, or application specific timeout values. The range of possible values is zero (no wait) to the specified maximum in the DBECon file for a given DBEnvironment.

For a multi-connect application operating in multi-transaction mode, it is *essential* to use the `SET USER TIMEOUT` statement to avoid an undetectable deadlock or wait condition. For information regarding transaction modes, see the following section, “Setting the Transaction Mode.”

The following general example shows how to set user timeout values:

1. Put multi-transaction mode in effect.

```
SET MULTITRANSACTION ON
```

2. Connect to the PartsDBE DBEnvironment.

```
CONNECT TO 'PartsDBE' AS 'Parts1'
```

3. Set the timeout value for the PartsDBE connection to an appropriate number of seconds. In this case, the application will wait five minutes for system resources when accessing the PartsDBE DBEnvironment.

```
SET USER TIMEOUT 300 SECONDS
```

4. Connect to the SalesDBE DBEnvironment.

```
CONNECT TO 'SalesDBE' AS 'Sales1'
```

5. Set the timeout value for the SalesDBE connection to an appropriate number of seconds. In this case, your application will wait 30 seconds for system resources when accessing the SalesDBE DBEnvironment.

```
SET USER TIMEOUT 30 SECONDS
```

6. Set the current connection to Parts1.

```
SET CONNECTION 'Parts1'
```

7. Begin a transaction for PartsDBE. If this transaction waits for system resources more than five minutes, it will time out and return an error message.

```
BEGIN WORK RC

SELECT  PartNumber, PartName, SalesPrice
  FROM  PurchDB.Parts
 WHERE  PartNumber BETWEEN 20000 AND 21000
```

   If DBERR 2825 is returned, the transaction has timed out, and your application must take appropriate action.

   .
   .
   .

8. Set the current connection to Sales1.

```
SET CONNECTION 'Sales1'
```

9. Begin a transaction for SalesDBE. If this transaction waits for system resources more than 30 seconds, it will timeout and return an error message to the application.

```
BEGIN WORK RC

BULK SELECT  PartNumber, Sales
       FROM  Owner.Sales
      WHERE  PartNumber = '1123-P-20'
        AND  SaleDate BETWEEN '1991-01-01' AND '1991-06-30'
```

```
.
.
.
```

If DBERR 2825 is returned, the transaction has timed out, and you must take appropriate action.

Further discussion of timeout functionality is provided in the *ALLBASE/SQL Advanced Application Programming Guide*.

Setting the Transaction Mode

The SET MULTITRANSACTION statement allows you to switch between single-transaction mode and multi-transaction mode. Single-transaction mode implies sequential execution of transactions across a set of DBEnvironment connections. When your application requires multiple, simultaneous transactions, you must choose multi-transaction mode.

WARNING **When using multi-transaction mode, be sure the current timeout value for all connections is set to a value other than NONE (infinity). This eliminates the possibility of an infinite wait if an undetectable deadlock or wait condition occurs.**

Using Single-Transaction Mode

If your application contains queries for two or more databases and you want to sequentially execute a single transaction against each database, you can use single-transaction mode. This mode is the default and is standards compliant. The following example illustrates the use of single-transaction mode in ISQL:

1. Put single-transaction mode in effect.

```
isql=> SET MULTITRANSACTION OFF;
```

2. Connect to two DBEnvironments.

```
isql=> CONNECT TO 'PartsDBE' AS 'Parts1';
isql=> CONNECT TO 'SalesDBE' AS 'Sales1';
```

3. Set the current connection to Parts1.

```
isql=> SET CONNECTION 'Parts1';
```

4. Begin a transaction for PartsDBE.

```
isql=> BEGIN WORK RC;

isql=> SELECT  PartNumber, PartName, SalesPrice
> FROM PurchDB.Parts
> WHERE PartNumber BETWEEN 20000 AND 21000;

.
.
.
```

5. End the PartsDBE transaction.

```
isql=>  COMMIT WORK;
```

6. Set the current connection to Sales1.

```
isql=> SET CONNECTION 'Sales1';
```

7. Begin a transaction for SalesDBE.

```
isql=> BEGIN WORK RC;

isql=>  SELECT PartNumber, Sales
> FROM Owner.Sales
> WHERE PartNumber = '1123-P-20';
              .
              .
              .
```

8. End the SalesDBE transaction.

```
isql=> COMMIT WORK;
```

Using Multi-Transaction Mode with Multiple DBEnvironments

The SET MULTITRANSACTION ON statement enables multiple implied or explicit BEGIN WORK statements across the set of currently connected database environments, with a maximum of one active transaction per database connection. While in multi-transaction mode, an application can hold resources in more than one DBEnvironment at a time.

Suppose your application is querying one DBEnvironment and inserting the query result into another DBEnvironment. You decide to use bulk processing with multi-transaction functionality. The DBEnvironments could be on different systems (using ALLBASE/NET) or on the same system, as in the following example using host variables:

1. Put multi-transaction mode in effect.

```
SET MULTITRANSACTION ON

   DECLARE PartsCursor
CURSOR FOR
    SELECT OrderNumber, VendorNumber, OrderDate
      FROM PurchDB.Orders
     WHERE OrderDate > Yesterday
```

2. Connect to two DBEnvironments and set an appropriate timeout value for each.

```
CONNECT TO 'PartsDBE' AS 'Parts1'
SET USER TIMEOUT 180 SECONDS

CONNECT TO 'Part2DBE' AS 'Parts2'
SET USER TIMEOUT 30 SECONDS
```

3. Set the current connection to Parts1.

```
SET CONNECTION 'Parts1'
```

4. Begin a transaction for PartsDBE.

```
BEGIN WORK RC

OPEN PartsCursor

BULK FETCH  PartsCursor
      INTO  :PartsArray, :StartIndex, :NumberOfRows
```

5. If there are qualifying rows, set the current connection to Parts2.

```
SET CONNECTION 'Parts2'
```

6. Begin a transaction for Parts2DBE.

```
BEGIN WORK RC
```

   At this point, there are two active transactions.

```
BULK INSERT
            INTO  PurchDB2.Orders2
          VALUES  (:PartsArray, :StartIndex, :NumberOfRows)
```

7. Test the sqlcode field of the sqlca. If it equals -2825, a timeout has occurred, and the transaction was rolled back. Take appropriate action.

8. End the transaction.

```
COMMIT WORK
```

   There is now one open transaction holding resources in PartsDBE.

9. Set the current connection to Parts1.

```
SET CONNECTION 'Parts1'
```

10. If there are more rows to fetch, loop back to execute the FETCH statement again. Otherwise, end the fetch transaction.

```
COMMIT WORK
      .
      .
      .
```

Note that in multi-transaction mode, the SET MULTITRANSACTION OFF statement is valid only if no more than one transaction is active. In addition, if an active transaction exists, it must have been initiated in the current connection, otherwise the SET MULTITRANSACTION OFF statement returns an error (DBERR 10087).

Using Multi-Transaction Mode with One DBEnvironment

Even when your application connects to just one DBEnvironment, you might require multiple, simultaneous transactions to be active. This technique involves connecting to one DBEnvironment multiple times and specifying a unique connection name each time. In this case, you issue a SET CONNECTION statement for the appropriate connection name before beginning each transaction. Note that just one transaction can be active per connection.

For example, suppose you want to keep a record of each time access to a particular table is

attempted. From a menu, the user chooses to view account information and specifies an account number. Before giving this information, the application logs the fact that the user is requesting it. The following pseudocode example illustrates how you might code two simultaneous transactions, each one accessing BankDBE using host variables:

1. Put multi-transaction mode in effect.

```
SET MULTITRANSACTION ON

   DECLARE  BankCursor
CURSOR FOR
    SELECT  TransactionType,
            DollarAmount,
            BankNumber
      FROM  Accounts
     WHERE  AccountNumber = :AccountNumber
```

2. Connect two times to BankDBE. Be sure to specify an appropriate timeout value for each connection.

```
CONNECT TO 'BankDBE' AS 'Bank2'
SET USER TIMEOUT 30 SECONDS

CONNECT TO 'BankDBE' AS 'Bank1'
SET USER TIMEOUT 30 SECONDS
```

   The user enters an account number.

3. Begin a transaction for the Bank1 connection.

```
BEGIN WORK RC

      .
      .
      .
```

4. Execute the following security audit subroutine:

   Set the current connection to Bank2.

```
SET CONNECTION 'Bank2'
```

   Begin a second transaction for BankDBE.

```
BEGIN WORK RC
```

   A security audit trail record is written whether or not the query in the first transaction completes.

```
INSERT INTO  BankSecurityAudit
     VALUES  (:UserID, :AccountNumber, CURRENT_DATETIME)
```

   Test the sqlcode field of the sqlca. If it equals -2825, a timeout has occurred, and the transaction was rolled back. Take appropriate action.

   End the transaction.

```
COMMIT WORK
```

Set the current connection to Bank1.

```
SET CONNECTION 'Bank1'
```

5. Return from the subroutine to complete the open transaction:

```
.
.
.
OPEN BankCursor

BULK FETCH  BankCursor
      INTO  :BankArray, :StartIndex, :NumberOfRows
.
.
.
```

Disconnecting from DBEnvironments

The DISCONNECT statement provides a means of closing one or all active connections within an application. An active connection is a connection established within the application that has not been released, stopped, or disconnected.

Your application might require that all connections be terminated when the application completes. In some cases, it might be desirable to terminate a specific connection at another point in the application.

In the following example, three database connections are established, and one is terminated immediately after a transaction completes:

1. Put multi-transaction mode in effect.

```
SET MULTITRANSACTION ON
```

2. Connect three times and set a timeout value for each connection. In this case, the DBEnvironment names and the connection names are specified as host variables.

```
CONNECT TO 'PartsDBE' AS 'Parts1'
SET USER TIMEOUT 60 SECONDS

CONNECT TO 'SalesDBE' AS 'Sales1'
SET USER TIMEOUT 60 SECONDS

CONNECT TO 'AccountingDBE' AS 'Accounting1'
SET USER TIMEOUT 60 SECONDS

SET CONNECTION 'Parts1'
```

3. Begin a transaction for PartsDBE.

```
BEGIN WORK RC

.
.
.
```

4. End the transaction that was initiated for the Parts1 connection and terminate the connection.

```
COMMIT WORK
DISCONNECT 'Parts1'
```

5. Set the current connection to 'Sales1'.

```
SET CONNECTION 'Sales1'
```

6. Begin transaction for SalesDBE.

```
BEGIN WORK RC

     .
     .
     .
```

7. Set the current connection to Accounting1.

```
SET CONNECTION 'Accounting1'
```

8. Begin transaction for Accounting1.

```
BEGIN WORK RC

     .
     .
     .
```

9. End both open transactions and disconnect the two active connections. Note that the COMMIT WORK statement is issued for the current connection's transaction.

```
COMMIT WORK

SET CONNECTION 'Sales1'
COMMIT WORK

DISCONNECT ALL
```

Note that following the execution of a `DISCONNECT CURRENT` statement, no current connection exists. To establish a current connection following a `DISCONNECT CURRENT` statement, you must either establish a connection or set the connection.

Administering a Database

Activities that protect and maintain a DBEnvironment and its databases are collectively referred to as **database administration**. Several of the SQL statements are used in the following database administration activities:

- Security management
- Restructuring
- Space management
- Logging
- Recovery
- DBEnvironment management
- DBEnvironment statistics maintenance

Refer to the *ALLBASE/SQL Database Administration Guide* for full details on these and other matters of database administration. That manual provides full information on SQLUtil, which is the primary tool for DBEnvironment reconfiguration and backup.

Understanding the System Catalog

The system catalog is a collection of tables and views that contain data about the following:

- Tables and views in a DBEnvironment
- Any indexes, hash structures, constraints, and rules defined for tables
- DBEFiles and DBEFileSets in the DBEnvironment
- Specific authorities granted to each user
- Programs that can access data in the DBEnvironment
- Current DBEnvironment statistics
- Temporary space for sorts
- Procedures

ALLBASE/SQL uses the system catalog to maintain data integrity and to optimize data access. The system views are primarily a tool for the DBA. Initially, only the DBA can access these views. Other users need to be granted SELECT authority by the DBA to access them. Users without SELECT authority can retrieve descriptions of database objects they own from the CATALOG views. For information on system and catalog views, refer to chapter “System Catalog” in the *ALLBASE/SQL Database Administration Guide*.

When a DBEnvironment is first configured, the information in the system catalog describes the system tables and views themselves. As database objects are defined, their

definitions are stored in the system catalog. As database activities occur, most of the information in the catalog is updated automatically, so the system catalog provides an up-to-date source of information on a DBEnvironment.

Immediately following an `UPDATE STATISTICS` statement, the views in the system catalog, summarized in Table 2-2, are a source of up-to-date information on a DBEnvironment and the structure and use of its databases. Refer to the *ALLBASE/SQL Database Administration Guide* for additional information on the system catalog.

Table 2-2. System Views

View Name	Purpose
SYSTEM.ACCOUNT	Identifies the I/O usage of current database sessions.
SYSTEM.CALL	Identifies current internal calls.
SYSTEM.CHECKDEF	Contains the search condition defined for each table check constraint. Contains the column name for each column check constraint.
SYSTEM.COLAUTH	Identifies users and groups and their column update and reference authorities.
SYSTEM.COLDEFAULT	Describes the default value of each column defined with a non-NULL default.
SYSTEM.COLUMN	Contains the definition of each column in each table and view.
SYSTEM.CONSTRAINT	Contains information on integrity constraints.
SYSTEM.CONSTRAINTCOL	Contains information on the columns within unique and referential constraints.
SYSTEM.CONSTRAINTINDEX	Describes each unique and referential constraint index.
SYSTEM.COUNTER	Describes the status of internal system counters.
SYSTEM.DBEFILE	Describes the characteristics of each DBEFile.
SYSTEM.DBEFILESET	Describes the characteristics of each DBEFileset.
SYSTEM.GROUP	Describes each authorization group.
SYSTEM.HASH	Describes each hash structure.
SYSTEM.IMAGEKEY	Describes each Master and Detail Dataset key associated with TurboIMAGE databases attached to the DBE.
SYSTEM.INDEX	Describes each index.
SYSTEM.INSTALLAUTH	Identifies users and authorization groups that have been granted `INSTALL` authority.
SYSTEM.MODAUTH	Identifies users and groups and the programs they can run.
SYSTEM.PARAMDEFAULT	Describes the default value of each parameter defined with a non-NULL default.
SYSTEM.PARAMETER	Describes each parameter of each procedure.

Table 2-2. System Views

View Name	Purpose
SYSTEM.PARTITION	Contains partition information.
SYSTEM.PLAN	Stores the result of one GENPLAN for each session.
SYSTEM.PROCAUTH	Identifies users and groups and the procedures they can execute.
SYSTEM.PROCEDURE	Describes each procedure.
SYSTEM.PROCEDUREDEF	Contains the definition of each procedure.
SYSTEM.PROCRESULT	Describes procedure result columns.
SYSTEM.RULE	Describes each rule.
SYSTEM.RULECOLUMN	Describes columns an update rule checks for.
SYSTEM.RULEDEF	Contains the referencing, WHERE, and EXECUTE PROCEDURE clause of each rule.
SYSTEM.SECTION	Describes stored modules and views.
SYSTEM.SETOPTINFO	Contains SETOPT settings for optimizing specific stored sections.
SYSTEM.SPACEAUTH	Identifies users and groups and what DBEFileSets they can use when creating tables, or stored sections.
SYSTEM.SPACEDEFAULT	Identifies the default DBEFileSet to use for a new table or stored section.
SYSTEM.SPECAUTH	Identifies users and groups who have special authorities.
SYSTEM.TABAUTH	Identifies users and groups and table/view operations they can perform.
SYSTEM.TABLE	Contains a description of each table and view in the DBEnvironment, including size, owner, and associated DBEFileSet.
SYSTEM.TEMPSPACE	Defines the TempSpace locations.
SYSTEM.TPINDEX	Describes third-party indexes used in TurboIMAGE databases attached to the DBE.
SYSTEM.TRANSACTION	Identifies transactions.
SYSTEM.USER	Identifies users currently using the database.
SYSTEM.VIEWDEF	Contains the SELECT statement that created each view defined in the system.

3 SQL Queries

This chapter describes SQL queries, through which you access the data in database tables. The following sections are presented:

- Using the SELECT Statement
- Simple Queries
- Complex Queries
- Using GENPLAN to Display the Access Plan
- Updatability of Queries

The other kinds of data manipulation, using the INSERT, UPDATE, and DELETE statements, were presented in the chapter “Using ALLBASE/SQL.”

Using the SELECT Statement

Use the SELECT statement to compose queries. The SELECT statement consists of the following components:

1. Select list
2. INTO clause
3. FROM clause
4. WHERE clause
5. GROUP BY clause
6. HAVING clause
7. ORDER BY clause

The select list and FROM clause are required; all other components of this statement are optional. The following example does not contain an INTO clause. Note the reference numbers identifying the above components:

```
                        1
                        |
          ______________|_____________
         |                            |
         |                            |
   SELECT PartNumber, COUNT(VendorNumber)
     FROM PurchDB.SupplyPrice            ---3
    WHERE DeliveryDays < 25              ---4
 GROUP BY PartNumber                     ---5
   HAVING COUNT(VendorNumber) > 2        ---6
 ORDER BY PartNumber                     ---7
```

The result is presented in the form of a table, called a **query result**. The result table (shown next) for this example has two columns: part numbers and a count of vendors who supply each part. The query result has rows only for parts that can be delivered in fewer than 25 days by more than two suppliers. The rows are ordered in ascending order by PartNumber.

```
----------------+-----------
PARTNUMBER      |(EXPR)
----------------+-----------
1123-P-01       |          4
1133-P-01       |          3
1243-MU-01      |          3
1323-D-01       |          3
1353-D-01       |          3
1433-M-01       |          3

  .
  .
  .
```

The **select list** identifies the columns you want in the query result. In the above example, the (EXPR) column contains the vendor count specified as COUNT(VendorNumber) in the select list. Computations of this kind are called **aggregate functions**, which are defined in the "Expressions" chapter. The count function counts rows, in this case rows that satisfy the conditions set up in the SELECT statement clauses.

This example contains no INTO clause because host variables are not being used. The INTO clause is used in application programs to identify host variables for storing the query result. For more information on host variables, refer to the appropriate ALLBASE/SQL application programming guide.

The FROM clause identifies tables and views from which data is to be retrieved, in this case, PurchDB.SupplyPrice.

The WHERE clause specifies a search condition for screening rows. Search conditions are comparisons and other operations you can have ALLBASE/SQL perform in order to screen rows for your query result. The "Search Conditions" chapter defines the ALLBASE/SQL search conditions. In this case, the search condition states that rows in the query result must contain information for parts that can be delivered in fewer than 25 days.

The GROUP BY clause tells ALLBASE/SQL how to group rows *before* performing an aggregate function in the select list. The rows that satisfy the WHERE clause are grouped. In this example, the rows are grouped by PartNumber. Then ALLBASE/SQL counts the number of vendors that supply each part. The result is a vendor count for each part number.

The HAVING clause screens the groups. In the above example, data for only groups having a vendor count greater than two becomes part of the query result.

The ORDER BY clause sorts the query result rows in order by specified column, in this case, PartNumber.

Simple Queries

A simple query contains a single SELECT statement and typically has a simple comparison predicate in the WHERE clause. The SELECT statement can be used to retrieve data from single tables or from multiple tables. To retrieve data from multiple tables, you join the tables on a common column value. In the following example, ALLBASE/SQL joins rows from the PurchDB.SupplyPrice and PurchDB.Parts tables that have the same PartNumber, as specified in the WHERE clause:

```
SELECT PartName, VendorNumber
  FROM PurchDB.SupplyPrice, PurchDB.Parts
 WHERE PurchDB.SupplyPrice.PartNumber =
       PurchDB.Parts.PartNumber
```

The query result is as follows:

```
--------------------------------|------------
PARTNAME                        |VENDORNUMBER
--------------------------------|------------
Central Processor               |        9002
Central Processor               |        9003
Central Processor               |        9007
Central Processor               |        9008
.
.
.
```

The following statement, using the explicit JOIN syntax, produces the same query result as the statement above.

```
SELECT PartName, VendorNumber
  FROM PurchDB.SupplyPrice
  JOIN PurchDB.Parts
    ON PurchDB.SupplyPrice.PartNumber =
       PurchDB.Parts.PartNumber
```

The same query result is also obtained using the following statement:

```
SELECT PartName, VendorNumber
  FROM PurchDB.SupplyPrice
  JOIN PurchDB.Parts
 USING (PartNumber)
```

The following NATURAL JOIN syntax would also produce the same result:

```
      SELECT PartName, VendorNumber
        FROM PurchDB.SupplyPrice
NATURAL JOIN PurchDB.Parts
```

In the four examples above, if a SELECT * is used instead of explicitly naming the displayed columns in the select list, the query result shows some differences. For the first two examples, the PartNumber column is displayed twice, once for each of the tables from which it is derived. For the last two examples, where the USING (*ColumnList*) clause or the NATURAL JOIN are used, the common columns are coalesced into a single column in the query result.

ALLBASE/SQL creates a row for the query result whenever a part number in table PurchDB.Parts matches a part number in table PurchDB.SupplyPrice, for example:

```
PurchDB.Parts:
PARTNUMBER  PARTNAME              SALESPRICE
-------------------------------------------
1123-P-01   Central processor         500.00

.
.
.

PurchDB.SupplyPrice:
PARTNUMBER  VENDORNUMBER    ...    DISCOUNTQTY
-----------------------------------------------
1123-P-01           9002                     1
1123-P-01           9003                     5
1123-P-01           9007                     3
1123-P-01           9008                     5

.
.
.
```

Any row containing a null part number is excluded from the join, as are rows that have a part number value in one table, but not the other.

You can also join a table to itself. This type of join is useful when you want to compare data in a table with other data in the same table. In the following example, table PurchDB.Parts is joined to itself to determine which parts have the same sales price as part 1133-P-01:

```
SELECT q.PartNumber, q.SalesPrice
 FROM PurchDB.Parts p,
      PurchDB.Parts q
WHERE p.SalesPrice = q.SalesPrice
  AND p.PartNumber  = '1133-P-01'
```

The same query result is obtained from the following explicit join syntax:

```
SELECT q.PartNumber, q.SalesPrice
  FROM Purchdb.Parts p
  JOIN Purchdb.Parts q
    ON p.SalesPrice = q.SalesPrice
   AND p.PartNumber = '1133-P-01'
```

To obtain the query result, ALLBASE/SQL joins one copy of the table with another copy of the table, as follows, using the **join condition** specified in the WHERE clause or the ON *SearchCondition3* clause:

- You name each copy of the table in the FROM clause by using a **correlation name**. In this example, the correlation names are *p* and *q*. You use the correlation names to qualify column names in the select list and other clauses in the query.
- The join condition in this example specifies that for each sales price, the query result should contain a row only when the sales price matches that of part 1133-P-01.

ALLBASE/SQL joins a row in q.PurchDB.Parts to a row in p.PurchDB.Parts having a part number of 1133-P-01 whenever the SalesPrice value in q.PurchDB.Parts matches that for 1133-P-01.

The query result for this self-join appears as follows:

```
----------------------|--------------
PARTNUMBER            |SALESPRICE
----------------------|--------------
1133-P-01             |        200.00
1323-D-01             |        200.00
1333-D-01             |        200.00
1523-K-01             |        200.00
```

For a two or more table join, if you do not use a join predicate in the ON *SearchCondition3* clause or the WHERE clause, or if there are no common columns with which to join the tables in a natural join, the result of the join is the **Cartesian product**. In the simplest case, for a two table join, the Cartesian product is the set of rows which contains every possible combination of each row in the first table concatenated with each row in the second table.

As an example, consider the simple Parts and Colors tables:

```
                Parts                              Colors

        PartNumber   PartName               PartNumber   Color
        ----------------------              ----------------------
             1       Widgit                    NULL      Red
           NULL      Thing                       2       NULL
             3       NULL                        3       Green
```

The following query generates the Cartesian product:

```
SELECT p.PartNumber, PartName, c.PartNumber, Color FROM Parts p, Colors c
```

The Cartesian product is shown in the query result:

```
SELECT p.PartNumber, PartName, c.PartNumber, Color FROM Parts p, Colors c
--------------+------------+----------------+-------------------
PARTNUMBER     |PARTNAME    |PARTNUMBER      |COLOR
--------------+------------+----------------+-------------------
             1 |Widgit      |            NULL|Red
             1 |Widgit      |               2|NULL
             1 |Widgit      |               3|Green
          NULL |Thing       |            NULL|Red
          NULL |Thing       |               2|NULL
          NULL |Thing       |               3|Green
             3 |NULL        |            NULL|Red
             3 |NULL        |               2|NULL
             3 |NULL        |               3|Green
```

The same algorithm is used to form the Cartesian product for a three or more table join. Thus, it can be said that the Cartesian product of a set of *n* tables is the table consisting of all possible rows *r*, such that *r* is the concatenation of a row from the first table, a row from the second table,..., and a row from the *n*th table.

As you can see, the Cartesian product for even a small two table join is much larger than the source tables. For a three or more table join of several large tables, the Cartesian product can be so large as to cause you to run out of memory and generate an error. Therefore it is important to be sure that you include the appropriate join predicate in your queries and to be sure that you specify columns common to the tables being joined.

In the example above, NULLs are included in the tables to show the difference between the behavior of NULLs in the production of the Cartesian product and the behavior of NULLs when a common column is specified in the WHERE clause join predicate.

Consider the following query:

```
SELECT p.PartNumber, PartName, c.PartNumber, Color
  FROM Parts p, Colors c
 WHERE p.PartNumber = c.PartNumber
```

The query result for the query is as follows:

```
SELECT p.PartNumber, PartName, c.PartNumber, Color FROM Parts p, Colors c....
---------------+------------+----------------+-------------------
PARTNUMBER     |PARTNAME    |PARTNUMBER      |COLOR
---------------+------------+----------------+-------------------
             3 |NULL        |               3|Green
```

The only rows selected for the query result are those rows for which the join predicate (p.PartNumber = c.PartNumber) evaluates to true. Because NULL has an undetermined value, for the cases where the values of the predicate are NULL = NULL, the value of the predicate is undetermined, and the row is not selected.

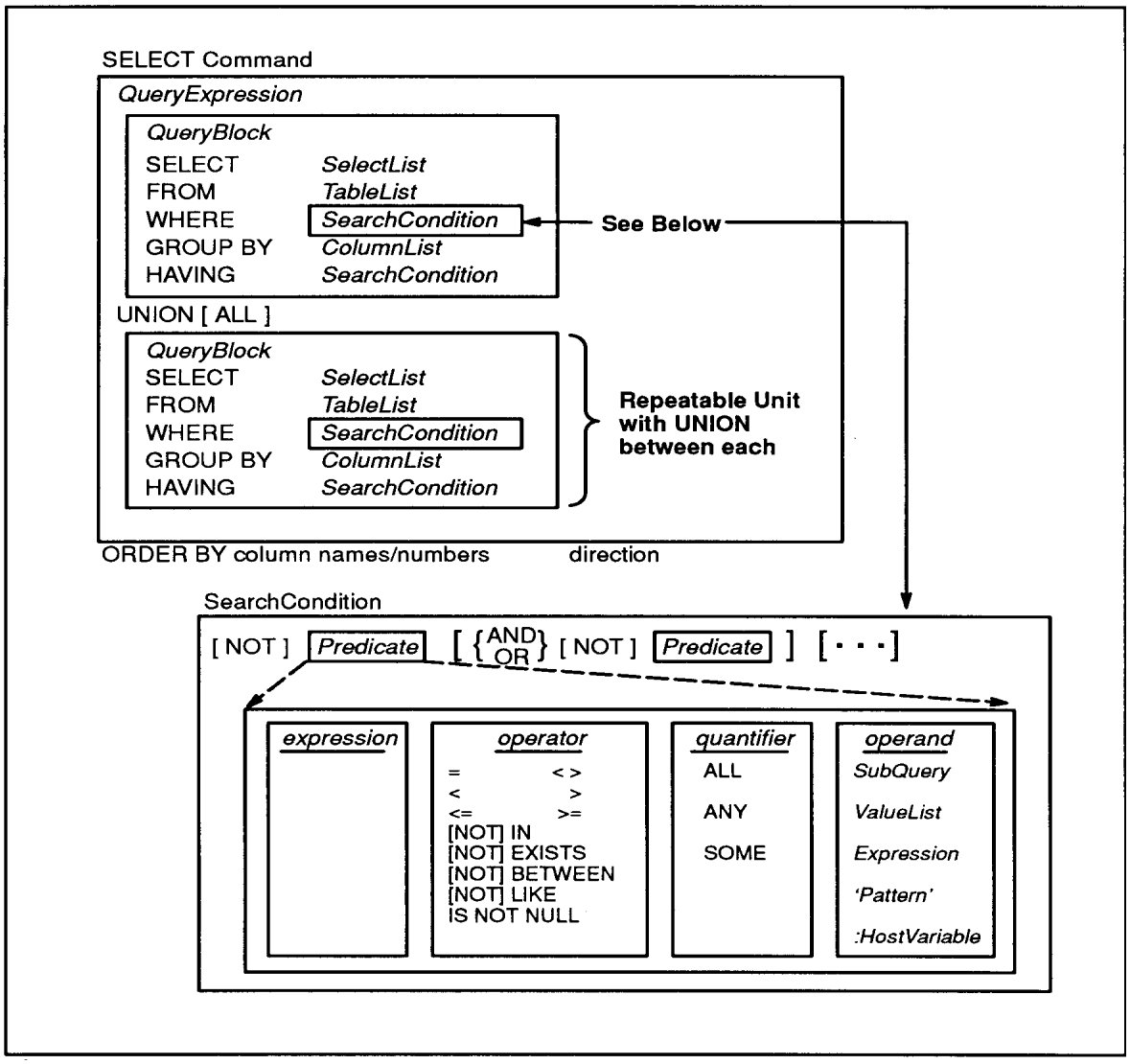
However, for the Cartesian product shown in the prior example, due to the absence of a join predicate, rows with NULLs in the common column are selected because the operation is the simple concatenation of the rows, regardless of value.

Complex Queries

In addition to the simple queries shown in the previous section, you can create complex queries, which may contain more than one SELECT statement. At the highest level, a query is a SELECT statement, which consists of a query expression followed by an optional ORDER BY clause. At the next lower level, you can combine different query blocks into a single query expression with the UNION operator. Lower still, inside each query block is an optional search condition, which can contain predicates that incorporate subqueries. A subquery is always a single query block (SELECT) that can contain other subqueries but cannot contain a UNION. A query expression can contain a maximum of 16 query blocks from all sources, including UNION, subqueries, and the outer query block.

Figure 3-1. shows the range of possibilities for complex queries.

Figure 3-1. Range of Complex Query Types



You can create a complex query by using the following:

- UNION operator, which allows you to take the union of all rows returned by several query blocks in one SELECT statement.
- Subqueries (also known as **nested queries**), which allow you to embed a query block within the search condition of an outer SELECT statement.
- Special predicates, such as ANY, ALL, SOME, EXISTS, and IN, which allow you to compare the value of an expression with the value of special structures and subqueries.

The next sections describe each type of complex query with examples.

UNION Queries

A SELECT statement can consist of several query blocks connected by UNION or UNION ALL statements. Each individual SELECT statement returns a query result which is a set of rows selected from a specified table or tables. The union of these query results is presented as a table that consists of all rows appearing in one or more of the original query results.

If only the UNION statement is used, all duplicate rows are removed from the final set of rows. In this case, the maximum size of a tuple in the query result is given by the following formula:

*(SelectListItems +1)*2 + (SumListLengths) <= 4000*

where:

SelectListItems is the number of items in the select list.

SumListLengths is the sum of the lengths of all the columns in the select list.

At compile time, *SumKeyLengths* is computed assuming columns of NULL and VARCHAR contain no data. At run time, the actual data lengths are assumed.

If the UNION ALL operator is used, duplicates are not removed. Candidates for duplicate removal are evaluated by comparing entire tuples, not just a single field. Only if two or more rows are entirely alike are the duplicates removed. In the case of the UNION ALL operator, the maximum size of a tuple in the query result is 3996 bytes, as it is for a non-UNION query expression. You cannot use LONG columns in a UNION statement.

Suppose you wanted to find out the part number for all parts that require 30 days or more for delivery, or are supplied by the vendor whose number is 9002. The following query delivers this information using the UNION form of the SELECT statement:

```
SELECT PartNumber
  FROM PurchDB.SupplyPrice
 WHERE DeliveryDays >= 30

UNION

SELECT PartNumber
  FROM PurchDB.SupplyPrice
 WHERE VendorNumber = 9002
```

```
ORDER BY PartNumber
   ---------------
   PARTNUMBER
   ---------------
   1123-P-01
   1133-P-01
   1143-P-01
   1153-P-01
   1223-MU-01
   1233-MU-01
   1323-D-01
   1333-D-01
   1343-D-01
   1523-K-01
   1623-TD-01
   1823-PT-01
```

Note that no rows are duplicated. When the UNION statement is not qualified by the ALL statement, all duplicate rows are removed from the query result. Notice that the ORDER BY clause must be at the *end* of the SELECT statement. It cannot be included in the separate query expressions that make up the overall statement. Only the final query result can be ordered.

If the UNION ALL statement is used in the previous query, the result can contain duplicate rows. The following example flags duplicate rows with two types of arrows that are described below:

```
   ---------------
   PARTNUMBER
   ---------------
   1123-P-01
   1123-P-01  <----
   1123-P-01  <---+
   1133-P-01
   1133-P-01  <---+
   1143-P-01
   1143-P-01  <----
   1153-P-01
   1153-P-01  <---+
   1223-MU-01
   1233-MU-01 <----
   1323-D-01
   1333-D-01
   1343-D-01
   1523-K-01
   1623-TD-01
   1823-PT-01
```

In the above example, rows are duplicated for the following:

- More than one vendor supplies some parts (these duplicates are indicated by <----)
- Vendor 9002 supplies some parts that take 30 or more days to deliver (these duplicates are indicated by <---+)

Note that you could get the same information in other ways. For example, you could use two separate queries. Alternatively, you could use two predicates in the search condition joined by the OR operator as follows:

```
SELECT PartNumber
    FROM PurchDB.Supplyprice
   WHERE DeliveryDays >= 30 OR
         VendorNumber = 9002
ORDER BY PartNumber
```

This query still contains duplicate rows where more than one vendor supplies a given part; but no duplicates are caused by vendor 9002 supplying some parts, and that some of these take 30 or more days to deliver. The duplicates could be eliminated by using the SELECT DISTINCT instead of SELECT statement.

Using Character Constants with UNION

If you want to see which SELECT statement in the UNION statement contributed each row to the query result, you can include character constants in your SELECT statements. A second column is then generated that shows the originating query block for each row, as in this example:

```
    SELECT PartNumber, 'deliverydays >= 30'
      FROM PurchDB.SupplyPrice
     WHERE DeliveryDays >= 30

UNION ALL
    SELECT PartNumber, 'supplied by 9002  '
     FROM PurchDB.SupplyPrice
    WHERE VendorNumber = 9002

 ORDER BY PartNumber
----------------+------------------
PARTNUMBER      |(CONST)
----------------+------------------
1123-P-01       |deliverydays >= 30
1123-P-01       |deliverydays >= 30  <----
1123-P-01       |supplied by 9002
1133-P-01       |supplied by 9002
1133-P-01       |deliverydays >= 30
1143-P-01       |deliverydays >= 30
1143-P-01       |deliverydays >= 30  <----
1153-P-01       |deliverydays >= 30
1153-P-01       |supplied by 9002
1223-MU-01      |deliverydays >= 30
1233-MU-01      |deliverydays >= 30
1323-D-01       |deliverydays >= 30
1333-D-01       |deliverydays >= 30
1343-D-01       |deliverydays >= 30
1523-K-01       |deliverydays >= 30
1623-TD-01      |deliverydays >= 30
1823-PT-01      |supplied by 9002
1923-PA-01      |supplied by 9002
```

The indicated duplicate rows would have been removed if the example contained the UNION statement instead of UNION ALL.

Subqueries

A subquery, also known as a nested query, is a query block that is completely embedded in a predicate. A subquery may appear within the search condition which is a part of the WHERE or HAVING clause of a SELECT, INSERT, UPDATE or DELETE statement. It is like any other query expression, except that it cannot contain a UNION operator. A subquery may be used only in the following types of predicates:

- Comparison predicate
- EXISTS predicate
- IN predicate
- Quantified predicate

Subqueries can be used to arrive at a single value that lets you determine the selection criteria for the outer query block. In the following simple example, the subquery (in parentheses) is evaluated to determine a single value used in selecting the rows for the outer query:

```
SELECT *
  FROM PurchDB.SupplyPrice
 WHERE PartNumber = (SELECT PartNumber
                       FROM PurchDB.Parts
                      WHERE PartName = 'Cache Memory Unit')
```

Subqueries are most frequently found within special predicates, which are described fully in the next section. Additional examples of subqueries can be found there.

Special Predicates

The three types of special predicate are listed here:

- The quantified predicate (ALL, ANY, or SOME), used to compare the value of an expression with some or all of the values of an operand.
- The IN predicate, used to check for inclusion of an expression in a set of values.
- The EXISTS predicate, used to check for the existence of a value in an operand.

With all these types, subqueries may be used; for ALL, ANY, SOME, and IN predicate, additional forms allow the use of a value list in place of a subquery. For each type of special predicate the examples in the next sections show both subquery and non-subquery forms of the predicate whenever both possibilities exist.

Quantified Predicate

A quantified predicate compares a value with a number of other values that are either contained in a value list or derived from a subquery. The quantified predicate has the following general form:

Expression ComparisonOperator Quantifier {*ValueListSubQuery*}

The comparison operators shown here are allowable:

```
=   <>    <    >    <=    >=
```

The quantifier is one of these three keywords:

```
ALL    ANY     SOME
```

The value list is of this form:

(*Val1*, *Val2*, ..., *Valn*)

Using the ANY or SOME Quantifier with a Value List

With the ANY or SOME quantifier (ANY and SOME are synonymous), the predicate is true if *any* of the values in the value list or subquery relate to the expression as indicated by the comparison operator.

Suppose you have a list of the part numbers for parts you have been buying from vendor 9011. You would like to start obtaining those parts from other vendors. The following example shows how you would find the part number and vendor number for all parts supplied by vendor 9011 that are also supplied by some other vendor:

```
SELECT PartNumber, VendorNumber
   FROM PurchDB.SupplyPrice
  WHERE PartNumber = ANY
        ('1343-D-01', '1623-TD-01', '1723-AD-01', '1733-AD-01')
AND NOT VendorNumber = 9011

----------------+------------
PARTNUMBER      |VENDORNUMBER
----------------+------------
1343-D-01       |        9001
1623-TD-01      |        9015
1723-AD-01      |        9004
1723-AD-01      |        9012
1723-AD-01      |        9015
1733-AD-01      |        9004
1733-AD-01      |        9012
```

The quantifier ANY is used to determine whether PurchDB.SupplyPrice contains any of the part numbers in the value list. If so, the query returns the part number and vendor number of vendors supplying that part. The final predicate eliminates all instances where the part is supplied by vendor 9011. Note that SOME could be used in place of ANY, because SOME and ANY are synonyms.

Using ANY or SOME with a Subquery

You can also use the subquery form of the quantified predicate. If you wanted to distribute some of the business you have been giving vendor 9004, you might want to find vendor numbers for each vendor supplying at least one part supplied by vendor 9004. The following query returns this information:

```
SELECT DISTINCT VendorNumber
           FROM PurchDB.SupplyPrice
          WHERE PartNumber = ANY (SELECT PartNumber
                                    FROM PurchDB.SupplyPrice
                                   WHERE VendorNumber = 9004)

------------
VENDORNUMBER
------------
        9004
        9007
        9008
        9009
        9011
        9012
        9015
```

The subquery obtains the part numbers for all parts supplied by vendor 9004. The quantifier ANY is then used to determine if PartNumber is the same as any of these parts. If so, the vendor number supplying that part is returned in the query result.

Some queries may require you to use ANY and SOME constructs in a manner that is not intuitive. Consider the following query:

```
SELECT T1.SalesPrice
  FROM T1
 WHERE T1.PartNumber <> ANY (SELECT T2.PartNumber
                               FROM T2)
```

The inexperienced SQL user might think that this means, “Select the sales price of parts from table T1 whose numbers are *not equal to any* part numbers in table T2.” However, the actual meaning is, “Select the sales price of parts from T1 such that the part number from T1 is *not equal to at least one* part number in T2.” This query returns the sales price of all the parts in T1 if T2 has more than one part.

A less ambiguous form using EXISTS is as follows:

```
SELECT T1.SalesPrice
  FROM T1
 WHERE EXISTS  (SELECT T2.PartNumber
                  FROM T2
                 WHERE T2.PartNumber <> T1.PartNumber)
```

Using the ALL Quantifier

With the ALL quantifier, the predicate is true only if *all* of the values in the value list or subquery relate to the expression as indicated by the comparison operator.

Assume you have been buying parts from vendor 9010. To get a discount from this vendor, you have been required to purchase parts in larger quantities than you would like. To

avoid large stockpiles of these parts, you want to find vendors whose discount is not dependent on the purchase of such large quantities. The following query uses two subqueries and an ALL quantifier to retrieve the information you want:

```
SELECT VendorNumber, PartNumber, DiscountQty
  FROM PurchDB.SupplyPrice
 WHERE DiscountQty < ALL (SELECT DiscountQty
                            FROM PurchDB.SupplyPrice
                           WHERE VendorNumber = 9010)
   AND PartNumber IN (SELECT PartNumber
                        FROM PurchDB.SupplyPrice
                       WHERE VendorNumber = 9010)

------------+----------------+-----------
VENDORNUMBER|PARTNUMBER      |DISCOUNTQTY
------------+----------------+-----------
        9006|1423-M-01       |          1
        9007|1433-M-01       |         15
```

The first subquery obtains the number of parts needed to qualify for a discount for each part supplied by vendor 9010. Using the quantifier ALL, rows are selected only when the quantity needed for a discount is less than that needed for any part supplied by 9010. The second subquery limits the selection to only those part numbers supplied by vendor 9010. Thus, the query result shows every part supplied by vendor 9010 which can be obtained from another vendor in smaller quantities with a discount.

IN Predicate

An IN predicate compares a value with a list of values or a number of values derived by the use of a subquery. The IN predicate has the following general form:

Expression [NOT] IN {*ValueList SubQuery*}

The ValueList and SubQuery forms of the IN predicate are described separately in the following sections.

Note that IN is the same as = ANY.

Using the IN Predicate with a Value List

If you wanted to obtain the numbers of all vendors who supplied a given list of parts, the following query could be used:

```
SELECT DISTINCT VendorNumber
           FROM PurchDB.SupplyPrice
          WHERE PartNumber
             IN ('1143-P-01', '1323-D-01', '1333-D-01', '1723-AD-01',
                 '1733-AD-01')
------------
VENDORNUMBER
------------
        9004
        9007
        9008
        9009
        .
        .
        .
```

Using the IN Predicate with a Subquery

If you wanted a list of all the vendors who supply the same parts that vendor 9004 supplies, the following query could be used:

```
SELECT DISTINCT VendorNumber
           FROM PurchDB.SupplyPrice
          WHERE PartNumber IN (SELECT PartNumber
                                 FROM PurchDB.SupplyPrice
                                WHERE VendorNumber = 9004)
------------
VENDORNUMBER
------------
        9004
        9007
        9008
        9009
        .
        .
        .
```

The subquery determines the part number of every part supplied by vendor 9004. The outer query selects every vendor who supplies one or more of those parts. DISTINCT removes duplicates from the final query result, as many vendors supply more than one such part.

EXISTS Predicate

The EXISTS predicate, also known as the existential predicate, tests for the existence of a row satisfying some condition. It has the following general format:

```
EXISTS  Subquery
```

EXISTS is true only if the query result of the subquery is not empty; that is, a row or rows are returned as a result of the subquery. If the query result is empty, the EXISTS predicate

is false.

In the following example, suppose you need to determine the names of all vendors who currently supply parts:

```
SELECT v.VendorName
  FROM PurchDB.Vendors v
 WHERE EXISTS (SELECT *
                 FROM PurchDB.SupplyPrice sp
                WHERE sp.VendorNumber = v.VendorNumber)

------------------------------
VENDORNAME
------------------------------
Remington Disk Drives
Dove Computers
Space Management Systems
Coupled Systems
Underwood Inc.
Pro-Litho Inc.
Eve Computers
Jujitsu Microelectronics
Latin Technology
KellyCo Inc.
Morgan Electronics
Seminational Co.
Seaside Microelectronics
Educated Boards Inc.
Proulx Systems Inc.
```

In this example, *v* and *sp* are correlation names, which enable ALLBASE/SQL to distinguish the two VendorNumber columns in the predicate without requiring you to repeat each table name in full.

You can also use the NOT EXISTS form of the existential predicate. If you wanted to find those vendors who are not currently supplying you with parts you could use a query of the form shown here:

```
SELECT v.VendorName
           FROM PurchDB.Vendors v
WHERE NOT EXISTS (SELECT *
                    FROM PurchDB.SupplyPrice sp
                   WHERE sp.VendorNumber = v.VendorNumber)

------------------------------
VENDORNAME
------------------------------
Covered Cable Co.
SemiTech Systems
Chocolate Chips
```

Correlated Versus Noncorrelated Subqueries

In many cases, it is possible to execute the subquery just once, and obtain a result which is passed to the outer query for its use. Here is an example:

```
SELECT *
  FROM PurchDB.SupplyPrice
WHERE PartNumber = (SELECT PartNumber
                      FROM PurchDB.Parts
                     WHERE PartName = 'Cache Memory Unit')
```

This kind of subquery is a **noncorrelated subquery**.

In other cases, however, it is necessary to evaluate a subquery once for every row in the outer query, as in the following:

```
SELECT v.VendorName
            FROM PurchDB.Vendors v
WHERE NOT EXISTS (SELECT *
                    FROM PurchDB.SupplyPrice sp
                   WHERE sp.VendorNumber = v.VendorNumber)
```

The predicate in the subquery references the column value v.VendorNumber, which is defined by the outer query block. When this type of relationship exists between a column value in the subquery and a column value in an outer query block, the query is called a **correlated subquery**.

Recognizing correlated subqueries is important when performance is a priority. Correlated subqueries require the optimizer to use an outer loop join algorithm rather than a sort-merge join. Because a sort-merge join is orders of magnitude faster than an outer loop join, correlated subqueries pay a performance penalty. In addition, when the `ANY, SOME, ALL`, or `IN` predicate makes use of subqueries, the queries are converted into correlated subqueries using the `EXISTS` predicate. Therefore, if at all possible, queries using `ANY, SOME, ALL, IN`, or the correlated form of the `EXISTS` predicate should be done as joins of two or more tables rather than by using subqueries if performance is an issue. In fact, it is possible to state a query as a join as well as in a form using subqueries; non-correlated subqueries are faster than sort-merge joins. Sort-merge joins are faster than correlated subqueries which use an outer loop join.

Outer Joins

An inner join returns only tuples for which matching values are found between the common columns in the joined tables. A natural inner join specifies that each pair of common columns is coalesced into a single column in the query result. The term join has become synonymous with the term natural inner join because that type of join is used so frequently.

To include in the query result those tuples from one table for which there is no match in the common columns of the other table you use an **outer join**. The term natural, when applied to an outer join, has the same meaning as with an inner join. Common columns are coalesced into a single column in the query result. No duplicate columns are returned.

Outer Joins Using Explicit JOIN syntax

Outer joins may be constructed using the explicit JOIN syntax of the SELECT statement (see the "SELECT" section of the "SQL Statements" chapter). In a two table outer join, the first table listed in the FROM clause of the SELECT statement is considered the left hand table and the second is considered the right hand table.

The set of rows in the result may be viewed as the union of the set of rows returned by an inner join (the inner part of the join) and the set of rows from one table for which no match is found in the corresponding table (the outer part of the join).

If the unmatched rows from both tables being joined are preserved, the join is a **symmetric outer join**. If the rows are preserved from only the left hand table, the join is a **left asymmetric outer join**. (The word asymmetric is usually omitted.) If the rows are preserved from only the right hand table, the join is a **right outer join**. The current syntax will allow you to specify either a left outer join or a right outer join, but not a symmetric outer join. A technique for creating a symmetric outer join using the UNION operator is described later in the section, "Symmetric Outer Joins Using the UNION Operator."

A left outer join obtains the rows from both tables for which there is a matching value in the common column or columns (the inner part) and the rows from the left hand table for which there is no match in the right hand table (the outer part). Each unmatched row from the left hand table is extended with the columns coming from the right hand table. Each column in that extension has a null value.

A right outer join obtains the rows from both tables for which there is a matching value in the common column or columns, and the rows from the right hand table for which there is no match in the left hand table. The unmatched rows from the right hand table are extended with the columns coming from the left hand table, with null column values returned in that extension for every result row which has no match in the left hand table.

For example, the following right outer join is between the SupplyPrice and the Vendors tables. For all vendors who supply parts, it returns the Part Number, Vendor Name and Vendor City. For all vendors who do not supply parts, it returns just the Vendor Name and Vendor City.

```
SELECT PartNumber, VendorName, VendorCity
      FROM Purchdb.SupplyPrice sp
RIGHT JOIN PurchdB.Vendors v
        ON sp.VendorNumber = v.VendorNumber
  ORDER BY PartNumber DESC

SELECT PartNumber, VendorName, VendorCity FROM Purchdb.SupplyPrice sp RIGHT...
---------------+---------------------------+-------------------------
PARTNUMBER     |VENDORNAME                 |VENDORCITY
---------------+---------------------------+-------------------------
               |Chocolate Chips            |Lac du Choc <--Unmatched
               |SemiTech Systems           |San Jose    <--rows from
               |Kinki Cable Co.            |Bakersfield <--Vendors table
1943-FD-01     |Eve Computers              |Snake River
1933-FD-01     |Remington DiskDrives       |Concord
1933-FD-01     |Educated Boards Inc.       |Phoenix
1933-FD-01     |Latin Technology           |San Jose
1933-FD-01     |Space Management Systems   |Santa Clara
1933-FD-01     |Eve Computers              |Snake River
1923-PA-01     |Jujitsu Microelectronics   |Bethesda
```

```
  .
  .
  .
-----------------------------------------------------------------
   Number of rows selected is 16
   U[p], d[own], l[eft], r[ight], t[op], b[ottom], pr[int] <n>, or e[nd] > e
```

When you use the ON clause of the JOIN syntax, it must contain, at a minimum, the predicate which specifies the join condition. Other predicates may be placed within the SELECT statement, but their location is critical as the following examples show.

Additional predicates may be placed in the ON clause. These predicates limit the rows participating in the inner join associated with the ON clause. All rows excluded by such predicates participate in the outer part of the associated join. The following query returns (in the inner part of the join) Part Numbers for all vendors who supply parts and are located in California (*italics*). It also returns, without the Part Number (in the outer part of the join) all vendors who do not supply parts (**BOLD**), and all vendors who do supply parts, but are not located in California.

```
   SELECT PartNumber, VendorName, VendorCity
         FROM Purchdb.SupplyPrice sp
   RIGHT JOIN PurchdB.Vendors v
           ON sp.VendorNumber = v.VendorNumber
          AND VendorState = 'CA'
     ORDER BY PartNumber DESC

  SELECT PartNumber, VendorName, VendorCity FROM Purchdb.SupplyPrice sp RIGHT...
  --------------+------------------------------+--------------------
  PARTNUMBER    |VENDORNAME                    |VENDORCITY
  --------------+------------------------------+--------------------
                |Underwood Inc.                |Atlantic City
                |Remington Disk Drives         |Concord
                |Coupled Systems               |Puget Sound
                |Kinki Cable Co.               |Bakersfield
                |Jujitsu Microelectronics      |Bethesda
                |Dove Computers                |Littleton
                |SemiTech Systems              |San Jose
                |KellyCo Inc.                  |Crabtree
                |Educated Boards Inc.          |Phoenix
                |Chocolate Chips               |Lac du Choc
                |Morgan Electronics            |Braintree
                |Eve Computers                 |Snake River
  1933-FD-01    |Latin Technology              |San Jose
  1933-FD-01    |Space Management Systems      |Santa Clara
-----------------------------------------------------------------
  First 16 rows have been selected.
  U[p], d[own], l[eft], r[ight], t[op], b[ottom], pr[int] <n>, or e[nd] > e
```

In the above example, the rows participating in the inner join are further restricted by adding to the ON clause, AND VendorState = 'CA'. All vendors that are not in California are placed in the outer part of the join.

If you move the limiting predicate from the ON clause to the WHERE clause, the query returns a different result. In the following query, the inner part of the join still contains all vendors who supply parts and are located in California. However, in the outer part of the join, only those vendors who do not supply parts and are in California are included.

```
   SELECT PartNumber, VendorName, VendorCity
         FROM Purchdb.SupplyPrice sp
   RIGHT JOIN PurchdB.Vendors v
           ON sp.VendorNumber = v.VendorNumber
        WHERE VendorState = 'CA'
     ORDER BY PartNumber DESC
```

```
   SELECT PartNumber, VendorName, VendorCity FROM Purchdb.SupplyPrice sp RIGHT...
---------------+-----------------------------+-------------------
   PARTNUMBER      |VENDORNAME                   |VENDORCITY
---------------+-----------------------------+-------------------
                  |SemiTech Systems             |San Jose
                  |Kinki Cable Co.              |Bakersfield
   1933-FD-01     |Latin Technology             |San Jose
   1933-FD-01     |Space Management Systems     |Santa Clara
   .
   .
   .
------------------------------------------------------------------------
   First 16 rows have been selected.
   U[p], d[own], l[eft], r[ight], t[op], b[ottom], pr[int] <n>, or e[nd] > e
```

In the above example, the WHERE clause is applied to all the rows returned, regardless of whether they are in the inner or outer part of the join. Thus no rows are returned unless the vendor is located in California.

If you want the inner part of the query to contain all vendors who do supply parts and are located in California while the outer part contains all vendors who do not supply parts, regardless of location, use the query shown below.

```
   SELECT PartNumber, VendorName, VendorCity
         FROM Purchdb.SupplyPrice sp
   RIGHT JOIN PurchdB.Vendors v
           ON sp.VendorNumber = v.VendorNumber
        WHERE VendorState = 'CA'
           OR VendorState <> 'CA' AND PartNumber IS NULL
     ORDER BY PartNumber DESC
```

```
  SELECT PartNumber, VendorName, VendorCity FROM Purchdb.SupplyPrice sp
RIGHT...
  ----------------+-----------------------------+-------------------
   PARTNUMBER     |VENDORNAME                    |VENDORCITY
  ----------------+-----------------------------+-------------------
                  |SemiTech Systems              |San Jose
                  |Chocolate Chips               |Lac du Choc
                  |Kinki Cable Co.               |Bakersfield
   1933-FD-01     |Latin Technology              |San Jose
   1933-FD-01     |Space Management Systems      |Santa Clara
   1923-PA-01     |Seminational Co.              |City of Industry
   1833-PT-01     |Seminational Co.              |City of Industry
   1833-PT-01     |Seaside Microelectronics      |Oceanside
   1823-PT-01     |Seaside Microelectronics      |Oceanside
   .
   .
   .
------------------------------------------------------------------------
   First 16 rows have been selected.
   U[p], d[own], l[eft], r[ight], t[op], b[ottom], pr[int] <n>, or e[nd] > e
```

If all common columns between the tables being joined are to be used for the join, the

keyword NATURAL may be used so long as the specification of the ON clause join predicate is omitted. This technique may be used when joining more than two tables, as in the query shown below:

```
SELECT PartName, DeliveryDays, VendorName
         FROM PurchDB.Parts
NATURAL RIGHT JOIN PurchDB.SupplyPrice
NATURAL RIGHT JOIN PurchDB.Vendors
        ORDER BY PartName DESC
```

```
SELECT PartName, DeliveryDays, VendorName FROM PurchDB.Parts NATURAL RIGHT...
-----------------------------+------------+-------------------------
PARTNAME                      |DELIVERYDAYS|VENDORNAME
-----------------------------+------------+-------------------------
                             |            |SemiTech Systems
                             |            |Kinki Cable Co.
                             |            |Chocolate Chips
Winchester Drive             |          20|Remington Disk Drives
Winchester Drive             |          30|Morgan Electronics
Video Processor              |          20|Latin Technology
Video Processor              |          30|Jujitsu Microelectronics
Video Processor              |          15|Eve Computers
.
.
.
-------------------------------------------------------------------------
First 16 rows have been selected.
U[p], d[own], l[eft], r[ight], t[op], b[ottom], pr[int] <n>, or e[nd] > e
```

Outer Joins Using the UNION Operator

An outer join can also be created by using the UNION operator.

Suppose you want to create a list of vendors who either supply some part with a unit price less than $100 or else do not supply any parts at all. To do this, merge two separate queries with a UNION ALL statement, as in the following examples.

The first query shown here selects the names of vendors who do *not* supply parts:

```
SELECT v.VendorName
  FROM PurchDB.Vendors v
 WHERE NOT EXISTS (SELECT *
                     FROM PurchDB.SupplyPrice sp
                    WHERE sp.VendorNumber = v.VendorNumber)
```

Notice that a second query block is embedded within the first query expression. It creates a temporary table containing the names of all vendors who *do* supply parts. Then note the special predicate EXISTS, which is negated in this case. The outer SELECT statement allows us to identify the name of each vendor in the Vendors table. Each VendorName is compared against the list of vendors who *do* supply parts. If the VendorName from the outer SELECT statement is *not* found in the temporary table created by the subquery, the outer VendorName is returned to the query result, providing us a list of all the Vendors who *do not* supply parts.

The second query shown here defines the vendors who supply at least one part with a unit price under $100:

```
SELECT DISTINCT v.VendorName
  FROM PurchDB.Vendors v, PurchDB.SupplyPrice sp
 WHERE v.VendorNumber = sp.VendorNumber
   AND sp.UnitPrice < 100.00
```

The next example shows this query joined to the previous one by the UNION ALL statement. It also shows the use of character constants to indicate which rows result from which query block.

```
SELECT DISTINCT v.VendorName, 'supplies parts under $100'
  FROM PurchDB.Vendors v, PurchDB.SupplyPrice sp
 WHERE v.VendorNumber = sp.VendorNumber
   AND sp.UnitPrice < 100.00

UNION ALL

SELECT v.VendorName, 'none supplied'
  FROM PurchDB.Vendors v
 WHERE NOT EXISTS (SELECT *
                     FROM PurchDB.SupplyPrice sp
                    WHERE sp.VendorNumber = v.VendorNumber)

------------------------------+-------------------------+
VENDORNAME                    |(CONST)                  |
------------------------------+-------------------------+
Dove Computers                |supplies parts under $100|
Educated Boards Inc.          |supplies parts under $100|
Jujitsu Microelectronics      |supplies parts under $100|
Proulx Systems Inc.           |supplies parts under $100|
Seaside Microelectronics      |supplies parts under $100|
Seminational Co.              |supplies parts under $100|
Underwood Inc.                |supplies parts under $100|
Covered Cable Co.             |none supplied            |
SemiTech Systems              |none supplied            |
Chocolate Chips               |none supplied            |
```

Symmetric Outer Join Using the UNION Operator

Since the syntax does not support a symmetric outer join, you might try to simulate a symmetric outer join using the left outer join syntax in combination with the right outer join syntax. Intuitively, the following query might seem correct:

```
SELECT PartName, PartNumber, VendorName, VendorCity
               FROM Purchdb.Parts
NATURAL LEFT  JOIN Purchdb.SupplyPrice
NATURAL RIGHT JOIN Purchdb.Vendors
          ORDER BY PartName, VendorName
```

This three table outer join does a left outer join between the Parts and the SupplyPrice tables. The result of that join is then used as the left hand table in a right outer join with the Vendors table.

It would seem as though the result first displays all parts supplied by a vendor, then all parts for which there is no supplier, followed by all vendors who do not supply parts.

But, the action of the query is subtle. The natural left join preserves the parts from the Parts table that is not supplied by any vendor. This supplies the left hand component for the simulated symmetric outer join. However, although the natural right join preserves the three vendors from the vendors table who do not supply parts (the right hand component for the simulated symmetric outer join), it eliminates the unmatched parts from the Parts table. This happens because the natural right join only preserves unmatched rows from the right hand table, eliminating the row from the Parts table.

NOTE If you test the next query on the sample database, you must first use the following ISQL INSERT statement to add a row with no vendor to the Parts table.

```
INSERT INTO PurchDB.Parts
            (PartNumber, PartName)
     VALUES ('XXXX-D-LO', 'test part');
```

To preserve all the unmatched rows from both sides, thus generating a full symmetric outer join, you must use the following syntax:

```
SELECT PartName, PartNumber, VendorName
               FROM PurchDB.Parts
NATURAL LEFT  JOIN PurchDB.SupplyPrice
NATURAL LEFT  JOIN PurchDB.Vendors
             UNION
            SELECT PartName, PartNumber, VendorName
              FROM PurchDB.Parts
NATURAL RIGHT JOIN PurchDB.SupplyPrice
NATURAL RIGHT JOIN PurchDB.Vendors
             UNION
            SELECT PartName, PartNumber, VendorName
              FROM PurchDB.Parts
NATURAL RIGHT JOIN PurchDB.SupplyPrice
NATURAL LEFT  JOIN PurchDB.Vendors
          ORDER BY PartName DESC, PartNumber;
```

The result from the natural left join...natural left join preserves the unmatched part from Parts. The natural right join...natural right join preserves the unmatched vendors from Vendors. The natural right join...natural left join would preserve all unmatched rows from SupplyPrice if there were any (in this example there are none). The union operation combines the three results, preserving the unmatched rows from all joins. There are three complete sets of rows that satisfy the inner join, but the union operation eliminates the duplicate rows unless UNION ALL is specified.

The result of the above query follows:

```
SELECT PartName, PartNumber, VendorName FROM PurchDB.Parts NATURAL LEFT...
-----------------------------+---------------+--------------------
PARTNAME                       |PARTNUMBER      |VENDORNAME
-----------------------------+---------------+--------------
                               |                |Kinki Cable Co.
                               |                |SemiTech Systems
                               |                |Chocolate Chips
                               |XXXX-D-LO|
Winchester Drive               |1343-D-01       |Remington Disk Drives
Winchester Drive               |1343-D-01       |Morgan Electronics
Video Processor                |1143-P-01       |Eve Computers
Video Processor                |1143-P-01       |Coupled Systems
.
.
.
```

Using GENPLAN to Display the Access Plan

When a statement is executed in ISQL or is preprocessed in an application program, the optimizer attempts to generate the most efficient path to the desired data. Taking into account the available indexes, the operations that must be executed, and the clauses in the predicates that may increase the selectivity of the statement, the optimizer decides what indexes to use and the proper order of the needed operations. The result of this evaluation process is an **access plan** produced by the optimizer.

In most cases, the optimizer chooses the best plan. But, there are times when you may want to display the access plan chosen by the optimizer. You may then evaluate that plan in light of your specific knowledge of the database and decide if the optimizer has generated the optimum access plan for your situation.

If you want to override the access plan chosen by the optimizer, issue the SETOPT statement.

The statements used to generate and display the access plan are the GENPLAN statement and a SELECT on the pseudotable SYSTEM.PLAN.

Generating a Plan

Suppose you want to generate the access plan for the query shown below.

```
isql=> GENPLAN FOR

> SELECT p.PartName, p.PartNumber, v.VendorName,
> s.UnitPrice, i.QtyOnHand
> FROM PurchDb.Parts p, PurchDB.Inventory i,
> PurchDB.SupplyPrice s, PurchDB.Vendors v
> WHERE p.PartNumber = i.PartNumber
> AND s.PartNumber = p.PartNumber
> AND s.VendorNumber = v.VendorNumber
> AND p.PartNumber = '1123-P-01';
```

The access plan will then be placed in the system pseudotable, SYSTEM.PLAN, but will not be displayed until you do a SELECT from SYSTEM.PLAN. You can also generate the access plan for a query that is stored in the database as a stored section. For example:

```
isql=> GENPLAN FOR MODULE SECTION MyModule(10);
```

Displaying a Query Access Plan

To display the access plan generated by the optimizer, showing the columns in the order most useful to you, execute the following statement:

```
isql=> SELECT Operation, TableName, IndexName, QueryBLock, Step, Level
> FROM System.Plan;
```

```
SELECT Operation, TableName, IndexName, QueryBlock, Step, Level FROM System.Plan
-----------------+------------+----------------+-----------+--------+--------
OPERATION        |TABLENAME   |INDEXNAME       |QUERYBLOCK |STEP       |LEVEL
-----------------+------------+----------------+-----------+--------|--------
index scan       |INVENTORY   |INVPARTNUMINDEX |          1|       1|       4
index scan       |PARTS       |PARTNUMINDEX    |          1|       2|       4
merge join       |            |                |          1|       3|       3
serial scan      |SUPPLYPRICE |                |          1|       4|       3
nestedloop join  |            |                |          1|       5|       2
index scan       |VENDORS     |VENDORNUMINDEX  |          1|       6|       2
nestedlopp join  |            |                |          1|       7|       1

-------------------------------------------------------------------------------
Number of rows selected is 7
U[p], d[own], l[eft], r[ight], t[op], b[ottom], pr[int] <n>, or e[nd] >r
```

Interpreting a Display

The information from the columns in SYSTEM.PLAN helps you to understand the access plan generated by the optimizer. The columns are discussed in the order most useful to you.

`OPERATION` shows each operation being executed to obtain the data. Because your greatest concern is usually whether indexes are being used effectively, you should look at this column first. For each index scan operation, indexes are being used to access the data.

If there is no limiting predicate in the WHERE clause of the statement, or if the predicate will cause the selection of a large percentage of the rows from the table, a serial scan will be chosen instead of an index scan.

When a join is specified, you can look at the join chosen to see if it is the most appropriate type of join, considering the specific data in your database.

For more information, see the “Understanding Data Access Paths” section of Chapter 2 , “Using ALLBASE/SQL.”

`TABLENAME` shows the table upon which an operation is being executed. Thus, you can see the tables for which indexes are being used, and the tables which are participating in various joins.

`INDEXNAME` shows which specific index is being used to access data in a particular table. This may be useful if multiple indexes exist for a given table.

`QUERYBLOCK` shows the block in which a given operation is taking place. A simple statement will have only one query block. More complex statements will be broken into additional blocks to simplify processing.

`STEP` shows the order in which operations are executed within a given queryblock. From this information you can determine the order of operations.

`LEVEL` shows the hierarchy of the operations so you can easily graph the operations as an execution tree. This is normally necessary only when your HP Service Representative is evaluating a query.

Updatability of Queries

INSERT, UPDATE and DELETE operations may be performed through views or as qualified by search conditions provided the views or search conditions are based on updatable queries. UPDATE WHERE CURRENT and DELETE WHERE CURRENT operations may be performed through cursors provided the cursors are based on updatable queries.

Queries that underlie views and cursors are called **updatable queries** when they conform to all of the following updatability criteria:

- No DISTINCT, GROUP BY, or HAVING clause is specified in the outermost SELECT statement; and no aggregate is specified in the outermost select list.
- The FROM clause specifies exactly one table, either directly or through a view. If the FROM clause specifies a view, the view must be based on an updatable query.
- For INSERT and UPDATE through views, the select list in the view definition must not contain any arithmetic expressions. It must contain only column names.
- For UPDATE WHERE CURRENT and DELETE WHERE CURRENT operating on cursors, the cursor declaration must not include an ORDER BY clause, and the query expression must not contain subqueries, the UNION or UNION ALL statement, or any nonupdatable views.
- The target table of an INSERT, UPDATE, or DELETE operation is the base table to which the changes are actually being made.
- For noncursor INSERT, UPDATE, or DELETE operations, the view definition must not include any subqueries which contain the target table in their FROM clause; and if a search condition is given, it must not include any subqueries which contain the target table in their FROM clause.

If a query is updatable by the previous rules, then the underlying table is an **updatable** table. Otherwise it is considered a **read-only** table and is locked accordingly. This means that in cursor operations, SIX, IX, and X locks are not used unless the query that underlies the cursor matches the updatability criteria and was declared with columns for UPDATE. In noncursor view operations, SIX, IX, and X locks are not obtained unless the table underlying the view is updatable. Refer to Chapter 5 , “Concurrency Control through Locks and Isolation Levels,” for a complete explanation of SIX, IX, and X locks.

4 Constraints, Procedures, and Rules

In addition to the basic tables and indexes in a DBEnvironment, ALLBASE/SQL lets you create database objects known as constraints, procedures, and rules, which provide for a high degree of data consistency and integrity inside the DBEnvironment without the need for extensive application programming. Constraints define conditions on the rows of a table; procedures define sequences of SQL statements that can be stored in the DBEnvironment and applied as a group either through rules or through execution by specific users; and rules let you define complex relationships among tables by tying specific procedures to particular kinds of data manipulation on tables. Together, these tools let you store many of your organization's business rules in the DBEnvironment itself, reducing the need for application code.

This chapter presents the following topics:

- Using Integrity Constraints
- Using Procedures
- Using Rules

Using Integrity Constraints

Using integrity constraints helps to ensure that a database contains only valid data. Integrity constraints provide a way to check data within the database system rather than by coding elaborate validation checks within application programs. An integrity constraint is either a unique constraint, a referential constraint, or a check constraint. All of these constraints are described in this section.

When a table is created, integrity constraints can be defined at the column level or at the table level. A constraint can be placed on an individual column (at the column or table level) or on a combination of columns (at the table level).

Unique Constraints

A unique constraint requires that no two rows in a table contain the same value in a given column or list of columns. You can create a unique constraint at either the table level or the column level. Unique constraints can be defined as either UNIQUE or PRIMARY KEY. The two types of unique constraints differ in that if a PRIMARY KEY is placed on a column or column list, the column name(s) can be omitted from the referential constraint syntax in the definition of the referencing table. A given column upon which a unique or primary constraint has been defined need not be referenced by a referential constraint; but a referential constraint can only refer to a column upon which a unique or primary key constraint has been defined. Referential constraints are discussed below.

Additionally, PRIMARY KEY can be specified only once per table. Duplicate unique

constraints are not allowed. Neither UNIQUE nor PRIMARY KEY columns can contain null values--they must be defined as NOT NULL.

The following syntax is used to define a unique constraint on an individual column or column list at the table level:

{UNIQUE PRIMARY KEY} (*ColumnName* [,...]) [CONSTRAINT *ConstraintID*]

ConstraintID is the name of the constraint. It is not necessary to name the constraint. If it is not named, ALLBASE/SQL names it SQLCON_*uniqueid*, where *uniqueid* is a unique string. The constraint names are maintained in the system catalog table SYSTEM.CONSTRAINT.

A column list cannot contain a column more than once. In the example below, a constraint is placed on a column at the table level:

```
CREATE PUBLIC TABLE RecDB.Clubs
                    (ClubName CHAR(15) NOT NULL,
                    UNIQUE (ClubName) CONSTRAINT ClubConstrnt)
                  IN RecFS;
```

The syntax for defining a unique constraint at the column level is part of the column definition. NOT NULL and either UNIQUE or PRIMARY KEY are included along with the other column parameters. In the example below, one column is defined with a unique constraint:

```
CREATE PUBLIC TABLE RecDB.Clubs
                   (ClubName CHAR(15) NOT NULL UNIQUE CONSTRAINT ClubConstrnt)
                 IN RecFS;
```

A table defined with a PRIMARY KEY followed by a column list is shown in the section “Examples of Integrity Constraints.”

Referential Constraints

A referential constraint requires that the value in a column or columns of the referencing table, must either be null or match the value of a column or columns of a unique constraint in the referenced table. To establish a referential constraint, a unique or primary key constraint must first be defined on the referenced table's column or column list and then a referential constraint must be defined on the referencing table's column or column list.

The Referenced Table

The referenced table must contain a unique constraint created with either a UNIQUE or PRIMARY KEY clause on a column or column list:

```
CREATE PUBLIC TABLE RecDB.Clubs
                    (ClubName CHAR(15) NOT NULL
                    PRIMARY KEY CONSTRAINT Clubs_PK, -- column level constraint
                    ClubPhone SMALLINT,
                    Activity CHAR(18))
                IN RecFS;
```

The referenced table must be created before the referencing table unless the referenced and referencing tables are created within a CREATE SCHEMA statement or if both the tables are created in the same transaction, the SET REFERENTIAL CONSTRAINTS DEFERRED statement has been executed and is still in effect.

The Referencing Table

A referential constraint is placed on columns which are dependent on other columns (in the referenced table). You can create a referential constraint at either the table level or the column level. Referencing columns need not be NOT NULL.

The following syntax is used to define a referential constraint at the table level in the CREATE TABLE statement for a referencing table:

FOREIGN KEY(*FKColumnName* [,...])

REFERENCES*RefTableName* [(*RefColumnName* [,...])] [CONSTRAINT *ConstraintID*]

FOREIGN KEY identifies a referencing column or column list. REFERENCES identifies the referenced table and referenced column list. The order and number of referencing columns in the FOREIGN KEY clause must be the same as that of the referenced columns in the REFERENCES clause. The referenced table cannot be a view.

The syntax for defining a referential constraint at the column level for a referencing column is shown here:

REFERENCES*RefTableName* [(*RefColumnName*)] [CONSTRAINT *ConstraintID*]

Only one *RefColumnName* is possible.

Note in the following example that the table's column definitions and table level constraints can be in any order within the parentheses and are separated from each other with commas:

```
CREATE PUBLIC TABLE RecDB.Members
                    (MemberName CHAR(20) NOT NULL,     column definition
                    Club CHAR(15) NOT NULL,
                    MemberPhone SMALLINT,
                    FOREIGN KEY (Club)                 table level
                    REFERENCES Clubs (ClubName))       referential constraint
                  IN RecFS;
```

If the REFERENCES clause does not specify a *RefColumnName*, then the table definition referenced must contain a unique constraint that specifies PRIMARY KEY. The primary key column list is the implicit *RefColumnName* list. It must have the appropriate number of columns.

The owner of the table containing referencing columns must have the REFERENCES authority on referenced columns, have OWNER authority on the referenced table, or have DBA authority, for the duration of the referential constraint.

Check Constraints

A check constraint specifies a condition which must be upheld for an insert or update to be successfully performed on a table or view. A table check constraint must not be false for any row of the table on which it is defined. A view check constraint must be true for the condition in the SELECT statement that defines the view.

A table check constraint is defined in the CREATE TABLE or ALTER TABLE statement with the following syntax:

CHECK(*SearchCondition*) [CONSTRAINT *ConstraintID*]

If a check constraint is added to an existing table, data already in the table is verified to ensure that the check constraint is satisfied. A constraint error occurs if the constraint is not satisfied; the ALTER TABLE statement adding the constraint fails.

The check is also performed when the INSERT or UPDATE statement is executed. A DELETE statement never causes a check constraint error.

The check search condition must not contain a subquery, aggregate function, TID function, local variable, procedure parameter, dynamic parameter, current function, USER, or host variable. The search condition expression also cannot contain a LONG column unless it is within a long column function. When adding a new column, the columns specified in the search condition must be defined in the same CREATE TABLE or ALTER TABLE ADD COLUMN statement. For the ALTER TABLE ADD COLUMN statements, the check constraint can only be specified for the column being added. When adding a constraint, columns specified in the check constraint search condition must already exist in the table.

The search condition is a boolean expression which must not be false for a table check constraint to be satisfied. If any value specified in the search condition expression is NULL, the result of the expression may be the boolean unknown value rather than true or false. The check constraint is satisfied if the result is true or unknown.

For example, consider the following check constraint:

```
CHECK (NumParts > 5)
```

If NumParts is 5, the result is false and the check is not satisfied. If NumParts is 10, the result is true and the check constraint for this row is satisfied. If NumParts is NULL, the result is unknown and the check constraint is also satisfied for this row.

A table check constraint can be defined at a column level or a table level. A check constraint defined on a column is specified before the comma that ends the column definition as shown below. A table constraint can be placed anywhere-- before, after, or among the column descriptions. These rules apply for columns defined with either the CREATE TABLE or ALTER TABLE statements.

For example, a column level check constraint on the Date column is defined as follows:

```
CREATE PUBLIC TABLE RecDB.Events
                    (SponsorClub CHAR(15),
                    Event CHAR(30),
                    Date DATE DEFAULT CURRENT_DATE     No comma here

                    (CHECK (Date >= '1990-01-01'),
                    Constraint Check_No_Old_Events),

                    Time TIME,
                    Coordinator CHAR(20),
                    FOREIGN KEY (Coordinator, SponsorClub)
                    REFERENCES RecDB.Members (MemberName, Club)
                    CONSTRAINT Events_FK)
                  IN RecFS;
```

However, the same constraint defined at the table level is defined as follows:

```
CREATE PUBLIC TABLE RecDB.Events
                    CHECK (Date >= '1990-01-01')      Check Constraint
                    CONSTRAINT Check_No_Old_Events
                    (SponsorClub CHAR(15),
                    Event CHAR(30),
                    Date DATE DEFAULT CURRENT_DATE,
                    Time TIME,
                    Coordinator CHAR(20),
                    FOREIGN KEY (Coordinator, SponsorClub)
                    REFERENCES RecDB.Members (MemberName, Club)
                    CONSTRAINT Events_FK)
                   IN RecFS;
```

This table level constraint could also be defined *after* the Date or Time column, or at any point in the parenthesized list. There is one difference between table and column level check constraints: a column level check constraint must reference only the column on which it is defined.

A check constraint that references more than one column *must* be defined at the table level. For example, the constraint CHECK (Date >= '1990-01-01' AND Time > '00:00.000') must be defined at the table level because both the Date and Time columns are specified in the check constraint.

A view check constraint is defined with the CREATE VIEW statement using the following syntax at the end of the view definition:

```
WITH CHECK OPTION  [CONSTRAINT ConstraintID]
```

The conditions of the SELECT statement defining the view become the view check constraint search conditions when the WITH CHECK OPTION clause is specified. A view can have only one WITH CHECK OPTION. This check constraint checks all of the conditions which are included in the SELECT statement. These SELECT statement conditions serve two purposes. First, they originally define the view. They also define the conditions of the check constraint that is applied when the underlying base table is modified through the view. When a table is modified through a view, the view check constraint is checked along with any table constraints. The view check constraint must be *true* (not *unknown*) to ensure that all changes made through a view can still be displayed. All underlying views are also checked, whether or not they are defined with check options. Unique and referential constraints cannot be defined on views.

See Chapter 10 , “SQL Statements A - D,” for the check constraint syntax, within the syntax of CREATE TABLE, ALTER TABLE, or CREATE VIEW statements.

Examples of Integrity Constraints

The schema example in this section shows the constraints among three tables: Clubs, Members, and Events. The tables are created as PUBLIC so as to be accessible to any user or program that can start a DBE session.

Constraints are placed on the tables to ensure that:

1. Events are coordinated by club members who are listed in the Members table

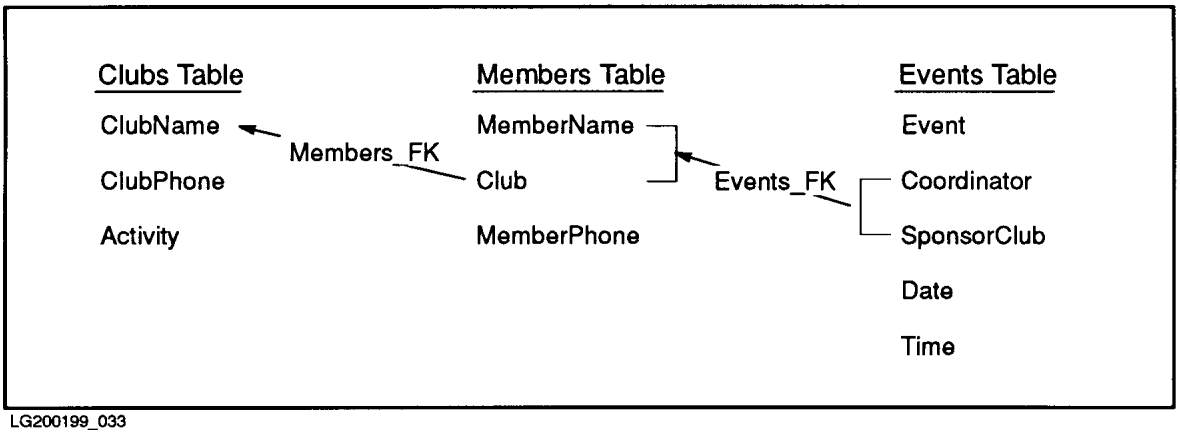
2. Clubs sponsoring the events are listed in the Clubs table
3. Events cannot be scheduled earlier than the current date.

```
CREATE PUBLIC TABLE RecDB.Clubs
                    (ClubName CHAR(15) NOT NULL
                    PRIMARY KEY CONSTRAINT Clubs_PK,
                    ClubPhone SMALLINT,
                    Activity CHAR(18))
                  IN RecFS;

CREATE PUBLIC TABLE RecDB.Members
                    (MemberName CHAR(20) NOT NULL,
                    Club CHAR(15) NOT NULL,
                    MemberPhone SMALLINT,
                    PRIMARY KEY (MemberName, Club) CONSTRAINT Members_PK,
                    FOREIGN KEY (Club) REFERENCES RecDB.Clubs
                    CONSTRAINT Members_FK)
                  IN RecFS;

CREATE PUBLIC TABLE RecDB.Events
                    (Event CHAR(30),
                    Coordinator CHAR(20),
                    SponsorClub CHAR(15),
                    Date DATE DEFAULT CURRENT_DATE,
                    CHECK (Date >= '1990-01-01'),
                    Time TIME,
                    FOREIGN KEY (Coordinator, SponsorClub)
                    REFERENCES RecDB.Members
                    CONSTRAINT Events_FK)
                  IN RecFS;
```

Note that updating the Members table before the Clubs table could cause a referential constraint error when error checking is at statement level. The RecDB.Members.Club column references the RecDB.Clubs.ClubName column which is not yet updated. However, if you deferred referential checking to the end of the transaction, no error would occur. A value could then be inserted into the RecDB.Clubs.ClubName column that would resolve the reference. When a `COMMIT WORK` statement is executed, no constraint errors will exist.

The illustration in Figure 4-1. shows the referential constraints based on this sample schema. The arrows point to the columns with unique constraints.

Figure 4-1. Referential Constraints in a Set of Tables



The Events table contains information about events. The combination of values in the Coordinator and SponsorClub columns of the Events table must be either be null or match the combination of values in the MemberName and Club columns of the Members table, as shown by the Events_FK constraint.

The Members table contains the names of members and clubs. A member can be in more than one club. For every Coordinator/SponsorClub pair of values exists a corresponding MemberName/Club match.

The Clubs table contains information about clubs. For every club entry in the Members table, a corresponding entry must exist in the Clubs table, as shown by the Members_FK constraint.

Inserting Rows in Tables Having Constraints

There are two ways you can insert data in tables having constraints. You can insert values in referenced columns before inserting values in referencing columns, or you can defer constraint error checking in a transaction until all constraints referring to each other have been resolved.

With the first method, using the tables defined in the previous example, the Clubs data should be loaded first, then the Members data, because the MemberName column is dependent on the ClubName column. The Events table should be loaded last as the Coordinator and SponsorClub columns are dependent on the MemberName and Club columns of the Members Table.

If the Clubs, Members, and Events tables were empty and you attempted to insert the values in the order shown below, you would receive the following corresponding results:

Order	Table	Values	Result
1	Members	'John Ewing', 'Energetics', 6925	Violates Members_FK because 'Energetics' club does not exist in the ClubName column of the Clubs table
2	Members	'John Ewing', NULL, 6925	Violates NOT NULL on Members_PK columns
3	Clubs	'Energetics', 1111, 'aerobics'	Valid
4	Clubs	'Windjammers', 2222, 'sailing'	Valid
5	Clubs	'Energetics', 3333,' lo-impact'	Violates Clubs_PK because 'Energetics' is already in the ClubName column of the Clubs table (entries must be unique in a primary key column)
6	Members	'John Ewing', 'Energetics', 6925	Valid
7	Events	'Energetics', 'advanced stretching', '1986-12-04', '15:30:00', 'Martha Mitchell'	Valid

Order	Table	Values	Result
8	Members	'Martha Mitchell', 'Energetics', 1605	Valid
9	Events	'Energetics', 'advanced stretching', '1986-12-04', '15:30:00', 'Martha Mitchell'	Violates check constraint which states that an event's date must be later or the same as January 1, 1990
10	Events	'Energetics', 'advanced stretching', '1990-01-01','15:30:00','Martha Mitchell'	Valid

Values cannot be inserted into Members or Events without the references being satisfied. To insert rows, either NULLs must be inserted and then the tuples updated after the referenced rows are inserted, or the referenced rows must be inserted first. Note that a NULL cannot be inserted into the Members_FK column Club because that column also participates in Members_PK -- and therefore was declared NOT NULL.

With the second method, you can also perform these inserts in one transaction, deferring constraint checking to the end of the transaction. While you are inserting data, constraint error violations are not reported because they will be resolved by the time the `COMMIT WORK` statement is executed. Use a `SET CONSTRAINTS` statement after a `BEGIN WORK` statement to defer constraint checking, as follows:

```
BEGIN WORK

SET REFERENTIAL CONSTRAINTS DEFERRED
```

Modify all tables that refer to each other.

```
COMMIT WORK
```

You can issue the `SET CONSTRAINTS` statement to defer several types of operation at one time. Refer to Chapter 12 , “SQL Statements S - Z,” for the syntax of the `SET CONSTRAINTS` statement.

How Constraints are Enforced

Constraints are controlled and checked by ALLBASE/SQL once they are defined. Once a constraint is placed on a column, ALLBASE/SQL performs the necessary checks each time a value is inserted, altered, or deleted. By default, integrity constraints are enforced on a statement level basis. That is, if an integrity constraint is not satisfied after the execution of an `INSERT, UPDATE`, or `DELETE` statement, then the statement has no effect on the database and an error message is generated.

You can, however, use the `SET CONSTRAINTS DEFERRED` statement to defer constraint enforcement until either the end of a transaction or a `SET CONSTRAINTS IMMEDIATE` statement is encountered. Deferred constraint enforcement avoids concern over the *order* of inserting or updating when a foreign key and primary key exist in the same table or different tables. The table can be modified without constraint violations being reported until either the end of a transaction or `SET CONSTRAINTS IMMEDIATE` statement is encountered. While a constraint check is deferred, you are responsible for ensuring that data placed in the database is free of constraint errors.

In addition, you can temporarily use the SET DML ATOMICITY statement to set the DML error checking level to row level. However, you must handle partially processed statements yourself, as statements that get errors will not undo their partial execution.

Constraint error checking is part of general error checking but you can override the checking level by setting constraint checking to deferred. However, when you set constraint checking back to IMMEDIATE, the level of constraint checking returns to the *current* level specified by the most recent SET DML ATOMICITY statement.

Refer to Chapter 12 , "SQL Statements S - Z," for detailed information on the SET DML ATOMICITY and SET CONSTRAINTS statements.

Using Procedures

An ALLBASE/SQL procedure consists of control flow and status statements together with SQL statements that are stored as sections in the system catalog for later execution at the user's request or through the firing of a rule. You can create a procedure through ISQL or through an application program; and you can execute the procedure through ISQL, through an application program, or through rules that are created separately. For more information about rules, refer to the section "Using Rules," later in this chapter.

Procedures offer the following features:

- They reduce communication between applications and the DBEnvironment, thereby improving performance.
- They provide additional security by controlling exactly which operations users can perform on database objects.
- Along with rules, they enable you to store business rules in the database itself rather than coding them in application programs.
- They let you protect application programs from changes in the database schema.

Often, procedures are built to accommodate a set of rules defined on particular tables. Although you can use procedures without rules, rules always operate in conjunction with procedures. When you create a rule, the referenced procedure must already exist. So you must create procedures first, then rules.

The following sections describe the use of procedures:

- Understanding Procedures
- Creating Procedures
- Executing Procedures
- Procedures and Transaction Management
- Using SQL Statements in Procedures
- Queries inside Procedures
- Using a Procedure Cursor in ISQL

- Error Handling in Procedures
- Using RAISE ERROR in Procedures
- Recommended Coding Practices for Procedures

Understanding Procedures

Procedures (defined either in ISQL or through applications) can include many of the operations available inside application programs. Within a procedure, you can use local variables, issue most SQL statements, create looping and control structures, test error conditions, print messages, and return data or status information to the caller. You can pass data to and from a procedure through parameters. You create a procedure with the CREATE PROCEDURE statement and execute it using an EXECUTE PROCEDURE statement. When it is no longer needed, you remove a procedure from the DBEnvironment with the DROP PROCEDURE statement. You cannot execute a procedure from within another procedure; however, a procedure can contain a statement that fires a rule that executes another procedure. This is called chaining of rules. Refer to “Using Rules,” below.

To create a procedure, you must have RESOURCE or DBA authority. In order to invoke a procedure, you need EXECUTE or OWNER authority for the procedure or DBA authority. If the procedure is invoked through a rule, the rule owner needs EXECUTE or OWNER authority for the procedure or DBA authority.

Creating Procedures

The following is a very simple example of procedure creation:

```
CREATE PROCEDURE ManufDB.FailureList
   (Operator CHAR(20) NOT NULL,
    FailureTime DATETIME NOT NULL,
    BatchStamp DATETIME NOT NULL) AS
      BEGIN
          INSERT INTO ManufDB.TestMonitor
             VALUES (:Operator, :FailureTime,
             :BatchStamp);
      END;
```

This example shows the definition of a procedure named FailureList owned by user ManufDB. This procedure enters a row into the ManufDB.TestMonitor table when a failure occurs during testing.

Three input parameters are declared with names and data types assigned--Operator, FailureTime, and BatchStamp. At run time, these parameters accept actual values into the procedure from the caller. The procedure body starts with the BEGIN keyword and concludes with the END keyword. The procedure body consists of a single INSERT statement that uses the parameters just as you would use host variables in an embedded SQL program. The effect of a call to the procedure is to create a new row in a table named ManufDB.TestMonitor containing a record of the current date and time along with the name of the operator, and the batch stamp (unique identifier) of the batch of parts that failed during testing.

Executing Procedures

You execute the procedure using an EXECUTE PROCEDURE statement. The EXECUTE PROCEDURE statement can be issued directly in ISQL or in an application program, or the EXECUTE PROCEDURE clause can appear inside a CREATE RULE statement. The following shows an invocation of a procedure in an ISQL session:

```
isql=> EXECUTE PROCEDURE
> ManufDB.FailureList (USER, CURRENT_DATETIME,
> '1984-06-14 11:13:15.437');
isql=>
```

The following shows an invocation of the same procedure within an application program:

```
EXECUTE PROCEDURE
:ReturnCode = ManufDB.FailureList (:Operator,
CURRENT_DATETIME, :BatchStamp)
```

This example shows the use of a return status and host variables, which cannot be employed in ISQL or with rules. For more information about using host variables and return status with procedures, refer to the *ALLBASE/SQL Advanced Application Programming Guide* chapter "Using Procedures in Application Programs."

The next example shows an invocation of the ManufDB.FailureList procedure through a CREATE RULE statement:

```
isql=> CREATE RULE AFTER INSERT TO ManufDB.TestData
> WHERE PassQty < TestQty
> EXECUTE PROCEDURE
> ManufDB.FailureList(USER, CURRENT_DATETIME, BATCHSTAMP);
isql=>
```

In this case, the invocation of the procedure takes place when an INSERT operation is performed on ManufDB.TestData for a batch of parts in which there were some failures. When executing the procedure from within a rule, you can refer to the names of columns in the table on which the rule is triggered. More information about invoking procedures from rules appears in the section "Techniques for Using Procedures with Rules," later in this chapter.

Procedures and Transaction Management

A procedure that is not executed from within a rule can execute any of the following transaction management statements:

```
BEGIN WORK
COMMIT WORK
ROLLBACK WORK
ROLLBACK WORK TO SAVEPOINT
SAVEPOINT
```

Since there are no restrictions on the use of these statements, you must ensure that transactions begin and end in appropriate ways. One recommended practice is to code procedures that are **atomic**, that is, completely contained in a transaction which the procedure ends with either a COMMIT or a ROLLBACK as its final statement. An alternative recommended practice is to code procedures without any transaction management statements at all. *Note* that when you issue the EXECUTE PROCEDURE statement in an

application, and if a transaction is not already in progress, a transaction is begun. If a transaction is already in progress at the time EXECUTE PROCEDURE is issued, and the procedure issues either a COMMIT or a ROLLBACK statement to end the transaction, the entire transaction, including the portion in the application, is affected.

In all cases, it is important to document procedures carefully. Refer to the section "Recommended Coding Practices for Procedures" later in this chapter.

When a procedure is executed from within a rule, all the transaction management statements are disallowed and result in an error.

Using SQL Statements in Procedures

Within a procedure, you can use most of the SQL statements that are allowed in embedded SQL application programs, including COMMIT WORK, ROLLBACK WORK, and ROLLBACK WORK TO SAVEPOINT. The following (including dynamic SQL statements) are *not* allowed in procedures:

```
ADVANCE

BEGIN DECLARE SECTION
BULK statements

CLOSE USING

COMMIT WORK RELEASE
CONNECT
CREATE PROCEDURE (including inside CREATE SCHEMA)

DECLARE CURSOR for EXECUTE PROCEDURE

DESCRIBE
DISCONNECT
END DECLARE SECTION
EXECUTE
EXECUTE IMMEDIATE
EXECUTE PROCEDURE
GENPLAN
INCLUDE

OPEN USING

PREPARE
RELEASE
ROLLBACK WORK RELEASE
SET CONNECTION
SET DML ATOMICITY
SET MULTITRANSACTION
SET SESSION
SET TRANSACTION
SQLEXPLAIN
START DBE
STOP DBE
```

In procedures that are invoked by execution of rules, the following statements result in an error:

```
BEGIN WORK
COMMIT WORK
ROLLBACK WORK
ROLLBACK WORK TO SAVEPOINT
SAVEPOINT
```

Another set of statements is provided for use *only* within procedures:

```
Assignment (=)
BEGIN...END
DECLARE Variable
GOTO
IF...THEN...ELSEIF...ELSE...ENDIF
Labeled Statements
PRINT
RETURN
WHILE...DO...ENDWHILE
```

Inside procedures, statements are terminated with a semicolon (;).

You can define **parameters** for passing information into and out of a procedure. In addition, procedures let you store data in **local variables**, which are declared inside the procedure with the DECLARE Variable statement.

Specifying Parameters

A parameter represents a value that is passed between a procedure and an invoking application or rule. You define formal parameters with the CREATE PROCEDURE statement.

When executing a procedure directly, you pass input parameter values in the EXECUTE PROCEDURE statement, and output parameter values are returned when the procedure terminates. However, when using a procedure cursor, input parameter values must be set before opening the cursor, and output parameter values are returned when the CLOSE statement executes.

Within the body of the procedure, a parameter name is prefixed with a colon (:).

You can specify up to 1023 parameters of any SQL data type except the LONG data types. Default values and nullability may be defined just as in a CREATE TABLE statement. If a language is specified for a parameter defined as a CHAR or VARCHAR type, it must be either the language of the DBEnvironment or else NATIVE 3000. The following shows a procedure with a single parameter:

```
CREATE PROCEDURE Process10 (PartNumber CHAR(16)) AS
   BEGIN
     .
     .
     .
   END;
```

If you wish to return values to a calling application program, specify the parameter for OUTPUT in both the CREATE PROCEDURE and EXECUTE PROCEDURE statements. If no input value is required for a parameter, specify OUTPUT ONLY. Note that no OUTPUT option is allowed in the EXECUTE PROCEDURE statement in ISQL nor in the EXECUTE PROCEDURE clause of the CREATE RULE statement.

Using Local Variables in Procedures

A local variable holds a data value within a procedure. Local variable declarations must appear at the beginning of the main body of the procedure using the DECLARE statement, and they must specify a data type and size. Optionally, the DECLARE statement can include nullability, language, and a default value. The following are typical examples:

```
DECLARE LastName CHAR(40);
DECLARE SalesPrice DECIMAL(6,2);
DECLARE LowPrice, HighPrice DECIMAL(6,2) NOT NULL;
DECLARE LocationCode INTEGER NOT NULL;
DECLARE Quantity INTEGER DEFAULT 0;
```

Types and sizes are the same as for column definitions, except that you cannot specify a LONG local variable. You can declare several variables in the same DECLARE statement by separating them with a comma provided they share the same data type, size, nullability, native language, and default value. Within the body of the procedure, a local variable name is prefixed with a colon (:). A local variable name cannot duplicate a parameter name.

Local variables function in procedures much as host variables function in application programs, but the two are not interchangeable. That is, you cannot use host variables from the application within the body of the procedure definition nor can you use local variables in the application. Since the application's host variables cannot be directly accessed from within the procedure, you must use local variables or parameters in the INTO clause of any FETCH, REFETCH, or SELECT statement within a procedure. Then, if necessary, you transfer data to a calling application through output parameters. If multiple rows must be returned to the calling application, a SELECT statement with no INTO clause should be used in conjunction with a procedure cursor. Further information regarding procedure cursors is found in the “Using Procedures in Application Programs” chapter of the *ALLBASE/SQL Advanced Application Programming Guide* and in this manual under related syntax statements (ADVANCE, CLOSE, CREATE PROCEDURE, DECLARE CURSOR, DESCRIBE, EXECUTE PROCEDURE, FETCH, OPEN).

In contrast to host variables, local variables do not use indicator variables to handle NULL values. A local variable itself contains the null indicator, if the variable is nullable. Declaring a local variable to be NOT NULL makes it work like a host variable that is used without an indicator variable.

Using Built-in Variables in Procedures

The following built-in variables can be used in error handling:

Table 4-1. Built-in Variables in Procedures

Variable	Data Type	Description
::sqlcode	INTEGER	DBERR number returned after the execution of an SQL statement, 0 if no errors.
::sqlerrd2	INTEGER	Number of rows processed in an SQL statement.
::sqlwarn0	CHAR(1)	Set to "W" if an SQL warning was detected.
::sqlwarn1	CHAR(1)	Set to "W" if a character string value was truncated when being stored in a variable or parameter.
::sqlwarn2	CHAR(1)	Set to "W" if a null value was eliminated from the argument set of an aggregate function.
::sqlwarn6	CHAR(1)	Set to "W" if the current transaction was rolled back.
::activexact	CHAR(1)	Indicates whether a transaction is in progress ("Y") or not ("N"). For information about transactions, see "Managing Transactions" in the chapter "Using ALLBASE/SQL."

The built-in variables are read-only, and are not available outside of procedures. The first six of these have the same meaning that they have as fields in the SQLCA in application programs. They are always prefixed by a double colon to differentiate them from any local variables or parameters.

Note that in procedures, sqlerrd2 returns the number of rows processed for all host languages. However, in application programs, sqlerrd3 is used in COBOL, Fortran, and Pascal, while sqlerrd2 is used in C.

For procedures returning multiple row result set(s), note that the built-in variables in the procedure do not reflect the status of any FETCH or ADVANCE statements issued by the application to manipulate a procedure cursor. After issuing such a statement, the application should examine the appropriate fields of the SQLCA to determine status and handle any errors.

Queries inside Procedures

Within a procedure, you can declare parameters or local variables to process either single row or multiple row query results. Multiple row query results within a procedure must be processed one row at a time, by means of a select cursor. A select cursor is a pointer indicating the current row in a set of rows retrieved by a SELECT statement. Bulk processing is not available for a select cursor within a procedure.

Multiple row query results for queries within a procedure can be processed by means of a procedure cursor declared in a calling application. A **procedure cursor** is a pointer used to indicate the current row in a set of rows retrieved by a set of SELECT statements within a procedure. When you issue an EXECUTE PROCEDURE statement in ISQL, and the procedure contains queries with no INTO clause, ISQL uses a procedure cursor to process

the query results. Further information regarding procedure cursors is found in the "Using Procedures in Application Programs" chapter of the *ALLBASE/SQL Advanced Application Programming Guide* and in this manual in the following section, "Using a Procedure Cursor in ISQL," and under related syntax statements (ADVANCE, CLOSE, CREATE PROCEDURE, DECLARE CURSOR, DESCRIBE, EXECUTE, EXECUTE IMMEDIATE EXECUTE PROCEDURE, FETCH, OPEN).

The following sections discuss the use of a simple select, a select cursor, and an ISQL procedure cursor.

Using a Simple SELECT

A simple SELECT statement with an INTO clause returns only a single row. If more than one row qualifies for the query result, only the first row is put into the parameter or local variable specified in the INTO clause, and a warning is issued. Example:

```
CREATE PROCEDURE PurchDB.DiscountPart(PartNumber CHAR(16))
AS BEGIN
      DECLARE SalesPrice DECIMAL(6,2);

      SELECT SalesPrice INTO :SalesPrice
        FROM PurchDB.Parts
       WHERE PartNumber = :PartNumber;

      IF ::sqlcode = 0 THEN
         IF :SalesPrice > 100. THEN
            :SalesPrice = :SalesPrice*.80;
            INSERT INTO PurchDB.Discounts
                 VALUES (:PartNumber, :SalesPrice);
         ENDIF;
      ENDIF;
   END;
```

The procedure inserts a row into the PurchDB.Discounts table containing the part number and 80% of the sales price if the current price of a given part is over $100. The parameter PartNumber supplies a value for the predicate in the SELECT statement and later supplies a value for the VALUES clause in the INSERT statement. The local variable :SalesPrice is used for the single-row result of the query on the Parts table, and it is also used in the expression in the VALUES clause of the INSERT statement. The procedure tests if the built-in variable ::sqlcode = 0 to ensure that the SELECT was successful before inserting data into the PurchDB.Discounts table.

Using a Select Cursor

If your procedure must process a set of rows one at a time, you can use a cursor to loop through the set and perform desired operations, as in the following:

```
CREATE PROCEDURE PurchDB.DiscountAll(Percentage DECIMAL(4,2))
AS BEGIN
    DECLARE SalesPrice DECIMAL(6,2);
    DECLARE C1 CURSOR FOR SELECT SalesPrice FROM PurchDB.Parts
        FOR UPDATE OF SalesPrice;
    OPEN C1;
    WHILE ::sqlcode = 0 DO
       FETCH C1 INTO :SalesPrice;
```

```
        IF ::sqlcode = 0 THEN
           IF :SalesPrice < 1000. THEN
              UPDATE PurchDB.Parts
                 SET SalesPrice = :SalesPrice*:Percentage
                 WHERE CURRENT OF C1;
           ELSEIF :SalesPrice >= 1000. THEN
              UPDATE PurchDB.Parts
                 SET SalesPrice = :SalesPrice*(:Percentage - .05)
                 WHERE CURRENT OF C1;
           ENDIF;
        ENDIF;
     ENDWHILE;
     IF ::sqlcode = 100 THEN
        PRINT 'Success';
        CLOSE C1;
        RETURN;
     ELSE
        PRINT 'Error in Fetch or Update';
        CLOSE C1;
        RETURN;
     ENDIF;
END;
```

This procedure discounts the prices of all part numbers by a specified percentage if the current sales price is less than $1000, and it discounts prices by five percentage points for part numbers whose current price is greater than or equal to $1000. The procedure displays a message indicating success or failure.

The use of select cursors for multiple row query results is presented in great detail in the ALLBASE/SQL application programming guides. Refer to the chapter “Processing with Cursors” in the guide for the programming language you use.

Using a Procedure Cursor in ISQL

When you issue an EXECUTE PROCEDURE statement in ISQL for a procedure containing one or more SELECT statements with no INTO clause, ISQL uses a procedure cursor to display the query results.

For example, create a procedure as follows:

```
CREATE PROCEDURE PurchDB.PartNo2 AS
BEGIN
SELECT *
  FROM PurchDB.Parts
    WHERE PartNumber LIKE '11%';

SELECT PartNumber, BinNumber, QtyOnHand
  FROM PurchDB.Inventory
    WHERE PartNumber LIKE '11%';
END;
```

When you execute the procedure, the following is displayed:

```
execute procedure purchdb.partno2;
---------------+----------------------------+-----------------
PARTNUMBER     |PARTNAME                    |SALESPRICE
---------------+----------------------------+-----------------
1123-P-01      |Central Processor           |          500.00
1133-P-01      |Communication Processor     |          200.00
1143-P-01      |Video Processor             |          180.00
1153-P-01      |Graphics Processor          |          220.00

------------------------------------------------------------------------
Number of rows selected is 4
U[p], d[own], l[eft], r[ight], t[op], b[ottom], pr[int] <n>, e[nd] or n[ext] >
```

Entering n[ext] moves you from one SELECT statement to the next. You would see the following:

```
execute procedure purchdb.partno2;
---------------+---------+---------
PARTNUMBER     |BINNUMBER|QTYONHAND
---------------+---------+---------
1123-P-01      |     4003|        5
1133-P-01      |     4007|       11
1143-P-01      |     4016|        8
1153-P-01      |     4027|        5


------------------------------------------------------------------------
Number of rows selected is 4
U[p], d[own], l[eft], r[ight], t[op], b[ottom], pr[int] <n>, e[nd] or n[ext] >
```

Entering n[ext] when the last result set is displayed produces a message like the following:

```
End of procedure result sets.
Procedure return status is 0.
isql=>
```

Note that although you can move back and forward through the current result set, you cannot move back to redisplay a previous result set.

Error Handling in Procedures Not Invoked by Rules

You must provide explicit mechanisms for error handling inside procedures. The techniques you use for this depend on whether or not the procedure is invoked by the firing of a rule. This section describes error handling within a procedure that is *not* invoked by a rule. For information about error handling in procedures invoked by rules, see the section "Error Handling in Procedures Invoked by Rules," below. For information about error handling in an application that invokes a procedure, see the section "Using Procedures in Application Programs" in the *ALLBASE/SQL Advanced Application Programming Guide*.

By default, when an error occurs in an SQL statement in a procedure, the effects of the SQL statement are undone, but the procedure continues on to the next statement. If you want errors in SQL statements to cause an immediate error return from the procedure, use the WHENEVER statement with the STOP option.

The syntax for the WHENEVER is as follows:

```
WHENEVER (SQLERROR
          SQLWARNING
          NOT FOUND}{STOP
                     CONTINUE
                     GOTO [:]Label
                     GO TO [:]Label}
```

The STOP option causes the current transaction to be rolled back, and the procedure's execution is terminated. If an error occurs in evaluating the condition in an IF or WHILE statement, or in evaluating the expression in a parameter or variable assignment statement, the execution of the procedure terminates, and control is returned to the caller with SQLCODE set to the last error encountered inside the procedure.

Within the procedure, the entire message buffer is not available. That is, SQLEXPLAIN cannot be used. The built-in variable *::sqlcode* holds only the error code from the first message in the message buffer (guaranteed to be the most severe error).

In procedures, as elsewhere in ALLBASE/SQL, the message buffer is cleared out only before executing an SQL statement. That is, execution of the following do *not* cause the message buffer to be reset:

- Assignment
- GOTO
- IF
- PRINT
- RETURN
- WHILE

The argument of any PRINT statement is passed back to the caller in the message buffer. When the message buffer is reset, PRINT statements are not removed.

Runtime errors are accompanied by a generic error message indicating, by number, which procedure statement caused the error. All SQL statements in a procedure and all non-SQL statements except variable declarations, ENDIF, ELSE, ENDWHILE, END, and THEN, are numbered consecutively from the beginning of the procedure. The following is an example of a sequence of errors returned when an EXECUTE PROCEDURE statement fails:

```
Integer divide by zero.  (DBERR 2601)
Error occurred executing procedure PURCHDB.DISCOUNT statement 2.(DBERR 2235)
Error occurred during evaluation of the condition in an IF or WHILE
statement or the expression in a parameter or variable assignment.
Procedure execution terminated.  (DBERR 2238)
```

Using RAISE ERROR in Procedures

You can use the RAISE ERROR statement to generate an error within a procedure and make a message available to users, as in the following example:

```
RAISE ERROR 7500 MESSAGE 'Error Condition';
RETURN 1;
```

The RAISE ERROR statement causes the message to be stored in the message buffer, and

the RETURN statement causes an immediate return from the procedure following the error. Following the return from a procedure, an application program can retrieve the messages from raised errors by using the SQLEXPLAIN statement. Since SQLCODE is 0 in this case (because the procedure executed correctly; it was an SQL statement within it that received the error), you should execute SQLEXPLAIN in a loop that tests SQLWARN[0], as follows:

```
while (sqlwarn[0]=='W')
   EXEC SQL SQLEXPLAIN :SQLMessage;
```

However, SQLEXPLAIN cannot be used within the procedure itself. You should document the cause of all errors generated by the RAISE ERROR statement in a procedure so that the procedure caller can understand the error condition.

NOTE The behavior of errors, including RAISE ERROR, in procedures called by rules differs somewhat from that described here. Refer to "Using RAISE ERROR in Procedures Invoked by Rules" for more information.

Recommended Coding Practices for Procedures

The use of procedures can have indirect consequences that the procedure writer and the procedure caller may not anticipate. Problems are most likely to arise in the areas of transaction management, cursor management, error handling, and DBEnvironment settings. In order to minimize difficulty, good communication between the procedure writer and the caller of the procedure is essential. Thus procedures should be carefully documented as to what is expected from the calling application, and applications should be carefully documented as to what they expect a called procedure to do and not to do.

Within a procedure, you can use ISQL comments or comment notation for the programming language of an application that invokes a procedure. See the *ALLBASE/ISQL Reference Manual* or the appropriate ALLBASE/SQL application programming guide for information about comments.

The following practices are suggested to ensure that a procedure is always called under the same conditions and with the same expectations:

- If the procedure might execute a COMMIT or ROLLBACK, the application should issue a COMMIT or ROLLBACK before calling the procedure. Any cursors opened in the application with the KEEP cursor option and subsequently committed should be closed and committed before the application calls the procedure.
- Documentation of the calling application should clearly state the following:
  - — Whether the procedure will be called with a transaction open.
  - — Whether the procedure is expected to have COMMIT or ROLLBACK statements.
  - — Whether the procedure is expected to be atomic.

The following practices are suggested to ensure that a procedure will always execute as expected:

- Procedure execution should not span transaction boundaries. Either the procedure should be treated as an atomic transaction, that is, it should always issue a COMMIT or

ROLLBACK statement upon completion of work and before termination; or it should be entirely contained within a transaction, that is, it should not contain any COMMIT or ROLLBACK statements.

- If the procedure executes any COMMIT or ROLLBACK statements, it should be treated as a transaction. This means that the last statement accessing the DBEnvironment within the procedure should be a COMMIT WORK or a ROLLBACK WORK statement.
- If the procedure uses any cursors, they should be closed before termination. If the procedure opens any cursors with the KEEP option, and subsequently executes any COMMIT statements, the cursors should be closed and committed before termination.
- A procedure should not change the application's environment without restoring it upon termination. The application's environment includes settings for isolation level, constraint checking, timeout values, and rule firing.
- Documentation of the procedure should clearly state the following:
  - — Whether or not a transaction should already exist at the time of procedure execution.
  - — Whether any COMMIT or ROLLBACK statements will be executed by the procedure.
  - — Whether the procedure modifies any environment settings.
  - — What types of errors are handled by the procedure and how they are handled.
  - — Meanings of all possible return status values.
  - — Meaning of any errors returned by RAISE ERROR statements.

Using Rules

Rules allow you to tie procedures to data manipulation statements. Rules are more flexible than simple integrity constraints, enabling you to incorporate complex business rules into the structure of a DBEnvironment with minimal application programming. The following sections describe the use of rules:

- Understanding Rules
- Creating Rules
- Techniques for Using Procedures with Rules
- Error Handling in Procedures Invoked by Rules
- Using RAISE ERROR in Procedures Invoked by Rules
- Enabling and Disabling Rules
- Special Considerations for Procedures Invoked by Rules
- Differences between Rules and Integrity Constraints

Understanding Rules

Rules allow you to define generalized constraints by invoking procedures whenever specified operations are performed on a table. The rule **fires**, that is, invokes a procedure, each time the specified operation (such as INSERT, UPDATE, or DELETE) is performed and the rule's search condition is satisfied.

Rules tie procedures to particular kinds of data manipulation statements on a table. This permits data processing to be carried out by the DBEnvironment itself. The effect is less application coding and more efficient use of resources. This is especially important for networked systems.

Rules will fire under the following conditions:

- The rule's statement types must include the statement type of the current statement. Statement types are INSERT, DELETE, and UPDATE. (You can have more than one statement type per rule.)
- If the rule's statement type includes UPDATE, and if the *StatementType* clause includes a list of columns in the table, and if the current statement is an update, it must be on at least one of the listed columns of that table.
- The rule's search condition must evaluate to TRUE for the current row of the current statement.

A rule fires once for each row operated on by the current statement that satisfies the rule's search condition.

Creating Rules

A rule is defined in a CREATE RULE statement, which identifies a table, types of data manipulation statements, a firing condition, and a procedure to be executed whenever the condition evaluates to TRUE and the data manipulation statement is of the right type.

The following is a simple example of a rule tied to deletions from the Parts table:

```
CREATE RULE PurchDB.RemovePart
   AFTER DELETE FROM PurchDB.Parts
   WHERE SUBSTRING(PartNumber,1,4) < > 'XXXX'
   EXECUTE PROCEDURE PurchDB.ListDeletes (OLD.PartNumber);
```

The table on which the rule is defined is PurchDB.Parts. The statement type required to trigger the procedure is the DELETE operation. The search condition that must be satisfied in addition to the statement type of DELETE is that the first four characters in PartNumber must not be "XXXX." The procedure to be executed is PurchDB.ListDeletes, shown in the following:

```
CREATE PROCEDURE PurchDB.ListDeletes (PartNumber CHAR(16) NOT NULL) AS
BEGIN
   INSERT INTO PurchDB.Deletions
   VALUES (:PartNumber, CURRENT_DATETIME);
END;
```

When a row containing a part number that does not start with XXXX is deleted from the Parts table, its number is inserted along with the current date and time, in the PurchDB.Deletions table.

Techniques for Using Procedures with Rules

One common use of the rule-and-procedure combination is to enforce integrity within a DBEnvironment. This can be done in different ways, depending on your needs. The following sections contrast two approaches to integrity enforcement:

- Using Rule Chaining
- Using a Single Procedure

Using a Chained Set of Procedures and Rules

The following example uses a chained set of procedures and rules to remove all references to a part number once it has been deleted from the database. In this case a rule fires a procedure, which causes another delete, which causes another rule to invoke an additional procedure, and so on.

```
CREATE PROCEDURE PurchDB.RemovePart (PartNum CHAR(16) NOT NULL)
AS BEGIN
   DELETE FROM PurchDB.Inventory WHERE PartNumber = :PartNum;
   DELETE FROM PurchDB.SupplyPrice WHERE PartNumber = :PartNum;
END;

CREATE RULE PurchDB.RemovePart
AFTER DELETE FROM PurchDB.Parts
EXECUTE PROCEDURE PurchDB.RemovePart (OLD.PartNumber);

CREATE PROCEDURE PurchDB.RemoveVendPart (VendPartNum CHAR(16) NOT NULL)
AS BEGIN
   DELETE FROM PurchDB.OrderItems WHERE VendPartNumber = :VendPartNum;
   DELETE FROM ManufDB.SupplyBatches WHERE VendPartNumber = :VendPartNum;
END;

CREATE RULE PurchDB.RemoveVendPart
AFTER DELETE FROM PurchDB.SupplyPrice
EXECUTE PROCEDURE PurchDB.RemoveVendPart (OLD.VendPartNumber);

CREATE PROCEDURE ManufDB.RemoveBatchStamp (BatchStamp DATETIME NOT NULL)
AS BEGIN
   DELETE FROM ManufDB.TestData WHERE BatchStamp = :BatchStamp;
END;

CREATE RULE ManufDB.RemoveBatchStamp
AFTER DELETE FROM ManufDB.SupplyBatches
EXECUTE PROCEDURE ManufDB.RemoveBatchStamp (OLD.BatchStamp);
```

Executing the Chained Set of Procedures and Rules

Whenever a user performs a DELETE operation on PurchDB.Parts, the procedures and rules are executed on each row of each table for the identified part number in the following order:

1. Delete from *Parts* table.
2. Fire rule *RemovePart*.
3. Invoke procedure *RemovePart*.
4. Delete from *Inventory* table.
5. Delete from *SupplyPrice* table.
6. Fire rule *RemoveVendPart*.
7. Invoke procedure *RemoveVendPart*.
8. Delete from *OrderItems* table.
9. Delete from *SupplyBatches* table.
10. Fire rule *RemoveBatchStamp*.
11. Delete from *TestData* table.

Using a Single Procedure with Cursors

The following example uses a single rule and one procedure to remove all references to a part number once it has been deleted from the database. In this case, a single procedure *RemovePart* determines which rows need to be deleted in the other tables once a part number is deleted from the Parts table. Since this method only uses one rule and one procedure, it would be effective only when a DELETE is done from the Parts table. Deletions of part numbers from other tables would not trigger any rules at all.

The single procedure uses two cursors to scan the PurchDB.SupplyPrice and ManufDB.SupplyBatches tables for entries that correspond to a deleted part number. The procedure then performs deletions of qualifying rows in PurchDB.OrderItems and ManufDB.TestData.

```
CREATE PROCEDURE PurchDB.RemovePart(PartNum CHAR(16) NOT NULL)
AS BEGIN
   DECLARE VendPartNum CHAR(16) NOT NULL;
   DECLARE BatchStamp DATETIME NOT NULL;
   DECLARE SupplyCursor CURSOR FOR
      SELECT VendPartNumber FROM PurchDB.SupplyPrice
      WHERE PartNumber = :PartNum;
   DECLARE BatchCursor CURSOR FOR
      SELECT BatchStamp FROM ManufDB.SupplyBatches
      WHERE VendPartNumber = :VendPartNum;

   DELETE FROM PurchDB.Inventory WHERE PartNumber = :PartNum;
```

Open the first cursor:

```
      OPEN SupplyCursor;
      FETCH SupplyCursor INTO :VendPartNum;

      WHILE ::sqlerrd2 = 1 DO
         DELETE FROM PurchDB.OrderItems WHERE VendPartNumber = :VendPartNum;
```

Open the second cursor:

```
        OPEN BatchCursor;
         FETCH BatchCursor INTO :BatchStamp;

         WHILE ::sqlerrd2 = 1 DO
            DELETE FROM ManufDB.TestData WHERE BatchStamp = :BatchStamp;
            FETCH BatchCursor INTO :BatchStamp;
         ENDWHILE;

         CLOSE BatchCursor;

         DELETE FROM ManufDB.SupplyBatches WHERE VendPartNumber = :VendPartNum;
         FETCH SupplyCursor INTO :VendPartNum;
      ENDWHILE;
      CLOSE SupplyCursor;
      DELETE FROM PurchDB.SupplyPrice WHERE PartNumber = :PartNum;
   END;
```

The single rule that invokes the above procedure is as follows:

```
   CREATE RULE PurchDB.RemovePart
   AFTER DELETE FROM PurchDB.Parts
   EXECUTE PROCEDURE PurchDB.RemovePart (OLD.PartNumber);
```

Error Handling in Procedures Invoked by Rules

When invoked by a rule, a procedure is executed inside the execution of a data manipulation statement. Therefore, if the procedure encounters an error, the effect of the procedure and the effect of the data manipulation statement as a whole are undone. Statements that may fire rules always execute with statement atomicity, regardless of the current general error checking level set by the SET DML ATOMICITY statement.

Inside procedures invoked by rules, SQL errors have the usual effect of issuing messages, halting execution of the current statement, rolling back a transaction, or ending a connection. In addition, even if the error does not result in rolling back a transaction or losing a connection, it results in the undoing of the effects of all procedures invoked in a chain by the current statement, and it results in the undoing of the effects of all rules triggered by the current statement. Thus the entire execution of the statement is undone.

Using RAISE ERROR in Procedures Invoked by Rules

Within a procedure which is triggered by a rule, the RAISE ERROR statement can be used to generate an error, which causes an immediate return and undoes the statement that triggered the rule. The text of the RAISE ERROR message can provide useful information to the user such as the procedure name, the exact reason for the error, the location in the

procedure, or the name of the rule that invoked the procedure (if the procedure is only fired by one rule).

Suppose the following rule executes whenever a user attempts to delete a row in the Vendors table:

```
CREATE RULE PurchDB.CheckVendor
AFTER DELETE FROM PurchDB.Vendors
EXECUTE PROCEDURE PurchDB.DelVendor (OLD.VendorNumber);
```

The procedure PurchDB.DelVendor checks for the existence of the use of a vendor number elsewhere in the database, and if it finds that the number is being used, it rolls back the delete on the Vendors table. The procedure is coded as follows:

```
CREATE PROCEDURE PurchDB.DelVendor (VendorNumber INTEGER NOT NULL) AS
BEGIN
   DECLARE rows INTEGER NOT NULL;

   SELECT COUNT(*) INTO :rows FROM PurchDB.Orders
      WHERE VendorNumber = :VendorNumber;
   IF :rows <> 0 THEN
      RAISE ERROR 1 MESSAGE 'Vendor number exists in the "Orders" table.';
   ENDIF;

   SELECT COUNT(*) INTO :rows FROM PurchDB.SupplyPrice
     WHERE VendorNumber = :VendorNumber;
   IF :rows <> 0 THEN

      RAISE ERROR 1 MESSAGE 'Vendor number exists in "SupplyPrice" table.';
   ENDIF;
END;
```

PurchDB.DelVendor checks for the existence of the use of a vendor number in two tables: PurchDB.Orders and PurchDB.SupplyPrice. If it retrieves any rows containing the vendor number, it returns an error code and a string of text to the caller by means of the RAISE ERROR statement.

The following shows the effect of the rule and procedure when you attempt to delete a row from the Vendors table in ISQL:

```
isql=> DELETE FROM purchdb.vendors WHERE vendornumber = 9006;
Vendor number exists in the "Orders" table.
Error occurred executing procedure PURCHDB.DELVENDOR statement 3.
(DBERR 2235)
INSERT/UPDATE/DELETE statement had no effect due to execution errors.
(DBERR 2292)
Number of rows processed is 0
isql=>
```

The DELETE statement triggers the rule, which executes the procedure PurchDB.DelVendor. If the vendor number that is to be deleted is not found in either of the two tables, sqlcode is 0, and no messages are displayed.

When a procedure is called through the use of a rule, the procedure exits as soon as an error occurs. This can be either an ordinary SQL error (but not a warning), or a user-defined error produced with the RAISE ERROR statement. After an error return, the statement that fired the rule is undone, and the operation of all other rules fired by the

statement is also undone.

In application programs, you use SQLEXPLAIN to retrieve the messages generated by RAISE ERROR and other SQL statements.

Enabling and Disabling Rules

Rule processing takes place by default in the DBEnvironment. However, the DBA can use the following statement to disable the operation of rules in the current session:

```
isql=> disable rules;
```

This statement, which is useful in debugging, should be employed only with great care, since it can affect the integrity of the database, if rules are being used to control data integrity. To restore the operation of rules in the session, use the following statement:

```
isql=> enable rules;
```

Rules are not fired retroactively when the ENABLE RULES statement is issued after the DISABLE RULES statement has been issued.

Special Considerations for Procedures Invoked by Rules

Procedures operate somewhat differently when invoked by rules than when invoked directly by a user. The differences are most pronounced in several areas:

- Transaction handling.
- Effects of rule chaining.
- Invalidation of sections.
- Changing session attributes.
- Performance considerations.

Transaction Handling in Rules

Since rules are fired by data manipulation statements that are already being executed, a transaction is always active when a rule invokes a procedure. Therefore, BEGIN WORK and BEGIN ARCHIVE statements will result in errors in a procedure invoked by a rule. The error will cause the rule to fail and the user's statement to be undone.

COMMIT WORK, COMMIT ARCHIVE, ROLLBACK WORK, ROLLBACK ARCHIVE, SAVEPOINT, and ROLLBACK TO SAVEPOINT statements will generate errors when encountered in procedures triggered by rules. The error causes the user's statement and all subsequent rule-driven statements to be undone. If you wish to include COMMIT WORK, COMMIT ARCHIVE, ROLLBACK WORK, ROLLBACK ARCHIVE, SAVEPOINT, or ROLLBACK TO SAVEPOINT statements in the procedure, because the procedure will be executed by users directly as well as by rules, you should include these statements within a condition that will only be true for non-rule invocation. To do this, add a flag parameter to the procedure. Have users invoking the procedure pass in a fixed value (such as 0), and have rules invoking the procedure pass in a different value (such as 1).

Then the procedure can be coded with IF statements like the following:

```
if :Flag = 0 then
   commit work;
endif;
```

The flag check ensures that the rule will not execute statements that would cause it to generate an error when the procedure is invoked by a rule, while user calls can commit or roll back changes automatically.

Effects of Rule Chaining

Procedures invoked by rules can include data manipulation statements that invoke rules that trigger the execution of other procedures. Excessive chaining of rules in this fashion uses additional system resources. When the chain length exceeds 20, an error occurs, which causes the user's statement to be undone. To avoid problems, be sure to trace the dependencies of statements within procedures invoked by rules so as to:

- avoid an endless loop of rule chaining.
- avoid exceeding a rule depth greater than the maximum of 20.
- control and maintain the rule system with minimal complexity.

To assist in tracing, the DBA can use the `SET PRINTRULES ON` statement to display the names of rules being fired.

The rule developer should also determine if multiple rules will apply to the same data manipulation statement. An analysis of the rule type and WHERE conditions can be done to see whether any rules overlap in statement type on a given table, and whether their conditions are mutually exclusive or not. The rules are checked for each row an `INSERT`, `DELETE`, or `UPDATE` statement affects. If multiple rules can affect a single row, the order of their execution is not guaranteed to be fixed if the section is ever revalidated. To avoid potential problems, it is best to ensure that rules affecting the same statement have mutually exclusive WHERE conditions or that the order of execution of the procedures they invoke is unimportant.

Invalidation of Sections

Procedures can include data definition statements that affect the execution of procedures and rules by invalidating sections. Use care when issuing the following statements inside procedures:

- `DROP PROCEDURE`. If a rule depends on the procedure, all sections checking that rule will be invalidated by the `DROP PROCEDURE` statement, and will fail to be revalidated.
- `CREATE RULE` and `DROP RULE`. Because rule enforcement is checked during the lifetime of the rule, `CREATE RULE` and `DROP RULE` should be used with care. If a rule that is currently among those checked for a statement is dropped within a procedure invoked by a rule on behalf of that statement, the statement will be invalidated while it is still being executed. In this situation, execution will halt, an error will occur, and the statement will be undone.
- Any data definition. Within a procedure invoked by a rule, if any DDL is performed which invalidates a statement currently being executed (either the user's statement, or

a statement within an invoked procedure which chained another rule), an error will occur, and the user's statement will be undone.

Changing Session Attributes

Procedures should avoid the following statements, which change the attributes of transactions or sessions:

- `SET CONSTRAINTS`
- `DISABLE RULES`
- `ENABLE RULES`
- `SET PRINTRULES`
- `SET USER TIMEOUT`

If you include one of these statements in a procedure invoked by a rule, consider its effect carefully. If any of these statements is executed by a procedure invoked by a rule, and it causes the setting of the attribute to change, then the user's statement will execute partly in the original mode and partly in the altered mode. In the event of rule chaining, attributes might change several times. If a statement that invokes a procedure is undone, any settings modified by the procedure are restored to their values prior to the issuing of the statement.

The `SET CONSTRAINTS` statement will change the application of check constraints as of the next statement in the procedure, and this change will affect the remainder of the set of rows defined by the triggering statement. The `SET CONSTRAINTS` statement will change the application of unique and referential constraints as of the user's next statement--that is, the statement following the one that invoked the procedure through a rule.

The `DISABLE RULES` statement will have no effect on the firing of the rules on their respective current rows. It will only affect rows not yet checked and rules not yet fired. `DISABLE RULES` can be used to ensure that the rule depth of 20 is not exceeded, if the chain of rule dependencies is understood well enough for the appropriate placement of this statement.

`SET PRINTRULES ON` and `SET PRINTRULES OFF` affect the printing of rule names of rules not yet fired, or of rows not yet checked.

Performance Considerations

The placement of conditions on execution of statements within the firing of a rule should be examined carefully. Firing conditions placed in the WHERE clause can avoid the overhead of loading and invoking the procedure, since the WHERE condition is checked before the procedure is invoked. Thus, it might be better to develop several rules with separate conditions and procedures with well-defined actions rather than a single rule with no condition and a single procedure that makes checks before deciding what steps to carry out. To determine the best design for your needs, weigh the overhead of frequent loading and executing of a procedure against the overhead of maintaining several procedures and rules.

Differences between Rules and Integrity Constraints

Rules are similar to integrity constraints in that when a rule is created, all existing `INSERT`, `UPDATE`, and `DELETE` statements will be affected by the rule (if the statement type is appropriate to the rule). Rules are viewed as changes to the table definition, and so all existing sections depending on the table are invalidated when a rule is created. When these sections are next revalidated, the rule definition is picked up and compared to the section; appropriate rules are then included in the revalidated section for checking at statement execution time.

The following are some of the most important ways in which rules differ from integrity constraints:

- Rules are entirely reactive. They are not fired at `CREATE RULE` time against the existing rows in the table. Moreover, after `DISABLE RULES`, no record is kept of rows the rule would have fired on; so, when the `ENABLE RULES` statement is next issued, the rule is not fired retroactively. Integrity constraints, on the other hand, are always checked when an `ALTER TABLE` statement is issued with the `ADD CONSTRAINT` clause, and when `SET CONSTRAINTS IMMEDIATE` is executed.
- Rules only fire on the statement types they are defined to fire on, whereas integrity constraints will be checked on all data change operations.
- Rules do not use index structures to enforce the constraints they define; some integrity constraints build special indexes.
- The only side effect of the integrity constraint is an error, while a rule can have many different side effects depending on the actions of the procedure it invokes.
- In addition to providing a general way of implementing constraints, rules can be used to define more abstract tasks such as logging the changes made to a table or enforcing stricter security measures developed by the database designer. Rules are most useful in defining complex relationships that cannot be modeled with existing check, unique, or referential constraints.

5 Concurrency Control through Locks and Isolation Levels

Concurrency control is the process of regulating access to the same data by multiple transactions operating in the same DBEnvironment. Without regulation, a database could easily become inconsistent or corrupt. Consider what can happen if two or more concurrent users access the same data *without* any concurrency control. For example, one user could delete a row while another user is in the process of updating it. Or one user might update a row, and a second user might make a decision based on the update, then the first user might decide to roll back the update, at which point the second user's decision becomes invalid. To avoid problems of this type, it is important to regulate the kinds of access to database tables available to concurrent users.

This chapter describes the methods employed by ALLBASE/SQL to provide concurrency control for multiuser DBEnvironments. A section is devoted to each of the following topics:

- Defining Transactions
- Understanding ALLBASE/SQL Data Access
- Use of Locking by Transactions
- Defining Isolation Levels between Transactions
- Details of Locking
- What Determines Lock Types
- Scope and Duration of Locks
- Examples of Obtaining and Releasing Locks
- Resolving Conflicts among Concurrent Transactions
- Monitoring Locking with SQLMON

The techniques of concurrency control described in this chapter are normally implemented through application programs, though you can use some of them interactively as well.

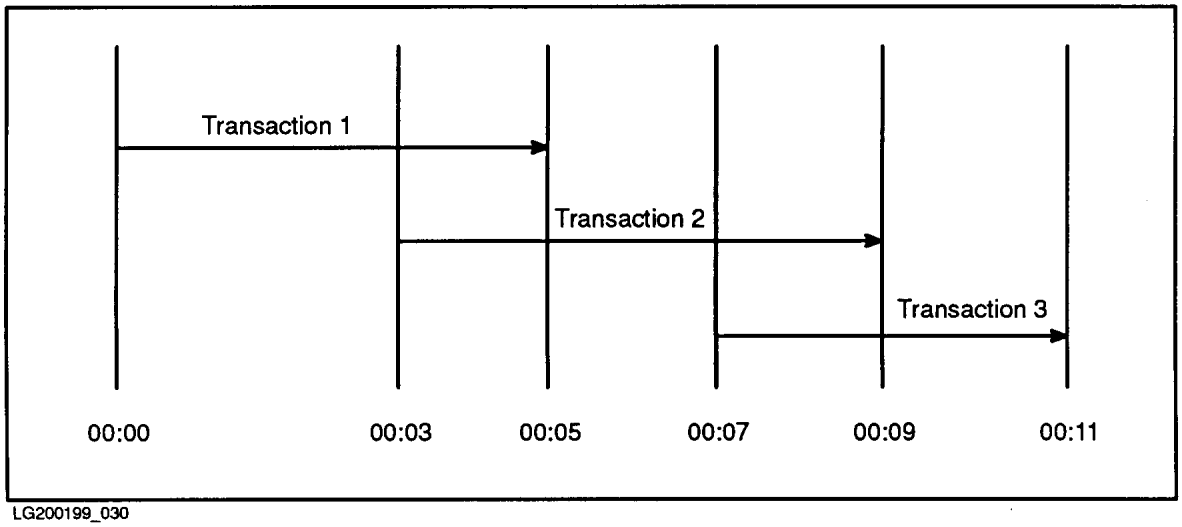
Concurrency is a complex subject. If you are a new user of relational technology or of ALLBASE/SQL, you should read the entire chapter before attempting to use any of the special features described here.

Defining Transactions

Concurrency control in ALLBASE/SQL operates at the level of the transaction, which identifies an individual user's unit of work within a multiuser DBEnvironment. As mentioned in a previous chapter, transactions are bounded by `BEGIN WORK` and `COMMIT WORK` statements. If you omit the `BEGIN WORK` statement, ALLBASE/SQL issues one automatically, using the RR (repeatable read) isolation level. ALLBASE/SQL keeps track of which transactions are accessing which pages of data at a particular moment in time. Transactions have unique ID numbers which are listed in the `SYSTEM.TRANSACTION` pseudotable in the system catalog.

Transactions can be seen as taking place over time, as in Figure 5-1.

Figure 5-1. Transactions over Time



In this example, transaction 2 begins before transaction 1 ends; therefore, transaction 1 and transaction 2 are concurrent transactions. Transaction 3 begins after transaction 1 has committed; therefore, transaction 1 and transaction 3 are not concurrent, since they do not occupy the same time.

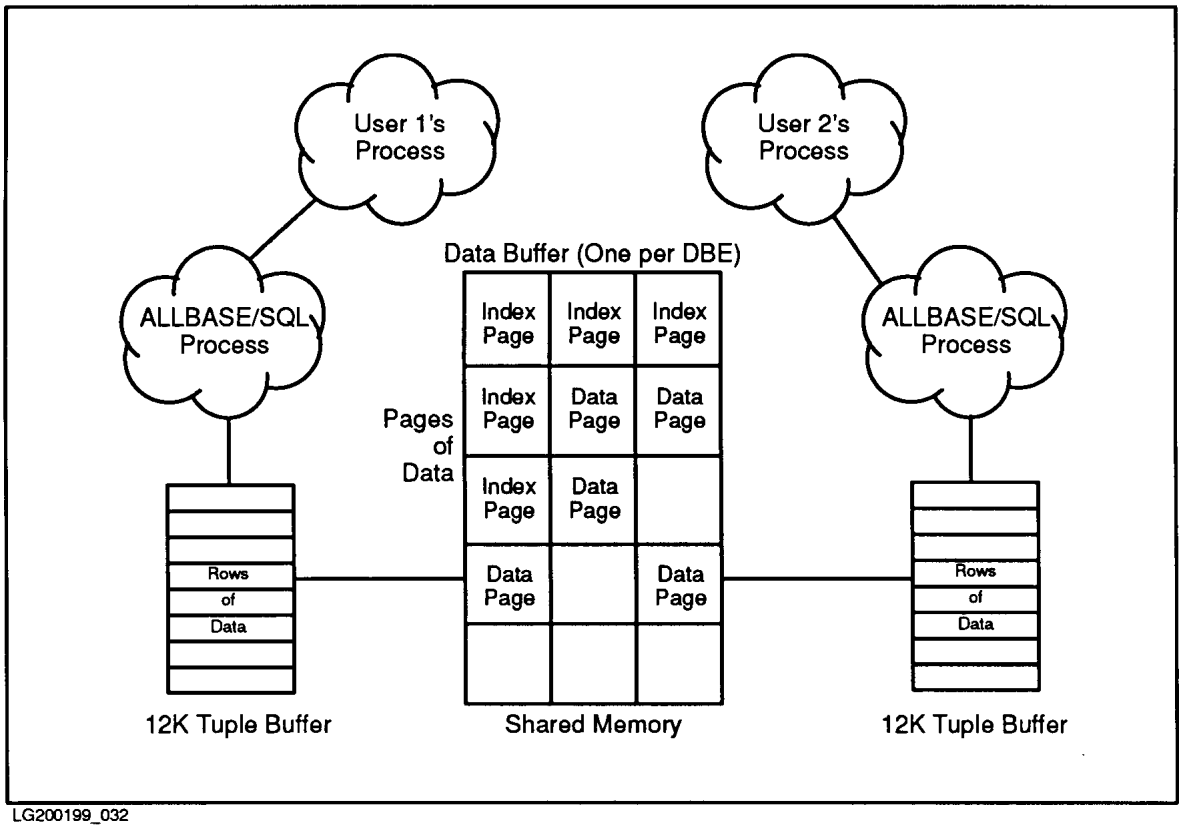
Concurrent transactions that need to access the same data pages may be in contention for a particular table, page, or row at a particular moment. Suppose transaction 1 needs to access an entire table as part of a reporting application. If transaction 2 needs to update parts of that table, it may need to wait until transaction 1 is complete before the update can proceed.

Understanding ALLBASE/SQL Data Access

Concurrent access to data by multiple users is facilitated by the use of a shared data buffer for all users of an ALLBASE/SQL DBEnvironment. Understanding how this buffer is used can clarify many concurrency issues.

A DBEnvironment running in multiuser mode is accessed by multiple processes, as shown in Figure 5-2.

Figure 5-2. Multiuser DBEnvironment



A single **data buffer** services the needs of all users of the DBEnvironment. In addition, each interactive user or application program has its own 12K tuple buffer associated with it. The data buffer holds 4096-byte pages from the DBEFiles in which tables and indexes are stored. All pages of data requested from tables in the DBEnvironment and all index pages required for access to the data are read first into this shared data buffer. In the case of queries, qualifying rows (tuples) are read from the data buffer into the tuple buffer, and then they are transferred to the screen (in the case of ISQL) or to host variables or arrays (in the case of an application program). All changes to existing data and index pages are placed in the data buffer before being written to disk.

The use of the data buffer makes access to data efficient, because pages of data are only read into the buffer when necessary. These data pages stay in the buffer until they are swapped out when buffer space is needed for some other page. The use of the buffer also

promotes quick access to the same pages of data by different transactions, because a page may not have to be read in from disk if it is already in the buffer.

When you issue a query, you request a specific set of rows and columns from different tables in a database. The content of this set of rows and columns is the query result. For every query, ALLBASE/SQL maintains a **cursor**, which is a pointer to a row in the query result. A query result may be much larger than the size of available memory, so result rows are read into your application's tuple buffer in blocks of up to 12K at a time. As your application advances through a query result, the cursor position advances. When the application has read the last row in the tuple buffer, a new set of rows is read in until the end of the query result is reached.

NOTE In procedures or embedded SQL applications, you can explicitly declare and open a cursor for each query result. In ISQL, you do not explicitly open cursors; ALLBASE/SQL maintains the pointer position for you.

For unsorted queries, the tuple buffer is filled with rows of data taken from pages found in the data buffer. Of course, the tuples in the query result are a subset of the content of each data page. In other words, the data buffer contains everything on each data page, but the tuple buffer contains only the columns and rows you have requested. As the cursor moves through the tuple buffer containing the query result, additional rows must be fetched from the data buffer. When data has been fetched from all qualifying pages in the data buffer, additional data pages must be read into the data buffer from disk, and then additional qualifying rows and columns must be read into the tuple buffer. In the case of sorting, the sort output is stored in a temporary table in the `SYSTEM DBEFileSet` before being read into the data buffer.

Use of Locking by Transactions

Transactions obtain **locks** to avoid the possible interference of one transaction with another. This is important when you use PUBLIC or PUBLICROW tables, which can be accessed by many concurrent users of a DBEnvironment. Within the framework of a transaction, the PUBLIC tables that contain the required data for the operation you are performing are locked to regulate access to the data they contain. In addition, individual pages in PUBLIC tables are locked as needed when they are read into the data buffer. In the case of PUBLICROW tables, individual rows are locked as needed before they are read into the tuple buffer. In some cases, the use of a table lock may make the use of individual locks on pages unnecessary. Locks are released on both tables and pages when the transaction that acquired them issues a COMMIT WORK or ROLLBACK WORK statement, or when other conditions are met (described further in the section on "Defining Isolation Levels").

Basics of Locking

The following are the two basic requirements of locking:

- Read operations on data pages must acquire *share* locks before data can be retrieved.
- Write operations on data pages must obtain *exclusive* locks before data is modified.

Lock types are described in more detail in a later section.

When a lock is obtained, the transaction ID (a number), the name of the object locked, and the type of lock acquired are stored in a **lock list** in shared memory. When a user needs a particular lock, a **lock request** is issued, and ALLBASE/SQL checks to see whether the object is already locked by some other transaction. If the lock request cannot be granted, the transaction waits until the other transaction releases the lock. If the request can be granted, the new lock is placed in the lock list. (Compatibility of locks is described in a later section.)

When one transaction is waiting for another transaction to release a lock, and the second transaction is also waiting for the first to release a lock, the transactions are said to be in **deadlock**. If a deadlock occurs, ALLBASE/SQL rolls back one transaction, and this allows the others to obtain the needed lock and continue.

When a transaction ends through a COMMIT WORK or ROLLBACK WORK statement, locks are released; that is, the entries are deleted from the lock list. If the transaction has obtained several different locks, they are all released in a group.

When a transaction ends through an abnormal termination, locks are released by the monitor.

Locks and Queries

During query processing on PUBLIC tables, the cursor is positioned on a row in the query result; by extension, the cursor also points to the underlying data buffer page from which the specific row was derived. Typically, the underlying page to which a cursor points is locked to restrict access to it by other transactions. When a page in the data buffer is locked, another transaction may only access that page in a compatible lock mode. For

example, if someone else is updating a row of user data on page A of a PUBLIC table, your transaction must wait until the update is committed before reading rows from page A into your tuple buffer.

During query processing on PUBLICROW tables, the underlying row to which a cursor points is locked, and the page on which the row resides is also locked (with an intent lock, explained in "Types of Locks", below). Other users can access the same row only in a compatible lock mode, but they can access different rows on the same page in different lock modes. For example, if someone else is updating a row of user data on page A, your transaction must wait until the update is committed before it can read the same row. However, you can read other rows from page A into your tuple buffer and update them.

Locks on System Catalog Pages

In addition to locks on user data, ALLBASE/SQL locks pages of data in the system catalog for the duration of the transaction. Data pages in one or more system tables are locked when any SQL statement is executed.

See the appendix, “Locks Held on the System Catalog By SQL Statements,” in the *ALLBASE/SQL Database Administration Guide* for more information.

Locks on Index Pages

B-tree indexes on PRIVATE and PUBLICREAD user tables are never locked, because concurrency control on the index is already achieved via the table level locks that are always acquired on these tables. B-tree indexes on PUBLIC or PUBLICROW user tables are not locked for read operations, but they are locked with intention exclusive (IX) page locks for write operations. B-tree indexes on PUBLIC and PUBLICROW tables are locked with exclusive (X) page locks only in the following cases:

- When an index row is inserted and the page must be compressed before the insertion. Compression is an attempt to recover non-contiguous space that has become available on an index page.
- When an insert is made and the page must be split into two new pages. Splitting occurs when compression does not result in enough space for inserting the new index row. In such a case, the data from the original page is moved to the two new pages, each of which receives half of the key values from the original page. The new index key is inserted on one of the new pages, and the original page is freed, that is, made available for reuse. A total of three X locks are obtained during this operation: one on the original page, and two on the newly allocated index pages.
- When a delete is made, and an index page becomes empty because the last key on the page was deleted. In this case, ALLBASE/SQL frees the page, which requires an X page lock.

Costs of Locking

The price paid for ensuring the integrity of the database through locking is a reduction in throughput because of lock waits and deadlock and the CPU time used to obtain locks. This price can be high. For example, one way to guarantee that two transactions do not interfere with one another is to allow only one transaction access to a database table at a time. This **serialization** of transactions avoids deadlocks, but it causes such a dramatic

reduction of throughput that it is obviously not desirable in most situations.

Another cost of locking is the use of shared memory resources. Each lock requires the use of some runtime control block space. The more locks used by a transaction, the more memory required for control blocks. This is especially important for PUBLICROW tables, which usually require more locks than PUBLIC tables.

To minimize the costs of locking on PUBLIC and PUBLICROW tables, you should design each transaction in such a way as to lock only as much data as necessary to keep out other transactions that might conflict with your transaction's work. The following sections explain the features of ALLBASE/SQL that you can use to accomplish this.

Defining Isolation Levels between Transactions

Isolation level is the degree to which a transaction is separated from all other concurrent transactions. Four levels are possible, shown here in order from most to least restrictive:

- Repeatable read (RR)--the default
- Cursor stability (CS)
- Read committed (RC)
- Read uncommitted (RU)

In general, you should choose the least restrictive possible isolation level for your needs in order to achieve the most concurrency. You select an isolation level in the `BEGIN WORK` statement, as in the following example:

```
isql=> BEGIN WORK CS;
```

An isolation level can also be specified with either the `SET TRANSACTION` or `SET SESSION` statement.

Repeatable Read (RR)

By default, transactions have the **Repeatable Read (RR)** isolation level, which means that within the transaction, you can access the same data as often as you wish with the certainty that it has not been modified by other transactions. In other words, other transactions are not allowed to modify any data pages that have been read by your transaction until you issue a `COMMIT WORK` or `ROLLBACK WORK` statement. This is the most restrictive level, allowing the least concurrency.

All the examples of transactions shown so far use the RR (repeatable read) isolation level. At the RR level, all locks are held until the transaction ends with a `COMMIT WORK` or `ROLLBACK WORK` statement. This option causes each data row or page read to be locked with a share lock, which forces any other user trying to update the data on the same row or page to wait until the current transaction completes. However, other transactions may read the data on the same row or page. For `PUBLICROW` tables, if you update a row during a transaction, the row receives an exclusive lock, which forces other transactions to wait for both reading or writing that row until your transaction ends. For `PUBLIC` tables, if you update a data page during a transaction, the page receives an exclusive lock, which forces other transactions to wait for both reading or writing until your transaction ends. Repeatable Read should be used if you must read the same data more than once in the current transaction with assurance of seeing the same data on successive reads.

Cursor Stability (CS)

The **Cursor Stability (CS)** isolation level guarantees the stability of the data your cursor points to. However, this isolation level permits other transactions to modify rows of data you have already read, provided you have not updated them and provided they are not still in the tuple buffer. CS also permits other transactions to update rows in the active set which your transaction has not yet read into the tuple buffer. With cursor stability, if you move your cursor and then try to reread data you read earlier in the transaction, that data

may have been modified by another transaction. At the CS level, share locks on data (whether at the row or page level) are released as soon as the associated rows are no longer in the tuple buffer. Exclusive locks are held until the transaction ends with a COMMIT WORK or ROLLBACK WORK statement. The following describes what using CS means:

- No other transactions can modify the row on which the transaction has a cursor positioned.
- A shared lock is kept on the row or page that the cursor is currently pointing to. When the cursor is advanced to the next page of data and nothing has been updated on the previous page, the lock on that previous page is released.
- If an update is done on a data page, the exclusive lock on that page is retained until the transaction ends with a COMMIT WORK or ROLLBACK WORK statement.

Use the CS isolation level for transactions in which you need to scan through large portions of a database to locate rows that need to be updated immediately. CS lets you do this without preventing other transactions from updating data pages that you have already passed by without updating. CS guarantees that a row of data will not be changed between the time you issue the FETCH statement and the time you issue an UPDATE WHERE CURRENT in the same transaction.

NOTE When you use CS for a query that involves a sort operation, such as an ORDER BY, DISTINCT, GROUP BY, or UNION, or when a sort/merge join is used to join tables for the query, the sort may use a temporary table for the query result. In such cases, your cursor actually points to rows in this temporary table, not to rows in the tuple buffer. Therefore, when sorting is involved, the locks held on data pages or rows are released before you manipulate the cursor. In other words, no locks are held at the cursor position for sorted scans at the CS isolation level. If it is important to retain locks in this situation, use the RR isolation level.

If you are updating a row based on the information in a sorted query result, use a simple SELECT statement to verify the continued existence of the data before doing the update operation. In this case, it is good practice to include the TID as part of the original SELECT, and then to use the TID in the WHERE clause of the SELECT that verifies the data.

Read Committed (RC)

With **Read Committed,** you are sure of reading consistent data with a high degree of concurrency. However, you are not guaranteed the ability to *reread* the data your cursor points to, because other transactions can modify that data as soon as it has been read into your application's tuple buffer. Also, you cannot read rows or pages from the data buffer that have been modified by another transaction unless that other transaction has issued a COMMIT WORK statement. At the RC level, share locks on data are released as soon as the data has been read into your buffer. Exclusive locks are held until the transaction ends with a COMMIT WORK or ROLLBACK WORK statement.

The following describes what using RC means:

- You can retrieve only rows that have been committed by some transaction or modified by your own transaction.
- Other transactions *can* write on the page on which the transaction has a cursor positioned, because locks are released as soon as data is read.
- If an update is done on a page, the lock is retained until the transaction ends with a COMMIT WORK or ROLLBACK WORK statement.

Use the RC isolation level for improved concurrency, especially in transactions which include a long duration of time between fetches. When you must update following a FETCH statement using the RC isolation level, use the REFETCH statement first, which obtains and holds locks on the current page, thus letting you verify the continued existence of the data you are interested in.

Read Uncommitted (RU)

The **Read Uncommitted (RU)** isolation level lets you read anything that is in the data buffer, whether or not it has been committed, in addition to pages read in from disk. For example, someone else's transaction might perform an update on a page, which you can then read; then the other transaction issues a ROLLBACK WORK statement which cancels the update. Your transaction has thus seen transitory data which was not committed to the database. At the RU level, *no share locks* are obtained on user data. Exclusive locks obtained during updates are held until the transaction ends with a COMMIT WORK or ROLLBACK WORK statement.

The following describes what using RU means:

- The transaction does not obtain any locks on user data when reading, and therefore may read uncommitted data.
- The transaction does not have to wait on locks on user data, so deadlocks are considerably reduced. However, transactions may still have to wait for system catalog locks to be released.
- If an update is done on a page, the transaction obtains an exclusive lock, which is retained until the transaction ends with a COMMIT WORK or ROLLBACK WORK statement.

RU is ideal for reporting and similar applications where the reading of uncommitted data is not of major importance. If you must update following a FETCH statement using the RU isolation level, use the REFETCH statement first, which obtains and holds the appropriate locks, letting you verify that you are not updating a row based on uncommitted data.

Details of Locking

To promote the greatest concurrency, ALLBASE/SQL supports a variety of granularities and lock types. **Granularity** is the size of the object locked. **Lock type** is the severity of locking provided. **Compatibility** refers to the ability of different transactions to hold locks at the same time on the same object.

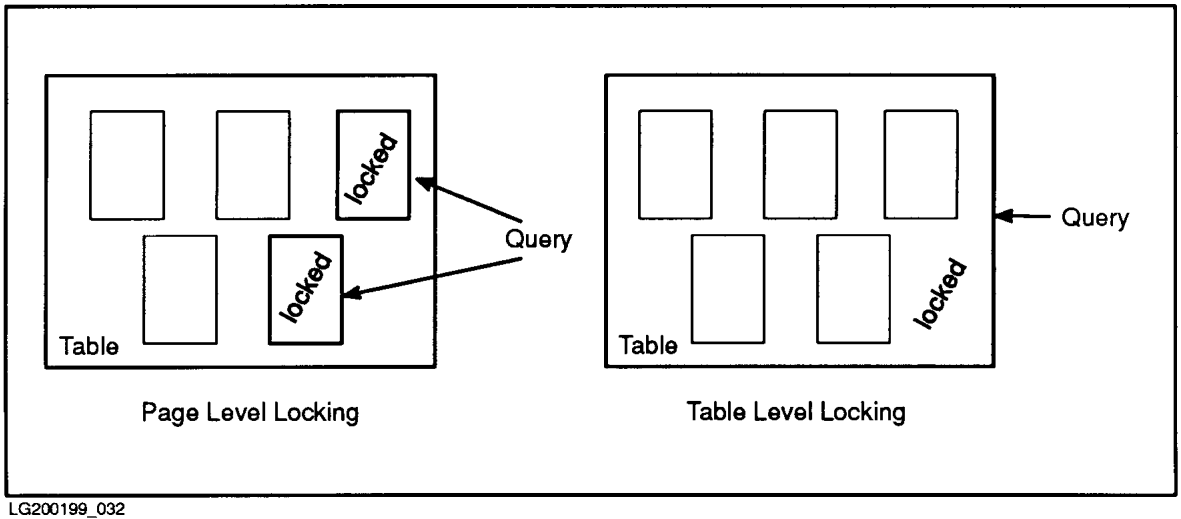
Lock Granularities

The use of different granularities of locking promotes a high level of concurrency. There are three levels of granularity in ALLBASE/SQL:

- Row (tuple) level
- Page level
- Table level

Although some system operations use row level locking internally, system operations acquire page locks by default. User-created tables can be locked at the row, page, or table level, depending on the table type. B-tree and constraint indexes are locked with weak locks at the page level for update operations and are not locked at all on reads. Table, page, and row level locking are illustrated in Figure 5-3. and Figure 5-4. Figure 5-3. portrays a query that accesses two pages of a table.

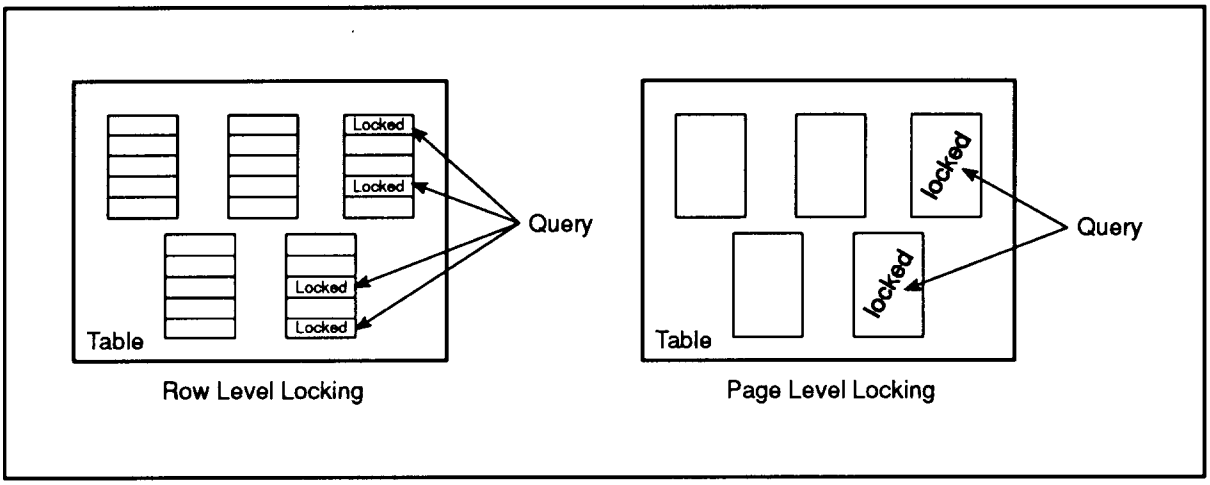
Figure 5-3. Page Versus Table Level Locking



With page level locking, pages containing data scanned for the query are locked. All other pages can be locked by other transactions. With table level locking, the same query locks the table as a whole, whether or not the individual pages are being used for a query. This means that when a table has an exclusive lock on it, no other transaction can obtain any locks on the table or any data page in it until the transaction holding the page lock terminates.

Figure 5-4. also portrays a query that accesses two pages of a table.

Figure 5-4. Row Versus Page Level Locking



LG200199_035

With row level locking, only the rows containing data scanned for the query are locked. All other rows can be locked by other transactions. With page level locking, the same query locks an entire page, even if the page contains row(s) not used by the query.

Table size can affect concurrency at the page level. For example, if a small table occupies only one page, then the effect of a page level lock is the same as locking the entire table. In the case of small tables where frequent access is needed by multiple transactions, row level locking can provide the best concurrency. After issuing an `UPDATE STATISTICS` statement on a table, you can query the `SYSTEM.TABLE` view to determine how many pages it occupies.

Table level locking serializes access to the table, that is, forces transactions with incompatible locks to operate on a table one at a time. This reduces deadlocks by keeping other users from accessing the table until the transaction is committed or otherwise terminated. A small table limits concurrency by its very nature since the probability is high that many users will want to access the limited number of pages or rows. By locking a small table at the table level, you can improve performance by reducing the work of retrying deadlocked transactions. On larger tables, the price of table level locking is higher, since the naturally higher concurrency of the large table is sacrificed to serialization.

Page level locking improves concurrency by allowing multiple users to access different pages in the same table concurrently. Row level locking maximizes concurrency by allowing multiple users to access different rows in the same table at the same time, even on the same page.

Because ALLBASE/SQL uses a buffer system in accessing data from database files, keep

in mind that the system can actually acquire several page or row locks, one at a time, before the data is exposed to the user. In effect, the user's transaction obtains and releases locks on *sets* of pages or rows at a time as it moves through a query result. This is because data from many pages and rows can be required to fill the 12K tuple buffer.

Types of Locks

Locks in ALLBASE/SQL can be classified into the following five types, listed from the lowest to the highest level of severity:

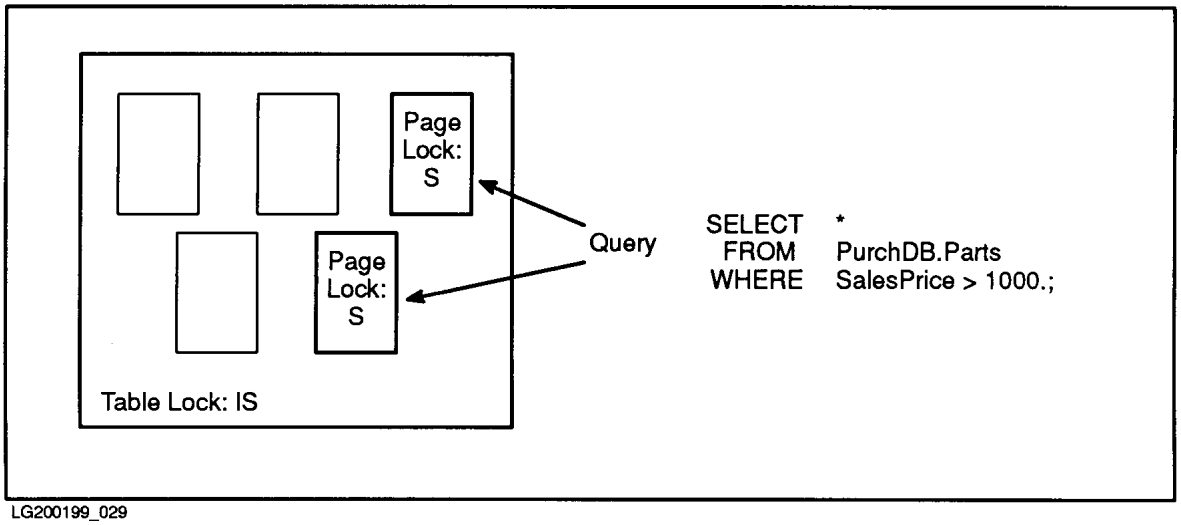
- **Intention Share** (IS): Indicates an intention to read data at a lower level of granularity. An IS lock on a `PUBLIC` table indicates an intention to read a page. An IS lock on a `PUBLICROW` table together with an IS lock on a page indicates an intention to read a row on that page. When a need to read data at a lower level is established, ALLBASE/SQL internally requests an IS lock at the higher level. For example, after an IS table lock has been granted on a `PUBLIC` table, requests are made for S locks on particular pages. In the case of a `PUBLICROW` table, after IS locks have been granted on both table and page, requests are made for S locks on particular rows.
- **Intention Exclusive** (IX): Indicates an intention to update or modify data at a lower level of granularity. An IX lock on a `PUBLIC` table indicates an intention to modify data on a page. An IX lock on a `PUBLICROW` table together with an IX lock on a page indicates an intention to modify a row on that page. When a need to write data at a lower level is established, ALLBASE/SQL internally requests an IX lock at the higher level. For example, after an IX table lock has been granted on a `PUBLIC` table, requests are made for X locks on particular pages. In the case of a `PUBLICROW` table, after IX locks have been granted on both table and page, requests are made for X locks on particular rows.
- **Share** (S): Permits reading by other transactions.
- **Share and Intention Exclusive** (SIX): Indicates a share lock at the current level and an intention to update or modify data at a lower level of granularity. SIX locks are placed on both tables, pages, and rows. When the need to write data at the page or row level is established, and there is also a need to be able to read every page in the table without its being modified by any other transaction, then ALLBASE/SQL internally requests a SIX lock on the table. After an SIX lock has been granted on a PUBLIC table, no additional locks are acquired when a page is read, but an X page lock is acquired when a page is written. After an SIX lock has been granted on a `PUBLICROW` table, no additional locks are acquired when a row is read, but an IX page lock and an X row lock are acquired when a row is written.
- **Exclusive** (X): Prevents any access by other users. An exclusive lock is required whenever data is inserted, deleted, or updated. Because no other user can read this data before the transaction completes, the integrity of the database is not endangered if the changes have to be rolled back, either at the user's request or on recovery after a system failure.

Some of these locks are **intention locks.** Intention locks are obtained at a higher level of granularity whenever a lock is obtained at a lower level. For example, when you obtain a share lock (S) on a page, the table is normally locked with an intention share lock (IS). This is done so that other transactions can quickly tell that a table is being read by someone without the need to determine which specific pages are being read. Suppose another

transaction wishes to lock the table in exclusive mode. The IS lock on the table would prevent the other transaction from locking the table in exclusive mode. Without the use of higher granularity locks, ALLBASE/SQL would have to search all page or row locks to determine whether the exclusive lock request could be granted.

Figure 5-5. shows the use of an intention lock at the table level and share locks on the page level. The example assumes that an index is being used for data access.

Figure 5-5. Locks at Different Granularities



Lock Compatibility

Table 5-1. shows the compatibility of different lock types. A Y (yes) at the intersection of a row and column in the table indicates that two locks are compatible at the same level of granularity; a blank space indicates that they are not compatible.

Table 5-1. Lock Compatibility Matrix

	IS	IX	S	SIX	X
IS	Y	Y	Y	Y	
IX	Y	Y			
S	Y		Y		
SIX	Y				
X					

When two lock requests are compatible, both transactions are allowed to access the table, page, or row concurrently, and the lock on this data object is promoted to or left at the lock mode of higher severity. For example, if transaction 2 wishes to update a page that is already being read by transaction 1, transaction 2 requests an IX lock on the table and an X lock on the page. Transaction 1 has an IS lock on the table, which is compatible with the requested IX, so the lock on the table is promoted to IX. Then, transaction 2 obtains the X lock on the page it needs to update only if transaction 1 is not already reading that same

page. Note that S and X locks on the same page are not compatible.

When locks are not compatible, the second access request must wait until the lock acquired by the first access request is released.

Weak Locks

Intention exclusive locks are called **weak locks** when there is no other lock at a finer level of granularity on the object being locked. This is the case for index pages, which are locked IX when concurrent transactions are updating different rows on the same page. Weak locks, also known as sublocks or concurrent locks, are used to prevent the deletion of an index page by another concurrent transaction. ALLBASE/SQL uses strong locks (exclusive locks) on index pages only for splitting, deleting, or compressing index pages.

What Determines Lock Types

ALLBASE/SQL locks one or more of the following three objects:

- **Tables**. Rows or pages of tables or entire tables are locked when you execute SQL statements referencing them.
- **PCRs**. Pages of PCRs (indexes that support referential constraints) are locked when ALLBASE/SQL updates a key value.
- **Indexes**. Pages of indexes are locked when ALLBASE/SQL updates an index.
- **System tables**. Rows or pages in one or more system tables are locked when you execute any SQL statement. System tables are always locked at the RR level regardless of the transaction isolation level, when they are accessed for execution of an SQL statement. Refer to the appendix "Locks Held on the System Catalog by SQL Statements" in the *ALLBASE/SQL Database Administration Guide* for complete information.

As this summary indicates, locks on *user* data and indexes are obtained at the row level, page level, or at the table level. Although some locking of *system* data is done at the row level, system catalog indexes are *always* locked at the page level.

The locks that are applied to pages and tables are determined by a combination of the following factors:

- Type of SQL statement.
- Locking structure implicit at CREATE TABLE time.
- Use of the LOCK TABLE statement.
- Optimizer's choice of a scan type.
- Choice of isolation level.
- Updatability of cursors or views used to access data.
- Use of sorting.

Type of SQL Statement

Specific SQL statements imply particular kinds of data access. Statements such as SELECT and FETCH, which merely read data, request share locks. INSERT, DELETE, and UPDATE, all of which modify tables, request exclusive locks. In addition, the cursor manipulation statements let you specify an intention to update certain rows of data. When you declare a cursor in a program for updating certain columns, and you then open the cursor, share update (SIX) locks may be obtained.

Data definition statements (CREATE and DROP, ADD and REMOVE) also request exclusive locks, both for the objects being defined, and for the system catalog pages containing descriptions of the objects. During data definition, locking of the system catalog can be extensive. Refer to the appendix "Locks Held on the System Catalog by SQL Statements" in the *ALLBASE/SQL Database Administration Guide* for a complete list of statements and their effects on the system catalog.

When data manipulation or data definition statements update a table that has a B-tree or constraint index defined on it, locks may also be placed on those index pages.

Locking Structure Implicit at CREATE TABLE Time

Table 5-2. shows the general locking structure used for a table depending on the type of locking assigned when the table is created. For clarity, the table shows only the locks obtained for index scans. (Scan type is described in a later section.)

Table 5-2. Locking Behavior Determined by CREATE TABLE Statement

Table Type	Read Locks	Write Locks
PRIVATE (default)	Table Exclusive (X)	Table Exclusive (X)
PUBLICREAD	Table Share (S)	Table Exclusive (X)
PUBLIC	Table Intent Share (IS) Page Share (S)	Table Intent Exclusive (IX) Page Exclusive (X)
PUBLICROW	Table Intent Share (IS) Page Intent Share (IS) Row Share (S)	Table Intent Exclusive (IX) Page Intent Exclusive (IX) Row Exclusive (X)

PUBLICROW and PUBLIC tables allow concurrent users to access the table for both reads and writes but they increase the chances of deadlock, because concurrent transactions can be waiting for each other to release locks. PUBLICROW tables obtain locks at the row level, which affords more concurrency than with PUBLIC tables, at the possible cost of obtaining more locks. PUBLICREAD tables allow only one transaction to write to a table, or they allow multiple transactions to read the table; no readers can access the table while any writing is going on. PRIVATE tables allow only one transaction to read from or write to a table at a time.

If the locking structure of a table does not allow a transaction to access the table, the transaction must wait. In a typical example, if one transaction is reading a PUBLICREAD table, and a second transaction executes a statement to update that table, the second transaction waits until the first transaction executes a COMMIT WORK or ROLLBACK WORK statement.

The implicit locking structure of a table can be changed by using the ALTER TABLE statement.

Use of the LOCK TABLE Statement

The LOCK TABLE statement is another determinant of lock types. With this statement, ALLBASE/SQL explicitly locks a table as a whole, making most page or row locking unnecessary. You can lock tables in SHARE mode, EXCLUSIVE mode, or in SHARE UPDATE mode. With SHARE locking (S locks), other transactions may read pages in the table you have locked but not update them. With EXCLUSIVE locking (X locks), no other transaction may access the locked table until your transaction commits. With share update locking (SIX locks), other transactions may read pages that are not being updated. However, no other transaction can obtain an exclusive lock until your transaction ends with a COMMIT WORK or ROLLBACK WORK statement.

You can upgrade the implicit locking mode of a table to a more severe level by using the LOCK TABLE statement. Thus, you can lock a PUBLIC, PUBLICROW, or PUBLICREAD table in EXCLUSIVE mode. However, you cannot downgrade the implicit locking mode. If you attempt to lock a PRIVATE table in SHARE mode, the LOCK TABLE statement has no effect.

Use the LOCK TABLE statement to reduce the following:

- The overhead of obtaining and maintaining locks
- The potential for deadlock

Choice of a Scan Type

Another factor that determines the kind of locking in a data access transaction is the type of scan used to process a query. There are four types of scan:

- **Serial scan**
- **Index scan**
- **Hash scan**
- **TID scan**

A sequential scan (also known as a serial scan) is one in which ALLBASE/SQL begins at

the first page of a table and reads each page, looking for rows that qualify for the query result, until it arrives at the end of the table. An index scan looks up the page locations of those rows that qualify for the query result in an index which you have separately created. A hash scan accesses an individual row by calculating the row's primary page location from a value supplied in the query's predicate. A TID scan obtains a specific row by obtaining its page number from the TID (tuple ID) directly. A hash scan accesses an individual row by calculating the row's primary page location from a value supplied by the query's predicate.

When a sequential scan is used to access a table, the data is being read at the table level. Depending on the isolation level of a transaction (described in the next section), a sequential scan either locks the whole table or else locks each page of a table in share mode (each row, in the case of a PUBLICROW table) in turn until it finds the row it is seeking.

When an index scan is used to access a table, the data is being read at the page level if the table is PUBLIC or at the row level if the table is PUBLICROW. An index scan has to read index pages, but no locks are acquired; a transaction only needs to lock the data page or row pointed to by the index. Thus, an index scan that retrieves only a few rows from a large PUBLIC table will obtain locks on fewer data pages than a sequential scan on the same table. (Index pages are locked with IX locks only when an index is updated.) A TID scan locks only the page or row pointed to by the TID. A hash scan locks only the data page containing the hash key, possibly with some overflow pages. Hashing is not possible with PUBLICROW tables.

By default, the choice of a plan of access to the data is made by the ALLBASE/SQL optimizer. You can override the access plan chosen by the optimizer with the SETOPT statement.

As a rule of thumb, you can assume that the optimizer chooses a sequential scan when the query needs to read a large proportion of the pages in a table. Similarly, the optimizer often chooses an existing index when a small number of rows (or only a single row) is to be retrieved, and the index was created on the columns referred to in the WHERE clause of the query. When you use a TID function, you can assume the optimizer will choose a TID scan. To display the access plan chosen by the optimizer, use the SQL GENPLAN statement, specifying the query of interest. Then perform a query on the SYSTEM.PLAN view in the system catalog to display the optimizer's choices. For more information, refer to the section "Using GENPLAN to Display the Access Plan" in Chapter 3 , "SQL Queries."

NOTE If you are reading a large table, and if you do not expect it to be updated by anyone while your transaction is running, you can avoid excessive overhead in shared memory from locks obtained on each page by using the LOCK TABLE statement in SHARE mode. This makes it unnecessary for ALLBASE/SQL to lock individual pages or rows.

Choice of Isolation Level

One more factor that determines the kinds of locks obtained on data objects is the isolation level of the transaction. A higher degree of isolation means less concurrency in operations involving PUBLIC and PUBLICROW tables. You can select the isolation level used in your transactions to maximize concurrency for the type of operation you are performing and to

minimize the chance of deadlocks.

The kind of lock obtained at different isolation levels depends on the other factors that determine locks--scan type, kind of SQL statement, and implicit table type. A simplified summary of locks obtained on PUBLIC tables and their indexes appears in Table 5-3.. Hash and TID scans are omitted.

Table 5-3. Locks Obtained on PUBLIC Tables with Different Isolation Levels

Isolation Level and Scan Type	Read Operations (SELECT, FETCH)		Read for Update[a]		Write Operations (UPDATE, INSERT, DELETE)	
	Table	Page	Table	Page	Table	Page
RR Sequential	S	-	SIX	-	SIX	X
RR Index	IS	S	IX	SIX	IX	X
CS Sequential	IS	S[b]	IX	SIX	IX	X
CS Index	IS	S	IX	SIX	IX	X
RC Sequential	IS	S[c]	IX	SIX	IX	X
RC Index	IS	S	IX	SIX	IX	X
RU Sequential	None	None	IX	SIX	IX	X
RU Index	None	None	IX	SIX	IX	X

a. Opening a cursor that was declared FOR UPDATE (RR and CS), or using REFETCH (RC and RU).
b. Lock released at the end of the next read.
c. Lock released at the end of the current read.

A simplified summary of locks obtained on PUBLICROW tables appears in Table 5-4. Hash and TID scans are omitted.

Table 5-4. Locks Obtained on PUBLICROW Tables with Different Isolation Levels

Isolation Level and Scan Type	Read Operations (SELECT, FETCH)			Read for Update[a]			Write Operations (UPDATE, INSERT, DELETE)		
	Table	Page	Row	Table	Page	Row	Table	Page	Row
RR Sequential	S	-	-	SIX	-	-	SIX	IX	X
RR Index	IS	IS	S	IX	IX	SIX	IX	IX	X[b]
CS Sequential	IS	IS[c]	Sc	IX	IXc	SIXc	IX	IX	X
CS Index	IS	ISc	Sc	IX	IXc	SIXc	IX	IX	Xb
RC Sequential	IS	IS[d]	Sd	IX	IXd	SIX	IX	IX	X
RC Index	IS	ISd	Sd	IX	IX	SIX	IX	IX	Xb
RU Sequential	None	None	None	IX	IX	SIX	IX	IX	X
RU Index	None	None	None	IX	IX	SIX	IX	IX	Xb

a. Opening a cursor that was declared FOR UPDATE (RR and CS), or using REFETCH (RC and RU).
b. Next higher key's data row is locked for an insert or delete, and the next two higher key's data rows are locked for an update.
c. Lock released at the end of the next read.
d. Lock released at the end of the current read.

NOTE ALLBASE/SQL locks system catalog *pages* at the *RR* isolation level when they are accessed or modified on behalf of an SQL statement. Refer to the appendix "Locks Held on the System Catalog by SQL Statements" in the *ALLBASE/SQL Database Administration Guide* for a list of locks acquired for each SQL statement.

Neighbor Locking

Neighbor locking is a way indexes are maintained. More than one object is locked within a Publicrow. SQLMon is the best tool to get the kind of locks held on SQL objects.

During an index scan, "weak" (IS, IX) locks are placed on index and data pages. A tuple (page) lock will be placed on the qualifying tuple(s). In order to insure RR (Repeatable Read), an additional tuple (page) lock is placed on the data tuple corresponding to the higher key next to the qualifying key. During a RR/CS/RC index scan, the qualifying data tuple are locked in S. During inserts and deletes, the higher key's tuple is locked in X for uniqueness and to insure RR for readers. Of course, the updated tuple is locked in X also. During an update where the key is updated, we end up with two higher key locks because

the update corresponds to an index delete followed by an index insert. What should you lock if there is no higher? Lock an imaginary tuple which has the highest possible key. Note that locks are placed at the tuple level for `PUBLICROW` or at the page level for `PUBLIC` tables.

Updatability of Cursors or Views

When a transaction uses cursors or views to access and manipulate data, the kinds of locks obtained depend partly on whether the cursors or views are updatable according to the rules presented under "Updatability of Queries" in Chapter 3 , “SQL Queries.” Table 5-3. shows the locks obtained on updatable views and on updatable cursors declared FOR UPDATE; they are listed in the "Read for Update" column in the table. In general, SIX, IX, and X locks will not be used unless the query that underlies the view or cursor is updatable.

Use of Sorting

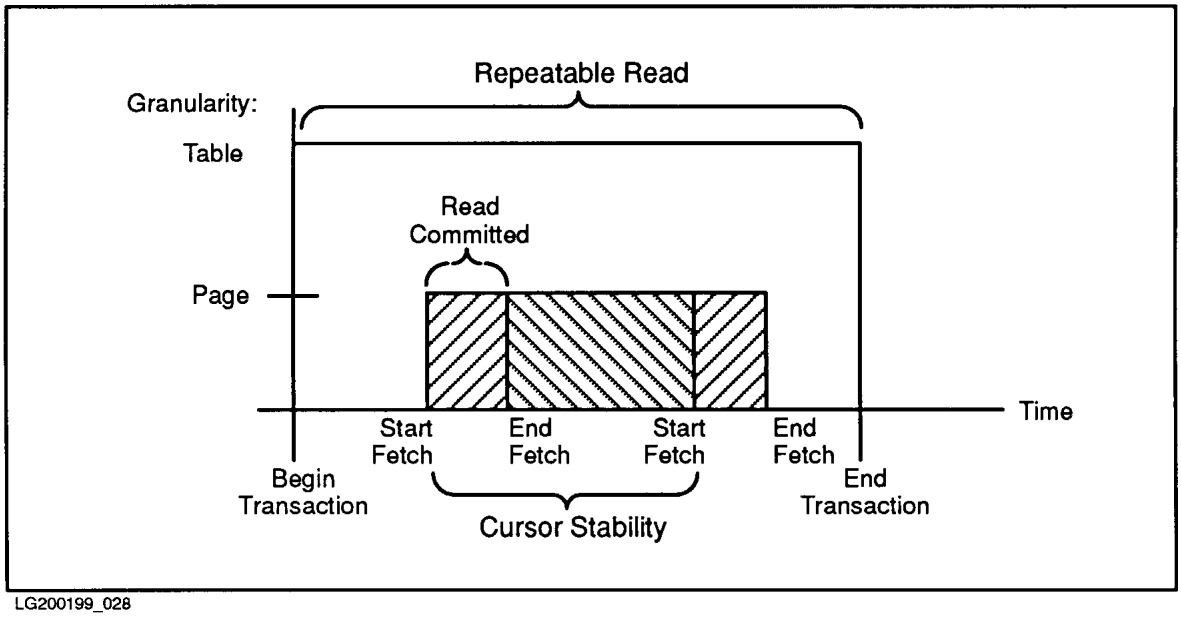
If a query involves a sort operation, locks are maintained only if the transaction is at the RR isolation level. When there is an `ORDER BY`, a `GROUP BY, UNION`, or `DISTINCT` clause in a query, or if the optimizer decides to use the sort/merge join method for joins or nested queries, the data in the tables is sorted and copied to a temporary table. The user's cursor is really defined on this temporary table, which does not require any locking since it is private to the user. Locks on the original tables underlying the view or cursor are retained only if the transaction was started at the RR isolation level. Locks obtained at the CS or RC level are released; locks are not obtained at all at the RU level.

Scope and Duration of Locks

In general, the length of a transaction affects concurrency. Long transactions hold locks longer, which increases the chances that another transaction is waiting for a lock. Short transactions are "in and out" quickly, which means they are less likely to interfere with other transactions.

The isolation level determines what kinds of locks are obtained in particular circumstances, and also how long these locks are held. Great differences can be found between isolation levels in the duration of locks. For example, a sequential scan that obtains share locks at the RR level holds them while the entire table is read, making updates impossible by others during that time. At the RU level, other users can update the table throughout an entire scan by another reader. Figure 5-6. shows the relative scope and duration of share locks obtained for a sequential scan by the RR, CS, and RC isolation levels on `PUBLIC` and `PUBLICROW` tables. RU is not shown, because it does not obtain any share locks on user data.

Figure 5-6. Scope and Duration of Share Locks for Different Isolation Levels



Examples of Obtaining and Releasing Locks

The following sections present a few scenarios that show how locks are obtained and released within concurrent transactions.

Simple Example of Concurrency Control through Locking

The following scenario illustrates in a simple way how locks are obtained and released. It is based on the sample DBEnvironment PartsDBE, which is fully described in Appendix C. Try this example yourself on a system that has several terminals available in physical proximity to one another, and observe the results:

- Four users each issue the following CONNECT statement (assume they are connecting from a different group and account than the one containing PartsDBE):

```
isql=> CONNECT TO 'PartsDBE.SomeGrp.SomeAcct';
```

- User 1 issues the following query (transaction 1):

```
isql=> SELECT SALESPRICE FROM PurchDB.Parts
> WHERE PartNumber = '1123-P-01';
```

  At this point, transaction 1 obtains a share lock on page A.

- User 2 issues the following UPDATE statement (transaction 2):

```
isql=> UPDATE PurchDB.Parts SET SalesPrice = 600.
> WHERE PartNumber = '1123-P-01';
```

  Transaction 2, executing concurrently, needs an exclusive lock on page A. Transaction 2 waits.

- Users 3 and 4 each issue the following query, independently (transactions 3 and 4):

```
isql=> SELECT * FROM PurchDB.Parts;
```

  Transactions 3 and 4, executing concurrently, each need a share lock on page A. Transactions 3 and 4 wait, because of an upcoming exclusive lock request.

- User 1 issues the following statement:

```
isql=> COMMIT WORK;
```

- Transaction 1 terminates, so transaction 2 obtains its exclusive lock on page A. Transactions 3 and 4 still wait.
- User 2 issues the following statement:

```
isql=> COMMIT WORK;
```

- Transaction 2 terminates, so transactions 3 and 4 both obtain share locks on page A.

This sequence is illustrated in Figure 5-7., Figure 5-8., and Figure 5-9..

Figure 5-7. Lock Requests 1: Waiting for Exclusive Lock

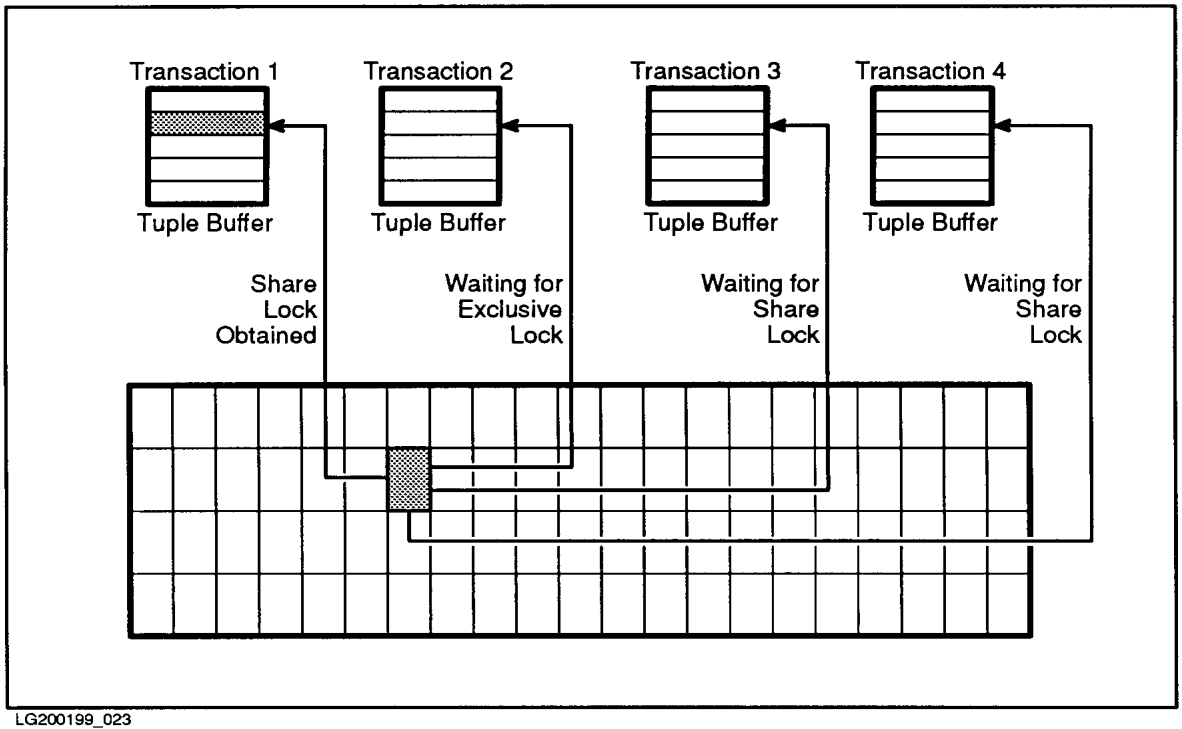


Figure 5-8. Lock Requests 2: Waiting for Share Locks

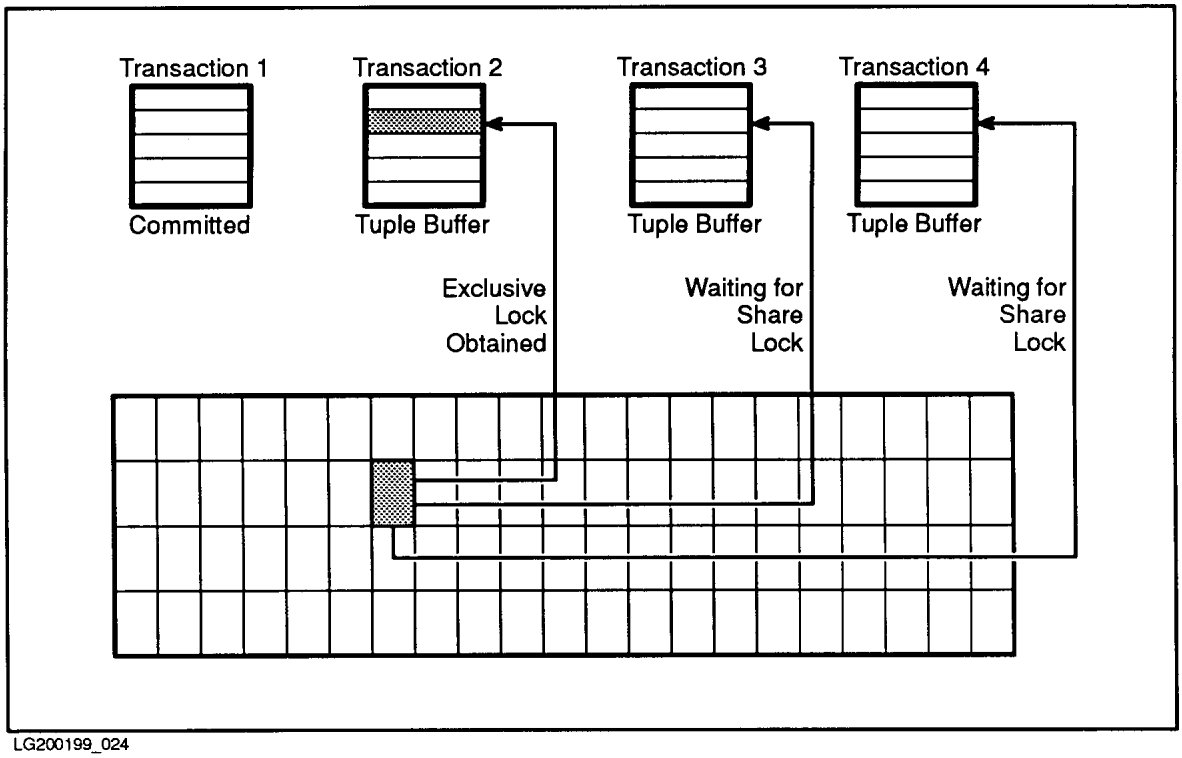
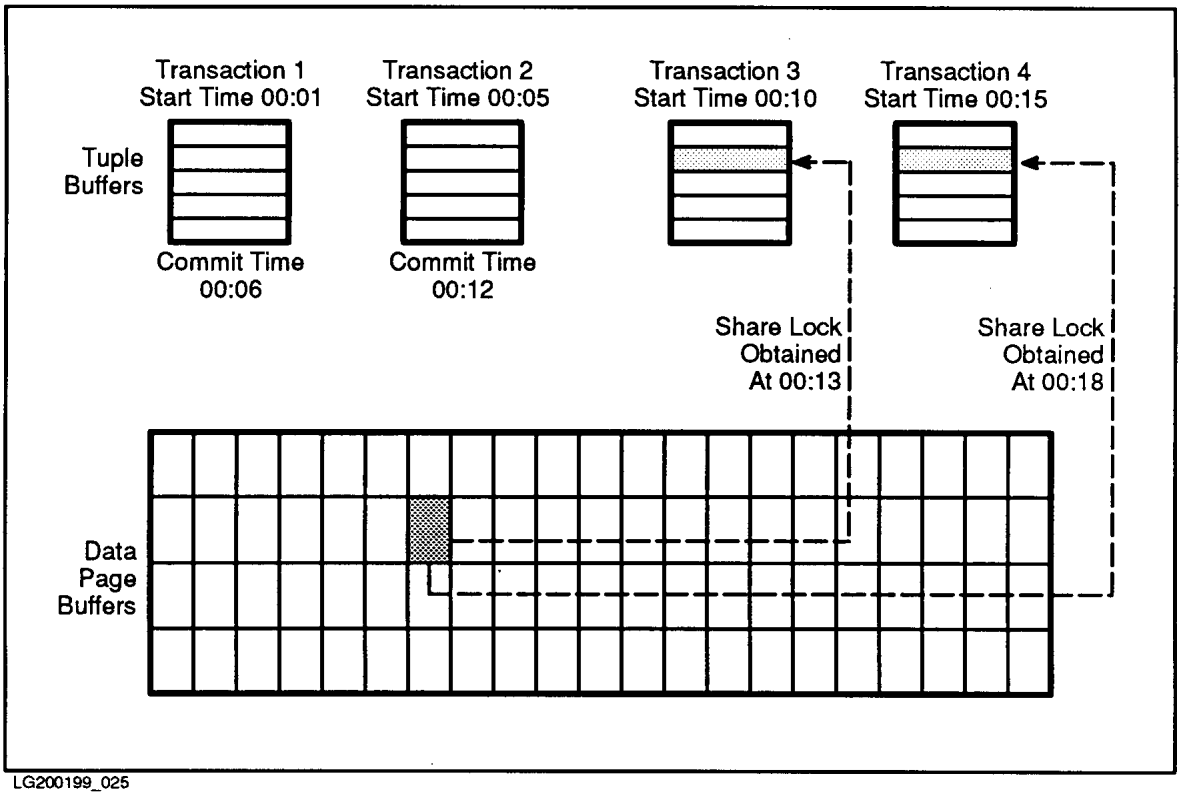


Figure 5-9. Lock Requests 3: Share Locks Granted



Sample Transactions Using Isolation Levels

The following sections show typical situations in which different isolation levels affect the behavior of your transactions when using the sample DBEnvironment PartsDBE.

Example of Repeatable Read

The following scenario illustrates the operation of the RR isolation level:

1. Two users each issue the following CONNECT statement (assume they are connecting from a different directory than the one containing PartsDBE):

   ```
   isql=> CONNECT TO 'PartsDBE.SomeGrp.SomeAcct';
   ```

2. User 1 then issues a query (transaction 1) as follows:

   ```
   isql=> SELECT * FROM PurchDB.Vendors;
   ```

   This implicitly issues a BEGIN WORK statement at the RR isolation level, and obtains a share lock (S) on the Vendors table, because the scan is a sequential one, reading the entire table. User 1 sees the query result in the ISQL browser, and exits the browser, but does *not* issue a COMMIT WORK statement.

3. User 2 then issues the following statement (which starts transaction 2 at the RR isolation level):

   ```
   isql=> UPDATE PurchDB.Vendors
   ```

```
> SET ContactName = 'Harry Jones'
> WHERE VendorNumber = 9001;
```

Transaction 2 now must wait for an IX lock on the Vendors table because an IX lock is not compatible with the S lock already held by transaction 1. Transaction 2 also must obtain an X lock on the page containing data for vendor 9001.

4. User 1 now issues the following statement:

```
isql=> COMMIT WORK;
```

5. Transaction 2 can now complete the update, because transaction 1 no longer holds the S lock on the Vendors table. This makes it possible for transaction 2 to obtain the IX lock on the Vendors table and the X lock on the page containing data for 9001.

Example of Cursor Stability

The following scenario illustrates the operation of the CS isolation level:

1. Two users each issue the following CONNECT statement (assume they are connecting from a different group and account than the one containing PartsDBE):

```
isql=> CONNECT TO 'PartsDBE.SomeGrp.SomeAcct';
```

2. User 1 then sets the CS isolation level for transaction 1 and issues the following query:

```
isql=> BEGIN WORK CS;
isql=> SELECT * FROM PurchDB.Vendors;
```

User 1 sees the query result in the ISQL browser, but does *not* exit the browser.

3. User 2 then issues the following statement (this statement implicitly starts transaction 2 at the RR isolation level):

```
isql=> UPDATE PurchDB.Vendors
> SET ContactName = 'Harry Jones'
> WHERE VendorNumber = 9001;
```

Transaction 2 now waits for an exclusive lock on a page in the Vendors table, because transaction 1 still has a cursor positioned on that page.

4. User 1 now exits from the ISQL browser, but does *not* issue a COMMIT WORK statement.

5. Transaction 2 can now complete the update, because transaction 1's cursor is no longer positioned on the page that transaction 2 wishes to update.

6. Transaction 1 now attempts to issue the same query again, using a REDO statement:

```
isql=> REDO;
SELECT * FROM PurchDB.Vendors;
```

Now transaction 1 waits, because transaction 2 has obtained an exclusive lock on the table.

7. Transaction 2 issues the following statement:

```
isql=> COMMIT WORK;
```

The query result for transaction 1 now appears in the ISQL browser again, this time with the changed row in the query result.

Example of Read Committed

The following scenario illustrates the operation of the RC isolation level in concurrent transactions in the sample DBEnvironment PartsDBE. Most of the details are the same as for the CS example just presented:

1. Two users each issue the following CONNECT statement (assume they are connecting from a different group and account than the one containing PartsDBE):

```
isql=> CONNECT TO 'PartsDBE.SomeGrp.SomeAcct';
```

2. User 1 then sets the RC isolation level for transaction 1 and issues the following query:

```
isql=> BEGIN WORK RC;
isql=> SELECT * FROM PurchDB.Vendors;
```

   User 1 sees the query result in the ISQL browser, but does *not* exit the browser.

3. User 2 then issues the following statement (this statement implicitly starts transaction 2 at the RR isolation level):

```
isql=> UPDATE PurchDB.Vendors
> SET ContactName = 'Harry Jones'
> WHERE VendorNumber = 9001;
```

   Transaction 2 is able to perform the update, because the locks on pages that were obtained by transaction 1's cursor were released as soon as the data was placed in transaction 1's tuple buffer. Notice the difference between RC and CS.

Example of Read Uncommitted

The following scenario illustrates the operation of the RU isolation level:

1. Two users each issue the following CONNECT statement (assume they are connecting from a different group and account than the one containing PartsDBE):

```
isql=> CONNECT TO 'PartsDBE.SomeGrp.SomeAcct';
```

2. User 1 issues the following update:

```
isql=> UPDATE PurchDB.Vendors SET ContactName = 'Rogers, Joan'
> WHERE VendorNumber = 9005;
```

3. User 2 then sets the RU isolation level for transaction 2 and issues a query:

```
isql=> BEGIN WORK RU;
isql=> SELECT * FROM PurchDB.Vendors WHERE VendorNumber = 9005;
```

   User 2 sees the desired row in the ISQL browser, where the contact name for vendor 9005 is *Rogers, Joan*, even though user 1 has not issued a COMMIT WORK statement. In other words, user 2 has read uncommitted data.

Resolving Conflicts among Concurrent Transactions

Several kinds of conflict can occur between transactions that are contending for access to the same data. The following three are typical cases:

- One transaction has locked an object that another transaction needs and is in a wait state.
- Two transactions each need an object the other transaction has locked in the same DBEnvironment and are both in a wait state.
- Two transactions each need an object the other transaction has locked in another DBEnvironment and are both in a wait state.

The first conflict results in a **lock wait**, which simply means that the second transaction must wait until the first transaction releases the lock. The second conflict is known as conventional deadlock, which is automatically resolved by ALLBASE/SQL. The third conflict is an **undetectable deadlock**, which cannot be automatically resolved.

Lock Waits

When a transaction is waiting for a lock, the application pauses until the lock can be acquired. When a transaction is in a wait state, some other transaction already has a lock on the row, page, or table that is needed. When the transaction that is holding a lock on the requested row, page, or table releases its lock through a `COMMIT WORK` or `ROLLBACK WORK` statement, the waiting transaction can then acquire a new lock and proceed.

The amount of time an application waits for a lock depends on the **timeout value**. A timeout value is the amount of time a user waits if a requested database resource is unavailable. If an application times out while waiting for a lock, an error occurs and the transaction is rolled back. See the `SET USER TIMEOUT` statement in the "SQL Statements" chapter of this manual for more information.

The larger the number of lock waits, the slower the performance of the DBEnvironment as a whole. You can observe the lock waits at any given moment in the DBEnvironment by issuing the following query:

```
isql=> SELECT * FROM SYSTEM.CALL WHERE STATUS = 'WAITING ON LOCK';
```

The use of isolation levels less severe than Repeatable Read can improve concurrency by reducing lock waits. For example, reporting applications that do not depend on perfect consistency can use the Read Uncommitted level, while applications that scan an entire table to update just a few rows can use Read Committed with `REFETCH` or Read Uncommitted with `REFETCH` for the greatest concurrency. Applications that intend to update a larger number of rows can use Cursor Stability.

You can set the amount of time a transaction will wait for a lock by using the `SET USER TIMEOUT` statement, or by setting a default timeout value using the `ALTDBE` command in SQLUtil. If no timeout value is set as a default, the transaction will wait until the resource is released. Consult your database administrator about default timeout values.

Deadlocks

The second kind of conflict is known as a deadlock between two transactions. This happens when two transactions both need data or indexes that the other already has locked. Deadlocks involving system catalog pages are also possible. ALLBASE/SQL detects and resolves deadlocks when they occur. If different priority numbers are assigned to the transactions in the BEGIN WORK statement, the transaction with the larger priority number is rolled back. If no priorities are assigned, the more recent transaction is rolled back.

ALLBASE/SQL resolves deadlocks between two transactions at a time. Therefore, if more than two transactions are deadlocked at one time, the transaction aborted may not be the transaction with the largest priority number or the newest transaction among all transactions deadlocked.

By default, the action taken to resolve a deadlock is to roll back one of the transactions. However, it is also possible to set the deadlock action for a transaction to roll back the current command instead of the entire transaction by using the SET SESSION or SET TRANSACTION statements.

Table Type and Deadlock

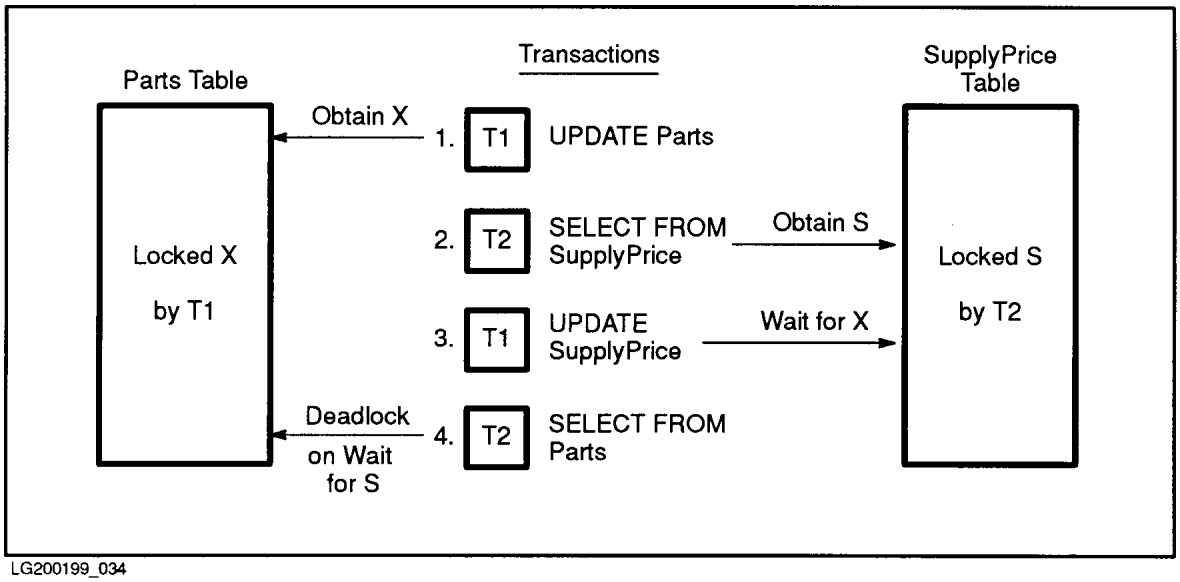
Specific table types are likely to incur particular types of deadlock. Two transactions can deadlock on the same PUBLIC or PUBLICROW table when the transactions attempt to access the same page or row. The larger the table, the less likely it is that two transactions will need to access the same page or row, so deadlock is reduced. If the table is small, there is less chance of deadlock when it is defined PUBLICROW rather than PUBLIC.

The following scenario illustrates the development of a deadlock involving two fairly large PUBLIC tables with indexes in the sample DBEnvironment PartsDBE. Assume that both transactions are at the RR isolation level.

```
Transaction 1: UPDATE PurchDB.Parts SET            Obtains IX lock on table,
               SalesPrice = 1.2*SalesPrice;        X on each page.
Transaction 2: SELECT * FROM PurchDB.SupplyPrice;  Obtains S lock on table.
Transaction 1: UPDATE PurchDB.SupplyPrice SET      Waits for IX on table
               UnitPrice = 1.2*UnitPrice;
Transaction 2:  SELECT * FROM PurchDB.Parts;    Deadlock.
```

This sequence results in a deadlock which causes ALLBASE/SQL to choose a transaction to roll back. In the example, since no priorities are assigned, ALLBASE/SQL rolls back both of user 2's queries and displays an error message. User 1's second update then completes. Figure 5-10. shows the deadlock condition that results from the previous example.

Figure 5-10. Deadlock



The use of PRIVATE tables ensures there will be no deadlock on the same table, because access to the table is serialized. However, deadlock across two or more tables is common with PUBLICREAD and PRIVATE tables that are accessed by different transactions in different order. The following example shows a deadlock involving a PRIVATE table:

```
Transaction 1: SELECT * FROM TABLEA;       Obtains X lock on table.
Transaction 2: SELECT * FROM TABLEB;       Obtains X lock on table.
Transaction 1: SELECT * FROM TABLEB;       Waits for X on table.
Transaction 2:   SELECT * FROM TABLEA;   Deadlock.
```

A common deadlock scenario for PUBLICREAD tables is to do a SELECT, thus obtaining a table level share lock, and then an UPDATE, which must upgrade the lock to exclusive:

```
Transaction 1: SELECT * FROM TABLEA;       Obtains S lock on table.
Transaction 2: SELECT * FROM TABLEA;       Obtains S lock on table.
Transaction 1: UPDATE TABLEA;              Waits to upgrade to X on table.
Transaction 2:   UPDATE TABLEA;          Deadlock.
```

The need to upgrade frequently results in deadlock.

Table Size and Deadlock

The size of a table is another factor affecting its susceptibility to deadlock. If the table is small, it is highly probable that several users may need the same pages, so deadlocks may be relatively frequent when page level locking is used. The probability of collision is highest when the table is small and its rows are also small, with many stored on one page. If the table is large, it is relatively unlikely that multiple users will want the same pages at the same time, so page level locking should cause relatively few deadlocks.

Avoiding Deadlock

The tradeoff between deadlock and throughput is one of the central issues in concurrency control. It is important to minimize the number of deadlocks while permitting the greatest possible concurrent access to database tables.

Avoiding Deadlock by Using the Same Order of Execution

To avoid deadlock among multiple tables, be sure to have all transactions access them in the same order. This can often be done by modifying programs to use the same algorithms to access data in the same order (for example, first update table 1, then table 2), rather than accessing data in random order. This strategy cannot always be followed, but when it can be used, processes will wait their turn to use a particular data object rather than deadlocking.

Avoiding Deadlock by Reading for Update

You can avoid deadlocks that stem from upgrading locks by designing transactions that use SIX locks, which have the effect of serializing updates on a table while permitting concurrent reads. To employ SIX locks, read the table with a cursor that includes a FOR UPDATE clause. You can also obtain SIX locks by using the LOCK TABLE statement, specifying the SHARE UPDATE option.

Avoiding Deadlock by Using the LOCK TABLE Statement

Locking at the table level should reduce deadlocks when all or most pages in a PUBLIC table (rows in a PUBLICROW table) are accessed in a query. Locking the table in share update mode obtains SIX locks on the table and its pages (or rows) when you are reading data with the intention of updating some data.

Avoiding Deadlock on Single Tables by Using PUBLICREAD and PRIVATE

The use of PUBLICREAD and PRIVATE tables decreases the chance of encountering a deadlock by forcing serialization of updates within a single table, that is, requiring one update transaction to be committed before another can obtain any locks on the same table. Obviously, this reduces concurrency during update operations. You can also use the LOCK TABLE statement for transactions on PUBLICREAD tables that read data prior to updating it.

Avoiding Deadlock by Using the KEEP CURSOR Option

In applications that declare cursors explicitly, you can use the KEEP CURSOR option in the OPEN statement to release exclusive locks as quickly as possible. When you use the KEEP CURSOR option for a cursor you explicitly open in a program, you can use the COMMIT WORK statement to end the transaction and release locks without losing the cursor's position. Furthermore, you can either retain or release the locks on the page or row pointed to by the current cursor position. When you use the KEEP CURSOR option, your transaction holds individual exclusive locks only for a very short time. Thus, the chance of deadlock is reduced, and throughput is improved dramatically. For details, refer to the chapter entitled "Processing with Cursors" in the ALLBASE/SQL application programming guide for the language of your choice.

Undetectable Deadlock

Applications that connect to multiple DBEnvironments may encounter deadlocks that cannot be detected and resolved by ALLBASE/SQL. An example follows:

```
Transaction 1:   SET CONNECTION 'DBE1';
                 UPDATE TABLEA SET COL1 = 5;   Obtains X table lock.
Transaction 2:   SET CONNECTION 'DBE2';
                 UPDATE TABLEB SET COL1 = 7;   Obtains X table lock.
Transaction 1:   SET CONNECTION 'DBE2';
                 SELECT * FROM TABLEB;         Waits.
Transaction 2:   SET CONNECTION 'DBE1';
                SELECT * FROM TABLEA;        Waits--Undetectable Deadlock.
```

This kind of deadlock is called undetectable because ALLBASE/SQL can only detect a deadlock within a single DBEnvironment session. It is your responsibility to coordinate your system's use of distributed transactions so as to prevent undetectable deadlock. You can enable ALLBASE/SQL to identify and roll back what probably are undetectable deadlocks by setting appropriate user timeout values for each DBEnvironment connection. For more information refer to "Using Multiple Connections and Transactions with Timeouts" in Chapter 2 , “Using ALLBASE/SQL.”

A similar condition known as an *undetectable wait* state can also arise when you are using multi-connect functionality. An undetectable wait occurs when you connect more than once to the same DBEnvironment from the same application in multi-transaction mode and attempt to obtain resources held by your other connection. For example:

```
CONNECT TO 'DBE1' AS 'CONNECT1';a
CONNECT TO 'DBE1' AS 'CONNECT2';
SET CONNECTION 'CONNECT1';
UPDATE TABLEA SET COL1 = 5;                  Obtains X table lock.
SET CONNECTION 'CONNECT2';
UPDATE TABLEA SET COL1 = 7;                Waits--Undetectable wait.
```

In this instance, you are waiting on your own resources. To avoid situations like this, *be sure to set user timeout values when you use multi-connect functionality*.

Monitoring Locking with SQLMON

SQLMON is an online diagnostic tool that monitors the activity of your DBEnvironment. In addition to providing information on file capacity, I/O, logging, tables, and indexes, SQLMON displays information on the locks currently held in your DBEnvironment. SQLMON is fully documented in the *ALLBASE/SQL Performance and Monitoring Guidelines*.

MONITOR Authority

Users with DBA authority or who are granted MONITOR authority can run SQLMON. Use the GRANT MONITOR command to allow users to run SQLMON. Use the REVOKE MONITOR command to revoke the authority. SYSTEM.SPECAUTH and CATALOG.SPECAUTH identify users with MONITOR authority.

Monitoring Tasks

Table 5-5 summarizes the monitoring tasks related to locking you can perform with SQLMON:

Table 5-5. SQLMON Monitoring Tasks

Task	Screens	Fields
Determining Size of Runtime Control Block	Overview	RUNTIME CB % Used Pages Max Pages
Monitoring DBEnvironment Lock Activity	Load	LOCK REQTS LOCK WAITS LOCK WAIT %
Comparing Number of Locks by Table	Lock TabSummary	OWNER.TABLE G TOTAL LOCKS
Comparing Number of Locks by Session	Lock Memory	TABLE PAGE ROW TOTAL MAXTOTAL
Identifying Locks on a Table or Referential Constraint (PCR)	Lock	OWNER.TABLE[/CONSTRAINT] G PAGE/ROW ID LOCK QUEUE
Determining Number of Sessions that are Accessing a Particular Lock	Lock	LOCK QUEUE

Table 5-5. SQLMON Monitoring Tasks

Task	Screens	Fields
Determining Number of Transactions that are Waiting for Locks	Overview Load	IMPEDE XACT
Identifying Locks for which Sessions are Waiting	Lock	all fields
Identifying Sessions that have Obtained a Particular Lock	Lock Object	GWC MOD PIN
Identifying Sessions that are Waiting to Obtain (or to Convert) a Particular Lock	Lock Object	GWC MOD NEW PIN
Identifying Lock Activity for a Particular Session	Lock Session	all fields
Identifying Locks Obtained by a Particular Session that are Causing Other Sessions to Wait	Lock Impede	all fields
Detecting Deadlocks	Load Load Session Load Program	DEADLOCKS
Resolving Deadlocks	Lock Lock Object Lock Impede	all fields

6 Names

This chapter contains general rules for names used in ALLBASE/SQL commands. Syntactically, names used in ALLBASE/SQL commands fall into several categories. This chapter includes a section for each category as follows:

- Basic Names
- Native Language Object Names
- DBEUserIDs
- Owner Names
- Authorization Names
- Compound Identifiers
- Host Variable Names
- Local Variable Names
- Parameter Names
- DBEnvironment and DBECon File Names
- DBEFile and Log File Identifiers
- TempSpace Names
- Special Names

Some programming languages define reserved words that cannot be defined as names by the user.

Basic Names

The syntax rules in this chapter apply to most SQL names. Names that are required to conform to the following rules can be classified as **basic names**:

- A basic name can be up to 20 bytes in length.
- A name can be made up of any combination of letters (A to Z), decimal digits (0 to 9), $, #, @, or underscore (_). However, the first character cannot be an underscore or a decimal digit.
- Lowercase letters (a to z) are automatically changed to the corresponding uppercase letters (A to Z) unless enclosed in double quotation marks.
- You can use any combination of characters in a basic name if you enclose it in double quotation marks. However, note that if you define a name using double quotes, you must use double quotes when you use the name later. Moreover, if the context in which you are using the name would itself require the use of double quotes, you must precede each of the quotes around the basic name with a backslash, as in the following example:

  ```
  UNLOAD TO EXTERNAL EParts FROM
  "SELECT * FROM \"PurchDB\".PARTS";
  ```

  In addition, application programs must be capable of distinguishing double-quoted names. To prevent any possible conflict, minimize the use of double-quoted basic names.

The following are classified as basic names:

```
Class names                Log file names
Column names               Module names
Constraint names           Procedure names
Cursor names               Rule names
DBEFile names              Table names
DBEFileSet names           TempSpace names
Group names                View names
Index names
```

Native Language Object Names

All the object names in a DBEnvironment can be represented in the DBEnvironment language or in NATIVE 3000. The following rules for object names are the same as for ASCII:

- The length of an object name is specified as a number of bytes. Note that this would mean a maximum of 20 characters for a table name in English and 10 in Chinese, because Chinese is represented in a two-byte character set.
- Table and view names can be qualified by prefixing the owner name followed by a period ('.') The period serves as the delimiter and is thus a part of the syntax of SQL. It cannot be represented by a native language delimiter but must be ASCII.

DBEUserIDs

A **DBEUserID** is made up of a user's MPE XL user and account names connected with the @ symbol. An example is WOLFGANG@DBMS, where Wolfgang is the user name, and DBMS is the account name.

When a DBEnvironment is configured, ALLBASE/SQL grants DBA authority to the DBEUserID of the DBECreator. You cannot revoke DBA authority from the DBECreator.

Owner Names

Owner names can be one of the following:

- DBEUserID
- Group name
- Class name

Authorization Names

An **authorization name** identifies an owner name defined in the AUTHORIZATION clause of the CREATE SCHEMA statement. Authorization names must be unique within the DBEnvironment. There cannot be another owner, authorization group, or grantor with the same name on the system when the CREATE SCHEMA statement is issued.

Authorization names can be one of the following:

- DBEUserID
- Group name
- Class name

Compound Identifiers

Basic names and DBEUserIDs are considered **simple names**. In some cases, simple names are combined to form a **compound identifier**, which consists of an owner name combined with one or more basic names, with periods (.) between them.

Often you can abbreviate a compound identifier by omitting one of its parts. If you do this, a default value is automatically used in place of the missing part. For example, you can omit the owner name (and the period) when you refer to tables you own; ALLBASE/SQL generates the owner name by using your logon name.

A complete compound identifier, including all of its parts, is called a **fully qualified name**. The following are compound identifiers:

Authorization group identifier—[*Owner*.]*GroupName*

Column identifier— [[*Owner*.]*TableName*.]*ColumnName*

Constraint identifier— [*Owner*.]*ConstraintName*

Index identifier— [*Owner*.]*IndexName*

Module identifier—[*Owner*.]*ModuleName*

Procedure identifier— [*Owner*.]*ProcedureName*

Rule identifier—[*Owner*.]*RuleName*

Section identifier— [*Owner*.]*ModuleName*(*SectionNumber*)

Table identifier— [*Owner*.]*TableName*

View identifier— [*Owner*.]*ViewName*

Different owners can have modules, tables, or views by the same name; the fully qualified name of these objects must be unique in the DBEnvironment. Group names, however, must be unique in the DBEnvironment.

Host Variable Names

Host variables are used to pass information between an application program and ALLBASE/SQL. They are ordinary application program variables that happen to be used in SQL commands.

A host variable name must be preceded by a colon (:) when used in an SQL command. When used elsewhere in an application program, no colon should be used.

Host variable names must conform to ALLBASE/SQL's rules for basic names; however, they are allowed to be up to 30 bytes in length. In addition, host variable names must conform to the rules of the language in which the application program is written.

Local Variable Names

Local variables are used to hold data within a procedure. A local variable is declared in a DECLARE statement in the procedure, and it is prefixed with a colon (:) when used in any other statement. Local variable names must conform to ALLBASE/SQL's rules for basic names.

Parameter Names

Parameters are used to pass information between the database and a procedure. A parameter is identified in the parameter list of a CREATE PROCEDURE statement, and it is prefixed with a colon (:) when used in the body of the procedure. Parameter names must conform to ALLBASE/SQL's rules for basic names.

DBEnvironment and DBECon File Names

The name of a DBEnvironment and the name of its DBECon file are identical. This name uses the form shown here, follows HP-UX file naming conventions, and cannot exceed 128 characters, including slashes:

```
FileName[.GroupName[.AccountName]]
```

This name must always be enclosed in single quotation marks when specified in SQL commands. If athe group and account are not given, ALLBASE/SQL assumes the name specified is in the current group and account.

DBEFile and Log File Identifiers

DBEFiles and log files have **logical names** which conform to the rules for ALLBASE/SQL basic names. DBEFile and log file names are stored in the system catalog.

In addition to logical names, the physical DBEFiles and log files are referred to in the SQL syntax by **system file names**. If the group and account are not given, ALLBASE/SQL assumes the name specified is in the current group and account. System file names are always enclosed in single quotation marks in SQL commands.

TempSpace Names

A TempSpace name is a logical name for the area where temporary files are stored by ALLBASE/SQL. This name conforms to the rules for ALLBASE/SQL basic names. TempSpace names are stored in the system catalog.

Special Names

ALLBASE/SQL has several names with special meaning. You should not create objects with these names as owner:

- TEMP— Modules owned by TEMP are deleted when the transaction in which they are created terminates.
- CATALOG— This name is the owner of the catalog views.
- SYSTEM— This name designates the owner of the system views.
- HPRDBSS and STOREDSECT— These names designate the owners of the system tables. STOREDSECT owns the tables used to store compiled sections and views; HPRDBSS owns all other system tables.
- PUBLIC— This name refers to *all* users and authorization groups who have been granted CONNECT authority.
- HPODBSS— This name is reserved.
- SEMIPERM— This name is the owner of all semi-permanent sections.

7 Data Types

Every value in SQL belongs to some data type. A data type is associated with each value retrieved from a table, each constant, and each value computed in an expression.

This chapter discusses data types. The following sections are presented:

- Type Specifications
- Value Comparisons
- Overflow and Truncation
- Underflow
- Type Conversion
- Null Values
- Decimal Operations
- Date/Time Operations
- Binary Operations
- Long Operations
- Native Language Data

A data type defines a set of values. Reference to a previously defined data type is a convenient way of specifying the set of values that can occur in some context. For example, in SQL the type `INTEGER` is defined as the set of integers from –2,147,483,648 through +2,147,483,647, plus the special value `NULL`. If you define a column with type `INTEGER`, each value stored in the column must be either an integer in the range –2,147,483,648 through +2,147,483,647, or a null value (if `NOT NULL` is not specified).

Type Specifications

All the data in a column must be of the same type. Specify the data type for each column when you create a table or when you add a column to an existing table. The ALLBASE/SQL data types and the values you can specify for data of each type are shown in Table 7-1.

Table 7-1. ALLBASE/SQL Data Types

Group	Data Type	Description
Alpha-numeric	CHAR[ACTER][(*n*)]	String of fixed length *n*, where *n* is an integer from 1 to 3996 bytes. The default size is CHAR (1). The keyword CHARACTER is a synonym for CHAR.
	VARCHAR(*n*)	String of variable length no greater than *n*, where *n* must be an integer from 1 to 3996 bytes.
Numeric	DEC[IMAL][(*p*[,*s*])] NUMERIC[(*p*[,*s*])]	Fixed-point packed decimal number with a **precision** (maximum number of digits excluding sign and decimal point) no greater than *p*, where *p* is 1 through 27, and a **scale** (number of digits to the right of the decimal) of *s*, where *s* is from 0 through *p*. E (exponential) and L (Pascal longreal) notation are not allowed in the specification of a decimal value. Operations on data of type DECIMAL are often much more precise than operations on data of type FLOAT. The default for NUMERIC and DECIMAL types is DECIMAL (27,0). DEC and NUMERIC are synonyms for DECIMAL.
	FLOAT[(*p*)] or DOUBLE PRECISION	Long (64-bit) floating point number. This is an approximate numeric value consisting of an exponent and a mantissa. The precision, *p*, is a positive integer that specifies the number of significant binary digits in the mantissa. The value of *p* can be from 25 to 53. The default is 53. The range of negative numbers that can be represented is –1.79769313486230E+308 to –2.22507385850721E–308. The range of positive numbers that can be represented is 2.22507385850721E–308 to 1.79769313486230E+308. E (exponential) or L (Pascal longreal) notation can be used to specify FLOAT values. DOUBLE PRECISION is a synonym for FLOAT(53).

Table 7-1. ALLBASE/SQL Data Types

Group	Data Type	Description
	FLOAT(*p*) or REAL	Short (32-bit) floating point number. This is an approximate numeric value consisting of an exponent and a mantissa. The precision, *p*, is a positive integer that specifies the number of significant binary digits in the mantissa. The value of *p* can be from 1 to 24. The default (using REAL) is 24. The range of negative numbers that can be represented is –3.402823E+38 to –1.175495E–38. The range of positive numbers that can be represented is 3.402823E+38 to 1.175495E–38. REAL is a synonym for FLOAT (24).
	INT[EGER]	Integer in the range –2147483648 (-2^{31}) to 2147483647 ($2^{31}-1$). INT is a synonym for INTEGER.
	SMALLINT	Integer in the range –32768 (-2^{15}) to 32767 ($2^{15}-1$).
Date/Time	DATE	String of form 'YYYY-MM-DD', where YYYY represents the calendar year, MM is the month, and DD is the day of the month. DATE is in the range from '0000-01-01' to '9999-12-31'.
	TIME	String of the form 'HH:MI:SS: where HH represents hours, MI is minutes, and SS is seconds. TIME is in the range from '00:00:00' to '23:59:59'.
	DATETIME	String of the form 'YYYY-MM-DD HH:MI:SS.FFF', where YYYY represents the calendar year, MM is the month, DD is the day, HH the hour, MI the minute, SS the second, and FFF thousandths of a second. The range is from '000-01-01 00:00:00.000' to '9999-12-31 23:59:59.999'.
	INTERVAL	String of the form 'DDDDDDD HH:MI:SS.FFF', where DDDDDDD is a number of days, HH a number of hours, MI a number of minutes, SS a number of seconds, and FFF a number of thousandths of a second. The range is from '0 00:00:00.000' to '3652436 23:59:59.999'.
Binary	BINARY(*n*)	Binary string of fixed length *n*, where *n* is an integer from 1 to 3996 bytes. Each byte stores 2 hexadecimal digits.
	VARBINARY(*n*)	Binary string of variable length no greater than *n*, where *n* is an integer from 1 to 3996 bytes. Each byte stores 2 hexadecimal digits.
	LONG BINARY(*n*)	Binary string of fixed length *n*, where *n* is an integer from 1 to ($2^{31}-1$) bytes.
	LONG VARBINARY(*n*)	Binary string of variable length no greater than *n*, where *n* is an integer from 1 to ($2^{31}-1$) bytes.

Your choice of data types can affect the following:

- How values are used in expressions. Some operations can be performed only with data of a certain type. For example, arithmetic operations are limited to numeric and date/time data types, such as INTEGER, SMALLINT, FLOAT, DECIMAL, DATE, TIME, DATETIME, or INTERVAL. Pattern matching with the LIKE predicate can be performed only with string data, that is, data of types CHAR or VARCHAR.
- The result of operations combining data of different types. When comparisons and expressions combining data of different but compatible types are evaluated, ALLBASE/SQL performs type conversion, as described later in this chapter.
- How values are transferred programmatically. When data is transferred between ALLBASE/SQL and an application program in host variables, ALLBASE/SQL uses the data type equivalencies described in the ALLBASE/SQL application programming guides.

Table 7-2. contains the storage requirements of the various data types.

Table 7-2. Data Type Storage Requirements

Type	Storage Required
CHAR (n)	n bytes (where n must be an integer from 1 to 3996)
VARCHAR (n)	n bytes (where n must be an integer from 1 to 3996)
DECIMAL (p[,s])	4 bytes (where p <= 7) or 8 bytes (where 7 < p <= 15) or 12 bytes (where 15 < p <= 23) or 16 bytes (where p > 23)
FLOAT	8 bytes
REAL	4 bytes
INTEGER	4 bytes. Integer values less than -2147483648 (-2**31) or larger than 2147483647 (2**31 - 1) up to 15 digits long are stored as decimals with a precision of 15 and a scale of 0, i.e., equivalent to DECIMAL (15,0)
SMALLINT	2 bytes
DATE	16 bytes
TIME	16 bytes
DATETIME	16 bytes
INTERVAL	16 bytes
BINARY (n)	n bytes (where n must be an integer from 1 to 3996)
VARBINARY (n)	n bytes (where n must be an integer from 1 to 3996)
LONG BINARY (n)	n bytes (where n must be an integer from 1 to 2^{31} - 1)
LONG VARBINARY (n)	n bytes (where n must be an integer from 1 to 2^{31} - 1)

Value Comparisons

When you compare a CHAR and a VARCHAR string, ALLBASE/SQL pads the shorter string with ASCII blanks to the length of the longer string. The two strings are equal if the characters in the shorter string match those in the longer string and if the excess characters in the longer string are all blank.

If a case sensitive CHAR column is compared to a CHAR column that is not case sensitive, both columns are treated as case sensitive. If a string constant is compared to a column that is not case sensitive, then the string constant is treated as not case sensitive.

Before comparing DECIMAL numbers having different scales, ALLBASE/SQL extends the shorter scale with trailing zeroes to match the larger scale.

Items of type DATE, TIME, DATETIME, and INTERVAL can be compared only with items of the same type, or with CHAR or VARCHAR strings in the correct format. All comparisons are chronological, which means the point which is farthest from '0000-01-01 00:00:00.000' is the greatest value. ALLBASE/SQL attempts to convert CHAR or VARCHAR strings to the default date/time format before performing the comparison.

When you compare a BINARY and a VARBINARY hexadecimal string, ALLBASE/SQL pads the shorter binary string with binary zeroes to the length of the longer string. When comparing two BINARY or VARBINARY hexadecimal strings having different lengths, ALLBASE/SQL compares the excess binary digits of the longer binary string with binary zeroes. The two strings are equal if the binary digits in the shorter string match those in the longer string and if the excess binary digits in the longer string are all binary zero.

The chapter "Search Conditions" provides more information on comparison operations.

Overflow and Truncation

Some operations can result in data overflow or truncation. Overflow results in loss of data on the left. Truncation results in loss of data on the right.

Overflow or truncation can occur in several instances as follows:

- During arithmetic operations, for example, when multiplication results in a number larger than the maximum value allowable in its type. Arithmetic operations are defined inChapter 8 , “Expressions.”
- When using aggregate functions, for example, when the sum of several numbers exceeds the maximum allowable size of the type involved. Aggregate functions are defined in Chapter 8 , “Expressions.”
- During type conversion, as when an INTEGER value is converted to a SMALLINT value. Type conversion is discussed later in this chapter.

Because large integers (less than –2147483648 (-2^{31}) or larger than 2147483647 ($2^{31}-1$) up to 15 digits long) are stored as decimals, large integer overflow actually results in a `DECIMAL OVERFLOW` message.

Overflow always causes an error.

Truncation can cause a warning for the following types of data:

- Alphanumeric data—A warning occurs if a string is truncated because it is too long for its target location. No error is given if truncation occurs on input.
- Numeric data—No error or warning occurs when zeroes are dropped from the left or when any digit is dropped from the fractional part of DECIMAL or FLOAT values. Otherwise, truncation of numeric values causes an error.
- LONG data—A warning occurs if LONG column data is truncated because it is too long for its target input file. The output file location is modified to fit the LONG column length, so no truncation error occurs on LONG column output. If the file system fills up, or the limit of shared memory is reached, a system error occurs.

Refer to the *ALLBASE/SQL Message Manual* for information on handling warnings and errors.

Underflow

Underflow occurs when a FLOAT or a REAL value is too close to zero to be represented by the hardware. Underflow always causes an error.

Type Conversion

ALLBASE/SQL converts the type of a value in the following situations:

- Including values of different types in the same expression.
- Moving data from a host variable to a column or a column to a host variable of a different type.

The valid type combinations are shown in Table 7-1.

Table 7-3. Valid Type Combinations

Source Data Type	Target Data Type
CHAR or VARCHAR	CHAR or VARCHAR
	DATE, TIME DATETIME, or INTERVAL when CHAR value involved in date/time math or inserted into or compared to a date/time column
CHAR or VARCHAR	BINARY or VARBINARY (from host variable/constant into a binary column only)
BINARY or VARBINARY	BINARY or VARBINARY
BINARY or VARBINARY	CHAR or VARCHAR (from column into host variable, or comparing a binary column with a char column or value)
DECIMAL, FLOAT, REAL, INTEGER, SMALLINT	Any numeric type
DATE, TIME, DATETIME, INTERVAL	CHAR or VARCHAR (except in LIKE predicate)

In some cases, such as the following, data conversion can lead to overflow or truncation:

- Overflow can occur during these conversions:

FLOAT	to DECIMAL, INTEGER or SMALLINT
FLOAT	to REAL
REAL	to DECIMAL, INTEGER, or SMALLINT
DECIMAL	to DECIMAL, INTEGER, or SMALLINT
INTEGER	to DECIMAL or SMALLINT
SMALLINT	to DECIMAL

- Overflow of the integer part and truncation of the fractional part of a number can occur during a FLOAT-to-DECIMAL conversion, because ALLBASE/SQL aligns the decimal points.

- Truncation of the fractional part of a value occurs during these conversions:

  DECIMAL to SMALLINT or INTEGER

  DECIMAL to DECIMAL when the target scale is smaller than the source scale

  FLOAT to INTEGER, SMALLINT, DECIMAL, or REAL

  REAL to INTEGER, SMALLINT, or DECIMAL

- Truncation can occur during these conversions if the target is too small:

  DATE, TIME, DATETIME or INTERVAL to VARCHAR or CHAR

  CHAR to VARCHAR, BINARY or VARBINARY

  VARCHAR to CHAR, BINARY or VARBINARY

  VARBINARY to BINARY, CHAR or VARCHAR

  BINARY to VARBINARY, CHAR, or VARCHAR

When you use numeric data of different types in an expression or comparison operation, the data type of the lesser type is converted to that of the greater type, and the result is expressed in the greater type. Numeric types have the following precedence:

FLOAT

REAL, DECIMAL

INTEGER

SMALLINT

Comparison operations or expressions involving different numeric data types result in conversion from one data type to another as explained in Table 7-4.

Table 7-4. Conversions from Combining Different Numeric Data Types

Operations containing:	Result:
DECIMAL and INTEGER types only	All participating integers are converted to DECIMAL quantities having a precision of 10 and a scale of 0.
DECIMAL and SMALLINT types only	All participating SMALLINT values are converted to DECIMAL quantities having a precision of 5 and a scale of 0.
One item of type FLOAT	All participating integer and decimal operands are converted to FLOAT quantities and precision can be lost.
One item of type REAL	All arithmetic involving REAL operands results in a type of FLOAT. All participating integer and decimal operands are converted to FLOAT quantities and precision can be lost.

Null Values

A null value is a special value that indicates the absence of a value. Any column in a table or parameter or local variable in a procedure, regardless of its data type, can contain null values unless you specify NOT NULL for the column when you create the table or the procedure. NULL is used as a placeholder for a value that is missing or unknown. These properties of null values affect operations on rows or parameters or local variables containing the following values:

- Null values always sort highest in a sequence of values.
- Two null values are not equal to each other except in a GROUP BY or SELECT DISTINCT operation, or in a unique index.
- An expression containing a null value evaluates to null; for example, five minus null evaluates to null.

Because of these properties, ALLBASE/SQL ignores columns or rows or parameters or local variables containing null values in these situations:

- Evaluating comparisons
- Joining tables, if the join is on a column containing null values
- Executing aggregate functions
- Evaluating if/while conditions or assignment expressions

In several SQL predicates, described in Chapter 9 , “Search Conditions,” you can explicitly test for null values. In an application program, you can use indicator variables to handle input and output null values.

Decimal Operations

The precision (p) and scale (s) of a DECIMAL result depend on the operation used to derive it. The following rules define the precision and scale that result from arithmetic operations on two decimal values having precisions p_1 and p_2 and respective scales s_1 and s_2. Rules are also provided for the resulting precision and scale of aggregate functions that operate on a single expression having a precision of p_1 and a scale of s_1. Arithmetic operations and aggregate functions are discussed further in Chapter 8 , “Expressions.”

Addition and Subtraction

$p = MIN(27, MAX (p_1 - s_1 , p_2 - s_2) + MAX(s_1, s_2)+ 1)$

$s = MAX (s_1,s_2)$

Multiplication

$p = MIN (27, p_1 + p_2)$

$s = MIN (27, s_1 + s_2)$

Division

$p = 27$

$s = 27 - MIN (27, p_1 - s_1 + s_2)$

where p_1 and s_1 describe the numerator operand, and p_2 and s_2 describe the denominator operand.

MAX and MIN Functions

$p = p_1$

$s = s_1$

SUM Function

$p = 27$

$s = s_1$

AVG Function

$p = 27$

$s = 27 - p_1 + s_1$

Date/Time Operations

DATE, TIME, DATETIME, or INTERVAL values may only be assigned to a column with a matching data type or to a fixed or variable length character string column or host variable. Otherwise an error condition is generated. All rules regarding assignment to a character string are also true for date/time assignment to a character string variable or column.

Conversions of the individual fields of a date/time data type follow the rules given earlier in this subsection for the corresponding data type.

NOTE The validity of dates prior to 1753 (transition of Julian to Gregorian calendar) cannot be guaranteed.

DATE, TIME, DATETIME, and INTERVAL data types behave similar to character strings in data manipulation statements. The examples below illustrate this.

Examples

INSERT

DATETIME, DATE, TIME and INTERVAL values:

```
INSERT INTO ManufDB.TestData
 (BatchStamp, TestDate, TestStart, TestEnd, LabTime, PassQty, TestQty)
  VALUES ('1984-08-19 08:45:33.123',
          '1984-08-23',
          '08:12:19', '13:23:01',
          '5 10:35:15.700',
           49, 50)
```

SELECT

DATE and TIME values:

```
SELECT TestDate, TestStart
 FROM ManufDB.TestData
 WHERE TestDate = '1984-08-23'
```

DATETIME and INTERVAL values:

```
SELECT BatchStamp, LabTime
 FROM ManufDB.TestData
 WHERE TestDate = '1984-08-23'
```

UPDATE

DATE and TIME values:

```
UPDATE ManufDB.TestData
   SET TestDate = '1984-08-25', TestEnd = '19:30:00'
 WHERE BatchStamp = '1984-08-19 08:45:33.123'
```

INTERVAL values:

```
UPDATE ManufDB.TestData
  SET LabTime = '5 04:23:00.000'
 WHERE TestEnd = '19:30:00'
```

Note that the radix of DATE and TIME data is seconds, whereas the radix of DATETIME and INTERVAL data is milliseconds.

Date/time data types can also be converted to formats other than the default formats by the date/time functions described in Chapter 8 , “Expressions.”

Use of Date/Time Data Types in Arithmetic Expressions

You can use a variety of operations to increment, decrement, add or subtract date, time, datetime, and interval values. Table 7-5. shows the valid operations and the data type of the result:

Table 7-5. Arithmetic Operations on Date/Time Data Types

Operanda	Operator	Operand b	Result Type
DATE	+,–	INTERVAL	DATE
INTERVAL	+	DATE	DATE
DATE	–	DATE	INTERVAL
TIME	+,–	INTERVAL	TIME
INTERVAL	+	TIME	TIME
TIME	–	TIME	INTERVAL
DATETIME	+,–	INTERVAL	DATETIME
INTERVAL	+	DATETIME	DATETIME
DATETIME	–	DATETIME	INTERVAL
INTERVAL	+,–	INTERVAL	INTERVAL
INTERVAL	*, /	INTEGER	INTERVAL
STRING[a]	–	DATE	INTERVAL
STRING[b]	+	DATE	DATE
DATE	–	STRINGa	INTERVAL
DATE	+	STRINGb	DATE
STRING[c]	–	DATETIME	INTERVAL
DATETIME	–	STRINGc	INTERVAL
STRINGb	+	DATETIME	DATETIME
DATETIME	+	STRING	DATETIME

Table 7-5. Arithmetic Operations on Date/Time Data Types

Operanda	Operator	Operand b	Result Type
STRING[d]	–	TIME	INTERVAL
STRINGb	+	TIME	TIME
TIME	–	STRINGd	INTERVAL
TIME	+	STRINGd	TIME
STRINGb	+,–	INTERVAL	INTERVAL
INTERVAL	+,–	STRINGb	INTERVAL

a. The format for string should be DATE.
b. The format for string should be INTERVAL.
c. The format for string should be DATETIME.
d. The format for string should be TIME.

These arithmetic operations obey the normal rules associated with dates and times. If a date/time arithmetic operation results in an invalid value (for example, a date prior to '0000-01-01'), an error is generated. If the format for the string does not match the above default type, an error is generated. Another solution is to apply `TO_DATE`, `TO_TIME`, `TO_DATETIME`, and `TO_INTERVAL` to the string so that the correct format is used.

You can also use the Add Months function to add or subtract from the month portion of the `DATE` or `DATETIME` column. In the result, the day portion is unaffected, only the month and, if necessary, the year portions are affected. However, if the addition of the month causes an invalid day (such as 89-02-30), then a warning message is generated and the value is truncated to the last day of the month.

Use of Date/Time Data Types in Predicates

`DATE`, `TIME`, `DATETIME`, and `INTERVAL` data types can be used in all predicates *except* the LIKE predicate. LIKE works only with CHAR or VARCHAR values and so requires the use of the `TO_CHAR` conversion function to be used with a `DATETIME` column. Items of type `DATE`, `TIME`, `DATETIME`, and `INTERVAL` can be compared with items of the same type or with literals of type CHAR or VARCHAR. All comparisons are chronological, which means that the point which is farthest from '0000-01-01 00:00:00.000' is the greatest value. String representations of each data type (in host variables or as literals) can also be compared following normal string comparison rules. Some examples follow:

```
   SELECT * FROM ManufDB.TestData
    WHERE BatchStamp = '1984-06-19 08:45:33.123'
      AND TestDate = '1984-06-27'

SELECT * FROM ManufDB.TestData
 WHERE Testend - TestStart <= '0 06:00:00.000'
```

Date/Time Data Types and Aggregate Functions

You can use the aggregate functions MIN, MAX, and COUNT in queries on columns of type DATE, TIME, DATETIME, and INTERVAL. SUM and AVG can be done on INTERVAL data types only.

Binary Operations

BINARY or VARBINARY values may be assigned to a column with a matching data type or to a fixed or variable length character string host variable. All rules regarding assignment to a character string are also true for binary assignment to a character string variable.

LONG BINARY and LONG VARBINARY values cannot be converted to any other type, and cannot participate in any expressions except as assignments to long functions and string functions.

Character (ASCII) or hexadecimal format is used when inserting BINARY and VARBINARY data literals into a column. Hexadecimal format is preceded by the hexadecimal indicator 0x when inserting data through ISQL, but not if you are inserting data through an application program. The result of a SELECT statement on a BINARY or VARBINARY column is in hexadecimal format.

You cannot insert BINARY literals (0's and 1's) into a CHAR column in ISQL; however, you can insert them in an application program using a host variable.

Long Operations

LONG columns in ALLBASE/SQL enable you to store a very large amount of binary data in your database and to reference that data using a column name. You might use LONG columns to store text files, software application code, voice data, graphics data, facsimile data, or test vectors. Storing data in the database gives you the the advantages of ALLBASE/SQL's recoverability, concurrency control, locking strategies, and indexes on related columns.

The concept of how LONG column data is stored and retrieved differs from that of non-LONG columns. LONG data is not processed by ALLBASE/SQL. Any formatting, viewing, or other processing must be accomplished by a preprocessed application program. Refer to the ALLBASE/SQL application programming guides for information on accessing LONG columns from a preprocessed application.

Like other column data types, the LONG column is defined with the CREATE TABLE or ALTER TABLE statement. A LONG column descriptor, called the LONG column I/O string, describes where the LONG column input data is located and where the data is placed when a SELECT or FETCH statement is executed. The LONG column I/O string is specified as an element in the VALUES clause of an INSERT or the SET clause of an UPDATE operation. When you use the SELECT or FETCH statement, the LONG column descriptor is returned to the ISQL display or the host variable and the long column data is placed either in the operating system file or stored memory.

Defining LONG Column Data with CREATE TABLE or ALTER TABLE

Following is the syntax for specifying a column definition for a LONG column in either the CREATE TABLE or ALTER TABLE statement. A maximum of 40 such LONG columns can be defined for a single table.

```
(ColumnName LONG ColumnDataType [IN DBEFileSetName]
[LANG = ColumnLanguageName] [NOT NULL]) [,...]
```

The LONG data is stored in DBEFiles. These files can occupy up to 2^{31} –1 bytes. For better performance and storage considerations, specify a separate DBEFileSet when defining the LONG column.

If IN *DBEFileSetName* is not specified for a LONG column, this column's data is stored in the same DBEFileSet as its related table. Do not specify the SYSTEM DBEFileSet as this could severely impact database performance.

In the following example, LONG data for PartPicture is stored in the DBEFileSet PartPictureSet, while data for columns PartName and PartNumber is stored in PartsIllusSet:

```
CREATE TABLE PurchDB.PartsIllus
             (PartName CHAR(16),
             PartNumber INTEGER,
             PartPicture LONG VARBINARY(1000000) IN PartPictureSet)
          IN PartsIllusSet
```

The next statement specifies that data for the new LONG column, PartModule, will be stored in PartPictureSet:

```
ALTER TABLE PurchDB.PartsIllus
        ADD PartModule LONG VARBINARY(50000) IN PartPictureSet
```

Since LONG data for PartMap will be stored in the same DBEFileSet as its related table, PartsIllus, it goes to PartsIllusSet.

```
ALTER TABLE PurchDB.PartsIllus
        ADD PartMap LONG VARBINARY(70000)
```

Defining Input and Output with the LONG Column I/O String

The INSERT and UPDATE statements use the LONG column I/O string to define the various input and output parameters for any LONG column. You need to understand this string in order to input, change, or retrieve LONG data.

The LONG column I/O string has an input portion (indicated with <) and an output portion (indicated with >). The input portion of the LONG column I/O string, also referred to as the **input device**, specifies the location of data that you want written to the database. You can indicate a file name or a heap address and heap length.

A variable length record file cannot be input to a LONG column.

The output portion of the LONG column I/O string (the **output device**) specifies where you want LONG data to be placed when you execute the SELECT or FETCH statement. You have the option of specifying a file name, part of a file name, or having ALLBASE/SQL specify a file name. You also can direct output to the heap address (in this case, ALLBASE/SQL will select the heap address). Additional output parameters allow you to append to or overwrite an existing file. The output device specification is stored in the database table and is available to you when you use the OUTPUT_DEVICE function or OUTPUT_NAME function together with a SELECT or FETCH statement. For more information on the OUTPUT_DEVICE and OUTPUT_NAME functions, see Chapter 8 , “Expressions,” in this document.

The examples in the following sections illustrate the use of the input and output portions of the LONG column I/O string. The complete syntax for the LONG column I/O string is presented under the INSERT, UPDATE, and UPDATE WHERE CURRENT statements.

It is important to note that files used for LONG column input and output are opened and closed by ALLBASE/SQL. You do not need to open or close the files for use in the DBEnvironment. ALLBASE/SQL does not control the input or output device files on the operating system. That is, if there is a rollback work, ALLBASE/SQL will not remove the physical operating system file generated by the SELECT statement.

Using INSERT with LONG Column Data

As with any column, you use the SQL INSERT statement or an ISQL INPUT command to initially put data in a LONG column. The LONG column I/O string requires an input device, but the output device is optional.

The following examples illustrate some of the options available to you.

Using INSERT with No Specified File Options

In this example, data from the file hammer.tools becomes the contents of the LONG column PartPicture. The output device is the file hammer. If this file already exists when the SELECT or FETCH statement is issued, it is not overwritten or appended to, and an error is generated.

```
INSERT INTO PurchDB.PartsIllus
 VALUES ('hammer'
          100,
         '<hammer.tools >hammer')
```

Using INSERT with the Overwrite Option

When you want to reuse an existing output device file when the inserted data is later selected or fetched, specify the overwrite option. Here if file wrench already exists at INSERT time, it is overwritten:

```
INSERT INTO PurchDB.PartsIllus
     VALUES ('hammer',
            100,
            '<hammer.tools >!wrench')
```

Using INSERT with the Append Option

You can append LONG data to an existing file. If the file limit for the wrench files is inadequate to hold the data that is to be appended, a warning is returned (DBWARN 2051), but data up to the file limit is added to the file. In this example, when the LONG column PartPicture is selected or fetched, output is appended at the end of the file wrench:

```
INSERT INTO PurchDB.PartsIllus
     VALUES ('hammer',
            100,
            '<hammer.tools >>wrench')
```

Using INSERT with the Wildcard Option

Depending on your application, you may need to assign a specific, known name to the output device. On the other hand, a partially generic name or a completely unknown name may be desirable. In this example, the output device name begins with PRT and is followed by a five-character, random wild card, for instance, 'PRT123AB':

```
INSERT INTO PurchDB.PartsIllus
     VALUES ('hammer'
            100,
            '< hammer.tools >PRT$')
```

Using INSERT with Heap Space Input and Output

You have the option of using a heap address to indicate the location of input data. Output datamay be directed toa heap address generated by ALLBASE/SQL at output time. In the next example, 4000 bytes of data flow from heap address 1230 to the PartsIllus table, and when this data is selected or fetched, it goes to the heap address:

```
INSERT INTO PurchDB.PartsIllus
     VALUES ('saw'
            300,
            '<%1230:4000 >%$')
```

Using SELECT with LONG Column Data

The concept of how data is retrieved differs from that of non-LONG columns. The output portion of the LONG column I/O string (rather than the data itself) is obtained with the SELECT or FETCH statement. The LONG data goes to a file or heap space.

In this example, the SELECT statement places the LONG data from the PartPicture column in a file or in heap space, as specified in the LONG column I/O string when the PartPicture column was inserted or updated. The SELECT statement puts the file name or heap space address in the PartPicture LONG column descriptor. In an application, the contents of the descriptor are placed in a host variable and may be parsed to extract the file name or heap space address. When a long field column is selected using ISQL, the file name or heap space address is displayed in the column whose heading is the long field name. Refer to the "Programming with LONG Columns" chapter of the appropriate application programming guide for information on the format of the LONG column descriptor.

```
SELECT PartPicture
  FROM PurchDB.PartsIllus
 WHERE PartName = 'saw'
```

Using UPDATE with LONG Column Data

When you issue an UPDATE on a LONG column, you have the following options:

- Change the stored data as well as the output device name and/or options.
- Change the stored data only.
- Change the output device name and/or options only.

You must specify either the input device, the output device, or both.

Examples

The following examples present a sampling of possible combinations.

Using UPDATE to Change Stored Data and Output Device Name

In this example, data from the file newhammer.tools is inserted into the LONG column PartPicture replacing the previously stored data. The output device name is changed to be the file newhammer. Should file newhammer already exist when the SELECT or FETCH statement is issued, it is not overwritten, and an error is generated.

```
UPDATE PurchDB.PartsIllus
   SET PartPicture = '<newhammer.tools >newhammer'
 WHERE PartName = 'hammer'
```

Using UPDATE to Change Stored Data Only

Here the stored data in LONG column PartPicture is replaced with data from the file `../tools/newhammer`. Assuming the original output device was named hammer, when you select or fetch the PartPicture column, the updated output still goes to a file named hammer.

```
UPDATE PurchDB.PartsIllus
   SET PartPicture = '<newhammer.tools'
 WHERE PartName = 'hammer'
```

Using UPDATE to Change the Output Device Name and Options

You may want to change the output file name but not the LONG data associated with a particular column. Here newhammer becomes the output device name. When LONG column PartPicture is SELECTed or FETCHed, output is appended to the file newhammer.

```
UPDATE PurchDB.PartsIllus
   SET PartPicture = '>>newhammer'
 WHERE PartName = 'hammer'
```

Using UPDATE with Heap Space Input and Output

You may decide to use heap space as your input input device. Output may be directed to a heap address. In this example, LONG data flows from file newsaw to the PartsIllus table, and when this data is selected or fetched it goes to a heap address:

```
UPDATE PurchDB.PartsIllus
  SET PartPicture = '< newsaw >%$'
  WHERE PartName = 'saw'
```

In the next example, 4000 bytes of data flow to the database from heap address 1000 and when the LONG column is selected or fetched, data goes to the file newsaw:

```
UPDATE PurchDB.PartsIllus
   SET PartPicture = '<%1000:4000 >newsaw'
 WHERE PartName = 'saw'
```

Using DELETE with LONG Column Data

DELETE and DELETE WHERE CURRENT syntax is unchanged when used with LONG columns. It is limited in that a LONG column cannot be used in the WHERE clause.

In the following example, any rows in PurchDB.PartsIllus with the PartName of hammer are deleted:

```
DELETE FROM PurchDB.PartsIllus
      WHERE PartName = 'hammer'
```

When LONG data is deleted, the space it occupied in the DBEnvironment is released when your transaction ends. But the data files still exist on the operating system.

Native Language Data

Character data in the DBEnvironment can be represented in the native language specified by the DBEnvironment language. When native language character columns are created, they follow the same rules as CHAR and VARCHAR columns. For character columns, size is defined in bytes. Thus a column defined as CHAR (20) could hold 20 characters in ASCII or 10 characters in Japanese Kanji.

Numeric data must be in ASCII representation.

Pattern matching is in terms of **conceptual characters** rather than bytes. This is necessary for languages in which there are both one-byte and two-byte characters frequently mixed in the same string. An example is Japanese, in which the Kanji and Hiragana characters occupy 16 bits each, whereas the Katakana characters use only 8 bits. Conceptual character matching is also necessary to establish a collating sequence that includes the one-byte ASCII character set as a subset of a two-byte character set such as Chinese.

Truncation is done on a character basis. For example, imagine a column defined as CHAR (20). If a string contains 11 Kanji characters, or 22 bytes, the last character is truncated if you try to insert it into the column. In a case where a string contains both Kanji and Katakana characters and is 21 bytes long, the truncation depends on the size of the last character. If it is a 2-byte Kanji character, the data is truncated to 19 bytes; if it is a 1-byte Katakana character, the data is truncated to 20 bytes.

An implicit type conversion occurs when an NATIVE 3000 string is compared to a native language CHAR or VARCHAR type. The shorter string is padded with ASCII blanks before the comparison is done.

When a case insensitive ASCII expression is compared to a case insensitive NLS expression, the two expressions are compared using the NLS collation rules. The case insensitive NLS comparison is done by using the `NLSCANMOVE` and `NLCOLLATE` intrinsics. The same ASCII characters in upper and lower case are equivalent. The same accent characters (extended characters) in upper and lower case are also equivalent. However, an accent character may not be the same as its ASCII equivalent, depending on the specific language collation table.

8 Expressions

This chapter discusses value specification. The following sections are presented:

- Expression
- Add Months Function
- Aggregate Functions
- CAST Function
- Constant
- Current Functions
- Date/Time Functions
- Long Column Functions
- String Functions
- TID Function

An **expression** specifies a **value** to be obtained in one of the following ways:

- From a column of a table
- From a host variable in an application program
- From a dynamic parameter
- From a local variable or parameter in a procedure
- From a constant
- By adding, subtracting, multiplying, dividing, or negating values
- By evaluating an aggregate function
- By evaluating a date/time (conversion, current, or add months) function
- By evaluating a long column or string function
- By a combination of these methods

Expressions are used for several purposes including:

- To identify columns. In the SELECT statement, expressions are used in the select list to identify column values to be retrieved.

  The SELECT statement is also part of the CREATE VIEW, DECLARE CURSOR, and INSERT statements. The expressions in this case identify columns that qualify for the view, the cursor, or the insert operation.

- To identify rows. In the search condition of the following statements, expressions help define the set of rows affected: SELECT, INSERT, UPDATE, DELETE, CREATE VIEW, and DECLARE. Refer to the “Search Conditions” chapter for more information.

- To define new column values. In the UPDATE statement, expressions define a new value for a column in an existing row.

Expression

An expression can consist of a **primary** or several primaries connected by arithmetic operators. A primary is a signed or unsigned value derived from one of the items listed in the SQL syntax below.

Scope

SQL Data Manipulation Statements

SQL Syntax

```
[+
 -] { ColumnName
        USER
        :HostVariable [[INDICATOR]:IndicatorVariable]
        ?
        :LocalVariable
        :ProcedureParameter
        ::Built-inVariable
        AddMonthsFunction
        AggregateFunction
        Constant
        DateTimeFunction
        CurrentFunction
        LongColumnFunction
        StringFunction
        CASTFunction
        (Expression)
        TIDFunction                                          }

[ { *
    /
    +
    -
    ||} [+
         - ] { ColumnName
               :HostVariable[[INDICATOR]:IndicatorVariable]
               ?
               :LocalVariable
               :ProcedureParameter
               ::Built-inVariable
               AddMonthsFunction
               AggregateFunction
               Constant
               DateTimeFunction
               CurrentFunction
               LongColumnFunction
               StringFunction
               CASTFunction
               Expression)                                       }] [ ...]
```

Parameters

+, – designate unary plus and unary minus. Unary plus assigns the primary a positive value. Unary minus assigns the primary a negative value. Default is positive.

ColumnName is the name of a column from which a value is to be taken; column names are defined in the "Names" chapter.

USER The keyword USER can be used as a character constant in several locations as follows:

- In a WHERE clause predicate when comparing it to a character string, for example:

```
WHERE Owner = USER
WHERE Owner IN ('ALLUSERS', USER)
```

- In the VALUES clause of the INSERT statement, for example:

```
VALUES (USER)
```

- In a DEFAULT clause of a column definition, for example:

```
Owner CHAR(20) DEFAULT USER NOT NULL
```

- In a SELECT list, returning a character string, for example:

```
SELECT USER, column1
```

- In an UPDATE SET clause, assigning a value to a character string, for example:

```
SET Owner = USER
```

USER evaluates to the current DBEUserID. In ISQL, it evaluates to the login name of the ISQL user. From an application program, it evaluates to the login name running the program. USER behaves like a CHAR(20) constant, with trailing blanks if the login name has fewer than 20 characters.

Note that if a column in your table is named USER, it must be preceded with the table name for column values to be selected. The function USER takes precedence over any column named USER.

HostVariable contains a value in an application program being input to the expression.

IndicatorVariable names an indicator variable, whose value determines whether the associated host variable contains a NULL value:

> = 0	the value is not NULL
< 0	the value is NULL (The value in the host variable will be ignored.)

? is a place holder for a dynamic parameter in a prepared SQL statement in an application program. The value of the dynamic parameter is supplied at run time.

LocalVariable contains a value in a procedure.

ProcedureParameter contains a value that is passed into or out of a procedure.

Built-inVariable is one of the following built-in variables used for error handling:

- ::sqlcode
- ::sqlerrd2
- ::sqlwarn0
- ::sqlwarn1
- ::sqlwarn2
- ::sqlwarn6
- ::activexact

The first six of these have the same meaning that they have as fields in the SQLCA in application programs. Note that in procedures, sqlerrd2 returns the number of rows processed for all host languages. However, in application programs, sqlerrd3 is used in COBOL, Fortran, and Pascal, while sqlerr2 is used in C. ::activexact indicates whether a transaction is in progress or not. For additional information, refer to the application programming guides and to Chapter 4 , “Constraints, Procedures, and Rules.”

AddMonthsFunction returns a value that represents a DATE or DATETIME value with a certain number of months added to it.

AggregateFunction is a computed value; aggregate functions are defined in this chapter.

Constant is a specific value; constants are defined in this chapter.

DateTimeFunction returns a value that is a conversion of a date/time data type into an INTEGER or CHAR value, or from a CHAR value.

CurrentFunction returns a value that represents the current DATE, TIME, or DATETIME.

LongColumnFunction returns information from a long column descriptor.

StringFunction returns a partial value or attribute of string data.

CASTFunction converts data from one data type to another.

(Expression) is one or more of the above primaries, enclosed in parentheses.

* multiplies two primaries.

/ divides two primaries.

\+ adds two primaries.

– subtracts two primaries.

|| concatenates two string operands.

TIDFunction returns the database address of a row (or rows for a BULK SELECT) of a table or an updatable view. You cannot use mathematical operators with this function except to compare it (using = or <>) to a value, host variable, or dynamic parameter.

Description

- Arithmetic operators can be used between numeric values, that is, those with data types of FLOAT, REAL, INTEGER, SMALLINT, or DECIMAL. Refer to the "Data Types" chapter for rules governing the resulting precision and scale of DECIMAL operations.
- Arithmetic operators can also be used between `DATE, TIME, DATETIME`, and `INTERVAL` values. Refer to the "Data Types" chapter for rules on the valid operations and the resulting data types.
- Elements in an expression are evaluated in the following order:

  — Aggregate functions and expressions in parentheses are evaluated first.

  — Unary plusses and minuses are evaluated next.

  — The * and / operations are performed next.

  — The + and – operations are then performed.
- You can enclose expressions in parentheses to control the order of their evaluation. For example:

```
10 * 2 – 1 = 19, but
10 * (2–1) = 10
```

- TO_INTEGER is the only date/time function that can be used in arithmetic expressions.
- When two primaries have the same data type, the result is of that data type. For example, when an INTEGER is divided by an INTEGER, the result is INTEGER. In such cases, the result will be truncated.
- If either arithmetic operand is the NULL value, then the result is the NULL value.
- Arithmetic operators cannot be used to concatenate string values. Use `||` to concatenate string operands.
- Both operands of concatenation operator should be one of the following: CHAR (or VARCHAR, or Native CHAR, or Native VARCHAR), BINARY (or VARBINARY), but no mix of CHAR and BINARY.
- If either concatenation operand is the NULL value, then the result of the concatenation is the NULL value.
- If one concatenation operand is a variable length string (VARCHAR, Native VARCHAR, VARBINARY), then the result data type of the concatenation is a variable length string.
- If both concatenation operands are fixed length string data type (CHAR, Native CHAR, BINARY), then the result of the concatenation is fixed length string.
- The concatenation result will consist of the first operand followed by the second operand. The trailing blanks of the string value are preserved by concatenation regardless of the string's data types. The resultant string may be truncated on the right, if the length exceeds the maximum string length of 3996 bytes. If truncation occurs, a truncation warning is sent.
- Type conversion, truncation, underflow, or overflow can occur when some expressions

are evaluated. For more information, refer to the chapter, "Data Types."

- If the value of an indicator variable is less than zero, the value of the corresponding host variable is considered to be NULL.

NOTE To be consistent with the standard SQL and to support portability of code, it is strongly recommended that you use a –1 to indicate a NULL value. However, ALLBASE/SQL interprets all negative indicator variable values as indicating a NULL value in the corresponding host variable.

- The following expressions can evaluate to NULL:
  - — Host variable with an indicator variable
  - — Local variable
  - — Procedure parameter
  - — Column
  - — Add Months function
  - — DateTime function
  - — Aggregate function
  - — CAST function
  - — String function
- A NULL value in an expression causes comparison operators and other predicates to evaluate to unknown. Refer to Chapter 9 , “Search Conditions,” for more information on evaluation of comparison operators and predicates containing NULL values.
- The ? can be used as a host variable or dynamic parameter in an expression as shown in the following examples:
  - — In the WHERE clause of any `SELECT` statement:

```
SELECT *
  FROM PurchDB.Orders
 WHERE PartNumber = ?
       AND OrderDate > ?
ORDER BY OrderDate
```

  - — In the WHERE and SET clauses of an `UPDATE` statement:

```
UPDATE PurchDB.Parts
   SET SalesPrice = ?
 WHERE PartNumber = ?
```

  - — In the WHERE clause of a `DELETE` statement:

```
DELETE FROM PurchDB.OrderItems
     WHERE ItemDueDate
    BETWEEN ? and ?
```

  - — In the VALUES clause of an `INSERT` or a `BULK INSERT` statement. In this example each ? corresponds in sequential order to a column in the PurchDB.OrderItems

table:

```
BULK INSERT INTO PurchDB.OrderItems VALUES (?,?,?,?)
```

See the syntax descriptions for each DML statement, and for the PREPARE, DESCRIBE, EXECUTE, and OPEN statements for details of dynamic parameter usage.

Example

The result length of PartNumber || VendPartNumber is 32 in this example.

```
CREATE TABLE PurchDB.SupplyPrice
       (Part Number     CHAR(16) NOT CASE SENSITVE not null unique,
        VendorNumber    INTEGER
        VendPartNumber  CHAR(16) lang=german,
        UnitPrice       DECIMAL (10,2),
        Delivery Days   SMALLINT,
        DiscountQty     SMALLINT)

SELECT PartNumber || VendPartNumber, UnitPrice from PurchDB.SupplyPrice;
```

Add Months Function

The Add Months function uses the keyword ADD_MONTHS to apply the addition operation to a DATE or DATETIME expression. It is different from a simple addition operator in that it adjusts the day field in the DATE or DATETIME value to the last day of the month if adding the months creates an invalid date (such as '1989-02-30').

Scope

SQL Data Manipulation Statements

SQL Syntax

```
ADD_MONTHS (DateExpression, {[+
                              -]IntegerValue
                             :HostVariable [[INDICATOR]:IndicatorVariable]
                             ?
                             :LocalVariable
                             :ProcedureParameter                          })
```

Parameters

DateExpression is either a DATE or DATETIME expression. See the "Expression" section of this chapter for details on the syntax.

HostVariable is a host variable of type INTEGER. It can be positive or negative. If negative, the absolute value is subtracted from *Value1*.

IndicatorVariable names an indicator variable, whose value determines whether the associated host variable contains a NULL value:

> = 0

the value is not NULL

< 0

the value is NULL (The value in the host variable will be ignored.)

? indicates a dynamic parameter in a prepared SQL statement. The value of the parameter is supplied when the statement is executed.

LocalVariable contains a value within a procedure.

ProcedureParameter contains a value that is passed into or out of a procedure.

Description

- The Add Months function adds a duration of months to a DATE or DATETIME expression. Only the month portion of the value is affected, and, if necessary, the year portion. The day portion of the date is unchanged unless the result would be invalid (for

example, '1989-02-31'). In this case, the day is set to the last day of the month for that year, and ALLBASE/SQL generates a warning indicating the adjustment.

- If either parameter is NULL, ADD_MONTHS will evaluate to NULL also.

Example

In this example, rows are returned which comprise the batch stamp and test date that have a pass quantity less than 48. A warning is generated because 7 months added to the '1984-07-30' date results in an invalid date, '1985-02-30'.

```
SELECT BatchStamp, ADD_MONTHS(TestDate,7)
  FROM ManufDB.TestData
 WHERE PassQty <= 48

ADD_MONTHS result adjusted to last day of month.  (DBWARN 2042)
```

Aggregate Functions

Aggregate functions specify a value computed using data described in an argument. The argument, enclosed in parentheses, is an expression. The value of the expression is computed using each row that satisfies a SELECT statement. Aggregate functions can be specified in the select list and the HAVING clause. Refer to the explanation of the SELECT statement for more details.

Scope

SQL SELECT Statements

SQL Syntax

```
{ AVG   ({ Expression
           [ALL
            DISTINCT] ColumnName} )
  MAX   ({ Expression
           [ALL
            DISTINCT] ColumnName} )
  MIN   ({ Expression
           [ALL
            DISTINCT] ColumnName} )
  SUM   ({ Expression
           [ALL
            DISTINCT] ColumnName} )
  COUNT ({ *
           [ALL
            DISTINCT] ColumnName} )  }
```

Parameters

Expression specifies a value to be obtained.

AVG computes the arithmetic mean of the values in the argument; NULL values are ignored. AVG can be applied only to numeric data types and to the INTERVAL type. When applied to FLOAT or REAL, the result is FLOAT. When applied to INTEGER or SMALLINT, the result is INTEGER, and fractions are discarded. When applied to DECIMAL, the result is DECIMAL. When applied to INTERVAL, the result is INTERVAL.

MAX finds the largest of the values in the argument; NULL values are ignored. MAX can be applied to numeric, alphanumeric, BINARY (not LONG), and date/time data types; the result is the same data type as that of the argument.

MIN finds the smallest of the values in the argument; NULL values are ignored. MIN can be applied to numeric, alphanumeric, BINARY (not LONG), and date/time data types; the result is the same data type as that of the argument.

SUM	finds the total of all values in the argument. NULL values are ignored. SUM can be applied to numeric data types and INTERVAL only. When applied to FLOAT or REAL, the result is FLOAT. When applied to INTEGER or SMALLINT, the result is INTEGER. When applied to DECIMAL, the result is DECIMAL. When applied to INTERVAL, the result is INTERVAL.
COUNT *	counts all rows in all columns, including rows containing NULL values. The result is INTEGER.
COUNT *ColumnName*	counts all rows in a specific column; rows containing NULL values are not counted. The data type of the column cannot be LONG BINARY or LONG VARBINARY. The result is INTEGER.
ALL	includes any duplicate rows in the argument of an aggregate function. If neither ALL nor DISTINCT is specified, ALL is assumed.
DISTINCT	eliminates duplicate column values from the argument of an aggregate function.

Description

- If an aggregate function is computed over an empty, ungrouped table, results are as follows:

  — COUNT returns 1; SQLCODE equals 0.

  — AVG, SUM, MAX, and MIN return NULL; SQLCODE equals 0.

- If an aggregate function is computed over an empty group or an empty grouped table, all aggregate functions return no row at all.
- Refer to the "Data Types" chapter for information on truncation and type conversion that may occur during the evaluation of aggregate functions.
- Refer to the "Data Types" chapter for information on the resulting precision and scale of aggregate functions involving DECIMAL arguments.
- A warning message is returned if a NULL is removed from the computation of an aggregate function.

Example

The average price of each part with more than five rows in table PurchDB.SupplyPrice is calculated.

```
    SELECT PartNumber, AVG(UnitPrice)
      FROM PurchDB.SupplyPrice
GROUP BY PartNumber
  HAVING COUNT * > 5
```

CAST Function

The CAST function converts data from one data type to another. The CAST function can be used anywhere a general expression is allowed. CAST is supported inside functions that support expressions including aggregate functions. CAST also takes general expressions including nested functions as input.

Scope

SQL Data Manipulation Statements

SQL Syntax

```
{ CAST ( { Expression
           NULL} { AS
                    ,} DataType [,FormatSpec]) }
```

Parameters

Expression is the value to be converted. See the "Expression" section in this chapter for details on the syntax.

DataType ALLBASE/SQL data type: CHAR(n), VARCHAR(n), DECIMAL(p[,s]), FLOAT, REAL, INTEGER, SMALLINT, DATE, TIME, DATETIME, INTERVAL, BINARY(n), VARBINARY(n), TID.

The LONG BINARY(n) and LONG VARBINARY(n) cannot be used in the CAST operations.

FormatSpec Format specification used for DATE, TIME, DATETIME, INTERVAL conversions. *FormatSpec* is the same as that used in the date/time conversion functions.

Description

The following table shows what data type conversions the CAST function supports. These are the status codes used in the table:

- Y—is supported
- N—is not supported
- E—is an ALLBASE/SQL Extension (not a part of ANSI standard)

Source Data Type	Target Data Type										
	EN[a]	AN[b]	VC	CHAR (n)	B	VB	DATE	TIME	DT	I	TID
ENa	Y[c]	Yc	Y[d]	Yd	Ed	Ed	Nd	N	N	N	N
ANb	Yc	Yc	Yd	Yd	Ed	Ed	N	N	N	N	N
VARCHAR(n)	Yd	Yd	Yc	Yc	Yc	Yc	Yc	Yc	Yc	Yc	Ed
CHAR(n)	Yd	Yd	Yc	Yc	Yc	Yc	Yc	Yc	Yc	Yc	Ed
BINARY	Ed	Ed	Yc	Yc	Yc	Yc	Ed	Ed	Ed	Ed	Ed
VARBINARY(n)	Ed	Ed	Yc	Yc	Yc	Yc	Ed	Ed	Ed	Ed	Ed
DATE	Ec	Ec	Yc	Yc	Ed	Ed	Yc	N	N	N	N
TIME	Ec	Ec	Yc	Yc	Ed	Ed	N	Yc	N	N	N
DATETIME	Ec	Ec	Yc	Yc	Ed	Ed	N	N	Yc	N	N
INTERVAL	Yc	Ec	Yc	Yc	Ed	Ed	N	N	N	Yc	N
TID	N	N	Ed	Ed	Ed	Ed	N	N	N	N	Yc

a. EN—Exact Numeric (SMALLINT, INT[EGER], DEC[IMAL][(p[,s])], NUMERIC[(p[,s])])
b. AN—Approximate Numeric (FLOAT[(p)] or DOUBLE PRECISION, REAL)
c. Implicit conversion also supported
d. Conversion supported only with CAST

- If input to CAST is NULL, then the result of the CAST operation is NULL.
- ALLBASE/SQL supports implicit data conversion between:
  - Numeric data types to numeric data types
  - Character data types to character data types
  - Binary data types to binary data types
  - Binary data types to character data types
  - Character data types to binary data types

  When CAST is used to do these conversions, all existing rules are applied.
- When a number is converted, if the number does not fit within the target precision, an overflow error occurs.
- When converting from an approximate numeric to an exact numeric or from an exact numeric to an exact numeric with less scale (integers have a scale of 0), the extra digits of scale beyond the target scale are dropped without rounding the result.

- If both source and target data type are character strings, the language of the result string is the same as the source.
- If the source data type is a character string and the target data type is a numeric, then the source value must only contain a character representation of a number. The result of the conversion is the numeric value that string represented.

  If the source value is not a numeric string, an error occurs.
- If the target data type is CHAR(n), and the source data type is an exact numeric, the result is a character representation of that exact numeric. If the source value is less than zero, the first character of the result is a minus sign. Otherwise, the first character is a number or a decimal point.

  If the length of the resulted string is less than n, then blanks are added on the right. If the length of the resulted string is greater than n, an error occurs. The same algorithm applies if the target data type is VARCHAR(n), except that there is no need to pad the numeric string if its length is less than n.
- If the target data type is CHAR(n) and the source data type is an approximate numeric, then the number is converted to a character representation in scientific notation.

  If the length of the resulted string is less than n, then blanks are added on the right. If the length of the resulted string is greater than n, then an error occurs. The same algorithm applies if the target data type is VARCHAR(n), except that there is no need to pad the numeric string if its length is less than n.
- Conversion between character and binary data types is supported implicity as well as with CAST. The same rules still apply with CAST. If a target is shorter than the source, truncation occurs. If the target is larger than the source, the target is zero-filled in the case of BINARY(n), and blank-filled in the case of CHAR(n).
- When converting a non-character data type to BINARY(n) or VARBINARY(n), the data is not modified. Only the type changes so that the data is treated as binary data. The size of the source and the target in bytes must be equal in the case of BINARY(n), and the size of the source must be less than or equal to the size of the target in the case of VARBINARY(n). Otherwise, an error occurs.

  For decimal numbers, each digit of precision contributes 4 bits and 4 bits for the sign. The overall size is rounded up to a 4-byte boundary. The storage size for DATE, TIME, DATETIME, and INTERVAL is 16 bytes.
- When converting from BINARY(n) or VARBINARY(n) into a non-character data type, the data is not modified. Only the type changes so that the data is treated as a number of the target data type. The actual size of the source and the target in bytes must be equal, or an error occurs.
- Conversion between binary data types and numeric data types is an ALLBASE extension and is not allowed according to the ANSI SQL2 standard.
- Converting a character string to a DATE, TIME, DATETIME or INTERVAL with CAST is equivalent to using the respective date/time function, TO_DATE, TO_TIME, TO_DATETIME, or TO_INTERVAL. All the same rules apply.
- Using CAST to convert numeric types directly to date/time types is not allowed. This

should be done by nesting the CAST functions so that the numeric value is first converted to a character string, and then converted to the date/time data type.

- Converting a date/time data type to:
  - A character type with CAST is equivalent to using the TO_CHAR date/time function. All the same rules apply.
  - An INTEGER is equivalent to using the TO_INTEGER date/time function. This function converts date/time column value into an INTEGER value which represents a portion of the date/time column. If the source data type of CAST is date/time data type, and the target data type is INTEGER, all rules for TO_INTEGER to convert date/time into INTEGER will be applied. The *FormatSpec* must be used to specify a single component of the date/time data type (i.e. HH, MM, SS, DAYS, etc.).
  - Other numeric types are also allowed using CAST. In this case, the date/time data type is first converted to an INTEGER applying all the TO_INTEGER rules, then is converted from INTEGER to the target data type.

Examples

1. You will see the result has VendorNumber presented as: *Vendor9000, Vendor9020, ...*

```
CREATE TABLE PurchDB.SupplyPrice
  ( PartNumber     CHAR(16) NOT CASE SENSITIVE not null unique,
    VendorNumber   INTEGER,
    VendPartNumber CHAR(16) lang=german NOT CASE SENSITIVE,
    UnitPrice      DECIMAL(10,2),
    DeliveryDays   CHAR(2),
    DiscountQty    SMALLINT)

SELECT PartNumber, 'Vendor' || CAST(VendorNumber AS VARCHAR(4))
FROM PurchDB.SupplyPrice
 WHERE VendorNumber BETWEEN 9000 AND 9020;
```

2. You will see the INTERVAL constant shown as: *0 23:00:00:000*

```
SELECT PartNumber, CAST(CAST(23,CHAR(2)),INTERVAL,'HH')
FROM PurchDB.SupplyPrice;
```

3. You will see the INTEGER constant shown as: *99*

```
SELECT PartNumber, CAST('9999-12-31',INTEGER,'CC')
FROM PurchDB.SupplyPrice;
```

4. SELECT SUM with CAST

```
SELECT SUM(CAST(DeliveryDays, SMALLINT))
FROM PurchDB.SupplyPrice
WHERE VendorNumber BETWEEN 9000 AND 9020;
```

5. EXEC SQL with CAST

```
EXEC SQL begin declare section;
   char hostvar1[16];
   sqlbinary hostvar2[8];
EXEC SQL end declare section;
```

Assume there is only one row qualified for the following query.

```
EXEC SQL select PartNumber, CAST(UnitPrice,BINARY(8))
INTO :hostvar1, :hostvar2
FROM PurchDB.SupplyPrice
WHERE VendorNumber BETWEEN 9000 AND 9020;
```

6. You will see the DECIMAL constant shown as: *99.99*

```
SELECT PartNumber, CAST(99.99,VARCHAR(10))
FROM PurchDB.SupplyPrice;
```

Constant

A constant is a specific numeric, character, or hexadecimal value.

Scope

SQL Data Manipulation Statements

SQL Syntax

```
{ IntegerValue
  FloatValue
  FixedPointValue
  'CharacterString'
  0xHexadecimalString}
```

Parameters

IntegerValue is a signed or unsigned whole number compatible with INTEGER or SMALLINT data types, for example:

```
-16746
   155
     5
```

FloatValue is a signed or unsigned floating point number compatible with the FLOAT or REAL data types, for example:

```
.2E-4
```

FixedPointValue is a signed or unsigned fixed-point number compatible with the DECIMAL data type, for example:

```
  -15.99
+1451.1
```

CharacterString is a character string compatible with CHAR, VARCHAR, DATE, TIME, DATETIME, or INTERVAL data types. String constants are delimited by single quotation marks, for example:

```
'DON''T JUMP!'
```

However, two single quotation marks in a row are interpreted as a single quotation mark, not as string delimiters.

HexadecimalString is a string of hexadecimal digits 0 through 9 and A through F (the lowercase a through f are also accepted) compatible with the BINARY and VARBINARY data types. A *HexadecimalString* constant must be prefaced with the characters 0x, for example:

```
0xFFFAB0880088343330FFAA7
0x000V001231
```

Current Functions

Current functions return a value that represents a current DATE, TIME, or DATETIME. The value returned is a string with the format of a DATE, TIME, or DATETIME data type.

Scope

SQL Data Manipulation Statements

SQL Syntax

```
{CURRENT_DATE
 CURRENT_TIME
 CURRENT_DATETIME}
```

Description

- CURRENT_DATE returns the current date as a string of the form 'YYYY-MM-DD', where YYYY represents the year, MM is the month, and DD is the day.
- CURRENT_TIME returns the current time as a string of the form 'HH:MI:SS', where HH represents hours, MI is minutes, and SS is seconds.
- CURRENT_DATETIME returns the current date and time as a string of the form 'YYYY-MM-DD HH:MI:SS.FFF', where YYYY represents the year, MM is the month, DD is the day, HH represents the hours, MI the minutes, SS the seconds, and FFF the thousandths of a second.

Examples

Set a column to the current DATE.

```
UPDATE ManufDB.TestData
   SET TestDate = CURRENT_DATE
 WHERE BatchStamp = '1984-07-25 10:15:58.159'
```

Set a column to the current DATETIME.

```
UPDATE ManufDB.SupplyBatches
   SET BatchStamp = CURRENT_DATETIME
 WHERE BatchStamp = '1984-07-25 10:15:58.159'
```

Date/Time Functions

The following text describes the two types of date/time conversion functions:

- The input functions convert character values into date/time values. With TO_DATE, TO_TIME, TO_DATETIME, and TO_INTERVAL you can enter date/time values in a format other than the default format.
- The output functions convert date/time values out to integer or character values. With TO_CHAR you can specify an output format for a date/time column value other than the default format. With TO_INTEGER you can extract an element as an INTEGER value.

Date/time columns are displayed in the default format.

Scope

SQL Data Manipulation Statements

SQL Syntax—Conversion Functions

```
{ { TO_DATE
    TO_TIME
    TO_DATETIME
    TO_INTERVAL } (StringExpression [,FormatSpecification])
  TO_CHAR (DateTimeExpression [,FormatSpecification])
  TO_INTEGER (DateTimeExpression ,FormatSpecification)       }
```

Parameters—Conversion Functions

TO_DATE, TO_TIME, TO_DATETIME, TO_INTERVAL — produce a result which is of the DATE, TIME, DATETIME, or INTERVAL type, respectively. Use these functions in any expression.

TO_CHAR — produces the character string representation of the value in the column named in the first parameter in the format specified in the second parameter. The result type is VARCHAR with the length as specified by the format specification. If a format is not specified, the default format for the data type (and length) is used. Use this output function in any expression.

TO_INTEGER — produces an INTEGER value which represents a portion of the date/time column. The format specification is not optional in this case, and must consist of a single element (of the format specification). Use this output function in any expression.

StringExpression — is a string expression. Refer to the "Expression" section in this chapter for details on the syntax. The expression must

be a CHAR or VARCHAR data type.

DateTimeExpression is a Date/Time expression. See the "Expression" section of this chapter for more details on the syntax. The expression must be a DATE, TIME, DATETIME, or INTERVAL data type.

FormatSpecification specifies the format of *ColumnName* or *CharacterValue*. Refer to the syntax for *FormatSpecification* later in this section. Format elements are presented in the "Description" section below.

SQL Syntax—FormatSpecification

```
{'FormatString'
  :HostVariable [[INDICATOR]:IndicatorVariable]
  ?
  :LocalVariable
  :ProcedureParameter
  ::Built-inVariable                              }
```

Parameters—FormatSpecification

FormatString is a character string literal representing the format of *DateTimeExpression* or *StringExpression*. It must be a string literal, of maximum length 72 characters. Format is composed of one or more elements. Available format elements for the date/time data types are described below. Only n-computer characters are allowed in the *FormatString*. The syntax for the format string follows:

{ *FormatElement* {*Punctuation or Blank*} [...] }

The format elements are listed in the "Description" section.

HostVariable identifies a host variable that contains the format specification which determines how the *DateTimeExpression* or *StringExpression* is to be converted.

IndicatorVariable names an indicator variable, whose value determines whether the associated host variable contains a NULL value:

> = 0

the value is not NULL

< 0

the value is NULL (The value in the host variable will be ignored.)

? is a place holder for a dynamic parameter in a prepared SQL statement in an application program. The value of the dynamic parameter is supplied at run time.

LocalVariable contains a value in a procedure.

ProcedureParameter contains a value that is passed into or out of a procedure.

::*Built-inVariable* is one of the following built-in variables used for error handling:

- ::sqlcode
- ::sqlerrd2
- ::sqlwarn0
- ::sqlwarn1
- ::sqlwarn2
- ::sqlwarn6
- ::activexact

The first six of these have the same meaning that they have as fields in the SQLCA in application programs. ::activexact indicates whether a transaction is in progress or not. For additional information, refer to the application programming guides and to Chapter 4 , “Constraints, Procedures, and Rules.”

Description

- If the format specification is optional and it is not supplied, the proper default format is used. If a date/time column or string literal appears in an expression without a conversion function, it is changed, if necessary, to the default format.
- Date format is used by the TO_DATE function and by the TO_CHAR function on DATE expressions. The default format is 'YYYY-MM-DD'.

  Listed here are format elements made up of numeric characters (digits 0 through 9):

CC	Century (00 to 99)
YYYY	Year (0000 to 9999)
YY	Year of century (00 to 99)
ZYY	YY with leading zeroes suppressed (0 to 99) (TO_CHAR only)
Q	Quarter (1 to 4) (TO_CHAR only)
MM	Month (01 to 12)
ZMM	MM with leading zeroes suppressed (1 to 12) (TO_CHAR only)
DAYS	Days since January 1, 0000 (0000000 to 3652436)
ZDAYS	DAYS with leading zeroes suppressed (0 to 3652436) (TO_CHAR only)
DDD	Day of year (001 to 366)
ZDDD	DDD with leading zeroes suppressed (1 to 366) (TO_CHAR only)
DD	Day of month (01 to 31)
ZDD	DD with leading zeroes suppressed (1 to 31) (TO_CHAR only)
D	Day of week (1 to 7) (TO_CHAR only)

The Z prefix and Q and D are only allowed for the function TO_CHAR. If YY is used without CC, the default CC is 19. The following elements are for representing alphabetic characters:

MONTH	Name of month
MON	Abbreviated name of month
DAYOFWEEK	Name of day
DAY	Abbreviated name of day
-/:.,	Punctuation marks reproduced in value (includes spaces)
"string"	Quoted string reproduced in value

Delimiting punctuation marks must be the same in the value parameter and the format specification parameter.

- Capitalization in alphabetic representations follows the capitalization of the corresponding format element. Elements may be represented in uppercase, lowercase, or initial caps. Other mixtures of uppercase and lowercase letters result in an error. For example:

```
'DAYOFWEEK' —--> MONDAY
'Dayofweek' —--> Monday
'dayofweek' —--> monday
'dAyOfWeEk' —--> error condition
```

- Time format is used by the TO_TIME function and by the TO_CHAR functions on TIME expressions. The default format is 'HH:MI:SS'.

  Listed here are formats for elements made up of numeric characters:

HH or HH24	Hour of day (00 to 23)
ZHH or ZHH24	HH or HH24 with leading zeroes suppressed (0 to 23) (TO_CHAR only)
HH12	Hour of day (00 to 12)
ZHH12	HH12 with leading zeroes suppressed (0 to 12) (TO_CHAR only)
MI	Minute (00 to 59)
ZMI	MI with leading zeroes suppressed (0 to 59) (TO_CHAR only)
SS	Second (00 to 59)
ZSS	SS with leading zeroes suppressed (0 to 59) (TO_CHAR only)
SECONDS	Seconds past midnight (00000 to 86399)
ZSECONDS	SECONDS with leading zeroes suppressed (0 to 86399) (TO_CHAR only)

  Z is not allowed for the input functions. The following elements are for representing alphabetic characters:

AM or PM	AM/PM indicator (use capital letters)
A.M. or P.M.	A.M./P.M. indicator with periods (use capital letters)

./:.,	Punctuation marks reproduced in value (includes spaces)
"string"	Quoted string reproduced in value

Delimiting punctuation marks must be the same in the value parameter and the format specification parameter.

- The TO_DATETIME function and the TO_CHAR function on TIME expressions use the date/time default format 'YYYY-MM-DD HH:MI:SS.FFF'.

  In addition to all formats shown for the date and time format specifications above, the following are also allowed for date/time formats (made up of the numeric characters 0 through 9):

F	Tenth of a second (.0 to .9)
FF	Hundredth of a second (.00 to .99)
FFF	Thousandth of a second (.000 to .999)

- The TO_INTERVAL function and the TO_CHAR function on INTERVAL expressions use the interval default format 'DAYS HH:MI:SS.FFF'.

  The following formats are allowed in an interval format specification:

```
DAYS            MI            SECONDS          FFF
ZDAYS           ZMI           ZSECONDS         -/:.,
HH or HH24      SS            F                "string"
ZHH or ZHH24    ZSS           FF
```

  These were described in the TIME and DATETIME format specifications above.

- Literals for date/time data types which do not specify all elements of the date/time value are expanded and filled as described below:
  - INTERVAL is zero filled on the left and the right.
  - DATE, TIME, and DATETIME are left-filled with the current values from the system clock, and right-filled with appropriate portions of the default '0000-01-01 00:00:00.000'.
- When YY is specified in the *FormatSpecification* and if its value in *StringExpression* is less than 50, then the century part of DATE and DATETIME defaults to 20, else it is set to 19. This behavior can be overridden by setting the environment variable HPSQLsplitcentury to a value between 0 and 100. If the YY part is less than the value of environment variable HPSQLsplitcentury then the century part is set to 20, else it is set to 19.
- Output values are truncated, not rounded, to fit in the specified format.
- The TO_INTEGER format specification is *not* optional, and must consist of one of the following single elements *only*:

```
CC          MM          DAYS          SS
YYYY        DDD         HH or HH24   SECONDS
YY          DD          HH12          F, FF, or FFF
Q           D           MI
```

- ADD_MONTHS is a related function. ADD_MONTHS adds a duration of months to a

DATE or DATETIME column. Refer to the Add Months Function for further information.

Examples

1. Date format

   In the example below, the format MM/DD/YY is used to enter a date instead of using the default format, which is YYYY-MM-DD:

```
INSERT INTO ManufDB.TestData(batchstamp, testdate)
    VALUES (TO_DATETIME ('07/02/89 03:20.000', 'MM/DD/YY HH12:MI.FFF'),
             TO_DATE('10/02/84','MM/DD/YY'))
```

   To return the date entered in the above example, in a format other than the default format, the desired format is specified in the second parameter of the TO_CHAR conversion function:

```
SELECT TO_CHAR(testdate, 'Dayofweek, Month DD')
    FROM ManufDB.TestData
 WHERE labtime < '0 05:00:00.000'
```

   The value "Friday, July 13" is selected from TestData.

   The following statement inserts different date values depending on the value of the environment variable HPSQLsplitcentury, if it is set.

```
INSERT INTO ManufDB.TestData(testdata)
        VALUES (TO_DATE ('30/10','YY/MM'))
```

   Case 1: HPSQLsplitcentury is not set; inserts 2030-10-01

   Case 2: HPSQLsplitcentury is set to 0; inserts 1930-10-01

   Case 3: HPSQLsplitcentury is set to 70; inserts 2030-10-01

2. Time format

```
INSERT INTO ManufDB.TestData(teststart, batchstamp)
    VALUES (TO_TIME('01:53 a.m.','HH12:MI a.m.'),
            TO_DATETIME('12.01.84 02.12 AM', 'DD.MM.YY HH12.MI AM'))
```

3. Datetime format

```
UPDATE ManufDB.TestData
   SET batchstamp = TO_DATETIME('12.01.84 02.12 AM', 'DD.MM.YY HH12.MI AM')
 WHERE batchstamp = TO_DATETIME('11.01.84 1.11 PM',  'DD.MM.YY HH12.MI PM')
```

4. Interval format

```
UPDATE ManufDB.TestData
   SET labtime = TO_INTERVAL('06 10:12:11.111', 'DAYS HH:MI:SS.FFF')
 WHERE testdate = TO_DATE('10.02.84','MM.DD.YY')
```

Long Column Functions

Long column functions return information from the long column descriptor.

Scope

SQL Data Manipulation Statements

SQL Syntax

```
{ OUTPUT_DEVICE(LongColumnName)
  OUTPUT_NAME(LongColumnName)    }
```

Parameters

`OUTPUT_DEVICE` returns an integer value indicating the output device type stored in the long column descriptor for *`LongColumnName`*. The values returned are shown in the table below:

Value Returned	Output Device Type
0	none specified
1	system file
2	shared memory

`OUTPUT_NAME` returns the output device name stored in the long column descriptor for *`LongColumnName`*. The string returned is a 44 byte value.

`LongColumnName` is the name of the column that has a long data type (LONG BINARY or LONG VARBINARY).

Description

- The long column functions can appear in the select list or search condition of an SQL data manipulation statement.
- The long column functions are useful when you need information about the long column descriptors, but do not want to fetch the data.
- For more information on long column data types, see the "Data Types" chapter.
- Referencing a LONG column in a LONG column function does not cause the LONG data to be written out to the output device.

Examples

1. OUTPUT_DEVICE example

   Change the PartPicture output device name to NewHammer in any row whose output device type for PartPicture is a system file.

```
UPDATE PartsIllus
   SET PartPicture = '> NewHammer'
 WHERE OUTPUT_DEVICE(PartPicture) = 1
```

2. OUTPUT_NAME example

   Select the output device name of the PartPicture column for any row with a PartNumber of 100.

```
SELECT OUTPUT_NAME(PartPicture)
  FROM PartsIllus
 WHERE PartNumber = 100
```

   Change all occurrences of the output device name of the PartPicture column to NewHammer if the current output device name is Hammer.

```
UPDATE PartsIllus
   SET PartPicture = '> NewHammer'
 WHERE OUTPUT_NAME(PartPicture) = 'Hammer'
```

String Functions

String functions return partial values or attributes of character and BINARY (including LONG) string data.

With the G3 release of ALLBASE/SQL and IMAGE/SQL, the supported SQL syntax has been enhanced to include the following string manipulation functions: UPPER, LOWER, POSITION, INSTR, TRIM, LTRIM and RTRIM. These string functions allow you to manipulate or examine the CHAR and VARCHAR values within the SQL syntax, allowing for more sophisticated queries and data manipulation commands to be formed. These string functions were designed to be compatible with functions specified in the ANSI SQL '92 standard and functions used in ORACLE. In cases where the ANSI SQL '92 standard and the ORACLE functions were not compatible (such as the LTRIM and RTRIM in ORACLE versus TRIM in the ANSI standard), both versions were implemented. The specifications for each of these functions follows.

Function Specification

LOWER

Converts all the characters in *stringexpr* to lower case

Syntax [**LOWER** (*stringexpr*)]

UPPER

Converts all the characters in *stringexpr* to upper case

Syntax [**UPPER** (*stringexpr*)]

POSITION

Searches for the presence of the string *stringexpr1* in the string *stringexpr2* and returns a numeric value that indicates the position at which *stringexpr1* is found in *stringexpr2*

Syntax [**POSITION** (*stringexpr*,*stringexpr2*)]

INSTR

Searches *stringexpr1* beginning with its *n*th character for the *m*th occurrence of *stringexpr2* and returns the position of the character in *stringexpr1* that is the first character of this occurrence. If *n* is negative, Instr counts and searches backward from the end of *stringexpr1*. The value of *m* must be positive. The default values of both *n* and *m* are 1, meaning Instr begins searching at the first character of *stringexpr1* for the first occurrence of *stringexpr2*. The return value is relative to the beginning of *stringexpr1* regardless of the value of *n*, and is expressed in characters. If the search is unsuccessful (if *stringexpr2* does not appear *m* times after the *n*th character of *stringexpr1*) the return value is 0.

If *n* and *m* are not specified the function is equivalent to the ANSI SQL-92 POSITION

function, except that the syntax is slightly different.

Syntax [**INST** (*stringexpr1*,*stringexpr2* [,*n*[,*m*]])]

LTRIM

LTRIM function trims the characters specified in *charset* from the beginning of the string *stringexpr*.

Syntax [**LTRIM** (*charset*,*stringexpr*)]

RTRIM

RTRIM function trims the characters specified in *charset* from the end of the string *stringexpr*.

Syntax [**RTRIM** (*charset*,*stringexpr*)]

TRIM

TRIM function allows you to strip the characters specified in *charset* from the beginning and/or the end of the string *stringexpr*. If *charset* is not specified, then blank characters would be stripped from *stringexpr*.

Syntax
[**TRIM** ({ **LEADING** | **TRAILING** | **BOTH**} (,*charset* ,*stringexpr*)]

Examples:

Example 1

```
SELECT LOWER (OWNER) || '.' || LOWER (NAME)
   FROM SYSTEM.TABLE
  WHERE NAME = UPPER ('vendors');
```

Returns "purchdb .vendors "

Example 2

```
SELECT POSITION ('world', 'hello world')
   FROM SYSTEM.TABLE
  WHERE NAME = UPPER('vendors');
```

Returns the numeric value 7

Example 3

```
SELECT INSTR ('hello world hello world', 'world', 5, 2)
   FROM SYSTEM.TABLE
  WHERE NAME = UPPER('vendors');
```

Returns the numeric value 18 (starting position of the second occurrence of the string 'world').

Example 4

```
SELECT * FROM SYSTEM.TABLE
   WHERE NAME = LTRIM ('?*', 'VENDORS?*???***')
   AND OWNER = 'PURCHDB';
```

Returns the system table entry for PURCHDB.VENDORS

Example 5

```
SELECT TRIM (BOTH '?*' FROM '??**?*hello ?* world???*')
   FROM SYSTEM.TABLE
  WHERE NAME = 'VENDORS';
```

Returns 'hello ?* world'.

Scope

SQL Data Manipulation Statements

SQL Syntax

```
{ STRING_LENGTH (StringExpression)
  SUBSTRING (StringExpression,StartPosition,Length)}
```

Parameters

STRING_LENGTH returns an integer indicating the length of the parameter. If *StringExpression* is a fixed length string type, STRING_LENGTH will return the fixed length. If *StringExpression* is a variable length string, the actual length of the string will be returned.

StringExpression is an expression of a string type. See the "Expression" section in this chapter for the syntax. The expression must be a CHAR, VARCHAR, BINARY, VARBINARY, Long Binary, or Long VARBINARY data type.

For example, the following are acceptable:

```
VendorName
'Applied Analysis'
SUBSTRING(VendorName,1,10)
```

SUBSTRING returns the portion of the *SourceString* parameter which begins at *StartPosition* and is *Length* bytes long.

StartPosition is an integer constant or expression. See the "Expression" section in this chapter for this syntax.

Length is an integer constant or expression. See the "Expression" section in this chapter for this syntax. The following are examples of acceptable lengths:

```
5
STRING_LENGTH(VendorName)-28
```

Description

- The string functions can appear in an expression, a select list, or a search condition of an SQL data manipulation statement.
- The string functions can be applied to any string data type, including binary and long column data types.
- The string returned by the SUBSTRING function is truncated if (*StartPosition* + *Length* -1) is greater than the length of the *StringExpression*. Only (*Length* - *StartPosition* +1) bytes is returned, and a warning is issued.
- If *Length* is a simple constant, the substring returned has a maximum length equal to the value of the constant. Otherwise, the length and data type returned by the SUBSTRING function depend on the data type of *StringExpression*, as shown in the following table:

Table 8-1. Data Type Returned by SUBSTRING

StringExpression Data Type	SUBSTRING Data Type	SUBSTRING Maximum Length
CHAR	VARCHAR	fixed length of *SourceString*
VARCHAR	VARCHAR	maximum length of *SourceString*
BINARY	VARBINARY	fixed length of *SourceString*
VARBINARY	VARBINARY	maximum length of *SourceString*
LONG BINARY	VARBINARY	3996 [a]
LONG VARBINARY	VARBINARY	3996a

a. 3996 is the maximum length of a VARBINARY data type

Examples

1. STRING_LENGTH example

   In the SELECT statement below, the PartsIllus table is searched for any row whose PartPicture contains more than 10000 bytes of data, and whose PartName is longer than 10 bytes.

```
CREATE TABLE PartsIllus
            (PartName  VARCHAR(16),
            PartNumber  INTEGER,
            PartPicture  LONG VARBINARY(1000000) in PartPictureSet)
         IN PartsIllusSet
SELECT PartNumber, PartName
  FROM PartsIllus
 WHERE STRING_LENGTH(PartPicture) > 10000
   AND STRING_LENGTH(PartName)    > 10
```

2. SUBSTRING example

   For every row in PartsIllus, the PartNumber and the first 350 bytes of the PartPicture are inserted into the DataBank table:

```
CREATE TABLE DataBank
            (IdNumber  INTEGER,
            Data  VARBINARY(1000))

INSERT INTO DataBank
     SELECT PartNumber, SUBSTRING(PartPicture,1,350)
       FROM PartsIllus
```

   Display a substring of the PartPicture column in the PartsIllus table if the Data column in the DataBank table contains more than 133 bytes:

```
SELECT DATA
  FROM DataBank
 WHERE STRING_LENGTH(Data) > 133
```

TID Function

Used in a select list, the TID function returns the database address of a row (or rows for BULK SELECT) of a table or an updatable view. Used in a WHERE clause, the TID function takes a row address as input and allows direct access to a single row of a table or an updatable view.

Scope

SQL Data Manipulation Statements

SQL Syntax

```
TID([ [Owner.]TableName
      [Owner.]ViewName
      CorrelationName   ])
```

Parameters

TID — is an 8 byte value representing the database address of a row of a table or an updatable view. A TID contains these elements:

Table 8-2. SQLTID Data Internal Format

Content	Byte Range
Always = 0	1 thru 2
File Number	3 thru 4
Page Number	5 thru 7
Slot	8

() — indicates that the row address is to be obtained from the first table or view specified (in the FROM clause of a SELECT statement or in an UPDATE statement).

Owner — indicates the owner of the table or view.

TableName — indicates the table from which to obtain the row address.

ViewName — indicates the updatable view from which to obtain row address.

CorrelationName — indicates the correlation name of the table or view from which to obtain the row address.

Description

- The TID function can be used with user tables and updatable views and with system base tables and system views. It cannot be used with non-updatable views (those containing JOIN, UNION, GROUP BY, HAVING, or aggregate functions) nor on system

pseudotables.

- In order to assure optimization (through the use of TID access) the expressions in the WHERE clause of a single query block must be ANDed together. No OR is allowed. In addition, only the following TID expressions can be optimized:

```
TID([ [Owner.]TableName
      [Owner.]ViewName
      CorrelationName    ]) =
{Constant
 HostVariableName [[INDICATOR]:IndicatorVariable]
 ?
 :LocalVariable
 :ProcedureVariable                                  }
```

- Only equal and not equal comparison operators are supported.
- The TID function cannot appear in an arithmetic expression.
- The TID function can be used in a restricted set of SELECT statements. A valid SELECT statement must *not* specify the following:
  - An ORDER BY or GROUP BY on the TID function.
  - A HAVING clause containing the TID function.
  - The TID function in the select list when a GROUP BY or HAVING clause is used.
  - An aggregate function on the TID function.
  - Any TID function along with an aggregate function in the select list.

Example

```
   isql=> SELECT tid(), PartNumber
   > FROM PurchDB.Parts;

   select tid(), PartNumber from PurchDB.Parts;
   ----------------------+---------------
   TID                   |PARTNUMBER
   ----------------------+---------------
                    3:3:0|1123-P-01
                    3:3:1|1133-P-01
                    3:3:2|1143-P-01
                    3:3:3|1153-P-01
                    3:3:4|1223-MU-01
                    3:3:5|1233-MU-01
                    3:3:6|1243-MU-01
                    3:3:7|1323-D-01
                    3:3:8|1333-D-01
                    3:3:9|1343-D-01
                   3:3:10|1353-D-01
                   3:3:11|1423-M-01
                   3:3:12|1433-M-01
                   3:3:13|1523-K-01
                   3:3:14|1623-TD-01
                   3:3:15|1723-AD-01
------------------------------------------------------------------------
   First 16 rows have been selected.
   U[p], d[own], l[eft], r[ight], t[op], b[ottom], pr[int] <n>, or e[nd] >
```

9 Search Conditions

This chapter discusses search condition clauses and the predicates used in them. The following sections are presented:

- Search Condition
- BETWEEN Predicate
- Comparison Predicate
- EXISTS Predicate
- IN Predicate
- LIKE Predicate
- NULL Predicate
- Quantified Predicate

A search condition specifies criteria for choosing rows to select, update, delete, insert, permit in a table, or fire rules on. Search conditions are parameters in the following statements:

- In the SELECT statement, search conditions are used for two purposes as follows:
  - In the WHERE clause, to determine rows to retrieve for further processing. The only expressions not valid in this clause are aggregate functions and expressions containing LONG columns that are not in long column functions.
  - In the HAVING clause, to specify a test to apply to each group of rows surviving the GROUP BY clause test(s). If a GROUP BY clause is not used, the test is applied to all the rows meeting the WHERE clause conditions. References in a HAVING clause to non-grouping columns must be from within aggregate functions. Grouping columns can be referred to by name or with an aggregate function.
- In the UPDATE statement, search conditions in the WHERE clause identify rows that qualify for updating.
- In the DELETE statement, search conditions in the WHERE clause identify rows that qualify for deletion.
- In the INSERT statement, search conditions in the embedded SELECT statement identify rows to copy from one or more tables or views into a table.
- In the DECLARE CURSOR statement, search conditions in the embedded SELECT statement identify rows and columns to be processed with a cursor.
- In the CREATE VIEW statement, search conditions in the embedded SELECT statement identify rows and columns that qualify for the view.
- In table CHECK constraints, the search condition identifies valid rows that a table may contain.
- In rule firing conditions, search conditions identify conditions that cause rules to fire.

Search Condition

A search condition is a single predicate or several predicates connected by the logical operators AND or OR. A predicate is a comparison of expressions that evaluates to a value of TRUE, FALSE, or unknown. If a predicate evaluates to TRUE for a row, the row qualifies for the select, update, or delete operation. If the predicate evaluates to FALSE or unknown for a row, the row is not operated on.

Scope

SQL Data Manipulation Statements

SQL Syntax

```
[NOT] {BetweenPredicate
       ComparisonPredicate
       ExistsPredicate
       InPredicate
       LikePredicate
       NullPredicate
       QuantifiedPredicate
       (SearchCondition)   } [ {AND
                                OR} [NOT] {BetweenPredicate
                                           ComparisonPredicate
                                           ExistsPredicate
                                           InPredicate
                                           LikePredicate
                                           NullPredicate
                                           QuantifiedPredicate
                                           (SearchCondition) } ] [...]
```

Parameters

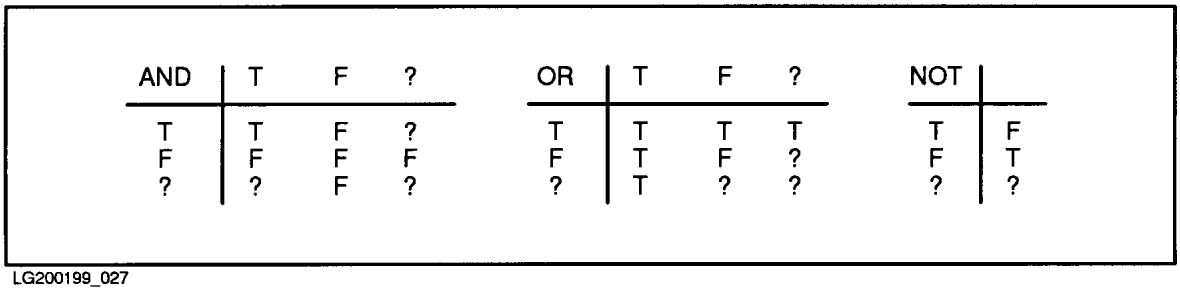
NOT, AND, OR		are logical operators with the following functions:
	NOT	reverses the value of the predicate that follows it.
	AND	evaluates predicates it joins to TRUE if they are both TRUE.
	OR	evaluates predicates it joins to TRUE if either or both are TRUE.
BetweenPredicate		determines whether an expression is within a certain range of values.
ComparisonPredicate		compares two expressions.
ExistsPredicate		determines whether a subquery returns any non-null values.
InPredicate		determines whether an expression matches an element within a specified set.

LikePredicate	determines whether an expression contains a particular character string pattern.
NullPredicate	determines whether a value is null.
QuantifiedPredicate	determines whether an expression bears a particular relationship to a specified set.
(SearchCondition)	is one of the above predicates, enclosed in parentheses.

Description

- Predicates in a search condition are evaluated as follows:
  - Predicates in parentheses are evaluated first.
  - NOT is applied to each predicate.
  - AND is applied next, left to right.
  - OR is applied last, left to right.
- When a predicate contains an expression that is null, the value of the predicate is unknown. Logical operations on such a predicate result in the following values, where a question mark (?) represents the unknown value:

Figure 9-1. Logical Operations on Predicates Containing NULL Values



LG200199_027

When the search condition for a row evaluates to unknown, the row does not satisfy the search condition and the row is not operated on. Check constraints are an exception; see the section on CREATE TABLE or CREATE VIEW.

- You can compare only compatible data types. INTEGER, SMALLINT, DECIMAL, FLOAT, and REAL are compatible. CHAR and VARCHAR are compatible, regardless of length. You can compare items of type DATE, TIME, DATETIME, and INTERVAL to literals of type CHAR or VARCHAR. ALLBASE/SQL converts the literal before the comparison. BINARY and VARBINARY are compatible, regardless of length.
- You cannot include a LONG BINARY or LONG VARBINARY data type in a predicate except within a long column function.
- A *SubQuery* expression cannot appear on the left-hand side of a predicate.
- Refer to Chapter 7 , “Data Types,” and Chapter 8 , “Expressions,” for information concerning value extensions and type conversion during comparison operations.

BETWEEN Predicate

A **BETWEEN predicate** determines whether a value is equal to or greater than a second value *and* equal to or less than a third value. The predicate evaluates to true if a value falls within the specified range. If the NOT option is used, the predicate evaluates to true if a value does *not* fall within the specified range.

Note that the second value must be less than or equal to the third value for BETWEEN to possibly be TRUE and for NOT BETWEEN to possibly be FALSE.

Scope

SQL Data Manipulation Statements

SQL Syntax

```
Expression1 [NOT]BETWEEN Expression2 AND Expression3
```

Parameters

Expression1, 2, 3	specify values used to identify columns, screen rows, or define new column values. The syntax for expressions is defined in the "Expressions" chapter. Both numeric and non-numeric expressions are allowed in BETWEEN predicates.
NOT	is a logical operator and reverses the value of the predicate that follows it.

Description

- *Expression2* and *Expression3* constitute a range of possible values for which *Expression2* is the lowest possible value and *Expression3* is the highest possible value. In the BETWEEN predicate, the low value must come before the high value. Also in the BETWEEN predicate, subqueries are not allowed.
- Comparisons are conducted as described under "Comparison Predicates" later in this chapter.

Example

Parts sold for under $250.00 and over $1500.00 are discounted by 25 percent.

```
UPDATE PurchDB.Parts SET SalesPrice = SalesPrice * .75
 WHERE SalesPrice NOT BETWEEN 250.00 AND 1500.00
```

Comparison Predicate

A **comparison predicate** compares two expressions using a **comparison operator**. The predicate evaluates to TRUE if the first expression is related to the second expression as specified in the comparison operator.

Scope

SQL Data Manipulation Statements

SQL Syntax

```
Expression { =
             <>
             >
             >=
             <
             <=} [ Expression
                   SubQuery    ]
```

Parameters

Expression	specifies a value used to identify columns, screen rows, or define new column values. The syntax of expressions is defined in Chapter 8 , “Expressions.” Both numeric and non-numeric expressions are allowed in comparison predicates. Predicates cannot include LONG columns.
SubQuery	is a QueryExpression whose result is used in evaluating another query. The syntax of QueryExpression is presented in the description of the SELECT statement.
=	is equal to. A comparison predicate using = is also known as an EQUAL predicate.
<>	is not equal to.
>	is greater than.
>=	is greater than or equal to.
<	is less than.
<=	is less than or equal to.

Description

- Character strings are compared according to the HP eight-bit ASCII collating sequence for ASCII data, or the collation rules for the native language of the DBEnvironment for NLS data. Column data would either be ASCII data or NLS data depending on how the column was declared upon its creation. Constants are ASCII data or NLS data depending on whether you are using NLS or not.

  If a case insensitive ASCII expression is compared to a case insensitive NLS expression,

the two expressions are compared using the NLS collation rules. The case insensitive NLS comparison is done by using the NLSCANMOVE and NLSCOLLATE intrinsics. The same ASCII characters in upper and lower case are equivalent. Accent characters (extended character) in upper and lower case are also equivalent. However, an accent character may not be the same as its ASCII equivalent, depending on the specific language collation table.

Extended upper and lower case characters are not equivalent to the ASCII expression. They are compared to the NLS collation table.

If a case sensitive character column is compared to a character column that is not case sensitive, both columns are treated as case sensitive. If a string constant is compared to a column that is not case sensitive, then the string constant is treated as not case sensitive.

- Refer to Chapter 7 , “Data Types,” for type conversion that ALLBASE/SQL performs when you compare values of different types.
- For purposes of the Comparison Predicate, a NULL value on either or both sides of the predicate causes it to evaluate to unknown. Thus, two NULL values on either side of an equals predicate will not result in a TRUE result but rather in unknown.
- A NULL value in an expression causes comparison operators to evaluate to unknown. Refer to the "Search Condition" section at the beginning of this chapter for more information on evaluation of operators.
- A subquery must return a single value (one column of one row). If the subquery returns more than one value, an error is given. If the subquery returns no rows, the predicate evaluates to unknown.

Example

The part numbers of parts that require fewer than 20 days for delivery are retrieved.

```
SELECT PartNumber
  FROM PurchDB.SupplyPrice
 WHERE DeliveryDays < 20
```

EXISTS Predicate

An **EXISTS predicate** tests for the existence of a row satisfying the search condition of a subquery. The predicate evaluates to TRUE if at least one row satisfies the search condition of the subquery.

Scope

SQL Data Manipulation Statements

SQL Syntax

EXISTS *SubQuery*

Parameters

SubQuery A subquery is a nested query. The syntax of subqueries is presented in the description of the SELECT statement in Chapter 12 , “SQL Statements S - Z.”

Description

Unlike other places in which subqueries occur, the EXISTS predicate allows the subquery to specify more than one column in its select list.

Example

Get supplier names for suppliers who provide at least one part.

```
SELECT S.SNAME
  FROM S
 WHERE EXISTS  ( SELECT * FROM SP
                 WHERE SP.SNO = S.SNO );
```

IN Predicate

An **IN predicate** compares an expression with a list of specified values or a list of values derived from a subquery. The predicate evaluates to TRUE if the expression is equal to one of the values in the list. If the NOT option is used, the predicate evaluates to TRUE if the expression is not equal to any of the values in the list.

Scope

SQL Data Manipulation Statements

SQL Syntax

```
Expression [NOT] IN { SubQuery
                    {ValueList)}
```

Parameters

Expression — An expression specifies a value to be obtained. The syntax of expressions is presented in Chapter 8 , “Expressions.” Both numeric and non-numeric expressions are allowed in quantified predicates. The expression may not include subqueries or LONG columns.

NOT — reverses the value of the predicate that follows it.

SubQuery — A subquery is a nested query. The syntax of subqueries is presented in the description of the SELECT statement in Chapter 12 , “SQL Statements S - Z.”

ValueList — defines a list of values to be compared against the expression's value. The syntax for *ValueList* is:

```
{ USER
  CurrentFunction
  [ + - ] {Integer
           Float
           Decimal}
  'CharacterString'
  OxHexadecimalString
  :HostVariable [[INDICATOR]:IndicatorVariable]
  ?
  :Local Variable
  :ProcedureParameter
  ::Built-inVariable
  LongColumnFunction
  StringFunction } [, ...]
```

USER — USER evaluates to the DBEUserID. In ISQL, it evaluates to the DBEUserID of the ISQL user. From an application program, it evaluates DBEUserID of the individual

	running the program. USER behaves like a CHAR(20) constant, with trailing blanks if the login name has fewer than 20 characters.
CurrentFunction	indicates the value of the current DATE, TIME, or DATETIME.
Integer	indicates a value of type INTEGER or SMALLINT.
Float	indicates a value of type FLOAT.
Decimal	indicates a value of type DECIMAL.
CharacterString	specifies a CHAR, VARCHAR, DATE, TIME, DATETIME, or INTERVAL value. Whichever is shorter -- the string or the expression value -- is padded with blanks before the comparison is made.
HexadecimalString	specifies a BINARY or VARBINARY value. If the string is shorter than the target column, it is padded with binary zeroes; if it is longer than the target column, the string is truncated.
HostVariable	contains a value in an application program being input to the expression.
IndicatorVariable	names an indicator variable, whose value determines whether the associated host variable contains a NULL value:
> = 0	the value is not NULL
< 0	the value is NULL (The value in the host variable will be ignored.)
?	is a place holder for a dynamic parameter in a prepared SQL statement in an application program. The value of the dynamic parameter is supplied at run time.
LocalVariable	contains a value in a procedure.
ProcedureParameter	contains a value that is passed into or out of a procedure.
Built-inVariable	is one of the following built-in variables used for error handling: • ::sqlcode • ::sqlerrd2 • ::sqlwarn0 • ::sqlwarn1 • ::sqlwarn2 • ::sqlwarn6 • ::activexact

	The first six of these have the same meaning that they have as fields in the SQLCA in application programs. Note that in procedures, sqlerrd2 returns the number of rows processed for all host languages. However, in application programs, sqlerrd3 is used in COBOL, Fortran, and Pascal, while sqlerr2 is used in C. ::activexact indicates whether a transaction is in progress or not. For additional information, refer to the application programming guides and to Chapter 4 , “Constraints, Procedures, and Rules.”
StringFunction	returns partial values or attributes of character and binary (including LONG) string data.
LongColumnFunction	returns information from the long column descriptor.

Description

- If X is the value of *Expression* and (a,b, ..., z) represent the result of a *SubQuery* or the elements in a *ValueList*, then the following are true:
  - `X IN (a,b,...,z)` is equivalent to `X = ANY (a,b,...,z)`
  - `X IN (a,b,...,z)` is equivalent to `X = a OR X = b OR...OR X = z`
  - `X NOT IN (a,b,...,z)` is equivalent to `NOT (X IN (a,b,...,z))`
- Refer to the "Data Types" chapter for information about the type conversions that ALLBASE/SQL performs when you compare values of different types.
- You can use host variables in the *ValueList*. If an indicator variable is used and contains a value less than zero, the value in the corresponding host variable is considered to be unknown.

NOTE To be consistent with the standard SQL and to support portability of code, it is strongly recommended that you use a –1 to indicate a NULL value. However, ALLBASE/SQL interprets all negative indicator variable values as indicating a NULL value in the corresponding host variable.

- If all values in the *ValueList* are NULL, the predicate evaluates to unknown.

Example

Get part numbers of parts whose weight is 12, 16, or 17.

```
SELECT P.PNO
  FROM P
 WHERE P.WEIGHT IN (12, 16, 17)
```

Get the names of suppliers who supply part number 'P2'.

```
SELECT S.SNAME
  FROM S
 WHERE S.SNO IN  (SELECT SP.SNO FROM SP
                  WHERE SP.SNO = 'P2')
```

If the indicator variable is >= 0 and PartNumber is one of '1123-P-01', '1733-AD-01', or :PartNumber, then the predicate evaluates to true.

If the indicator variable is < 0, the rows containing the part numbers 1123-P-01 and 1733-AD-01 are selected; but no rows will be selected based upon the value in :PartNumber.

```
EXEC SQL SELECT PartNumber
           FROM PurchDB.Parts
          WHERE PartNumber
             IN ('1123-P-01', '1733-AD-01', :PartNumber :PartInd)
```

LIKE Predicate

A LIKE predicate determines whether an expression contains a given pattern. The predicate evaluates to TRUE if an expression contains the pattern. If the NOT option is used, the predicate evaluates to TRUE if the expression does not contain the pattern.

Scope

SQL Data Manipulation Statements

SQL Syntax

```
Expression [NOT]LIKE {'PatternString'
                      :HostVariable1[[INDICATOR]:IndicatorVariable1]
                      ?
                      :LocalVariable1
                      :ProcedureParameter1                          }
[ESCAPE{'EscapeChar'
        :HostVariable2[[INDICATOR]:IndicatorVariable2]
        ?
        :LocalVariable2
        :ProcedureParameter2                              }]
```

Parameters

Expression	specifies a value used to identify columns, screen rows, or define new column values. The syntax of expressions is presented in the "Expressions" chapter. Only CHAR and VARCHAR expressions are valid in LIKE predicates. Date/time columns cannot be referred to directly; however, they can be placed inside the conversion function TO_CHAR and be converted to a CHAR value. *Expression* cannot be a subquery.
NOT	reverses the value of the predicate.
PatternString	describes what you are searching for in the expression. The pattern can consist of characters only (including digits). For example, NAME LIKE 'Annie' evaluates to true only for a name of Annie. Uppercase and lowercase are significant. You can also use the predicate to test for the existence of a partial match, by using the following symbols in the pattern:
_	represents any single character; for example, BOB and TOM both satisfy the predicate NAME LIKE '_O_'.

%	represents any string of zero or more characters; for example, THOMAS and TOM both satisfy the predicate NAME LIKE '%O%'. The _ and % symbols can be used multiple times and in any combination in a pattern. You cannot use these symbols literally within a pattern unless the ESCAPE clause appears, and the escape character precedes them. Note that they must be ASCII and not your local representations.
HostVariable1	identifies the host variable in which the pattern is stored.
IndicatorVariable1	names an indicator variable, an input host variable whose value determines whether the associated host variable contains a NULL value: >= 0 the value is not NULL < 0 the value is NULL
EscapeChar	describes an optional escape character which can be used to include the symbols _ and % in the pattern. The escape character must be a single character, although it can be a one- or two-byte NLS character. When it appears in the pattern, it must be followed by the escaped character, host variable or, _, or %. Each such pair represents a single literal occurrence of the second character in the pattern. The escape character is always case sensitive. All other characters are interpreted as described before.
HostVariable2	identifies the host variable containing the escape character.
IndicatorVariable2	names an indicator variable, an input host variable whose value determines whether the associated host variable contains a NULL value: >=0 the value is not NULL < 0 the value is NULL If the escape character is NULL, the predicate evaluates to unknown.
LocalVariable2	contains the escape character.

ProcedureParameter2	contains the escape character that is passed into or out of a procedure.
?	indicates a dynamic parameter in a prepared SQL statement. The value of the parameter is supplied when the statement is executed.

Description

- If an escape character is not specified, then the _ or % in the pattern continues to act as a wildcard. No default escape character is available. If an escape character is specified, then the wildcard or escape character which follows an escape character is treated as a constant. If the character following an escape character is not a wildcard or the escape character, an error results.
- If the value of the expression, the pattern, or the escape character is NULL, then the LIKE predicate evaluates to unknown.

Example

Vendors located in states beginning with an A are identified.

```
SELECT VendorName FROM PurchDB.Vendors
 WHERE VendorState LIKE 'A%'
```

Vendors whose names begin with ACME_ are identified.

```
SELECT VendorName FROM PurchDB.Vendors
 WHERE VendorName LIKE 'ACME!_%' ESCAPE '!'
```

NULL Predicate

A NULL predicate determines whether a primary has the value NULL. The predicate evaluates to true if the primary is NULL. If the NOT option is used, the predicate evaluates to true if the primary is not NULL.

Scope

SQL Data Manipulation Statements

SQL Syntax

```
{ColumnName
 :HostVariable[[INDICATOR]:IndicatorVariable]
 ?
 :LocalVariable
 :ProcedureParameter
 ::Built-inVariable
 AddMonthsFunction
 AggregateFunction
 Constant
 DateTimeFunction
 CurrentFunction
 LongColumnFunction
 StringFunction
 CASTFunction
 TIDFunction
 (Expression)                                    } IS [NOT] NULL
```

Parameters

ColumnName is the name of a column from which a value is to be taken; column names are defined in Chapter 6 , “Names.”.

HostVariable contains a value in an application program being input to the expression.

IndicatorVariable names an indicator variable, whose value determines whether the associated host variable contains a NULL value:

> = 0

the value is not NULL

< 0

the value is NULL (The value in the host variable will be ignored.)

? is a place holder for a dynamic parameter in a prepared SQL statement in an application program. The value of the dynamic parameter is supplied at run time.

LocalVariable	contains a value in a procedure.
ProcedureParameter	contains a value that is passed into or out of a procedure.
Built-inVariable	is one of the following built-in variables used for error handling: • ::sqlcode • ::sqlerrd2 • ::sqlwarn0 • ::sqlwarn1 • ::sqlwarn2 • ::sqlwarn6 • ::activexact The first six of these have the same meaning that they have as fields in the SQLCA in application programs. Note that in procedures, sqlerrd2 returns the number of rows processed for all host languages. However, in application programs, sqlerrd3 is used in COBOL, Fortran, and Pascal, while sqlerr2 is used in C. ::activexact indicates whether a transaction is in progress or not. For additional information, refer to the application programming guides and to Chapter 4 , “Constraints, Procedures, and Rules.”
AddMonthsFunction	returns a value that represents a DATE or DATETIME value with a certain number of months added to it.
AggregateFunction	is a computed value; aggregate functions are defined in this chapter.
Constant	is a specific value; constants are defined later in this chapter.
ConversionFunction	returns a value that is a conversion of a date/time data type into an INTEGER or CHAR value, or from a CHAR value.
CurrentFunction	returns a value that represents the current DATE, TIME, or DATETIME.
LongColumnFunction	returns information from a long column descriptor.
StringFunction	returns a partial value or attribute of string data.
TIDFunction	returns the database address of a row (or rows for a BULK SELECT) of a table or an updatable view. You cannot use mathematical operators with this function except to compare it to a value, host variable, or dynamic parameter (using =, or <>) .
(Expression)	is one or more of the above primaries, enclosed in parentheses.
NOT	reverses the value of the predicate that follows it.

Description

The primary may be of any data type except LONG BINARY or LONG VARBINARY.

Example

Vendors with no personal contact named are identified.

```
SELECT *
  FROM PurchDB.Vendors
 WHERE ContactName IS NULL
```

Quantified Predicate

A quantified predicate compares an expression with a list of specified values or a list of values derived from a subquery. The predicate evaluates to true if the expression is related to the value list as specified by the comparison operator and the quantifier.

Scope

SQL Data Manipulation Statements

SQL Syntax

```
Expression{=
          <>
          >
          >=
          <
          <=} {ALL
               ANY
               SOME} {SubQuery
                      (ValueList)}
```

Parameters

Expression An expression specifies a value to be obtained. The syntax of expressions is presented in Chapter 8 , “Expressions.”

= is equal to.

<> is not equal to.

> is greater than.

>= is greater than or equal to.

< is less than.

<= is less than or equal to.

ALL, ANY, SOME are quantifiers which indicate how many of the values from the *ValueList* or *SubQuery* must relate to the expression as indicated by the comparison operator in order for the predicate to be true. Each quantifier is explained below:

- ALL — the predicate is true if *all* the values in the *ValueList* or returned by the *SubQuery* relate to the expression as indicated by the comparison operator.
- ANY — the predicate is true if *any* of the values in the *ValueList* or returned by the *SubQuery* relate to the expression as indicated by the comparison operator.
- SOME — a synonym for ANY.

SubQuery A subquery is a nested query. Subqueries are presented fully in the description of the SELECT statement.

ValueList defines a list of values to be compared against the expression's value. The syntax for *ValueList* is:

```
{ USER
  CurrentFunction
  [ +
    - ] {Integer
         Float
         Decimal}
  'CharacterString'
  0xHexadecimalString
  :HostVariable [[INDICATOR]:IndicatorVariable]
  ?
  :Local Variable
  :ProcedureParameter
  ::Built-inVariable
  LongColumnFunction
  StringFunction                                  } [, ...]
```

USER evaluates to login name. In ISQL, it evaluates to the login name of the ISQL user. From an application program, it evaluates to the login name of the individual running the program. USER behaves like a CHAR(20) constant, with trailing blanks if the login name has fewer than 20 characters.

CurrentFunction indicates the value of the current DATE, TIME, or DATETIME.

Integer indicates a value of type INTEGER or SMALLINT.

Float indicates a value of type FLOAT.

Decimal indicates a value of type DECIMAL.

CharacterString specifies a CHAR, VARCHAR, DATE, TIME, DATETIME, or INTERVAL value. Whichever is shorter -- the string or the expression value -- is padded with blanks before the comparison is made.

HexadecimalString specifies a BINARY or VARBINARY value. If the string is shorter than the target column, it is padded with binary zeroes; if it is longer than the target column, it is truncated.

HostVariable identifies the host variable containing the column value.

IndicatorVariable1 names an indicator variable, an input host variable whose value determines whether the associated host variable contains a NULL value:

>= 0

the value is not NULL

< 0

the value is NULL

LocalVariable contains a value in a procedure.

ProcedureParameter contains a value that is passed into or out of a procedure.

? indicates a dynamic parameter in a prepared SQL statement. The value of the parameter is supplied when the statement is executed.

Description

- If X is the value of *Expression*, and (a,b, ..., z) represent the result of a *SubQuery* or the elements in a *ValueList*, and OP is a comparison operator, then the following are true:
  - `X OP ANY (a,b,...,z)` is equivalent to `X OP a OR X OP b OR...OR X OP z`
  - `X OP ALL (a,b,...,z)` is equivalent to `X OP a AND X OP b AND...AND X OP z`
- Character strings are compared according to the HP 8-bit ASCII collating sequence for ASCII data, or the collation rules for the native language of the DBEnvironment for NLS data. Column data would either be ASCII data or NLS data depending on how the column was declared upon its creation. Constants will be ASCII data or NLS data depending on whether the user is using NLS or not. If an ASCII expression is compared to an NLS expression, the two expressions are compared using the NLS collation rules.
- Refer to Chapter 7 , “Data Types,” for information about the type conversions that ALLBASE/SQL performs when you compare values of different types.
- If any value of any element in the value list is a NULL value, then that value is not considered a part of the *ValueList*.

NOTE To be consistent with the standard SQL and to support portability of code, it is strongly recommended that you use a –1 to indicate a NULL value. However, ALLBASE/SQL interprets all negative indicator variable values as indicating a NULL value in the corresponding host variable.

Example

Get supplier numbers for suppliers who supply at least one part in a quantity greater than every quantity in which supplier S1 supplies a part.

```
SELECT DISTINCT SP.SNO
  FROM SP
 WHERE SP.QTY > ALL ( SELECT SP.QTY
                        FROM SP
                       WHERE SP.SNO = 'S1')
```

An alternative, possibly faster form of the query is:

```
SELECT DISTINCT SP.SNO
  FROM SP
 WHERE SP.QTY > (SELECT MAX(SP.QTY)
                   FROM SP
                  WHERE SP.SNO = 'S1')
```

10 SQL Statements A - D

Chapters 10, 11 and 12 describe all the SQL statements in alphabetical order, giving syntax, parameters, descriptions, authorization requirements, and examples for each statement. Examples often consist of groups of statements so you can see how each statement is related to other statements functionally.

SQL Statement Summary

SQL statements fall into four groups. General-purpose statements are used programmatically, interactively, and in procedures. Application programming statements are used in application programs. Database administration statements are usually used interactively. Procedure, control flow, and status statements are used only in procedures. Within each of these groups, the SQL statements fall into categories, as shown in Table 10-1.

Table 10-1. SQL Statement Summary

Group	Category	Statement	Statement Use
General Purpose Statements			
	DBEnvironment session management		
		`CONNECT`	Begins a DBEnvironment session.
		`DISCONNECT`	Terminates a connection to a DBEnvironment, or all connections.
		`SET CONNECTION`	Sets the current connection within the currently connected set of DBEnvironments.
		`SET MULTI TRANSACTION`	Switches between single-transaction mode and multi-transaction mode.
		`RELEASE`	Terminates a DBEnvironment session.
	Data definition		
	Databases	`CREATE SCHEMA`	Defines a database and associates it with an authorization name.
	Indexes	`CREATE INDEX`	Defines an index for a table based on one or more of its columns.
		`DROP INDEX`	Deletes an index.

Table 10-1. SQL Statement Summary

Group	Category	Statement	Statement Use
	Tables	ALTER TABLE	Adds to a table new columns and constraints, or drops constraints from a table, and assigns a table to a partition or removes it from a partition.
		RENAME COLUMN	Defines a new name for an existing column.
		RENAME TABLE	Defines a new name for an existing table.
		CREATE TABLE	Defines a table and assigns it to a partition.
		TRUNCATE TABLE	Deletes all rows from a table.
		DROP TABLE	Deletes a table and any authorities, indexes, rules, and views based on it.
	Views	CREATE VIEW	Defines a view based on a table, another view, or a combination of tables and views.
		DROP VIEW	Deletes the definition of a view as well as authorities or views based on the view.
	Rules	CREATE RULE	Defines a rule for a table and associates it with INSERTS, UPDATES, and/or DELETES.
		DROP RULE	Deletes a rule.
	Groups, DBEFileSet, DBEFiles	Refer to the database administration statements.	
	Procedures	CREATE PROCEDURE	Defines a procedure for storage in the DBEnvironment.
		DROP PROCEDURE	Deletes a procedure.
	Partitions	CREATE PARTITION	Defines a partition for audit logging in the DBEnvironment.
		DROP PARTITION	Deletes a partition.
	Data manipulation		
		DELETE	Deletes one or more rows from a single table or view.

Table 10-1. SQL Statement Summary

Group	Category	Statement	Statement Use
		INSERT	Adds a row to a single table or view.
		SELECT	Retrieves data from one or more tables or views.
		UPDATE	Changes the values of one or more columns in all rows of a specific table or view that satisfy a search condition.
		DROP MODULE	Deletes a preprocessed module.
		EXECUTE	Executes dynamically preprocessed statements.
		EXECUTE IMMEDIATE	Defines and executes dynamic statements.
		PREPARE	Dynamically preprocesses statements, storing them as a module if issued interactively.
	Transaction management		
		BEGIN WORK	Begins a transaction and optionally sets its isolation level and priority.
		COMMIT WORK	Ends a transaction and makes permanent any changes it made to the DBEnvironment.
		ROLLBACK WORK	Ends a transaction and undoes changes made to the DBEnvironment during the whole transaction or back to a savepoint within the transaction.
		SAVEPOINT	Defines a point within a transaction back to which you can roll back work.
		SET DML ATOMICITY	Sets the general error checking level.
		SET CONSTRAINTS	Sets the level of constraint error checking.
		SET SESSION	Sets transaction attributes for a session.
		SET TRANSACTION	Sets execution attributes for a transaction.

Table 10-1. SQL Statement Summary

Group	Category	Statement	Statement Use
	Executing procedures	EXECUTE PROCEDURE	Invokes a procedure.
	Other	RAISE ERROR	Causes a user-defined error to occur and specifies the error number and text to be raised.
	Concurrency		
		CREATE TABLE	Defines the automatic locking strategy and implicit authority grants used for a table.
		LOCK TABLE	Locks a table, explicitly overriding ALLBASE/SQL's automatic locking strategy.
		START DBE	Defines the maximum number of transactions that can execute concurrently, when used with the TRANSACTION= parameter.
	Module maintenance		
		DROP MODULE	Deletes a module from the system catalog, optionally retaining authorization information.
		GENPLAN	Places optimizer's access plan in SYSTEM.PLAN (from ISQL only).
		SETOPT	Modifies access optimization plan used by queries.
		VALIDATE	Validates modules and procedures.
Application Programming Statements			
	Single row data manipulations		
		FETCH	Retrieves a single row from an active set associated with a cursor.
		INSERT	Inserts a single row into a table.
		SELECT	Retrieves a single row not associated with a cursor.
	Bulk manipulations		
		BULK FETCH	Retrieves multiple rows from an active set associated with a cursor. (See *FETCH*.)

Table 10-1. SQL Statement Summary

Group	Category	Statement	Statement Use
		BULK INSERT	Inserts multiple rows into a single table. (See INSERT.)
		BULK SELECT	Retrieves multiple rows not associated with a cursor. (See SELECT.)
	Cursor management		
		ADVANCE	Advances a procedure cursor.
		CLOSE	Closes a cursor currently in the open state.
		DECLARE CURSOR	Associates a cursor with a specific SELECT or EXECUTE PROCEDURE statement.
		DELETE WHERE CURRENT	Deletes the current row of an active set.
		FETCH	Advances the position of an open cursor to the next row of the active set and copies columns into host variables.
		REFETCH	Copies columns from the current cursor position in the active set into host variables. Used with the RU and RC isolation levels to verify the continued existence of data and to obtain stronger locks prior to updating.
		OPEN	Makes an active set available to manipulation statements.
		UPDATE WHERE CURRENT	Changes columns in the current row of the active set.
	Preprocessor directives		
		BEGIN DECLARE SECTION	Indicates the beginning of the host variable declarations in an application program.
		END DECLARE SECTION	Indicates the end of the host variable declarations in an application program.
		INCLUDE	Includes declarations for structures used to pass information between ALLBASE/SQL and a program.

Table 10-1. SQL Statement Summary

Group	Category	Statement	Statement Use
		WHENEVER	Specifies an action to be taken depending on the outcome of an SQL statement.
	Dynamically preprocessed queries		
		DESCRIBE	Obtains information about the results of a dynamic statement.
		EXECUTE EXECUTE IMMEDIATE PREPARE	Refer to general-purpose statements.
	Status messages	SQLEXPLAIN	Retrieves a message describing the status of SQL statement execution.
Database Administration Statements			
	Authorization		
		GRANT	Grants authorities to all users, specific users, or groups.
		REVOKE	Revokes authorities from all users, specific users, or groups.
		TRANSFER OWNERSHIP	Makes a different user or authorization group the owner of a table, view, authorization group, or procedure.
	Authorization groups		
		ADD TO GROUP	Adds one or more users or groups to an authorization group.
		CREATE GROUP	Defines an authorization group.
		DROP GROUP	Removes the definition of an authorization group from the system catalog.
		REMOVE FROM GROUP	Removes one or more users or groups from an authorization group.
	DBEnvironment configuration and use		
		START DBE NEW	Configures a new DBEnvironment.
		START DBE	Makes a DBEnvironment available in a mode different from that defined in the DBECon file; also starts up a DBEnvironment when the autostart flag is off.

Table 10-1. SQL Statement Summary

Group	Category	Statement	Statement Use
	STOP DBE		
		Terminates all DBE sessions and causes a checkpoint to be taken, recovering log file space if nonarchive logging is in effect.	
		TERMINATE QUERY	Terminates a running Query.
		TERMINATE TRANSACTION	Stops the transaction.
		TERMINATE USER	Stops the DBE session for a specific user.
	DBEnvironment settings		
		ENABLE RULES	Turns rule checking on for the current DBEnvironment session.
		DISABLE RULES	Turns rule checking off for the current DBEnvironment session.
		SET PRINTRULES	Specifies whether rule names and statement types are to be issued as messages when the rules are fired during a DBEnvironment session.
		SET USER TIMEOUT	Specifies the amount of time the user waits if requested database resource is unavailable.
	Space Management		
	DBEFiles	ADD DBEFILE	Associates a DBEFile with a DBEFileSet.
		ALTER DBEFILE	Changes the type attribute of a DBEFile.
		CREATE DBEFILE	Defines and creates a DBEFile.
		DROP DBEFILE	Removes the definition of an empty DBEFile not associated with a DBEFileSet.
		REMOVE DBEFILE	Disassociates a DBEFile from a DBEFileSet.
	DBEFileSets	CREATE DBEFILESET	Defines a DBEFileSet.

Table 10-1. SQL Statement Summary

Group	Category	Statement	Statement Use
		`SET DEFAULT DBEFILESET`	Sets a default DBEFileSet.
		`DROP DBEFILESET`	Removes the definition of a DBEFileSet from the system catalog.
	Temporary sort space		
		`CREATE TEMPSPACE`	Defines and creates a temporary storage space.
		`DROP TEMPSPACE`	Removes the definition of a temporary storage space from the system catalog.
	Logging		
	Recovery of log space	`BEGIN ARCHIVE` `COMMIT ARCHIVE`	Starts a new archive log file before a DBEnvironment is back up.
		`CHECKPOINT`	Causes an ALLBASE/SQL system checkpoint to be taken. A system checkpoint causes data and log buffers to be written to disk and makes old log space, occupied by completed transactions, available for reuse if nonarchive logging is in effect. Returns values in host variable.
		`START DBE NEWLOG`	Reinitializes log file(s) when you need to change the size. Makes audit logging effective when used with AUDIT LOG option.
		`START DBE`	Initiates the first DBE session if the DBE is not in autostart mode and causes a checkpoint to be taken, recovering log file space if nonarchive logging is in effect.
		`STOP DBE`	Terminates all DBE sessions and causes a checkpoint to be taken, recovering log file space if nonarchive logging is in effect.
	Dual logging	`START DBE NEW`	Causes ALLBASE/SQL to maintain two separate, identical logs, when used with the DUAL LOG option. Makes audit logging effective when used with AUDIT LOG option.

Table 10-1. SQL Statement Summary

Group	Category	Statement	Statement Use
	Audit logging	`DISABLE AUDIT LOGGING`	Disables current audit logging for a session.
	Log comment	`LOG COMMENT`	Enters a user comment in the log file.
		`ENABLE AUDIT LOGGING`	Enables audit logging for a session after being disabled.
	Recovery Rollback		
		`START DBE`	Rolls back transactions that were incomplete the last time the DBEnvironment was shut down.
		`TERMINATE USER`	Ends a user's transactions, backing out any work not committed.
		`STOP DBE`	Terminates all DBE sessions and causes a checkpoint to be taken,
	Rollforward	`BEGIN ARCHIVE` `COMMIT ARCHIVE`	Creates an archive record in the rollforward log(s) and initiates archive mode logging.
	DBEnvironment statistics		
		`RESET`	Resets ALLBASE/SQL accounting and statistical data activity management.
		`UPDATE STATISTICS`	Updates system catalog information used to optimize data access operations on a per table basis.
Procedure Statements			
	General statements		
		Assignment (=)	Assigns a value to a local variable or parameter in a procedure.
		`DECLARE` *`Variable`*	Defines a local variable within a procedure.
		`PRINT`	Stores information to be displayed by ISQL or an application program.
	Control flow statements		
		`BEGIN`	Begins a single statement or group of statements within a procedure.
		`GOTO`	Permits a jump to a labeled statement within a procedure.

Table 10-1. SQL Statement Summary

Group	Category	Statement	Statement Use
		Label	Labels a statement in a procedure.
		IF	Allows conditional execution of one or more statements within a procedure.
		RETURN	Permits an exit from a procedure with an optional return code.
		WHILE	Allows looping within a procedure.

ADD DBEFILE

The ADD DBEFILE statement updates a row in SYSTEM.DBEFile to show the DBEFileSet with which the file is associated.

Scope

ISQL or Application Program

SQL Syntax

ADD DBEFILE *DBEFileName* **TO DBEFILESET** *DBEFileSetName*

Parameters

DBEFileName	is the name of a DBEFile previously defined and created by the CREATE DBEFILE statement.
DBEFileSetName	is the name of a previously defined DBEFileSet. You can use the CREATE DBEFILESET statement to define DBEFileSets.

Description

- You cannot insert any rows or create any indexes for a table or put any non-null values in a LONG column until the DBEFileSet it is located in has DBEFiles associated with it.
- You can add DBEFiles to the SYSTEM DBEFileSet.
- Before a DBEFile can be added to the SYSTEM DBEFileSet, other users' transactions must complete. Other users must wait until the transaction that is adding the DBEFile to SYSTEM has completed.
- ADD DBEFILE increases the number of files associated with the DBEFileSet shown in the DBEFSNDBEFILES column of SYSTEM.DBEFileSet by one.

Authorization

You must have DBA authority to use this statement.

Example

```
CREATE DBEFILE ThisDBEFile WITH PAGES = 4,
               NAME = 'ThisFile', TYPE = TABLE

CREATE DBEFILESET Miscellaneous

ADD DBEFILE ThisDBEFile TO DBEFILESET Miscellaneous
```

The DBEFile is used to store rows of a new table. When the table needs an index, a

DBEFile to store rows of the index is created:

```
CREATE DBEFILE ThatDBEFile WITH PAGES = 4,
               NAME = 'ThatFile', TYPE = INDEX

ADD DBEFILE ThatDBEFile

TO DBEFILESET Miscellaneous
```

When the index is subsequently dropped, its file space can be assigned to another DBEFileSet.

```
REMOVE DBEFILE ThatDBEFile FROM DBEFILESET Miscellaneous

ADD DBEFILE ThatDBEFile TO DBEFILESET SYSTEM

ALTER DBEFILE ThisDBEFile SET TYPE = MIXED
```

All rows are later deleted from the table, so you can reclaim file space.

```
REMOVE DBEFILE ThisDBEFile FROM DBEFILESET Miscellaneous

DROP DBEFILE ThisDBEFile

The DBEFileSet definition can now be dropped.

DROP DBEFILESET Miscellaneous
```

ADD TO GROUP

The ADD TO GROUP statement adds one or more users or groups, or a combination of users and groups, to an authorization group.

Scope

ISQL or Application Program

SQL Syntax

```
ADD {DBEUserID
     GroupName
     ClassName}[,...]TO GROUP TargetGroupName
```

Parameters

DBEUserID	identifies a user to be added. You cannot specify the name of the DBECreator.
GroupName	identifies a group to be added.
ClassName	identifies a class to be added.
TargetGroupName	is the name of the authorization group to which the specified users, groups, and classes are to be added.

Description

- You can specify a single parameter chosen from the available types. You can also specify multiple parameters (using the same or multiple types) separating them with commas.
- Two authorization groups cannot be members of each other, that is group membership cannot follow a circular chain. If, for example, group3 is a member of group2, and group2 is a member of group1, group1 cannot be a member of group2 or group3.
- You cannot add an authorization group to itself.
- When you specify several users or groups in one ADD TO GROUP statement, ALLBASE/SQL ignores any invalid names, but processes the valid names.

Authorization

You can use this statement if you have OWNER authority for the authorization group or if you have DBA authority.

Example

```
CREATE GROUP Warehse

GRANT CONNECT TO Warehse
```

```
GRANT SELECT,
      UPDATE (BinNumber,QtyOnHand,LastCountDate)
   ON PurchDB.Inventory
   TO Warehse
```

These two users will be able to start DBE sessions on PartsDBE, retrieve data from table PurchDB.Inventory, and update three columns in the table.

```
ADD Clem, George TO GROUP Warehse
```

Clem will no longer have any of the authorities associated with group Warehse.

```
REMOVE Clem FROM GROUP Warehse
```

Because this group does not own any database objects, it can be deleted. George no longer has any of the authorities once associated with the group.

```
DROP GROUP Warehse
```

ADVANCE

The ADVANCE statement is a procedure cursor manipulation statement. It is used in conjunction with procedures having one or more multiple row result sets to advance the position of an opened procedure cursor to the first or next query result set and to initialize information in the associated sqlda_type and sqlformat_type data structures.

Scope

Application Programs Only

SQL Syntax

```
ADVANCE CursorName [USING [SQL]DESCRIPTOR {SQLDA
                                           AreaName}]
```

Parameters

CursorName	identifies a procedure cursor. The procedure cursor's current active query result set, the procedure's statements, and the values of any procedure input parameters, determine the format information to be returned by each successive ADVANCE statement.
USING [SQL] DESCRIPTOR	defines where to place the data format information of a query result for an EXECUTE PROCEDURE statement on which a procedure cursor has been defined. Specify a location that does not conflict with that of another SQL statement such as OPEN, CLOSE, DESCRIBE, EXECUTE, or any FETCH that is not associated with this ADVANCE statement.
SQLDA	specifies that a data structure of sqlda_type named SQLDA is to be used to pass information about the next result set between the application and ALLBASE/SQL.
AreaName	specifies the user defined name of a data structure of sqlda_type that is to be used to pass information about the next result set between the application and ALLBASE/SQL.

Description

- The query result set to which the procedure cursor points is called the active result set. You use the information in the associated sqlda_type and sqlformat_type data structures to process the query result set via FETCH statements.
- For a procedure that returns multiple row results of a single format, if the procedure was created with the WITH RESULT clause, it is unnecessary to issue an ADVANCE statement to get format information for each result set, since the format is already

known from the DESCRIBE RESULT statement.

- The ADVANCE statement cancels any current, active query result set. It can be used as an efficient way to throw away any unread rows resulting from the most recently executed multiple row result set SELECT statement in the procedure. The execution of the procedure continues with the next statement. Control returns to the application when the next multiple row result set statement is executed, or when procedure execution terminates.
- Refer to the ALLBASE/SQL *Advanced Application Programming Guide* for further explanation and examples of how to use the ADVANCE statement.

Authorization

You do not need authorization to use the ADVANCE statement.

Example

Refer to the ALLBASE/SQL *Advanced Application Programming Guide* for a pseudocode example of procedure cursor usage.

ALTER DBEFILE

The ALTER DBEFILE statement changes the TYPE attribute of a DBEFile.

Scope

ISQL or Application Program

SQL Syntax

```
ALTER DBEFILE DBEFileName SET TYPE = {TABLE
                                      INDEX
                                      MIXED}
```

Parameters

DBEFileName specifies the DBEFile to be altered.

TYPE = specifies the new setting of the DBEFile's TYPE attribute. The following are valid settings:

TABLE	Only data (table, LONG column, or HASH) pages can be stored in the DBEFile.
INDEX	Only index pages can be stored in the DBEFile.
MIXED	A mixture of data and index pages can be stored in the DBEFile.

Description

- The type of an empty DBEFile, that is, a DBEFile in which no table or index entries exist, can be changed without restriction.
- The type of a nonempty DBEFile can be changed from TABLE or INDEX to MIXED; no other changes are allowed.
- Once a DBEFile contains primary pages for a HASH table, no other nonhash table, index, or LONG data can be placed in that DBEFile.
- Before you can alter the type of a DBEFile in the SYSTEM DBEFileSet, other users' transactions must complete. Other users must wait until the transaction that is altering the DBEFile has completed.

Authorization

You must have DBA authority to use this statement.

Example

```
CREATE DBEFILE ThisDBEFile WITH PAGES = 4,
               NAME = 'ThisFile', TYPE = TABLE

CREATE DBEFILESET Miscellaneous

  ADD DBEFILE ThisDBEFile
TO DBEFILESET Miscellaneous
```

The DBEFile is used to store rows of a new table. When the table needs a DBEFile in which to store an index, one is created as follows:

```
CREATE DBEFILE ThatDBEFile WITH PAGES = 4,
               NAME = 'ThatFile', TYPE = INDEX

  ADD DBEFILE ThatDBEFile
TO DBEFILESET Miscellaneous
```

When the index is subsequently dropped, its file space can be assigned to another DBEFileSet.

```
REMOVE DBEFILE ThatDBEFile FROM DBEFILESET Miscellaneous

  ADD DBEFILE ThatDBEFile
TO DBEFILESET SYSTEM

ALTER DBEFILE ThisDBEFile SET TYPE = MIXED
```

All rows are later deleted from the table, so you can reclaim file space.

```
 REMOVE DBEFILE ThisDBEFile
FROM DBEFILESET Miscellaneous

DROP DBEFILE ThisDBEFile
```

The DBEFileSet definition can now be dropped.

```
DROP DBEFILESET Miscellaneous
```

ALTER TABLE

The ALTER TABLE statement is used to add one or more new columns or constraints, to drop one or more constraints, or to reassign the table audit partition. This statement is also used to change the type of table access, updatability, and locking strategies. New columns are appended following already existing columns of a table. New column definitions must either allow null values or provide default values if the table is not empty. Added columns may specify constraints.

Scope

ISQL or Application Programs

SQL Syntax

```
ALTER TABLE [Owner.]TableName {AddColumnSpecification
                               AddConstraintSpecification
                               DropConstraintSpecification
                               SetTypeSpecification
                               SetPartitionSpecification    }
```

Parameters—ALTER TABLE

[Owner.]TableName designates the table to be altered.

AddColumnSpecification allows a new column to be added to an existing table. This parameter is discussed in a separate section below.

AddConstraintSpecification allows a new constraint to be added to an existing table. This parameter is discussed in a separate section below.

DropConstraintSpecification allows an existing constraint to be dropped from an existing table. This parameter is discussed in a separate section below.

SetTypeSpecification allows the locking mode of the table and related authorities to be changed. This parameter is discussed in a separate section below.

SetPartitionSpecification allows a table or DBEnvironment partition to be changed.

SQL Syntax—AddColumnSpecification

```
ADD { (ColumnDefinition [,...])
       ColumnDefinition          } [CLUSTERING ON CONSTRAINT [ConstraintID]]
```

Parameters—AddColumnSpecification

ColumnDefinition The syntax of *ColumnDefinition* is presented under the

CREATE TABLE statement.

CLUSTERING ON CONSTRAINT	specifies that the named unique or referential constraint specified within the Column Definition be managed through a clustered index structure rather than nonclustered. The unique constraint's unique column list, or referential constraint's referencing column list, becomes the clustered key.
ConstraintID	specifies the unique or referential constraint on which clustering is to be applied. If not specified, the primary key of the table is assumed. The *ConstraintID* must be for a constraint being added with the ALTER TABLE statement.

SQL Syntax—AddConstraintSpecification

```
ADD CONSTRAINT ({UniqueConstraint
                 ReferentialConstraint
                 CheckConstraint}[,...])
[CLUSTERING ON CONSTRAINT [ConstraintID1]]
```

Parameters—AddConstraintSpecification

UniqueConstraint	defines a unique constraint being added. This parameter is described under the CREATE TABLE statement.
ReferentialConstraint	defines a referential constraint being added. This parameter is described under the CREATE TABLE statement.
CheckConstraint	defines a check constraint being added. This parameter is described under the CREATE TABLE statement.
CLUSTERING ON CONSTRAINT	specifies that the named unique or referential constraint be managed through a clustered index structure rather than nonclustered. The unique constraint's unique column list, or referential constraint's referencing column list, becomes the clustered key.
ConstraintID1	specifies the unique or referential constraint name on which clustering is to be applied. If not specified, the primary key of the table is assumed. *ConstraintID1* must be for a constraint being added with the ALTER TABLE statement.

SQL Syntax—DropConstraintSpecification

```
DROP CONSTRAINT {(ConstraintID [,...])
                 ConstraintID        }
```

Parameters—DropConstraintSpecification

ConstraintID	is the name of the constraint optionally defined when the

constraint was defined.

SQL Syntax—SetTypeSpecification

```
SET TYPE {PRIVATE
          PUBLICREAD
          PUBLIC
          PUBLICROW   }[RESET AUTHORITY
                        PRESERVE AUTHORITY]
```

Parameters—SetTypeSpecification

PRIVATE enables the table to be used by only one transaction at a time. Locks are applied at the table level. This is the most efficient option for tables that do not need to be shared because ALLBASE/SQL spends less time managing locks.

If RESET AUTHORITY is specified, the option automatically revokes all authorities on the table from PUBLIC. Otherwise, the authority on the table remains unchanged.

PUBLICREAD enables the table to be read by concurrent transactions, but allows no more than one transaction at a time to update the table. Locks are applied at the table level.

If RESET AUTHORITY is specified, the option automatically issues GRANT SELECT on *Owner.TableName* to PUBLIC, and revokes all other authorities on the table from PUBLIC. Otherwise, the authority on the table remains unchanged.

PUBLIC enables the table to be read and updated by concurrent transactions. The locking unit is a page. A transaction locks a page in share mode before reading it and in exclusive mode before updating it.

If RESET AUTHORITY is specified, the option automatically issues GRANT ALL on *Owner.TableName* to PUBLIC. Otherwise, the authority on the table remains unchanged.

PUBLICROW enables the table to be read and updated by concurrent transactions. The locking unit is a row. A transaction locks a row in share mode before reading it and in exclusive mode before updating it.

If RESET AUTHORITY is specified, the option automatically issues GRANT ALL on *Owner.TableName* to PUBLIC. Otherwise, the authority on the table remains unchanged.

RESET AUTHORITY is used to indicate that the authority on the table should be changed to reflect the new table type. If not specified, the authority on the table remains unchanged.

PRESERVE AUTHORITY is used to indicate that the authority currently in effect on the table should be preserved. This is the default.

SQL Syntax—SetPartitionSpecification

```
SET PARTITION {PartitionName
               DEFAULT
               NONE          }
```

Parameters—SetPartitionSpecification

PartitionName	specifies the new partition of the table.
DEFAULT	specifies the new partition of the table to be the default partition of the DBEnvironment. If the default partition number is later changed, that change will automatically be recorded the next time an INSERT, UPDATE, or DELETE operation is executed on the table. If the default partition is NONE at that time, audit logging of the operation is not done.
NONE	specifies that the table is no longer in any partition. No further audit logging will be done on the table.

Description

- Unless the table is currently empty, you cannot specify the NOT NULL attribute for any new columns unless you specify a default value.
- If no DEFAULT clause is given for an added column, an implicit DEFAULT NULL is assumed. Any INSERT statement which does not include a column for which a default has been declared causes the default value to be inserted into that column for all rows inserted.
- All rows currently in the table are updated with the default value for any new column which specifies default values.
- The ALTER TABLE statement can invalidate stored sections.
- Character strings are accepted as date/time default values.
- If an added constraint is violated when it is defined, an error message is immediately issued and the ALTER TABLE statement has no effect.
- A unique constraint referenced by a FOREIGN KEY cannot be dropped without first dropping the referential constraint.
- Constraints being added in AddConstraintSpecification must be on existing columns of the table.
- The ALTER TABLE statement can be used to change the type of an existing table. Changing the type of a table redefines the locking strategy that ALLBASE/SQL uses when the table is accessed. You can decide whether to use page or row level locking for your applications.
- No other transaction can access the table until the transaction that issued the ALTER TABLE statement has committed.
- The type of a table is changed permanently when you issue a COMMIT WORK statement.

- When altering the type of an existing table, you can also specify the option to preserve existing authority on the table or change the authority to the default for the new table type. If you specify RESET AUTHORITY, the following changes are made to the table authority:

Table 10-2. Changes to Table Authority in ALTER TABLE

Old Table Type	New Table Type	Changes to Authority
PRIVATE	PUBLIC	Grant ALL to PUBLIC
	PUBLICROW	Grant ALL to PUBLIC
	PUBLICREAD	Grant SELECT to PUBLIC
PUBLICREAD	PUBLIC	Grant ALL to PUBLIC
	PUBLICROW	Grant ALL to PUBLIC
	PRIVATE	Revoke ALL from PUBLIC
PUBLIC	PUBLICROW	No change
	PUBLICREAD	Revoke ALL from PUBLIC Grant SELECT to PUBLIC
	PRIVATE	Revoke ALL from PUBLIC
PUBLICROW	PUBLIC	No change
	PUBLICREAD	Revoke ALL from PUBLIC Grant SELECT to PUBLIC
	PRIVATE	Revoke ALL from PUBLIC

- To indicate that a table is in no partition, the partition NONE can be specified.
- The *PartitionName* specified must be one previously defined in a CREATE PARTITION statement, must be the DEFAULT partition, or must be specified as NONE.
- Changing the partition number of the table causes all future audit logging on the table to use the new partition number. Past audit log records will not be altered to reflect a change in a table's partition number; that is, the effect of this statement is not retroactively applied to existing log records. If NONE was specified, there will be no more audit logging done on this table (until another ALTER TABLE SET PARTITION statement is issued on the table).
- When specifiying CLUSTERING ON CONSTRAINT, an error is returned if the table is already clustered on a constraint or index or if the table is hashed.
- Adding a clustered constraint does not affect the physical placement of rows already in the table.
- See syntax for the CREATE TABLE and CREATE INDEX statements for more information on clustering.

Authorization

You can issue this statement if you have ALTER or OWNER authority for the table or if you have DBA authority.

To define added referential constraints, the table owner must have REFERENCES authority on the referenced table and referenced columns, own the referenced table, or have DBA authority.

To specify a *DBEFileSetName* for a long column, the table owner must have TABLESPACE authority on the referenced DBEFileSet.

To specify a *DBEFileSetName* for a check constraint, the section owner must have SECTIONSPACE authority on the referenced DBEFileSet.

Examples

Two new columns, ShippingWeight and PartDescription, are added to table PurchDB.Parts. ShippingWeight must be greater than 0.

```
ALTER TABLE PurchDB.Parts
      ADD (ShippingWeight DECIMAL(6,3) CHECK (ShippingWeight > 0)
      CONSTRAINT Check_Weight,
     PartDescription CHAR(40))
```

A constraint is added to table PurchDB.Parts to ensure that the sales price is greater than $100.

```
ALTER TABLE PurchDB.Parts
      ADD CONSTRAINT CHECK (SalesPrice > 100.) CONSTRAINT Check_Price
```

A column named DiscountPercent is added to table PurchDB.OrderItems, with a default value of 0 percent.

```
ALTER TABLE PurchDB.OrderItems
      ADD (DiscountPercent FLOAT DEFAULT 0)
```

The constraint named Check_Price is dropped.

```
ALTER TABLE PurchDB.Parts
      DROP CONSTRAINT Check_Price
```

The type of a table is changed:

```
ALTER TABLE PurchDB.OrderItems
      SET TYPE PUBLICROW
```

The table's partition is modified to be partition PartsPart2.

```
ALTER TABLE PurchDB.Parts
      SET PARTITION PartsPart2;
```

No more audit logging will be done on the table.

```
ALTER TABLE PurchDB.Parts
      SET PARTITION NONE;
```

Assignment (=)

The assignment statement is used in a procedure to assign a value to a local variable or procedure parameter.

Scope

Procedures only

SQL Syntax

```
{:LocalVariable
 :ProcedureParameter}= Expression;
```

Parameters

LocalVariable	identifies the local variable to which a value is being assigned. The variable name has a : prefix. Local variables are declared in the procedure definition using the DECLARE statement.
ProcedureParameter	identifies the procedure parameter to which a value is being assigned. The procedure parameter has a : prefix. Parameters are declared in parentheses following the procedure name in the procedure definition.
Expression	identifies an expression whose value is assigned to the local variable. The *Expression* may include anything that is allowed in an SQL expression except host variables, subqueries, column references, dynamic parameters, aggregate functions, date/time functions involving column references, string functions, TID functions, and long column functions. Local variables, built-in variables, and procedure parameters may be included. See Chapter 8 , “Expressions,” for more information.

Description

- Host variables are not allowed anywhere in procedures, including *Expressions* assigned to local variables or parameters. However, local variables, built-in variables, and parameters may be used in an *Expression* anywhere a host variable would be allowed in an application program.
- The data type of the expression result must be compatible with that of the parameter or variable to which it is being assigned.

Authorization

Anyone can use the assignment statement in a procedure definition.

Example

```
:msg = 'Vendor number found in "Orders" table.';
:SalesPrice = :OldPrice;
:NewPrice = :SalesPrice*.80;
:nrows = ::sqlerrd2;
```

BEGIN

The BEGIN statement is a compound statement and defines a group of statements within a procedure.

Scope

Procedures only

SQL Syntax

```
BEGIN [Statement;][...] END;
```

Parameters

Statement is the statement or statements between the begin and end of the statement.

Description

- This statement can be used to improve readability.

Authorization

Anyone can use the BEGIN statement.

Example

```
CREATE PROCEDURE PurchDB.DiscountPart(PartNumber CHAR(16))
AS BEGIN
      DECLARE SalesPrice DECIMAL(6,2);

      SELECT SalesPrice INTO :SalesPrice
        FROM PurchDB.Parts
       WHERE PartNumber = :PartNumber;

      IF ::sqlcode = 0 THEN
         IF :SalesPrice > 100. THEN
         BEGIN
            :SalesPrice = :SalesPrice*.80;
            INSERT INTO PurchDB.Discounts
               VALUES (:PartNumber, :SalesPrice);
         END
         ENDIF;
      ENDIF;
   END;
```

BEGIN ARCHIVE

The BEGIN ARCHIVE statement in conjunction with the COMMIT ARCHIVE statement starts a new archive log file before a static backup is done to a DBEnvironment. However, this method is no longer recommended. The recommended approach to initiate archive logging and dynamic backup is to use the SQLUtil STOREONLINE command.

Scope

ISQL or Application Programs

SQL Syntax

```
BEGIN ARCHIVE
```

Description

Use of the BEGIN ARCHIVE statement is no longer recommended. Refer to the *ALLBASE/SQL Database Administration Guide* for detailed backup and recovery procedures and recommended practices.

Authorization

You must have DBA authority to use this statement.

BEGIN DECLARE SECTION

The BEGIN DECLARE SECTION preprocessor directive indicates the beginning of the host variable eclaration section in an application program.

Scope

Application Programs Only

SQL Syntax

```
BEGIN DECLARE SECTION
```

Description

- This directive cannot be used interactively.
- Use this directive in conjunction with the END DECLARE SECTION directive.

Authorization

You do not need authorization to use the BEGIN DECLARE SECTION statement.

Example

You define host variables here, including indicator variables, if any.

```
BEGIN DECLARE SECTION

END DECLARE SECTION
```

BEGIN WORK

The BEGIN WORK statement begins a transaction and, optionally, sets one or more transaction attributes.

Scope

ISQL, Application Programs, or Procedures

SQL Syntax

```
BEGIN WORK [Priority][RR
                     CS
                     RC
                     RU ][LABEL {'LabelString'
                                 :HostVariable}][[PARALLEL
                                                  NO      ]FILL]
```

Parameters

Priority is an integer from 0 to 255 specifying the priority of the transaction. Priority 127 is assigned if you do not specify a priority. ALLBASE/SQL uses the priority to resolve a deadlock. The transaction with the largest priority number is aborted to remove the deadlock.

For example, if a priority-0 transaction and a priority-1 transaction are deadlocked, the priority-1 transaction is aborted. If two transactions involved in a deadlock have the same priority, the deadlock is resolved by aborting the newer transaction (the last transaction begun, either implicitly or with a BEGIN WORK statement).

RR Repeatable Read. Means that the transaction uses locking strategies to guarantee repeatable reads.

RR is the default isolation level.

CS Cursor Stability. Means that your transaction uses locking strategies to assure cursor-level stability only.

RC Read Committed. Means that your transaction uses locking strategies to ensure that you retrieve only rows that have been committed by some transaction.

RU Read Uncommitted. Means that the transaction can read uncommitted changes from other transactions. Reading data with RU does not place any locks on the table being read.

LabelString is a user defined character string of up to 8 characters. The default is a blank string.

The label is visible in the SYSTEM.TRANSACTION pseudo-table and also in SQLMON. Transaction labels can be useful for troubleshooting and performance tuning. Each transaction in an application program can be

marked uniquely, allowing the DBA to easily identify the transaction being executed by any user at any moment.

HostVariable is a host variable containing the *LabelString*.

FILL is used to optimize I/O performance when loading data and creating indexes.

PARALLEL FILL is used to optimize I/O performance for multiple, concurrent loads to the same table. The PARALLEL FILL option must be in effect for each load.

NO FILL turns off the FILL or PARALLEL FILL option for the duration of the transaction. This is the default fill option.

Description

- Detailed information about isolation levels is presented in the "Concurrency Control through Locks and Isolation Levels" chapter.
- When you use most SQL statements, ISQL or the preprocessor automatically issues the BEGIN WORK statement on your behalf, unless a transaction is already in progress. However, to clearly delimit transaction boundaries and to set attributes for a transaction (isolation level, priority, transaction label, and fill options), you can use explicit BEGIN WORK statements.

  The following statements do *not* force an automatic BEGIN WORK to be processed:

```
ASSIGN                  BEGIN ARCHIVE           BEGIN DECLARE SECTION
BEGIN WORK              CHECKPOINT              COMMIT ARCHIVE
COMMIT WORK             CONNECT                 DECLARE VARIABLE
DISABLE AUDIT LOGGING   ENABLE AUDIT LOGGING    END DECLARE SECTION
ASSIGN                  BEGIN ARCHIVE           BEGIN DECLARE SECTION
BEGIN WORK              CHECKPOINT              COMMIT ARCHIVE
COMMIT WORK             CONNECT                 DECLARE VARIABLE
DISABLE AUDIT LOGGING   ENABLE AUDIT LOGGING    END DECLARE SECTION
GOTO                    IF                      INCLUDE
PRINT                   RAISE ERROR             RELEASE
RESET                   RETURN                  ROLLBACK TO SAVEPOINT
ROLLBACK WORK           SET SESSION             SET TIMEOUT
SET TRANSACTION         START DBE               STOP DBE
SQLEXPLAIN              TERMINATE USER          WHENEVER
WHILE
```

- See Chapter 2 , “Using ALLBASE/SQL,” "Scoping of Transaction and Session Attributes" section for information about statements used to set transaction attributes.
- Within a given transaction, the isolation level, priority, and label can be changed by issuing a SET TRANSACTION statement. Attributes specified in a SET TRANSACTION statement within a transaction override any attributes set by a BEGIN WORK statement for the same transaction.
- An application or ISQL can have one or more active transactions at a time. Refer to the SET MULTITRANSACTION statement syntax in this chapter.

- The following sequences of statements must be in the same transaction in a program:

  PREPARE and EXECUTE

  PREPARE, DESCRIBE, OPEN, FETCH USING DESCRIPTOR, EXECUTE, and CLOSE

  OPEN, FETCH, DELETE WHERE CURRENT, UPDATE WHERE CURRENT, and CLOSE (unless KEEP CURSOR is used)

- To end your transaction, you must issue a COMMIT WORK or ROLLBACK WORK statement. Otherwise, locks set by your transaction are held until a STOP DBE, DISCONNECT, RELEASE, or TERMINATE USER statement is processed.
- If the maximum number of concurrent DBEnvironment transactions has been reached, the application is placed on a wait queue. If the application times out while waiting, an error occurs. Default and maximum timeout values are specified at the DBEnvironment level. To set a timeout for a session or transaction, use the SET USER TIMEOUT statement. Refer to Chapter 2 , “Using ALLBASE/SQL,” "Setting Timeout Values" section for further information.
- To avoid lock contention in a given DBEnvironment, do not allow simultaneous transactions when performing data definition operations.
- When using RC or RU, you should verify the existence of a row before you issue an UPDATE statement. In application programs that employ cursors, you can use the REFETCH statement prior to updating. REFETCH is not available in ISQL. Therefore, you should use caution in employing RC and RU in ISQL if you are doing updates.
- If the FILL or PARALLEL FILL option has already been set for the session with a SET SESSION statement, and you do not want either of these options in effect for a given transaction, specify NO FILL in the transaction's BEGIN WORK statement.

Authorization

You do not need authorization to use the BEGIN WORK statement.

Examples

1. BEGIN WORK and ROLLBACK WORK

   Transaction begins:

   ```
   BEGIN WORK CS
   statement-1
   SAVEPOINT :MyVariable
   statement-2
   statement-3
   ```

   Work of statements 2 and 3 is undone:

   ```
   ROLLBACK WORK TO :MyVariable
   ```

   Work of statement-1 is committed and the transaction ends:

   ```
   COMMIT WORK
   ```

2. `BEGIN WORK` and set attributes

   Begin the transaction and set priority, isolation level, label name, and fill option:

```
BEGIN WORK 32 CS LABEL 'xact1' FILL
.
.
.
Execute SQL statements.
.
.
.
```

   Work is committed and the transaction ends.

```
COMMIT WORK
```

   Begin another transaction and set priority, isolation level, and label name. Note that since a fill option is not specified, the default (NO FILL) is in effect.

```
BEGIN WORK 64 RC LABEL 'xact2'
.
.
.
Execute SQL statements.
.
.
.
```

   Work is committed and the transaction ends.

```
COMMIT WORK;
```

CHECKPOINT

The CHECKPOINT statement causes an ALLBASE/SQL system checkpoint to be taken.

Scope

ISQL or Application Programs

SQL Syntax

```
CHECKPOINT [:HostVariable
            :LocalVariable
            :ProcedureParameter]
```

Parameters

HostVariable	identifies an output host variable used to communicate the amount of log space available for use. The host variable is an integer.
LocalVariable	contains a value in a procedure.
ProcedureParameter	contains a value that is passed into or out of a procedure.

Description

- Specifying a host variable with CHECKPOINT statement in an application allows you to determine how much free space is available in the log file.
- The *LocalVariable* parameter is used in the stored procedure for obtaining free log space.
- When you can use the host variable in a CHECKPOINT statement in an application program or procedure, the host variable can be omitted if you don't need to know the number of free blocks available.
- When you enter a CHECKPOINT statement interactively in ISQL, you cannot specify a host variable. Returned information is displayed on the screen.
- Checkpoint processing is as follows:
  - Contents of the log buffers are written to the log files(s).
  - Data buffers containing changed pages are written to DBEFiles.
  - A checkpoint record containing a list of the transactions currently in progress is written in the log.
  - When nonarchive logging is in effect, space containing log records written prior to the beginning of the oldest incomplete transaction is made available for reuse. When archive logging is in effect, however, this step is skipped and no log file space is recovered by checkpoints.

  - For a brief interval while a checkpoint is being taken, SQL statements that modify the DBEnvironment continue to be accepted but their processing is temporarily suspended. This suspension occurs for the amount of time needed to write the log buffers and changed pages to permanent storage. Retrieval from the DBEnvironment is not suspended during a checkpoint.
- Contents of the log buffer are also written to the log file(s) when a `COMMIT WORK` is executed.
- When you submit a `START DBE` statement, ALLBASE/SQL processes all log records created since the last checkpoint record. Therefore taking a checkpoint just before stopping the DBE reduces the amount of time that is needed when a DBEnvironment is started up.
- ALLBASE/SQL automatically takes a checkpoint when the log file is full, when the data buffer is full, and when the `STOP DBE` and `COMMIT ARCHIVE` statements are processed. When the `START DBE` statement is processed, ALLBASE/SQL writes a checkpoint record.
- Submitting a `CHECKPOINT` statement allows you to determine how much free space is available in the log file.

Authorization

You must have DBA authority to use this statement.

Example

A stored procedure retrieves the number of free blocks of log space available. Create a stored procedure with an output parameter.

```
EXEC SQL create procedure cp (freeblock integer OUTPUT) as
         begin
          checkpoint :freeblock;
         end;
```

Pass the host variable as an output parameter to procedure.

```
EXEC SQL execute procedure cp (hstfblk output);

writeln('free log space available', hstfblk);
if hstfblk <= TOOLOW then
writeln('Add new log files ');
```

A log block is a 512-byte allocation of storage. When you submit the `CHECKPOINT` statement interactively, ISQL displays the amount of log space available for use.

```
isql=> CHECKPOINT;
Number of free log blocks is 240
isql=>
```

ISQL assigns and displays the free log space.

A program retrieves the number of free blocks of log space available. In a Pascal application program, declare a host variable.

```
EXEC SQL begin declare section;
  hstfblk : integer;
EXEC SQL end declare section;
```

Submit a checkpoint with host variable to obtain free log space available.

```
EXEC SQL checkpoint :hstfblk;

writeln('free log space: ',hstfblk);
if hstfblk <= TOOLOW then
writeln('Add new log files ');
```

CLOSE

The CLOSE statement is used to close an open cursor.

Scope

Application Programs or Procedures

SQL Syntax

```
CLOSE CursorName [USING {[SQL]DESCRIPTOR {SQLDA
                                          Areaname}
                         :HostVariable [[INDICATOR]:Indicator][,...]}]
```

Parameters

CursorName		designates the open cursor to be closed.
USING		defines where to place return status and output parameters after closing a dynamic procedure cursor.
HostVariable		identifies a host variable for holding return status and output parameters after closing a dynamic procedure cursor. These must be specified in the same order as in the associated EXECUTE PROCEDURE statement.
Indicator		names the indicator variable, an output host variable whose value depends on whether the host variable contains a null value. The following integer values are svalid:
	0	meaning the output parameter's value is not null
	–1	meaning the output parameter's value is null
	>0	meaning the output parameter's value is truncated (for CHAR, VARCHAR, BINARY, and VARBINARY columns)
DESCRIPTOR		defines where to place return status and output parameters after closing a procedure cursor. Specify the same location (SQLDA, area name, or host variable) as you specified in the DESCRIBE OUTPUT statement.
SQLDA		specifies that a data structure of sqlda_type named SQLDA is to be used to pass information about the prepared statement between the application and ALLBASE/SQL.
AreaName		specifies the user defined name of a data structure of sqlda_type that is to be used to pass information about the prepared statement.

Description

- When it applies to a **select cursor** (one that is declared for a SELECT statement), the CLOSE statement can be issued in an application program or in a procedure.

  When it applies to a **procedure cursor** (one that is declared for an EXECUTE PROCEDURE statement), the CLOSE statement can be issued only in an application program.
- The CLOSE statement cannot be used in ISQL.
- CLOSE returns an error if the cursor is not in the open state.
- The COMMIT WORK and ROLLBACK WORK statements automatically close all cursors not opened with the KEEP CURSOR option.
- To close a select cursor opened with the KEEP CURSOR option, you must perform an explicit CLOSE followed by a COMMIT WORK.
- When you close a select cursor, its active set becomes undefined, and it can no longer be used in DELETE, FETCH, or UPDATE statements. To use the cursor again you must reopen it by issuing an OPEN statement.
- When you close a procedure cursor, its active result set becomes undefined, and it can no longer be used in FETCH statements. To use the procedure cursor again you must reopen it by issuing an OPEN statement.
- When used with a procedure cursor, CLOSE discards any pending rows or result sets from the procedure. Execution of the procedure continues with the next statement. Control returns to the application when the procedure terminates.

  Note that following processing of the last multiple row result set, procedure execution cannot continue until you close or advance the procedure cursor in the application.
- Upon execution of the CLOSE statement used with a procedure cursor, return status and output parameter values are available to the application in either the SQLDA or the *HostVariableSpecification* of the USING clause or in any host variables specified in the related DECLARE CURSOR statement.
- The USING clause is allowed only for dynamic procedure cursors.

Authorization

You do not need authorization to use the CLOSE statement.

Examples

Declare and open a cursor for use in updating values in column QtyOnHand.

```
        DECLARE NewQtyCursor CURSOR FOR
         SELECT PartNumber,QtyOnHand FROM PurchDB.Inventory
   FOR UPDATE OF QtyOnHand

   OPEN NewQtyCursor
```

Statements setting up a FETCH-UPDATE loop appear next.

```
   FETCH NewQtyCursor INTO :Num :Numnul, :Qty :Qtynul
```

Statements for displaying a row to a user and accepting a new QtyOnHand value go here. The new value is stored in :NewQty.

```
            UPDATE PurchDB.Inventory
               SET QtyOnHand = :NewQty
   WHERE CURRENT OF NewQtyCursor
     .
     .
     .
   CLOSE NewQtyCursor USING sqldaout
```

COMMIT ARCHIVE

The COMMIT ARCHIVE statement in conjunction with the BEGIN ARCHIVE statement starts a new archive log file before a static backup is done to a DBEnvironment. However, this method is no longer recommended. The recommended approach to initiate archive logging and do a dynamic backup is to use the SQLUtil STOREONLINE command.

Scope

ISQL or Application Programs

SQL Syntax

```
COMMIT ARCHIVE
```

Description

- Use of the COMMIT ARCHIVE statement is no longer recommended.

Refer to the *ALLBASE/SQL Database Administration Guide* for detailed backup and recovery procedures and recommended practices.

Authorization

You must have DBA authority to use this statement.

COMMIT WORK

The COMMIT WORK statement ends the current transaction. All changes made during the transaction are committed (made permanent).

Scope

ISQL or Application Programs

SQL Syntax

```
COMMIT WORK [RELEASE]
```

Parameters

RELEASE	terminates your DBE session after the changes made during the transaction are committed. Specifying RELEASE has the same effect as issuing a COMMIT WORK statement followed by a RELEASE statement.

Description

- The COMMIT WORK statement has no effect if you do not have a transaction in progress.
- The COMMIT WORK statement releases all locks held by the transaction, except those associated with a kept cursor in an application program.
- In an application program, the COMMIT WORK statement closes all cursors opened without the KEEP CURSOR option in the current transaction.
- For cursors opened with the KEEP CURSOR option, the COMMIT WORK statement (but not the COMMIT WORK RELEASE statement) implicitly starts a new transaction that maintains the current cursor position and inherits the isolation level. Whether or not locks on data objects pointed to by these cursors are released depends on the use of the WITH LOCKS or WITH NOLOCKS option in the OPEN statement.
- If a procedure invoked by a rule executes a COMMIT WORK statement, an error occurs.
- If a commit is done while constraints are deferred, and constraint errors exist, the system will roll back the transaction and report that constraint errors exist.
- Short transactions (frequent COMMIT WORK statements) are recommended to improve concurrency.
- If RELEASE is used, all cursors are closed and the current connection is terminated
- The RELEASE option is not allowed within a procedure.

Authorization

You do not need authorization to use the COMMIT WORK statement.

Example

Transaction begins.

```
BEGIN WORK
```

statement-1

```
SAVEPOINT :MyVariable
```

statement-2

statement-3

Work of statements 2 and 3 is undone.

```
ROLLBACK WORK TO :MyVariable
```

Work of statement 1 is committed; the transaction ends.

```
COMMIT WORK
```

CONNECT

The CONNECT statement initiates a connection with a given DBEnvironment. This connection is the current connection. Any SQL statements issued apply to the current connection.

Scope

ISQL or Application Programs

SQL Syntax

```
CONNECT TO {'DBEnvironmentName'
            :HostVariable1      }[AS {'ConnectionName'
                                      :HostVariable2   }]
[USER {'UserID'
       :HostVariable3}[USING :HostVariable4]]
```

Parameters

DBEnvironmentName	identifies the DBEnvironment to be used. Any path name you specify, unless absolute, is assumed to be relative to your current working directory.
HostVariable1	is a character string host variable containing the name of a DBEnvironment.
ConnectionName	is a string literal identifying the name associated with this connection. This name must be unique for each DBEnvironment connection within an application or an ISQL session. If a *ConnectionName* is not specified, *DBEnvironmentName* is the default. *ConnectionName* cannot exceed 128 bytes.
HostVariable2	is a character string host variable containing the *ConnectionName* associated with this connection.
UserID	is a string literal identifying the user associated with this connection. *UserID* cannot exceed 64 bytes.
HostVariable3	is a character string host variable containing the *UserID* associated with this connection.
HostVariable4	is a character string host variable containing the connection password associated with the specified user identifier. The connection password assigned to *HostVariable4* cannot exceed 64 bytes.

Description

- ALLBASE/SQL creates an implicit, brief transaction when the CONNECT statement is issued.
- When the value of the autostart flag is ON, the CONNECT statement initiates a single-user DBE session if the DBECon file user mode is currently set to *single* and no other user is accessing the DBEnvironment. A multiuser DBE session is established if the DBECon file user mode is currently set to MULTI.
- If the value of the autostart flag is OFF, the CONNECT statement is used to initiate a multiuser session after a START DBE statement has been processed.
- When more than one CONNECT statement is issued, the application (or ISQL) is currently connected to the DBEnvironment specified by the most recent CONNECT statement. The current connection can be changed with the SET CONNECTION statement.
- The USER and USING clauses are implementation-defined features intended for use in determining if a CONNECT statement should be accepted or rejected. They are *not* currently used by ALLBASE/SQL as criteria for accepting or rejecting a CONNECT statement. However, other database products in a network environment may require them in order to granulize authorization to a connection level.

Authorization

You can use this statement if you have CONNECT or DBA authority for the specified DBEnvironment.

Example

A user whose current working directory is just above the sampledb directory begins a DBE session; the value of the autostart mode is ON. The PartsDBE DBEnvironment is currently configured to operate in multiuser mode, so other users can also initiate DBE sessions.

```
CONNECT TO 'sampledb/PartsDBE'
```

A second user starts a DBE session from a different directory.

```
CONNECT TO '../sampledb/PartsDBE'
```

Specifying a connection name

```
CONNECT TO 'sampledb/PartsDBE' AS 'Parts1'
```

Parts1 is the connection name to be used with multiconnect functionality.

CREATE DBEFILE

The CREATE DBEFILE statement defines and creates a DBEFile and places a row describing the file in SYSTEM.DBEFile. A DBEFile is a file that stores tables, indexes, hash structures, and/or LONG data.

Scope

ISQL or Application Programs

SQL Syntax

```
CREATE DBEFILE DBEFileName WITH PAGES = DBEFileSize, NAME = 'SystemFileName'
[, INCREMENT = DBEFileIncrSize[, MAXPAGES = DBEFileMaxSize]]
[, TYPE = {TABLE
           INDEX
           MIXED}]
```

Parameters

DBEFileName	is the logical name to be assigned to the new DBEFile. Two DBEFiles in one DBEnvironment cannot have the same logical name.
DBEFileSize	specifies the number of 4096-byte pages in the new DBEFile. The minimum DBEFile size is 2 pages. The maximum DBEFile size is 524,287 pages.
SystemFileName	identifies how the DBEFile is known to the operating system. The system file name is in the format [*Pathname/*]*FileName*. The DBEFile is created relative to the directory where the DBECon file resides unless an absolute path name is specified. The maximum length for *SystemFileName* is 44 bytes.
DBEFileIncrSize	is a number you must supply with the INCREMENT clause when you want to expand the DBEFILE. The *DBEFileIncrSize* should be 8 pages or greater but it cannot exceed 65,535. No system default is provided by ALLBASE/SQL; if this number is omitted, no DBEFile expansion takes place.
DBEFileMaxSize	is a number that you can supply with the MAXPAGES clause if you have already specified a *DBEFileIncrSize*. If the *DBEFileMaxSize* is not a multiple of *DBEFileIncrSize*, the number may be rounded up or down as follows: The smallest higher multiple is tried first. If the smallest higher multiple is not a valid size, the largest lower multiple is used. A warning message is returned to let you know that the *DBEFileMaxSize* is

		rounded based on the *DBEFileIncrSize* provided. If you omit the MAXPAGES clause, the value defaults to the ALLBASE/SQL DBEFile maximum size.
TYPE =		specifies the setting of the DBEFile's TYPE attribute. The following are valid settings:
	TABLE	Only data pages (table, HASH, or LONG) can be stored in the DBEFile.
	INDEX	Only index pages can be stored in the DBEFile.
	MIXED	A mixture of data and index pages can be stored in the DBEFile.

Description

- You use this statement to create all DBEFiles except DBEFile0, which is created when a START DBE NEW statement is processed.
- The CREATE DBEFILE statement formats the DBEFile. The name and characteristics of the DBEFile are stored in the system catalog.
- The DBEFile created is owned by hpdb and has the following permissions:

```
-rw-----
```

- To use a DBEFile for storing a table, LONG data, and/or an index, you add it to a DBEFileSet with the ADD DBEFILE statement, then reference the name of the DBEFileSet in the CREATE TABLE statement. You may add a DBEFile to the SYSTEM DBEFileSet.
- To delete the row describing a DBEFile from SYSTEM.DBEFile, use the DROP DBEFILE statement.
- INCREMENT and MAXPAGES are optional clauses. If they are omitted, no DBEFile expansion takes place.
- It is highly recommended that you provide the *DBEFileMaxSize* along with the *DBEFileIncrSize*. Not specifying the *DBEFileMaxSize* causes it to be set to the system maximum. This results in a high value for the ratio for this file. The *DBEFileMaxSize* is stored internally as an integer multiple of the *DBEFileIncrSize*; if the *DBEFileMaxSize* is not a multiple of *DBEFileIncrSize*, rounding can occur. Refer to the description of *DBEFileMaxSize* in the previous section for information on the rounding process.
- The *DBEFileMaxSize*, after rounding, should be equal to or greater than the *DBEFileSize*. It should not exceed the maximum DBEFile size of 524,287 pages.
- The optimal *DBEFileIncrSize* depends on the expected rate of expansion for the file. Refer to the section "Calculating Storage for Database Objects" in the *ALLBASE/SQL Database Administration Guide* for information about estimating size requirements for tables and indexes.
- Expandable DBEFiles do not expand dynamically during the creation of hash tables.

- DBEFiles that contain hash tables are not expanded even though they were specified as expandable when created.

Authorization

You must have DBA authority to use this statement. hpdb must have write permission in the directory where the DBEFile will reside.

Example

```
CREATE DBEFILE ThisDBEFile\
              WITH PAGES = 4, NAME = 'ThisFile', TYPE = TABLE

CREATE DBEFILESET Miscellaneous

ADD DBEFILE ThisDBEFile TO DBEFILESET Miscellaneous
```

The DBEFile is used to store rows of a new table. When the table needs an index, a DBEFile is created to store an index:

```
CREATE DBEFILE ThatDBEFile\
            WITH PAGES = 4, NAME = 'ThatFile', TYPE = INDEX

ADD DBEFILE ThatDBEFile TO DBEFILESET Miscellaneous
```

When the index is subsequently dropped, its file space can be assigned to another DBEFileSet.

```
REMOVE DBEFILE ThatDBEFile FROM DBEFILESET Miscellaneous

ADD DBEFILE ThatDBEFile TO DBEFILESET SYSTEM

ALTER DBEFILE ThisDBEFile SET TYPE = MIXED
```

All rows are later deleted from the table, so you can reclaim file space.

```
REMOVE DBEFILE ThisDBEFile FROM DBEFILESET Miscellaneous

DROP DBEFILE ThisDBEFile
```

The DBEFileSet definition can now be dropped.

```
DROP DBEFILESET Miscellaneous

CREATE DBEFILE NewDBEFile\
            WITH PAGES = 4, NAME = 'ThatFile', TYPE = INDEX

ADD DBEFILE NewDBEFile TO DBEFILESET SYSTEM
```

CREATE DBEFILESET

The CREATE DBEFILESET statement defines a DBEFileSet. A DBEFileSet is a group of related DBEFiles; as such, it serves as a mechanism for allocating and deallocating file space for tables.

Scope

ISQL or Application Programs

SQL Syntax

CREATE DBEFILESET *DBEFileSetName*

Parameters

DBEFileSetName specifies the name to be given to the new DBEFileSet. Two DBEFileSets in the same DBEnvironment cannot have the same name.

Description

- The CREATE DBEFILESET statement records the new DBEFileSet name in the system catalog with an indication that no physical storage is associated with the DBEFileSet.
- You associate physical storage with a DBEFileSet by associating DBEFiles with the DBEFileSet, using the ADD DBEFILE statement. Then you can associate a table and its indexes with the DBEFileSet by using the CREATE TABLE statement. ALLBASE/SQL allocates all data and index pages for a table to DBEFiles in the DBEFileSet named in the IN clause of the CREATE TABLE statement. If automatic DBEFile expansion is not being used when you need more space for a table, you add another DBEFile to the DBEFileSet associated with the table when the CREATE TABLE statement was issued.
- To remove a DBEFile from a DBEFileSet, you use the REMOVE DBEFILE statement.
- If a LONG column uses the IN *DBEFileSet* clause, ALLBASE/SQL allocates all LONG data pages for that column in DBEFiles in the DBEFileSet specified. If automatic DBEFile expansion is not being used when more space is needed for the LONG column, you add another DBEFile to the DBEFileSet associated with the LONG column when the column was defined.
- To delete the definition of a DBEFileSet, use the DROP DBEFILESET statement.
- One DBEFileSet is created automatically when the START DBE NEW statement is issued -- the SYSTEM DBEFileSet. The system catalog resides in the SYSTEM DBEFileSet. Those parts of the system catalog that are needed to start up a DBEnvironment reside in DBEFile0. You may add a DBEFile to the SYSTEM DBEFileSet.

Authorization

You must have DBA authority to use this statement.

Example

The DBEFile is used to store rows of a new table.

```
CREATE DBEFILE ThisDBEFile WITH PAGES = 4,
               NAME = 'ThisFile', TYPE = TABLE

CREATE DBEFILESET Miscellaneous

ADD DBEFILE ThisDBEFile TO DBEFILESET Miscellaneous
```

When the table needs a DBEFile to hold an index, one is created as follows:

```
CREATE DBEFILE ThatDBEFile WITH PAGES = 4,
               NAME = 'ThatFile', TYPE = INDEX

ADD DBEFILE ThatDBEFile TO DBEFILESET Miscellaneous
```

When the index is subsequently dropped, its file space can be assigned to another DBEFileSet.

```
REMOVE DBEFILE ThatDBEFile
FROM DBEFILESET Miscellaneous

  ADD DBEFILE ThatDBEFile
TO DBEFILESET SomethingElse

ALTER DBEFILE ThisDBEFile SET TYPE = MIXED
```

Now you can use this DBEFile to store an index later if you need one.

All rows are later deleted from the table, so you can reclaim file space.

```
 REMOVE DBEFILE ThisDBEFile
FROM DBEFILESET Miscellaneous

DROP DBEFILE ThisDBEFile
```

The DBEFileSet definition can now be dropped.

```
DROP DBEFILESET Miscellaneous

CREATE DBEFILE NewDBEFile

  ADD DBEFILE NewDBEFile
TO DBEFILESET SYSTEM
```

CREATE GROUP

The CREATE GROUP statement defines a new authorization group.

Scope

ISQL or Application Programs

SQL Syntax

CREATE GROUP [*Owner.*]*GroupName*

Parameters

[*Owner.*]*GroupName* specifies the group name to be assigned to the new authorization group. The group name must conform to the syntax rules for basic names, described in the "Names" chapter.

You can specify the owner of the new group if you have DBA authority. Non-DBA users can specify as owner the name of any group of which they are a member. If you do not specify the owner name, your login name becomes the owner of the new group.

Although the owner name can be specified as a prefix to the group name in this statement, the owner name is not actually considered a part of the group identifier. The group name by itself uniquely identifies a group within the database.

The group name you specify cannot be the same as any of the following names:

- Name of an existing authorization group.
- Owner name of an existing table, view, module, or authorization group.
- DBEUserID existing in the authorization tables of the system catalog.
- DBEUserID associated with any DBE session currently in progress.
- Special names PUBLIC, SYSTEM, CATALOG, HPRDBSS, STOREDSECT, SEMIPERM, HPODBSS, and TEMP.

Description

- When you create an authorization group, its owner name and group name are entered into the system catalog. You can then refer to the group in the ADD TO GROUP, REMOVE FROM GROUP, GRANT, REVOKE, TRANSFER OWNERSHIP, and DROP GROUP statements.

Authorization

You must have RESOURCE or DBA authority to use this statement.

Example

```
CREATE GROUP Warehse

GRANT CONNECT TO Warehse

GRANT SELECT,
      UPDATE (BinNumber,QtyOnHand,LastCountDate)
   ON PurchDB.Inventory
   TO Warehse
```

These two users will be able to start DBE sessions for PartsDBE, retrieve data from table PurchDB.Inventory, and update three columns in the table.

```
ADD Clem, George TO GROUP Warehse
```

Clem will no longer have any of the authorities associated with group Warehse.

```
REMOVE Clem FROM GROUP Warehse
```

Because this group does not own any database objects, it can be deleted. George no longer has any of the authorities once associated with the group.

```
DROP GROUP Warehse
```

CREATE INDEX

The CREATE INDEX statement creates an index on one or more columns of a table and assigns a name to the new index.

Scope

ISQL or Application Programs

SQL Syntax

```
CREATE [UNIQUE][CLUSTERING]INDEX [Owner.]Indexname ON
[Owner.]TableName ({ColumnName[ASC
                               DESC]}[,...])
```

Parameters

UNIQUE — prohibits duplicates in the index. If UNIQUE is specified, each possible combination of index key column values can occur in only one row of the table. If UNIQUE is omitted, duplicate values are allowed. Because all null values are equivalent, a unique index allows only one row with a null value in an indexed column. When you create a unique index, all existing rows must have unique values in the indexed column(s).

CLUSTERING — can increase the efficiency of sequential processing.

If CLUSTERING is specified, rows added to the table after the index is created are placed physically near other rows with similar key values whenever space is available in the page. If CLUSTERING is omitted, the key values in a newly inserted row do not necessarily have any relationship with the row's physical placement in the database.

No more than one index for a table can have the CLUSTERING attribute.

If the table was declared to use a HASH structure, no clustering indexes may be defined upon it. See the CREATE TABLE statement for information on HASH structures.

[*Owner.*]*IndexName* is the name to be assigned to the new index. A table cannot have two indexes with the same name. If the owner is specified, it must be the same as the owner of the table. The default owner name is the owner name of the table it is being defined on. The usual default owner rules do not apply here.

[*Owner.*]*TableName* designates the table for which an index is to be created.

ColumnName is the name of a column to be used as an index key. You can specify up to 16 columns in order from major index key to minor index key. The data type of the column cannot be a LONG data type.

ASC | DESC specifies the order of the index to be either ascending or descending, respectively. The default is ascending. Specifying DESC does not create a

descending index. It is the same index as ascending. Therefore, SELECT statements that require data to be retrieved in descending order must specify ORDER BY *columnID* DESC.

Description

- If the table does not contain any rows, the CREATE INDEX statement enters the definition of the index in the system catalog and allocates a root page for it. If the table has rows, the CREATE INDEX statement enters the definition in the system catalog and builds an index on the existing data.

  If the UNIQUE option is specified and the table already contains rows having duplicate values in the index key columns, the CREATE INDEX statement is rejected.

  The CLUSTERING option does not affect the physical placement of rows that are already in the table when the CREATE INDEX statement is issued.
- The new index is maintained automatically by ALLBASE/SQL until the index is deleted by a DROP INDEX statement or until the table it is associated with is dropped.
- The following equation determines the maximum key size for a B-tree or hash index:

  ```
  (NumberOfIndexColumns + 2)*2 + SumKeyLengths + 8 <= 1024
  ```

  If the index contains only one column, the maximum length that column can be is 1010 bytes. At compile time, *SumKeyLengths* is computed assuming columns of NULL and VARCHAR columns contain no data. At run time, the actual data lengths are assumed.

  At most 16 columns are allowed in a user-defined index.
- Indexes cannot be created for views, including the system views and pseudotables.
- Index entries are sorted in ascending order. Null compares higher than other values for sorting.
- An index is automatically stored in the same DBEFileSet as its table.
- The CREATE INDEX statement can invalidate stored sections. Refer to the *ALLBASE/SQL Database Administration Guide* for additional information on section validation.
- The CREATE INDEX statement allocates file space for sorting under any available TempSpace location, or in the default sort space. After the index has been created, this file space is deallocated.
- Indexes created with the CREATE INDEX statement are not associated with referential or unique constraints in any manner, and are not used to support any constraints. So a unique index created with the CREATE INDEX statement cannot be referenced as a primary key in a referential constraint.

Authorization

You can issue this statement if you have INDEX or OWNER authority for the table or if you have DBA authority.

Example

This unique index ensures that all part numbers are unique.

```
CREATE UNIQUE INDEX PurchDB.PartNumIndex
                  ON PurchDB.Parts (PartNumber)
```

This clustering index causes rows for order items associated with one order to be stored physically close together.

```
CREATE CLUSTERING INDEX OrderItemIndex
                      ON PurchDB.OrderItems (OrderNumber)
```

CREATE PARTITION

The CREATE PARTITION statement defines a partition to be used for audit logging purposes.

Scope

ISQL or Application Programs

SQL Syntax

```
CREATE PARTITION PartitionName WITH ID = PartitionNumber
```

Parameters

PartitionName specifies the logical name to be given to the new partition. Two partitions in the same DBEnvironment cannot have the same name. *PartitionName* may not be DEFAULT or NONE.

PartitionNumber is an integer specifying the partition number. It must be a positive integer in the range 1 to 32767. The partition number identifies the partition in the audit log record.

Description

- The CREATE PARTITION statement creates a new audit partition, which is a unit of data logging for an audit DBEnvironment.
- The partition number may already be assigned to another partition, including the default partition. For example, several partitions with different partition names may have the same partition number in the audit log file. This allows the Audit Tool to gather statistics for all of these partitions as one unit while preserving the ability to manipulate each partition separately.
- Creation of a partition does not cause a check against the maximum number of partitions. Only creation of audit log records in a partition checks if the maximum number of partitions is exceeded. The process of determining the number of partitions in a DBEnvironment is described under the START DBE NEW statement.
- One data partition can be defined with the START DBE NEW or START DBE NEWLOG statements -- the DEFAULT partition. Before tables are assigned to a particular partition, they are placed in the DEFAULT partition.
- To put a table in a partition, use the CREATE TABLE or ALTER TABLE SET PARTITION statement.
- To remove a table from a partition, or change the partition it is in, use the ALTER TABLE SET PARTITION statement.
- To delete the definition of a partition, use the DROP PARTITION statement.
- Partitions can be created and tables placed in them without audit logging being enabled

for a DBEnvironment. However, the partition information is only used in audit log records. Thus, partition information will not be utilized in logging until the DBEnvironment has audit logging enabled.

- Data partition information (including the default partition) appears in the system view SYSTEM.PARTITION. If the default partition is set to NONE, or is never defined, no row appears in SYSTEM.PARTITION for it.
- The `DROP PARTITION` and `CREATE PARTITION` statements are used to change the partition number assigned to a partition other than the default partition. The `START DBE NEWLOG` statement is used to change the partition number of the default partition.
- The partition number, not the partition name, is used in audit logging. A partition name is used in the `CREATE TABLE` and `ALTER TABLE` statements to associate a table with a partition.

Authorization

You must have DBA authority to use this statement.

Example

To create a partition containing tables, first create the partition.

```
CREATE PARTITION PartsPart WITH ID = 10;
```

Then assign tables(s) to the partition.

```
ALTER TABLE PurchDB.Parts SET PARTITION PartsPart;
```

To drop a partition, first assign all tables in the partition to the NONE partition.

```
ALTER TABLE PurchDB.Parts SET PARTITION NONE;
```

Then drop the partition.

```
DROP PARTITION PartsPart;
```

CREATE PROCEDURE

The CREATE PROCEDURE statement defines a procedure for storage in a DBEnvironment. A procedure may subsequently be executed through the firing of a rule by an INSERT, UPDATE, or DELETE statement, or by using the EXECUTE PROCEDURE statement or a procedure cursor.

Scope

ISQL or Application Programs

SQL Syntax

```
CREATE PROCEDURE [Owner.]ProcedureName [LANG = ProcLangName]
[(ParameterDeclaration [, ParameterDeclaration][...])]
[WITH RESULT ResultDeclaration [, ResultDeclaration ][...]]
AS BEGIN [ProcedureStatement][...] END [IN DBEFileSetName]
```

Parameters

[*Owner*.]*ProcedureName* specifies the owner and the name of the procedure. If an owner name is not specified, the owner is the current user's DBEUserID or the schema's authorization name, or the ISQL SET OWNER value. You can specify the owner of the new procedure if you have DBA authority. If you do not have DBA authority, you can specify as owner the name of any group of which you are a member. Two procedures cannot have the same owner and procedure name.

ProcLangName is the name of the default language used within the procedure for parameters and local variables. This language may be either the language of the DBEnvironment or n-computer. The default is the language of the DBEnvironment.

ParameterDeclaration specifies the attributes of parameter data to be passed to or from the procedure. The syntax of *ParameterDeclaration* is presented separately below.

ResultDeclaration specifies the attributes of a result column in a multiple row result set or sets returned from a procedure to an application or ISQL. The syntax of *ResultDeclaration* is presented separately below.

ProcedureStatement Specifies a statement in the procedure body. The statement may be any one of the following:

- Local variable declaration (see DECLARE*Variable*).
- Parameter or local variable assignment (see Assignment).
- Compound statement. A compound statement has the following syntax:

  ```
  BEGIN [Statement;] [...] END;
  ```

- Control flow and status statements

- IF...THEN...ELSEIF...ELSE...ENDIF
- WHILE...DO...ENDWHILE
- Jump statement (GOTO, GO TO, or RETURN)
- PRINT

- Any SQL statement allowed in an application *except* the following:

```
ADVANCE
BEGIN DECLARE SECTION
BULK statements
CLOSE (when the USING clause is specified)
COMMIT WORK RELEASE
CONNECT
CREATE PROCEDURE (including inside CREATE SCHEMA)
DECLARE CURSOR (when declaring a cursor for an EXECUTE
PROCEDURE statement)
DESCRIBE
DISCONNECT
END DECLARE SECTION
EXECUTE
EXECUTE IMMEDIATE
EXECUTE PROCEDURE
GENPLAN
INCLUDE
OPEN CURSOR USING DESCRIPTOR
OPEN CURSOR USING HostVariableList
PREPARE
RELEASE
ROLLBACK WORK RELEASE
SET CONNECTION
SET DML ATOMICITY
SET MULTITRANSACTION
SET SESSION
SET TRANSACTION
SQLEXPLAIN
START DBE
STOP DBE
```

A *ProcedureStatement* must be terminated by a semicolon.

DBEFileSetName identifies the DBEFileSet in which ALLBASE/SQL is to store sections associated with the procedure. If not specified, the SECTIONSPACE DBEFileSet is used.

SQL Syntax—ParameterDeclaration

ParameterName ParameterType [**LANG** = *ParameterLanguage*]
[**DEFAULT** *DefaultValue*][**NOT NULL**][**OUTPUT** [**ONLY**]]

Parameters—ParameterDeclaration

ParameterName is the name assigned to a parameter in the procedure. No two parameters in the procedure can be given the same name. You can define no more than 1023 parameters in a procedure.

ParameterType indicates what type of data the parameter will contain. The *ParameterType* cannot be a LONG data type. For a list of data types, refer to the "Data Types" chapter.

ParameterLanguage specifies the language for the parameter. A LANG may only be specified for a parameter with a character data type. This language may be either the language of the procedure or n-computer. The default is the language of the procedure.

DefaultValue specifies the default value for the parameter. The default can be a constant, NULL, or a date/time current function. The data type of the default value must be compatible with the data type of the column.

NOT NULL means that the parameter cannot contain null values. If NOT NULL is specified, any statement that attempts to place a null value in the parameter is rejected.

OUTPUT specifies that the parameter can be used for procedure output as well as input (the default). If OUTPUT is *not* specified, the parameter can only be used for input to the procedure.

If procedure output is required, OUTPUT must also be specified for any corresponding parameter in the EXECUTE PROCEDURE statement.

ONLY specifies that the parameter can be used for procedure output only. ONLY should be used, when applicable, to avoid unnecessary initialization of procedure parameters.

You must also specify OUTPUT for any corresponding parameter in the EXECUTE PROCEDURE statement.

The DEFAULT option cannot be specified for OUTPUT ONLY parameters.

SQL Syntax—ResultDeclaration

ResultType [**LANG** = *ResultLanguage*][**NOT NULL**]

Parameters—ResultDeclaration

ResultType indicates the data type of a result column in a query result for a query or queries in the procedure. The "Data Types" chapter describes the data types available in ALLBASE/SQL.

ResultLanguage specifies the language of the result column. A LANG may only be specified for a result column with a character data type. This language may be either the language of the procedure or n-computer. The default is the language of the procedure.

NOT NULL indicates that the result column cannot contain null values.

Description

- A procedure may be created through ISQL or through an application program.
- A **procedure result set** is the set of rows returned by a procedure SELECT, FETCH, or REFETCH statement.
- A **select cursor** (one declared for a SELECT statement) opened in an application program (i.e, outside the procedure) cannot be accessed within the procedure. However, a procedure can open and access its own select cursors.
- A **procedure cursor** (one declared for an EXECUTE PROCEDURE statement) must be opened and accessed outside of the specified procedure, in an application program. An application can open more than one procedure cursor.
- A **procedure with multiple row result sets** is a procedure containing one or more SELECT statements with no INTO clause. In order to retrieve one or more multiple row result sets from a procedure, you must execute the procedure using a procedure cursor. The application can then either process data from a result (by issuing the FETCH statement within the application) or advance past the result set (by issuing the ADVANCE or the CLOSE statement within the application).

  If you execute a procedure without using a procedure cursor in the above case, a warning is returned to the application, no result set data is returned, and any return status and output parameters are returned as usual.
- Transaction statements (COMMIT WORK, ROLLBACK WORK, WHENEVER .. STOP) executed have the usual effect on non-KEEP cursors, i.e. such cursors are closed.

  A procedure executing transaction statements can close a cursor defined on itself. Therefore, transaction statements must be used with care in procedures containing statements returning multiple row result sets.
- Procedures may reference the following set of built-in variables in non-SQL statements only:
  - ::sqlcode
  - ::sqlerrd2
  - ::sqlwarn0
  - ::sqlwarn1
  - ::sqlwarn2
  - ::sqlwarn6
  - ::activexact

  The first six of these have the same meaning that they have as fields in the SQLCA in application programs. Note that in procedures, sqlerrd2 returns the number of rows processed for all host languages. However, in application programs, sqlerrd(3) is used in COBOL and Fortran, sqlerrd[3] is used in Pascal, and sqlerr[2] is used in C. ::activexact indicates whether a transaction is in progress or not. For additional information, refer to the application programming guides and to the chapter "Constraints, Procedures, and Rules."

- Built-in variables cannot be referenced in any SQL statement. They may be referenced in ASSIGNMENT, IF, WHILE, RETURN, and PRINT statements. Refer to the section "Using Procedures" in the chapter "Constraints, Procedures, and Rules" for more explanation of built-in variables.
- Control flow and status statements, local variable declarations, parameter or local variable assignments, and labeled statements are allowed only within procedures.
- Each *ProcedureStatement* must be terminated with a semicolon.
- A label may appear only at the start of a *ProcedureStatement* that is not a compound statement, a local variable declaration, or a WHENEVER directive.
- Host variables cannot be accessed within a procedure.
- No more than 1024 result columns can be defined in a procedure result set.
- Within a procedure, any SELECT, FETCH, or REFETCH statement with an INTO clause specifying parameters and/or local variables returns at most a one row result.
- A **procedure with single format multiple row result sets** is a procedure having one or more multiple row result sets, whose result format is defined in the WITH RESULT clause. Each SELECT statement with no INTO clause must return rows of a format compatible with this defined result format. When using the WITH RESULT clause, all such result sets in the procedure must return the same number of columns. The corresponding result columns of each result set must be compatible in data type, language and nullability. The corresponding result columns of each result set must be no longer than defined in the WITH RESULT clause. (For more information about data type compatibility, refer to chapter 7, "Data Types.")
- The WITH RESULT clause is used to describe the data format of a procedure's multiple row result sets. Since, by definition, all single format multiple row result sets have the same format, there is no distinction made between result sets. There is no need to issue any ADVANCE statement in the application. Use the WITH RESULT clause only when you do not need to know the boundary between result sets.

  ALLBASE/SQL attempts to verify compatibility of each result set format with the format defined in the WITH RESULT clause at the time the procedure is created. In addition, since verification is not always possible at procedure creation time (sections may be created as invalid), compatibility is also verified at procedure execution time for each procedure result set. If incompatibility is detected during procedure creation, the create statement returns a warning. If incompatibility is detected during procedure execution, the execution of the procedure result set statement fails with an error, and no more data is returned (For an ADVANCE or CLOSE, procedure execution continues with the next statement).
- An attempt to execute a CREATE PROCEDURE statement containing a WITH RESULT clause but no multiple row result set causes an error and the procedure is not created.
- When a procedure with single format multiple row result sets is created using the WITH RESULT clause, the format specified in this clause is stored in the system catalog PROCRESULT table. This format information can be returned after defining a cursor on a procedure (at procedure execution time) with a DESCRIBE RESULT statement *before* (opening and fetching) from the cursor.

- Indicator variables are not allowed or needed inside procedures. However, you can include an indicator variable with a host variable in supplying a value to a parameter in EXECUTE PROCEDURE, DECLARE CURSOR, OPEN, or CLOSE statements.

  Indicator variables specified for output host variables in CLOSE, DECLARE CURSOR, or EXECUTE PROCEDURE statements are set by ALLBASE/SQL.
- Syntactic errors are returned along with an indication of the location of the error inside the CREATE PROCEDURE statement.
- Statements that support dynamic processing are not allowed within a procedure.
- Within a procedure, a single row SELECT statement (one having an INTO clause) that returns multiple rows will assign the first row to output host variables or procedure parameters, and a warning is issued. In an application, this case would generate an error.
- If the IN *DBEFileSetName* clause is specified, but the procedure owner does not have SECTIONSPACE authority for the specified DBEFileSet, a warning is issued and the default SECTIONSPACE DBEFileSet is used instead.

Authorization

You must have RESOURCE or DBA authority to create a procedure. If you do not have all appropriate authorities on the objects referenced by the procedure when you create the procedure, warnings are returned. If you do not have the appropriate authorities at execution time, errors are returned but (except in a rule) the execution of the rest of the procedure does not stop. The procedure owner becomes the owner of any object created by the procedure with no owner explicitly specified. A user granted authority to execute a procedure need not have any direct authority on the objects accessed by the procedure.

To specify a *DBEFileSetName*, the procedure owner must have SECTIONSPACE authority on the referenced DBEFileSet.

Examples

1. DELETE

```
CREATE PROCEDURE ManufDB.RemoveBatchStamp (BatchStamp DATETIME NOT NULL)
AS
BEGIN
   DELETE FROM ManufDB.TestData WHERE BatchStamp = :BatchStamp;
   IF ::sqlcode < > 0 THEN
      PRINT 'Delete failed.';
   ENDIF;
END;
```

2. INSERT

```
CREATE PROCEDURE PurchDB.ReportMonitor (Name CHAR(20) NOT NULL,
       Owner CHAR(20) NOT NULL, Type CHAR(10) NOT NULL)
AS
BEGIN
   INSERT INTO PurchDB.ReportMonitor
     VALUES (:Type, CURRENT_DATETIME,
     USER, :Name, :Owner);
   RETURN ::sqlcode;
IN PurchFS;
END
```

3. SELECT (multiple row and single row)

```
CREATE PROCEDURE ReportOrder (OrderNumber INTEGER,
                              TotalPrice  DECIMAL (10,2) OUTPUT) AS
BEGIN
```

Multiple row result set is returned to the application for processing using a procedure cursor.

```
SELECT ItemNumber, OrderQty, PurchasePrice
  FROM PurchDB.OrderItems
  WHERE OrderNumber = :OrderNumber;
```

Single row result set value is returned to the application via an OUTPUT parameter.

```
SELECT SUM (OrderQty * PurchasePrice)
  INTO :TotalPrice
  FROM PurchDB.OrderItems
  WHERE OrderNumber = :OrderNumber;
END;
```

CREATE RULE

The CREATE RULE statement defines a rule and associates it with specific kinds of data manipulation on a particular table. The rule definition specifies the name of a procedure to be executed when the rule fires.

Scope

ISQL or Application Programs

SQL Syntax

```
CREATE RULE [Owner.]RuleName
AFTER StatementType [,...][ON
                           OF
                           FROM
                           INTO ][Owner.]TableName
[REFERENCING {OLD AS OldCorrellationName
              NEW AS NewCorrelationName}[...]] [WHERE FiringCondition
EXECUTE PROCEDURE [OwnerName.]ProcedureName [(ParameterValue [,...])]
[IN DBEFileSetName]
```

Parameters

[*Owner.*]*RuleName* is the name of the new rule. Two rules cannot have the same owner and rule names.

The rule owner must be the same as the owner of the table the rule is defined upon. The default owner name is the owner name of the table it is being defined on. The usual default owner rules do not apply here.

StatementType specifies which statements will cause the rule to fire for the given table. *StatementType* must be one of the following:

- INSERT

UPDATE [(*ColumnName* [,...])]

- DELETE

Each statement type can be listed in the CREATE RULE statement only once for a given rule. If *ColumnNames* are specified for a *StatementType* of UPDATE, they must exist in the table.

For UPDATE statements in which more than one column is specified, any one of the column names listed here may be used in the UPDATE for the rule to affect the statement. When you issue the UPDATE, it is not necessary to specify all the *ColumnNames* in the CREATE RULE statement. At most, 1023 column names may be specified in this column name list.

[*Owner.*]*TableName* designates the table on which the rule is to operate. Rules cannot be created on views.

OldCorrelationName specifies the correlation name to be used within the

FiringCondition and *ParameterValue* to refer to the old values of the row (before it was changed by the DELETE or UPDATE statement). The default *OldCorrelationName* is OLD. If the *StatementType* is INSERT, an *OldCorrelationName* will refer to the new values of the row, since no old values are available.

NewCorrelationName specifies the correlation name to be used within the *FiringCondition* and *ParameterValue* to refer to the new values of the row (after it was changed by the INSERT or UPDATE statement). The default *NewCorrelationName* is NEW. If the *StatementType* is DELETE, a *NewCorrelationName* will refer to old values of the row, since no new values are available.

FiringCondition specifies a search condition the current row must meet once the rule's statement type has matched before the rule can fire on that row. Refer to the "Search Conditions" chapter for possible predicates.

The search condition must evaluate to TRUE to invoke the specified procedure. The search condition cannot contain any subqueries, aggregate functions, host variables, local variables, procedure parameters, dynamic parameters, or the TID function.

[*Owner.*]*Procedure Name* specifies the procedure to invoke when a rule fires. The procedure must exist when the rule is created.

ParameterValue specifies a value for a parameter in the procedure. The parameter values must correspond in sequential order to the parameters defined for the procedure.

ParameterValue has the following syntax:

```
{ NULL
  Expression }
```

The *Expression* may include anything allowed within an SQL expression except a subquery, aggregate function, host variable, TID function, local variable, procedure parameter, dynamic parameter, or a long column value. Refer to the "Expressions" chapter for the complete syntax of expressions. In particular, column references *are* allowed within the EXECUTE PROCEDURE clause of the CREATE RULE statement. Column references may be of the form:

```
{ OldCorrelationName.ColumnName
  NewCorrelationName.ColumnName
  [ [Owner.] TableName. ] ColumnName }
```

DBEFileSetName specifies the DBEFileSet in which sections associated with the rule are to be stored. If not specified, the default SECTIONSPACE DBEFileSet is used. (Refer to syntax for the SET DEFAULT DBEFILESET statement.)

Description

- A rule may be created through ISQL or through an application program.
- When a rule is created, information about the rule is stored in the system catalog, and may be examined through the following system views: SYSTEM.RULE, SYSTEM.RULECOLUMN, and SYSTEM.RULEDEF.
- The *FiringCondition* and *ParameterValue* can reference both the unchanged and the changed values of the row being considered for the firing of a rule. The unchanged values are known as *old* values and are referred to by using the *OldCorrelationName.* Changed values are known as *new* values and are referred to by using the *NewCorrelationName*.
- For an INSERT, there is no old value to reference, so the use of *OldCorrelationName* will be treated as if *NewCorrelationName* had been specified.
- For a DELETE, there is no new value to reference, so the use of *NewCorrelationName* will be treated as if *OldCorrelationName* had been specified.
- If no *OldCorrelationName* is defined, OLD is the default.
- If no *NewCorrelationName* is defined, NEW is the default.
- At most one *OldCorrelationName* and one *NewCorrelationName* can be specified.
- Use of the *TableName* has the same effect as use of the *NewCorrelationName* if the *StatementType* is INSERT or UPDATE. Use of the *TableName* has the same effect as use of the *OldCorrelationName* if the *StatementType* is DELETE.
- *NewCorrelationName* and *OldCorrelationName* must differ from each other. If either is the same as the *TableName*, then the correlation name will be assumed to be used wherever that name qualifies a column reference without an owner qualification also being used. If the table is called OLD, reference it by using *OwnerName*.OLD.*ColumnName*.
- Rules can execute in a forward-chaining manner. This occurs when a fired rule invokes a procedure which contains a statement that causes other rules to fire. The maximum nesting of rule invocations is 20 levels.
- If multiple rules are to be fired by a given statement, the order in which the rules fire may change when the section is revalidated. You can use the SET PRINTRULES ON statement to generate messages giving the names of rules as they fire.
- If an error occurs during the execution of a rule or its invoked procedure, it will have its normal effect, that is, a message may be generated, the execution of the statement may be halted, the effects of the statement may be rolled back, or the connection may be lost. Even if the error has not caused the transaction to roll back or the connection to be lost, the statement issued by the user and all rules fired on behalf of that statement (or chained to by such rules) are undone and have no effect on the database.
- The procedure invoked by a rule cannot execute a COMMIT WORK, ROLLBACK WORK, COMMIT/ROLLBACK ARCHIVE, or SAVEPOINT statement. If the procedure executes one of these statements, an error occurs, and the effect of the statement that triggered the procedure is undone.

- If a *CurrentFunction* is used within the *FiringCondition* or a *ParameterValue*, it will be evaluated at the time of the statement that fires the rule.
- Any value returned by the procedure with a RETURN statement is ignored by the rule and not returned to the statement firing the rule.
- An EXECUTE PROCEDURE call from within a rule is different from one issued as a regular SQL statement. Within a rule, you cannot specify host variables, local variables, procedure parameters, or dynamic parameters as parameter values, since host variables are not accessible from the rule. Also, the key word OUTPUT cannot be specified, since a procedure called from a rule cannot return any values. A rule *does* permit the specification of columns within the procedure call, since in this context column values are available to be passed to the procedure from the row the rule is firing on.
- The CREATE RULE statement invalidates sections that contain dependencies upon the table the rule is defined upon. This is to enable the rule to be included when those sections are revalidated.
- If a procedure specified in a CREATE RULE statement returns multiple row result set(s), a warning is issued when the rule is created. Note that no warning is issued when the procedure is invoked by the rule.
- If the IN *DBEFileSetName* clause is specified, but the rule owner does not have SECTIONSPACE authority for the specified DBEFileSet, a warning is issued and the default SECTIONSPACE DBEFileSet is used instead. (Refer to syntax for the GRANT statement and the SET DBEFILESET statement.)

Authorization

The CREATE RULE statement requires you to have OWNER authority for the table and OWNER or EXECUTE authority for the procedure, or to have DBA authority. Once the rule is defined, users issuing statements which cause the rule to fire need not have EXECUTE authority for the procedure.

To specify a DBEFileSetName for a rule, the rule owner must have SECTIONSPACE authority on the referenced DBEFileSet.

Example

First, create a procedure to monitor operations on the Reports table:

```
CREATE PROCEDURE PurchDB.ReportMonitor (Name CHAR(20) NOT NULL,
       Owner CHAR(20) NOT NULL, Type CHAR(10) NOT NULL) AS
            BEGIN
                INSERT INTO PurchDB.ReportMonitor
                VALUES (:Type, CURRENT_DATETIME,
                          USER, :Name, :Owner);
            RETURN ::sqlcode;
            END
       IN PurchDBFileSet;
```

Next, create three rules that invoke the procedure with parameters:

```
CREATE RULE PurchDB.InsertReport
      AFTER INSERT TO PurchDB.Reports
      EXECUTE PROCEDURE PurchDB.ReportMonitor (NEW.ReportName,
          NEW.ReportOwner, 'INSERT')
IN PurchDBFileSet;

CREATE RULE PurchDB.DeleteReport
      AFTER DELETE FROMPurchDB.Reports
      EXECUTE PROCEDURE PurchDB.ReportMonitor (OLD.ReportName,
          OLD.ReportOwner, 'DELETE')
IN PurchDBFileSet;

CREATE RULE PurchDB.UpdateReport
      AFTER UPDATE TO PurchDB.Reports
      EXECUTE PROCEDURE PurchDB.ReportMonitor (NEW.ReportName,
          NEW.ReportOwner, 'UPDATE')
IN PurchDBFileSet;
```

CREATE SCHEMA

The CREATE SCHEMA statement creates a schema and associates an authorization name with it. The schema defines a database containing tables, views, indexes, procedures, rules, and authorization groups with the same owner name. Entries are created in the system catalog views upon completion of the execution of this statement.

Scope

ISQL or Application Programs

SQL Syntax

```
CREATE SCHEMA AUTHORIZATION AuthorizationName [TableDefinition
                                               ViewDefinition
                                               IndexDefinition
                                               ProcedureDefinition
                                               RuleDefinition
                                               CreateGroup
                                               AddToGroup
                                               GrantStatement       ][...]
```

Parameters

AuthorizationName specifies the owner of the database objects.

If you have RESOURCE authority, the AuthorizationName must be your DBEUserID, a class name, or an authorization group name to which you belong. You cannot specify a different owner for the objects you create.

If you have DBA authority, the AuthorizationName can be any DBEUserID, class name, or authorization group name. The owner of the objects you create does not have to match the AuthorizationName if the owner has DBA authority.

You must specify an AuthorizationName; there is no default.

TableDefinition defines a table and automatic locking strategy. For complete syntax, refer to the CREATE TABLE syntax.

ViewDefinition defines a view of a table, another view, or a combination of tables and views. For complete syntax, refer to the CREATE VIEW syntax.

IndexDefinition creates an index on one or more columns. For complete syntax, refer to the CREATE INDEX syntax.

ProcedureDefinition creates a procedure which defines a sequence of SQL statements. For correct syntax, refer to the CREATE PROCEDURE syntax.

RuleDefinition creates a rule to fire a stored procedure. For complete syntax, refer to the CREATE RULE syntax.

CreateGroup defines an authorization group. For complete syntax, refer to the CREATE GROUP syntax.

AddToGroup adds one or more users, authorization groups, or combination of users and authorization groups to an authorization group. For complete syntax, refer to the ADD TO GROUP syntax.

GrantStatement specifies the type of authorities for a table, view, or module. For complete syntax, refer to the GRANT syntax.

Description

- Note that a comma or semicolon is not allowed between the object definitions in the CREATE SCHEMA syntax.
- You cannot use the following CREATE statements within the CREATE SCHEMA statement:
  - CREATE DBEFILE
  - CREATE DBEFILESET
- You cannot use this statement to add to a schema that already exists. A schema for a given authorization name exists if there are any objects (tables, views, indexes, procedures, rules, or groups) owned by that authorization name.
- When the CREATE SCHEMA statement is part of a procedure, no *ProcedureDefinition* may be included.

Authorization

You can execute this statement if you have RESOURCE authority or DBA authority. With RESOURCE authority you can create a schema by using your own name or the authorization group name to which you belong. If you have DBA authority, then you can create a schema with any AuthorizationName.

Example

In the following example, RecDB is the AuthorizationName (owner name). All the tables created here are owned by RecDB; it is not necessary to repeat the owner name for each creation statement.

```
CREATE SCHEMA AUTHORIZATION RecDB
CREATE PUBLIC TABLE Clubs
                    (ClubName CHAR(15) NOT NULL
                    PRIMARY KEY CONSTRAINT Clubs_PK,
                    ClubPhone SMALLINT,
                    Activity CHAR(18))
                  IN RecFS

CREATE PUBLIC TABLE Members
                    (MemberName CHAR(20) NOT NULL,
                    Club CHAR(15) NOT NULL,
                    MemberPhone SMALLINT,
                    PRIMARY KEY (MemberName, Club) CONSTRAINT Members_PK,
                    FOREIGN KEY (Club) REFERENCES Clubs (ClubName)
                    CONSTRAINT Members_FK)
                  IN RecFS
```

```
CREATE PUBLIC TABLE Events
                    (SponsorClub CHAR(15),
                    Event CHAR(30),
                    Date DATE DEFAULT CURRENT_DATE,
                    Time TIME,
                    Coordinator CHAR(20),
                    FOREIGN KEY (Coordinator, SponsorClub)
                    REFERENCES Members (MemberName, Club) CONSTRAINT Events_FK)
                  IN RecFS
```

CREATE TABLE

The CREATE TABLE statement defines a table. It also defines the locking strategy that ALLBASE/SQL uses automatically when the table is accessed and in some cases automatically issues a GRANT statement. It can also define the storage structure of the table and restrictions or defaults placed on values which the table's columns can hold. You can also use this statement to assign a table to a partition for audit logging purposes.

Scope

ISQL or Application Programs

SQL Syntax—CREATE TABLE

```
CREATE [PRIVATE
        PUBLICREAD
        PUBLIC
        PUBLICROW  ]TABLE [Owner.]TableName
[LANG = TableLanguageName]
({ColumnDefinition
  UniqueConstraint
  ReferentialConstraint
  CheckConstraint       }[,...])
[UNIQUE HASH ON (HashColumnName [,...]) PAGES = PrimaryPages
 HASH ON CONSTRAINT [ConstraintID] PAGES = PrimaryPages
 CLUSTERING ON CONSTRAINT [ConstraintID                    ]
[IN PARTITION {PartitionName
               DEFAULT
               NONE          }]
[IN DBEFileSetName1]
```

Parameters—CREATE TABLE

PRIVATE — enables the table to be used by only one transaction at a time. This is the most efficient option for tables that do not need to be shared because ALLBASE/SQL can spend less time managing locks.

This option is in effect by default; grants are not automatically issued.

PUBLICREAD — enables the table to be read by concurrent transactions, but allows no more than one transaction at a time to update the table.

This option automatically issues a statement GRANT SELECT ON *TableName* TO PUBLIC. This gives any user with CONNECT authority the ability to read the table. To change this grant, use the REVOKE statement and the GRANT statement. The locking strategy remains unchanged, even if you change the grant.

PUBLIC — enables the table to be read and updated by concurrent transactions. In general, a transaction locks a page in share mode before reading it and in exclusive mode before updating it.

This option automatically issues the statement GRANT ALL ON *TableName*

TO PUBLIC. This gives any user with CONNECT authority the ability to read and modify the table as well as to alter the table and create indexes on it. To change this grant, use the REVOKE statement and the GRANT statement. The locking strategy remains unchanged, even if you change the grant.

PUBLICROW enables the table to be read and updated by concurrent transactions. The locking unit is a row (tuple) in PUBLICROW tables. In general, a transaction locks a row in share mode before reading it and in exclusive mode before updating it. For small tables with small rows, concurrency can be maximized by using the PUBLICROW type.

This option automatically issues the statement GRANT ALL ON *TableName* TO PUBLIC. This gives any user with CONNECT authority the ability to read and modify the table as well as to alter the table and create indexes on it. To change this grant, use the REVOKE statement and the GRANT statement. The locking strategy remains unchanged, even if you change the grant.

[*Owner*.]*TableName* is the name to be assigned to the new table. Two tables cannot have the same owner name and table name.

You can specify the owner of the new table if you have DBA authority. If you do not have DBA authority, you can specify the owner as the name of any group to which you belong. If you do not specify the owner name, your DBEUserID, schema authorization name, procedure owner, or the ISQL SET OWNER name becomes the owner of the new table. For more information, refer to the section "Default Owner Rules" in the chapter "Using ALLBASE/SQL."

TableLanguageName specifies the language for the new table. This name must be either n-computer or the language of the DBEnvironment. The default is the language of the DBEnvironment.

ColumnDefinition defines an individual column in a table. Each table must have at least one column. The syntax for a CREATE TABLE column definition is presented separately in another section below.

UniqueConstraint defines a uniqueness constraint for the table. Each table can have multiple unique constraints, but can have only one specifying PRIMARY KEY. The syntax for a *UniqueConstraint* is presented separately in another section below.

ReferentialConstraint defines a referential constraint of this table with respect to another (or the same) table. The referencing table (this one) and the referenced table (the other one) satisfy the constraint if, and only if each row in the referencing table contains either a NULL in a referencing column, or values in the rows of the referencing columns equal the values in the rows of the referenced columns. The syntax of a *ReferentialConstraint* is presented separately in another section below.

CheckConstraint defines a check constraint for the table. A table can have multiple check constraints. The syntax for a check constraint is presented

separately in another section below.

`UNIQUE HASH ON` specifies a hash structure for the table. Only UNIQUE HASH structures may be created, and updates on hash key columns are not permitted (you must first delete, then insert the row with the new key value).

`HashColumnName` specifies a column defined in the table that is to participate in the hash key of this table.

`PrimaryPages` specifies the number of pages used as primary hash buckets. The minimum is 1 and the maximum is determined by the formula $16*((|2^{31})–2072)$. For good results, use a prime number.

`HASH ON CONSTRAINT` specifies that the named unique constraint be managed through the use of hash table storage. The unique constraint's columns become the hash key columns.

`ConstraintID` is an optional name specified for the constraint. If none is supplied, one is generated, as described under "Description" below.

`IN PARTITION` specifies what partition the table will be in for the purposes of audit logging.

`PartitionName` specifies the partition for the table.

`DEFAULT` specifies that the default partition of the database will be used. The number associated with the default partition is determined at the time the `INSERT, UPDATE`, or `DELETE` is executed on the table. If the default partition is NONE at that time, audit logging of the operation is not done. Any change to the default partition number occurring in a `START DBE NEWLOG` statement alters the partition number that audit logging uses on tables that are in the default partition.

`NONE` specifies that this table is assigned to no partition, and so will have no audit logging done on it.

`CLUSTERING ON CONSTRAINT` specifies that the named unique or referential constraint be managed through a clustered index structure rather than nonclustered. The unique constraint's unique column list, or referential constraint's referencing column list, becomes the clustered key.

`IN` *`DBEFileSetName1`* causes the index and data pages in which table information is stored to be allocated from DBEFiles associated with the specified DBEFileSet. (Names of available DBEFileSets are recorded in the SYSTEM.DBEFILESET view.) If a DBEFileSet name is not specified, the table is created in the default TABLESPACE DBEFileSet. (Refer to syntax for the `SET DEFAULT DBEFILESET` statement.)

You can create a nonhash table in an empty DBEFileSet, but cannot INSERT any rows or create any indexes for the table until the DBEFileSet has DBEFiles associated with it.

You cannot create a hash structure in an empty DBEFileSet.

SQL Syntax—Column Definition

```
ColumnName{ColumnDataType
          LongColumnType [IN DBEFileSetName2]}
[LANG = ColumnLanguageName]
[[NOT]CASE SENSITIVE]
[DEFAULT{Constant
         USER
         NULL
         CurrentFunction}]
[NOT NULL [{UNIQUE
            PRIMARY KEY} [CONSTRAINT ConstraintID]]
 REFERENCES RefTableName [(RefColumnName)][CONSTRAINT ConstraintID]]
 [...]

 CHECK (SearchCondition) [CONSTRAINT ConstraintID]
 [IN DBEFileSetName3]                                              ][...]
```

Parameters—Column Definition

ColumnName is the name to be assigned to one of the columns in the new table. No two columns in the table can be given the same name. You can define a maximum of 1023 columns in a table.

ColumnDataType indicates what type of data the column can contain. Some data types require that you include a length. See the "Data Types" chapter for the data types that can be specified.

LongColumnType specifies a LONG data type for the new column. At most 40 columns with a *LongColumnType* may be defined in a single table.

DBEFileSetName2 specifies the DBEFileSet where long column data is to be stored. This DBEFileSet may be different from that of the table. If a DBEFileSet is not specified, the LONG data is stored in the DBEFileSet containing the table.

ColumnLanguageName specifies the language for the column. This can only be specified for CHAR or VARCHAR columns. This name must be either n-computer or the language of the DBEnvironment. The default is the language of the DBEnvironment.

CASE SENSITIVE indicates that upper and lower case letters stored in the column are not considered equivalent. If the column is defined as NOT CASE SENSITIVE, then its upper and lower case letters are considered equivalent. The default is CASE SENSITIVE. This clause is allowed only with CHAR and VARCHAR columns.

DEFAULT specifies the default value to be inserted for this column. The default can be a constant, NULL, or a date/time current function The data type of the default value must be compatible with the data type of the column. DEFAULT cannot be specified for LONG data type columns.

NOT NULL means the column cannot contain null values. If NOT NULL is specified, any statement that attempts to place a null value in the column is rejected. However, if atomicity is set to row level, only the NULL row receives the error and the statement halts.

UNIQUE | PRIMARY KEY specifies a unique constraint placed on the column. The table level constraint { UNIQUE | PRIMARY KEY } (*ColumnName*) is equivalent. See the discussion on table level unique constraints below.

REFERENCES specifies a Referential Constraint placed on the column. This is equivalent to the table level constraint FOREIGN KEY (*ColumnName*) REFERENCES *RefTableName* [(*RefColumnName*)]. See the discussion on table level referential constraint below.

CHECK specifies a check constraint placed on the column.

SearchCondition specifies a boolean expression that must not be false. The result of the boolean expression may be unknown if a value in the expression is NULL. See the discussion on a table level check constraint below. In addition, for a column definition check constraint, the only column the search condition can reference is *ColumnName*.

ConstraintID is an optional name specified for the constraint. If none is supplied, one is generated, as described under "Description" below.

DBEFileSetName3 specifies the DBEFileSet to be used for storing the section associated with the check constraint. If not specified, the default SECTIONSPACE DBEFileSet is used. (Refer to syntax for the SET DEFAULT DBEFILESET statement.)

SQL Syntax—Unique Constraint (Table Level)

```
{UNIQUE
 PRIMARY KEY}( ColumnName [,...]) [CONSTRAINT ConstraintID]
```

Parameters—Unique Constraint (Table Level)

UNIQUE Each *ColumnName* shall identify a column of the table, and the same column shall not be identified more than once. Also, NOT NULL shall be specified for each column in this unique constraint column list.

PRIMARY KEY In addition to the rules for the UNIQUE option, PRIMARY KEY may only be specified once in a table definition. It provides a shorthand way of referencing its particular unique constraint column list in a referential constraint.

ColumnName [,...] is the unique constraint column list, or key list, of the constraint. No two unique constraints may have identical column lists. The maximum number of columns in a unique column list is 15. None of the columns may be a LONG data type.

ConstraintID is an optional name specified for the constraint. If none is supplied, one is generated, as described under "Description" below.

SQL Syntax—Referential Constraint (Table Level)

```
FOREIGN KEY (FKColumnName [,...])
REFERENCES RefTableName [( RefColumnName [,...])] [CONSTRAINT ConstraintID]
```

Parameters—Referential Constraint (Table Level)

FKColumnName [,...] identifies the referencing column list. Each referencing column shall be a column defined in the referencing table, and the same column name shall not be identified more than once. The number of referencing and referenced columns would be the same. The maximum number of columns in a referencing column list is 15. None of the columns may be a LONG data type.

RefTableName identifies the base table being referenced. If no *RefColumnName* list follows this, the base table must contain a PRIMARY KEY unique constraint with the correct number of columns, each of the correct data type.

RefColumnName [,...] identifies the referenced column list. This column list must be identical to a unique constraint column list of the referenced table.

ConstraintID is an optional name specified for the constraint. If none is supplied, one is generated, as described under "Description" below.

SQL Syntax—Check Constraint (Table Level)

CHECK (*SearchCondition*) [**CONSTRAINT** *ConstraintID*] [**IN** *DBEFileSetName3*]

Parameters—Check Constraint (Table Level)

CHECK specifies a check constraint.

SearchCondition specifies a boolean expression for the check constraint. The result of the boolean expression must not be false for any row of the table. The result may be unknown if a column that is part of the expression is NULL. The search condition may only contain LONG columns if they are within long column functions. (Refer to long column functions in the "Expressions" and "Data Types" chapters.) The search condition cannot contain a subquery, host variable, aggregate function, built-in variable, local variable, procedure parameter, dynamic parameter, TID function, current function, or USER. Refer to the chapter, "Constraints, Procedures, and Rules," for more information on check constraints.

ConstraintID is an optional name specified for the constraint. If none is supplied, one is generated, as described under "Description" below.

DBEFileSetName3 specifies the DBEFileSet to be used for storing the section associated with the check constraint. If not specified, the default SECTIONSPACE DBEFileSet is used. (Refer to syntax for the SET DEFAULT DBEFILESET statement.)

Description

- PUBLIC, PUBLICROW, PUBLICREAD, and PRIVATE are locking modes. They define the type of locking ALLBASE/SQL uses automatically when the table is accessed. The LOCK TABLE statement can be used to override automatic locking during any transaction, if the override is to a more restrictive lock. If no locking mode is specified,

PRIVATE is assumed. For complete information on locking, refer to the chapter "Concurrency Control through Locks and Isolation Levels."

- For nonhash tables, `CREATE TABLE` simply enters the new table's definition into the system catalog. Until you insert a row into the new table, the table does not occupy any storage. For hash tables, the number of primary pages is allocated at `CREATE TABLE` time.
- Data and index values of columns defined as NOT CASE SENSITIVE are not converted to upper case when stored. However, during comparison, sorting, and indexing operations, upper and lower case letters are considered equivalent. If a case sensitive column is compared to a column that is not case sensitive, both columns are treated as case sensitive. When defining a referential constraint, the case sensitivity of the referenced and referencing columns must match.

  The NOT CASE SENSITIVE clause has no effect if the character set does not differentiate between upper and lower case, such as Chinese.
- Upper and lower case extended characters are treated as equivalent. They are compared to the collation table of a specific language regardless of case.
- If no DEFAULT clause is given for a column in the table, an implicit DEFAULT NULL is assumed. Any `INSERT` statement, which does not include a column for which a default has been declared, causes the default value to be inserted into that column for all rows inserted.
- For a CHAR column, if the specified default value is shorter in length than the target column, it is padded with blanks. For a CHAR or VARCHAR column, if the specified default value is longer than the target column, it is truncated.
- For a BINARY column, if the specified default value is shorter in length than the target column, it is padded with zeroes. For a BINARY or VARBINARY column, if the specified default value is longer than the target column, it is truncated.
- If a constraint is defined without a *`ConstraintID`*, one is generated of the following form:

  `SQLCON_`*`uniqueid`*

  where the *`uniqueid`* is unique across all constraints. You cannot define a constraint starting with `SQLCON_`. All constraint names must be unique for a given owner, regardless of which table they are in.
- Unique constraints are managed through the use of B-tree indexes unless the constraint is named and its name is referenced in the HASH ON CONSTRAINT clause. If the name is referenced in the CLUSTERING ON CONSTRAINT clause, the B-tree index is clustered.
- Referential constraints are managed through the use of virtual indexes. A virtual index is created by ALLBASE/SQL. Virtual indexes can be clustered with respect to the referencing columns' values if the constraint is named in the CLUSTERING ON CONSTRAINT clause.
- The behavior by which integrity constraints are enforced is determined by the setting of the `SET DML ATOMICITY` and `SET CONSTRAINTS` statements. Refer to the discussion of these statements in this chapter for more information.

- Unique constraint indexes use space in this table's DBEFileSet; but referential constraint virtual indexes use space in the *referenced* table's DBEFileSet.
- If the HASH or CLUSTERING ON CONSTRAINT clause is used without a constraint name, the PRIMARY KEY of the table is used. If a PRIMARY KEY is not defined, an error results.
- At most 15 columns may be used in a unique or referential constraint key. The maximum length of the index key for unique or referential constraints is obtained from the following formula:

  ```
  (NumberOfColumns + 3)* 2 + SumColumnLengths + 10 = 1024
  ```

  An extra 2 bytes must be added for each column that allows NULLS or is a VARCHAR data type.
- The data types of the corresponding columns in a referential constraint's referencing and referenced column lists must be the same with the following exceptions. CHAR and VARCHAR are allowed to refer to each other, as are the pairs BINARY and VARBINARY, and NATIVE CHAR and NATIVE VARCHAR. DECIMAL columns must exactly match in precision and scale. SMALLINT, INTEGER, FLOAT, and REAL references cannot refer to a data type other than their same data type. LONG columns may not be used in integrity constraints.
- You can use the same set of foreign key columns to reference two different primary keys.
- The maximum size of a hash key is the same as a user-defined index key, which is determined in the following formula:

  ```
  (NumberOfHashColumns+2)* 2 + SumKeyColumnLengths + 8 <= 1024
  ```

  An extra 2 bytes must be added for each column that allows NULLS or is a VARCHAR datatype.

  At most 16 columns are allowed in a hash structure key.
- A hash structure may not be dropped, except by dropping the table upon which it is defined with the `DROP TABLE` statement.
- You cannot create a hash structure as a PUBLICROW table.
- If the table is created with a HASH structure, enough empty data and mixed DBEFiles must exist to contain the primary pages for the hash table data at the time the table is created. Primary pages for hash tables cannot be placed in DBEFile0, an index DBEFile, or a nonempty DBEFile. Similarly, data for nonhash tables cannot be placed in a DBEFile containing primary pages for hash tables.
- The partition must be already created by the `CREATE PARTITION` statement, it must be the default partition, or it must be specified as NONE.
- The partition number of a table's partition is recorded in any audit logging generated on that table. Audit logging is done on any `INSERT, UPDATE`, or `DELETE` performed on a table while the DBEnvironment is enabled for DATA audit logging, unless the table is in the partition NONE.
- Audit logging is not done on any LONG column data for the table.

- If no partition is specified, the table is placed in the DEFAULT partition.
- To specify that a table is in no partition, the partition NONE can be specified.
- Partitions can be created and tables placed in them without DATA audit logging being enabled for a DBEnvironment. However, the partition information is only used in audit log records. Thus, partition information is not utilized until the DBEnvironment has DATA audit logging enabled through the START DBE NEWLOG statement.
- If the IN *DBEFileSetName1* clause is specified for the table or the IN *DBEFileSetName2* clause is specified for a long column, but the table owner does not have TABLESPACE authority for the specified DBEFileSet, a warning is issued and the default TABLESPACE DBEFileSet is used instead. (Refer to syntax for the GRANT statement and the SET DEFAULT DBEFILESET statement.)
- If the IN *DBEFileSetName3* clause is specified for a check constraint, but the table owner does not have SECTIONSPACE authority for the specified DBEFileSet, a warning is issued and the default SECTIONSPACE DBEFileSet is used instead. (Refer to syntax for the GRANT statement and the SET DEFAULT DBEFILESET statement.)

Authorization

You must have RESOURCE or DBA authority to use this statement. To define referential constraints, the table owner must have REFERENCES authority on the referenced table and referenced columns, own the referenced table, or have DBA authority for the life of the referential constraint. The REVOKE, DROP GROUP, and REMOVE FROM GROUP statements are not permitted if they remove REFERENCES authority from the table's owner until the referential constraint or table is dropped or ownership is transferred to someone else.

To specify a *DBEFileSetName* for a long column, the table owner must have TABLESPACE authority on the referenced DBEFileSet.

To specify a *DBEFileSetName* for a check constraint, the section owner must have SECTIONSPACE authority on the referenced DBEFileSet.

Examples

1. Creating and accessing tables

   This public table is accessible to any user or program that can start a DBE session. It is also accessible by concurrent transactions.

```
CREATE PUBLIC TABLE PurchDB.SupplyPrice
                    (PartNumber  CHAR(16) NOT NULL,
                    VendorNumber  INTEGER  NOT NULL,
                    VendPartNumber CHAR(16) NOT NULL,
                    UnitPrice  DECIMAL(10,2),
                    DeliveryDays  SMALLINT DEFAULT 0,
                    DiscountQty  SMALLINT)
                 IN PARTITION PartsPart
            IN PurchFS;

REVOKE ALL PRIVILEGES ON PurchDB.SupplyPrice FROM PUBLIC

GRANT SELECT,UPDATE ON Purch.DB.SupplyPrice TO Accounting
```

Now only the DBA and members of authorization group Accounting can access the table. Later, the accounting department manager is given control.

```
TRANSFER OWNERSHIP OF PurchDB.SupplyPrice TO MgrAcct
```

2. Creating a table using constraints and LONG columns

   In this example, the tables are created with the PUBLIC option so as to be accessible to any user or program that can start a DBE session. RecDB.Clubs defines those clubs which can have members and hold events, as shown by the constraint Members_FK. RecDB.Members defines those members who can have events for certain clubs, as shown by constraint Events_FK. The LONG column Results is used to hold a text file containing the results of a completed event. No date can be entered for an event that is prior to the current date. RecDB.Members and RecDB.Events are both created PUBLICROW since they are small tables on which a large amount of concurrent access is expected.

```
CREATE PUBLIC TABLE RecDB.Clubs
                    (ClubName CHAR(15) NOT NULL
                    PRIMARY KEY CONSTRAINT Clubs_PK,
                    ClubPhone SMALLINT,
                    Activity CHAR(18))
                  NOT CASE SENSITIVE
                  IN RecFS

CREATE PUBLICROW TABLE RecDB.Members
                        (MemberName CHAR(20) NOT NULL,
                        Club CHAR(15) NOT NULL,
                        MemberPhone SMALLINT,
                        PRIMARY KEY (MemberName, Club) CONSTRAINT Members_PK,
                        FOREIGN KEY (Club) REFERENCES RecDB.Clubs (ClubName)
                        CONSTRAINT Members_FK)
                      IN RecFS

CREATE PUBLICROW TABLE RecDB.Events
                        (SponsorClub CHAR(15),
                        Event CHAR(30),
                        Date DATE DEFAULT CURRENT_DATE,
                       CHECK (Date >= '1990-01-01') CONSTRAINT Events_Date_Ck,
                        Time TIME,
                        Coordinator CHAR(20),
                        Results LONG VARBINARY(10000) IN LongFS,
                        FOREIGN KEY (Coordinator, SponsorClub)
                        REFERENCES RecDB.Members (MemberName, Club)
                        CONSTRAINT Events_FK)
                      IN RecFS
```

3. Creating a table with a hash structure

```
BEGIN WORK
```

Statements to create a DBEFile and add it to a DBEFileSet should be in the same transaction as the statement to create the hash structure. This makes it impossible for other transactions to use the new DBEFile for hashing before the hash structure is created.

```
CREATE DBEFILE PurchHashF1 WITH PAGES = 120,
               NAME = 'PurchHF1',
               TYPE = TABLE
   ADD DBEFILE PurchHashF1
 TO DBEFILESET PurchFS

CREATE PUBLIC TABLE PurchDB.Vendors
                    (VendorNumber INTEGER  NOT NULL,
                    VendorName CHAR(30) NOT NULL,
                    ContactName CHAR(30),
                    PhoneNumber CHAR(15),
                    VendorStreet CHAR(30) NOT NULL,
                    VendorCity  CHAR(20) NOT NULL,
                    VendorState  CHAR(2) NOT NULL,
                    VendorZipCode  CHAR(10) NOT NULL,
                    VendorRemarks  VARCHAR(60) )
     UNIQUE HASH ON (VendorNumber) PAGES = 101
                  IN PurchFS
COMMIT WORK
```

4. Specify a DBEFileSet for a Check Constraint in the Column Definition

```
CREATE PUBLIC TABLE RecDB.Events
                    (SponsorClub CHAR(15),
                    Event CHAR(30),
                    Date DATE DEFAULT CURRENT_DATE,
                   CHECK (Date >= '1990-01-01') CONSTRAINT Events_Date_Ck
                   IN RecFS,
                    Time TIME,
                    Coordinator CHAR(20),
                    Results LONG VARBINARY(10000) IN LongFS,
                    FOREIGN KEY (Coordinator, SponsorClub)
                    REFERENCES RecDB.Members (MemberName, Club)
                    CONSTRAINT Events_FK)
                  IN RecFS;
```

CREATE TEMPSPACE

The CREATE TEMPSPACE statement defines and creates a temporary storage space known as a TempSpace. A TempSpace is a location where ALLBASE/SQL creates temporary files to store temporary data when performing a sort, if disk space permits.

Scope

ISQL or Application Programs

SQL Syntax

```
CREATE TEMPSPACE TempSpaceName
WITH [MAXFILEPAGES = MaxTempFileSize,]LOCATION ='PhysicalLocation'
```

Parameters

TempSpaceName is the logical name to be assigned to the new TempSpace. More than one TempSpace can be defined but only one per physical location. All TempSpace names must be unique within the DBEnvironment.

MaxTempFileSize specifies the maximum number of 4096-byte pages allocated for each temporary file in the *PhysicalLocation*. The number of pages must be a number between 128 and 524,284. The default is 256. Each file may grow in size up to *MaxTempFileSize*.

PhysicalLocation identifies the directory *path* where the TempSpace will be located. The directory must exist prior to defining a TempSpace. A TempSpace is created relative to the directory where the DBECon file resides unless an absolute path name is specified.

The maximum length for the path name is 35 bytes.

Description

- If no TempSpaces are defined for a DBEnvironment, sorting is done in the /tmp directory.
- It is recommended that each TempSpace reside in a different disk partition. If TempSpaces are located in different disk partitions, then the disk space of those partitions is available for creating temporary files. Creating a single TempSpace per partition is sufficient, because all TempSpaces on a particular partition would share the space in that partition.
- When the size of a temporary file exceeds *MaxTempFileSize* pages, ALLBASE/SQL opens a temporary file in another defined TempSpace. If additional TempSpace is not available, then temporary files are created in the same TempSpace, if space permits.
- The total temporary space required for a DBEnvironment depends on the size of the tables to be sorted or indexes to be created. It also depends on the expected number of concurrent sort operations on the system at one time. The *MaxTempFileSize* (of each

file) should fit within the space available in the partition where the TempSpace is located. Use the HP-UX `bdf` command to determine the space available in a partition.

- The location and characteristics of the TempSpace are stored in the system catalog. TempSpace files are physically created only when needed. When the TempSpace is no longer needed (the present task completes), the temporary file or files are deleted and the space is available for use again.
- The directory specified must be accessible to the DBEnvironment. The directory must be accessible to `hpdb` for both read and write operations. If this read-write (`rw`) capability is not granted or if the directory does not exist when a TempSpace is created, errors are returned.

  If the TempSpace cannot be accessed when a statement requiring temporary space is issued, a system error is returned due to failure in opening the temporary file.
- To delete the definition of a TempSpace, use the `DROP TEMPSPACE` statement.

Authorization

You must have DBA authority to use this statement. hpdb must have write permission in the directory where the TempSpace files will reside.

Example

TempSpace temporary files are created in the /sort/PurchDB directory when SQL Statements require sorting.

```
CREATE TEMPSPACE ThisTempSpace WITH MAXFILEPAGES = 360,
              LOCATION = '/sort/PurchDB'
```

TempSpace temporary files are no longer available in the /sort/PurchDB, directory but can be allocated under /tmp as needed.

```
DROP TEMPSPACE ThisTempSpace
```

CREATE VIEW

The CREATE VIEW statement creates a view of a table, another view, or a combination of tables and views.

Scope

ISQL or Application Programs

SQL Syntax

```
CREATE VIEW [Owner.]ViewName [(ColumnName[,...])]
AS QueryExpression [IN DBEFileSetName]
[WITH CHECK OPTION [CONSTRAINT ConstraintID]]
```

Parameters

[Owner.]ViewName is the name to be assigned to the view. One owner cannot own more than one view with the same name. The view name cannot be the same as the table name.

You can specify the owner of the new view if you have DBA authority. Non-DBA users can specify as owner the name of any group of which they are a member. If you do not specify the owner name, your DBEUserID, schema authorization name, procedure owner, or the ISQL SET OWNER name becomes the owner of the new table. For more information, refer to the section "Default Owner Rules" in the chapter "Using ALLBASE/SQL."

ColumnName specifies the names to be assigned to the columns of the new view. The names are specified in an order corresponding to the columns of the query result produced by the query expression. You can specify a maximum of 1023 columns for a view.

You must specify the column names if any column of the query result is defined by a computed expression, aggregate function, reserved word, or constant in the select list of the query expression. You must also specify column names if the same column name (possibly from different table) appears in the select list more than once.

If you do not specify column names, the columns of the view are assigned the same names as the columns from which they are derived. The * is expanded into the appropriate list of column names.

QueryExpression is the query expression from which the view is derived. The select list can contain as many as 1023 columns. The query expression may refer to tables or views or a combination of tables and views. The query expression may include UNION and/or UNION ALL operations.

DBEFileSetName specifies the DBEFileSet to be used for storing the section associated with the view. If not specified, the default SECTIONSPACE DBEFileSet is used. (Refer to syntax for the SET DEFAULT DBEFILESET statement.)

ConstraintID is the optional name of the view check constraint.

Description

- A view definition with * in the select list generates a view that refers to all the columns that exist in the base table(s) *at the time the view is created*. Adding new columns to the base tables does not cause these columns to be added to the view.
- A view is said to be updatable when you can use it in DELETE, UPDATE, or INSERT statements to modify the base table. A view is updatable only if the query from which it is derived matches the following updatability criteria:
  - No DISTINCT, GROUP BY, or HAVING clause is specified in the outermost SELECT clause, and no aggregate appears in its select list.
  - The FROM clause specifies exactly one table, which must be an updatable table. See "Updatability of Queries" in the "SQL Queries" chapter.
  - To use INSERT and UPDATE statements through views, the select list in the view definition must not contain any arithmetic expressions. It must contain only column names.
  - For DELETE WHERE CURRENT and UPDATE WHERE CURRENT statements operating on cursors defined with views, the view definition must not contain subqueries.
  - For noncursor UPDATE, DELETE, and INSERT statements, the view definition must not contain any subqueries which contain in their FROM clauses a table reference to the same table as the outermost FROM clause.
- You cannot define an index on a view or alter a view.
- You cannot use host variables, local variables, procedure parameters, or dynamic parameters in the CREATE VIEW statement.
- Creating a view causes a section to be stored in the system catalog. A description of the section appears in the SYSTEM.SECTION view.
- If you use the CREATE VIEW statement within the CREATE SCHEMA statement, the default owner of the view is the schema's AuthorizationName.
- When you create a view, an entry containing the SELECT statement in the view definition is stored in the SYSTEM.VIEWDEF view in the system catalog. The view's name is stored in SYSTEM.TABLE, and the description of its columns appears in SYSTEM.COLUMN.
- If you use the CREATE VIEW statement with a CREATE PROCEDURE statement, the default owner is the procedure owner.
- Any attempt to write through a view defined having a WITH CHECK OPTION must satisfy any conditions specified in the query specification. All underlying view definitions are also checked. Any constraints in the table on which the view is based are also checked.
- View check constraints are not deferrable.
- To drop a constraint on a view, you must drop the view and recreate it without the

constraint.

- You cannot use an ORDER BY clause when defining a view.
- If the IN *DBEFileSetName* clause is specified, but the view owner does not have SECTIONSPACE authority for the specified DBEFileSet, a warning is issued and the default SECTIONSPACE DBEFileSet is used instead. (Refer to syntax for the GRANT statement and the SET DBEFILESET statement.)

Authorization

You can create a view if you have SELECT or OWNER authority for the tables and views mentioned in the FROM clause of the SELECT statement or if you have DBA authority. To operate on a table on which the view is based, the authority you need depends on whether or not you own the view. The authority needed in either case is specified as follows:

- If you own the view, you need authority for the table(s) or view(s) on which the view is based.
- If you do not own the view, you need authority granted specifically for the view. To specify a *DBEFileSetName* for a view, the view owner must have SECTIONSPACE authority on the referenced DBEFileSet.

Examples

1. The following view provides information on the value of current orders for each vendor. Because the view is derived by joining tables, the base tables cannot be updated via this view.

```
CREATE VIEW PurchDB.VendorStatistics
            (VendorNumber,
            VendorName,
            OrderDate,
            OrderQuantity,
            TotalPrice)
  AS SELECT PurchDB.Vendors.VendorNumber,
            PurchDB.Vendors.VendorName,
            OrderDate,
            OrderQty,
            OrderQty*PurchasePrice
       FROM PurchDB.Vendors,
            PurchDB.Orders,
            PurchDB.OrderItems
      WHERE PurchDB.Vendors.VendorNumber =
            PurchDB.Orders.VendorNumber
        AND PurchDB.Orders.OrderNumber =
            PurchDB.OrderItems.OrderNumber
IN PurchDBFileSet
```

2. The following view is updatable because it is created from one table. When the table is updated through the view, column values in the SET or VALUES clause are checked against the WHERE clause in the view definitions.

If the table on which the view is based has any check constraints of its own, these conditions are checked along with the WITH CHECK OPTION of the view.

```
CREATE VIEW RecDB.EventView
            (Event,
             Date)
 AS  SELECT RecDB.Event,
            RecDB.Date
       FROM RecDB.Events
      WHERE Date >= CURRENT_DATE
            WITH CHECK OPTION CONSTRAINT EventView_WCO
```

DECLARE CURSOR

The DECLARE CURSOR statement associates a cursor with a specified SELECT or EXECUTE PROCEDURE statement.

Scope

Application Programs and Procedures

SQL Syntax

```
DECLARE CursorName [IN DBEFileSetName]CURSOR FOR
{ {QueryExpression
   SelectStatementName}[FOR UPDATE OF {ColumnName}[,...]
                        FOR READ ONLY]
  ExecuteProcedureStatement
  ExecuteStatementName                                  }
```

Parameters

CursorName is the name assigned to the newly declared cursor. Two cursors in an application program cannot have the same name. The cursor name must conform to the SQL syntax rules for a basic name, described in the "Names" chapter of this manual, and must also conform to the requirements of the application programming language.

DBEFileSetName identifies the DBEFileSet in which ALLBASE/SQL is to store the section associated with the cursor. If not specified, the default SECTIONSPACE DBEFileSet is used.

QueryExpression is a static SELECT statement. It determines the rows and columns to be processed by means of a select cursor. The rows defined by the query expression when you open the cursor are called the active set of the cursor. Parentheses are optional.

The BULK and INTO clauses and dynamic parameters are disallowed.

SelectStatementName is specified when declaring a select cursor for a dynamically preprocessed SELECT statement. It is the *StatementName* specified in the related PREPARE statement.

FOR UPDATE OF *ColumnName* specifies the column or columns which may be updated using this cursor. The order of the column names is not important. The column(s) to be updated need not appear in the select list of the SELECT statement. If you use a FOR UPDATE clause, the query expression must be updatable.

FOR READ ONLY indicates that data is to be read and not updated. Specify this clause when you preprocess and application using the FIPS 127.1 flagger, and the cursor you are declaring reads and does not update columns. FOR READ ONLY assures optimum performance in this case.

ExecuteProcedureStatement is a static EXECUTE PROCEDURE statement. It determines the rows and columns of the query result set or sets to be processed by means of a procedure cursor. The rows defined when you open and advance the cursor are called the active set of the cursor.

ExecuteStatementName is specified when declaring a procedure cursor for a dynamically preprocessed EXECUTE PROCEDURE statement. It is the *StatementName* specified in the related PREPARE statement.

Dynamic parameters are allowed in *ExecuteStatementName*.

Description

- There are two types of cursors. A **select cursor** is a pointer used to indicate the current row in a set of rows retrieved by a SELECT statement. A **procedure cursor** is a pointer used to indicate the current result set and row in result sets retrieved by SELECT statements in a procedure and returned to a calling application or ISQL.
- The DECLARE CURSOR statement cannot be used interactively.
- A cursor must be declared before you refer to it in other cursor manipulation statements.
- The active set is defined and the value of any host variables in the associated SELECT or EXECUTE PROCEDURE statement is evaluated when you issue the OPEN statement.
- Use the FETCH statement to move through the rows of the active set.
- For procedure cursors only, use the ADVANCE statement to move to the next active set (query) within a procedure.
- For select cursors only, you can operate on the current row in the active set (the most recently fetched row) with the UPDATE WHERE CURRENT and DELETE WHERE CURRENT statements.

  When using the Read Committed or Read Uncommitted isolation levels, use the REFETCH statement to verify that the row you want to update or delete still exists.
- A select cursor is said to be updatable when you can use it in DELETE WHERE CURRENT OF CURSOR or UPDATE WHERE CURRENT OF CURSOR statements to modify the base table. A select cursor is updatable only if the query from which it is derived matches the following updatability criteria:
  - No ORDER BY, UNION, or UNION ALL operation is specified.
  - No DISTINCT, GROUP BY, or HAVING clause is specified in the outermost SELECT clause, and no aggregate appears in its select list.
  - The FROM clause specifies exactly one table, whether directly or through a view. If it specifies a table, the table must be an updatable table. If it specifies a view, the view definition must satisfy the cursor updatability rules stated here.
  - For the UPDATE WHERE CURRENT statement, you can only update columns in the FOR UPDATE list.
  - For DELETE WHERE CURRENT and UPDATE WHERE CURRENT statements, the *SelectStatement* parameter must not contain any subqueries or reference any

view whose view definition contains a subquery.

- For select cursors only, use the UPDATE statement with the CURRENT OF option to update columns; you can update the columns identified in the FOR UPDATE OF clause of the DECLARE CURSOR statement. The restrictions that govern updating via a select cursor are described above.
- For select cursors only, use the DELETE WHERE CURRENT statement to delete a row in the active set.
- Use the CLOSE statement when you are finished operating on the active set or (for a procedure cursor) set(s).
- Declaring a cursor causes a section to be stored in the system catalog. A description of the section appears in the SYSTEM.SECTION view.
- The *ExecuteStatementName*, *SelectStatementName*, and *ExecuteProcedureStatement* parameters of the DECLARE CURSOR statement are not allowed within a procedure.
- Host variables for return status and input and output parameters are allowed in *ExecuteProcedureStatement*, which is a static EXECUTE PROCEDURE statement. The appropriate values for input host variables must be set before the OPEN statement. The output host variables, including return status and output parameters from executing the procedure are accessible after the CLOSE statement.
- Dynamic parameters for return status and input and output parameters of the procedure are allowed in *ExecuteStatementName*. The appropriate values for any input dynamic parameters or host variables must be placed into the SQLDA or host variables before issuing the OPEN statement. The USING DESCRIPTOR clause of the FETCH statement is used to identify where to place selected rows and properly display the returned data. Output host variables or values in the SQLDA, including return status and output parameters from executing the procedure, are accessible after the CLOSE statement executes.
- If the IN *DBEFileSetName* clause is specified, but the module owner does not have SECTIONSPACE authority for the specified DBEFileSet, a warning is issued and the default SECTIONSPACE DBEFileSet is used instead. (Refer to syntax for the GRANT statement and the SET DEFAULT DBEFILESET statement.)

Authorization

For a select cursor, you must have SELECT or OWNER authority for all the tables or views listed in the FROM clause, or you must have DBA authority.

For a procedure cursor, you must have OWNER or EXECUTE authority on the procedure or DBA authority.

If you specify the FOR UPDATE clause, you must also have authority to update the specified columns.

To specify a *DBEFileSetName* for a cursor, the cursor owner must have SECTIONSPACE authority on the referenced DBEFileSet.

Examples

1. Deleting with a cursor

   The active set of this cursor will contain values for the OrderNumber stored in :OrdNum.

```
DECLARE DeleteItemsCursor CURSOR FOR
 SELECT ItemNumber,OrderQty FROM PurchDB.OrderItems
  WHERE OrderNumber = :OrdNum
```

   Statements setting up a FETCH-DELETE WHERE CURRENT loop appear here.

```
OPEN DeleteItemsCursor
```

   Statements for displaying values and requesting whether the user wants to delete the associated row go here.

```
FETCH DeleteItemsCursor
 INTO :Lin :Linnul, :Orq :Orqnul

     DELETE FROM PurchDB.OrderItems
WHERE CURRENT OF DeleteItemsCursor
   .
   .
   .
CLOSE DeleteItemsCursor
```

2. Updating with a cursor

   A cursor for use in updating values in column QtyOnHand is declared and opened.

```
       DECLARE NewQtyCursor CURSOR FOR
        SELECT PartNumber,QtyOnHand FROM PurchDB.Inventory
 FOR UPDATE OF QtyOnHand

OPEN NewQtyCursor
```

   Statements setting up a FETCH-UPDATE loop appear next.

```
FETCH NewQtyCursor INTO :Num :NumNul, :Qty :Qtynul
```

   Statements for displaying a row to and accepting a new QtyOnHand value from the user go here. The new value is stored in :NewQty.

```
          UPDATE PurchDB.Inventory
             SET QtyOnHand = :NewQty
WHERE CURRENT OF NewQtyCursor
   .
   .
   .
CLOSE NewQtyCursor
```

3. Bulk fetching

   In some instances, using the BULK option is more efficient than advancing the cursor a row at a time through many rows, especially when you want to operate on the rows with non-ALLBASE/SQL statements.

```
DECLARE ManyRows CURSOR FOR
 SELECT *
   FROM PurchDB.Inventory

OPEN ManyRows

BULK FETCH ManyRows INTO :Rows, :Start, :NumRow
```

4. Dynamically preprocessed SELECT

   If you know in advance that the statement to be dynamically preprocessed is not a SELECT statement, you can prepare it and execute it in one step. In other instances, it is more appropriate to prepare and execute the statement in separate operations.

```
EXECUTE IMMEDIATE :Dynam1
```

   The statement stored in :Dynam1 is dynamically preprocessed.

```
PREPARE Dynamic1 FROM :Dynam1
```

   If Dynamic1 is not a SELECT statement, the SQLD field of the SQLDA data structure is 0, and you use the EXECUTE statement to execute the dynamically preprocessed statement.

```
DESCRIBE Dynamic1 INTO SQLDA

EXECUTE Dynamic1
```

   If Dynamic1 is a SELECT statement and the language you are using supports dynamically defined SELECT statements, use a cursor to manipulate the rows in the query result.

   After you open the cursor and place the appropriate values into the SQL Descriptor Area (SQLDA), use the USING DESCRIPTOR clause of the FETCH statement to identify where to place the rows selected and properly display the returned data.

```
DECLARE Dynamic1Cursor CURSOR FOR Dynamic1

OPEN Dynamic1Cursor

FETCH Dynamic1Cursor USING DESCRIPTOR SQLDA
.
.
.
CLOSE Dynamic1Cursor
```

5. Refer to the *ALLBASE/SQL Advanced Application Programming Guide* for a pseudocode example of procedure cursor usage.

DECLARE *Variable*

The DECLARE *Variable* statement lets you define a local variable within a procedure. Local variables are used only within the procedure.

Scope

Procedures only

SQL Syntax

```
DECLARE { LocalVariable}[,...] VariableType {LANG = VariableLangName]
[DEFAULT {Constant
          NULL
          CurrentFunction}][NOT NULL]
```

Parameters

LocalVariable specifies the name of the local variable. A variable name may not be the same as a parameter name in the same procedure.

VariableType specifies the data type of the local variable. All the ALLBASE/SQL data types are permitted except LONG data types.

VariableLangName specifies the language of the data (for character data types only) to be stored in the local variable. This name must be either n-computer or the current language of the DBEnvironment.

DEFAULT specifies the default value of the local variable. The default can be a constant, NULL, or a date/time current function. The data type of the default value must be compatible with the data type of the variable.

NOT NULL means the variable cannot contain null values. If NOT NULL is specified, any statement that attempts to place a null value in the variable is rejected.

Description

- Declarations must appear at the beginning of the stored procedure body, following the first BEGIN statement.
- No two local variables or parameters in a procedure may have the same name.
- Local variable declarations may not be preceded by labels.
- If no DEFAULT clause is given for a column in the table, an implicit DEFAULT NULL is assumed. Any INSERT statement, which does not include a column for which a default has been declared, causes the default value to be inserted into that column for all rows inserted.
- For a CHAR column, if the specified default value is shorter in length than the target column, it is padded with blanks. For a CHAR or VARCHAR column, if the specified

default value is longer than the target column, it is truncated.

- For a BINARY column, if the specified default value is shorter in length than the target column, it is padded with zeroes. For a BINARY or VARBINARY column, if the specified default value is longer than the target column, it is truncated.

Authorization

Anyone can use the DECLARE statement in a procedure.

Example

```
DECLARE input, output CHAR(80);
DECLARE nrows INTEGER;
DECLARE PartNumber CHAR(16) NOT NULL;
```

DELETE

The DELETE statement deletes a row or rows from a table.

Scope

ISQL or Application Programs

SQL Syntax

```
DELETE [WITH AUTOCOMMIT]FROM {[Owner.]TableName
                              [Owner.]ViewName} [WHERE SearchCondition
```

Parameters

`WITH AUTOCOMMIT` executes a `COMMIT WORK` automatically at the beginning of the `DELETE` statement and also after each batch of rows is deleted.

`[`*`Owner`*`.]`*`TableName`* designates a table from which any rows satisfying the search condition are to be deleted.

`[`*`Owner`*`.]`*`ViewName`* designates a view based on a single table. ALLBASE/SQL finds which rows of the view satisfy the search condition; the corresponding rows of the view's base table are deleted. Refer to the `CREATE VIEW` statement for restrictions governing modifications via a view.

`WHERE` *`SearchCondition`* specifies which rows are to be deleted. If no rows satisfy the search condition, the table is not changed. If the WHERE clause is omitted, all rows are deleted.

Description

- If all the rows of a table are deleted, the table is empty but continues to exist until you issue a `DROP TABLE` statement.
- Use the `TRUNCATE TABLE` statement to delete all rows from a table instead of the `DELETE` statement. The `TRUNCATE TABLE` statement is faster, and generates fewer log records.
- If ALLBASE/SQL detects an error during a `DELETE` statement, the action taken will vary, depending on the setting of the `SET DML ATOMICITY`, and the `SET CONSTRAINTS` statements. Refer to the description of both of these statements in this chapter for more details.
- Using `DELETE` with views requires that the views be based on updatable queries. See "Updatability of Queries" in the "SQL Queries" chapter.
- The target table of the `DELETE` statement is specified with *`TableName`* or is the base table underlying the view definition of *`ViewName`*. It must be an updatable table, and it must *not* appear in the FROM clause of any subquery specified in the *`SearchCondition`* parameter or any subquery of *`ViewName`*.

- The search condition is effectively executed for each row of the table or view before any row is deleted. If the search condition contains a subquery, each subquery in the search condition is effectively executed for each row of the table or view and the results used in the application of the search condition to the given row. If any executed subquery contains an outer reference to a column of the table or view, the reference is to the value of that column in the given row.
- A deletion from a table with a primary key (a referenced unique constraint) fails if any primary key row affected by the DELETE statement is currently referred to by some referencing foreign key row. In order to delete such referenced rows, you must first change the referencing foreign key rows to refer to other primary key rows, to contain a NULL value in one of the foreign key columns, or to delete these referencing rows. Alternatively, you can defer error checking (with the SET CONSTRAINT statement) and fix the error later.
- The DELETE syntax is unchanged for use with LONG columns. It is limited in that a LONG column cannot be used in the WHERE clause. When LONG data is deleted, the space it occupied in the DBEnvironment is released when your transaction ends. But the physical operating system data file created when you selected the long field earlier still exists and you are responsible for removing it if you desire.
- A check constraint search condition defined on a table never prevents a row from being deleted, whether or not constraint checking is deferred.
- A rule defined with a *StatementType* of DELETE will affect DELETE statements performed on the rule's target table. When the DELETE is performed, each rule defined on that operation for the table is considered. If the rule has no condition, it will fire for all rows affected by the statement and invoke its associated procedure with the specified parameters on each row. If the rule has a condition, it will evaluate the condition on each row. The rule will fire on rows for which the condition evaluates to TRUE and invoke the associated procedure with the specified parameters for each row. Invoking the procedure could cause other rules, and thus other procedures, to be invoked if statements within the procedure trigger other rules.
- If a DISABLE RULES statement is in effect, the DELETE statement will not fire any otherwise applicable rules. When a subsequent ENABLE RULES is issued, applicable rules will fire again, but only for subsequent DELETE statements, not for those processed when rule firing was disabled.
- In a rule defined with a *StatementType* of DELETE, any column reference in the *Condition* or any *ParameterValue* will refer to the value of the column as it exists in the database before it is removed by the DELETE statement, regardless of the use of *OldCorrelationName*, *TableName*, or *NewCorrelationName* in the rule definition.
- The set of rows to be affected by the DELETE statement is determined before any rule fires, and this set remains fixed until the completion of the rule. If the rule adds to, deletes from, or modifies this set, such changes are ignored.
- When a rule is fired by this statement, the rule's procedure is invoked after the changes have been made to the database for that row and all previous rows. The rule's procedure, and any chained rules, will thus see the state of the database with the current partial execution of the statement.
- If an error occurs during processing of any rule considered during execution of this

statement (including execution of any procedure invoked due to a rule firing), the statement and any procedures invoked by any rules have no effect, regardless of the current DML ATOMICITY. Nothing has been altered in the DBEnvironment as a result of this statement or the rules it fired. Error messages are returned in the normal way.

- When the WITH AUTOCOMMIT clause is *not* used, rows that qualify according to the SearchCondition are deleted internally in batches by ALLBASE/SQL.

  When the WITH AUTOCOMMIT clause is used, a COMMIT WORK statement is executed automatically at the beginning of the DELETE statement and also after each batch of rows is deleted. This can reduce both log-space and shared-memory requirements for the DELETE statement. You cannot control the number of rows in each batch.

- The WITH AUTOCOMMIT clause cannot be used in these cases:
  - When deleting rows from a TurboIMAGE data set.
  - If a SET CONSTRAINTS DEFERRED statement is in effect.
  - If a rule exists on the table and rules are enabled for the DBEnvironment. Consider issuing a DISABLE RULES statement to temporarily disable rules for the DBEnvironment, issuing the DELETE WITH AUTOCOMMIT statement, and then issuing an ENABLE RULES statement to turn rule checking back on.
  - In the DELETE WHERE CURRENT statement.
- If an active transaction exists when the DELETE WITH AUTOCOMMIT is issued, then the existing transaction is committed.
- When WITH AUTOCOMMIT is used, any previously issued SET DML ATOMICITY statements are ignored. For the duration of that DELETE command, row-level atomicity is used.
- If the DELETE WITH AUTOCOMMIT statement fails, it may be true that some (but not all) rows that qualify have been deleted.
- The DELETE WITH AUTOCOMMIT statement can be used in procedures, but a rule may not execute that procedure.

Authorization

If you specify the name of a table, you must have DELETE or OWNER authority for that table or you must have DBA authority.

If you specify the name of a view, you must have DELETE or OWNER authority for that view or you must have DBA authority. Also, the owner of the view must have DELETE or OWNER authority with respect to the view's base tables, or the owner must have DBA authority.

Example

Rows for orders created prior to July 1983 are deleted.

```
DELETE WITH AUTOCOMMIT FROM PurchDB.Orders
              WHERE OrderDate < '19830701'
```

DELETE WHERE CURRENT

The DELETE WHERE CURRENT statement deletes the current row of an active set. The current row is the row pointed to by a cursor after the FETCH or REFETCH statement is issued.

Scope

Application Programs

SQL Syntax

```
DELETE FROM {[Owner.]TableName
             [Owner.]ViewName} WHERE CURRENT OF CursorName
```

Parameters

[*Owner.*]*TableName* designates the table from which you are deleting a row.

[*Owner.*]*ViewName* designates a view based on a single table. ALLBASE/SQL finds the row of the base table corresponding to the row of the view indicated by the cursor, and deletes the row from the base table. Refer to the CREATE VIEW statement for restrictions governing modifications via a view.

CursorName specifies the name of a cursor. The cursor must be open and positioned on a row of the table. The DELETE WHERE CURRENT statement deletes this row, leaving the cursor with no current row. (The cursor is said to be positioned between the preceding and following rows of the active set). You cannot use the cursor for further updates or deletions until you reposition it using a FETCH statement, or until you close and reopen the cursor.

Description

- This statement cannot be used interactively.
- Although the SELECT statement associated with the cursor may specify only some of the columns in a table, the DELETE WHERE CURRENT statement deletes an entire row.
- The DELETE WHERE CURRENT statement can be used on an active set associated with a cursor defined using the FOR UPDATE clause.
- Do not use this statement in conjunction with rows retrieved using a BULK FETCH.
- Using the DELETE statement with the WHERE CURRENT OF CURSOR clause requires that the cursor be defined on the basis of an updatable query. See "Updatability of Queries" in the "SQL Queries" chapter.
- The target table of the DELETE WHERE CURRENT statement is specified with *TableName* or is the base table underlying *ViewName*. The base table restrictions that govern deletions via cursors are presented in the description of the DECLARE CURSOR statement.

- If a referential constraint should be violated during processing of the DELETE statement, the row is not deleted (unless error checking is deferred and the violation is corrected before you COMMIT WORK). Refer to the discussion of the SET CONSTRAINTS statement in this chapter for more information.
- A deletion from a table with a primary key (a referenced unique constraint) will fail if any primary key row affected by the DELETE statement is currently referred to by some referencing foreign key row. In order to delete such referenced rows, you must first change the referencing foreign key rows to refer to other primary key rows, to contain a NULL value in one of the foreign key columns, or to delete these referencing rows. Alternatively, you can defer error checking (with the SET CONSTRAINT statement) and fix the error later.
- The DELETE syntax is unchanged for use with LONG columns. When LONG data is deleted, the space it occupied in the DBEnvironment is released when your transaction ends. But the physical operating system data file created when you selected the long field earlier still exists and you are responsible for removing it if you desire.
- A rule defined with a *StatementType* of DELETE will affect DELETE WHERE CURRENT statements performed on the rule's target table. When the DELETE WHERE CURRENT is performed, each rule defined on that operation for the table is considered. If the rule has no condition, it will fire and invoke its associated procedure with the specified parameters on the current row. If the rule has a condition, it will evaluate the condition and fire if the condition evaluates to TRUE and invoke the associated procedure with the specified parameters on the current row. Invoking the procedure could cause other rules, and thus other procedures, to be invoked if statements within the procedure trigger other rules.
- If a DISABLE RULES statement is in effect, the DELETE WHERE CURRENT statement will not fire any otherwise applicable rules. When a subsequent ENABLE RULES is issued, applicable rules will fire again, but only for subsequent DELETE WHERE CURRENT statements, not for those rows processed when rule firing was disabled.
- In a rule defined with a *StatementType* of DELETE, any column reference in the *Condition* or any *ParameterValue* will refer to the value of the column as it exists in the database before it is removed by the DELETE WHERE CURRENT statement, regardless of the use of *OldCorrelationName*, *TableName*, or *NewCorrelationName* in the rule definition.
- When a rule is fired by this statement, the rule's procedure is invoked after the changes have been made to the database for that row. The rule's procedure, and any chained rules, will thus see the state of the database with the current partial execution of the statement.
- If an error occurs during processing of any rule considered during execution of this statement (including execution of any procedure invoked due to a rule firing), the statement and any procedures invoked by any rules will have no effect. Nothing will have been altered in the DBEnvironment as a result of this statement or the rules it fired. Error messages are returned in the normal way.

Authorization

If you specify the name of a table, you must have DELETE or OWNER authority for that

table or you must have DBA authority.

If you specify the name of a view, you must have DELETE or OWNER authority for that view or you must have DBA authority. Also, the owner of the view must have DELETE or OWNER authority with respect to the view's base tables, or the owner must have DBA authority.

Example

The active set of this cursor will contain values for the OrderNumber stored in :OrdNum.

```
DECLARE DeleteItemsCursor CURSOR FOR
 SELECT ItemNumber,OrderQty FROM PurchDB.OrderItems
  WHERE OrderNumber = :OrdNum
```

Statements setting up a FETCH-DELETE WHERE CURRENT loop appear here.

```
OPEN DeleteItemsCursor
```

Statements for displaying values and requesting whether the user wants to delete the associated row go here.

```
FETCH DeleteItemsCursor INTO :Lin :Linnul, :Orq :Orqnul

  DELETE FROM PurchDB.OrderItems
WHERE CURRENT OF DeleteItemsCursor
.
.
.
CLOSE DeleteItemsCursor
```

DESCRIBE

The DESCRIBE statement is used in an application program to pass information about a dynamic statement between the application and ALLBASE/SQL. It must refer to a statement preprocessed with the PREPARE statement.

Scope

C and Pascal Applications Only

SQL Syntax

```
DESCRIBE [OUTPUT
          INPUT
          RESULT] StatementName {INTO [[SQL] DESCRIPTOR]
                                 USING [SQL] DESCRIPTOR} {SQLDA
                                                          AreaName}
```

Parameters

OUTPUT — specifies that the characteristics of any output values in the prepared *StatementName* be described in the associated sqlda_type and sqlformat_type data structures. This applies to query result column definitions in a SELECT statement or to dynamic return status or output parameters specified as question marks in an EXECUTE PROCEDURE statement.

OUTPUT is the default.

INPUT — specifies that the characteristics of any dynamic input parameters in the prepared *StatementName* be described in the associated sqlda_type and sqlformat_type data structures. This applies to dynamic input parameters specified as question marks in any DML statement.

RESULT — specifies that the characteristics of any single format multiple row result sets in a procedure created using the WITH RESULT clause be described in the associated sqlda_type and sqlformat_type data structures. This applies to any prepared EXECUTE PROCEDURE statement.

StatementName identifies a previously preprocessed (prepared) ALLBASE/SQL statement.

INTO — specifies the sqlda_type data structure where data is to be described.

USING — specifies the sqlda_type data structure where data is to be described.

SQLDA — specifies that a data structure of sqlda_type named sqlda is to be used to pass information about the prepared statement between the application and ALLBASE/SQL.

AreaName — specifies the user defined name of a data structure of sqlda_type that is to be used to pass information about the prepared statement between the application and ALLBASE/SQL.

Description

- This statement cannot be used in ISQL, in COBOL and FORTRAN programs, or in procedures.
- If *StatementName* refers to a SELECT statement, the DESCRIBE statement with the (default) OUTPUT option sets the *sqld* field of the associated sqlda_type data structure to the number of columns in the query result and sets the associated sqlformat_type data structure to each column's name, length, and data type. On the basis on this information, an application can parse a data buffer to obtain the column values in the query result. The application reads the query result by associating the *StatementName* with a select cursor and using select cursor manipulation statements (OPEN, FETCH, and CLOSE).
- If *StatementName* does not refer to a SELECT statement, the DESCRIBE statement used with the OUTPUT option sets the *sqld* field of the associated sqlda_type data structure to zero.
- If *StatementName* refers to a statement in which dynamic parameters have been specified, the DESCRIBE statement with the INPUT option obtains the number of input dynamic parameters (in the *sqld* field of the associated sqlda_type data structure) and sets the associated sqlformat_type data structure to each column's name, length, and data type. The application can use this information to load the appropriate data buffer with dynamic parameter values.
- If *StatementName* refers to an EXECUTE PROCEDURE statement for a procedure with multiple row result sets, the *sqlmproc* field of the associated sqlda_type data structure is set to a non-zero value. The program reads the query results by associating the *StatementName* with a procedure cursor name and using procedure cursor manipulation statements (OPEN, ADVANCE, FETCH, and CLOSE).
- If *StatementName* refers to an EXECUTE PROCEDURE statement containing output dynamic parameters, the DESCRIBE statement with the (default) OUTPUT option returns the number of output dynamic parameters in the *sqloparm* field of the associated sqlda_type data structure.
- If *StatementName* refers to an EXECUTE PROCEDURE statement containing both input and output dynamic parameters, you can issue the EXECUTE statement specifying the USING INPUT AND OUTPUT option to execute the dynamically preprocessed statement.
- If *StatementName* is an EXECUTE PROCEDURE statement containing single format multiple row result set(s), the DESCRIBE statement with the RESULT option returns the format information of the multiple row result set(s). If the procedure contains more than one multiple row result set, all must return rows with compatible formats.
- If the RESULT option is specified when describing an EXECUTE PROCEDURE statement for a procedure created with no WITH RESULT clause, the sqld field of the related SQLDA is set to zero, and no format information is written to the SQL descriptor area.
- If the RESULT option is specified when describing a statement other than an EXECUTE PROCEDURE statement, the DESCRIBE RESULT statement returns an error, and nothing is written to the SQL descriptor area.

- Detailed descriptions of how to use this statement are found in the "Using Dynamic Operations" chapters of the *ALLBASE/SQL C Application Programming Guide* and the *ALLBASE/SQL Advanced Application Programming Guide*, and in the "Using Parameter Substitution in Dynamic Statements" chapter and the "Using Procedures in Application Programs" chapter of the *ALLBASE/SQL Advanced Application Programming Guide*.

Authorization

To describe a previously preprocessed SELECT statement, you must have authority that would permit you to execute the SELECT statement. To describe a previously preprocessed EXECUTE PROCEDURE statement, you must have authority that would permit you to execute the procedure. You do not need authorization to describe other previously preprocessed statements.

Examples

1. Prepared statement with known format

   If you know in advance that the statement to be dynamically preprocessed is neither a SELECT statement nor an EXECUTE PROCEDURE statement with results, and does not contain dynamic parameters nor input/output host variables, you can prepare it and execute it in one step, as follows:

   ```
   EXECUTE IMMEDIATE :Dynam1
   ```

2. Prepared statement with unknown format

   In other instances, it is more appropriate to prepare and execute the statement in separate operations. For example, if you don't know the format of a statement, you could do the following:

   ```
   PREPARE Dynamic1 FROM :Dynam1
   ```

   The statement stored in :Dynam1 is dynamically preprocessed.

   ```
   DESCRIBE Dynamic1 INTO SqldaOut
   ```

   If Dynamic1 is neither a SELECT statement (Sqld field of the Sqlda data structure is 0) nor an EXECUTE PROCEDURE statement with results (sqlmproc = 0) and you know there are no dynamic parameters in the prepared statement, use the EXECUTE statement to execute the dynamically preprocessed statement.

   If Dynamic1 is an EXECUTE PROCEDURE statement containing dynamic output parameters, the sqloparm field of the Sqlda data structure contains the number of such parameters in the statement. You can access the appropriate format array and data buffer to obtain the data.

   If it is possible that dynamic input parameters are present in the prepared statement or that the statement is an EXECUTE PROCEDURE statement for a procedure with multiple row result sets, you must further describe it. See the exproc function below which emphasizes steps needed to process an EXECUTE PROCEDURE statement for a procedure with multiple row result sets.

   To check for dynamic input parameters in any type of DML statement, describe the

statement for input:

```
DESCRIBE INPUT Dynamic1 USING SQL DESCRIPTOR SqldaIn
```

If dynamic input parameters are present, the appropriate data buffer or host variables must be loaded with the values of any dynamic parameters. Then if the statement is not a query, it can be executed, as in this example using a data buffer:

```
EXECUTE Dynamic1 USING SQL DESCRIPTOR SqldaIn
```

If Dynamic1 is a SELECT statement and the language you are using supports dynamically defined SELECT statements, use a cursor to manipulate the rows in the query result:

```
DECLARE Dynamic1Cursor CURSOR FOR Dynamic1
```

Place the appropriate values into the SQL descriptor areas. Use the USING DESCRIPTOR clause of the OPEN statement to identify where dynamic input parameter information is located. Load related dynamic parameter data into the input data buffer.

```
OPEN Dynamic1Cursor USING SQL DESCRIPTOR SqldaIn
```

Use the USING DESCRIPTOR clause of the FETCH statement to identify where to place the rows selected.

```
FETCH Dynamic1Cursor USING DESCRIPTOR SqldaOut
.
.
.
```

When all rows have been processed, close the cursor:

```
CLOSE Dynamic1Cursor
```

3. Prepared statement is EXECUTE PROCEDURE

   If the described statement is an EXECUTE PROCEDURE statement for a procedure with multiple row result sets, the sqlmproc field of the sqlda data structure contains the number of multiple row result sets (0 if there are none) following execution of the DESCRIBE statement with default OUTPUT option. For example, if the statement you described looks like the following, and the procedure was created with two multiple row result SELECT statements and a WITH RESULT clause:

   ```
   DynamicCmd = "EXECUTE PROCEDURE ? = proc(?, ? OUTPUT)"

   PREPARE cmd FROM :DynamicCmd
   ```

   Assuming you don't know the format of this prepared statement:

   ```
   DESCRIBE OUTPUT cmd INTO sqldaout
   ```

   The sqld of sqlda is set to 0, sqlmproc to 2, and sqloparm to 2.

   ```
   DESCRIBE INPUT cmd USING sqldain
   ```

The sqld of sqlda is set to 2, sqlmproc to 2, and sqloparm to 0.

a. If sqldaout.*sqlmproc* <> 0 then, use procedure cursor processing statements to process multiple row result set(s) from the procedure.

```
DESCRIBE RESULT cmd USING sqldaresult
  .
  .
  .
  DECLARE Dynamic1Cursor CURSOR FOR cmd
  OPEN Dynamic1Cursor USING sqldain
  .
  .
  .
  FETCH Dynamic1Cursor using DESCRIPTOR sqldaresult
  .
  .
  .
  CLOSE Dynamic1Cursor USING sqldaout
  .
  .
  .
```

b. Else, execute the procedure with both input and output dynamic parameters.

```
EXECUTE cmd USING DESCRIPTOR INPUT sqldain AND OUTPUT
sqldaout;
```

DISABLE AUDIT LOGGING

The DISABLE AUDIT LOGGING statement stops audit logging for the DBEnvironment session. It allows you to avoid creating audit log records for SQL statements while hard resynchronization is performed.

Scope

ISQL or Application Programs

SQL Syntax

```
DISABLE AUDIT LOGGING
```

Description

- This statement disables audit logging in the current session only. It suspends the generation of audit log records for any statements issued during the session.
- This statement and ENABLE AUDIT LOGGING are *not* used to turn on and off the AUDIT LOG option specified for processing of all sessions in the DBEnvironment. These statements affect your current session only. (The statements that affect *all* processing in the DBEnvironment are the START DBE NEW and START DBE NEWLOG statements.)
- This statement is not affected by transaction management statements and remains in effect until an ENABLE AUDIT LOGGING statement is issued, or until the end of the current session.

Authorization

This statement requires DBA authority.

Example

Perform an initial load of a table without audit logging.

```
DISABLE AUDIT LOGGING;

LOAD FROM INTERNAL PartsData TO PurchDB.Parts;

COMMIT WORK;
```

Reenable audit logging and continue.

```
ENABLE AUDIT LOGGING;
```

DISABLE RULES

The DISABLE RULES statement turns rule checking off for the current DBEnvironment session. The statement is for DBA use in testing the operation of rules.

Scope

ISQL or Application Programs

SQL Syntax

```
DISABLE RULES
```

Description

- DISABLE RULES turns rule invocation off in the DBEnvironment for the current session or until the ENABLE RULES statement is issued.
- The statement only affects the current SID (session id). Other users are not affected.
- The DISABLE RULES statement is not cumulative; issuing additional DISABLE RULES statements will have no effect, and a warning will be issued to this effect.
- Rules are not fired retroactively when the ENABLE RULES statement is issued after the DISABLE RULES statement has been issued. If a DISABLE RULES statement is issued, rules that would otherwise be applicable will not fire. Then, when a subsequent ENABLE RULES is issued, applicable rules fire again, but only for subsequent data manipulation statements, not for those statements executed while rule firing was disabled.
- COMMIT WORK and ROLLBACK WORK statements have no effect on whether rules are enabled or disabled.

Authorization

You must have DBA authority.

Example

The DBA turns off rule invocation for the current session.

```
DISABLE RULES
```

The DBA performs operations without rule firing.

.
.
.

The DBA turns on rule invocation.

```
ENABLE RULES
```

Normal firing of rules resumes.

DISCONNECT

The DISCONNECT statement terminates a connection with a DBEnvironment or terminates all DBEnvironment connections established within an application or an ISQL session.

Scope

ISQL or Application Programs

SQL Syntax

```
DISCONNECT {'ConnectionName'
            'DBEnvironmentName'
            :HostVariable
            ALL
            CURRENT            }
```

Parameters

ConnectionName is a string literal identifying the name associated with this connection. *ConnectionName* must be unique for each DBEnvironment connection within an application (or ISQL). *ConnectionName* cannot exceed 128 bytes.

'*DBEnvironmentName*' is the DBEnvironment to which you have connected to using a CONNECT TO '*DBEnvironmentName*' statement.

HostVariable is a character string host variable containing the *ConnectionName* associated with this connection.

ALL specifies that all DBEnvironment connections in effect (for an application or an ISQL session) are to be terminated.

CURRENT specifies that the current connection is to be terminated. Within an application (or ISQL), the **current connection** to a DBEnvironment is set by the most recent statement that connects to or sets the connection to the DBEnvironment. If there is no current connection in effect, an error is generated.

Description

- If a *ConnectionName* refers to a DBEnvironment that is not the one associated with the current connection, the specified connection is terminated, and the context of the currently connected DBEnvironment remains unchanged.
- Any active transaction associated with a connection is rolled back before the connection is terminated.
- No stored section is created for the DISCONNECT statement. DISCONNECT cannot be used with the PREPARE or EXECUTE IMMEDIATE statements.
- An active transaction is not required to execute a DISCONNECT statement. An

automatic transaction will *not* be started when executing a DISCONNECT statement.

- Any connection name associated with a disconnected connection can be reused.
- A DISCONNECT CURRENT statement is equivalent to a RELEASE statement.
- Following a RELEASE or DISCONNECT CURRENT command, there is no current connection until a SET CONNECTION command is used to set the current connection to another existing connection, or a new connection is established by using the CONNECT, START DBE, START DBE NEW, or START DBE NEW LOG commands.

Authorization

You do not need authorization to use the DISCONNECT statement.

Example

Connect three times to PartsDBE and once to SalesDBE:

```
CONNECT TO :PartsDBE AS 'Parts1'
CONNECT TO :PartsDBE AS 'Parts2'
CONNECT TO :PartsDBE AS 'Parts3'
CONNECT TO :SalesDBE AS 'Sales1'
.
.
.
```

Terminate the connection associated with connection name Parts1:

```
DISCONNECT 'Parts1'
```

Terminate the connection associated with the most recently connected DBEnvironment (the current connection). Following the execution of this statement, SalesDBE is no longer connected, and no current connection exists:

```
DISCONNECT CURRENT
```

Note that another DISCONNECT CURRENT statement at this point would generate an error. Also any SQL statement that operates on a transaction will fail since there is no current connection and therefore no current transaction.

Set the current connection to Parts3:

```
SET CONNECTION 'Parts3'
```

Terminate the connection associated with the most recently connected DBE (the current connection). Following the execution of this statement, the Parts3 connection to PartsDBE no longer exists, and no current connection exists:

```
DISCONNECT CURRENT
```

Terminate all established connections. Following this statement, the Parts2 connection to PartsDBE no longer exists:

```
DISCONNECT ALL
```

DROP DBEFILE

The DROP DBEFILE statement removes the row describing a DBEFile from the SYSTEM.DBEFile.

Scope

ISQL or Application Programs

SQL Syntax

DROP DBEFILE *DBEFileName*

Parameters

DBEFileName is the name of the DBEFile to be dropped.

Description

- Before dropping a DBEFile previously associated with a DBEFileSet via an ADD DBEFILE statement, you must use the DROP INDEX and DROP TABLE statements to empty the DBEFile, then use the REMOVE DBEFILE statement to remove the DBEFile from the DBEFileSet.
- Although information for the dropped DBEFile is removed from the SYSTEM.DBEFile, the file is not removed until the transaction is committed.

Authorization

You must have DBA authority to use this statement.

Example

```
CREATE DBEFILE ThisDBEFile WITH PAGES = 4,
               NAME = 'ThisFile', TYPE = TABLE

CREATE DBEFILESET Miscellaneous

ADD DBEFILE ThisDBEFile TO DBEFILESET Miscellaneous
```

The DBEFile is used to store rows of a new table. When the table needs an index, one is created.

```
CREATE DBEFILE ThatDBEFile WITH PAGES = 4,
               NAME = 'ThatFile', TYPE = INDEX

ADD DBEFILE ThatDBEFile to DBEFILESET Miscellaneous
```

When the index is subsequently dropped, its file space can be assigned to another DBEFileSet.

```
REMOVE DBEFILE ThatDBEFile FROM DBEFILESET Miscellaneous

ADD DBEFILE ThatDBEFile TO DBEFILESET SYSTEM

ALTER DBEFILE ThisDBEFile SET TYPE = MIXED
```

Now you can use this DBEFile to store an index later if you need one. All rows are later deleted from the table, so you can reclaim file space.

```
REMOVE DBEFILE ThisDBEFile FROM DBEFILESET Miscellaneous

DROP DBEFILE ThisDBEFile
```

The DBEFileSet definition can now be dropped.

```
DROP DBEFILESET Miscellaneous
```

DROP DBEFILESET

The DROP DBEFILESET statement removes the definition of a DBEFileSet from the system catalog.

Scope

ISQL or Application Programs

SQL Syntax

DROP DBEFILESET *DBEFileSetName*

Parameters

DBEFileSetName is the name of the DBEFileSet to be dropped.

Description

- Before you can drop a DBEFileSet, you must use the REMOVE DBEFile statement to remove any DBEFiles associated with the DBEFileSet.
- You cannot DROP a default DBEFileSet. You must first change the default to some other DBEFileSet.
- DROP also removes any authorities associated with the DBEFileSet. (Refer to syntax for the GRANT statement with the ON DBEFILESET clause.)

Authorization

You must have DBA authority to use this statement.

Example

```
CREATE DBEFILE ThisDBEFile WITH PAGES = 4,
               NAME = 'ThisFile', TYPE = TABLE

CREATE DBEFILESET Miscellaneous

ADD DBEFILE ThisDBEFile TO DBEFILESET Miscellaneous
```

The DBEFile is used to store rows of a new table. When the table needs an index, one is created as follows:

```
CREATE DBEFILE ThatDBEFile WITH PAGES = 4,
               NAME = 'ThatFile', TYPE = INDEX

ADD DBEFILE ThatDBEFile to DBEFILESET Miscellaneous
```

When the index is subsequently dropped, its file space can be assigned to another DBEFileSet.

```
REMOVE DBEFILE ThatDBEFile FROM DBEFILESET Miscellaneous

CREATE DBEFILESET OtherDBEFileSet

ADD DBEFILE ThatDBEFile TO DBEFILESET OtherDBEFileSet
```

The following statement allows you to use ThisDBEFile to store an index later, if you need one.

```
ALTER DBEFILE ThisDBEFile SET TYPE = MIXED
```

If, later, all rows are deleted from the table, you can reclaim file space.

```
REMOVE DBEFILE ThisDBEFile FROM DBEFILESET Miscellaneous

DROP DBEFILE ThisDBEFile
```

If it is *not* a default DBEFileSet, you can now drop its definition.

```
DROP DBEFILESET Miscellaneous
```

DROP GROUP

The DROP GROUP statement removes the definition of an authorization group from the system catalog.

Scope

ISQL or Application Programs

SQL Syntax

DROP GROUP *GroupName*

Parameters

GroupName identifies the authorization group to be dropped.

Description

- You cannot drop an authorization group if it owns any tables, views, modules, or authorization groups.
- You cannot drop a group if it has access to a DBA or REFERENCES privilege which was used to validate the creation of a currently existing foreign key in a table owned by the group or one of its members.
- You can drop a group even if it still has members.

Authorization

You can use this statement if you have OWNER authority for the authorization group or if you have DBA authority.

Example

```
CREATE GROUP Warehse

GRANT CONNECT TO Warehse

GRANT SELECT,
      UPDATE (BinNumber,QtyOnHand,LastCountDate)
   ON PurchDB.Inventory
   TO Warehse
```

These two users will be able to start DBE sessions, retrieve data from table PurchDB.Inventory, and update three columns in the table.

```
ADD Clem, George TO GROUP Warehse
```

Clem no longer has any of the authorities associated with group Warehse.

```
REMOVE Clem FROM GROUP Warehse
```

Because this group does not own any database objects, it can be deleted. George no longer has any of the authorities once associated with the group.

```
DROP GROUP Warehse
```

DROP INDEX

The DROP INDEX statement deletes the specified index.

Scope

ISQL or Application Programs

SQL Syntax

DROP INDEX [*Owner.*]*IndexName*][**FROM**][*Owner.*]*TableName*]

Parameters

[*Owner.*]*IndexName* is the name of the index to be dropped. It may include the name of the owner of the table which has the index.

[*Owner.*]*TableName* is the name of the table upon which the index was created.

Description

- If a table name is not specified, the index name must be unique for the specified or implicit owner. The implicit owner, in the absence of a specified table or owner, is the current DBEUserID.
- Only indexes appearing in the system view SYSTEM.INDEX may be removed with this statement. Hash table structures cannot be dropped by using this statement; the hash structure can only be removed by dropping the table with the DROP TABLE statement. Neither unique constraint indexes nor referential constraint virtual indexes can be dropped with this statement. Constraints can only be removed through the ALTER TABLE DROP CONSTRAINT statement or the DROP TABLE statement.
- Issuing the DROP INDEX statement can invalidate stored sections. Refer to the *ALLBASE/SQL Database Administration Guide* for additional information on section validation.
- If no index owner is specified and no table is specified, the default owner is the current DBEUserID.
- If no index owner is specified and a table is specified, the default rule owner is the table owner.
- If a table is specified and no owner is specified for it, the default table owner is the current DBEUserID.
- The table and index owners must be the same.

Authorization

You can issue this statement if you have INDEX or OWNER authority for the table or if you have DBA authority.

Example

```
DROP INDEX PartsOrderedIndex
      FROM PurchDB.OrderItems
```

Alternatively:

```
DROP INDEX PurchDB.PartsOrderedIndex
```

If you discover that an index does not improve the speed of data access, you can delete it. If applications change, you can redefine the index.

DROP MODULE

The DROP MODULE statement deletes any sections associated with preprocessed SQL statements from the ALLBASE/SQL system catalog.

Scope

ISQL or Application Programs

SQL Syntax

```
DROP MODULE [Owner.]ModuleName [PRESERVE]
```

Parameters

[*Owner.*]*ModuleName* identifies the module to be dropped.

PRESERVE causes ALLBASE/SQL to retain the module's authorization records. If you preprocess a new version of an application program, you do not have to repeat the process of granting RUN authority to everyone who will run the program. If you do not specify the PRESERVE option, all authority that had been granted for the module is revoked.

Description

- When an application program is preprocessed, information needed for efficient database access is stored as a module in the system catalog. The system catalog also contains a record of the module's owner and any GRANT statements that have been issued to authorize other users to run the program. The DROP MODULE statement deletes all this information unless the PRESERVE option is specified; if the PRESERVE option is specified, the DROP MODULE statement deletes all but the RUN authorization information.
- A module name can also identify a set of one or more dynamically preprocessed statements created in the interactive environment with the PREPARE statement. The DROP MODULE statement can be used to drop such a set of dynamically preprocessed statements, and optionally any associated authorization data, in addition to the uses of the DROP MODULE statement described above.
- The DROP MODULE statement can invalidate stored sections. Refer to the *ALLBASE/SQL Database Administration Guide* for additional information on section validation.

Authorization

You can use the DROP MODULE statement if you have OWNER authority for the module or if you have DBA authority.

Examples

1. Dropping preprocessed application programs

   A module for the application program MyProg is created and stored in the system catalog by one of the preprocessors.

   ```
   GRANT RUN ON MyProg TO PUBLIC

   DROP MODULE MyProg PRESERVE
   ```

   Authorization information for MyProg is retained, but the module is deleted from the system catalog. You can re-preprocess MyProg and not have to redefine its authorization.

2. Dropping interactively prepared modules

   Two sections for a module named Statistics are stored in the system catalog.

   ```
   PREPARE Statistics (1)
      FROM 'UPDATE STATISTICS FOR TABLE PurchDB.Orders'

   PREPARE Statistics (2)
      FROM 'UPDATE STATISTICS FOR TABLE PurchDB.OrderItems'
   ```

   This only executes Statistics(1). The statistics for table PurchDB.Orders are updated:

   ```
   EXECUTE Statistics
   ```

   The statistics for table PurchDB.OrderItems are updated:

   ```
   EXECUTE Statistics(2)
   ```

   Both sections of the module are deleted.

   ```
   DROP MODULE Statistics
   ```

DROP PARTITION

The DROP PARTITION statement removes the definition of a partition for audit logging purposes.

Scope

ISQL or Application Programs

SQL Syntax

```
DROP PARTITION PartitionName
```

Parameters

PartitionName specifies the name of the partition to be dropped.

Description

- The partition being dropped must not have any tables associated with it. Use the ALTER TABLE SET PARTITION statement to remove any tables associated with it before dropping the partition.
- The DEFAULT partition cannot be dropped. It can be reset to NONE or to another partition with the START DBE NEWLOG statement.

Authorization

You must have DBA authority to use this statement.

Example

A partition can be dropped after all tables in it are assigned to PARTITION NONE.

```
CREATE PARTITION PartsPart WITH ID = 10;

ALTER TABLE PurchDB.Parts SET PARTITION PartsPart;

ALTER TABLE PurchDB.Parts SET PARTITION NONE;

DROP PARTITION PartsPart;
```

DROP PROCEDURE

The DROP PROCEDURE statement deletes the specified procedure.

Scope

ISQL or Application Programs

SQL Syntax

DROP PROCEDURE [*Owner.*]*ProcedureName* [**PRESERVE**]

Parameters

[*Owner.*]*ProcedureName* specifies the name of the procedure that is to be dropped.

PRESERVE specifies that EXECUTE authorities associated with the procedure should be retained in the system catalog.

Description

- If you do not specify PRESERVE, the EXECUTE authorities associated with the procedure are removed.
- If a rule attempts to execute a procedure that has been dropped, the rule will fail and all the effects of the statement that fired the rule are undone.
- The DROP PROCEDURE statement does *not* drop rules that invoke the procedure. All rules invoking the procedure are preserved.
- The DROP PROCEDURE statement will invalidate stored sections that depend on invoking the procedure from a rule. The loss of the procedure will be reported as an error when there is an attempt to revalidate these sections.

Authorization

You must be the owner of the procedure or have DBA authority to use the DROP PROCEDURE statement.

Example

```
DROP PROCEDURE Process12 PRESERVE
```

DROP RULE

The DROP RULE statement deletes the specified rule.

Scope

ISQL or Application Programs

SQL Syntax

DROP RULE [*Owner.*]*RuleName* [**FROM TABLE** [*Owner.*]*TableName*]

Parameters

[*Owner.*]*RuleName* identifies the rule to be dropped.

[*Owner.*]*TableName* identifies the table the rule is defined on.

Description

- If a *TableName* is specified, the rule must exist on that table or an error will be returned.
- If no *TableName* is specified, the rule is located and dropped. Since rule names are unique per owner, not per table, there is no ambiguity in references to *RuleName*.
- The DROP RULE statement invalidates stored sections that have dependencies defined upon the table the rule is defined on. This will permit the rule to be removed when the sections are revalidated.
- The procedure a rule invokes will not be affected by the removal of that rule.
- If no rule owner is specified and no table is specified, the default owner is the current DBEUserID.
- If no rule owner is specified and a table is specified, the default rule owner is the table owner.
- If a table is specified and no owner is specified for it, the default table owner is the current DBEUserID.
- The table and rule owners must be the same.

Authorization

You can issue this statement if you have OWNER authority for the rule or DBA authority.

Example

```
DROP RULE PurchDB.InsertReport
DROP RULE PurchDB.DeleteReport
DROP RULE PurchDB.UpdateReport
```

DROP TABLE

The DROP TABLE statement deletes the specified table, including any hash structure or constraints associated with it, all indexes, views, and rules defined on the table, and all authorizations granted on the table.

Scope

ISQL or Application Programs

SQL Syntax

```
DROP TABLE [Owner.]Tablename
```

Parameters

[*Owner.*]*TableName* identifies the table to be dropped.

Description

- The DROP TABLE statement may invalidate stored sections. Refer to the *ALLBASE/SQL Database Administration Guide* for additional information on section validation.
- You cannot drop a table which has a primary or unique constraint referenced by a foreign key in another table. (You can, however, if the only foreign keys are within the same table.)
- Any authorities used to authorize a foreign key on the table are released when the table is dropped.

Authorization

You can issue this statement if you have OWNER authority for the table or if you have DBA authority.

Example

This table is private by default.

```
CREATE TABLE VendorPerf
            (OrderNumber  INTEGER  NOT NULL,
            ActualDelivDay  SMALLINT,
            ActualDelivMonth  SMALLINT,
            ActualDelivYear  SMALLINT,
            ActualDelivQty  SMALLINT
            Remarks  VARCHAR(60) )
         IN Miscellaneous
```

```
CREATE UNIQUE INDEX VendorPerfIndex
                 ON VendorPerf
                    (OrderNumber)

CREATE VIEW VendorPerfView
            (OrderNumber,
            ActualDelivQty,
            Remarks)
  AS SELECT OrderNumber,
            ActualDelivQty,
            Remarks
       FROM VendorPerf
```

Only the table creator and members of authorization group Warehse can update table VendorPerf.

```
GRANT UPDATE ON VendorPerf TO Warehse
```

The table, the index, and the view are all deleted; and the grant is revoked.

```
DROP TABLE VendorPerf
```

DROP TEMPSPACE

The DROP TEMPSPACE statement removes the definition of a temporary storage space (TempSpace) from the system catalog.

Scope

ISQL or Application Programs

SQL Syntax

DROP TEMPSPACE *TempSpaceName*

Parameters

TempSpaceName is the name of the TempSpace to be dropped.

Description

- If a TempSpace is dropped while temporary files currently exist under the path name it specifies, those files remain until the sort using them completes. However, no further temporary files are created in that TempSpace.
- If a TempSpace is being used by another user when the DROP TEMPSPACE statement is issued, then the DROP statement is blocked until the TempSpace usage is finished.

Authorization

You must have DBA authority to use this statement.

Example

TempSpace temporary files are created in the /sort/PurchDB directory when SQL statements require sorting.

```
CREATE TEMPSPACE ThisTempSpace WITH MAXFILEPAGES = 360,
                 LOCATION = '/sort/PurchDB'
DROP TEMPSPACE ThisTempSpace
```

TempSpace temporary files are no longer available in the /sort/PurchDB, directory but can be allocated under /tmp as needed.

DROP VIEW

The DROP VIEW statement deletes the definition of the specified view from the system catalog, all authorization granted on the view, and any view that references the dropped view.

Scope

ISQL or Application Programs

SQL Syntax

```
DROP VIEW [Owner.]ViewName
```

Parameters

[*Owner*.]*ViewName* identifies the view to be dropped.

Description

- This statement does not affect the base tables on which the views were defined.
- The DROP VIEW statement can invalidate stored sections. Refer to the *ALLBASE/SQL Database Administration Guide* for additional information on stored section validation.
- You cannot use this statement on system views.
- If the view was defined with a WITH CHECK OPTION constraint, the view check constraint is also deleted.

Authorization

You can use the DROP VIEW statement if you have OWNER authority for the view or if you have DBA authority.

Example

The view is dropped. Any grants referencing the view are automatically revoked.

```
DROP VIEW ReorderParts
```

11 SQL Statements E - R

Chapter 10, 11 and 12 describe all the SQL statements in alphabetical order, giving syntax, parameters, descriptions, authorization requirements, and examples for each statement. Examples often consist of groups of statements so you can see how each statement is related to other statements functionally.

ENABLE AUDIT LOGGING

The ENABLE AUDIT LOGGING statement restarts audit logging for the DBEnvironment after a DISABLE AUDIT LOGGING has been performed.

Scope

ISQL or Application Programs

SQL Syntax

```
ENABLE AUDIT LOGGING
```

Description

- This statement reenables audit logging in the current session only.
- This statement and DISABLE AUDIT LOGGING are *not* used to turn on and off the AUDIT LOG option specified for processing of all sessions in the DBEnvironment. These statements affect your current session only. (The statements that affect *all* processing in the DBEnvironment are the START DBE NEW and START DBE NEWLOG statements.)
- This statement is not affected by transaction management statements and remains in effect until a DISABLE AUDIT LOGGING statement is issued.

Authorization

This statement requires DBA authority.

Example

Perform an initial load of a table without audit logging.

```
DISABLE AUDIT LOGGING;

LOAD FROM INTERNAL PartsData TO PurchDB.Parts;
```

```
COMMIT WORK;
```

Reenable audit logging and continue.

```
ENABLE AUDIT LOGGING;
```

ENABLE RULES

The ENABLE RULES statement turns rule checking on for the current DBEnvironment session. DBAs use it to tune the DBEnvironment and test the operation of rules.

Scope

ISQL or Application Programs

SQL Syntax

```
ENABLE RULES
```

Description

- ENABLE RULES returns the DBEnvironment session to its default behavior of firing all applicable rules.
- The statement only affects the current SID (session id). Other users are not affected.
- The ENABLE RULES statement is not cumulative; issuing additional ENABLE RULES statements will have no effect, and a warning will be issued to this effect.
- Rules are not fired retroactively when the ENABLE RULES statement is issued after the DISABLE RULES statement has been issued. That is, if a DISABLE RULES statement is issued, rules that would otherwise be applicable will not fire. Then, when a subsequent ENABLE RULES is issued, applicable rules will fire again, but only for subsequent data manipulation statements, not for rows processed while rule firing was disabled.
- COMMIT WORK and ROLLBACK WORK statements have no effect on whether rules are enabled or disabled.

Authorization

You must have DBA authority.

Example

The DBA turns off rule invocation.

```
DISABLE RULES
```

The DBA performs operations without rule firing.

.
.
.

The DBA turns on rule invocation.

```
ENABLE RULES
```

Normal firing of rules resumes.

END DECLARE SECTION

The END DECLARE SECTION preprocessor directive indicates the end of the host variable declaration section in an application program.

Scope

Application Programs Only

SQL Syntax

```
END DECLAARE SECTION
```

Description

- This directive cannot be used interactively.
- Use this directive in conjunction with the BEGIN DECLARE SECTION directive.

Authorization

You do not need authorization to use the END DECLARE SECTION statement.

Example

```
BEGIN DECLARE SECTION
```

Define host variables here, including indicator variables, if any.

```
END DECLARE SECTION
```

EXECUTE

The EXECUTE statement causes ALLBASE/SQL to execute a statement that has been dynamically preprocessed by means of the PREPARE statement.

Scope

ISQL or Application Programs

SQL Syntax

```
EXECUTE {StatementName
         [Owner.]ModuleName[(SectionNumber) ]}
[USING {[SQL]DESCRIPTOR{[INPUT]{SQLDA
                                AreaName1}
                          [AND OUTPUT{SQLDA
                                      AreaName2}]
                          OUTPUT{SQLDA
                                 AreaName}}
        [INPUT]HostVariableSpecification1
        [AND OUTPUT HostVariableSpecification2]
        OUTPUT HostVariableSpecification
        :Buffer[,:StartIndex[,:NumberOfRows]]      }]
```

Parameters

StatementName identifies a dynamically preprocessed statement to be executed in an application program. The *StatementName* corresponds to one specified in a previous PREPARE statement. This form of the EXECUTE statement cannot be used interactively.

[*Owner.*]*ModuleName* [(*SectionNumber*)] identifies a dynamically preprocessed statement to be executed interactively. The preprocessed statement cannot be a SELECT statement. This form of the EXECUTE statement cannot be used in an application program. If the section number is omitted, section number one is assumed. You can omit the verb EXECUTE interactively.

USING allows dynamic parameter substitution in a prepared statement in an application program.

[SQL]DESCRIPTOR indicates that a data structure of sqlda_type is used to pass dynamic parameter information between the application and ALLBASE/SQL.

SQLDA specifies that a data structure of sqlda_type named **sqlda** is used to pass dynamic parameter information between the application and ALLBASE/SQL.

AreaName specifies the user defined name of a data structure of type sqlda_type that is used to pass dynamic parameter information between the application and ALLBASE/SQL.

HostVariableSpecification specifies host variable(s) that hold dynamic parameter values at run time. The syntax of *HostVariableSpecification* is

presented separately below.

`INPUT` is the default for any EXECUTE statement and can be specified, as required, for any type of prepared statement containing input dynamic parameters.

`OUTPUT` is only allowed when the prepared statement is an EXECUTE PROCEDURE statement. It can be used when the statement contains output dynamic parameters.

`INPUT AND OUTPUT` is only allowed when the prepared statement is an EXECUTE PROCEDURE statement. It can be used when the statement contains both input and output dynamic parameters.

`Buffer` is a host variable array structure containing rows that are the input for a BULK INSERT statement. This structure contains fields for each column to be inserted and indicator variables for columns that can contain null values. Whenever a column can contain nulls, an indicator variable must be included in the array definition immediately after the definition of that column. This indicator variable is an integer that can have the following values:

> = 0	the value is not NULL
< 0	the value is NULL

NOTE To be consistent with standard SQL and to support portability of code, it is strongly recommended that you use a –1 to indicate a null value. However, ALLBASE/SQL interprets all negative indicator variable values to mean a null value.

`StartIndex` is a host variable whose value specifies the array subscript denoting where the first row to be inserted is stored; default is the first element of the array.

`NumberOfRows` is a host variable whose value specifies the number of rows to insert; default is to insert from the starting index to the end of the array.

SQL Syntax — HostVariableSpecification

```
:HostVariableName [[INDICATOR]:IndicatorVariable] [,...]
```

Parameters — HostVariableSpecification

`HostVariableName` specifies a host variable name that at run time contains the data value that is assigned to a dynamic parameter defined in a prepared statement.

Host variables must be specified in the same order as the dynamic parameters in the prepared statement they represent. There must be a one to one correspondence between host variable names and the dynamic parameters in the prepared statement. A maximum of 1024 host variable names can be specified.

IndicatorVariable names an indicator variable, whose value determines whether the associated host variable contains a NULL value:

> = 0	the value is not NULL
< 0	the value is NULL

Description

- There must be a one to one mapping of the input and/or output parameters in a prepared statement and its associated EXECUTE statement.
- INPUT is the default for any EXECUTE statement and can be specified, as required, for any type of prepared statement.
- The OUTPUT clause is only allowed when the prepared statement is an EXECUTE PROCEDURE statement containing output dynamic parameters.
- An INPUT AND OUTPUT clause is only allowed when the prepared statement is an EXECUTE PROCEDURE statement containing both input and output dynamic parameters.
- If *StatementName* is an EXECUTE PROCEDURE statement without any input and output dynamic parameters, you can execute the procedure by issuing EXECUTE *StatementName*.
- If *StatementName* is an EXECUTE PROCEDURE statement with either input or output dynamic parameters, you can use the EXECUTE USING statement with INPUT (default) or OUTPUT option to execute the dynamically preprocessed statement.
- If *StatementName* is an EXECUTE PROCEDURE statement with both input and output dynamic parameters, you can use the EXECUTE USING statement with the INPUT AND OUTPUT option to execute the dynamically preprocessed statement.
- Use the USING clause for either an SQLDA DESCRIPTOR or a *HostVariableSpecification* for input and/or output dynamic parameter substitution in a prepared statement.
- The :*Buffer* [,:*StartIndex* [, :*NumberOfRows* option is only used in association with a BULK INSERT statement.
- If *StatementName* is an EXECUTE PROCEDURE statement, and there are multiple row result sets from the procedure, you must use the procedure cursor method to retrieve result sets. A warning is returned if a procedure cursor is not used in this case; the return status and output parameters are returned as usual.

Authorization

In an application program, the EXECUTE statement does not require any special authorization. The user running the program must have whatever authorization is required by the dynamically preprocessed statement being executed.

To use the EXECUTE statement in the interactive environment, you must have RUN or OWNER authority for the dynamically preprocessed statement or have DBA authority. In addition, the owner of the dynamically preprocessed statement must have whatever

authorization the dynamically preprocessed statement itself requires.

Examples

1. Interactive execution

```
isql=> PREPARE Statistics(1)
> FROM 'UPDATE STATISTICS FOR TABLE PurchDB.Orders'

isql=> PREPARE Statistics(2)
>  FROM 'UPDATE STATISTICS FOR TABLE PurchDB.OrderItems'

Two sections for module Statistics are stored in the system catalog.

isql=> EXECUTE Statistics(1)

The statistics for table PurchDB.Orders are updated.

isql=> EXECUTE Statistics(2)

The statistics for table PurchDB.OrderItems are updated.

isql=> DROP MODULE Statistics

Both sections of the module are deleted.
```

2. Programmatic execution

   If you know that the statement to be dynamically preprocessed is not a SELECT statement and does not contain dynamic parameters, you can prepare it and execute it in one step, as follows:

```
EXECUTE IMMEDIATE :Dynam1
```

   You can prepare and execute the statement in separate operations. For example, if you don't know the format of a statement, you could do the following:

```
PREPARE Dynamic1 FROM :Dynam1

The statement stored in :Dynam1 is dynamically preprocessed.

DESCRIBE Dynamic1 INTO SqldaOut
```

   If Dynamic1 is not a SELECT statement, the Sqld field of the Sqlda data structure is 0. If you know there are no dynamic parameters in the prepared statement, use the EXECUTE statement to execute the dynamically preprocessed statement.

   If it is possible that dynamic parameters are in the prepared statement, you must describe the statement for input:

```
DESCRIBE INPUT Dynamic1 USING SQL DESCRIPTOR SqldaIn
```

If the prepared statement could be an EXECUTE PROCEDURE statement (sqld = zero on DESCRIBE OUTPUT) with dynamic output parameters, you must describe it for output:

```
DESCRIBE OUTPUT Dynamic1 USING SQL DESCRIPTOR SqldaOut
```

If only dynamic input parameters are present, the appropriate data buffer or host variables must be loaded with the values of any dynamic parameters. Then if the statement is not a query, it can be executed, as in this example using a data buffer:

```
EXECUTE Dynamic1 USING SQL DESCRIPTOR SqldaIn
```

However, if the prepared statement is an EXECUTE PROCEDURE statement with multiple row result sets (sqlmproc = non-zero) and dynamic input and output parameters execute it as follows:

```
EXECUTE Dynamic1 USING SQL INPUT DESCRIPTOR SqldaIn
                       and OUTPUT DESCRIPTOR SqldaOut
```

EXECUTE IMMEDIATE

The EXECUTE IMMEDIATE statement dynamically prepares and executes an SQL statement.

Scope

ISQL or Application Programs

SQL Syntax

```
EXECUTE IMMEDIATE {'String'
                   :HostVariable}
```

Parameters

String is the ALLBASE/SQL statement to be executed.

HostVariable identifies a character-string host variable containing the ALLBASE/SQL statement to be executed.

Description

- When used interactively, a host variable cannot be specified.
- The SQL statement cannot contain host variables nor dynamic parameters.
- You *cannot* use the EXECUTE IMMEDIATE statement for any of the following statements:

BEGIN DECLARE SECTION	EXECUTE	SELECT
CLOSE	EXECUTE IMMEDIATE	SQLEXPLAIN
DECLARE CURSOR	FETCH	UPDATE WHERE CURRENT
DELETE WHERE CURRENT	INCLUDE	WHENEVER
DESCRIBE	OPEN	
END DECLARE SECTION	PREPARE	

Authorization

You can use EXECUTE IMMEDIATE if your authorization permits you to issue the statement to be executed.

Example

If you know that the statement to be dynamically preprocessed is neither a SELECT statement nor an EXECUTE PROCEDURE statement with results, and has neither input nor output dynamic parameters, you can prepare it and execute it in one step.

```
EXECUTE IMMEDIATE :Dynam1
```

In other instances, it is more appropriate to prepare and execute the statement in separate operations.

EXECUTE PROCEDURE

The EXECUTE PROCEDURE statement invokes a procedure.

Scope

ISQL or Application Programs

Syntax

```
EXECUTE PROCEDURE [:ReturnStatusVariable = ][Owner.]ProcedureName
[ ( [ActualParameter ][, [ActualParameter ]][...]) ]
```

Parameters

ReturnStatusVariable is an integer host variable, or, for a prepared EXECUTE PROCEDURE statement, a dynamic parameter, that receives the return status from the procedure. *ReturnStatusVariable* can only be used when invoking a procedure from an application program, and it is always an output variable.

[*Owner*.]*ProcedureName* specifies the owner and the name of the procedure to execute. If an owner name is not specified, the owner is assumed to be the current DBEUserID.

ActualParameter specifies a parameter value that is passed into and/or out of the procedure. The syntax of *ActualParameter* is presented separately below.

SQL Syntax—ActualParameter

```
[ParameterName = ]ParameterValue [OUTPUT[ONLY]]
```

Parameters—ParameterDeclaration

ParameterName is the parameter name.

ParameterValue a value that is passed into and/or out of the procedure.

For an input only parameter, the value can be any expression that does not include any aggregate function, add_months function, LONG column function, TID function, local variable, procedure parameter, or built-in variable. Column values are allowed only when the EXECUTE PROCEDURE statement is defined in a rule.

For an OUTPUT or OUTPUT ONLY parameter, the value must be a single host variable, or in a prepared EXECUTE PROCEDURE statement, a single dynamic parameter.

You can omit a parameter in calling the procedure by using a comma by itself, which is equivalent to specifying a value of NULL or the default (if

one was defined when the procedure was created). However, if a *ParameterName* is specified, use of a comma by itself is disallowed.

OUTPUT — specifies that the caller wishes to retrieve the output value of the parameter. OUTPUT must also have been specified for the corresponding parameter in the CREATE PROCEDURE statement.

If OUTPUT is *not* specified, no output value is returned to the caller.

ONLY — specifies that the caller wishes to retrieve the output value of the parameter and will not provide an input value. You must also have specified ONLY for the corresponding parameter in the CREATE PROCEDURE statement. ONLY should be used, when applicable, to avoid unnecessary initialization of procedure parameters.

Description

- You cannot execute a procedure from within another procedure.
- If OUTPUT ONLY is not specified, a parameter that is not given a value in the EXECUTE PROCEDURE statement is assigned its default value if one was specified, or otherwise NULL if the parameter was not declared NOT NULL.

  If OUTPUT ONLY is not specified, no value is provided for a parameter, a default is not specified, and NOT NULL is specified, an error is returned and the procedure is not executed.
- If a procedure terminates abnormally (an error occurs in evaluating the condition in an IF or WHILE statement, or in evaluating the expression in a parameter or variable assignment), any cursors opened by the procedure (including KEEP cursors) are closed. Otherwise, except in a procedure invoked by a rule, any cursor opened by the procedure, and left open when the procedure terminates, remains open and may therefore be accessed when the procedure is executed again.
- If OUTPUT has been specified for a parameter in both the CREATE PROCEDURE and EXECUTE PROCEDURE statements, any changes made to the parameter value within the procedure are returned to the calling application. The actual parameter for an output parameter can be a host variable or a dynamic parameter.
- If you execute a procedure that returns multiple row result sets (contains one or more SELECT statements with no INTO clause) without using a procedure cursor, a warning is returned to the application, no result set data is returned, and any return status and output parameters are returned as usual.
- You can execute procedures in ISQL, through application programs, or via rules. Further information on executing a procedure from an application is found in the *ALLBASE/SQL Advanced Application Programming Guide*.. For the execution of procedures through rules, refer to the CREATE RULE statement.
- In ISQL, you cannot specify OUTPUT for a parameter. Although return status cannot be specified in the EXECUTE PROCEDURE statement, ISQL does report the return status. Also, within ISQL, actual parameter values cannot include host variables.
- If you attempt to execute a procedure that contains invalid sections, ALLBASE/SQL silently revalidates the sections. You can also use the VALIDATE statement to revalidate

invalid sections in procedures.

- You can PREPARE and EXECUTE an EXECUTE PROCEDURE statement containing dynamic parameters.

  You can use EXECUTE PROCEDURE inside an EXECUTE IMMEDIATE statement, provided the EXECUTE PROCEDURE statement includes neither dynamic parameters nor host variables.

- If you do not specify OUTPUT for a parameter declared as OUTPUT in the CREATE PROCEDURE statement, no value is returned.
- You cannot specify OUTPUT for a parameter not declared as OUTPUT in the CREATE PROCEDURE statement.
- OUTPUT ONLY must be specified for any parameter declared as OUTPUT ONLY in the CREATE PROCEDURE statement if an actual parameter is provided. Use of OUTPUT ONLY improves performance, since no time is spent initializing the parameter to the input value, default value, or null.
- Within a procedure, a single row SELECT statement (one having an INTO clause) that returns multiple rows will assign the first row to output parameters or local variables, and a warning is issued. In an application, this case would generate an error.

Authorization

You must have OWNER or EXECUTE authority for the procedure or DBA authority to use this statement.

Examples

1. From an application program:

```
EXECUTE PROCEDURE :Status = Process12(:PartName, :Quantity,
                                      :SalesPrice OUTPUT ONLY)
```

2. Within ISQL:

```
isql=> execute procedure Process12('Widget',150);
```

FETCH

The FETCH statement advances the position of an opened cursor to the next row of the active set and copies selected columns into the specified host variables or data buffer. The row to which the cursor points is called the current row.

Scope

Application Programs Only

SQL Syntax

```
[BULK] FETCH CursorName{INTO HostVariableSpecification
                        USING { [SQL] DESCRIPTOR {SQLDA
                                                  AreaName}
                                HostVariableSpecification   } }
```

Parameters

BULK — is specified in an application program to retrieve multiple rows with a single execution of the FETCH statement. After a BULK FETCH statement, the current row is the last row fetched.

BULK can be specified with the INTO clause (for a statically executed cursor), but *not* with the USING clause (for a dynamically executed cursor).

BULK is disallowed in a procedure.

CursorName — identifies a cursor. The cursor's active set, determined when the cursor was opened, and the cursor's current position in the active set determine the data to be returned by each successive FETCH statement.

INTO — The INTO clause defines where to place rows fetched for a statically preprocessed SELECT or EXECUTE PROCEDURE statement.

USING — The USING clause defines where to place rows fetched for a dynamically preprocessed SELECT or EXECUTE PROCEDURE statement, or for a statically preprocessed EXECUTE PROCEDURE statement with an unknown format.

HostVariableSpecification — identifies one or more host variables for holding and describing the row(s) in the active set.

When used with the INTO clause, the syntax of *HostVariableSpecification* depends on whether the BULK option is specified. If BULK is specified, *HostVariableSpecification* identifies an array that holds the rows fetched. If BULK is not specified, the host variable declaration identifies a list of individual host variables. The syntax of BULK and non-BULK variable declarations is shown in separate sections below.

The USING clause with a *HostVariableSpecification* allows non-BULK

variable declarations only.

DESCRIPTOR — The DESCRIPTOR identifier defines where to place rows selected in accord with a dynamically preprocessed SELECT or EXECUTE PROCEDURE statement that has been described by a DESCRIBE statement. For a select cursor, specify the same location (SQLDA, area name, or host variable) as you specified in the DESCRIBE statement. For a procedure cursor, specify the same location you specified in the ADVANCE statement or DESCRIBE RESULT statement (for a procedure created WITH RESULT).

SQLDA — specifies that a data structure of sqlda_type named **sqlda** is to be used to pass information about the prepared statement between the application and ALLBASE/SQL.

AreaName — specifies the user defined name of a data structure of sqlda_type that is to be used to pass information about the prepared statement between the application and ALLBASE/SQL.

SQL Syntax — BULK HostVariableSpecification

```
:Buffer [,:StartIndex [,:NumberOfRows]]
```

Parameters — BULK HostVariableSpecification

Buffer — is a host array structure that is to receive the output of the FETCH statement. This structure contains fields for each column in the active set and indicator variables for columns that contain null values. Whenever a column can contain nulls, an indicator variable must be included in the structure definition immediately after the definition of that column. The indicator variable can receive the following integer values after a FETCH:

0	meaning the column's value is not null
–1	meaning the column's value is null
>0	meaning the column's value is truncated (for CHAR, VARCHAR, BINARY, and VARBINARY columns)

StartIndex — is a host variable whose value specifies the array subscript denoting where the first row fetched should be stored; default is the first element of the array.

NumberOfRows — is a host variable whose value specifies the maximum number of rows to fetch; default is to fill from the starting index to the end of the array.

The total number of rows fetched is returned in the SQLERRD field of the SQLCA. You should check this area in case the number of rows returned is less than the maximum number of rows so that you don't process an incomplete result.

SQL Syntax — non-BULK HostVariableSpecification

```
{:HostVariable [[INDICATOR] :Indicator ] } [,...]
```

Parameters — non-BULK HostVariableSpecification

HostVariable identifies the host variable corresponding to one column in the row fetched.

Indicator names the indicator variable, an output host variable whose value depends on whether the host variable contains a null value. The following integer values are valid:

0	meaning the column's value is not null
–1	meaning the column's value is null
>0	meaning the column's value is truncated (for CHAR, VARCHAR, BINARY, and VARBINARY columns)

Description

- This statement cannot be used interactively.
- When using this statement to access LONG columns, the name of the file is returned in the appropriate field in the host variable declaration parameter, SQLDA, or area name parameter specified. If the output mode is specified with $, then each LONG column in each row accessed is stored in a file with a unique name.
- The use of a descriptor area implies a multiple row result set. You cannot use the BULK keyword if you employ the DESCRIPTOR identifier.
- For a procedure cursor that returns results of a single format, if the procedure was created with the WITH RESULT clause, since all result sets have the same format, it is not necessary to issue an ADVANCE statement to advance from one result set to the next. No end of result set condition is generated on a FETCH statement until all result sets have been fetched. When the end of a result set has been reached, the next FETCH statement issued causes procedure execution to continue either until the next result set is encountered and the first row of the next result set is returned or until procedure execution terminates.
- The USING clause is not allowed within a procedure.
- The BULK option is not allowed within a procedure.

Authorization

You do not need authorization to use the FETCH statement.

Examples

1. Static update

   A cursor for use in updating values in column QtyOnHand is declared and opened.

```
DECLARE NewQtyCursor CURSOR FOR
  SELECT PartNumber,QtyOnHand FROM PurchDB.Inventory
  FOR UPDATE OF QtyOnHand

OPEN NewQtyCursor
```

   Statements setting up a FETCH-UPDATE loop appear next.

```
FETCH NewQtyCursor INTO :Num :Numnul, :Qty :Qtynul
```

   Statements for displaying a row to and accepting a new QtyOnHand value from a user go here. The new value is stored in :NewQty.

```
UPDATE PurchDB.Inventory
  SET QtyOnHand = :NewQty
  WHERE CURRENT OF NewQtyCursor

CLOSE NewQtyCursor
```

2. Static bulk fetch

```
DECLARE ManyRows CURSOR FOR
  SELECT * FROM PurchDB.Inventory
```

   In some instances, using the BULK option is more efficient than advancing the cursor a row at a time through many rows, especially when you want to operate on the rows with non-ALLBASE/SQL statements.

```
OPEN ManyRows

BULK FETCH ManyRows INTO :Rows, :Start, :NumRow
```

   The query result is returned to an array called Rows.

3. Dynamic select cursor using an sqlda_type data structure

   Assume that host variable Dynam1 contains a SELECT statement. The statement stored in :Dynam1 is dynamically preprocessed.

```
PREPARE Dynamic1 FROM :Dynam1
```

   The DESCRIBE statement loads the specified sqlda_type data structure with the characteristics of the FETCH statement. See the ALLBASE/SQL for complete information regarding this data structure.

```
DESCRIBE Dynamic1 INTO SQLDA
```

   Define a cursor to be used to move through the query result row by row.

```
DECLARE Dynamic1Cursor CURSOR FOR Dynamic1
```

   Open the cursor to define rows of the active set.

```
OPEN Dynamic1Cursor
```

Fetch the selected data into the data buffer. Additional rows are fetched with each execution of the FETCH statement until all rows have been fetched. See the ALLBASE/SQL for more detailed examples.

```
FETCH Dynamic1Cursor USING DESCRIPTOR SQLDA
```

Close the cursor to free the active set.

```
CLOSE Dynamic1Cursor
```

4. Dynamic select cursor using host variables

   Assume that host variable Dynam1 contains a SELECT statement. The statement stored in :Dynam1 is dynamically preprocessed.

   ```
   PREPARE Dynamic1 FROM :Dynam1
   ```

   Define a cursor to be used to move through the query result row by row.

   ```
   DECLARE Dynamic1Cursor CURSOR FOR Dynamic1
   ```

   Open the cursor to define rows of the active set.

   ```
   OPEN Dynamic1Cursor
   ```

   Fetch the selected data into the specified host variables. With each execution of the FETCH statement one additional row is fetched until all rows have been fetched.

   ```
   FETCH Dynamic1Cursor USING :HostVariable1, :HostVariable2
   ```

   Close the cursor to free the active set.

   ```
   CLOSE Dynamic1Cursor
   ```

5. Refer to the *ALLBASE/SQL Advanced Application Programming Guide* for a pseudocode example of procedure cursor usage.

GENPLAN

The GENPLAN statement places the access plan generated by the optimizer for a SELECT, UPDATE, or DELETE statement into the pseudotable SYSTEM.PLAN. You can then view the access plan by issuing the following statement from within the same transaction:

```
isql=> SELECT * FROM SYSTEM.PLAN;
```

Scope

ISQL or Application Programs

SQL Syntax

```
GENPLAN [WITH (HostVariableDefinition)] FOR
{SQLStatement
 MODULE SECTION [Owner.]ModuleName(Section Number)
 PROCEDURE SECTION [Owner.]ProcedureName(Section Number)}
```

Parameters

WITH is used when simulating embedded statements taken from application programs. The WITH clause defines variables of a specified data type. The variables are used in the WHERE clause where an input host variable would appear if the *SQLStatement* were embedded in an application.

HostVariableDefinition designates a variable used to simulate a host variable that would appear in a statement in an application program. This clause is only allowed for an *SQLStatement*.

SQLStatement can be any valid SQL SELECT, UPDATE, or DELETE statement including complex statements containing UNION, OUTER JOIN, or nested subqueries.

[*Owner*].*ModuleName* (*Section Number*) identifies the module section whose access plan is to be generated. The owner name is the DBEUserID of the person who preprocessed the program or the owner name specified when the program was preprocessed. The *Module Name* is the name stored in the CATALOG.SECTION view.

[*Owner*.]*ProcedureName* (*Section Number*) identifies the procedure section whose access plan is to be generated. The owner name is the DBEUserID of the person who created the procedure or the owner name specified when the procedure was created. The *ProcedureName* is the name stored in the CATALOG.PROCEDURE view or CATALOG.SECTION view.

Description

- The GENPLAN statement can only be used in ISQL. It cannot be used in an application, in a static SQL statement, or in dynamic preprocessing.

NOTE GENPLAN checks only for syntax errors. It does not check for mismatched data types or other errors that may occur. In order to guarantee complete error checking, do not include a statement in GENPLAN unless it has previously run without errors.

- You should take the following steps when embedding a statement from an application in the GENPLAN statement:
  - In the GENPLAN WITH clause, define variable names and compatible SQL data types for each input host variable appearing in the application statement. Do not include indicator variables in the WITH clause for columns that allow nulls. Indicator variables are not used by GENPLAN.
  - Remove the INTO clause and its associated output host variables. Only input host variables are considered when generating the access plan.
- The following language specific tables show the SQL data type that must be placed in the WITH clause of the GENPLAN statement for each type of host variable, if an accurate access plan is to be generated. In some cases, the data type specified in the WITH clause of the GENPLAN statement is not the same data type which is compatible with the SQL data type of the column containing the data. The data type specified below must be used, regardless of the SQL column data type. This ensures that the plan displayed by the GENPLAN statement is the same as the plan chosen by the optimizer when the statement is preprocessed in an application.

Table 11-1. GENPLAN WITH Clause Data Types — COBOL

COBOL Host Variable Data Type Declaration	GENPLAN WITH Clause SQL Data Type
01 *DATA-NAME* PIC X.	CHAR
01 *DATA-NAME* PIC X(*n*).	CHAR(n)
01 *GROUP-NAME*. 49 *LENGTH-NAME* PIC S9(9) COMP. 49 *VALUE-NAME* PIC X(*n*).	VARCHAR(n)
01 *DATA-NAME* PIC S9(4) COMP.	SMALLINT
01 *DATA-NAME* PIC S9(9) COMP.	INTEGER
01 *DATA-NAME* PIC S9(p-s)V9(s) COMP-3.	DECIMAL(p,s)

Table 11-2. GENPLAN WITH Clause Data Types — Pascal

Pascal Host Variable Data Type Declaration	GENPLAN WITH Clause SQL Data Type
DataName : char;	CHAR
DataName : array [1..*n*] of char;	CHAR(n)
DataName : packed array [1..*n*] of char;	CHAR(n)
DataName : string[*n*];	VARCHAR(*n*)
DataName : smallint;	SMALLINT
DataName : integer;	INTEGER
DataName : longreal;	FLOAT
DataName : real;	REAL

Table 11-3. GENPLAN WITH Clause Data Types — FORTRAN

FORTRAN Host Variable Data Type Declaration	GENPLAN WITH Clause SQL Data Type
CHARACTER *DataName*	CHAR
CHARACTER*n *DataName*	CHAR(n)
INTEGER*2 *DataName*	SMALLINT
INTEGER *DataName*	INTEGER
REAL *DataName*	REAL
REAL*4 *DataName*	REAL
DOUBLE PRECISION *DataName*	FLOAT
REAL*8 *DataName*	FLOAT

Table 11-4. GENPLAN WITH Clause Data Types — C

C Host Variable Data Type Declaration	GENPLAN WITH Clause SQL Data Type
char *dataname*;	CHAR
char *dataname*[n+1];	VARCHAR(n)
short *dataname*;	SMALLINT
short int *dataname*;	SMALLINT
int *dataname*;	INTEGER
long int *dataname*;	INTEGER
long *dataname*;	INTEGER
float *dataname*;	REAL
double *dataname*;	FLOAT

NOTE It is your responsibility to ensure that for each simulated host variable defined in the GENPLAN statement WITH clause, you use the SQL data type shown in the tables. If you use an incorrect data type, GENPLAN will generate a plan. However, it may not be the plan the optimizer will choose when your application is preprocessed.

- For each individual session, SYSTEM.PLAN stores the result of only one GENPLAN at a time. If GENPLAN is issued twice in succession, the second plan will replace the first. The access plan generated by GENPLAN is removed from SYSTEM.PLAN as soon as a COMMIT WORK or ROLLBACK WORK statement is issued.
- GENPLAN can be applied to a type II insert query.
- The active SETOPT will be used for the statement of GENPLAN on an *SQLStatement* only. A currently active SETOPT is ignored if a GENPLAN statement is executed on a section.
- You can find the section number from the source file produced by the preprocessor after the application is processed.
- Use the following information to find the section number for a procedure statement:
  - A section exists for each SQL statement in a procedure except:
    - BEGIN WORK
    - ROLLBACK WORK
    - SAVEPOINT
    - OPEN cursor
    - CLOSE cursor
  - Procedure sections are numbered consecutively, starting with 1, from the start of the

procedure, with no regard to any branching or looping constructs in the procedure.

- Multiple sessions may issue the GENPLAN statement at the same time because each session has its own individual copy of SYSTEM.PLAN.
- See the section "Using GENPLAN to Display the Access Plan" in the "SQL Queries" chapter for information on how to interpret the plan.
- You cannot use GENPLAN with the SYSTEM or CATALOG views.

Authorization

To execute GENPLAN, you must have DBA authority or the appropriate combination of SELECT, UPDATE, or DELETE authorities for the tables and views accessed by the included SQL statement. In the case of views, you must have the appropriate authorities for all underlying views and base tables, as well.

Examples

1. Interactive SQL statement for the following query:

```
>isql=> SELECT PartName, VendorNumber, UnitPrice
> FROM Purchdb.Parts p, PurchDB.SupplyPrice sp
> WHERE p.PartNumber = sp.PartNumber
> AND p.PartNumber = '1123-P-01';
```

Generate the Plan:

```
isql=> GENPLAN FOR
> SELECT partname, vendornumber, UnitPrice
> FROM PurchDB.Parts p, PurchDB.SupplyPrice sp
> WHERE p.PartNumber = sp.PartNumber
> AND p.PartNumber = '1123-P-01';
```

Display the Plan:

```
isql=> SELECT * FROM System.Plan;

SELECT * FROM System.Plan;
-----------+-----------+-----------+--------------------+---------------
QUERYBLOCK |STEP       |LEVEL      |OPERATION           |TABLENAME
-----------+-----------+-----------+--------------------+---------------
          1|          1|          2|index scan          |PARTS
          1|          2|          2|serial scan         |SUPPLYPRICE
          1|          3|          1|nestedloop join     |
-------------------------------------------------------------------------
Number of rows selected is 3
U[p], d[own], l[eft], r[ight], t[op], b[ottom], pr[int], <n>, or e[nd] >r
+--------------------+----------------
|OWNER               |INDEXNAME
+--------------------+----------------
|PURCHDB             |PARTNUMINDEX
|PURCHDB             |
|                    |
-------------------------------------------------------------------------
Number of rows selected is 3
U[p], d[own], l[eft], r[ight], t[op], b[ottom], pr[int], <n>, or e[nd] >e
```

2. SQL statement simulating use of host variables in an application for the following query taken from an application:

```
EXEC SQL SELECT PartName, VendorNumber, UnitPrice
           INTO :PartName, :VendorNumber, :UnitPrice
           FROM PurchDB.Parts p, PurchDB.SupplyPrice sp
          WHERE p.PartNumber = sp.PartNumber
            AND p.PartNumber = :PartNumber
```

Remove INTO clause when placing the statement into GENPLAN.

Generate the plan in ISQL:

Define input host variable names and compatible SQL data types in WITH clause.

```
isql=> GENPLAN WITH (PartNumber char(16)) FOR
> SELECT PartName, VendorNumber, UnitPrice
>   FROM PurchDB.Parts p, PurchDB.SupplyPrice sp
>  WHERE p.PartNumber = sp.PartNumber
>    AND p.PartNumber = :PartNumber;
```

Display the plan:

```
isql=> SELECT * FROM System.Plan;

SELECT * FROM System.Plan;
-----------+-----------+-----------+--------------------+--------------
QUERYBLOCK |STEP       |LEVEL      |OPERATION           |TABLENAME
-----------+-----------+-----------+--------------------+--------------
          1|          1|          2|index scan          |PARTS
          1|          2|          2|serial scan         |SUPPLYPRICE
          1|          3|          1|nestedloop join     |
-------------------------------------------------------------------------
Number of rows selected is 3
U[p], d[own], l[eft], r[ight], t[op], b[ottom], pr[int], *lt;n>, or e[nd] >r
+--------------------+----------------
|OWNER               |INDEXNAME
+--------------------+----------------
|PURCHDB             |PARTNUMINDEX
|PURCHDB             |
|                    |
-------------------------------------------------------------------------------
Number of rows selected is 3
U[p], d[own], l[eft], r[ight], t[op], b[ottom], pr[int], <n>, or e[nd] >e
```

3. Example of GENPLAN for a MODULE SECTION.

```
GENPLAN FOR MODULE SECTION MyModule(10);
```

GOTO

The GOTO statement permits a jump to a labeled statement within a procedure.

Scope

Procedures only

SQL Syntax

```
{GOTO
 GO TO}{Label
       Integer}
```

Parameters

Label	specifies an identifier label for branching within the procedure.
Integer	specifies an integer label for branching within the procedure.

Description

The label or integer referred to in a GOTO statement is followed by a colon and a statement.

Authorization

Anyone can use the GOTO statement.

Example

```
CREATE PROCEDURE Process10 AS
   BEGIN
      INSERT INTO SmallOrders VALUES ('Widget', 10);
      IF ::sqlcode <> 0 THEN
         GOTO Errors;
      ENDIF;
      RETURN 0;

      Errors: PRINT 'There were errors.';
      RETURN 1;
   END;
```

GRANT

The GRANT statement gives specified authority to one or more users or authorization groups. The following forms of the GRANT statement are described individually:

- Grant table or view authority.
- Grant RUN or EXECUTE authority.
- Grant CONNECT, DBA, INSTALL, MONITOR, or RESOURCE authority.
- Grant SECTIONSPACE or TABLESPACE authority for a DBEFileSet.

For detailed information about security schemes, refer to the "DBEnvironment Configuration and Security" chapter of the *ALLBASE/SQL Database Administration Guide*.

Scope

ISQL or Application Programs

SQL Syntax — Grant Table or View Authority

```
GRANT {ALL [PRIVILEGES]
       {SELECT
        INSERT
        DELETE
        ALTER
        INDEX
        UPDATE [({ColumnName}[,...])]
        REFERENCES [({ColumnName}[,...])]}|,...|}
ON {[Owner.]TableName
    [Owner.]ViewName}TO {DBEUserID
                         GroupName
                         ClassName
                         PUBLIC    } [,...][WITH GRANT OPTION]
[BY {DBEUserID
     ClassName}]
```

Parameters — Grant Table or View Authority

`ALL [PRIVILEGES]` is the same as specifying all privileges you can grant on that table or view. For OWNER or DBA the privileges are SELECT, INSERT, DELETE, ALTER, INDEX, UPDATE, and REFERENCES. The word PRIVILEGES is not required; you can include it if you wish to improve readability. ALTER, INDEX, and REFERENCES are not applied when using GRANT ALL on views.

`SELECT` grants authority to retrieve data.

`INSERT` grants authority to insert rows.

`DELETE` grants authority to delete rows.

ALTER grants authority to add new columns. ALTER authority is not allowed for a view.

INDEX grants authority to create and drop indexes. INDEX authority is not allowed for a view.

UPDATE grants authority to change data in existing rows. A list of column names can be specified to grant UPDATE authority only for specific columns. Omitting the list of column names grants authority to update all columns.

REFERENCES grants authority to reference columns in the table from the foreign keys in other tables. A list of column names can be specified to grant REFERENCES authority only for specific columns. Omitting the list of column names grants authority to reference *all* columns. REFERENCES authority is not allowed for a view.

[*Owner.*]*TableName* designates a table for which authority is to be granted.

[*Owner.*]*ViewName* designates a view for which authority is to be granted.

TO The TO clause designates the users, authorization groups, and classes to be given the specified authority. You must specify a login name when specifying a DBEUserID. Authority granted to PUBLIC can be exercised by *all* users having CONNECT or DBA authority. Granting authority to a class is useful when program modules are owned by a class.

WITH GRANT OPTION allows the grantee of a privilege to grant that same privilege to another user. If WITH GRANT OPTION is specified, then all privileges being granted in the statement are granted with the grant option to all grantees. The grantee cannot be a group. The authority to grant cannot come solely from group membership.

BY specifies a DBEUserID or class as grantor of a privilege. This clause is used to provide a parent for an orphaned privilege. The named grantor cannot be a group or PUBLIC.

Authorization — Grant Table or View Authority

If you have DBA or OWNER authority directly (not due to group membership), or were previously granted table privileges with the WITH GRANT OPTION clause, you can issue the GRANT statement with the WITH GRANT OPTION clause for that table or view.

The BY clause can only be used by a DBA.

A user may be granted a privilege from one grantor only. OWNER, DBA, or grantable authority is required to issue the GRANT statement.

SQL Syntax — Grant RUN or EXECUTE Authority

```
GRANT {RUN ON [Owner.]ModuleName
       EXECUTE ON PROCEDURE [Owner.]ProcedureName}TO
{{DBEUserID
  GroupName
  ClassName} [,...]
  PUBLIC             }
```

Parameters — Grant RUN or EXECUTE Authority

RUN — grants authority to execute a specified module created interactively or by using a preprocessor.

[*Owner.*]*ModuleName* specifies the name of the module for which authority is to be granted.

EXECUTE — grants authority to execute a specified procedure.

[*Owner.*]*ProcedureName* specifies the name of the procedure for which authority is to be granted.

TO — The TO clause tells which users and authorization groups are to be granted the specified authority. You must specify a login name when specifying a DBEUserID. Authority granted to PUBLIC can be exercised by any user with CONNECT authority.

Authorization — Grant RUN or EXECUTE Authority

If you have DBA authority or OWNER authority, you can issue GRANT statements for any module or procedure.

To grant CONNECT, DBA, or RESOURCE authority, you must have DBA authority.

SQL Syntax — Grant CONNECT, DBA, INSTALL, MONITOR, or RESOURCE Authority

```
GRANT {CONNECT
       DBA
       INSTALL [AS OwnerID]
       MONITOR
       RESOURCE              }TO {DBEUserID
                                  GroupName
                                  ClassName } [,...]
```

Parameters — Grant CONNECT, DBA, INSTALL, MONITOR, or RESOURCE Authority

CONNECT — grants authority to use the CONNECT statement.

DBA — grants authority to issue any valid ALLBASE/SQL statement. A user with DBA authority is exempt from all authorization restrictions.

RESOURCE — grants authority to create tables and authorization groups.

MONITOR — grants authority to run SQLMON.

INSTALL — grants authority to INSTALL modules where the owner name equals the *OwnerID*. If the "AS *OwnerID*" clause is omitted, then grants authority to INSTALL modules having any owner name.

Modules for an application are created and installed when that application is preprocessed using one of the SQL preprocessors. Modules can also be installed by using the ISQL INSTALL command. See the *ALLBASE/ISQL*

Reference Manual for more details.

TO — The TO clause specifies the users, authorization groups, and classes to be given the specified authority. You must specify a login name when specifying a DBEUserID. Granting DBA authority to a class is useful when program modules are owned by a class.

Description — Grant CONNECT, DBA, INSTALL, MONITOR, or RESOURCE Authority

- If MONITOR authority is granted to a user, authorization group, or class that already has DBA authority, a warning is returned and explicit MONITOR authority is not granted since a DBA already has MONITOR authority.
- If DBA authority is granted to a user, authorization group, or class that already has MONITOR authority, MONITOR authority is upgraded to DBA authority.

Authorization — Grant CONNECT, DBA, INSTALL, MONITOR, or RESOURCE Authority

If you have OWNER authority for a table, view, or module, you can issue GRANT statements for that table or view. If you have DBA authority, you can issue GRANT statements for any table, view, or module. To grant CONNECT, DBA, INSTALL, MONITOR, or RESOURCE authority, you must have DBA authority.

SQL Syntax — Grant DBEFileSet Authority

```
GRANT {SECTIONSPACE
       TABLESPACE   } [,...] ON DBEFILESET DBEFileSetName TO
{DBEUserID
 GroupName
 ClassName
 PUBLIC     } [,...]
```

Parameters — Grant DBEFileSet Authority

SECTIONSPACE grants authority to store sections in the specified DBEFileSet.

A grant of SECTIONSPACE causes a check to see whether the STOREDSECT table has yet been created for the DBEFileSet. If there is no related STOREDSECT table, it is created.

When a user specifies a DBEFileSet for a section in a CREATE TABLE (check constraints), ALTER TABLE (check constraints), CREATE PROCEDURE, CREATE RULE, or PREPARE statement, in preprocessing, or in the ISQL INSTALL command, the owner of the section is checked for SECTIONSPACE authority on the DBEFileSet. If the user does not have SECTIONSPACE authority, the default SECTIONSPACE DBEFileSet is used instead. (See the SET DEFAULT DBEFILESET statement.) This applies even if the user has DBA authority.

TABLESPACE grants authority to store table and long column data in the specified

DBEFileSet.

When a user specifies the IN DBEFileSet clause in a CREATE TABLE statement for either the table or for a LONG column, the owner of the table is checked for TABLESPACE authority on the DBEFileSet. If the user does not have TABLESPACE authority, the default TABLESPACE DBEFileSet is used instead (See the SET DEFAULT DBEFILESET statement.) This applies even if the user has DBA authority.

DBEFileSetName designates the DBEFileSet for which authority is to be granted.

Description

- The execution of this statement causes modification to the HPRDBSS.SPACEAUTH system catalog table. Refer to the *ALLBASE/SQL Database Administration Guice* "System Catalog" chapter.

Authorization — Grant DBEFilesSet Authority

To grant SECTIONSPACE or TABLESPACE, you must have DBA authority. If you have DBA authority, you can issue the GRANT statement for any DBEFileSet.

Examples

1. Authorization groups

```
CREATE GROUP Warehse

GRANT CONNECT TO Warehse

GRANT SELECT,
      UPDATE (BinNumber,QtyOnHand,LastCountDate)
   ON PurchDB.Inventory
   TO Warehse
```

These two users will be able to start DBE sessions for PartsDBE, retrieve data from table PurchDB.Inventory, and update three columns in the table.

```
ADD Clem, George TO GROUP Warehse
```

Clem no longer has any of the authorities associated with group Warehse.

```
REMOVE Clem FROM GROUP Warehse
```

Because this group does not own any database objects, it can be deleted. George no longer has any of the authorities once associated with the group.

```
DROP GROUP Warehse
```

2. Using the WITH GRANT OPTION clause

   Clem and George have the SELECT privilege on the Inventory table as well as the ability to grant the SELECT privilege on this table to other users or a class with the WITH GRANT OPTION clause or to a group or PUBLIC (without the WITH GRANT OPTION).

```
GRANT SELECT
   ON PurchDB.Inventory
   TO Clem, George WITH GRANT OPTION
```

3. Module grants

```
GRANT RUN ON Statistics TO HelperDBA
GRANT RUN ON MyProg TO PUBLIC
```

   Rows associated with module Statistics are deleted from the system catalog.

```
DROP MODULE Statistics
```

   Authorization information for MyProg is retained, but the program is deleted from the system catalog. You can re-preprocess MyProg and do not have to redefine its authorization.

```
DROP MODULE MyProg PRESERVE
```

4. Procedure grants

```
GRANT EXECUTE ON PROCEDURE Process10 TO Managers
GRANT EXECUTE ON PROCEDURE Process12 TO AllUsers
```

5. DBEFileSet grants

   Grant the ability to store sections in DBEFileSet1 to PUBLIC.

```
GRANT SECTIONSPACE ON DBEFILESET DBEFileSet1 TO PUBLIC;
```

   Grant the ability to store table and long column data in DBEFileSet2 to PUBLIC.

```
GRANT TABLESPACE ON DBEFILESET DBEFileSet2 TO PUBLIC;
```

6. Grant authority to run SQLMON

```
GRANT MONITOR TO HelperDBA;
```

7. Grant a *DBEUserID* the authority to create modules owned by a specified *OwnerID*.

```
GRANT INSTALL AS John TO Clem;
```

IF

The IF statement is used to allow conditional execution of one or more statements within a procedure.

Scope

Procedures only

SQL Syntax

```
IF Condition THEN [Statement;[...]]
[ELSEIF Condition THEN [Statement; [...]]]
[ELSE [Statement; [...]]] ENDIF;
```

Parameters

Condition specifies anything that is allowed in a search condition except subqueries, column references, host variables, dynamic parameters, aggregate functions, string functions, date/time functions involving column references, long column functions, or TID functions. Local variables, built-in variables, and parameters may be included. See Chapter 9 , “Search Conditions,” for more information.

Statement is any statement allowed in a procedure, including a compound statement. The statement may also be empty.

Description

- IF statements can be nested. In a nested IF statement, each ELSE is associated with the closest preceding IF.
- Local variables, and parameters can be used anywhere a host variable would be allowed.
- Each *Statement* may be a single simple statement, a compound statement, or empty.

Authorization

Anyone can use the IF statement.

Example

Create a procedure to enter orders into different tables according to the size of the order:

```
CREATE PROCEDURE OrderEntry (PartName CHAR(20) NOT NULL,
      Quantity INTEGER NOT NULL) AS
   BEGIN
      IF :Quantity < 100 THEN
          INSERT INTO SmallOrders
          VALUES (:PartName, :Quantity);
      ELSE
          INSERT INTO LargeOrders
          VALUES (:PartName, :Quantity);
      ENDIF;
   END
```

Execute the procedure with different parameters. The first execution adds a row to the LargeOrders table.

```
EXECUTE PROCEDURE Reorder ('Widget', 1500)
```

The second execution adds a row to the SmallOrders table.

```
EXECUTE PROCEDURE Reorder ('Widget', 15)
```

INCLUDE

The INCLUDE preprocessor directive is used in an application program to declare the SQLCA or the SQLDA.

Scope

Application Programs Only

SQL Syntax

```
INCLUDE {SQLCA [[IS]EXTERNAL]
         SQLDA                }
```

Parameters

SQLCA and SQLDA identify data structures with special predefined meaning as follows:

- SQLCA is an area for ALLBASE/SQL output messages concerning the status of each SQL statement.
- SQLDA is an area for use in conjunction with dynamic preprocessing of SELECT statements.

Refer to the *ALLBASE/SQL Application Programming Guide* for the language you are using for more information on these data structures.

IS EXTERNAL — for the COBOL preprocessor only; declares the SQLCA structure as EXTERNAL. Then the SQLCA will not have to be passed explicitly to subprograms.

Description

- This directive cannot be used interactively or in procedures.
- You must always include the SQLCA in your ALLBASE/SQL application programs by using the INCLUDE statement or explicitly declaring the SQLCA yourself. At run time, ALLBASE/SQL puts information into the SQLCA that describes how SQL statements in the program executed.

Authorization

You do not need authorization to use the INCLUDE statement.

Example

```
INCLUDE SQLCA IS EXTERNAL

INCLUDE SQLDA
```

INSERT

The INSERT command adds rows to a table. The following two forms of the INSERT command are described individually:

- The form used to add rows having values you define. You can add a single row or (in an application program) you can insert multiple rows using the bulk facility. There is special syntax for prepared INSERT and BULK INSERT statements that use dynamic parameter substitution.
- The form used to add rows defined by a SELECT command. This form copies rows from one or more tables or views into a table and is called a Type 2 INSERT.

Rules defined with a *StatementType* of INSERT will affect both forms of INSERT command.

Scope

ISQL or Application Programs

SQL Syntax - Insert Rows with Defined Values

```
[BULK]INSERT INTO { [Owner.]TableName
                  [Owner.]ViewName}
[({ColumnName}[,...])]
VALUES ({SingleRowValues
         BulkValues
         ?                })
```

Parameters - Insert Rows with Defined Values

BULK — is specified in an application program to insert multiple rows with a single execution of the INSERT command.

[*Owner.*]*TableName* identifies the table to which data is to be added.

[*Owner.*]*ViewName* identifies a view on a single table; the data is added to the table upon which the view is based. Refer to the CREATE VIEW command for restrictions governing insertion via a view.

ColumnName specifies a column for which values are supplied.

If you omit any of the table's columns from the column name list, the INSERT command places the default value of the respective column definitions in the omitted columns. For columns with no default value, the null value is placed in the omitted columns. If the table definition specifies NOT NULL for any of the omitted columns, the INSERT command fails.

You can omit the column name list if you provide values for all columns of the table in the same order the columns were specified in the CREATE TABLE (or CREATE VIEW) command.

VALUES — The VALUES clause specifies the values corresponding to the columns in the column name list, or the columns specified in the CREATE TABLE or

CREATE VIEW commands, if no column name list exists. Character and date/time literals must be in single quotes.

SingleRowValues defines column values when you insert a single row. The syntax for *SingleRowValues* is presented separately below and includes single row syntax for statements that do not use dynamic parameter substitution.

BulkValues defines values when you use the BULK option. The syntax for *BulkValues* is presented separately below and includes bulk value syntax for statements that do not use dynamic parameter substitution.

? is a dynamic parameter value that defines column values within a prepared insert statement that uses dynamic parameter substitution. The syntax for *DynamicParameterValues* is presented separately below and includes both single row and bulk processing for such statements.

SQL Syntax — SingleRowValues

The following syntax applies to single row inserts that do not use dynamic parameter substitution.

```
{NULL
 USER
 :HostVariable [[INDICATOR]:IndicatorVariable]
 ?
 :LocalVariable
 :ProcedureParameter
 ::Built-inVariable
 ConversionFunction
 CurrentFunction
 [+
  -]{Integer
     Float
     Decimal}
 'CharacterString'
 0xHexadeciamalString
 'LongColumnIOString'  }[,...]
```

Parameters — SingleRowValues

NULL indicates a null value.

USER evaluates to the current DBEUserID. In ISQL, it evaluates to the login name of the ISQL user. From an application program, it evaluates to the login name of the individual running the program.

USER behaves like a CHAR(20) constant, with trailing blanks if the login name has fewer than 20 characters.

HostVariable contains a value in an application program being input to the expression.

IndicatorVariable names an indicator variable, whose value determines whether the associated host variable contains a NULL value:

> = 0	the value is not NULL
< 0	the value is NULL (The value in the host variable will be

ignored.)

NOTE To be consistent with the standard SQL and to support portability of code, it is strongly recommended that you use a –1 to indicate a null value. However, ALLBASE/SQL interprets all negative indicator variable values to mean a null value.

? is a place holder for a dynamic parameter in a prepared SQL statement in an application program. The value of the dynamic parameter is supplied at run time.

LocalVariable contains a value in a procedure.

ProcedureParameter contains a value that is passed into or out of a procedure.

Built-inVariable is one of the following built-in variables used for error handling:

- ::sqlcode
- ::sqlerrd2
- ::sqlwarn0
- ::sqlwarn1
- ::sqlwarn2
- ::sqlwarn6
- ::activexact

The first six of these have the same meaning that they have as fields in the SQLCA in application programs. Note that in procedures, sqlerrd2 returns the number of rows processed, for all host languages. However, in application programs, sqlerrd3 is used in COBOL, Fortran, and Pascal, while sqlerr2 is used in C. ::activexact indicates whether a transaction is in progress or not. For additional information, refer to the application programming guides and to Chapter 4 , “Constraints, Procedures, and Rules.”

ConversionFunction returns a value that is a conversion of a date/time data type into an INTEGER or CHAR value, or from a CHAR value.

CurrentFunction indicates the value of the current DATE, TIME, or DATETIME function.

Integer specifies a value of type INTEGER or SMALLINT.

Float specifies a value of type FLOAT or REAL.

Decimal specifies a value of type DECIMAL.

CharacterString specifies a CHAR, VARCHAR, DATE, TIME, DATETIME, or INTERVAL value.

HexadecimalString specifies a BINARY or VARBINARY value. If the string is shorter than the target column, it is padded with binary zeroes; if it is longer than the target column, the string is truncated.

LongColumnIOString specifies the input and output locations for the LONG data. The specification for this string is given below.

SQL Syntax — LongColumnIOString

```
<{[PathName/]FileName
  %SharedMemoryAddress}
[{>
  >>
  >!}[PathName/]{FileName
                 CharSting$
                 CharString$ CharString}
  >%{SharedMemoryAddress
     $                  }              ]
```

Parameters — LongColumnIOString

< [*PathName*/] *FileName* is the location of the input file.

<% *SharedMemoryAddress* is the shared memory address where the input is located.

> specifies that output is placed in the following file. If the file already exists, it is not overwritten nor appended to, and an error is generated.

>> specifies that output is appended to the following file name. If the file does not exist, it is created.

>! specifies that output is placed in the following file name. If the file already exists, it is overwritten.

>% *SharedMemoryAddress* is the shared memory address where the output is placed.

>%$ is the shared memory address, determined by ALLBASE/SQL, where the output is placed.

$ is the wildcard character that represents a random, five-byte alphanumeric character string generated by ALLBASE/SQL. This is a file name.

Description — LongColumnIOString

- The input device must have a permission allowing the login user to access it. For example, if the file belongs to the login user, permission must be at least 400. If the file belongs to another user, in a different group, permission must be at least 004.
- When an output device has been specified and it exists prior to a SELECT or FETCH command, ALLBASE/SQL does not change the file's owner or permission.
- The output device, if it does not exist prior to a SELECT or FETCH command, is created with the following characteristics.

Table 11-5. Default Output Device Characteristics

Device Type	Permission	UserID (uid)	GroupID (gid)
OUTPUT create	700	Current user login id	Current user login group
OUTPUT append	200	Current user login id	Current user login group
OUTPUT overwrite	200	Current user login id	Current user login group

- If the output device exists prior to a SELECT or FETCH command, in order for ALLBASE/SQL to access it for append or overwrite, the above characteristics are recommended.
- When no portion of the output device name is specified, the default file name, *tmp$.LF*, is used. The wildcard character ($) indicates a random, five-byte, alphanumeric character string. This file is created in the local directory.
- When you specify a portion of the output file name in conjunction with the wildcard character ($), a five-byte, alphanumeric character string replaces the wildcard. The wildcard character can appear in any position of the output device name *except* the first. The maximum file name being 14 bytes, you can specify 9 bytes of the device name.
- The wildcard character, whether user specified or part of the default output device name, is a unique five-byte, alphanumeric character string.
- When a file is used as the LONG column input or output device and you do not give it a specific path name in the LONG column I/O string, the default is the path where ISQL or your program is running.
- The output device cannot be overwritten with a SELECT or FETCH command unless you use the INSERT or UPDATE command with the overwrite option.
- LONG columns cannot be used as follows:
  - — In a WHERE clause.
  - — In a type II INSERT command.
  - — Remotely through ALLBASE/NET.
  - — As hash or B-tree index key columns.
  - — In a GROUP BY, ORDER BY, DISTINCT, or UNION clause.
  - — In an expression.
  - — In a subquery.
  - — In aggregate functions (AVG, SUM, MIN, MAX).
  - — As columns to which integrity constraints are assigned.
  - — With the DEFAULT option of the CREATE or ALTER TABLE commands.

SQL Syntax — BulkValues

The following syntax applies only to statements that do not use dynamic parameter substitution.

```
:Buffer [,:StartIndex [,:NumberOfRows]]
```

Parameters — BulkValues

Buffer is a host array or structure containing rows that are the input for the INSERT command. This array contains elements for each column to be inserted and indicator variables for columns that can contain null values. Whenever a column can contain nulls, an indicator variable must be included in the array definition immediately after the definition of that column. This indicator variable is an integer that can have the following values:

> = 0	the value is not NULL
< 0	the value is NULL

NOTE To be consistent with the standard SQL and to support portability of code, it is strongly recommended that you use a –1 to indicate a null value. However, ALLBASE/SQL interprets all negative indicator variable values to mean a null value.

StartIndex is a host variable whose value specifies the array subscript denoting where the first row to be inserted is stored in the array; default is the first element of the array.

NumberOfRows is a host variable whose value specifies the number of rows to insert; default is to insert from the starting index to the end of the array.

Description — Insert Rows with SingleRowValues and BulkValues

- When you enter SQL commands interactively, you cannot use host variables or the BULK option.
- You cannot use the BULK option in a procedure.
- If you omit any of the table's columns from the column name list, the INSERT command places the default value of the respective column definitions in the omitted columns. For columns with no default value, the null value is placed in the omitted columns. If the table definition specifies NOT NULL for any of the omitted columns, the INSERT command fails.
- If ALLBASE/SQL detects an error during a BULK INSERT operation, the error handling behavior is determined by the setting of the SET DML ATOMICITY and SET CONSTRAINTS statements. Refer to the discussion of these statements in this chapter for more information.
- For CHAR and VARCHAR data, if a *CharacterString* literal is shorter than the target column, it is padded with blanks; if it is longer than the target column, the string

is truncated. Refer to Chapter 7 , “Data Types,” for information on overflow and truncation of other data types.

- No error or warning condition is generated by ALLBASE/SQL when a character or binary string is truncated during an INSERT operation.
- Using the INSERT command with views requires that the views be based on queries that are updatable. See "Updatability of Queries" in Chapter 3 , “SQL Queries.”
- Values in referenced (primary key) columns must be inserted before values in referencing (foreign key) columns. However, if you do a bulk insertion, inserting the primary key rows after the foreign key rows does not cause an error message, because the constraints are satisfied by the time you COMMIT WORK.
- A table on which a unique constraint is defined cannot contain duplicate rows.
- BINARY and VARBINARY data can be inserted in character or hexadecimal format. Character format requires single quotes and hexadecimal requires a 0x before the value.
- Under the default settings for the SET DML ATOMICITY and SET CONSTRAINTS statements, integrity constraints on tables and views are enforced on a statement level basis and if a constraint should be violated during processing of the insert, no rows are inserted. However, the SET DML ATOMICITY and SET CONSTRAINTS statements both override the default behavior. For more information, it is important that you refer to the section "Error Conditions in ALLBASE/SQL" in Chapter 1 , “Introduction,” and the SET DML ATOMICITY or the SET CONSTRAINTS statements in this chapter.
- Rows being inserted must not cause the search condition of the table check constraint to be false and must cause the search condition of the view check constraint to be true.
- Rows being inserted in the table through a view having a WITH CHECK OPTION must satisfy the check constraint of the view and any underlying views in addition to satisfying any constraints of the table. Refer to the "Check Constraints" section inChapter 4 , “Constraints, Procedures, and Rules,” for more information on check constraints.
- Rules defined with a *StatementType* of INSERT will affect all kinds of INSERT statements performed on the rules' target tables. When the INSERT is performed, ALLBASE/SQL considers all the rules defined for that table with the INSERT *StatementType*. If the rule has no condition, it will fire for all rows affected by the statement and invoke its associated procedure with the specified parameters on each row. If the rule has a condition, it will evaluate the condition on each row. The rule will fire on rows for which the condition evaluates to TRUE and invoke the associated procedure with the specified parameters for each row. Invoking the procedure could cause other rules, and thus other procedures, to be invoked if statements within the procedure trigger other rules.
- If a DISABLE RULES statement is in effect, the INSERT statement will not fire any otherwise applicable rules. When a subsequent ENABLE RULES is issued, applicable rules will fire again, but only for subsequent INSERT statements, not for those rows processed when rule firing was disabled.
- In a rule defined with a *StatementType* of INSERT, any column reference in the *Condition* or any *ParameterValue* will refer to the value of the column as it is

assigned in the INSERT statement, or by the default value of the column if it is not included in the INSERT statement.

- When a rule is fired by this statement, the rule's procedure is invoked after the changes have been made to the database for that row and all previous rows. The rule's procedure, and any chained rules, will thus see the state of the database with the current partial execution of the statement.
- If an error occurs during processing of any rule considered during execution of this statement (including execution of any procedure invoked due to a rule firing), the statement and any procedures invoked by any rules will have no effect. Nothing will have been altered in the DBEnvironment as a result of this statement or the rules it fired. Error messages are returned in the normal way.
- The BULK option is not allowed within a procedure.

SQL Syntax — DynamicParameterValues

The following syntax applies to single row and bulk inserts that use

dynamic parameter substitution.

```
(? [,...])
```

Parameters — DynamicParameterValues

`(? [,...])` represents one or more host variables in a prepared INSERT statement. Each ? corresponds in sequential order to a column in the column name list of the prepared statement (even when BULK is used).

When you use a data structure of sqlda_type to pass dynamic parameter information between the application and ALLBASE/SQL, the number of "?"s specified must match the sqld field of the descriptor area and the number of values in a single element of the data buffer.

When you use host variables to pass dynamic parameter data values between the application and ALLBASE/SQL, the number of "?"s specified must match the number and order of the host variables in the related EXECUTE statement. This does not apply when you use the BULK option as you cannot mix host variables and dynamic parameters.

Description — Insert Rows with DynamicParameterValues

- Statements using question marks (?) indicating dynamic parameters can be intermixed with items in *SingleRowValues* and they can return either a value or a format. When using dynamic parameters for values, the dynamic parameter becomes the data type of the column. When using dynamic parameters for conversion functions, they become the data type to which they are assigned (CHAR 72). Only TO_DATE, TO_TIME, TO_DATETIME, and TO_INTERVAL are allowed here; TO_CHAR and TO_INTEGER are not allowed.
- When using the BULK option, statements using question marks (?), indicating dynamic parameters, can contain only question marks (and no host variables) to indicate column

input.

- The BULK option used with host variables is available for C, COBOL, and FORTRAN applications.
- The BULK option used with an sqlda_type data structure is available for C and Pascal applications.

Authorization — Insert Rows with SingleRowValues and Bulk Values

If you specify the name of a table, you must have INSERT or OWNER authority for that table or you must have DBA authority.

If you specify the name of a view, you must have INSERT or OWNER authority for that view or you must have DBA authority. Also, the owner of the view must have INSERT or OWNER authority with respect to the view's base tables, or the owner must have DBA authority.

SQL Syntax — INSERT Rows Defined by a SELECT Command (Type 2 Insert)

```
INSERT INTO { [Owner.]TableName
              [Owner.]ViewName}[(ColumnName [,...])] QueryExpression
```

Parameters — INSERT Rows Defined by a SELECT Command (Type 2 Insert)

[Owner.]TableName identifies the table to which data is to be added.

[Owner.]ViewName identifies a view on a single table; the data is added to the table upon which the view is based. Refer to the CREATE VIEW command for restrictions governing inserts via a view.

ColumnName specifies a column for which data is supplied from the select list in the SELECT command. Each column named must have a corresponding select list item. You can omit the column name list if you provide a select list item for all columns in the target table in the same order the columns were specified in the CREATE TABLE (or CREATE VIEW) command.

QueryExpression defines the rows to be inserted based on one or more tables and/or views in the DBEnvironment. The name of the target table cannot appear within the FROM clause or in a FROM clause of any subquery. The query expression cannot contain an INTO clause or a union operation.

The data types of each column in the select list must be compatible with the data types of corresponding columns in the target table. The first select list item defines the first column in the target table, the second select list item defines the second column in the target table, and so forth. The number of select list items must equal the number of columns in the target table.

Any column in the target table can contain null values only if it was not

defined with the NOT NULL attribute. Therefore ensure either that select list items are not null for any NOT NULL target column, or that the NOT NULL target columns have default values defined for them.

Description — INSERT Rows Defined by a SELECT Command (Type 2 Insert)

- You cannot use the ORDER BY clause in a Type 2 Insert.
- You cannot insert into a LONG column with this kind of `INSERT` operation.
- You cannot specify a LONG column in the QueryExpression in this kind of `INSERT` operation, except in a long column or string function.
- If you omit any of the table's columns from the column name list, the `INSERT` command places the default value of the respective column definitions in the omitted columns. For columns with no default value, the null value is placed in the omitted columns. If the table definition specifies NOT NULL for any of the omitted columns, the `INSERT` command fails.
- If ALLBASE/SQL detects an error during this kind of `INSERT` operation, error handling behavior is determined by the setting of the `SET DML ATOMICITY` and `SET CONSTRAINTS` statements. Refer to the discussion of these statements in this chapter.
- Using the INSERT command with views requires that the views be based on updatable queries. See "Updatability of Queries" in Chapter 3 , “SQL Queries.”
- A table on which a unique constraint is defined cannot contain duplicate rows.
- Under the default settings for the `SET DML ATOMICITY` and `SET CONSTRAINTS` statements, integrity constraints on tables and views are enforced on a statement level basis and if a constraint should be violated during processing of the insert, no rows are inserted. However, the `SET DML ATOMICITY` and `SET CONSTRAINTS` statements both override the default behavior. For more information, it is important that you refer to the section "Error Conditions in ALLBASE/SQL" in Chapter 1 , “Introduction,” and the `SET DML ATOMICITY` or the `SET CONSTRAINTS` statements in this chapter.
- Rows being inserted must not cause the search condition of the table check constraint to be false and must cause the search condition of the view check constraint to be true.
- Rows being inserted in the table through a view having a WITH CHECK OPTION must satisfy the check constraint of the view and any underlying views in addition to satisfying any constraints of the table. Refer to the "Check Constraints" section of Chapter 4 , “Constraints, Procedures, and Rules,” for more information on check constraints.
- Values in referenced (primary key) columns must be inserted before values in referencing (foreign key) columns. However, if you do a bulk insertion, inserting the primary key rows after the foreign key rows does not cause an error message, as the constraints are satisfied by the time you `COMMIT WORK.`
- BINARY and VARBINARY data can be inserted in character or hexadecimal format. Character format requires single quotes and hexadecimal requires a `0x` before the value.

- Rules defined with a *StatementType* of INSERT will affect all kinds of INSERT statements performed on the rules' target tables. When the INSERT is performed, ALLBASE/SQL considers all the rules defined for that table with the INSERT *StatementType*. If the rule has no condition, it will fire for all rows affected by the statement and invoke its associated procedure with the specified parameters on each row. If the rule has a condition, it will evaluate the condition on each row. The rule will fire on rows for which the condition evaluates to TRUE and invoke the associated procedure with the specified parameters for each row. Invoking the procedure could cause other rules, and thus other procedures, to be invoked if statements within the procedure trigger other rules.
- If a DISABLE RULES statement is in effect, the INSERT statement will not fire any otherwise applicable rules. When a subsequent ENABLE RULES is issued, applicable rules will fire again, but only for subsequent INSERT statements, not for those rows processed when rule firing was disabled.
- In a rule defined with a *StatementType* of INSERT, any column reference in the *Condition* or any *ParameterValue* will refer to the value of the column as it is assigned in the INSERT statement, or by the default value of the column if it is not included in the INSERT statement.
- The set of rows to be inserted by a type 2 INSERT (that is, an INSERT defined by a SELECT statement) is determined before any rule fires, and this set remains fixed until the completion of the rule. In other words, if the rule adds to, deletes from, or modifies this set, such changes are ignored.
- When a rule is fired by this statement, the rule's procedure is invoked after the changes have been made to the database for that row and all previous rows. The rule's procedure, and any chained rules, will thus see the state of the database with the current partial execution of the statement.
- If an error occurs during processing of any rule considered during execution of this statement (including execution of any procedure invoked due to a rule firing), the statement and any procedures invoked by any rules will have no effect. Nothing will have been altered in the DBEnvironment as a result of this statement or the rules it fired. Error messages are returned in the normal way.

Authorization — INSERT Rows Defined by a SELECT Command (Type 2 Insert)

To insert rows into a table, you must have INSERT or OWNER authority for that table or you must have DBA authority.

To insert rows using a view, you must have INSERT or OWNER authority for that view or you must have DBA authority. Also, the owner of the view must have INSERT or OWNER authority with respect to the view's base tables, or the owner must have DBA authority.

If you specify the name of a table in the FROM clause of the SELECT command, you must have SELECT or OWNER authority for the table or you must have DBA authority. If you specify the name of a view in the FROM clause of the SELECT command, you must have SELECT or OWNER authority for the view or you must have DBA authority. Also, the owner of the view must have SELECT or OWNER authority with respect to the view's definition, or the owner must have DBA authority.

Examples

1. Single-row insert

```
INSERT INTO PurchDB.Vendors
  VALUES  (9016,
          'Secure Systems, Inc.',
          'John Secret',
          '454-255-2087',
          '1111 Encryption Way',
          'Hush',
          'MD',
          '00007',
          'discount rates are carefully guarded secrets')
```

   A new row is added to the PurchDB.Vendors table.

2. Bulk insert

```
BULK INSERT INTO PurchDB.Parts
  (PartNumber, PartName)
  VALUES (:NewRow, :Indx, :NumRow)
```

   Programmatically, you can insert multiple rows with one execution of the INSERT command if you specify the BULK option. In this example, the rows to be inserted are in the array called NewRow.

3. Insert using SELECT operation

```
CREATE PUBLIC TABLE PurchDB.CalifVendors
  (VendorName        CHAR(30)       NOT NULL,
   PartNumber        CHAR(16)       NOT NULL,
   UnitPrice         DECIMAL(10,2),
   DeliveryDays      SMALLINT,
   VendorRemarks     VARCHAR(60)    )
   IN PurchFS
```

   This table has the same column attributes as corresponding columns in PurchDB.SupplyPrice and PurchDB.Vendors.

```
INSERT INTO PurchDB.CalifVendors
  SELECT  VendorName, PartNumber, UnitPrice, DeliveryDays, VendorRemarks
     FROM  PurchDB.Supplyprice, PurchDB.Vendors
     WHERE PurchDB.SupplyPrice.VendorNumber =
PurchDB.Vendors.VendorNumber
       AND VendorState = 'CA'
```

   Rows for California vendors are inserted based on a query result obtained by joining PurchDB.SupplyPrice and PurchDB.Vendors. A column list is omitted because all columns in the target table have a corresponding select list item.

4. Single row insert using dynamic parameters with host variables

```
PREPARE CMD FROM 'INSERT INTO PurchDB.Parts (PartNumber, PartName)
       VALUES(?,?);'
```

A new row is added to the PurchDB.Parts table based on the prepared INSERT statement called CMD. Row values are provided at run time, and an EXECUTE statement using two host variables is required to complete the INSERT.

```
EXECUTE CMD USING :PartNumber, :PartName;
```

5. Bulk insert using dynamic parameters with host variables

```
PREPARE CMD FROM 'BULK INSERT INTO PurchDB.Parts (PartNumber,
PartName)
        VALUES(?,?);'
```

Multiple rows can be added to the PurchDB.Parts table. Row values are provided at run time, and an EXECUTE statement using the address of a host variable array containing dynamic parameter data and host variables containing the starting index and number of rows to be inserted complete the INSERT.

```
EXECUTE CMD USING :DataBuffer, :StartIndex, :NumberOfRows;
```

6. Bulk insert or single row insert using dynamic parameters with sqlda_type and related data structures

```
PREPARE CMD FROM 'BULK INSERT INTO PurchDB.Parts (PartNumber, PartName)
        VALUES(?,?);'
```

One or more rows can be added to the PurchDB.Parts table. Row values are provided at run time, and an EXECUTE statement using a descriptor area is required to complete the INSERT.

Before issuing the execute statement, you must set certain fields in the descriptor area. (The ALLBASE/SQL application programming guides contain detailed information regarding this technique.) Then you describe the input to ALLBASE/SQL.

```
DESCRIBE INPUT CMD INTO Sqlda;
EXECUTE CMD USING DESCRIPTOR Sqlda;
```

Labeled Statement

A Label identifies an SQL statement that can be referred to within the procedure.

Scope

Procedures only

SQL Syntax

Label:*Statement*

Parameters

Label is an integer or a name which conforms to the SQL syntax rules for a basic name.

Statement is the statement within a procedure to be labeled.

Description

- A label may appear only at the start of a *ProcedureStatement* that is not part of a compound statement. It cannot appear with a local variable declaration or a WHENEVER directive.
- Labels within a procedure should be unique.
- A label can only be referred to from a GOTO statement and WHENEVER...GOTO directive.

Authorization

Anyone can use this statement.

Example

```
CREATE PROCEDURE Process19 (param1 integer, param2 float) AS
BEGIN
DECLARE value1 integer;
WHENEVER sqlerror GOTO errorexit;
DECLARE cursor1 CURSOR FOR
  SELECT column1
    FROM table1
   WHERE column1 > :param1;
OPEN cursor1;
WHILE ::sqlcode < > 100 do
  FETCH cursor1 into :value1;
  IF ::sqlcode = 100 THEN
    GOTO loopexit;
  ENDIF;
```

```
  INSERT INTO table2
       VALUES (:value1, :param2);
  UPDATE table3 SET column1 = CURRENT_DATE WHERE column2 = :value1;
  IF ::sqlerrd2 < 1 THEN
    INSERT INTO table3
         VALUES (CURRENT_DATE, :value1);
  ENDIF;
ENDWHILE;
loopexit:
CLOSE cursor1;
RETURN 0;
errorexit:
PRINT 'Procedure terminated due to error:';
PRINT ::
sqlcode;
END;
EXECUTE PROCEDURE Process19;
```

LOCK TABLE

The LOCK TABLE statement provides a means of explicitly acquiring a lock on a table, to override the automatic locking provided by ALLBASE/SQL in accord with the CREATE TABLE locking modes.

Scope

ISQL or Application Programs

SQL Syntax

```
LOCK TABLE [Owner.]TableName IN {SHARE [UPDATE]
                                EXCLUSIVE      }MODE
```

Parameters

[*Owner*.]*TableName* specifies the table to be locked.

SHARE allows other transactions to read but not change the table during the time you hold the lock.

Your transaction is delayed until any active transactions that have changed the table have ended. Then you can retrieve from the specified table with no further delays or overhead due to locking. Automatic locking of pages or rows takes place as usual any time your transaction changes the table.

SHARE UPDATE indicates that you may wish to update the rows selected. Other transactions may not update the data page you are currently reading. If you decide to update the row, an exclusive lock is obtained, so that other transactions cannot read or update the page; This lock is held until the transaction ends with a COMMIT WORK or ROLLBACK WORK statement.

EXCLUSIVE prevents other transactions from reading or changing the table during the time you hold the lock.

Your transaction is delayed until any transactions that were previously granted locks on the table have ended. Then your transaction experiences no further overhead or delays due to locking on the specified table.

Description

- Of the three lock types described here, the highest level is exclusive (X), the next share update (SIX), and the lowest share (S). When you request a lock on an object which is already locked with a higher severity lock, the request is ignored.
- This statement can be used to avoid the overhead of acquiring many small locks when scanning a table. For example, if you know that you are accessing all the rows of a table, you can lock the entire table at once instead of letting ALLBASE/SQL automatically lock each individual page or row as it is needed.

- LOCK TABLE can be useful in avoiding deadlocks by locking tables in a predetermined order.
- To ensure data consistency, all locks are held until the end of the transaction, at which point they are released. For this reason no UNLOCK statement is available or necessary.

Authorization

You can issue this statement if you have SELECT or OWNER authority for the table or if you have DBA authority.

Examples

1. Share Mode Lock

   ```
   BEGIN WORK
   ```

   Other transactions can issue only SELECT statements against the table until this transaction is terminated.

   ```
   LOCK TABLE PurchDB.OrderItems in SHARE MODE
   ```

   The lock is released when the transaction is either committed or rolled back.

   ```
   COMMIT WORK
   ```

2. Share Update Mode Lock

   ```
   BEGIN WORK
   ```

   Other transactions can issue only SELECT statements against the table:

   ```
   LOCK TABLE PurchDB.OrderItems in SHARE UPDATE MODE
   ```

   Other transactions can read the same page as the current transaction.

   ```
   SELECT ...  FROM PurchDB.OrderItems
   ```

   The shared lock is now upgraded to an exclusive lock for the page on which the update is taking place. Other transactions must wait for this transaction to be committed or rolled back.

   ```
   UPDATE PurchDB.OrderItems SET ...
   ```

   All locks are released when the transaction is either committed or rolled back.

   ```
   COMMIT WORK
   ```

LOG COMMENT

The LOG COMMENT statement permits the entry of comments into the ALLBASE/SQL DBELog file. These comments can be extracted using the Audit Tool.

Scope

ISQL or Application Programs

SQL Syntax

```
LOG COMMENT {'String'
             :HostVariable
             :ProcedureParameter
             :ProcedureLocalVariable
             ?                      }
```

Parameters

String specifies the comment as a constant character string (up to 3996 bytes)

HostVariable specifies the comment to be logged as a host variable. No indicator may be specified. The data type of the host variable must be CHAR or VARCHAR. If the value is null, an error is returned and no comment is logged.

ProcedureParameter or *ProcedureLocalVariable* specifies the comment to be logged as a procedure parameter or local variable. If the value is null, an error is returned and no comment is logged. The data type must be CHAR or VARCHAR.

? specifies the comment to be logged as a dynamic parameter. The data type is assumed to be VARCHAR(3996). If the value is null, an error is returned and no comment is logged.

Description

- The maximum length of a comment is 3996 bytes.
- A comment can use the DBEnvironment language or the native language.
- An error is returned if LOG COMMENT is used and audit logging is not enabled with the COMMENT audit element or the COMMENT PARTITION is NONE.

Authorization

Any user can issue this statement from within a database session.

Example

Generate a comment audit log record.

```
LOG COMMENT 'Select From Table PurchDB.Parts';
SELECT PartNo FROM PurchDB.Parts WHERE PartNo='1234';
```

OPEN

The OPEN statement is used in an application program or a procedure to open a cursor, that is, make the cursor and its associated active set available to manipulate a query result.

Scope

Application Programs and Procedures Only

SQL Syntax

```
OPEN CursorName [KEEP CURSOR [WITH LOCKS
                              WITH NOLOCKS]]
[USING { [SQL]DESCRIPTOR {SQLDA
                          AreaName}
          HostVariableName[[INDICATOR]:IndicatorVariable][,...]} ]
```

Parameters

CursorName specifies the cursor to be opened. The cursor name must first be defined with a DECLARE CURSOR statement.

KEEP CURSOR maintains the cursor position across transactions until a CLOSE statement is issued on the cursor.

This clause is not available for procedure cursors (those declared for an EXECUTE PROCEDURE statement).

WITH LOCKS keeps only those locks associated with the position of the kept cursor after a COMMIT WORK statement, and releases all other locks. This is the default.

WITH NOLOCKS releases all locks associated with the kept cursor after a COMMIT WORK statement.

USING allows dynamic parameter substitution in a prepared statement.

This clause can only be specified within an application when opening a cursor on a dynamically prepared SELECT or EXECUTE PROCEDURE statement.

SQL DESCRIPTOR specifies a location that at run time contains the data value assigned to an input dynamic parameter specified in a prepared SELECT or EXECUTE PROCEDURE statement.

Specify the same location (SQLDA or *AreaName*) as you specified in the DESCRIBE INPUT statement.

SQLDA specifies that a data structure of sqlda_type named **sqlda** is used to pass dynamic parameter data between the application and ALLBASE/SQL.

AreaName specifies the user defined name of a data structure of type sqlda_type that is used to pass dynamic parameter data between the application and ALLBASE/SQL.

HostVariableName specifies a host variable name that at run time contains the data value that is assigned to an input dynamic parameter specified in the parameter list of a prepared SELECT or EXECUTE PROCEDURE statement.

Host variables must be specified in the same order as the dynamic parameters in the prepared statement they represent. There must be a one to one correspondence between host variable names and the dynamic parameters. A maximum of 1023 host variables names can be specified.

IndicatorVariable names an indicator variable, whose value determines whether the associated host variable contains a NULL value:

> = 0	the value is not NULL
< 0	the value is NULL

Description

- For a select cursor, ALLBASE/SQL examines any input host variables and input dynamic parameters used in the cursor definition, determines the cursor's active set, positions the cursor before the first row of the active set, and leaves the cursor in the open state. No rows are actually available to your application program until a FETCH statement is executed.
- For a procedure cursor, ALLBASE/SQL examines any input host variables and input dynamic parameters used in the cursor definition. No rows are actually available to your application program, nor does procedure execution begin, until ADVANCE and/or FETCH statements are executed.
- For a select cursor, the KEEP CURSOR option lets you maintain the cursor position in an active set beyond transaction boundaries. When you use this option, the COMMIT WORK and ROLLBACK WORK statements do not automatically close the cursor. Instead, you must explicitly close the cursor and then issue a COMMIT WORK.
- Cursors not using the KEEP CURSOR option are automatically closed when a transaction terminates or a ROLLBACK WORK TO SAVEPOINT is executed.

PREPARE

The PREPARE statement dynamically preprocesses an SQL statement for later execution.

Scope

ISQL or Application Programs

SQL Syntax

```
PREPARE [REPEAT]{StatementName
                 [Owner.]ModuleName [(SectionNumber)]}
[IN DBEFileSetName]FROM {'String'
                         :HostVariable}
```

Parameters

REPEAT specifies the use of semi-permanent sections for queries. Unlike temporary sections, semi-permanent sections are retained in memory until the DBEnvironment session ends, not when the current transaction ends.

To improve performance, you can set the Authorize Once per Session flag to ON with the SQLUtil ALTDBE command when using semi-permanent sections. However, you must take care to ensure that a prepared statement is not executed after authorization has been revoked from the object that contains that statement.

StatementName This option of the PREPARE statement is used in an application program; it cannot be used interactively. Refer to the ALLBASE/SQL application programming guide for the language you are using to determine whether this statement is supported in that language.

StatementName specifies a name for the statement being preprocessed. You reference *StatementName* in an EXECUTE statement later in the current transaction to execute the dynamically preprocessed statement. *StatementName* must conform to the ALLBASE/SQL rules for a basic name given in the "Names" chapter. Two PREPARE statements in an application program cannot specify the same *StatementName*.

When necessary, you use the DESCRIBE statement to determine whether the prepared statement is a SELECT statement. If so, other information provided by the DESCRIBE statement helps you determine how much storage to dynamically allocate for the query result; then you reference the *StatementName* in a DECLARE CURSOR statement and use the cursor to execute the dynamically preprocessed SELECT statement.

If it is possible that dynamic parameters are present in the prepared statement, you must use the DESCRIBE statement with the INPUT clause. If dynamic parameters are present, the appropriate data buffer or host variables must be loaded with the values of any dynamic parameters before the statement can be executed.

See related ALLBASE/SQL statements in this manual and the appropriate ALLBASE/SQL application programming guide for details of these programming methods.

[*Owner*.]*ModuleName* [(*SectionNumber*)] This option of the PREPARE statement is used interactively; it cannot be used in an application program.

This option specifies an identifier to be assigned to the statement being preprocessed. Later, the identifier can be specified in an EXECUTE statement to execute the dynamically preprocessed statement.

The section number is an integer to be used in identifying the dynamically preprocessed statement. You can group several related sections under the same module name by giving each one a different section number. You can specify any section number from 1 to 32767. If you do not specify a section number, section number 1 is assumed.

You must not already have a dynamically preprocessed statement with the same module name and section number. You must not already have a preprocessed application program with the specified module name.

You can specify an owner name if you have DBA authority. Non-DBA users can specify the name of any group of which they are a member. Otherwise, ALLBASE/SQL assigns your login name as the owner name of the module.

You cannot interactively prepare a SELECT statement.

DBEFileSetName identifies the DBEFileSet used to store the dynamically prepared statement. If not specified, the default SECTIONSPACE DBEFileSet is used. (Refer to syntax for the SET DEFAULT DBEFILESET statement.)

String is the statement to be preprocessed. The preprocessor cannot process more than 32,762 characters. If the string contains embedded strings, delimit the embedded strings with double single quotation marks as follows:

```
PREPARE MyStatement FROM 'DELETE FROM PurchDB.Parts
   WHERE PartNumber = ''1123-P-01'''
```

HostVariable specifies a host variable having as its value a character string which is the statement to be preprocessed. The preprocessor cannot process more than 32,762 characters. However, the length of a string contained in a host variable is limited by the defined length of the host variable.

Description

- You cannot use the PREPARE statement to preprocess the following statements:

```
ADVANCE                  BEGIN DECLARE SECTION    BEGIN WORK
CLOSE                    COMMIT WORK              CONNECT
DECLARE CURSOR           DELETE WHERE CURRENT     DESCRIBE
DISCONNECT               END DECLARE SECTION      EXECUTE
EXTRACT                  FETCH                    INCLUDE
OPEN                     PREPARE                  RELEASE
ROLLBACK WORK            SET CONNECTION           SET SESSION
SET TRANSACTION          SETOPT                   START DBE
STOP DBE                 SQLEXPLAIN               UPDATE WHERE CURRENT
TERMINATE USER           WHENEVER
```

- You cannot interactively prepare a SELECT statement.
- A statement to be dynamically preprocessed in an application program must be terminated with a semicolon.
- You cannot prepare a statement which contains host variables. Dynamic parameters should be used instead. (Use PREPARE without the REPEAT option.)
- In an application program, a dynamically preprocessed statement (PREPARE without the REPEAT option) is automatically deleted from the system at the end of the transaction in which it was prepared. It cannot be executed in any other transaction.
- When a PREPARE statement is issued interactively, the dynamically preprocessed statement is stored in the system catalog until deleted by a DROP MODULE statement. The statement is not stored, however, if you specify an owner name of TEMP.
- If the IN *DBEFileSetName* clause is specified, but the module owner does not have SECTIONSPACE authority for the specified DBEFileSet, a warning is issued and the default SECTIONSPACE DBEFileSet is used instead. (Refer to syntax for the GRANT statement and the SET DEFAULT DBEFILESET statement.)

Authorization

You do not need authorization to use the PREPARE statement. However, the authority required to execute the dynamically preprocessed statement depends on whether the statement is executed programmatically or interactively. Refer to the EXECUTE statement authorization for details.

To specify a *DBEFileSetName* for a prepared section, the module owner must have SECTIONSPACE authority on the referenced DBEFileSet.

Examples

1. Interactive use

```
PREPARE Statistics(1)
FROM 'UPDATE STATISTICS FOR TABLE PurchDB.Orders'

PREPARE Statistics(2)
FROM'UPDATE STATISTICS FOR TABLE PurchDB.OrderItems'
```

Two sections for module Statistics are stored in the system catalog.

```
EXECUTE Statistics(1)
```

The statistics for table PurchDB.Orders are updated.

```
EXECUTE Statistics(2)
```

The statistics for table PurchDB.OrderItems are updated.

```
DROP MODULE Statistics
```

Both sections of the module are deleted.

2. Programmatic use

If you know in advance that the statement to be dynamically preprocessed is not a SELECT statement and does not contain dynamic parameters, you can prepare it and execute it in one step, as follows:

```
EXECUTE IMMEDIATE :Dynam1
```

It may be more appropriate to prepare and execute the statement in separate operations. For example, if you don't know the format of a statement:

```
PREPARE Dynamic1 FROM :Dynam1
```

The statement stored in :Dynam1 is dynamically preprocessed.

```
DESCRIBE Dynamic1 INTO Sqlda
```

If Dynamic1 is not a SELECT statement, the Sqld field of the Sqlda data structure is 0. In this case, if you know there are no dynamic parameters in the prepared statement, use the EXECUTE statement to execute the dynamically preprocessed statement.

If it is possible that dynamic parameters are present in the prepared statement, you must describe the statement for input:

```
DESCRIBE INPUT Dynamic1 USING SQL DESCRIPTOR SqldaIn
```

If dynamic parameters are present, the appropriate data buffer or host variables must be loaded with the values of any dynamic parameters. Then if the statement is not a query, it can be executed, as in this example using a data buffer:

```
EXECUTE Dynamic1 USING SQL DESCRIPTOR SqldaIn
```

If Dynamic1 is a SELECT statement and the language you are using supports dynamically defined SELECT statements, use a cursor to manipulate the rows in the query result:

```
DECLARE Dynamic1Cursor CURSOR FOR Dynamic1
```

Place the appropriate values into the SQL descriptor areas. Use the USING DESCRIPTOR clause of the OPEN statement to identify where dynamic parameter information is located. Use the USING DESCRIPTOR clause of the FETCH statement to identify where to place the rows selected.

```
OPEN Dynamic1Cursor USING SQL DESCRIPTOR SqldaIn
```

Load related dynamic parameter data into the input data buffer.

```
FETCH Dynamic1Cursor USING DESCRIPTOR SqldaOut
.
.
.
```

When all rows have been processed, close the cursor:

```
CLOSE Dynamic1Cursor
```

PRINT

The PRINT statement is used inside a procedure to store the content of user-defined strings, local variables, parameters, or built-in variables in the message buffer for display by ISQL or an application program.

Scope

Procedures only

SQL Syntax

```
PRINT {'Constant'
       :LocalVariable
       :Parameter
       ::Built-inVariable};
```

Parameters

Constant is a string literal.

LocalVariable is a local variable declared within the procedure. Types and sizes are the same as for column definitions, except you cannot specify a LONG data type.

Parameter is a parameter declared within the procedure.

Built-inVariable is one of the following built-in variables used for error handling:

- ::sqlcode
- ::sqlerrd2
- ::sqlwarn0
- ::sqlwarn1
- ::sqlwarn2
- ::sqlwarn6
- ::activexact

The first six of these have the same meaning that they have as fields in the SQLCA in application programs. Note that in procedures, sqlerrd2 returns the number of rows processed for all host languages. However, in application programs, sqlerrd3 is used in COBOL, Fortran, and Pascal, while sqlerr2 is used in C. ::activexact indicates whether a transaction is in progress or not. For additional information, refer to the application programming guides and to the chapter "Constraints, Procedures, and Rules."

Description

- The results of any PRINT statements issued during the execution of a procedure are placed in the ALLBASE/SQL message buffer, and may be displayed like other messages. In an application program, they can be retrieved with SQLEXPLAIN upon exiting the procedure.
- The message number 5000 is used for all PRINT statements.

Authorization

Anyone can issue the PRINT statement.

Examples

```
CREATE PROCEDURE Process15 (PartNumber CHAR (16) NOT NULL) AS
   BEGIN
      DECLARE PartName CHAR(30);

      SELECT PartName INTO :PartName
      FROM PurchDB.Parts
      WHERE PartNumber = :PartNumber;
      IF ::sqlcode <> 0 THEN
          PRINT 'Row not retrieved.  Error code:';
          PRINT ::sqlcode;
      ELSE
          PRINT :PartName;
      ENDIF;
   END;
```

When an application program calls a procedure, you can include PRINT statements in the procedure for later retrieval by the application:

```
IF ::sqlcode = 100 THEN
   PRINT 'Row was not found';
ELSE
   PRINT 'Error in SELECT statement';
ELSEIF ::sqlcode=0 THEN
   PRINT :PartName;
ENDIF;
```

On returning from the procedure, use SQLEXPLAIN in a loop to extract all the messages generated by PRINT during the operation of the procedure.

In C:

```
while (sqlcode != 0 || sqlwarn[0]=='W') {
   EXEC SQL SQLEXPLAIN :SQLMessage;
   printf("%s\n",SQLMessage);
   }
```

In COBOL:

```
IF SQLCODE IS NOT ZERO OR SQLWARN0 = "W"
   PERFORM M100-DISPLAY-MESSAGE
   UNTIL SQLCODE IS ZERO AND SQLWARN0 = "W".
.
.
.
M100-DISPLAY-MESSAGE.
   EXEC SQL SQLEXPLAIN :SQLMESSAGE END-EXEC.
   DISPLAY SQLMESSAGE.
M100-EXIT.
   EXIT.
```

RAISE ERROR

The RAISE ERROR statement causes an error to occur and causes the given error number to be put into the ALLBASE/SQL message buffer, together with the given error text. This statement is most useful within procedures invoked by rules, to cause the rule to fail and the statement firing the rule to have no effect. The effect of RAISE ERROR is to return with an error status; this statement can never "execute successfully."

Scope

ISQL or Application Programs

SQL Syntax

```
RAISE ERROR [ErrorNumber] [MESSAGE ErrorText]
```

Parameters

ErrorNumber specifies the number of the error being raised. This can be any integer value. *ErrorNumber* has the following syntax:

```
{Integer
 :HostVariable
 ?
 :LocalVariable
 :ProcedureParameter}
```

The data type of the parameter, host variable, or local variable must be INTEGER or SMALLINT. The data type expected for the dynamic parameter is INTEGER.

If no *ErrorNumber* is given, 2350 is the default error number. The error range 7000 - 7999 is reserved for the RAISE ERROR statement. No ALLBASE/SQL errors are in this range.

Parameters and local variables may only be used within procedures. Host variables may only be used within embedded SQL. Dynamic parameters may only be used within dynamic SQL.

ErrorText specifies text to be returned with the error. *ErrorText* has the following syntax:

```
{'CharacterString'
 :HostVariable
 ?
 :LocalVariable
 :ProcedureParameter}
```

The data type of the parameter, host, or local variable must be CHAR or VARCHAR. The data type expected for the dynamic parameter is CHAR(250). The value will be truncated to 250 bytes.

If no *ErrorText* is given, the default is an empty string.

Parameters and local variables are only used within procedures. Host variables are only used within embedded SQL. Dynamic parameters are only used within dynamic SQL.

Description

- RAISE ERROR is for user-defined errors. The errors returned are application specific.
- If *ErrorNumber* or *ErrorText* is NULL, an error is returned and the message is not generated.
- *ErrorNumber*, if specified, must be greater than 0.
- Execution of RAISE ERROR causes the number of the raised error to be placed in sqlcode and the RAISE ERROR text to be placed in the message buffer.

  Since an error condition is the expected result of the statement, no corrective action need be taken except as directed by the application developer. Applications can use SQLEXPLAIN to fetch the text of the message and interpret it appropriately. Applications can also examine and/or display sqlcode.
- You can use the DESCRIBE INPUT statement on this statement after you PREPARE it to show the number and characteristics of dynamic parameters, if any are used.

Authorization

Any user can issue this statement.

Examples

1. Example coded in a procedure to be invoked by a rule

```
SELECT COUNT(*) INTO :rows FROM PurchDB.Orders
    WHERE VendorNumber = :VendorNumber;
IF :rows <> 0 THEN
    RAISE ERROR 1 MESSAGE 'Vendor number exists in the "Orders" table.';
ENDIF;
```

2. Interactive example

```
isql=> raise error 1 message 'This is error 1';
This is error 1
isql=>
```

3. Example using dynamic parameters

```
EXEC SQL PREPARE MyCmd from 'RAISE ERROR ? MESSAGE ?';
```

   Accept values for error number and message text into host variables :ErrorNumber and :ErrorText, then execute the prepared command:

```
EXEC SQL EXECUTE MyCmd USING :ErrorNumber, :ErrorText;
```

REFETCH

The REFETCH statement allows Read Committed (RC) and Read Uncommitted (RU) transactions to acquire intended-for-update locks on data objects and to revalidate data before an update operation is issued. A refetch should always be done in RC and RU transactions before updating data to avoid update anomalies.

Scope

Application Programs Only

SQL Syntax

```
REFETCH CursorName INTO {:HostVariable [[ INDICATOR] :Indicator]}[,...]
```

Parameters

CursorName identifies a cursor. The cursor's active set is determined when the cursor is opened. The cursor's current position in the active set is determined by the last FETCH statement. The REFETCH statement retrieves the current row.

The cursor specified in the REFETCH statement must be declared for update and must be updatable.

INTO The INTO clause defines where to place the row fetched.

HostVariable identifies the host variable corresponding to one column in the fetched row.

Indicator names the indicator variable, an output host variable whose value (see following) depends on whether the host variable contains a null value:

0	the value is not NULL
–1	the value is NULL
> 0	the value is truncated (for CHAR, VARCHAR, BINARY, and VARBINARY values only).

Description

- The purpose of the REFETCH statement is to revalidate data prior to carrying out an update when using the Read Committed (RC) or Read Uncommitted (RU) isolation level in a transaction. If you do not use the REFETCH statement prior to updating a row in a RC or RU transaction, you may not be able to determine whether the row has already been modified by some other transaction.
- The comparison of the refetched data with the data selected with the RC or RU statement must be on a row by row basis rather than the whole buffer because slack or filler bytes between columns are not initialized and can incorrectly influence the comparison.

- Because UPDATE WHERE CURRENT does not accept a DESCRIPTOR clause for input values, the REFETCH statement does not support the USING DESCRIPTOR clause found in the FETCH statement.
- No BULK option is available.
- This statement cannot be used interactively or in procedures.
- If there is no current row during a REFETCH, you receive the following message in the SQLCODE:

```
Row not found.
```

Authorization

You do not need authorization to use REFETCH.

Example

```
label 1000;
var
   EXEC SQL INCLUDE SQLCA;
   EXEC SQL BEGIN DECLARE SECTION;
     sqlmessage : packed array [1..132] of char;
     host1, host2, updatevalue : integer;
   EXEC SQL END DECLARE SECTION;

begin
.
.
.
  EXEC SQL BEGIN WORK RU;
  EXEC SQL DECLARE C1 CURSOR FOR UPDATE OF Col1 FROM T1 WHERE Predicate;
  EXEC SQL OPEN C1;

  repeat
    EXEC SQL FETCH C1 INTO :Host1;
    if SQLCA.sqlcode <> 0 then
       begin
          EXEC SQL SQLEXPLAIN :sqlmessage;
          write sqlmessage;
          goto 1000;
       end;
    write Host1;
```

Read Input. If an update is needed:

```
          begin
             read updatevalue;
             EXEC SQL REFETCH C1 INTO :Host2;
             if SQLCA.sqlcode <> 0 then
                 begin
                    EXEC SQL SQLEXPLAIN :sqlmessage;
                    write sqlmessage;
                    goto 1000;
                 end;
             if Host1 = Host2 then
                EXEC SQL UPDATE T1 SET Col1 = updatevalue
                     WHERE CURRENT OF C1;
             else
                write "data changed to ", Host2;
          end;
   1000:
     until SQLCA.sqlcode = 100
```

No More Rows Found

RELEASE

The RELEASE statement terminates your DBE session.

Scope

ISQL or Application Programs

SQL Syntax

RELEASE

Description

- A ROLLBACK is performed on any transactions in progress.
- Any locks still held are released. Any cursors still open are closed, including kept cursors.
- If the AUTOSTART option is in effect and your session is the only one in process, a RELEASE statement forces a checkpoint.
- Following a RELEASE or DISCONNECT CURRENT command, there is no current connection until a SET CONNECTION command is used to set the current connection to another existing connection, or a new connection is established by using the CONNECT, START DBE, START DBE NEW, or START DBE NEW LOG commands.

Authorization

You do not need authorization to use the RELEASE statement.

Example

```
CONNECT TO '../sampledb/PartsDBE'
```

ALLBASE/SQL establishes a DBE session for you. Once you have a DBE session, you can submit SQL statements. After submitting the statements, terminate your DBE session:

```
RELEASE
```

REMOVE DBEFILE

The REMOVE DBEFILE statement removes the name of the DBEFileSet that the DBEFile was associated with from SYSTEM.DBEFile.

Scope

ISQL or Application Programs

SQL Syntax

REMOVE DBEFILE *DBEFileName* **FROM DBEFILESET** *DBEFileSetName*

Parameters

DBEFileName is the name of the DBEFile to be removed. The DBEFile must be empty (contain no tables, long data, or indexes).

DBEFileSetName is the name of the DBEFileSet with which the DBEFile is currently associated.

Description

- You must have exclusive access to all tables associated with the DBEFileSet.
- After you remove a DBEFile from a DBFileSet, you can drop the DBEFile or add it to another DBEFileSet.
- Before a DBEFile can be removed from the SYSTEM DBEFileSet, other users' transactions must complete. Other users must wait until the transaction that is removing the DBEFile from SYSTEM has completed.
- REMOVE DBEFILE also decreases the number of files associated with the DBEFileSet shown in the DBEFSNDBEFILES column of SYSTEM.DBEFileSet by one.

Authorization

You must have DBA authority to use this statement.

Example

```
CREATE DBEFILE ThisDBEFile WITH PAGES = 4,
               NAME = 'ThisFile', TYPE = TABLE

CREATE DBEFILESET Miscellaneous

ADD DBEFILE ThisDBEFile TO DBEFILESET Miscellaneous
```

The DBEFile is used to store rows of a new table. When the table needs an index, one is created as follows:

```
CREATE DBEFILE ThatDBEFile WITH PAGES = 4,
```

```
                NAME = 'ThatFile', TYPE = INDEX

   ADD DBEFILE ThatDBEFile to DBEFILESET Miscellaneous
```

When the index is subsequently dropped, its file space can be assigned to another DBEFileSet.

```
   REMOVE DBEFILE ThatDBEFile FROM DBEFILESET Miscellaneous

   ADD DBEFILE ThatDBEFile TO DBEFILESET SYSTEM

   ALTER DBEFILE ThisDBEFile SET TYPE = MIXED
```

Now you can use this DBEFile to store an index later if you need one. All rows are later deleted from the table, so you can reclaim file space.

```
   REMOVE DBEFILE ThisDBEFile FROM DBEFILESET Miscellaneous

   DROP DBEFILE ThisDBEFile
```

The DBEFileSet definition can now be dropped.

```
   DROP DBEFILESET Miscellaneous
```

REMOVE FROM GROUP

The REMOVE FROM GROUP statement removes one or more users or authorization groups from membership in a specified authorization group.

Scope

ISQL or Application Programs

SQL Syntax

```
REMOVE {DBEuserID
        GroupName
        ClassName}[,...]FROM GROUP [Owner.]TargetGroupName
```

Parameters

DBEUserID identifies a user to be removed from the specified authorization group. If you specify several names, any invalid names are ignored but valid names are removed.

GroupName identifies a group to be removed from the specified authorization group. If you specify several names, any invalid names are ignored but valid names are removed.

ClassName identifies a class to be removed from the specified authorization group. If you specify several names, any invalid names are ignored but valid names are removed.

TargetGroupName is the name of the authorization group from which the specified users and groups are to be removed.

Description

You cannot remove a member from a group if that member (or members of that member) used a DBA or REFERENCES privilege to which that member had access through that group to validate the creation of a currently existing foreign key in a table he or she owns.

Authorization

You can use this statement if you have OWNER authority for the authorization group or if you have DBA authority.

Example

```
CREATE GROUP Warehse

GRANT CONNECT TO Warehse

GRANT SELECT,
      UPDATE (BinNumber,QtyOnHand,LastCountDate)
   ON PurchDB.Inventory
   TO Warehse

ADD Clem, George TO GROUP Warehse
```

These two users now are able to start DBE sessions on PartsDBE and PurchDB.Inventory, and to update three columns in the table.

```
REMOVE Clem FROM GROUP Warehse
```

Clem no longer has any of the authorities associated with group Warehse.

```
DROP GROUP Warehse
```

Because this group does not own any database objects, it can be deleted. George no longer has any of the authorities once associated with the group.

RENAME COLUMN

The RENAME COLUMN statement defines a new name for an existing column in the DBEnvironment.

Scope

Application Programs

SQL Syntax

RENAME COLUMN [*Owner*.]*TableName*.*ColumnName* **TO** *NewColumnName*

Parameters

[*Owner*.]*TableName*.*ColumnName* designates the table column to be renamed.

NewColumnName is the new column name.

Description

- All indexes, columns, default columns, constraints, referential authorization, rules, and user authorities tables dependent on a renamed column will be renamed.
- All views dependent on a renamed column will be dropped.
- If a column has check constraints, then that column cannot be renamed.

Authorization

You must have DBA authority to use this statement.

Example

```
RENAME COLUMN Parts.PartNumber to NewPartNum;
```

RENAME TABLE

The RENAME TABLE statement defines a new name for an existing table in the DBEnvironment.

Scope

Application Programs

SQL Syntax

```
RENAME TABLE [Owner.]TableName TO NewTableName
```

Parameters

[*Owner*.]*TableName* designates the table to be renamed.

NewTableName is the new table name.

Description

- All indexes, columns, default columns, constraints, referential authorization, rules, and user authorities tables dependent on a renamed table will be renamed.
- When using RENAME command, data and grants made for tables are carried forward for the new name. No unload, load data, or recreating index is necessary.
- All views dependent on a renamed table will be dropped.
- If a table has check constraints, then that table cannot be renamed.

Authorization

You must have DBA authority to use this statement.

Example

```
RENAME TABLE PurchDB.Parts to NewParts;
```

RESET

The RESET statement resets ALLBASE/SQL accounting and statistical data.

Scope

ISQL or Application Program

SQL Syntax

```
RESET {SYSTEM.ACCOUNT [FOR USER {*
                                 DBEUserID}]
       SYSTEM.COUNTER                       }
```

Parameters

`SYSTEM.ACCOUNT` is specified to reset accounting data for one user's DBE session or for all active sessions.

`*` specifies all active sessions. This is the default if the FOR USER clause is omitted.

`DBEUserID` identifies the user of a specific DBE session.

`SYSTEM.COUNTER` is specified to reset ALLBASE/SQL statistical counters.

Description

- Refer to the *ALLBASE/SQL Database Administration Guide* for more information about the SYSTEM.ACCOUNT and SYSTEM.COUNTER views.

Authorization

You must have DBA authority to use this statement.

Example

The I/O resource usage for NewUser's current database session is set to zero.

```
RESET SYSTEM.ACCOUNT FOR USER NewUser
```

RETURN

The RETURN statement permits you to exit from a procedure with an optional return code.

Scope

Procedures only

SQL Syntax

```
RETURN [ReturnStatus];
```

Parameters

ReturnStatus is an integer value that is returned to the caller. The syntax is:

```
{INTEGER
 :LocalVariable
 :ProcedureParameter
 ::Built-inVariable}
```

Description

- The RETURN statement causes the execution of the procedure to halt and causes control to return to the invoking user, application program, or rule. When it returns to a rule, the value of *ReturnStatus* is ignored.
- The RETURN statement is optional within a procedure.
- If the procedure terminates without executing a RETURN statement, the *ReturnStatus* will be 0.
- You can only access *ReturnStatus* from an application program. Call the procedure from the program using an integer host variable for *ReturnStatusVariable* if you wish to test the *ReturnStatus*.

Example

```
CREATE PROCEDURE Process10 (PartName CHAR(20) NOT NULL,
   Quantity INTEGER NOT NULL) AS
   BEGIN
      INSERT INTO SmallOrders VALUES (:PartName, :Quantity);
      IF ::sqlcode <> 0 THEN
         GOTO Errors;
      ENDIF;
      RETURN 0;
      Errors: PRINT 'There were errors.';
      RETURN 1;
   END
```

Call the procedure using a ReturnStatusVariable named Status:

```
EXECUTE PROCEDURE :Status = Process10 ('Widget', 10)
```

On returning from the procedure, test SQLCODE and Status both to determine whether an error occurred inside the procedure.

```
if(sqlca.sqlcode==0)
 if(Status!=0) do {
             EXEC SQL SQLEXPLAIN :SQLMessage;
                printf("%s\n",SQLMessage);
              } while (sqlwarn[0]=='W');
```

REVOKE

The REVOKE statement takes away authority that was previously granted by means of the GRANT statement.The following forms of the REVOKE statement are described individually:

- Revoke table or view authority.
- Revoke RUN or EXECUTE authority.
- Revoke CONNECT, DBA, INSTALL, MONITOR, or RESOURCE authority.
- Revoke SECTIONSPACE or TABLESPACE authority for a DBEFileSet.

For detailed information about security schemes, refer to the "DBEnvironment Configuration and Security" chapter of the *ALLBASE/SQL Database Administration Guide*.

Scope

ISQL or Application Programs

SQL Syntax — Revoke Table or View Authority

```
REVOKE {ALL [PRIVILEGES]
        [SELECT
         INSERT
         DELETE
         ALTER
         INDEX
         UPDATE [({ColumnName}[,...])]
         REFERENCES [({ColumnName}[,...])]]|,...|}
ON {[Owner.]TableName
    [Owner.}ViewName  } FROM {DBEUserID
                              GroupName
                              ClassName
                              PUBLIC     }[,...][CASCADE]
```

Parameters — Revoke Table or View Authority

`ALL [PRIVILEGES]` is the same as specifying the SELECT, INSERT, DELETE, ALTER, INDEX, UPDATE, and REFERENCES options all at one time. The word PRIVILEGES is not required; you can include it if you wish to improve readability. ALTER, INDEX, and REFERENCES are not applied when using REVOKE ALL on views.

`SELECT` revokes authority to retrieve data.

`INSERT` revokes authority to insert rows.

`DELETE` revokes authority to delete rows.

`ALTER` revokes authority to add new columns.

`INDEX` revokes authority to create and drop indexes.

UPDATE — revokes authority to change data in existing rows. A list of column names can be specified to revoke UPDATE authority for only those columns *if* the columns were named in a GRANT statement UPDATE clause. Omitting the list of column names revokes authority to update all columns.

REFERENCES — revokes authority to reference columns in the table from foreign keys in another table. A list of column names can be specified to revoke REFERENCES authority for only those columns *if* the columns were named in a GRANT statement REFERENCES clause. Omitting the list of column names revokes REFERENCES authority on all columns.

[*Owner*.]*TableName* designates the table for which authority is to be revoked.

[*Owner*.]*ViewName* designates the view for which authority is to be revoked.

FROM — The FROM clause designates the users, authorization groups, and classes whose authority is to be revoked. PUBLIC is specified to revoke authority previously granted to PUBLIC. You cannot revoke table or view authorities from the current owner of a table or view.

CASCADE — If the revoked privilege was grantable (granted with the WITH GRANT OPTION clause), then any grants of the privilege by the revokee will also be revoked. However, if a grantee is DBA or owner of an object, cascading stops at that point for the grantee, and any grants and subsequent chains issued by him or her are still in effect. CASCADE can be specified by any user who can revoke authorities on the table or view.

If CASCADE is not specified and you are not DBA, you cannot revoke a grantable privilege if it had been granted to another user (as this would create an orphaned privilege). For more information on privileges, refer to "Using the GRANT OPTION Clause" in the "Database Creation and Security" chapter of the *ALLBASE/SQL Database Administration Guide*..

Description — Revoke Table or View Authority

- If a view relies on a SELECT authority on a table and the REVOKE with CASCADE option is issued against that table, then the view is destroyed and a warning is returned. If the CASCADE option is not specified, the view remains, but you will receive authority errors when you try to use it.
- If a referential constraint relies on a REFERENCES privilege on a table, and the REVOKE REFERENCES with the CASCADE option is issued against that table or column in it, then that particular REFERENCES privilege is destroyed. This can include any REFERENCES in the chain of privileges that are revoked in the CASCADE. A warning is returned when a constraint is destroyed.

Authorization — Revoke Table or View Authority

If you are DBA, the owner, or the grantor of table privileges and still have that grantability, you can issue the REVOKE statement and optionally the CASCADE option.

SQL Syntax — Revoke RUN or EXECUTE or Authority

```
REVOKE [RUN ON [Owner.]ModuleName
        EXECUTE ON PROCEDURE [Owner.] ProcedureName] FROM
{{DBEUserID
  GroupName
  ClassName}[,...]
  PUBLIC          }
```

Parameters--Revoke RUN or EXECUTE Authority

RUN — revokes authority to access the DBEnvironment using the specified module.

[*Owner.*]*ModuleName* specifies the module for which RUN authority is to be revoked.

EXECUTE — revokes authority to execute the specified procedure.

[*Owner.*]*ProcedureName* specifies the procedure for which EXECUTE authority is to be revoked.

FROM — The FROM clause lists the users, authorization groups, and classes that were previously granted the authority that is now to be revoked. PUBLIC can be specified to revoke authority that was previously granted to PUBLIC.

SQL Syntax — Revoke CONNECT, DBA, INSTALL, MONITOR, or RESOURCE Authority

```
REVOKE {CONNECT
        DBA
        INSTALL [AS OwnerID]
        MONITOR
        RESOURCE             }FROM {DBEUserID
                                    GroupName
                                    ClassName}[,...]
```

Parameters — Revoke CONNECT, DBA, INSTALL, MONITOR, or RESOURCE Authority

CONNECT — revokes authority to use the CONNECT statement.

DBA — revokes the authority which exempts a user from all authorization restrictions. You can never revoke DBA authority from the DBECreator.

INSTALL — revokes authority to INSTALL modules where the owner name equals *OwnerID*. If the "AS *OwnerID*" clause is omitted, then revokes authority to INSTALL modules having any owner name.

Modules for an application are created and installed when that application is preprocessed using one of the SQL preprocessors. Modules can also be installed by using the ISQL INSTALL command. See the *ALLBASE/ISQL Reference Manual* for more details.

MONITOR — revokes authority to run SQLMON.

RESOURCE revokes authority to create tables and authorization groups.

FROM The FROM clause specifies the users, authorization groups, and classes whose authority is to be revoked.

Description — Revoke CONNECT, DBA, INSTALL, MONITOR, or RESOURCE Authority

- The REVOKE statement may invalidate stored sections. Refer to the VALIDATE statement and to the *ALLBASE/SQL Database Administration Guide* for additional information on the validation of stored sections.
- Issue a REVOKE INSTALL FROM *DBEUserID* statement that omits the "AS *OwnerID*" clause to remove all INSTALL authorities for a particular user.

Authorization — Revoke CONNECT, DBA, INSTALL, MONITOR, or RESOURCE Authority

If you have OWNER or DBA authority for a module, you can issue REVOKE statements for that module.

SQL Syntax — Revoke DBEFileSet Authority

```
REVOKE {SECTIONSPACE
        TABLESPACE   } |,...| ON DBEFILESET DBEFileSetName FROM
{ {DBEUserID
   GroupName
   ClassName}[,...]
   PUBLIC           }
```

Parameters — Revoke DBEFileSet Authority

SECTIONSPACE revokes authority to store sections in the specified DBEFileSet.

TABLESPACE revokes authority to store table and long column data in the specified DBEFileSet.

DBEFileSetName designates the DBEFileSet for which authority is to be revoked.

Description — Revoke DBEFileSet Authority

- In order for the statement to complete successfully, the authority being revoked must have been previously granted to the specific user. In addition, the DBEFileSet cannot be the current default for that user.
- When SECTIONSPACE authority is revoked, current stored section information for the DBEFileSet remains (and thus any section revalidation continues to use that DBEFileSet). No new sections for the user(s) whose authority was revoked can be placed there.
- When TABLESPACE authority is revoked, table and long column data currently in the DBEFileSet remain there. No new tables or long columns for the user(s) whose authority was revoked can be place there.

- If a REVOKE SECTIONSPACE statement completes successfully, the STOREDSECT table for the specified DBEFileSet is automatically dropped if it is empty and if no other user has SECTIONSPACE authority on the DBEFileSet.
- The execution of this statement causes modification to the HPRDBSS.SPACEAUTH system catalog table. Refer to the *ALLBASE/SQL Database Administration Guide* "System Catalog" chapter.

Authorization — Revoke DBEFileSet Authority

To revoke SECTIONSPACE or TABLESPACE, you must have DBA authority. If you have DBA authority, you can issue the REVOKE statement for any DBEFileSet.

Examples

1. Explicitly revoking authority

   A public table is accessible to any user or program that can start a DBE session. It is also accessible by concurrent transactions.

```
CREATE PUBLIC TABLE PurchDB.Parts
                    (PartNumber  CHAR(16) NOT NULL,
                    PartName  CHAR(30),
                    SalesPrice  DECIMAL(10,2))
                  IN WarehFS

REVOKE ALL PRIVILEGES ON PurchDB.Parts FROM PUBLIC

GRANT SELECT,UPDATE ON PurchDB.Parts TO Accounting
```

   Now only the DBA and members of authorization group Accounting can access the table. Later, the accounting department manager is given control over this table.

```
TRANSFER OWNERSHIP OF PurchDB.Parts TO MgrAccount
```

2. Implicitly revoking authority

   The table is private by default.

```
CREATE TABLE VendorPerf
             (OrderNumber  INTEGER  NOT NULL,
             ActualDelivDay  SMALLINT,
             ActualDelivMonth  SMALLINT,
             ActualDelivYear  SMALLINT,
             ActualDelivQty  SMALLINT
             Remarks  VARCHAR(60) )
          IN Miscellaneous

CREATE UNIQUE INDEX VendorPerfIndex
                 ON VendorPerf (OrderNumber)
```

   Only the table creator and members of authorization group Warehse can update table VendorPerf.

```
GRANT UPDATE ON VendorPerf TO Warehse
```

   The table and the index are both deleted, and the grant is revoked.

```
DROP TABLE VendorPerf
```

3. Using CASCADE

   The DBA grants Clem privileges with the ability to grant them to others. Now Clem has all privileges on the Inventory table as well as the authority to grant any of the privileges to individual users or a class.

   ```
   GRANT ALL
      ON PurchDB.Inventory
      TO Clem WITH GRANT OPTION
   ```

   Clem grants Amanda all privileges on the Inventory table as well as the authority to grant any of the privileges to individual users or a class.

   ```
   GRANT ALL
      ON PurchDB.Inventory
      TO AMANDA WITH GRANT OPTION
   ```

   The DBA revokes privileges from both Clem and Amanda.

   ```
   REVOKE ALL
       ON PurchDB.Inventory
     FROM Clem CASCADE
   ```

4. REVOKE on DBEFileSet

   Revoke from PUBLIC the ability to store sections in DBEFileSet1.

   ```
   REVOKE SECTIONSPACE ON DBEFILESET DBEFileSet1 FROM PUBLIC
   ```

   Revoke from PUBLIC the ability to store tables and long column data in DBEFileSet2.

   ```
   REVOKE TABLESPACE ON DBEFILESET DBEFileSet2 FROM PUBLIC
   ```

5. Revoke INSTALL or MONITOR authority. Revoke from George the ability to run SQLMON.

   ```
   REVOKE MONITOR FROM George;
   ```

   Revoke from Clem the ability to create modules having any owner name.

   ```
   REVOKE INSTALL FROM Clem;
   ```

   Revoke from Clem the ability to create modules owned by JOHN@BROCK.

   ```
   REVOKE INSTALL AS John FROM Clem;
   ```

ROLLBACK WORK

The ROLLBACK WORK statement undoes changes you have made to the DBEnvironment during the current transaction, releases locks held by the transaction, and closes cursors opened during the transaction. Other transactions active in this session are not affected.

Scope

ISQL or Application Programs

SQL Syntax

```
ROLLBACK WORK [TO {SavePointNumber
                   :HostVariable
                   :LocalVariable
                   :ProcedureParameter}
               RELEASE                 ]
```

Parameters

TO — The TO clause is used to roll back to a savepoint without ending the current transaction.

If the TO clause is omitted, ROLLBACK WORK ends the current transaction and undoes any changes that have been made in the transaction.

SavePointNumber is the number assigned by ISQL to a savepoint when you issue the SAVEPOINT statement interactively.

HostVariable is defined as an integer variable to which you assign a value when you issue the SAVEPOINT statement programmatically, or in a procedure.

LocalVariable contains a value in a procedure.

ProcedureParameter contains a value that is passed into or out of a procedure.

RELEASE — terminates your DBE session.

Description

- When you omit the TO clause, all changes you have made to the DBEnvironment since the most recent BEGIN WORK statement are undone. In an application program, all open cursors are automatically closed except those opened with the KEEP CURSOR option. Any savepoints defined in the transaction are lost and become invalid. The transaction is ended. Any cursor opened with the KEEP CURSOR option is repositioned to its scan position as of the most recent BEGIN WORK statement, and a new transaction is implicitly started with the same isolation level.
- The TO clause may not be used if any cursors that were opened with the KEEP CURSOR option are still open. Issuing a ROLLBACK WORK to a savepoint in this context results in an error message, and no rollback is done.
- When you specify the TO clause, all changes you have made to the DBEnvironment

since the designated savepoint are undone. If any cursors opened with the KEEP CURSOR option were active in this transaction, the statement fails and the rollback is not done. In an application program or procedure, all open cursors are automatically closed.

Any savepoints defined more recently than the designated savepoint are lost and become invalid. The designated savepoint is still valid and can be specified in a future `ROLLBACK WORK` statement. The transaction is not ended. Any locks obtained since the savepoint was set are released.

- If the current transaction is the one in which you opened a cursor with the KEEP CURSOR option, then the `ROLLBACK WORK` statement closes the cursor and undoes any changes made through it.
- Under some circumstances ALLBASE/SQL automatically rolls back a transaction. For example, when service is restored after a system failure, all uncommitted transactions are automatically backed out.
- If RELEASE is used, all cursors are closed and the current connection is terminated.
- The RELEASE option is not allowed within a procedure.

Authorization

You do not need authorization to use the `ROLLBACK WORK` statement.

Example

Transaction begins.

```
BEGIN WORK
statement-1
SAVEPOINT :MyVariable
statement-2
statement-3
```

Work of statements 2 and 3 is undone.

```
ROLLBACK WORK
   TO :MyVariable
```

Work of statement 1 is committed; transaction ends.

```
COMMIT WORK
```

12 SQL Statements S - Z

Chapters 10, 11 and 12 describe all the SQL statements in alphabetical order, giving syntax, parameters, descriptions, authorization requirements, and examples for each statement. Examples often consist of groups of statements so you can see how each statement is related to other statements functionally.

SAVEPOINT

The SAVEPOINT statement defines a savepoint within a transaction. DBEnvironment changes made after a savepoint can be undone at any time prior to the end of the transaction. A transaction can have multiple savepoints.

Scope

ISQL or Application Programs

SQL Syntax

```
SAVEPOINT [:HostVariable
          :LocalVariable
          :ProcedureParameter]
```

Parameters

HostVariable identifies an output host variable used to communicate the savepoint number. The host variable's value can be from 1 to (2^{31})–1. In an application program, you must use a host variable with the SAVEPOINT statement. In a procedure, you must use either a local variable or a procedure parameter with the SAVEPOINT statement.

When you enter a SAVEPOINT statement interactively, you cannot specify a host variable. ISQL assigns and displays the savepoint number as follows:

```
isql=> savepoint;
Savepoint number is n.
Use this number to do ROLLBACK WORK to n.
```

LocalVariable contains a value in a procedure. Identifies an output host variable used to communicate the savepoint number. The host variable's value can be from 1 to (2^{31})–1.

ProcedureParameter contains a value that is passed into or out of a procedure. Identifies an output host variable used to communicate the savepoint number. The host variable's value can be from 1 to (2^{31})–1.

Description

- Specify the savepoint number in the TO clause of a ROLLBACK WORK statement to roll back to a savepoint.
- If a procedure invoked by a rule executes a COMMIT WORK statement, an error occurs.

Authorization

You do not need authorization to use the SAVEPOINT statement.

Example

Transaction begins.

```
BEGIN WORK
statement-1
SAVEPOINT :MyVariable
statement-2
statement-3
```

Work of statements 2 and 3 is undone.

```
ROLLBACK WORK
       TO :MyVariable
```

Work of statement-1 is committed; transaction ends.

```
COMMIT WORK
```

SELECT

The SELECT statement retrieves data from one or more tables or views. The retrieved data is presented in the form of a table, called the result table or query result. The explanation of SQL Select syntax is broken down into several levels for easier understanding. An overview of the syntax at each of these levels is presented here starting with the Select Statement Level and continuing through the syntax for the FromSpec.

Detailed discussion of each of these syntax levels is presented in the same order, on the following pages.

Scope

ISQL or Application Programs

SQL Syntax — Select Statement Level

```
[BULK]QueryExpression [ORDER BY {ColumnID [ASC
                                           DESC]}[,...]]
```

SQL Syntax — Subquery Level

```
(QueryExpression)
```

SQL Syntax — Query Expression Level

```
{QueryBlock
 (QueryExpression)}[UNION [ALL]{QueryBlock
                                (QueryExpreession)}][...]
```

SQL Syntax — Query Block Level

```
SELECT [ALL
        DISTINCT] SelectList [INTO HostVariableSpecification]
FROM FromSpec [,...]
 [WHERE SearchCondition1]
 [GROUP BY GroupColumnList]
 [HAVING SearchCondition2]
```

SelectList

```
{*
 [Owner.]Table.*
 Correlation.Name*
 Expression
 [[Owner.]Table.]ColumnName
 CorrelationName.ColumnName}[,...]
```

HostVariableSpecification — With BULK Option

```
:Buffer [,:StartIndex [,:NumberOfRows]]
```

HostVariableSpecification — Without BULK Option

```
{:HostVariable [ [ INDICATOR] :Indicator] ) [,...]
```

FromSpec

```
{TableSpec
 (FromSpec)
 FromSpec NATURAL [INNER
                   LEFT [OUTER]
                   RIGHT [OUTER]] JOIN {TableSpec
                                        (FromSpec)}
 FromSpec [INNER
           LEFT [OUTER]
           RIGHT [OUTER]] JOIN {TableSpec
                                (FromSpec)}{ON SearchCondition3
                                            USING (ColumnList)}}
```

TableSpec

```
[Owner.] TableName [CorrelationName]
```

A SELECT statement can be examined at the following four levels:

Select Statement A select statement is a syntactically complete SQL statement containing one or more SELECT statements but having a single query result that can optionally be sorted with an ORDER BY clause. At its simplest, a select statement is a query expression consisting of a single query block.

Subquery A subquery (also known as a nested query) is a query expression enclosed in parentheses and embedded in a search condition. A subquery returns a value which is used in evaluating the search condition.

Query Expression A query expression is a complex expression consisting of one or more query blocks and UNION/UNION ALL operators.

Query Block A query block is the primary query syntax for specifying which tables to query and which columns to return.

The syntax and usage of each of these levels is described below. For additional information, refer to the chapter "SQL Queries."

SQL Syntax — Select Statement Level

```
[BULK]QueryExpression [ORDER BY {ColumnID [ASC
                                           DESC]}[,...]]
```

Parameters — Select Statement Level

BULK is specified in an application program to retrieve multiple rows with a single execution of the SELECT statement.

Do not use this option in select statements associated with a cursor. Instead, use the BULK option of the FETCH statement.

QueryExpression is a complex expression specifying what is to be selected. The query expression is made up of one or more query blocks, as described in the chapter "SQL Queries."

ORDER BY sorts the result table rows in order by specified columns. Specify the sort key columns in order from major sort key to minor sort key. You can specify as many as 1023 columns. The column specified in the ORDER BY parameter must be one of the columns appearing in the SELECT list. Data is returned in descending order when the ORDER BY *columnID* DESC clause is specified.

For each column you can specify whether the sort order is to be ascending or descending. If neither ASC nor DESC is specified, ascending order is used.

ColumnID must correspond to a column in the select list. You can identify a column to be sorted by giving its name or by giving its ordinal number, with the first column in the select list being column number 1. You must use a column number when referring to columns in the query result that are derived from column expressions. You must also use a column number to refer to columns if the expression contains more than one query block.

The syntax for a column ID in the ORDER BY clause follows:

```
{ColumnNumber
 [[Owner.]TableName.
  CorrelationName.]ColumnName}
```

Description — Select Statement Level

- The SELECT statement is considered updatable if the query expression it contains is updatable and if no ORDER BY clause is present.
- The BULK option cannot be used interactively or in a procedure.
- ALLBASE/SQL uses file space in the defined TempSpaces, and in the system files when processing queries containing ORDER BY clauses or UNION operators. (No such space is used during UNION ALL.)
- When using this statement to select LONG columns, the name of the file is returned in the appropriate field in the *HostVariableSpecification* specified within the *QueryExpression*. With the BULK option, if the output mode is specified with $, then each LONG column in each row accessed has a file with a unique name containing the LONG data retrieved. Additionally, the data file is generated in the directory specified when the LONG column was defined.

SQL Syntax — Subquery Level

(*QueryExpression*)

Parameters — Subquery Level

QueryExpression is the basic syntax of a query or *SELECT* statement. The query expression in a subquery may not contain any UNION or UNION ALL operations.

Description — Subquery Level

- Subqueries are used to retrieve data that is then used in evaluating a search condition. For example, get supplier numbers for the suppliers who supply the maximum quantity of part 'P1'.

```
SELECT SP.SNO
  FROM    SP
 WHERE SP.PNO = 'P1'
   AND SP.QTY = ( SELECT MAX(SP.QTY)
                    FROM SP
                   WHERE SP.PNO = 'P1')
```

  Without using nested queries, the same answer would require the two following queries — one to find the maximum, the other to list the supplier number:

```
SELECT MAX(SP.QTY)
  FROM SP
 WHERE  SP.PNO = 'P1'
```

  and

```
SELECT SP.SNO
  FROM SP
 WHERE SP.PNO = 'P1'
   AND SP.QTY = MaxQty
```

  where *MaxQty* is the result of the first query.

- A subquery may be used only in the following types of predicates:
  - EXISTS predicate.
  - Quantified predicate.
  - IN predicate.
  - Comparison predicate.
- A subquery may be used in the WHERE or HAVING clause of SELECT statements and in the WHERE clause of UPDATE, INSERT, and DELETE statements.
- A subquery may also be nested in the WHERE or HAVING clause of another subquery. No ALLBASE/SQL statement can have more than 16 query blocks within it.
- A subquery may reference a column value in a higher level of the query (or **outer query**). Such a reference is called an **outer reference**. A subquery making an outer reference is called a correlated subquery. Because a correlated subquery depends on a value of the outer query, the subquery must be reevaluated for each new value of the outer query, as in the following example to get supplier numbers for those who supply the most parts for each part number.

```
SELECT SP1.SNO
  FROM SP SP1
 WHERE SP1.QTY = (SELECT MAX(SP2.QTY)
                    FROM SP SP2
                   WHERE SP1.PNO = SP2.PNO)
```

Note that the reference to SP1.PNO in the WHERE clause of the subquery is an outer reference. In this case, because both the outer query and the subquery refer to table SP, correlation names SP1 and SP2 are assigned to make the distinction between the outer and normal references. Within the subquery, any unqualified column names (that is, those which are specified without a table name) are assumed to refer only to tables specified in the FROM clause of that subquery.

- If a query has a HAVING clause with subqueries in it, any outer reference made from those subqueries to the query with the HAVING clause must refer to a column specified in a GROUP BY clause.

SQL Syntax — Query Expression Level

```
{QueryBlock
 (QueryExpression)} [UNION [ALL] {QueryBlock
                                  (QueryExpression}][...]
```

Parameters — Query Expression Level

QueryBlock is the primary query stating which tables to query, which columns to return, and which search conditions to use for filtering data. The query block is further described in one of the next sections.

UNION unites two query expressions into a combined query expression.

The union of two sets is the set of all elements that belong to either or both of the original sets. Because a table is a set of rows, the union of two tables is possible. The resulting table consists of all rows appearing in either or both of the original tables.

ALL indicates that duplicates are not removed from the result table when UNION is specified. If UNION is specified without ALL, duplicates *are* removed.

(*QueryExpression*) may be embedded within another query expression if enclosed in parentheses. Parentheses are optional when a query expression is not embedded.

Description — Query Expression Level

- For the following, assume that T1 is the result of the query block or query expression on the left of the UNION operator, and T2 is the result of the query block or query expression on the right of the UNION operator. (The same conditions must be met if there are additional UNION operators which include results from T3, ...T*n*.):

  — T1 and T2 must have the same number of columns. (They may be derived from tables with varying numbers of columns.)

— The union is derived by first inserting each row of T1 and each row of T2 into a result table and then eliminating any redundant rows unless ALL is specified.

— The result of the union inherits the column names specified for T1.

— The maximum number of query blocks within a query expression is 16.

— Data types of corresponding columns in T1 and T2 must be comparable. When columns are of the same type but of different sizes, the result has the length of the longer of the source columns.

- The ORDER BY clause can specify the ordinal number or the column name of a column in the *leftmost* query expression in a UNION.
- You cannot use LONG columns in a UNION statement except in long string functions.

Table 12-1. shows the conversion rules for comparable data types:

Table 12-1. Conversion Rules for Data in Query Expressions

Data Type	Source Columns	Result Column	Comment
Character	One CHAR, one VARCHAR	VARCHAR	Result has the length of the longer of the two source columns.
	One NATIVE CHAR, one NATIVE VARCHAR	NATIVE VARCHAR	Result has the length of the longer of the two source columns.
	One NATIVE CHAR, one CHAR	NATIVE CHAR	Result has the length of the longer of the two source columns.
	One NATIVE VARCHAR, one CHAR or VARCHAR	NATIVE VARCHAR	Result has the length of the longer of the two source columns.
	One NATIVE CHAR, one VARCHAR	NATIVE VARCHAR	Result has the length of the longer of the two source columns.
	One NATIVE VARCHAR, one VARCHAR	NATIVE VARCHAR	Result has the length of the longer of the two source columns.

Table 12-1. Conversion Rules for Data in Query Expressions

Data Type	Source Columns	Result Column	Comment
Numeric	One FLOAT or REAL	FLOAT	
	Both DECIMAL	DECIMAL	If p1 and s1 are the precision and scale of C1, and p2 and s2 are the precision and scale of C2, the precision and scale of the result column is as follows: `MIN(27, MAX(s1,s2) + MAX(p1-s1, p2-s2))` and the following is the scale of the result column: `MAX(s1,s2)`
	One DECIMAL, one SMALLINT or INTEGER	DECIMAL	Precision and scale are derived as above. The precision and scale for an integer is (10,0); for a smallint, (5,0).
	One INTEGER, one SMALLINT	INTEGER	
Date/Time	Both DATE, TIME, DATETIME, or INTERVAL	DATE, TIME, DATETIME, or INTERVAL, respectively	
	One CHAR or VARCHAR and one DATE, TIME, DATETIME, or INTERVAL	DATE, TIME, DATETIME, or INTERVAL, respectively	
Binary	One BINARY, one VARBINARY	VARBINARY	Result has length of the longer of the two source columns.

SQL Syntax — Query Block Level

```
SELECT [ALL
        DISTINCT] SelectList [INTO HostVariableSpecification]
FROM FromSpec [,...]
 [WHERE SearchCondition1]
 [GROUP BY GroupColumnList]
 [HAVING SearchCondition2]
```

Parameters — Query Block Level

ALL prevents elimination of duplicate rows from the result. If neither ALL nor DISTINCT is specified, the ALL option is assumed.

DISTINCT ensures that each row in the query result is unique. All null values are considered equal. You cannot specify this option if the select list contains an aggregate function with DISTINCT in the argument. This option cannot be used for a select list longer than 255 items. Avoid DISTINCT in subqueries since the query result is not changed, and it hinders rather than helping performance.

SelectList tells how the columns of the result table are to be derived. The syntax of *SelectList* is presented separately below.

INTO The INTO clause defines host variables for holding rows returned in application programs. Do not use this clause for SELECT statements associated with a cursor or dynamically preprocessed SELECT statements, query blocks within subqueries, nested query expressions, or any but the first query block in a SELECT statement.

HostVariableSpecification identifies one or more host variables for holding rows returned in application programs. Do not use this clause for SELECT statements associated with a cursor or dynamically preprocessed SELECT statements, query blocks within subqueries, nested query expressions, or any but the first query block in a SELECT statement. The syntax of BULK and non-BULK types of *HostVariableSpecification* are presented separately below.

FROM The FROM clause identifies the tables and views referenced anywhere in the SELECT statement. The maximum number of tables per query is 31.

FromSpec identifies the tables and views in a query block and explicitly defines inner and outer joins. The syntax of *FromSpec* is presented separately below.

WHERE The WHERE clause determines the set of rows to be retrieved. Rows for which *SearchCondition1* is false or unknown are excluded from processing. If the WHERE clause is omitted, no rows are excluded. Aggregate functions cannot be used in the WHERE clause.

Rows that do not satisfy *SearchCondition1* are eliminated *before* groups are formed and aggregate functions are evaluated.

When you are joining tables or views, the WHERE clause also specifies the condition(s) under which rows should be joined. You cannot join on a column in a view derived using a GROUP BY clause. If you omit a join condition, ALLBASE/SQL joins each row in each table in the FROM clause with each row in all other tables in the FROM clause.

SearchCondition1 may contain subqueries. Each subquery is effectively executed for each row of the outer query and the results used in the application of *SearchCondition1* to the given row. If any executed subquery contains an outer reference to a column of a table or view in the FROM clause, then the reference is to the value of that column in the given row.

Refer to the "Search Conditions" chapter for additional information on search conditions.

GROUP BY The GROUP BY clause identifies the columns to be used for grouping

when aggregate functions are specified in the select list and you want to apply the function to groups of rows. You can specify as many as 1023 columns, unless the select list contains an aggregate function with the DISTINCT option, in which case you can specify as many as 254 columns.

The syntax for the group column list in the GROUP BY clause follows:

```
{ [Owner.]TableName.
   CorrelationName.]ColumnName}[,...]
```

When you use the GROUP BY clause, the select list can contain *only* aggregate functions and columns referenced in the GROUP BY clause. If the select list contains an *, a *TableName.**, or an *Owner.TableName.** construct, then the GROUP BY clause must contain all columns that the * includes. Specify the grouping column names in order from major to minor.

Null values are considered equivalent in grouping columns. If all other columns are equal, all nulls in a column are placed in a single group.

If the GROUP BY clause is omitted, the entire query result table is treated as one group.

HAVING — The HAVING clause specifies a test to be applied to each group. Any group for which the result of the test is false or unknown is excluded from the query result. This test, referred to as *SearchCondition2*, can be a predicate containing either an aggregate function or a column named in the GROUP BY clause.

Each subquery in *SearchCondition2* is effectively checked for each group created by the GROUP BY clause, and the result is used in the application of *SearchCondition2* to the given group. If any executed subquery contains an outer reference to a column, then the reference is to the values of that column in the given group. Only grouping columns can be used as outer references in a subquery in *SearchCondition2*.

SQL Syntax — SelectList

```
{*
 [Owner.]Table.*
 Correlation.Name*
 Expression
 [[Owner.]Table.]ColumnName
 CorrelationName.ColumnName}[,...]
```

Parameters — SelectList

* — includes, as columns of the result table, all columns of all tables and views specified in the FROM clause.

[*Owner.*]*Table.** — includes all columns of the specified table or view in the result.

*CorrelationName.** — includes all columns of the specified table or view in the result. The correlation name is a synonym for the table or view as defined in the FROM clause.

Expression — produces a single column in the result table; the result column values are

computed by evaluating the specified expression for each row of the result table.

The expression can be of any complexity. For example, it can simply designate a single column of one of the tables or views specified in the FROM clause, or it can involve aggregate functions, multiple columns, and so on. When you specify one or more aggregate functions in a select list, the only other entity you can specify is the name(s) of the column(s) you group by.

[[*Owner*.]*Table*.] *ColumnName* includes a particular column from the named owner's indicated table.

CorrelationName. ColumnName includes a specific column from the table whose correlation name is defined in the FROM clause.

SQL Syntax — BULK HostVariableSpecification

```
:Buffer [,:StartIndex [,:NumberOfRows] ]
```

Parameters — BULK HostVariableSpecification

Buffer is a host array or structure that is to receive the output of the SELECT statement. This array contains elements for each column in the *SelectList* and indicator variables for columns that can contain null values. Whenever a column can contain nulls, an indicator variable must be included in the array definition immediately after the definition of that column. The indicator variable can receive the following integer values after a SELECT statement:

0	the column's value is not NULL
–1	the column's value is NULL
> 0	is truncated; the number indicates the data length before truncation

StartIndex is a host variable whose value specifies the array subscript denoting where the first row in the query result should be stored; default is the first element of the array.

Number- OfRows is a host variable whose value specifies the maximum number of rows to store; default is to fill from the starting index to the end of the array.

The total number of rows stored is returned in the SQLERRD[3] field of the SQLCA. (SQLERRD[2] for the C language.)

SQL Syntax — non-BULK HostVariableSpecification

```
{ :HostVariable [INDICATOR]:Indicator] } [,...]
```

Parameters — non-BULK HostVariableSpecification

HostVariable identifies the host variable corresponding to one column in the row.

Indicator names an indicator variable, an output host variable whose value (see following) depends on whether the host variable contains a null value:

0	the column's value is not NULL
–1	the column's value is NULL
> 0	is truncated; the number indicates the data length before truncation

The order of the host variables must match the order of their corresponding items in the select list.

SQL Syntax — FromSpec

```
{TableSpec
 (FromSpec)
 FromSpec NATURAL [INNER
                   LEFT [OUTER]
                   RIGHT [OUTER]] JOIN {TableSpec
                                        (FromSpec)}
 FromSpec [INNER
           LEFT [OUTER]
           RIGHT [OUTER]] JOIN {TableSpec
                                (FromSpec)}{ON SearchCondition3
                                            USING (ColumnList)  } }
```

Parameters — FromSpec

TableSpec identifies a table or view from which rows are selected.

The syntax for a *TableSpec* in a *FromSpec* follows:

[*Owner*.] *TableName* [*CorrelationName*]

[*Owner*.]*TableName* identifies a table or view to be referenced. The *TableName* may be preceded by an *OwnerName*, and may be followed by the definition of a *CorrelationName*.

CorrelationName specifies a synonym for the immediately preceding table or view. The correlation name can be used instead of the actual table or view name anywhere within the SELECT statement when accessing columns or TID values of that table.

The correlation name must conform to the syntax rules for a basic name. All correlation names within one SELECT statement must be unique. They cannot be the same as any table name or view name in the FROM clause that does not also have a correlation name associated with it.

Correlation names are useful when you join a table to itself. You name the table twice in the FROM clause, and assign it two different correlation names.

(*FromSpec*) allows the placement of parentheses around a *FromSpec* in order to alter the order of evaluation of the components of a complex *FromSpec*, such as

one used to describe a three or more table outer join.

NATURAL — indicates that for both inner and outer joins, columns which are common to two tables being joined will be coalesced into a single column when the query result is returned. Also, ALLBASE/SQL will automatically identify and use the columns common to both tables to execute the join. When using the keyword NATURAL you do not use an ON *SearchCondition3* clause or a USING (*ColumnList*) clause to specify the join columns.

INNER — join type indicates that the only rows selected in the join will be those rows for which a match is found in the join column(s) of both tables being joined. If the join type is not specified, INNER is the default.

LEFT — defines the join as a LEFT OUTER JOIN. For a LEFT OUTER JOIN the query result will contain not only the matched rows from both tables being joined, but will also preserve (contain) those rows from the left hand table in the *FromSpec* for which there is no match in the right hand table. The preserved rows are extended to the right with null column values for each column obtained from the right hand table.

For each instance of the keyword JOIN in a *FromSpec*, the named table or the result table immediately preceding JOIN is the left hand table, the named table or the result table immediately following JOIN is the right hand table.

RIGHT — defines the join as a RIGHT OUTER JOIN. For a RIGHT OUTER JOIN the query result will contain not only the matched rows from both tables being joined, but will also preserve (contain) those rows from the right hand table in the *FromSpec* for which there is no match in the left hand table. The preserved rows are extended to the left with null column values for each column obtained from the left hand table.

For each instance of the keyword JOIN in a *FromSpec*, the named table immediately following JOIN is the right hand table, the named table immediately preceding JOIN is the left hand table.

OUTER — is optional as a keyword. If either LEFT or RIGHT are used, the join type is, by default, an outer join.

JOIN — specifies that a join is being defined. Evaluation of the *FromSpec* is from left to right. For a three or more table join, the two tables associated with the left most instance of the JOIN keyword are joined first, and the result of that join is considered the left hand table for the next occurring instance of the keyword JOIN. The same algorithm applies for each additional occurrence of JOIN. Parentheses can be used to force a change in this order of evaluation of the *FromSpec*.

ON *SearchCondition3* may only be used when the keyword NATURAL is not used. Two types of predicates are specified in *SearchCondition3*.

The first type of predicate contains the equality which specifies the join columns to be used for the associated join. For each occurrence in the *FromSpec* of the keyword JOIN, in the ON *SearchCondition3* clause the column names specified on each side of the equality must be fully

qualified.

The second type of predicate limits, for the associated join only, the rows which participate in the inner part of the join. Rows which are excluded from the inner part of the join will be added to those preserved in the outer part of the join. This predicate follows all general rules for search conditions as specified in the "Search Conditions" chapter.

Predicates placed in the ON *SearchCondition3* clause, associated with an instance of JOIN, apply only to that associated inner join. However, predicates placed in the WHERE clause of the SELECT statement apply to the entire query result, after all joins have been evaluated. Therefore you must consider carefully the placement of limiting predicates to decide whether they belong in the WHERE clause, or in an ON *SearchCondition3* clause associated with a particular instance of JOIN in the *FromSpec*. See "Outer Joins" in the "SQL Queries" chapter for specific examples illustrating the changes to the query result brought about by changes in placement of the limiting predicates.

`USING(`*`ColumnList`*`)` specifies participating columns common to both tables being joined, and can only be used if the keyword NATURAL has not been used in the *FromSpec*. The column names must be unqualified because the columns occur in more than one table.

Description — Query Block Level

- The BULK option and INTO clause cannot be used interactively or in procedures.
- The clauses must be specified in the order given in the syntax diagram.
- A result column in the select list can be derived in any of these following ways:
  - A result column can be taken directly from one of the tables or views listed in the FROM clause.
  - Values in a result column can be computed, using an arithmetic expression, from values in a specified column of a table or view listed in the FROM clause.
  - Values in several columns of a single table or view can be combined in an arithmetic expression to produce the result column values.
  - Values in columns of various different tables or views can be combined in an arithmetic expression to produce the result column values.
  - Aggregate functions (AVG, MAX, MIN, SUM, and COUNT) can be used to compute result column values over groups of rows. Aggregate functions can be used alone or in an expression. If you specify more than one aggregate function containing the DISTINCT option, all these aggregate functions must operate on the same column. If the GROUP BY clause is not specified, the function is applied over all rows that satisfy the query. If the GROUP BY clause is specified, the function is applied once for each group defined by the GROUP BY clause. When you use aggregate functions with the GROUP BY clause, the select list can contain *only* aggregate functions and columns referenced in the GROUP BY clause.
  - A result column containing a fixed value can be created by specifying a constant or

an expression involving only constants.

- In addition to specifying how the result columns are derived, the select list also controls their relative position from left to right in the result table. The first result column specified by the select list becomes the leftmost column in the result table.
- The maximum number of columns in a query result is 1024, except when the query contains the DISTINCT option or is within a UNION query expression. In this case, the maximum number of columns is 1023. The maximum number of LONG data type columns which can be directly selected or fetched in a select list is 40. However, any number can be referenced in long string functions. They must be referenced by column name only and cannot participate in an expression in the select list, unless they are being accessed through long string functions.
- Result columns in the select list are numbered from left to right. The leftmost column is number 1. Result columns can be referred to by column number in the ORDER BY clause; this is especially useful if you want to refer to a column defined by an arithmetic expression.
- When you specify the NATURAL....JOIN:
  - You can not use the ON *SearchCondition3* or USING (*ColumnList*) clauses.
  - Each pair of columns with the same column name, which are common to the two tables being joined, will be coalesced into a single common column in the query result. ALLBASE/SQL will automatically determine which columns to use for the join. All columns which have the same column name in each of the tables being joined will be used for the join.
  - When common columns are referenced in the query, such as in the select list, you must use only the unqualified name of the column.
  - Each pair of columns common to two tables being joined must have the same or compatible data types.
  - For a SELECT *, each pair of columns, common to the two tables being joined, will be coalesced into a single common column and will be the first columns displayed in the result, in the order in which they were defined in the left hand table. They will be followed by the columns from the left hand table that were not used for the join. The last columns displayed will be those from the right hand table not participating in the join. Columns not used for the join will be displayed in the order in which they are defined in their respective tables.
  - For any other SELECT, the columns displayed will be those specified in the select list, in the order specified.
  - If there are no common columns between the tables being joined, the columns resulting from the join are the same as the columns that would result from the Cartesian product of the joined tables. See the "SQL Queries" chapter.
- When you specify JOIN....ON *SearchCondition3*:
  - You cannot use the keyword NATURAL or the USING *ColumnList* clause.
  - Column Names from common columns used in the join predicate in *SearchCondition3* must be fully qualified. If additional predicates are used in

SearchCondition3 to limit the rows returned from the join, each column name used must unambiguously reference a column in one of the tables being joined, or must be an outer reference (as in the case of nested subqueries).

- For a SELECT *, the columns contained in the result of the join are the same as the columns of the Cartesian product of the tables being joined.
- For any other SELECT, the columns displayed will be those specified in the select list, in the order specified.
- The result of the INNER JOIN....ON *SearchCondition3* contains the multiset of rows of the Cartesian Product of the tables being joined for which all predicates in *SearchCondition3* are true.

- When you specify JOIN....USING (*ColumnList*):
  - You must not use the keyword NATURAL or the ON *SearchCondition3* clause.
  - You place in the *ColumnList* one unqualified column name for each pair of common columns being used for the join.
  - No column name may be used if it is not common to both tables being joined.
  - For SELECT *, the result of the INNER JOIN....USING (*ColumnList*) contains the multiset of rows of the Cartesian product of the tables being joined for which the corresponding join columns have equal values. The coalesced common columns are returned first. (No duplicate columns are displayed in the case of common columns). The non-join columns from both tables appear next. If there is no common column, the result contains the multiset of rows of the Cartesian product of the tables being joined.
- The result of the [NATURAL] LEFT [OUTER] JOIN is the union of two components. The first component is the result of the equivalent [NATURAL] INNER JOIN. The second component contains those rows in the left hand table that are not in the INNER JOIN result. These rows are extended to the right with null values in the column positions corresponding to the columns from the right hand table. For a natural join, the column values in the common columns are taken from the left hand table.
- The result of the [NATURAL] RIGHT [OUTER] JOIN is the union of two components. The first component is the result of the equivalent [NATURAL] INNER JOIN. The second component contains those rows in the right hand table that are not in the INNER JOIN result. These rows are extended to the left with null values in the column positions corresponding to the columns from the left hand table. For a natural join, the column values in the common columns are taken from the right hand table.
- The ON clause (which is associated with the OUTER JOIN in a join condition) and all predicates in a WHERE clause are filters. At each OUTER JOIN block, the INNER JOIN result (which matches the join condition in an ON clause) will be presented. Then all tuples in the preserving table (which is not in the INNER JOIN result) will be presented by matching columns in the non-preserving table with nulls.
- For three or more table joins, care must be taken when mixing NATURAL....JOIN, JOIN ON *SearchCondition3*, and JOIN USING (*ColumnList*) clauses.
  - The JOIN ON *Searchcondition3* clause produces a result table with the common columns appearing twice, once for each table participating in the join.

  - If this result table is used as input to a NATURAL....JOIN clause or a JOIN USING (*ColumnList*) clause, and the column appearing twice in the result table is named as a join column in the JOIN USING (*ColumnList*) clause or is selected by ALLBASE/SQL as the join column in the NATURAL JOIN, an error will result. This happens because it is impossible to specify which of the two common columns in the result table is to participate in the following join.
  - When writing a three or more table join with explicit join syntax, make sure that for any single result table participating in a join, there are no duplicate column names which will be named as a join column. To ensure this, make each join clause a NATURAL...JOIN or a JOIN...USING (*ColumnList*), except for the final join, which may contain these types or a JOIN...ON *SearchCondition3* clause. Otherwise, ensure that each join clause is a JOIN...ON *SearchCondition3* clause.
- To join tables, without using explicit JOIN syntax, list the tables in the FROM clause, and specify a join predicate in the WHERE clause.
  - If you specify SELECT * and in the WHERE clause an equal predicate specifies the join but there are no other limiting predicates, the result of this procedure is the same as that obtained when using the INNER JOIN described above. The common column appears twice in the query result, once for each table from which it was obtained.
  - If you select each column explicitly, naming each column only once (and appropriately fully qualify a single column name for each pair of column names that is common to both tables) the result is the same as that obtained when using the NATURAL INNER JOIN, above. The common column appears only once in the query result, and is taken from the table specified in the fully qualified column name.
- To join a table with itself, define correlation names for the table in the FROM clause; use the correlation names in the select list and the WHERE clause to qualify columns from that table.
- NULLs affect joins and Cartesian products as follows:
  - Rows are only selected for an inner join when the join predicate evaluates to true. Since the value of NULL is undetermined, the value of the predicate NULL = NULL is unknown. Thus, if the value in the common columns being joined is NULL, the rows involved will not be selected.
  - Rows excluded from the inner part of an outer join because the common column values are NULL, are included in the outer part of the outer join.
  - The existence of NULLs does not exclude rows from being included in a Cartesian product. See the "SQL Queries" chapter for more information.
- When you use the GROUP BY clause, one answer is returned per group, in accord with the select list:
  - The WHERE clause eliminates rows before groups are formed.
  - The GROUP BY clause groups the resulting rows.
  - The HAVING clause eliminates groups.

  - The select list aggregate functions are computed for each group.
- ALLBASE/SQL allocates sort file space in /tmp, by default, or in the space specified using the CREATE TEMPSPACE statement. The space is deallocated once the statement completes.
- The query block is considered updatable if, and only if, it satisfies the following conditions:
  - — No DISTINCT, GROUP BY, or HAVING clause is specified in the outermost SELECT clause, and no aggregates appear in the select list.
  - — No INTO clause is specified.
  - — The FROM clause specifies exactly one table or view (contains no inner or outer joins) and if a view is specified, it is an updatable view.
  - — For INSERT and UPDATE through views, the select list in the view definition must not contain any arithmetic expressions. It must contain only column names.
  - — For DELETE WHERE CURRENT and UPDATE WHERE CURRENT operations, the cursor definition must not contain subqueries.
  - — For noncursor UPDATE, DELETE, or INSERT, the view definition, or the WHERE clause must not contain any subqueries referencing the target table in their FROM clause.

Authorization

If you specify the name of a table, you must have SELECT or OWNER authority for the table, or you must have DBA authority.

If you specify the name of a view, you must have SELECT or OWNER authority for the view, or you must have DBA authority. Also, the owner of the view must have SELECT or OWNER authority with respect to the view's definition, or the owner must have DBA authority.

Examples

1. Simple queries

   One value, the average number of days you wait for a part, is returned.

   ```
   SELECT AVG(DeliveryDays)
     FROM PurchDB.SupplyPrice
   ```

   The part number and delivery time for all parts that take fewer than 20 days to deliver are returned. Multiple rows may be returned for a single part.

   ```
   SELECT PartNumber, DeliveryDays
     FROM PurchDB.SupplyPrice
    WHERE DeliveryDays < 20
   ```

2. Grouping

   The part number and average price of each part are returned.

```
SELECT PartNumber, AVG(UnitPrice)
  FROM PurchDB.SupplyPrice
GROUP BY PartNumber
```

The query result is the same as the query result for the previous SELECT statement, except it contains rows only for parts that can be delivered in fewer than 20 days.

```
SELECT PartNumber, AVG(UnitPrice)
  FROM PurchDB.SupplyPrice
GROUP BY PartNumber
  HAVING MAX(DeliveryDays) < 20
```

3. Joining

This join returns names and locations of California suppliers. Rows are returned in ascending PartNumber order; rows containing duplicate PartNumbers are returned in ascending VendorName order. The FROM clause defines two correlation names (v and s), which are used in both the select list and the WHERE clause. VendorNumber is the only common column between Vendors and SupplyPrice.

```
SELECT PartNumber, VendorName, s.VendorNumber, VendorCity
    FROM PurchDB.SupplyPrice s, PurchDB.Vendors v
   WHERE s.VendorNumber = v.VendorNumber
     AND VendorState = 'CA'
ORDER BY PartNumber, VendorName
```

This query is identical to the query immediately above except that it uses the explicit JOIN syntax.

```
SELECT PartNumber, VendorName, VendorNumber, VendorCity
  FROM PurchDB.SupplyPrice
NATURAL JOIN PurchDB.Vendors
 WHERE VendorState = 'CA'
ORDER BY PartNumber, VendorName
```

This query joins table PurchDB.Parts to itself in order to determine which parts have the same sales price as part 1133-P-01.

```
SELECT q.PartNumber, q.SalesPrice
  FROM PurchDB.Parts p, PurchDB.Parts q
 WHERE p.SalesPrice = q.SalesPrice
   AND p.PartNumber = '1133-P-01'
```

This query does a left outer join between the Vendors and SupplyPrice tables. Since every part supplied by a vendor has an entry in the SupplyPrice table, the result first displays every vendor who supplies a part. The result then displays every vendor who does not supply any parts.

```
SELECT PartNumber, VendorName, VendorCity
  FROM Purchdb.Vendors v
LEFT  JOIN Purchdb.SupplyPrice s
    ON s.VendorNumber = v.VendorNumber
ORDER BY PartNumber, VendorName
```

4. BULK SELECT

Programmatically, when you do not need to use the capabilities associated with a

cursor, you can use the BULK option to retrieve multiple rows.

```
BULK SELECT *
       INTO :Items, :Start, :NumRow
       FROM PurchDB.Inventory
```

5. UNION Option

   Retrieves all rows from two Parts tables into a single query result ordered by PartNumber. PartNumber and PartValue are comparable; SalesPrice and Price are comparable.

```
  SELECT PartNumber, SalesPrice
    FROM P1988.Parts
   UNION
  SELECT PartValue, Price
    FROM P1989.Parts
ORDER BY PartNumber
```

6. Nested query or subquery

   Obtain a list of customer orders whose totals are higher than the largest order of 1988.

```
  SELECT OrderNumber, SUM(PurchasePrice)
    FROM PurchDB.OrderItems
GROUP BY OrderNumber
  HAVING SUM(PurchasePrice) > (SELECT MAX(PurchasePrice)
                                 FROM FY1988.Orders)
```

   Get vendor numbers for all vendors located in the same city as vendor number 9005.

```
SELECT VendorNumber
  FROM PurchDB.Vendors
 WHERE VendorCity = (SELECT VendorCity
                       FROM PurchDB.Vendors
                       WHERE VendorNumber = '9005')
```

   Get supplier names for suppliers who provide at least one red part.

```
SELECT SNAME
  FROM S
 WHERE SNO IN ( SELECT SNO
                  FROM SP
                 WHERE EXISTS (SELECT PNO
                                 FROM P
                                WHERE P.PNO = SP.PNO
                                  AND COLOR = 'RED' ))
```

   Get supplier number for suppliers who supply the most parts.

```
  SELECT SNO
    FROM SP
GROUP BY SNO
  HAVING COUNT(DISTINCT PNO) >= ALL ( SELECT COUNT(DISTINCT PNO)
                                        FROM SP
                                    GROUP BY SNO )
```

Insert into table T, supplier names of each supplier who does not supply any part.

```
INSERT INTO T (SNO)
     SELECT SNO
       FROM S
      WHERE NOT EXISTS (SELECT *
                          FROM SP
                         WHERE SP.SNO = S. SNO)
```

Delete all suppliers from the supplier table who do not supply any parts.

```
DELETE FROM S
 WHERE NOT EXISTS ( SELECT *
                      FROM SP
                     WHERE SP.SNO = S.SNO)
```

SET CONNECTION

The SET CONNECTION statement sets the current connection within the list of connected DBEnvironments. Any SQL statements issued apply to the current connection.

Scope

ISQL or Application Programs

SQL Syntax

```
SET CONNECTION {'ConnectionName'
                :HostVariable   }
```

Parameters

ConnectionName is a string literal identifying the name associated with this connection. This name must be unique for each DBEnvironment connection within an application or an ISQL session. *ConnectionName* cannot exceed 128 bytes.

HostVariable is a character string host variable containing the *ConnectionName* associated with this connection.

Description

- A connection to any one of the list of connected DBEnvironments can be the current connection. When the current connection is set from one DBEnvironment to another, any previously connected DBEnvironment is said to be suspended.
- If a previously suspended DBEnvironment connection again becomes the current connection, all DBEnvironment context information for the current connection is restored to the same state as at the time when the DBEnvironment was suspended.
- A connection with the DBEnvironment referenced in this statement must have previously been established using either a CONNECT, START DBE, START DBE NEW, or START DBE NEWLOG statement. This connection must not have been terminated by a DISCONNECT, RELEASE, or STOP DBE statement.
- No stored section is created for the SET CONNECTION statement. SET CONNECTION cannot be used with the PREPARE or EXECUTE IMMEDIATE statements or procedures.
- An active transaction is not required to execute a SET CONNECTION statement. An automatic transaction will *not* be started when executing a SET CONNECTION statement.

Authorization

You do not need authorization to use the SET CONNECTION statement.

Example

Establish two connections:

```
CONNECT TO :PartsDBE AS 'Parts1'
CONNECT TO :SalesDBE AS 'Sales1'
```

At this point, Sales1 is the current connection.

.
.
.

Set the current connection to Parts1:

```
SET CONNECTION 'Parts1'
```

SET CONSTRAINTS

The SET CONSTRAINTS statement sets the UNIQUE, REFERENTIAL or CHECK constraint error checking mode.

Scope

ISQL or Application Programs

SQL Syntax

```
SET ConstraintType [,...]CONSTRAINTS {DEFERRED
                                      IMMEDIATE}
```

Parameters

ConstraintType identifies the type of constraint that is to be affected by the statement. Each *ConstraintType* can be one of the following:

UNIQUE

REFERENTIAL

CHECK

DEFERRED specifies that constraint error violations are not checked until the constraint checking mode is reset to IMMEDIATE, or the current transaction ends.

IMMEDIATE specifies that constraint errors are checked at the level set by the SET DML ATOMICITY statement, when the SET CONSTRAINTS IMMEDIATE statement successfully executes. This is the default constraint error checking mode.

Description

- Setting constraint checking to DEFERRED does not defer checking of non-constraint errors. They are still checked at the current level specified by the SET DML ATOMICITY statement.
- When you use SET CONSTRAINTS DEFERRED, error checking for constraint violations is not enabled until you either SET CONSTRAINTS IMMEDIATE or end the transaction with a COMMIT WORK.
- You can set the constraint error checking mode to IMMEDIATE at any time in the flow of processing.
- When you set constraint checking to IMMEDIATE, and constraint errors currently exist, the SET CONSTRAINTS statement does not succeed. The constraint violations cause an error message to be issued and constraint checking to remain deferred.
- You have the option of correcting the error before issuing a COMMIT WORK or allowing

the COMMIT WORK statement to be executed.

  - If errors remain when you COMMIT WORK, no matter to what level DML atomicity is set, error checking is done at the *transaction level* and *the entire transaction* will be rolled back.
  - When no constraint errors exist, SET CONSTRAINTS IMMEDIATE succeeds, and error checking thereafter occurs at the level in effect from the SET DML ATOMICITY statement.

- If constraint checking is set to DEFERRED and you again set it to DEFERRED, a warning message is issued. If constraint checking is set to IMMEDIATE and you again set it to IMMEDIATE, a warning message is issued.
- COMMIT WORK and ROLLBACK WORK statements both reset constraint checking to IMMEDIATE.
- The SET CONSTRAINTS statement is sensitive to savepoints. If you establish a save point, then change the constraint checking mode, and then roll back to the savepoint, the constraint mode set after the savepoint will be undone.
- When UNIQUE is specified as a *ConstraintType*, unique indexes are checked for errors also.
- HASH unique constraint checking cannot be deferred. Refer to the CREATE TABLE statement for information on HASH unique constraints.
- View check constraint checking cannot be deferred.
- The SET CONSTRAINTS statement affects only the current session.
- The current setting does not appear in the ISQL LIST SET command.

Authorization

Anyone can issue a SET CONSTRAINTS statement.

Example

```
BEGIN WORK
```

Constraints are deferred so that the insert and update statements will succeed even though they have unresolved constraint errors. By the end of the transaction, the constraint errors must be resolved or the entire transaction is rolled back.

```
SET REFERENTIAL CONSTRAINTS DEFERRED
```

A transaction appears here that contains some insert, update, and delete statements:

```
INSERT ...
UPDATE ...
DELETE ...
UPDATE ...
UPDATE ...
```

If there are unresolved referential constraints, an error message appears and constraint checking remains in the deferred mode.

```
SET REFERENTIAL CONSTRAINTS IMMEDIATE
```

You can correct the constraint errors so you can successfully COMMIT WORK.

If you do not, the COMMIT WORK will roll back the entire transaction because of the remaining violations. Issue error correction statements, here.

Constraint error checking is set to IMMEDIATE by the COMMIT WORK statement or a ROLLBACK WORK statement.

```
COMMIT WORK
```

SET DEFAULT DBEFILESET

The SET DEFAULT statement is used to set the default DBEFileSet for stored sections or for tables and long columns associated with a DBEnvironment. Before initial issue of this statement, the SYSTEM DBEFileSet is the default.

Scope

ISQL or Application Programs

SQL Syntax

```
SET DEFAULT {SECTIONSPACE
             TABLESPACE   } TO DBEFILESET DBEFileSetName FOR PUBLIC
```

Parameters

SECTIONSPACE sets the default DBEFileSet for stored sections.

TABLESPACE sets the default DBEFileSet for tables and long columns.

DBEFileSetName designates the DBEFileSet for which the default is to be set.

Description

- PUBLIC must have the appropriate authority on the specified DBEFileSet. (Refer to syntax for the GRANT statement.)
- You can grant SECTIONSPACE or TABLESPACE authority for a DBEFileSet to a specific user, thereby giving that user the ability to explicitly put sections, tables, or long columns in the granted DBEFileSet when they are created. However, you cannot set a default DBEFileSet for a specific user.
- If a section is created without the IN *DBEFileSet* clause, or if the owner of the section does not have SECTIONSPACE authority for the DBEFileSet specified when the section was created, the section is stored in the default SECTIONSPACE DBEFileSet for PUBLIC. This applies to rules, stored procedures, check constraints, views, and prepared or preprocessed statements and cursors, all of which have sections associated with them.
- If a table is created without the IN *DBEFileSet* clause, or if the owner of the table does not have TABLESPACE authority for the DBEFileSet specified when the table was created, the table is placed in the default TABLESPACE DBEFileSet for PUBLIC.
- If a long column is created without the IN *DBEFileSet* clause, it is placed in the same DBEFileSet as the table unless the owner of the table does not have TABLESPACE authority for the DBEFileSet the table resides in. In this case, the long column is placed in the default TABLESPACE DBEFileSet for PUBLIC.

Authorization

You must have DBA authority to set a DBEFileSet default.

Example

Set Default DBEFileSet

```
GRANT SECTIONSPACE ON DBEFILESET SectionDBESet to PUBLIC;

GRANT TABLESPACE ON DBEFILESET TableDBESet to PUBLIC;

SET DEFAULT SECTIONSPACE TO DBEFILESET SectionDBESet FOR PUBLIC;

SET DEFAULT TABLESPACE TO DBEFILESET TableDBESet FOR PUBLIC;
```

SET DML ATOMICITY

The SET DML ATOMICITY statement sets the general error checking level in data manipulation statements.

Scope

ISQL or Application Programs

SQL Syntax

```
SET DML ATOMICITY AT {ROW
                      STATEMENT} LEVEL
```

Parameters

ROW	specifies that general error checking occurs at the row level. The term **general error checking** refers to any errors, for example, arithmetic overflows or constraint violation errors.
STATEMENT	specifies that general error checking occurs at the statement level. This is the default general error checking level.

Description

- **Constraint errors** (UNIQUE, REFERENTIAL, or CHECK constraint violations) are handled just like any other general error when constraint checking is in IMMEDIATE mode. In this case, error handling follows the behavior outlined below. However, when you SET CONSTRAINTS DEFERRED, constraint error checking behaves differently as described in the SET CONSTRAINTS statement in this chapter. The following discussion assumes that constraint checking is in IMMEDIATE mode.
- Setting DML ATOMICITY affects the BULK INSERT, DELETE, UPDATE, UPDATE WHERE CURRENT, DELETE WHERE CURRENT statements, and the ISQL LOAD command when they operate on a *set* of rows.
- When you use SET DML ATOMICITY AT STATEMENT LEVEL (the default), and if an error occurs:
  - Work done by the statement before an error occurs *is undone*, and the statement is no longer in effect.
  - At COMMIT WORK, work done by statements within the transaction that executed without error will be written to the DBEnvironment, while statements with errors will have no effect.
- When you use SET DML ATOMICITY AT ROW LEVEL (not the default), and if an error occurs:

  — Work done by a statement before an error occurs *is not undone*, but no further action is taken by the statement.

— At COMMIT WORK, work done by statements within the transaction that executed without error will be written to the DBEnvironment. Within statements which generated errors at a specific row, work done on rows prior to the row generating the error will be written to the DBEnvironment; no work will be done from the erroneous row, forward.

- Unless you have a severe error (4008, 4009, or -14024 or greater), the transaction is not rolled back, and previous statements within the transaction are still in effect.
- When a transaction ends, DML ATOMICITY remains at or is returned to STATEMENT level.
- The SET DML ATOMICITY statement is sensitive to savepoints. If you establish a save point, then change the atomicity level, and then roll back to the savepoint, the atomicity level set after the savepoint will be undone.
- If DML ATOMICITY is set at ROW and you set it to ROW again, a warning message is issued. If DML ATOMICITY is set at STATEMENT and you set it to STATEMENT again, a warning message is issued.
- DML ATOMICITY does not apply to DDL statements. DDL statements are always checked at statement level.
- DML ATOMICITY does not apply to statements that may fire rules. Such statements are always checked at statement level.
- When the SET CONSTRAINTS statement sets constraint error checking to IMMEDIATE, constraint error checking will be performed at the level set by the most recent SET DML ATOMICITY statement. Refer to the SET CONSTRAINTS statement for more information.

Authorization

Anyone can use the SET DML ATOMICITY statement.

Example

The user wants to load supposedly error-free data into PurchDB.Parts.

```
BEGIN WORK
```

Immediately after DBEnvironment creation, when initially loading the tables while non-archive mode logging is in effect, performance can be improved if you SET DML ATOMICITY to ROW LEVEL. However, if an error is encountered, the insertion of rows prior to the erroneous row will not be rolled back.

Error checking is set at row level.

```
SET DML ATOMICITY AT ROW LEVEL
```

The rows to be inserted are in the array called PartsArray.

```
BULK INSERT INTO PurchDB.Parts
          VALUES (:PartsArray, :StartIndex, :NRows)
```

You can set the level back to statement level before the transaction ends.

```
.
.
.
Other statements are listed here.
.
.
.
COMMIT WORK
```

If you have not already set error checking back to statement level, it is automatically set back to statement level when the transaction ends.

SET MULTITRANSACTION

When you are using multiconnect functionality, the SET MULTITRANSACTION statement provides the capability of switching between single-transaction mode and multitransaction mode.

Scope

ISQL or Application Programs

SQL Syntax

```
SET MULTITRANSACTION {ON
                      OFF}
```

Parameters

ON	enables multiple implied or explicit BEGIN WORK statements to be active across the set of connected DBEnvironments. This is termed **multitransaction mode**.
OFF	permits one implied or explicit BEGIN WORK statement to be active across the set of connected DBEnvironments. This is termed **single-transaction mode**. This is the default.

Description

- A given SET MULTITRANSACTION statement is in effect until another such statement is issued or until the application (or ISQL) terminates.
- Single-transaction mode is the default.
- While in single-transaction mode, the SET MULTITRANSACTION ON statement is always valid.
- While in multitransaction mode, the SET MULTITRANSACTION OFF statement is valid only if no more than one transaction is active. If an active transaction exists, it must be in the currently connected DBEnvironment, otherwise the SET MULTITRANSACTION OFF statement will be rejected and an error will be generated.
- No stored section is created for the SET MULTITRANSACTION statement. SET MULTITRANSACTION cannot be used with the PREPARE or EXECUTE IMMEDIATE statements or in procedures.
- An active transaction is not required to execute a SET MULTITRANSACTION statement. An automatic transaction will *not* be started when executing a SET MULTITRANSACTION statement.

Authorization

You do not need authorization to use the SET MULTITRANSACTION statement.

Example

Put single-transaction mode in effect:

```
SET MULTITRANSACTION OFF
```

Put multitransaction mode in effect:

```
SET MULTITRANSACTION ON
```

SETOPT

The SETOPT statement modifies the access optimization plan used by queries.

Scope

ISQL or Application Programs

Syntax — SETOPT

```
SETOPT {CLEAR
        GENERAL {ScanAccess
                 JoinAlgorithm}[,...]
        BEGIN {GENERAL {ScanAccess
                        JoinAlgorithm}}[,...] END
```

Syntax — Scan Access

```
[NO]{SERIALSCAN
     INDEXSCAN
     HASHSCAN
     SORTINDEX}
```

Syntax — Join Algorithm

```
[NO] {NESTEDLOOP
      NLJ
      SORTMERGE
      SMJ}
```

Parameters

CLEAR	specifies that the access plan set by any previous SETOPT statement is to be cleared.
SERIALSCAN	specifies serial scan access.
INDEXSCAN	indicates index scan access for those tables with indexes.
HASHSCAN	designates hash scan access for tables with hash structures.
SORTINDEX	indicates index scan access when an ORDER BY or GROUP BY clause is specified in a SELECT statement. Therefore, the extra sort operation is eliminated. The index scanned is the one defined upon the column referenced in the ORDER BY or GROUP BY clause.
NESTEDLOOP	specifies nested loop joins.
NLJ	is equivalent to NESTEDLOOP.
SORTMERGE	designates sort merge join.
SMJ	is equivalent to SORTMERGE.

Description

- Use the SETOPT statement when you want to override the default access plan used in queries.
- The SETOPT statement affects only those queries in the current transaction. When the transaction ends, the settings specified by SETOPT are cleared.
- To view the plan specified by SETOPT, query the SYSTEM.SETOPTINFO view.
- Use the GENPLAN command in ISQL to display the current access plan.
- NLJ is equivalent to NESTEDLOOP, and SMJ is equivalent to SORTMERGE.
- To store a user defined access plan in a module or procedure, run ISQL and issue the SETOPT statement followed by a VALIDATE statement.
- To remove the access plan specified by a SETOPT statement from a module or procedure, execute the VALIDATE statement with the DROP SETOPTINFO option.
- When using the EXTRACT command in ISQL, specify the NO SETOPTINFO option if you want to prevent the access plan specified by a SETOPT statement from being included in the installable module file.
- Use the GENPLAN command in ISQL to see the optimizer's access plan for an ALLBASE/SQL statement.
- For more information on joins, see "Join Methods" in the *ALLBASE/SQL Performance and Monitering Guidelines*.

Authorization

You do not need authorization to use the SETOPT statement.

Examples

In the following example, the SETOPT statement specifies that all tables with indexes are accessed with an index scan. Since PurchDB.Parts has an index defined upon the PartNumber column, an index scan is executed by the first SELECT statement. The effect of a SETOPT statement lasts only until the end of the transaction. Therefore, the second SELECT statement may, or may not, use an index scan.

```
BEGIN WORK
SETOPT GENERAL INDEXSCAN
SELECT * FROM PurchDB.Parts
COMMIT WORK

BEGIN WORK
SELECT * FROM PurchDB.Parts
COMMIT WORK
```

The next SETOPT statement indicates that hash scans are not to be performed.

```
SETOPT GENERAL NO HASHSCAN
```

The following two SETOPT statements are equivalent.

```
SETOPT GENERAL HASHSCAN, NO SORTMERGE

SETOPT BEGIN
       GENERAL HASHSCAN;
       GENERAL NO SORTMERGE;
       END
```

In the following two SELECT statements, an index scan is performed upon the PartNumber because the PartNumber column is referenced in the ORDER BY and GROUP BY clauses.

```
SETOPT GENERAL SORTINDEX

SELECT   PartNumber, UnitPrice
FROM     PurchDB.SupplyPrice
ORDER BY PartNumber, UnitPrice

SELECT   PartNumber, AVG (UnitPrice)
FROM     PurchDB.SupplyPrice
GROUP BY PartNumber
```

After the following sequence of statements is executed, all of the modules stored in the DBEnvironment will use an index scan when accessing tables with indexes. The cex09 module is an exception, however, because it is validated with the DROP SETOPTINFO keywords. When the cex03 module is copied into the installable module file with the EXTRACT command, the index scan specified by the SETOPT statement is not included in the installable module file.

```
SETOPT GENERAL INDEXSCAN
VALIDATE ALL MODULES
SETOPT CLEAR
VALIDATE DROP SETOPTINFO MODULE cex09
EXTRACT MODULE cex03 NO SETOPTINFO INTO Modfile
```

SET PRINTRULES

The SET PRINTRULES statement specifies whether rule names and statement types are to be issued as messages when the rules are fired during a DBEnvironment session.

Scope

ISQL or Application Programs

SQL Syntax

```
SET PRINTRULES [ON
                OFF]
```

Parameters

ON	specifies that rule name and statement type should be issued as a message when the rule is fired.
OFF	specifies that rule name and statement type should not be issued as a message when the rule is fired. This is the default for all sessions.

Description

- SET PRINTRULES OFF returns the DBEnvironment session to its default behavior of not issuing messages with rule names and statement types as rules fire.
- SET PRINTRULES ON causes rule names and statement types (INSERT, DELETE, UPDATE) to be issued as messages in the current DBEnvironment session until the session completes or a SET PRINTRULES OFF statement is executed.
- SET PRINTRULES ON has no effect if rule printing is already on in the DBEnvironment.
- SET PRINTRULES OFF has no effect if rule printing is already off in the DBEnvironment.
- The statement only affects the current SID (session id). Other users are not affected.
- Rule names are printed by issuing an informative message DBWARN 2021, with the following text:

  Rule *Owner.RuleName* fired on *StatementType* statement.

  StatementType is one of the following:

  INSERT

  UPDATE

  DELETE

- The effects of this statement are not undone by a ROLLBACK WORK or COMMIT WORK statement.

Authorization

You must have DBA authority.

Example

The DBA enables the issuing of messages when rules fire.

```
SET PRINTRULES ON
```

The DBA issues statements that fire rules.

```
INSERT INTO PurchDB.Parts VALUES (9213, 'Widget', 12.95)

Rule PurchDB.InsertParts fired on INSERT statement. (DBWARN 2021)
```

The DBA disables the issuing of messages when rules fire.

```
SET PRINTRULES OFF
```

SET SESSION

The SET SESSION statement sets one or more transaction attributes for the duration of a session to be applied to the next and subsequent transactions. These attributes include: isolation level, priority, user label, constraint checking mode, DML atomicity level, timeout rollback, user timeout, termination level, and fill options.

Scope

ISQL or Application Programs

SQL Syntax

```
SET SESSION {ISOLATION LEVEL {RR
                              CS
                              RC
                              RU
                              REPEATABLE READ
                              SERIALIZABLE
                              CURSOR STABILITY
                              READ COMMITTED
                              READ UNCOMMITTED
                              :HostVariable1    }
            PRIORITY {Priority
                      :HostVariable2}
            LABEL {'LabelString'
                   :HostVariable3}
            ConstraintType [,...]CONSTRAINTS {DEFERRED
                                              IMMEDIATE}
            DML ATOMICITY AT {STATEMENT
                              ROW       }LEVEL
            ON {TIMEOUT
                DEADLOCK}ROLLBACK {QUERY
                                   TRANSACTION}
            USER TIMEOUT [TO] {DEFAULT
                               MAXIMUM
                               TimeoutValue[{SECONDS
                                             MINUTES} ]
                               :HostVariable4[{SECONDS
                                               MINUTES} ] }
            TERMINATION AT {SESSION
                            TRANSACTION
                            QUERY
                            RESTRICTED    }LEVEL
            [{PARALLEL
              NO       }]FILL                                  }[,...]
```

Parameters

RR — Repeatable Read. Means that the transaction uses locking strategies to guarantee repeatable reads.

RR is the default isolation level.

CS Cursor Stability. Means that your transaction uses locking strategies to assure cursor-level stability only.

RC Read Committed. Means that your transaction uses locking strategies to ensure that you retrieve only rows that have been committed by some transaction.

Read Uncommitted. Means that the transaction reads data without obtaining additional locks.

Use the RU isolation level in applications in which the reading of uncommitted data is not of concern.

REPEATABLE READ Same as RR.

SERIALIZABLE Same as RR.

CURSOR STABILITY Same as CS.

READ COMMITTED Same as RC.

READ UNCOMMITTED Same as RU.

HostVariable1 is a string host variable containing one of the isolation level specifications above.

Priority is an integer from 0 to 255 specifying the priority of the transaction. Priority 127 is the default. ALLBASE/SQL uses the priority to resolve a deadlock. The transaction with the largest priority number is aborted to remove the deadlock.

For example, if a priority-0 transaction and a priority-1 transaction are deadlocked, the priority-1 transaction is aborted. If two transactions involved in a deadlock have the same priority, the deadlock is resolved by aborting the newer transaction (the last transaction begun, either implicitly or with a BEGIN WORK statement).

HostVariable2 is an integer host variable containing the priority specification.

LabelString is a user defined character string of up to 8 characters. The default is a blank string.

The label is visible in the SYSTEM.TRANSACTION pseudo-table and also in SQLMON. Transaction labels can be useful for troubleshooting and performance tuning. Each transaction in an application program can be marked uniquely, allowing the DBA to easily identify the transaction being executed by any user at any moment.

HostVariable3 is a string host variable containing the *LabelString*.

ConstraintType identifies the types of constraints that are affected by the DEFERRED and IMMEDIATE options. Each *ConstraintType* can be one of the following:

```
UNIQUE
REFERENTIAL
CHECK
```

DEFERRED specifies that constraint errors are not checked until the constraint

checking mode is reset to IMMEDIATE or the current transaction ends.

`IMMEDIATE` specifies that constraint errors are checked when a statement executes. This is the default.

`STATEMENT` specifies that error checking occurs at the statement level. This is the default.

`ROW` specifies that error checking occurs at the row level.

`QUERY` sets the action for timeouts or deadlocks to rollback the statement or query.

`TRANSACTION` sets the action for timeouts or deadlocks to rollback the transaction.

`DEFAULT` specifies to use the default timeout duration for the DBE specified in the `START DBE` statement.

`MAXIMUM` specifies to use the maximum timeout duration for the DBE specified in the `START DBE` statement.

`TimeoutValue` specifies the timeout duration to use in seconds or minutes.

`:HostVariable4` is an integer host variable specifying the timeout duration to use in seconds or minutes.

`SESSION` specifies self-termination at the session level, and allows external termination at the session level only.

`TRANSACTION` specifies self-termination at the transaction level, and allows external termination at the session or transaction level.

`QUERY` specifies self-termination at the query level, and allows external termination at the session, transaction, or query level.

`RESTRICTED` specifies no self-termination, and allows external termination at the session level only. This is the default.

`FILL` is used to optimize I/O performance when loading data and creating indexes.

`PARALLEL FILL` is used to optimize I/O performance for multiple, concurrent loads to the same table. The PARALLEL FILL option must be in effect for each load.

`NO FILL` turns off the FILL or PARALLEL FILL option for the duration of the session. This is the default fill option.

Description

- Detailed information about isolation levels is presented in the "Concurrency Control through Locks and Isolation Levels" chapter.
- You can issue the `SET SESSION` statement at any point in an application or ISQL session. Whether issued within or outside of a transaction, the attributes specified in a `SET SESSION` statement apply to the next and subsequent transactions.
- Any attribute specified in a `SET SESSION` statement remains in effect until the session terminates unless reset by another statement. See the "Using ALLBASE/SQL" chapter, "Scoping of Transaction and Session Attributes" section for information about

statements used to set transaction attributes.

- When using RC or RU, you should verify the existence of a row before you issue an UPDATE statement. In application programs that employ cursors, you can use the REFETCH statement prior to updating. REFETCH is not available in ISQL. Therefore, you should use caution in employing RC and RU in ISQL if you are doing updates.
- If the FILL or PARALLEL FILL option has already been set for the session with a SET SESSION statement, and you do not want either of these options in effect for a given transaction, specify NO FILL in the transaction's BEGIN WORK statement.
- As with the SET CONSTRAINTS statement, the SET SESSION statement allows you to set the UNIQUE, REFERENTIAL or CHECK constraint error checking mode. If the constraint checking mode is deferred, checking of constraints is deferred until the end of a transaction or until the constraint mode is set back to immediate. If the constraint mode is immediate, integrity constraints are checked following processing of each SQL statement (if statement level atomicity is in effect) or each row (if row level atomicity is in effect). Refer to the SET DML ATOMICITY statement in this chapter for further information on statement and row level error checking. The following paragraph assumes that statement level atomicity is in effect.

  When constraint checking is deferred, a COMMIT WORK, or SET CONSTRAINTS IMMEDIATE statement executes if zero constraint violations exist at that time, otherwise a constraint error is reported. When constraint checking is immediate (the default), zero constraint violations must exist when an SQL statement executes, otherwise a constraint error is reported and the statement is rolled back. The SET CONSTRAINTS statement in this chapter gives further detail about constraint checking.
- As with the SET DML ATOMICITY statement, the SET SESSION statement allows you to set the general error checking level in data manipulation statements. General error checking refers to any errors, for example, arithmetic overflows or constraint violation errors.

  Setting ROW LEVEL atomicity guarantees that internal savepoints are not generated. For example, if an error occurs on the *n*th row of a bulk statement such as LOAD, BULK INSERT, or Type2 INSERT, the row is not processed, statement execution terminates, and any previously processed rows are *not* rolled back. In contrast, STATEMENT LEVEL atomicity guarantees that the entire statement is rolled back if it does not execute without error. STATEMENT LEVEL atomicity is the default. Refer to the SET DML ATOMICITY statement in this chapter for further information on statement and row level error checking.
- In contrast to the SET TRANSACTION statement, transaction attributes set within a transaction by a SET SESSION statement are *not* sensitive to savepoints. That is, if you establish a savepoint, then issue the SET SESSION statement to change attribute(s)for the session, and then roll back to the savepoint, the transaction attribute(s) set after the savepoint are *not* undone. In this case, the attribute(s) would go into effect for the next and subsequent transactions, just as if no rollback to savepoint had occurred. See Chapter 2 , “Using ALLBASE/SQL,” "Scoping of Transaction and Session Attributes" section for information about statements used to set transaction attributes.
- The SET SESSION statement is not allowed within a stored procedure.
- When ON TIMEOUT ROLLBACK or DEADLOCK ROLLBACK is set to

TRANSACTION, the whole transaction is aborted as a result of a timeout or deadlock.

- When ON TIMEOUT ROLLBACK or DEADLOCK ROLLBACK is set to QUERY, only the SQL statement which has timed out will be rolled back. This means rolling back results of statements that modify the database and closing cursor for the cursor-related statements. (Cursor-related statements change the cursor position, and are not statements like UPDATE or DELETE WHERE CURRENT.)
- In general, if a transaction with KEEP cursor(s) is committed, the new transaction started on behalf of the user inherits the most recent transaction attributes of the old transaction. The KEEP cursor(s) are an exception; they inherit the isolation level attribute of the old transaction at the time the cursor(s) were opened. Note, however, that session isolation level is *not* used for keep cursor transactions. Session isolation level does not take effect until KEEP cursors are closed, the transaction is committed, and the next transaction is begun. For example:

```
.
.
.
BEGIN WORK RC
.
.
.
OPEN C1 KEEP CURSOR
.
.
.
SET TRANSACTION ISOLATION LEVEL CS
.
.
.
OPEN C2 KEEP CURSOR
.
.
.
SET TRANSACTION ISOLATION LEVEL RU
.
.
.
SET SESSION ISOLATION LEVEL CS
.
.
.
COMMIT WORK
.
.
.
OPEN C3               Session isolation level does not take effect.
.
.
.
CLOSE C1
CLOSE C2
CLOSE C3
.
.
.
```

```
COMMIT WORK
BEGIN WORK            Session isolation level CS takes effect.
.
.
.
```

In the above example, the new transaction started on behalf of the user after the first COMMIT WORK has isolation level RU; cursor C1 has isolation RC; cursor C2 has isolation level CS; and cursor C3 has isolation level RU.

Authorization

You do not need authorization to use the SET SESSION statement.

Example

The following example illustrates setting session level deferred constraint checking, DML atomicity, and the FILL option to enhance load performance within ISQL.

```
COMMIT WORK;

SET LOAD_BUFFER 65536;
SET AUTOSAVE 3000
SET LOAD_ECHO AT_COMMIT;
SET AUTOCOMMIT ON;
SET AUTOLOCK ON;
SET SESSION UNIQUE, REFERENTIAL, CHECK CONSTRAINTS DEFERRED,
            DML ATOMICITY AT ROW LEVEL,
            FILL;
.
.
.
BEGIN WORK;
LOAD FROM EXTERNAL Price TO PurchDB.SupplyPrice;
LOAD FROM EXTERNAL Parts TO PurchDB.Parts;
.
.
.
COMMIT WORK;
```

In the above example, a COMMIT WORK is automatically performed when 3000 rows have been loaded from external files into the database tables. A new transaction is started on behalf of the user to continue to load remaining rows. Each new transaction uses the default session isolation level (RR).

SET TRANSACTION

The SET TRANSACTION statement sets one or more transaction attributes for a transaction. These attributes include: isolation level, priority, user label, constraint checking mode, timeout rollback, user timeout, termination level, and DML atomicity level.

Scope

ISQL or Application Programs

SQL Syntax

```
SET TRANSACTION {ISOLATION LEVEL {RR
                                  CS
                                  RC
                                  RU
                                  REPEATABLE READ
                                  SERIALIZABLE
                                  CURSOR STABILITY
                                  READ COMMITTED
                                  READ UNCOMMITTED
                                  :HostVariable1    }
                 PRIORITY {Priority
                           :HostVariable2}
                 LABEL {'LabelString'
                        :HostVariable3}
                 ConstraintType [,...]CONSTRAINTS {DEFERRED
                                                   IMMEDIATE}
                 DML ATOMICITY AT {STATEMENT
                                   ROW       }LEVEL
                 ON {TIMEOUT
                     DEADLOCK}ROLLBACK {QUERY
                                        TRANSACTION}
                 USER TIMEOUT [TO] {DEFAULT
                                    MAXIMUM
                                    TimeoutValue[{SECONDS
                                                  MINUTES}]
                                    :HostVariable4[{SECONDS
                                                    MINUTES}]
                 TERMINATION AT {SESSION
                                 TRANSACTION
                                 QUERY
                                 RESTRICTED   }LEVEL             }[,...]
```

Parameters

RR	Repeatable Read. Means that the transaction uses locking strategies to guarantee repeatable reads. RR is the default isolation level.
CS	Cursor Stability. Means that your transaction uses locking strategies to assure cursor-level stability only.

RC Read Committed. Means that your transaction uses locking strategies to ensure that you retrieve only rows that have been committed by some transaction.

RU Read Uncommitted. Means that the transaction reads data without obtaining additional locks.

REPEATABLE READ Same as RR.

SERIALIZABLE Same as RR.

CURSOR STABILITY Same as CS.

READ COMMITTED Same as RC.

READ UNCOMMITTED Same as RU.

HostVariable1 is a string host variable containing one of the isolation level specifications above.

Priority is an integer from 0 to 255 specifying the priority of the transaction. Priority 127 is the default. ALLBASE/SQL uses the priority to resolve a deadlock. The transaction with the largest priority number is aborted to remove the deadlock.

For example, if a priority-0 transaction and a priority-1 transaction are deadlocked, the priority-1 transaction is aborted. If two transactions involved in a deadlock have the same priority, the deadlock is resolved by aborting the newer transaction (the last transaction begun, either implicitly or with a BEGIN WORK statement).

HostVariable2 is an integer host variable containing the priority specification.

LabelString is a user defined character string of up to 8 characters. The default is a blank string.

The label is visible in the SYSTEM.TRANSACTION pseudo-table and also in SQLMON. Transaction labels can be useful for troubleshooting and performance tuning. Each transaction in an application program can be marked uniquely, allowing the DBA to easily identify the transaction being executed by any user at any moment.

Labels for a new transaction can be specified with the BEGIN WORK, SET TRANSACTION, and SET SESSION statements. SET TRANSACTION can also be used to change the existing label of an active transaction. If a transaction consists of multiple queries and unique labels are set between each query, a DBA can identify the actual query being executed by an active transaction.

HostVariable3 is a string host variable containing the *LabelString*.

ConstraintType identifies the types of constraints that are affected by the DEFERRED and IMMEDIATE options. Each *ConstraintType* can be one of the following:

```
UNIQUE
REFERENTIAL
CHECK
```

DEFERRED	specifies that constraint errors are not checked until the constraint checking mode is reset to IMMEDIATE or the current transaction ends.
IMMEDIATE	specifies that constraint errors are checked when a statement executes. This is the default.
STATEMENT	specifies that error checking occurs at the statement level. This is the default.
ROW	specifies that error checking occurs at the row level.
QUERY	sets the action for timeouts or deadlocks to rollback the statement or query.
TRANSACTION	sets the action for timeouts or deadlocks to rollback the transaction.
DEFAULT	specifies to use the default timeout duration for the DBE specified in the START DBE statement.
MAXIMUM	specifies to use the maximum timeout duration for the DBE specified in the START DBE statement.
TimeoutValue	specifies the timeout duration to use in seconds or minutes.
:HostVariable4	is an integer host variable specifying the timeout duration to use in seconds or minutes.
SESSION	specifies self-termination at the session level, and allows external termination at the session level only.
TRANSACTION	specifies self-termination at the transaction level, and allows external termination at the session or transaction level.
QUERY	specifies self-termination at the query level, and allows external termination at the session, transaction, or query level.
RESTRICTED	specifies no self-termination, and allows external termination at the session level only. This is the default.

Description

- Detailed information about isolation levels is presented in the "Concurrency Control through Locks and Isolation Levels" chapter.
- You can issue the SET TRANSACTION statement at any point in an application or ISQL session. If the SET TRANSACTION statement is issued outside of an active transaction, its attribute(s) apply to the next transaction. If issued within a transaction, its attribute(s) apply to the current transaction.
- Within a transaction, any attribute specified in a SET TRANSACTION statement remains in effect until the transaction terminates or until reset by another statement issued within the transaction. See Chapter 2 , “Using ALLBASE/SQL,” "Scoping of Transaction and Session Attributes" section for information about statements used to set transaction attributes.
- When using RC or RU, you should verify the existence of a row before you issue an UPDATE statement. In application programs that employ cursors, you can use the

REFETCH statement prior to updating. REFETCH is not available in ISQL. Therefore, you should use caution in employing RC and RU in ISQL if you are doing updates.

- Within a transaction, different isolation levels can be set for different DML statements. For example, a cursor opened following a SET TRANSACTION statement is opened with the specified isolation level, but any cursor opened prior to this SET TRANSACTION statement maintains the isolation level with which it was opened.
- As with the SET CONSTRAINTS statement, the SET TRANSACTION statement allows you to set the UNIQUE, REFERENTIAL or CHECK constraint error checking mode. If the constraint checking mode is deferred, checking of constraints is deferred until the end of a transaction or until the constraint mode is set back to immediate. If the constraint mode is immediate, integrity constraints are checked following processing of each SQL statement (if statement level atomicity is in effect) or each row (if row level atomicity is in effect). Refer to the SET DML ATOMICITY statement in this chapter for further information on statement and row level error checking. The following paragraph assumes that statement level atomicity is in effect.

  When constraint checking is deferred, a COMMIT WORK, or SET CONSTRAINTS IMMEDIATE statement executes if zero constraint violations exist at that time, otherwise a constraint error is reported. When constraint checking is immediate (the default), zero constraint violations must exist when an SQL statement executes, otherwise a constraint error is reported and the statement is rolled back. The SET CONSTRAINTS statement in this chapter gives further detail about constraint checking.
- As with the SET DML ATOMICITY statement, the SET TRANSACTION statement allows you to set the general error checking level in data manipulation statements. General error checking refers to any errors, for example, arithmetic overflows or constraint violation errors.

  Setting ROW LEVEL atomicity guarantees that internal savepoints are not generated. For example, if an error occurs on the *n*th row of a bulk statement such as LOAD, BULK INSERT, or Type2 INSERT, the row is not processed, statement execution terminates, and any previously processed rows are *not* rolled back. In contrast, STATEMENT LEVEL atomicity guarantees that the entire statement is rolled back if it does not execute without error. STATEMENT LEVEL atomicity is the default. Refer to the SET DML ATOMICITY statement in this chapter for further information on statement and row level error checking.
- All transaction attributes are sensitive to savepoints. That is, if you establish a savepoint, then change the transaction attribute(s) by issuing a SET TRANSACTION statement, and then roll back to the savepoint, the transaction attribute(s) set after the savepoint are undone.
- When ON TIMEOUT ROLLBACK or ON DEADLOCK ROLLBACK is set to TRANSACTION, the whole transaction is aborted as a result of a timeout or deadlock.
- When ON TIMEOUT ROLLBACK or ON DEADLOCK ROLLBACK is set to QUERY, only the SQL statement which has timed out will be rolled back. This means rolling back results of statements that modify the database and closing cursor for the cursor-related statements. (Cursor-related statements change the cursor position, and are not statements like UPDATE or DELETE WHERE CURRENT.)
- In general, if a transaction with KEEP cursor(s) is committed, the new transaction

started on behalf of the user inherits the most recent transaction attributes of the old transaction. However, the KEEP cursor(s) inherit the isolation level attribute of the old transaction at the time the cursor(s) were opened. For example:

```
BEGIN WORK RC
.
.
.
OPEN C1 KEEP CURSOR ...
.
.
.
SET TRANSACTION ISOLATION LEVEL CS
.
.
.
OPEN C2 KEEP CURSOR ...
.
.
.
SET TRANSACTION ISOLATION LEVEL RU
.
.
.
COMMIT WORK
.
.
.
OPEN C3
.
.
.
```

In the above example, the new transaction started on behalf of the user after the `COMMIT WORK` has isolation level RU; cursor C1 has isolation RC; cursor C2 has isolation level CS; and cursor C3 has isolation level RU.

- The `SET TRANSACTION` statement is not allowed within a stored procedure.

Authorization

You do not need authorization to use the `SET TRANSACTION` statement.

Example

Declare multiple cursors

```
DECLARE C1 CURSOR FOR SELECT BranchNo FROM Branches
         WHERE TellerNo > :TellerNo

DECLARE C2 CURSOR FOR SELECT BranchNo FROM Tellers
         WHERE BranchNo = :HostBranchNo FOR UPDATE OF Credit

DECLARE C3 CURSOR FOR SELECT * FROM PurchDB.Parts
```

Set the isolation level to RC.

```
SET TRANSACTION ISOLATION LEVEL RC, PRIORITY 100, LABEL 'xact1'
.
.
.
Implicit BEGIN WORK with transaction isolation level RC.

OPEN C1
FETCH C1 INTO :HostBranchNo1
.
.
.
```

Change isolation level to CS.

```
SET TRANSACTION ISOLATION LEVEL CS
OPEN C2
FETCH C2 INTO :HostBranchNo2
UPDATE Tellers SET Credit = Credit * 0.005
         WHERE CURRENT OF C2
CLOSE C2                                    Close cursor C2.
CLOSE C1                                    Close cursor C1.
```

Change the transaction isolation level back to RC.

```
SET TRANSACTION ISOLATION LEVEL RC
OPEN C3
FETCH C3 INTO :PartsBuffer
.
.
.
```

End the transaction. Transaction attributes return to those set at the session level or to the session default.

```
COMMIT WORK
```

SET USER TIMEOUT

The SET USER TIMEOUT statement specifies the amount of time the user will wait if the requested database resource is unavailable.

Scope

ISQL or Application Programs

SQL Syntax

```
SET USER TIMEOUT [TO] {{TimeoutValue
                        :HostVariable}[SECONDS
                                       MINUTES]
                      DEFAULT
                      MAXIMUM                  }
```

Parameters

TimeoutValue is an integer literal greater than or equal to zero. If the *TimeoutValue* is not qualified by MINUTES, SECONDS is assumed. If representing seconds, *TimeoutValue* must be in the range of 0 to 2,147,483,647. If representing minutes, *TimeoutValue* must be in the range of 0 to 35,791,394.

HostVariable identifies an integer host variable containing a timeout value.

DEFAULT indicates that the user timeout value will be set to the default timeout value specified by the database administrator.

MAXIMUM indicates that the user timeout value will be set to the maximum timeout value specified by the database administrator.

Description

- The value specified by SET USER TIMEOUT remains in effect only for the duration of the user's session and only affects that session, and does not modify the value stored in the DBECon file.
- Database resources that may cause a user to wait include the following:

  Locks — The user attempts to lock a database object that is already locked by another transaction in a conflicting mode.

  Transaction Slots — The application tries to begin a transaction but the maximum number of transactions allowed has been reached. ALLBASE/SQL creates an implicit, brief transaction when the CONNECT statement is issued.
- If the *TimeoutValue* is zero and the database resource is unavailable, the user will not wait and an error will occur. To implement locking with no waiting, set the *TimeoutValue* to zero.

- The *TimeoutValue* may not exceed the maximum timeout value set by the database administrator.
- The database administrator may specify the maximum and default timeout values with the SQLUtil ALTDBE command, or with the following SQL statements:
  - START DBE
  - START DBE NEW
  - START DBE NEWLOG
- You may view the current maximum and default timeout values with the SQLUtil SHOWDBE command.
- The SET USER TIMEOUT statement is not allowed in the PREPARE statement. A host variable is not permitted if the SET USER TIMEOUT statement is used in the EXECUTE IMMEDIATE statement. No section is created for the SET USER TIMEOUT statement.
- An active transaction is not required to execute a SET USER TIMEOUT statement. An automatic transaction is not started when a SET USER TIMEOUT statement is executed.

Authorization

You do not need authorization to use SET USER TIMEOUT.

Example

Examples of setting the user timeout value in seconds:

```
SET USER TIMEOUT TO 10

SET USER TIMEOUT TO 5 SECONDS
```

Set user timeout in minutes:

```
SET USER TIMEOUT 1 MINUTES
```

When setting the user timeout value to 0, the user will not wait for a database resource that is unavailable, such as a lock.

```
SET USER TIMEOUT 0
```

Set the user timeout value to the default or the maximum value.

```
SET USER TIMEOUT DEFAULT

SET USER TIMEOUT MAXIMUM
```

SQLEXPLAIN

The SQLEXPLAIN statement places a message describing the meaning of a return code into a host variable. The text of messages comes from the ALLBASE/SQL message catalog.

Scope

Application Programs Only

SQL Syntax

SQLEXPLAIN :*HostVariable*

Parameters

HostVariable identifies a host variable used to hold an ALLBASE/SQL exception message. The message describes the meaning of a return code. ALLBASE/SQL puts a return code into the SQLCA after each SQL statement in a program is executed. The SQLCA is an area for information on errors, warnings, truncation, null values, and other conditions related to the execution of an SQL statement.

Description

- This statement cannot be used interactively or in procedures.
- If more than one error occurs, SQLEXPLAIN can be used to obtain more than one message. You execute SQLEXPLAIN repeatedly until the SQLCODE field of the SQLCA data structure is equal to zero. Refer to the ALLBASE/SQL application programming guide for the language you are using for more information on status checking in a program.
- The fully qualified name for the default message catalog is as follows:

  /usr/lib/nls/n-computer/hpsqlcat

  For native language users, the following is the name of the catalog:

  /usr/lib/nls/$LANG/hpsqlcat

  where $LANG is the current language.

  If this catalog is not available, ALLBASE/SQL uses the default catalog instead.

Authorization

You do not need authorization to use SQLEXPLAIN.

Example

```
INCLUDE SQLCA

SQLStatement1
```

The host variable named :Message contains a message characterizing the execution of SQLStatement1.

```
SQLEXPLAIN :Message
```

START DBE

The START DBE statement establishes a connection with a given DBEnvironment and establishes a set of startup parameters that apply to this and all subsequent connections until all connections to the DBEnvironment have been terminated. Any startup parameters not explicitly specified are taken from the DBECon file. The changes are only temporary for START DBE parameters; use START DBE NEW to specify the start up parameters to be stored in the DBECon file in a new DBE. Use SQLUtil to change the parameters in the DBECon file in an existing DBE.

Scope

ISQL or Application Programs

SQL Syntax

```
START DBE 'DBEnvironmentName' [AS 'ConnectionName'][MULTI]
[BUFFER = DataBufferPages, LogBufferPages)
 TRANSACTION = MaxTransactions
 MAXIMUM TIMEOUT = {TimeoutValue [SECONDS
                                  MINUTES]
                   NONE                    }
 DEFAULT TIMEOUT = {TimeoutValue [SECONDS
                                  MINUTES]
                   MAXIMUM                 }
 RUN BLOCK = ControlBlockPages]|,...|
```

Parameters

DBEnvironmentName identifies the DBEnvironment in which the session is established. Unless you specify an absolute path name, the name you specify is assumed to be relative to your current working directory.

ConnectionName associates a user specified name with this connection. *ConnectionName* must be unique for each DBEnvironment connection within an application or an ISQL session. If a *ConnectionName* is not specified, *DBEnvironmentName* is the default. *ConnectionName* cannot exceed 128 bytes.

MULTI indicates the DBEnvironment can be accessed by multiple users simultaneously. If omitted, the DBEnvironment can be accessed only by the user issuing the START DBE statement. If the MULTI option is specified, other users can start DBE sessions by using the CONNECT statement.

DataBufferPages specifies the number of 4096-byte data buffer pages to be used. Data buffer pages hold index and data pages.

You can request up to 50,000 data buffer pages. The minimum number of data buffers is 15. The default number is 100. The total number of data buffer pages and runtime control block pages cannot exceed 256 Mbytes.

See the "ALLBASE/SQL Limits" Appendix in the *ALLBASE/SQL Database Administration Guide.*

LogBufferPages specifies the number of 512-byte log buffer pages to be used. You can request from 24 to 1024 log buffer pages, limited by the amount of storage available. The default number of log buffer pages is 24.

MaxTransactions specifies the maximum number of concurrent transactions that can be concurrently active. You can specify a value in the range from 2 to 240. The default is 50.This value overrides the maximum value stored in the DBECon file. Any attempt to start a transaction beyond the maximum limit waits for the specified TIMEOUT and returns an error if TIMEOUT is exceeded. For each user logged on to the system at any point in time you should allow 2 concurrent transactions for just being connected to the DBE.

MAXIMUM TIMEOUT specifies the maximum user timeout value. This value temporarily overrides the maximum stored in the DBECon file. When no value is specified, the DBECon file value is the default.

DEFAULT TIMEOUT specifies the maximum user timeout value. This value temporarily overrides the maximum stored in the DBECon file. When no value is specified, the DBECon file value is the default.

TimeoutValue is an integer literal greater than zero. If the *TimeoutValue* is not qualified by MINUTES, SECONDS is assumed. If representing seconds, *TimeoutValue* must be in the range of 1 to 2,147,483,647. If representing minutes, *TimeoutValue* must be in the range of 1 to 35,791,394.

ControlBlockPages specifies the number of runtime control blocks allocated.

You can specify a value from 17 to 2,000 pages for this parameter. The default is 37 pages. The total number of data buffer pages and runtime control block pages cannot exceed 256 Mbytes.

Description

- Any parameters (except the MULTI option) not specified in the START DBE statement are assigned values currently stored in the DBECon file (which has the same name as the DBEnvironment name specified). The user mode in the DBECon file is *always* overridden by the user mode specified in the START DBE statement.
- Normally, if autostart mode has the value of ON, DBE sessions are established by using the CONNECT statement. The START DBE statement is needed only to temporarily override DBECon file parameters for the DBEnvironment, as when you need to start a single-user DBEnvironment in multiuser mode or vice versa. When you issue the CONNECT statement and autostart mode is on, ALLBASE/SQL executes a START DBE statement on your behalf if no sessions for the DBEnvironment are active. ALLBASE/SQL starts your session using all parameters in the current DBECon file.
- If autostart mode has a value of OFF, you always use the START DBE statement to start up a DBEnvironment. If the MULTI option is not specified, the DBEnvironment is started up in single-user mode. If the MULTI option is specified, the DBEnvironment is started up in multiuser mode and other users can initiate DBE sessions by using the

CONNECT statement.

- Timeout values set in the START DBE statement remain in effect only as long as there is a session established for connected DBEnvironments and do not modify the values stored in the DBECon file.
- If no MAXIMUM TIMEOUT limit is specified, the MAXIMUM TIMEOUT limit stored in the DBECon file remains in effect. If no DEFAULT TIMEOUT value is specified, the DEFAULT TIMEOUT value stored in the DBECon file remains in effect.
- If MAXIMUM TIMEOUT = NONE, infinity (no timeout) is assumed. If DEFAULT TIMEOUT = MAXIMUM, the value of MAXIMUM TIMEOUT is assumed. The DEFAULT TIMEOUT value may not exceed the MAXIMUM TIMEOUT value.
- No connections to the DBEnvironment can be in effect when this command is issued.
- The START DBE statement is also used before using the START DBE NEWLOG statement. Refer to the START DBE NEWLOG statement for additional information.

Authorization

You can issue the START DBE statement only if you have DBA authority.

Example

The sample DBEnvironment is started in single-user mode. All parameters except the user mode in the DBECon file are used for the duration of the single-user session. After this session ends, a CONNECT statement can be used to establish a multiuser session for this DBEnvironment, because the user mode in the DBECon file is still multiuser and the autostart flag is still ON.

```
START DBE '../sampledb/PartsDBE'
```

START DBE NEW

The START DBE NEW statement configures and establishes a connection with a new DBEnvironment. It establishes a set of startup parameters that apply to this and all subsequent connections until all connections to the DBEnvironment have been terminated. Startup parameters are also stored in the DBECon file.

Scope

ISQL or Application Programs

SQL Syntax — START DBE NEW

```
START DBE 'DBEnvironmentName' [AS 'ConnectionName'][MULTI]NEW
[{DUAL
  AUDIT}|...|LOG
 BUFFER = DataBufferPages, LogBufferPages)
 LANG = LanguageName
 TRANSACTION = MaxTransactions
 MAXIMUM TIMEOUT = {TimeoutValue [SECONDS
                                  MINUTES]
                    NONE                  }
 DEFAULT TIMEOUT = {TimeoutValue [SECONDS
                                  MINUTES]
                    MAXIMUM               }
 RUN BLOCK = ControlBlockPages]
 DEFAULT PARTITION = {DefaultPartitionNumber
                      NONE                }
 COMMENT PARTITION = {CommentPartitionNumber
                      DEFAULT
                      NONE                }
 MAXPARTITIONS = MaximumNumberOfPartitions
 AUDIT NAME = 'AuditName'
 {COMMENT
  DATA
  DEFINITION
  STORAGE
  AUTHORIZATION
  SECTION
  ALL           }|...|AUDIT ELEMENTS
 DBEFile0Definition
 DBELogDefinition                              ]|,...|
```

Parameters — START DBE NEW

DBEnvironmentName identifies the DBEnvironment name used in the CONNECT statement. This name also identifies the DBECon file that stores the values of all parameters specified in the START DBE NEW statement that are also used in the CONNECT statement. Unless you specify an absolute path name, ALLBASE/SQL assumes the name is relative to your current working directory.

ConnectionName associates a user specified name with this connection.

ConnectionName must be unique for each DBEnvironment connection within an application. If a *ConnectionName* is not specified, *DBEnvironmentName* is the default. *ConnectionName* cannot exceed 128 bytes.

MULTI indicates the DBEnvironment can be accessed by multiple users simultaneously. If omitted, the DBEnvironment can be accessed only in single-user mode.

DUAL LOG causes ALLBASE/SQL to maintain two separate logs, preferably on different media. Keeping the log files on separate media ensures that a media failure on one device leaves the other log undamaged. Each log write operation is performed on both logs; if an error is detected, the write continues on the good log only. Normally, only one log is read, but if an error is encountered, ALLBASE/SQL switches to the other log. Data integrity is maintained provided is at least one good copy of each log record is on at least one of the logs.

AUDIT LOG identifies the DBEnvironment as one that will have audit logging performed on it with the elements specified in the AUDIT ELEMENTS clause. This causes ALLBASE/SQL to create audit log records as well as normal log records in the log file so that the database can be audited.

DataBufferPages specifies the number of 4096-byte data buffer pages to be used. Data buffer pages hold index and data pages.

You can request up to 50,000 data buffer pages. The minimum number of data buffer pages is 15. The default number is 100. The total number of data buffer pages and runtime control block pages cannot exceed 256 Mbytes.

LogBufferPages specifies the number of 512-byte log buffer pages to be used. You can request from 24 to 1024 log buffer pages, limited by the amount of storage available. The default number of log buffer pages is 24.

LANG specifies the language for the DBEnvironment. If the name of the language contains a hyphen, use double quotes in specifying it, as in the following (c-french means Canadian French):

```
LANG = "c-french"
```

MaxTransactions specifies the maximum number of concurrent transactions to be supported. You can specify a value from 2 to 240. The default is 50. Any attempt to start a transaction beyond the maximum limit waits for the specified TIMEOUT and returns an error if TIMEOUT is exceeded. For each user logged on to the system at any point in time you should allow 2 concurrent transactions for just being connected to the DBE.

MAXIMUM TIMEOUT specifies the maximum user timeout value that is stored in the DBECon file. The default is the MAXIMUM.

DEFAULT TIMEOUT specifies the default user timeout value that is stored in the DBECon file. The default is NONE (infinity).

TimeoutValue is an integer literal greater than zero. If the *TimeoutValue* is not qualified by MINUTES, SECONDS is assumed. If representing seconds, *TimeoutValue* must be in the range of 1 to 2,147,483,647. If representing minutes, *TimeoutValue* must be in the range of 1 to 35,791,394.

ControlBlockPages specifies the number of runtime control blocks to be allocated. The value specified is stored in the DBECon file.

You can specify a value from 17 to 2,000 pages for this parameter. The default is 37 pages. The total number of data buffer pages and runtime control block pages cannot exceed 256 Mbytes.

DefaultPartitionNumber specifies the default partition number for the DBEnvironment. This clause must be specified if AUDIT LOG is specified. *DefaultPartitionNumber* must be in the range 1 and 32767. If NONE is specified, tables in the DBEnvironment that are in the default partition do not generate audit log records. See the CREATE TABLE and ALTER TABLE statements for information on assigning a table to a partition.

CommentPartitionNumber specifies the partition number for comments made in the DBEnvironment. *CommentPartitionNumber* must be a number between 1 and 32767. If no COMMENT PARTITION is specified, DEFAULT is implied.

If the comment partition is DEFAULT and the default partition number is later changed in a START DBE NEWLOG statement (but the comment partition is not changed from DEFAULT), the comment partition number will also change to the new default partition number.

MaximumNumberOfPartitions specifies the maximum number of partitions for the DBEnvironment. This clause must be specified if AUDIT LOG has been specified. *MaximumNumberOfPartitions* is required to be a number between 1 and 831. This number indicates the number of partition instances the DBEnvironment is expected to track.

For audit logging purposes, the number of partition instances is calculated as the sum of the number of DATA partitions and the number of elements (not counting the DATA element) specified in the AUDIT ELEMENTS clause. Specifying ALL audit elements (see below) includes 6 elements, implying that 6 partitions are used. Set this value only as high as needed so that unnecessary space is not reserved unless you plan more partitions or audit elements in the future.

AuditName specifies the name of this audit DBEnvironment. *AuditName* is limited to 8 bytes. This clause must be specified if AUDIT LOG has been specified. The *AuditName* appears in outputs of the Audit Tool.

AUDIT ELEMENTS Specifies the types of audit logging that will be done for the database. If this clause is omitted and AUDIT LOG is specified, DATA AUDIT ELEMENTS is implicit. The audit elements are as follows:

COMMENT permits use of the LOG COMMENT statement in the DBEnvironment. Comments are logged to the defined COMMENT PARTITION. If this element is not chosen,

the LOG COMMENT statement returns an error.

DATA — is the default element. It causes audit log records to be done for any data operations (INSERT, UPDATE, or DELETE) on tables that are in an audit partition of the DBEnvironment other than NONE. (Tables can be specified to be in partition NONE and thus not participate in the audit logging process.)

DEFINITION — includes audit logging of the following statements:

```
CREATE TABLE
ALTER TABLE
DROP TABLE
CREATE INDEX
CREATE VIEW
DROP VIEW
CREATE RULE
DROP RULE
CREATE PROCEDURE
DROP PROCEDURE
TRANSFER OWNERSHIP
CREATE GROUP
DROP GROUP
CREATE DBEFILESET
DROP DBEFILESET
CREATE PARTITION
DROP PARTITION
TRUNCATE TABLE
```

STORAGE — includes audit logging of the following statements:

```
CREATE DBEFILE
DROP DBEFILE
ADD DBEFILE TO DBEFILESET
REMOVE DBEFILE FROM DBEFILESET
CREATE TEMPSPACE
DROP TEMPSPACE
```

AUTHORIZATION — includes audit logging of the following statements:

```
GRANT
REVOKE
ADD TO GROUP
REMOVE FROM GROUP
```

SECTION — includes audit logging of the creation and deletion of permanent sections or procedures. Permanent sections or procedures are created when a program is preprocessed, and are deleted by the DROP MODULE statement. The DROP PROCEDURE statement deletes procedures. Logging of section creation does not include any SETOPT information associated with the section. See the SETOPT statement.

ALL — is equivalent to specifying COMMENT DATA DEFINITION STORAGE AUTHORIZATION SECTION

AUDIT ELEMENTS as described above.

DBEFile0Definition is a clause that provides the information ALLBASE/SQL needs to automatically create DBEFile0 and add it to the SYSTEM DBEFileSet. The syntax for this clause is presented separately below. If *DBEFile0Definition* is omitted, ALLBASE/SQL assumes the following:

```
DBEFILE0 DBEFILE DBEFILE0
   WITH PAGES = 150,
   NAME = 'DBEFile0'
```

By default, DBEFile0 resides in the same directory as the DBECon file. However, you can use the SQLUtil MOVEFILE command to move it to another directory.

DBELogDefinition is a clause that provides ALLBASE/SQL with the information needed to create one or more log files. Syntax for this clause is presented separately below. If *DBELogDefinition* is omitted, ALLBASE/SQL assumes the following:

```
LOG DBEFILE DBELOG1
   WITH PAGES = 250,
   NAME = 'DBELOG1
```

By default, DBELOG1 resides in the same directory as the DBECon file.

SQL Syntax — DBEFile0Definition

```
DBEFILE0 DBEFILE DBEFile0ID
WITH PAGES = DBEFile0Size,
NAME = 'SystemFileName1'
```

Parameters — DBEFile0Definition

DBEFILE0 DBEFILE describes a DBEFile known as DBEFile0, which contains the portion of the system catalog needed for activating a DBEnvironment, including definitions of other DBEFiles. Each DBEnvironment must have a DBEFile0 associated with a unique *SystemFileName*, which is assigned in this clause.

DBEFile0ID is the basic name identifying DBEFile0.

DBEFile0Size specifies the number of 4096-byte pages in DBEFile0. You can specify from 150 to 524,287 pages. The default and minimum is 150.

SystemFileName1 identifies how DBEFile0 is known to the operating system. DBEFile0 is created relative to the directory specified in the DBEnvironment name parameter unless an absolute path name is specified. The default file name is 'DBEFile0'.

SQL Syntax — DBELogDefinition

```
LOG DBEFILE DBELog1ID [AND DBELog2ID]
WITH PAGES = DBELogSize,
NAME = 'SystemFileName2' [AND 'SystemFileName3']
```

Parameters — DBELogDefinition

`LOG DBEFILE` describes the two log files if the DUAL LOG option is specified, or a single log file otherwise. If you give information for two log files but omit the DUAL LOG option, the information for the second log file is ignored.

DBELog1ID and *DBELog2ID* are the basic names identifying the log files.

DBELogSize specifies the number of 512-byte pages in one log file. If dual logging is used, both logs must be the same size. The DBE log size should be at least 250 pages and no greater than 524,287 pages. The default is 250. If you choose an odd number of pages, the number is rounded up to an even number.

SystemFileName2 and *SystemFileName3* identify how the logs are known to HP-UX. The logs are created relative to the directory specified in the DBEnvironment name parameter unless an absolute path name is specified.

Description

- When you issue this statement, ALLBASE/SQL creates a DBECon file with the same name as the *DBEnvironmentName*.
- The following parameters defined in the `START DBE NEW` statement are stored in the DBECon file:
  - DBEnvironment language
  - User mode (single versus multi)
  - Number of data buffer pages
  - Number of log buffer pages
  - Maximum transactions
  - Maximum timeout value
  - Default timeout value
  - Number of runtime control block pages
  - DBEFile0 system file name
  - Log system file name(s)
  - Audit logging (chosen versus not)
  - Audit name
  - Audit elements
  - Default partition
  - Comment partition
  - Maximum number of partitions

- The following additional parameters are stored in the DBECon file:
  - The **autostart flag** determines how DBE sessions are started. If the value of autostart is ON, a DBE session can be established by using the CONNECT statement. If the value of autostart is OFF, the START DBE statement must be used to start up a DBEnvironment; if the START DBE statement contains the MULTI option, other users establish DBE sessions with the CONNECT statement. Autostart is on by default.
  - The **DDL Enabled flag** determines whether data definition is enabled for the DBEnvironment. The DDL Enabled flag is set to YES by default. See "Maintenance" in the *ALLBASE/SQL Database Administration Guide* for additional information about the DDL Enabled flag.
  - The **archive mode flag** determines whether the DBEnvironment is operating in archive mode. In archive mode, ALLBASE/SQL does rollforward logging. The rollforward log can be used to redo transactions in case it is necessary to restore the DBEnvironment from a backed up (archival) copy. When archive mode has the value of OFF, log space can be recovered by using the CHECKPOINT statement. If you want to do rollforward recovery, you must always operate in archive mode. Rollback recovery is enabled regardless of the archive mode. Archive mode is off by default.
- When you choose an odd number of log pages using the WITH PAGES clause of the DBEFile definitions, the number is rounded up to an even number, which is displayed in SHOWLOG.
- The size of DBEFile0 is fixed at the time you configure a DBEnvironment and cannot be changed. If you need more space at a later time, add a DBEFile to the SYSTEM DBEFileSet.
- DBEFile0 cannot be restricted to containing data pages only or index pages only; the storage in DBEFile0 is used for both data and index pages.
- You can reconfigure a DBEnvironment by using SQLUtil to alter DBECon file parameters. All parameters except the name of the DBECon file and DBEFile0 may be changed. Refer to the *ALLBASE/SQL Database Administration Guide* for additional information.
- The files created with this statement are owned by hpdb and have the following permissions:

  ```
  -rw------
  ```

- If no MAXIMUM TIMEOUT limit is specified, or if MAXIMUM TIMEOUT = NONE, infinity (no timeout) is assumed. If no DEFAULT TIMEOUT value is specified, or if DEFAULT TIMEOUT = MAXIMUM, the value of MAXIMUM TIMEOUT is assumed. The DEFAULT TIMEOUT value may not exceed the MAXIMUM TIMEOUT value.
- You cannot use the START DBE NEW statement on a diskless machine. A DBEnvironment for use in diskless clusters must be created on the cluster server. Refer to the *ALLBASE/NET User's Guide* for further information.
- You can reconfigure a DBEnvironment by using SQLUtil to alter DBECon file parameters. All parameters except the audit information (logging, audit elements, name, default, comment and maximum partition), or the name of the DBECon file and

DBEFile0 may be changed. Refer to the *ALLBASE/SQL Database Administration Guide* for additional information.

- If AUDIT LOG is specified, the clauses AUDIT NAME, DEFAULT PARTITION, and MAXPARTITIONS must also be specified. Further, if no AUDIT ELEMENTS are specified, DATA is used as a default. If no COMMENT PARTITION is specified, DEFAULT is assumed. The DEFAULT PARTITION or the COMMENT PARTITION can be specified as NONE.
- Use of the clause ALL AUDIT ELEMENTS implies specification of all of the audit elements.
- Additional log files should be added using the `SQLUtil ADDLOG` command.

Authorization

No authorization is needed for using the `START DBE NEW` statement. `hpdb` must have write permission in the target directory for all the files the START DBE NEW statement creates.

Example

The DBEnvironment for the sample database is a multiuser DBEnvironment in which as many as five transactions can execute concurrently. The DBEnvironment is initially configured for two rollback logs and a DBEFile0 residing in PartsF0. The number of runtime control pages to be used is 500. By default, autostart mode is set to ON.

```
START DBE '../sampledb/PartsDBE' MULTI NEW
          DUAL LOG,
          TRANSACTION = 5,
          DBEFILE0 DBEFILE PartsDBE0
            WITH PAGES = 150, NAME = 'PartsF0',
          LOG DBEFILE PartsDBELog1 AND PartsDBELog2
            WITH PAGES = 256, NAME = 'PartsLg1' AND 'PartsLg2',
          RUN BLOCK = 500
```

The DBEnvironment has all the above parameters listed and it is enabled for audit logging. All DML and DDL changes in the DBEnvironment are subject to audit logging since all audit elements are selected. Up to 20 partitions can coexist in this DBEnvironment, allowing for 14 data partitions in addition to the other elements' partitions. The log files should be made large enough for the added audit log records.

```
START DBE '../sampledb/PartsDBE' MULTI NEW
          DUAL AUDIT LOG,
          TRANSACTION = 5,
          RUN BLOCK = 500,
          AUDIT NAME = 'PrtsDBE1',
          DEFAULT PARTITION = 1,
          COMMENT PARTITION = 2,
          MAXPARTITIONS = 20,
          ALL AUDIT ELEMENTS,
        DBEFILE0 DBEFILE PartsDBE0
          WITH PAGES = 150, NAME = 'PartsF0',
        LOG DBEFILE PartsDBELog1 AND PartsDBELog2
          WITH PAGES = 1000, NAME = 'PartsLg1' AND 'PartsLg2'
```

START DBE NEWLOG

The START DBE NEWLOG statement establishes a connection with a given DBEnvironment and creates one or two new log files for that DBEnvironment. It establishes a set of startup parameters that apply to this and all subsequent connections until all connections to the DBEnvironment have been terminated. Any start up parameters not explicitly specified are taken from the DBECon file except the enabling of audit logging. This statement reinitializes log file(s) when you need to change the size, invoke a dual logging or startup, or alter audit logging.

Scope

ISQL or Application Programs

SQL Syntax — START DBE NEWLOG

```
START DBE 'DBEnvironmentName' [AS 'ConnectionName'][MULTI]NEWLOG
[{ARCHIVE
  DUAL
  AUDIT}|...|LOG
 BUFFER = (DataBufferPages, LogBufferPages)
 TRANSACTION = MaxTransactions
 MAXIMUM TIMEOUT = {TimeoutValue [SECONDS
                                  MINUTES]
                    NONE                  }
 DEFAULT TIMEOUT = {TimeoutValue [SECONDS
                                  MINUTES]
                    MAXIMUM               }
 RUN BLOCK = ControlBlockPages
 DEFAULT PARTITION = {DefaultPartitionNumber
                      NONE                }
 COMMENT PARTITION = {CommentPartitionNumber
                      DEFAULT
                      NONE                }
 MAXPARTITIONS = MaximumNumberOfPartitions
 AUDIT NAME = 'AuditName'
 {COMMENT
  DATA
  DEFINITION
  STORAGE
  AUTHORIZATION
  SECTION
  ALL          }|...|AUDIT ELEMENTS            ]|,...| NewLogDefinition
```

Parameters — START DBE NEWLOG

DBEnvironmentName identifies the DBEnvironment in which you want to initialize one or two new log files. Unless you specify an absolute path name, ALLBASE/SQL assumes the name is relative to your current working directory.

ConnectionName associates a user specified name with this connection. This name must

be unique for each DBEnvironment connection within an application. If a *ConnectionName* is not specified, *DBEnvironmentName* is the default. *ConnectionName* cannot exceed 128 bytes.

MULTI indicates the DBEnvironment can be accessed after log initialization in multiuser mode.

ARCHIVE causes ALLBASE/SQL to initialize a new log in archive mode. If you omit this parameter, the log starts in nonarchive mode.

DUAL causes ALLBASE/SQL to maintain two separate logs, preferably on different media. Keeping the log files on separate media ensures that a media failure on one device leaves the other log undamaged. Each log write operation is performed on both logs. Normally, only one log is read; but if an error is encountered, ALLBASE/SQL switches to the other log. Data integrity is maintained provided at least one good copy of each log record is on at least one of the logs.

AUDIT Identifies the DBEnvironment as one that will have audit logging performed on it with the elements specified in the AUDIT ELEMENTS clause. This causes ALLBASE/SQL to create audit log records as well as normal log records in the log file so that the database can be audited.

DataBufferPages specifies the number of 4096-byte data buffer pages to be used. Data buffer pages hold index and data pages.

You can request up to 50,000 data buffer pages. The minimum number of data buffer pages is 15. The default number is 100. The total number of data buffer pages and runtime control block pages cannot exceed 256 Mbytes.

LogBufferPages specifies the number of log buffer pages to be used. You as can request from 24 to 1024 log buffer pages, limited by the amount of storage available. The default number of log buffer pages is 24.

MaxTransactions specifies the maximum number of *concurrent* transactions to be supported. You can specify a value from 2 to 240. The default is 50. This value overrides the maximum value stored in the DBECon file. Any attempt to start a transaction beyond the maximum limit waits for the specified TIMEOUT and returns an error if TIMEOUT is exceeded. For each user logged on to the system at any one time, you should allow 2 concurrent transactions for just being connected to the DBE.

MAXIMUM TIMEOUT specifies the maximum user timeout value. This value temporarily overrides the maximum stored in the DBECon file. When no value is specified, the DBECon file value is the default.

DEFAULT TIMEOUT specifies the default user timeout value. This value temporarily overrides the maximum stored in the DBECon file. When no value is specified, the DBECon file value is the default.

TimeoutValue is an integer literal greater than zero. If the *TimeoutValue* is not qualified by MINUTES, SECONDS is assumed. If representing seconds, *TimeoutValue* must be in the range of 1 to 2,147,483,647. If representing minutes, *TimeoutValue* must be in the range of 1 to 35,791,394.

ControlBlockPages specifies the number of runtime control blocks to be allocated. Any value specified here temporarily overrides the value specified in the DBECon file.

You can specify a value from 17 to 2,000 pages for this parameter. The default is 37 pages. The total number of data buffer pages and runtime control block pages cannot exceed 256 Mbytes.

DefaultPartitionNumber Specifies the default partition number for the DBEnvironment. *DefaultPartitionNumber* must be in the range 1 and 32767. If NONE is specified, tables assigned to the DEFAULT PARTITION do not generate audit log records. no tables in the DBEnvironment are prepared for audit logging and no operation done on these tables is logged. See the CREATE TABLE and ALTER TABLE statements for information on assigning a partition for a table.

CommentPartitionNumber Specifies the partition number for comments made in the DBEnvironment. *CommentPartitionNumber* must be a number between 1 and 32767. If no COMMENT PARTITION is specified, DEFAULT is implied.

If the comment partition is DEFAULT and the default partition number is later changed in a START DBE NEWLOG statement (but the comment partition is not changed from DEFAULT), the comment partition number will also change to the new default partition number.

MaximumNumberOfPartitions Specifies the maximum number of partitions for the DBEnvironment. *MaximumNumberOfPartitions* is required to be a number between 1 and 831. This number indicates the number of partition instances the DBEnvironment is expected to track.

For audit logging purposes, the number of partition instances is calculated as the sum of the number of DATA partitions and the number of elements (except the DATA audit element) specified in the AUDIT ELEMENTS clause. Specifying ALL audit elements (see below) includes 6 elements, implying that 6 partitions are used. Set this value only as high as needed so that unnecessary space is not reserved unless you plan for more partitions or audit elements.

AuditName Specifies the name of this audit DBEnvironment. *AuditName* is limited to 8 bytes. This clause must be specified if AUDIT LOG has been specified. The *AuditName* appears in outputs of the Audit Tool.

AUDIT ELEMENTS Specifies the types of audit logging that will be done for the database. If this clause is omitted and AUDIT LOG is specified, DATA AUDIT ELEMENTS is implicit. The audit elements are as follow:

COMMENT	This permits use of the LOG COMMENT statement in the DBEnvironment. Comments are logged to the defined COMMENT PARTITION. If this element is not chosen, the LOG COMMENT statement returns an error.
DATA	This is the default element. It causes audit log records to be generated for any data operations (INSERT, UPDATE,

or DELETE) on tables that are in an audit partition of the DBEnvironment other than NONE. (Tables can be specified to be in partition NONE and thus not participate in the audit logging process.)

DEFINITION This includes audit logging of the following statements:

```
CREATE TABLE
ALTER TABLE
DROP TABLE
CREATE INDEX
DROP INDEX
CREATE VIEW
DROP VIEW
CREATE RULE
DROP RULE
CREATE PROCEDURE
DROP PROCEDURE
TRANSFER OWNERSHIP
CREATE GROUP
DROP GROUP
CREATE DBEFILESET
DROP DBEFILESET
CREATE PARTITION
DROP PARTITION
TRUNCATE TABLE
```

STORAGE This includes audit logging of the following statements:

```
CREATE DBEFILE
DROP DBEFILE
ADD DBEFILE TO DBEFILESET
REMOVE DBEFILE FROM DBEFILESET
CREATE TEMPSPACE
DROP TEMPSPACE
```

AUTHORIZATION This includes audit logging of the following statements:

```
GRANT
REVOKE
ADD TO GROUP
REMOVE FROM GROUP
```

SECTION This includes audit logging of the creation and deletion of permanent sections. Permanent sections are created when a program is preprocessed, and are deleted by the DROP MODULE statement. Logging of section creation does not include any SETOPT information associated with the section. See the SETOPT statement in this chapter.

ALL This is equivalent to specifying COMMENT DATA DEFINITION STORAGE AUTHORIZATION SECTION AUDIT ELEMENTS as described above.

NewLogDefinition is a clause that provides ALLBASE/SQL with the information needed to create one or more new log files. The syntax for this clause is presented in the next section.

SQL Syntax — NewLogDefinition

```
LOG DBEFILE DBELog1ID [AND DBELog2ID]
WITH PAGES = DBELogSize,
NAME = 'SystemFileName1' [AND 'SystemFileName2']
```

Parameters — NewLogDefinition

`LOG DBEFILE` describes the two log files if the DUAL LOG option is specified, or a single log file otherwise.

DBELog1ID and *DBELog2ID* are the basic names identifying the log files.

DBELogSize specifies the number of 512-byte pages in one log file. If dual logging is used, both logs must be the same size. The DBE log size should be at least 250 pages and no greater than 524,287 pages. The default is 250. If you choose an odd number of pages, the number is rounded up to an even number.

SystemFileName1 and *SystemFileName2* identify how the logs are known to the operating system. The logs are created relative to the DBECon file directory unless an absolute path name is specified. If a log file by the same name already exists, use SQLUtil to purge it before issuing the `START DBE NEWLOG` statement.

Description

- The usual reason for using `START DBE NEWLOG` is to increase or decrease log file space or to invoke dual logging.
- The logs are always created in the same group and account as the DBEnvironment.
- When you choose an odd number of log pages using the WITH PAGES clause of the new log definition, the number is rounded up to an even number, which is displayed in SHOWLOG.
- Do not use this statement unless you are certain that the preceding termination of ALLBASE/SQL was normal and all active sessions terminated normally. Before using the `START DBE NEWLOG` statement, it is recommended that you issue a `START DBE` statement in single user mode to ensure the DBEnvironment is in a consistent state before the existing log(s) are disassociated from the DBEnvironment.
- Use the ARCHIVE option *only* as a part of a static backup procedure with archive logging. Refer to the "Backup and Recovery" chapter in the *ALLBASE/SQL Database Administration Guide* for more information. The preferred method for starting archive logging is to use the `SQLUtil STOREONLINE` command after initial loading of the DBEnvironment is complete.
- No DBE sessions for the DBEnvironment can be in effect when this statement is processed.
- Timeout values set in the `START DBE NEWLOG` statement remain in effect only as long as there is a DBEnvironment session connected to the DBEnvironment, and do not modify the values stored in the DBECon file.

- If no MAXIMUM TIMEOUT limit is specified, the MAXIMUM TIMEOUT limit stored in the DBECon file remains in effect. If no DEFAULT TIMEOUT value is specified, the DEFAULT TIMEOUT value stored in the DBECon file remains in effect.
- If MAXIMUM TIMEOUT = NONE, infinity (no timeout) is assumed. If DEFAULT TIMEOUT = MAXIMUM, the value of MAXIMUM TIMEOUT is assumed. The DEFAULT TIMEOUT value may not exceed the MAXIMUM TIMEOUT value.
- The following parameters defined in the `START DBE NEW` statement are stored in the DBECon file:
  - DBEnvironment language
  - User mode (single versus multi)
  - Number of data buffer pages
  - Number of log buffer pages
  - Maximum transactions
  - Maximum timeout value
  - Default timeout value
  - Number of runtime control block pages
  - DBEFile0 system file name
  - Log system file name(s)
  - Audit logging (chosen versus not)
  - Audit name
  - Audit elements
  - Default partition
  - Comment partition
  - Maximum number of partitions
- You can reconfigure a DBEnvironment by using SQLUtil to alter DBECon file parameters. All parameters except the audit information (logging, audit elements, name, default, comment and maximum partition), or the name of the DBECon file and DBEFile0 may be changed. Refer to the *`ALLBASE/SQL Database Administration Guide`* for additional information.
- If AUDIT LOG is specified, the clauses AUDIT NAME, DEFAULT PARTITION, and MAXPARTITIONS must also be specified. Further, if no AUDIT ELEMENTS are specified, DATA is used as a default.
- Use of the clause ALL AUDIT ELEMENTS implies specification of all of the audit elements.
- The usual reason for using `START DBE NEWLOG` is to increase or decrease log file space, to invoke dual logging or audit logging, or to alter audit logging parameters.
- Audit parameters set in the `START DBE NEWLOG` statement modify the values stored in the DBECon file.

- If an audit parameter is not specified in the statement, the audit parameter remains unchanged. The parameters AUDIT NAME, DEFAULT PARTITION, MAXPARTITIONS, COMMENT PARTITION, and AUDIT ELEMENTS can be changed at any time through the START DBE NEWLOG statement.
- If AUDIT LOG is not specified in this statement, the default is that it is disabled. Thus if the DBEnvironment had audit logging enabled and then specified a START DBE NEWLOG statement without AUDIT LOG, audit logging would then be disabled.
- Changing MAXPARTITIONS on a START DBE NEWLOG prevents roll forward recovery through prior log files, since their structure will differ from the new logs' structure. The same is true of going from non-audit logs to audit logs and vice versa. Therefore, it is recommended that an audit DBEnvironment be designed with a large enough MAXPARTITIONS when it is created.
- Additional log files should be created with the SQLUtil ADDLOG command.
- Refer to the *ALLBASE/SQL Database Administration Guide* for additional information on log file management.

Authorization

You need to be the DBECreator or super-user to issue the START DBE NEWLOG statement. hpdb must have write permission in the target directory for the log file(s).

Example

The DBEnvironment is restored to a consistent state. Any transactions incomplete when the DBEnvironment was last shut down are rolled back, and work done by completed transactions is committed.

```
START DBE '../sampledb/PartsDBE'
```

SQLUtil is used to delete the existing log files: PartsLg1 and PartsLg2.

```
STOP DBE

The log files are reinitialized.

START DBE '../sampledb/PartsDBE' MULTI NEWLOG DUAL LOG
          LOG DBEFILE PartsDBELog1 AND PartsDBELog2
            WITH PAGES = 250, NAME = 'PartsLg1' AND 'PartsLg2'
```

The DBEnvironment is restored to a consistent state. Any transactions incomplete when the DBEnvironment was last shut down are rolled back, and work done by completed transactions is committed.

```
START DBE '../sampledb/PartsDBE'

STOP DBE
```

After the DBEnvironment is stopped, the log files can be purged.

New log files are reinitialized and audit logging is enabled.

```
START DBE '../sampledb/PartsDBE' MULTI NEWLOG DUAL
          AUDIT LOG,
          AUDIT NAME = 'PrtsDBE1',
          DEFAULT PARTITION = 1,
          MAXPARTITIONS = 20,
          DATA AUDIT ELEMENTS,
          LOG DBEFILE PartsDBELog1 AND PartsDBELog2
            WITH PAGES = 1000, NAME = 'PartsLg1' AND 'PartsLg2'
```

You must create additional log files with the `SQLUtil ADDLOG` command.

STOP DBE

The STOP DBE statement concludes ALLBASE/SQL operations and shuts down DBEnvironment operations.

Scope

ISQL or Application Programs

SQL Syntax

```
STOP DBE
```

Description

- Any transactions in progress are aborted, but their changes are not backed out until the START DBE statement is processed.
- A checkpoint is taken.
- Any locks still held are released. Any cursors still open are closed.

Authorization

You must have DBA authority to use this statement.

Example

Two users establish DBE sessions.

```
CONNECT TO '../sampledb/PartsDBE'
CONNECT TO '../sampledb/PartsDBE'
```

The DBA shuts down ALLBASE/SQL, and the two DBE sessions are aborted. Any incomplete transactions are rolled back when the DBEnvironment is next started up.

```
STOP DBE
```

TERMINATE QUERY

The TERMINATE QUERY statement terminates a running QUERY.

Scope

ISQL or Application Programs

SQL Syntax

```
TERMINATE QUERY FOR {CID ConnectionID
                     XID TransactionID}
```

Parameters

CID*ConnectionID* identifies a specific connection in which the 'query' to be terminated is running.

XID*TransactionID* identifies a specific transaction in which the 'query' to be terminated is running.

Description

- A 'query' in this case refers to a command being executed.
- The current command in progress for the specified connection or transaction is terminated and any changes are backed out.

Authorization

You can terminate a query, if it is your own query, or if you have DBA authority. Also the TERMINATE AT QUERY LEVEL option must have been set for the specified connection or transaction. See SET TRANSACTION or SET SESSION for more information on the TERMINATE AT QUERY LEVEL option.

Example

```
TERMINATE QUERY FOR CID ConnectionID3
```

TERMINATE TRANSACTION

The TERMINATE TRANSACTION statement terminates a given transaction.

Scope

ISQL or Application Programs

SQL Syntax

```
TERMINATE TRANSACTION FOR {CID ConnectionID
                           XID TransactionID}
```

Parameters

CID *ConnectionID* identifies the specific connection in which the transaction to be terminated is running.

XID *TransactionID* identifies a specific transaction to be terminated.

Description

- The transaction in progress for the connection is terminated and any changes are backed out.

Authorization

You can terminate a transaction if it is your own transaction, or if you have DBA authority. Also the TERMINATE AT QUERY LEVEL or TERMINATE AT TRANSACTION LEVEL option must have been set for the specified connection or transaction. See SET TRANSACTION or SET SESSION for more information on these options.

Example

```
TERMINATE TRANSACTION FOR CID ConnectionID3
```

TERMINATE USER

The TERMINATE USER statement terminates one or more DBE sessions associated with your user name or another user name.

Scope

ISQL or Application Programs

SQL Syntax

```
TERMINATE USER {DBEUserID
                SessionID
                CID ConnectionID}
```

Parameters

DBEUserID identifies the user to terminate *all* sessions for. Users currently on the system appear in the system view SYSTEM.USER.

SessionID identifies a specific session to be terminated. Session identifiers can be found in the system view SYSTEM.USER. One user may have several session IDs active at the same time.

CID *ConnectionID* identifies the specific connection to terminate.

Description

- Any transactions in progress in the session(s) are terminated and any changes are backed out. Any locks still held are released, and any cursor still open is closed.

Authorization

You can terminate a session if it is your own session, or if you have DBA authority.

Example

User1 starts a DBE session SessionID1

```
CONNECT TO '../sampledb/PartsDBE'
```

User1 starts a DBE session SessionID2

```
CONNECT TO '../sampledb/PartsDBE'
```

User2 starts a DBE session SessionID3

```
CONNECT TO '../sampledb/PartsDBE'
```

User2 starts a DBE session SessionID4

```
CONNECT TO '../sampledb/PartsDBE'
```

Both of User1's DBE sessions terminate. Either User1 or a DBA can enter this statement.

```
TERMINATE USER User1
```

One of User2's DBE sessions terminates. Either User2 or a DBA can enter this statement.

```
TERMINATE USER SessionID3
```

TRANSFER OWNERSHIP

The TRANSFER OWNERSHIP statement makes a different user or authorization group or class name the owner of a table, view, procedure, or authorization group.

Scope

ISQL or Application Programs

SQL Syntax

```
TRANSFER OWNERSHIP OF {[TABLE][Owner.]TableName
                       [VIEW][Owner.]ViewName
                       PROCEDURE [Owner.]ProcedureName
                       GROUP GroupName                  } TO NewOwnerName
```

Parameters

[TABLE][*Owner.*]*TableName* is the name of a table to transfer. All indexes, constraints and rules are also transferred.

[VIEW] [*Owner.*]*ViewName* is the name of a view to transfer.

PROCEDURE [*Owner.*]*ProcedureName* is the name of a procedure to transfer.

GROUP *GroupName* is the name of an authorization group to transfer.

NewOwnerName designates the new owner. The new owner can be a user or an authorization group or a class name.

Description

- The TRANSFER OWNERSHIP statement may invalidate stored sections. Refer to the *ALLBASE/SQL Databast Administration Guide* for additional information on the validation of stored sections.
- You cannot use this statement on system tables or system views.
- Transferring ownership of a table changes the owner's grants to have the new owner as grantor.
- Transferring ownership of a table drops any views based on the table as well as revoking all authorities related to the views.
- Indexes and rules are owned by the owner of the table with which they are associated. When the owner of a table is transferred, then the owner of the indexes and rules associated with it are automatically transferred.

Authorization

You can transfer ownership of a table, view, procedure, or authorization group if you have OWNER authority for that table, view, procedure, or group, or if you have DBA authority.

Transfers of ownership for tables involving referential constraints are subject to the following additional considerations:

- The new owner must have the REFERENCES or DBA authorities necessary to allow ownership of a table containing such constraints. If the new owner does not have the needed authorities, the transfer is not allowed.
- The name of any constraint or rule defined on the table must not already be in use by the new owner.
- The new owner is dependent on these authorizations for the duration of the ownership (the old dependencies are dropped). The authorities cannot be removed from the new owner by the `REVOKE, REMOVE FROM GROUP`, or `DROP GROUP` statements.

Example

```
CREATE PUBLIC TABLE Parts
                    (PartNumber CHAR(16) NOT NULL,
                    PartName CHAR(30),
                    SalesPrice DECIMAL(10,2))
                  IN WarehFS
```

The table is owned by the DBEUserID of its creator.

```
TRANSFER OWNERSHIP OF Parts TO PurchDB
```

Now the table is owned by the class named PurchDB.

TRUNCATE TABLE

The TRUNCATE TABLE statement deletes all rows from the specified table.

Scope

ISQL, Application Programs, or Stored Procedures

SQL Syntax

```
TRUNCATE TABLE [Owner.]TableName
```

Parameters

[*Owner*.]*TableName* identifies the table whose rows are deleted.

Description

- Use the TRUNCATE TABLE when you want to delete all rows from a table, yet leave the table's structure intact. The TRUNCATE TABLE statement is faster than the DELETE statement and generates fewer log records.
- The table definition is not removed or modified. All indexes, views, constraints, rules, default values, and authorizations defined for the table are unchanged.
- The DBEFile space occupied by the table cannot be reused until the transaction ends.
- The DDL (data definition language) flag must be set to YES. Use the ALTDBE command in SQLUtil to set the DDL flag.
- If audit logging is enabled and the DEFINITION AUDIT ELEMENT option is specified, then audit log records are generated.
- If a constraint violation occurs when constraint checking is deferred, a warning is generated and the rows are deleted unless the transaction is explicitly rolled back. Should the violation occur when constraint checking is set to immediate, an error is generated and the rows are not deleted.
- All sections that access the specified table are invalidated.
- If the table is specified by a referential constraint, it may be more efficient to drop the constraint, issue the TRUNCATE TABLE statement, and specify the constraint again.
- Rules are automatically disabled during execution of the TRUNCATE TABLE statement.

Authorization

You can issue this statement if you have OWNER authority for the table or if you have DBA authority.

Example

The following statement deletes all rows from the PurchDB.Parts table:

```
TRUNCATE TABLE PurchDB.Parts
```

UPDATE

The UPDATE statement updates the values of one or more columns in all rows of a table or in rows that satisfy a search condition.

Scope

ISQL or Application Programs

SQL Syntax

```
UPDATE {[Owner.]TableName
        [Owner.]ViewName)
SET {ColumnName = {Expression
                   'LongColumnIOString'
                    NULL                } } [,...]
[WHERE SearchCondition ]
```

Parameters

[Owner.]TableName specifies the table to be updated.

[Owner.]ViewName specifies a view; the table on which the view is based is updated. Refer to the CREATE VIEW statement for restrictions governing updates via views.

ColumnName designates a column to be updated. You can update several columns of the same table with a single UPDATE statement.

Expression is any expression that does not contain an aggregate function or a LONG column (except via the long column function). The expression is evaluated for each row qualifying for the update operation. The data type of the expression must be compatible with the column's data type.

'*LongColumnIOString*' specifies the input and output locations for the LONG data. The syntax for this string is presented in a separate section below.

NULL puts a null value in the specified column of each row satisfying the WHERE clause. The column must allow null values.

SearchCondition specifies a search condition; the search condition cannot contain an aggregate function. All rows for which the search condition is true are updated as specified in the SET clause. Rows that do not satisfy the search condition are not affected. If no rows satisfy the search condition, the table is not changed.

Description

- If the WHERE clause is omitted, all rows of the table are updated as specified by the SET clause.
- If the WHERE clause is present, then the search condition is evaluated for each row of

the table before updating any row. Each subquery in the search condition is effectively executed for each row of the table, and the results used in the application of the search condition to the given row. If any executed subquery contains an outer reference to a column of the table, the reference is to the value of that column in the given row.

- If ALLBASE/SQL detects an error during a multiple-row UPDATE operation, the error handling behavior depends on the setting of the SET DML ATOMICITY and the SET CONSTRAINTS statements. Refer to the discussion of these statements in this chapter.
- No error or warning condition is generated by ALLBASE/SQL when a character or binary string is truncated during an UPDATE operation.
- Using UPDATE with views requires that the views be updatable. See "Updatability of Queries" in Chapter 3 , “SQL Queries.”
- The target table of the UPDATE is designated by *TableName* or is the base table of *ViewName*. This target table must be updatable and must *not* be identified in a FROM clause of any subquery contained in the *SearchCondition*.
- A table on which a unique constraint is defined cannot contain duplicate rows.
- An update of a primary key column in either a referential or unique constraint will fail if any of the rows being updated are currently referred to by any table's foreign key row or if any of the rows being updated ends up matching the value of another unique row. In order to update such primary key rows, the foreign keys must be changed to refer to other primary keys, changed to a value of NULL, or deleted. An update of a foreign key column will fail if it leaves a non-NULL foreign key row without any matching primary key row.
- Integrity constraints on tables or views are enforced on a statement level basis, when SET DML ATOMICITY and SET CONSTRAINTS are at their default values. Thus it is possible to update constraint keys using SET clauses like the following:

  ```
  SET Column1 = Column1 + 1
  ```

  even when the initial values of Column1 are a set of sequential integers, such as 1, 2, 3, 4 (which causes a temporary unique constraint violation). If at the end of the UPDATE statement (that is, after all rows have been incremented), the unique constraint is satisfied, no error message is generated.
- Rows being updated must not cause the search condition of the table check constraint to be false and must cause the search condition of the view check constraint to be true when error checking is done.
- Rows being updated in the table through a view having a WITH CHECK OPTION must be visible through the query expression of the view and any underlying views, in addition to satisfying any constraints of the table. Refer to the "Check Constraints" section of the "Constraints, Procedures, and Rules" chapter.
- Rules defined with a *StatementType* of UPDATE will affect UPDATE statements performed on the rules' target tables. Rules defined with a *StatementType* of UPDATE including a list of column names will affect only those UPDATE statements performed on the rules' target tables that include at least one of the columns in the UPDATE's SET clause. When the UPDATE is performed, ALLBASE/SQL considers all the rules defined for that table with the UPDATE *StatementType* and a matching

column. If the rule has no condition, it will fire for all rows affected by the statement and invoke its associated procedure with the specified parameters on each row. If the rule has a condition, it will evaluate the condition on each row. The rule will fire on rows for which the condition evaluates to TRUE and invoke the associated procedure with the specified parameters for each row. Invoking the procedure could cause other rules, and thus other procedures, to be invoked if statements within the procedure trigger other rules.

- If a DISABLE RULES statement is issued, the UPDATE statement will not fire any otherwise applicable rules. When a subsequent ENABLE RULES is issued, applicable rules will fire again, but only for subsequent UPDATE statements, not for those rows processed when rule firing was disabled.
- In a rule defined with a *StatementType* of UPDATE, any column reference in the *Condition* or any *ParameterValue* that specifies the *OldCorrelationName* will refer to the value of the column before the SET clause assignment is performed on it. Any column reference that specifies the *NewCorrelationName* or *TableName* will refer to the value of the column after the SET clause assignment is performed on it.
- The set of rows to be affected by the UPDATE statement is determined before any rule fires, and this set remains fixed until the completion of the rule. If the rule adds to, deletes from, or modifies this set, such changes are ignored.
- When a rule is fired by this statement, the rule's procedure is invoked after the changes have been made to the database for that row and all previous rows. The rule's procedure, and any chained rules, will thus see the state of the database with the current partial execution of the statement.
- If an error occurs during processing of any rule considered during execution of this statement (including execution of any procedure invoked due to a rule firing), the statement and any procedures invoked by any rules will have no effect. Nothing will have been altered in the DBEnvironment as a result of this statement or the rules it fired. Error messages are returned in the normal way.

SQL Syntax — LongColumnIOString

```
{ [<{[PathName/]FileName
     %SharedMemoryAddress}]
  [{>
    >>
    >!}[PathName/]{FileName
                   CharString$
                   CharString$ CharString}
   >% {SharedMemoryAddress
       $                                  } ] } |...|
```

Parameters — LongColumnIOString

< [*PathName*/] *FileName* is the location of the input file.

<% *SharedMemoryAddress* is the shared memory address where the input is located.

> specifies that output is placed in the following file. If the file already exists, it is not overwritten nor appended to, and an error is generated.

>> specifies that output is appended to the following file name. If the file does not exist, it is created.

>! specifies that output is placed in the following file name. If the file already exists, it is overwritten.

>% *SharedMemoryAddress* is the shared memory address where the output is placed.

>%$ is the shared memory address, determined by ALLBASE/SQL, where the output is placed.

$ is the wildcard character that represents a random, five-byte alphanumeric character string generated by ALLBASE/SQL. This is a file name.

Description — LongColumnIOString

- The input device must have a permission allowing the login user to access it. For example, if the file belongs to the login user, permission must be at least 400. If the file belongs to another user, in a different group, permission must be at least 004.
- When an output device has been specified and it exists prior to a SELECT or FETCH statement, ALLBASE/SQL does not change the file's owner or permission.
- The output device, if it does not exist prior to a SELECT or FETCH statement, is created with the following characteristics.

Table 12-2. Default Output Device Characteristics

Device Type	Permission	UserID (uid)	GroupID (gid)
OUTPUT create	700	Current user login id	Current user login group
OUTPUT append	200	Current user login id	Current user login group
OUTPUT overwrite	200	Current user login id	Current user login group

- If the output device exists prior to a SELECT or FETCH statement, in order for ALLBASE/SQL to access it for append or overwrite, the above characteristics are recommended.
- When you specify a portion of the output file name in conjunction with the wildcard character ($), a five-byte, alphanumeric character string replaces the wildcard. The wildcard character can appear in any position of the output device name *except* the first. The maximum file name being 14 bytes, you can specify 9 bytes of the device name.
- When no portion of the output device name is specified, the default file name, *tmp$.LF*, is used. The wildcard character ($) indicates a random, five-byte, alphanumeric character string. This file is created in the local directory.
- The wildcard character, whether user specified or part of the default output device name, is a unique five-byte, alphanumeric character string.
- When a file is used as the LONG column input or output device and you do not give it a specific path name in the LONG column I/O string, the default is the path where ISQL

or your program is running.

- The output device cannot be overwritten with a SELECT or FETCH statement unless you use the INSERT or UPDATE statement with the overwrite option.
- LONG columns cannot be used as follows:
  - In a WHERE clause.
  - In a type II INSERT statement.
  - Remotely through ALLBASE/NET.
  - As hash or B-tree index key columns.
  - In a GROUP BY, ORDER BY, DISTINCT, or UNION clause.
  - In an expression.
  - In a subquery.
  - In aggregate functions (AVG, SUM, MIN, MAX).
  - As columns to which integrity constraints are assigned.
  - With the DEFAULT option of the CREATE or ALTER TABLE statements.
- If no input device is specified, only output information of LONG columns is reset.
- If no output device is specified, only value is reset.

Authorization

You can update a table if you have UPDATE authority for the entire table, UPDATE authority for all of the columns specified in the SET clause, OWNER authority for the table, or DBA authority.

To update using a view, the authority needed as described below depends on whether you own the view:

- If you own the view, you need UPDATE or OWNER authority for the base table, or UPDATE authority for each column of the base table to be updated as specified in the SET clause, or DBA authority.
- If you do not own the view, you must have UPDATE authority for the view, or UPDATE authority for each column of the view specified in the SET clause, or DBA authority. In addition, the owner of the view must have UPDATE or OWNER authority with respect to the view's definition, or the owner must have DBA authority.
- Using UPDATE with views requires that the views be updatable. See "Updatability of Queries" in the "SQL Queries" chapter.

Example

```
UPDATE PurchDB.Parts SET SalesPrice = SalesPrice * 1.25
WHERE SalesPrice > 500.00
```

UPDATE STATISTICS

The UPDATE STATISTICS statement updates the system catalog to reflect a table's current characteristics, such as the number of rows and average row size. ALLBASE/SQL uses these statistics to choose an optimal way to process a query.

Scope

ISQL or Application Programs

SQL Syntax

```
UPDATE STATISTICS FOR TABLE {[Owner.]TableName
                            SYSTEM.SystemViewName}
```

Parameters

[*Owner.*]*TableName* identifies a table.

SYSTEM.*SystemViewName* identifies a system view.

Description

- The UPDATE STATISTICS statement affects specific columns in certain system catalog views:

View Name	Columns Affected
SYSTEM.DBEFILE	DBEFUPAGES
SYSTEM.DBEFILESET	DBEFSUPAGES
SYSTEM.COLUMN	AVGLEN
SYSTEM.INDEX	CCOUNT
	NPAGES
SYSTEM.TABLE	AVGLEN
	NPAGES
	NROWS
	USTIME

- Any sections that reference a table named in the UPDATE STATISTICS statement are marked invalid, but are revalidated the next time they are executed or the VALIDATE statement is issued if access and authorization criteria are satisfied.
- Use this statement sparingly before preprocessing, after creating an index, and after periods of heavy update activity. For more information, on the UPDATE STATISTICS statement, refer to the *ALLBASE/SQL Performance Guidelines*.

- The only views this statement works for are system views. Refer to the *ALLBASE/SQL Database Administration Guide* for a description of the system views.
- UPDATE STATISTICS *cannot* be used with pseudotables — SYSTEM.ACCOUNT, SYSTEM.CALL, SYSTEM.COUNTER, SYSTEM.TRANSACTION, and SYSTEM.USER.
- You may find it convenient to use the VALIDATE statement after an UPDATE STATISTICS. If you issue both statements during a period of low activity for the DBEnvironment, the optimizer will have current statistics on which to base its calculations, with minimal performance degradation.

Authorization

You can issue this statement if you have OWNER authority for the table or if you have DBA authority.

Example

You issue this statement after periods of heavy data update activity in order to keep access paths optimal.

```
UPDATE STATISTICS FOR TABLE PurchDB.Orders
```

UPDATE WHERE CURRENT

The UPDATE WHERE CURRENT statement updates the values of one or more columns in the current row associated with a cursor. The current row is the row pointed to by a cursor after the FETCH or REFETCH statement is issued.

Scope

Application Programs Only

SQL Syntax

```
UPDATE {[Owner.]TableName
        [Owner.]ViewName}
SET { ColumnName = {Expression
                    'LongColumnIOString'
                    NULL                  }}[,...]
WHERE CURRENT OF CursorName
```

Parameters

[*Owner.*]*TableName*	specifies the table to be updated.
[*Owner.*]*ViewName*	specifies a view; the table on which the view is based is updated. Refer to the CREATE VIEW statement for restrictions governing updates via views.
ColumnName	designates a column to be updated. You can update several columns of the same table with a single UPDATE WHERE CURRENT statement.
Expression	is any expression that does not contain an aggregate function or a LONG column (except via a long column function). The data type of the expression must be compatible with the column's data type.
'*LongColumnIOString*'	specifies the input and output locations for the LONG data. The syntax for this string is presented in a separate section below.
NULL	puts a null value in the specified column. The column must allow null values.
CursorName	designates an opened cursor. The current row of the cursor is updated as specified by the SET clause. The column(s) named in the SET clause must also be named in the FOR UPDATE clause of the DECLARE CURSOR statement defining the cursor. After the update, the row updated remains the current row.

Description

- This statement cannot be used interactively and should not be used in conjunction with rows fetched using the BULK FETCH statement.
- For constraint violations, the error handling behavior depends on the setting of the SET CONSTRAINTS statement. Refer to the discussion of this statement in this chapter.
- No error or warning condition is generated by ALLBASE/SQL when a character or binary string is truncated during an UPDATE operation.
- Using UPDATE WHERE CURRENT OF CURSOR requires that the cursor be based on an updatable query. See "Updatability of Queries" in the "SQL Queries" chapter.
- The target table of the UPDATE WHERE CURRENT is designated by *TableName* or is the base table underlying the *ViewName*. The base table restrictions that govern updates via a cursor were presented in the description of the DECLARE CURSOR statement.
- A table on which a unique constraint is defined cannot contain duplicate rows.
- For constraint violations, the error handling behavior depends on the setting of the SET CONSTRAINTS statement. Refer to the discussion of this statement in this chapter.
- An update of a primary key column in either a referential or unique constraint will fail if any of the rows being updated are currently referred to by any table's foreign key row or if any of the rows being updated ends up matching the value of another unique row. In order to update such primary key rows, the foreign keys must be changed to refer to other primary keys, changed to a value of NULL, or deleted. An update of a foreign key column will fail if it leaves a non-NULL foreign key row without any matching primary key row.
- Rows being updated must not cause the search condition of the table check constraint to be false and must cause the search condition of the view check constraint to be true when error checking is done.
- Rows being updated in the table through a view having a WITH CHECK OPTION must still be visible through the query expression of the check constraint of the view and any underlying views, in addition to satisfying any constraints of the table. Refer to the "Check Constraints" section of Chapter 4 , “Constraints, Procedures, and Rules,” for a further discussion on check constraints.
- A rule defined with a *StatementType* of UPDATE will affect UPDATE WHERE CURRENT statements performed on the rules' target tables. Rules defined with a *StatementType* of UPDATE including a list of column names will affect only those UPDATE WHERE CURRENT statements performed on the rules' target tables that include at least one of the columns in their SET clause. When the UPDATE WHERE CURRENT is performed, ALLBASE/SQL considers all the rules defined for that table with the UPDATE *StatementType* and a matching column. If the rule has no condition, it will fire for the current row and invoke its associated procedure with the specified parameters. If the rule has a condition, it will evaluate the condition and fire if the condition evaluates to TRUE, invoking the associated procedure with the specified parameters for the current row. Invoking the procedure could cause other rules, and thus other procedures, to be invoked if statements within the procedure trigger other rules.
- If a DISABLE RULES statement is in effect, the UPDATE WHERE CURRENT statement

will not fire any otherwise applicable rules. When a subsequent ENABLE RULES is issued, applicable rules will fire again, but only for subsequent UPDATE WHERE CURRENT statements, not for those rows processed when rule firing was disabled.

- In a rule defined with a *StatementType* of UPDATE, any column reference in the *Condition* or any *ParameterValue* that specifies the *OldCorrelationName* will refer to the value of the column before the SET clause assignment is performed on it. Any column reference that specifies the *NewCorrelationName* or *TableName* will refer to the value of the column after the SET clause assignment is performed on it.
- When a rule is fired by this statement, the rule's procedure is invoked after the changes have been made to the database for that row. The rule's procedure, and any chained rules, will thus see the state of the database with the current partial execution of the statement.
- If an error occurs during processing of any rule considered during execution of this statement (including execution of any procedure invoked due to a rule firing), the statement and any procedures invoked by any rules will have no effect. Nothing will have been altered in the DBEnvironment as a result of this statement or the rules it fired. Error messages are returned in the normal way.

SQL Syntax — LongColumnIOString

```
{ [<{[PathName/]FileName
     %SharedMemoryAddress}]
  [{>
    >>
    >!}[PathName/]{FileName
                   CharString$
                   CharString$CharString}
   >% {SharedMemoryAddress
       $                                  } ] } |...|
```

Parameters — LongColumnIOString

< [*PathName*/] *FileName* is the location of the input file.

<% *SharedMemoryAddress* is the shared memory address where the input is located.

>	specifies that output is placed in the following file. If the file already exists, it is not overwritten nor appended to, and an error is generated.
>>	specifies that output is appended to the following file name. If the file does not exist, it is created.
>!	specifies that output is placed in the following file name. If the file already exists, it is overwritten.

>% *SharedMemoryAddress* is the shared memory address where the output is placed.

>%$	is the shared memory address, determined by ALLBASE/SQL, where the output is placed.
$	is the wildcard character that represents a random, five-byte alphanumeric character string generated by ALLBASE/SQL. This is a file name.

Description — LongColumnIOString

- The input device must have a permission allowing the login user to access it. For example, if the file belongs to the login user, permission must be at least 400. If the file belongs to another user, in a different group, permission must be at least 004.
- When an output device has been specified and it exists prior to a SELECT or FETCH statement, ALLBASE/SQL does not change the file's owner or permission.
- The output device, if it does not exist prior to a SELECT or FETCH statement, is created with the following characteristics.

Table 12-3. Default Output Device Characteristics

Device Type	Permission	UserID (uid)	GroupID (gid)
OUTPUT create	700	Current user login id	Current user login group
OUTPUT append	200	Current user login id	Current user login group
OUTPUT overwrite	200	Current user login id	Current user login group

- If the output device exists prior to a SELECT or FETCH statement, in order for ALLBASE/SQL to access it for append or overwrite, the above characteristics are recommended.
- When you specify a portion of the output file name in conjunction with the wildcard character ($), a five-byte, alphanumeric character string replaces the wildcard. The wildcard character can appear in any position of the output device name *except* the first. The maximum file name being 14 bytes, you can specify 9 bytes of the device name.
- When no portion of the output device name is specified, the default file name, *tmp$.LF*, is used. The wildcard character ($) indicates a random, five-byte, alphanumeric character string. This file is created in the local directory.
- The wildcard character, whether user specified or part of the default output device name, is a unique five-byte, alphanumeric character string.
- When a file is used as the LONG column input or output device and you do not give it a specific path name in the LONG column I/O string, the default is the path where ISQL or your program is running.
- The output device cannot be overwritten with a SELECT or FETCH statement unless you use the INSERT or UPDATE statement with the overwrite option.
- LONG columns cannot be used as follows:
  - In a WHERE clause.
  - In a type II INSERT statement.
  - Remotely through ALLBASE/NET.
  - As hash or B-tree index key columns.
  - In a GROUP BY, ORDER BY, DISTINCT, or UNION clause.

  - In an expression.
  - In a subquery.
  - In aggregate functions (AVG, SUM, MIN, MAX).
  - As columns to which integrity constraints are assigned.
  - With the DEFAULT option of the CREATE or ALTER TABLE statements.
- If no input device is specified, only output information of LONG columns is reset.
- If no output device is specified, only value is reset.

Authorization

You can update a table if you have UPDATE authority for the entire table, UPDATE authority for all of the columns specified in the SET clause, OWNER authority for the table, or DBA authority.

To update using a view, authority needed depends on whether you own the view:

- If you own the view, you need UPDATE or OWNER authority for the base table, or UPDATE authority for each column of the base table to be updated as specified in the SET clause, or DBA authority.
- If you do not own the view, you must have UPDATE authority for the view, or UPDATE authority for each column of the view specified in the SET clause, or DBA authority. In addition, the owner of the view must have UPDATE or OWNER authority with respect to the view's definition, or the owner must have DBA authority.

Example

A cursor for use in updating values in column QtyOnHand is declared and opened.

```
DECLARE NewQtyCursor CURSOR FOR
 SELECT PartNumber,QtyOnHand FROM PurchDB.Inventory
        FOR UPDATE OF QtyOnHand

OPEN NewQtyCursor
```

Statements setting up a FETCH-UPDATE loop appear next.

```
FETCH NewQtyCursor INTO :Num :Numnul, :Qty :Qtynul
```

Statements for displaying a row to and accepting a new QtyOnHand value from a user go here. The new value is stored in :NewQty.

```
           UPDATE PurchDB.Inventory\
              SET QtyOnHand = :NewQty\
WHERE CURRENT OF NewQtyCursor
.
.
.
CLOSE NewQtyCursor
```

VALIDATE

The VALIDATE statement validates modules and procedures that have already been created.

Scope

ISQL or Application Programs

SQL Syntax

```
VALIDATE [FORCE
          DROP SETOPTINFO]
{MODULE { {[Owner.]ModuleName} [,...]
          {SECTION [ Owner.]ModuleName (SectionNumber)} [,...] }
 PROCEDURE { {[Owner.]ProcedureName} [,...]
             {SECTION [Owner.]ProcedureName (SectionNumber)} [,...] }
 ALL{MODULES
     PROCEDURES} [WITH AUTOCOMMIT]                                    }
```

Parameters

WITH AUTOCOMMIT	executes automatically a COMMIT WORK after each module or procedure is updated.
[*Owner.*]*ModuleName*	identifies the module containing sections to be validated. The owner name is the DBEUserID of the person who preprocessed the program or the owner name specified when the program was preprocessed. The module name is the name stored in the SYSTEM.SECTION view.
[*Owner.*]*ModuleName* (*SectionNumber*)	identifies the section number as well as the module to be validated.
[*Owner.*]*ProcedureName*	identifies the procedure to validate. The owner name is the DBEUserID of the person who created the procedure or the owner name specified when the procedure was created. The procedure name is the name stored in the SYSTEM.SECTION view.
[*Owner.*]*ProcedureName* (*SectionNumber*)	identifies the section number as well as the procedure to be validated.

Description

- When you validate a module or procedure, all the sections within it are checked and validation is attempted. If an embedded SQL statement accesses an object that does not exist or that the module or procedure owner is not authorized to execute, then the corresponding section is marked invalid.
- You may find it convenient to use the VALIDATE statement after an UPDATE

STATISTICS, since UPDATE STATISTICS will invalidate stored sections. If you issue both statements during a period of low activity for the DBEnvironment, the optimizer will have current statistics on which to base its calculations, with minimal performance degradation.

- A temporary section cannot be validated.
- Users can specify the access plan of a query with the SETOPT statement. To validate a module or procedure without the user-specified access plan, include the DROP SETOPTINFO keyword in the VALIDATE statement. The default access plan determined by ALLBASE/SQL is stored in the system catalog instead.
- If a module or procedure cannot be validated, ALLBASE/SQL returns an error.
- If a section is still invalid after revalidation, the module is considered invalid.
- To find the names of procedures with invalid sections, use ISQL to query the SYSTEM.SECTION view with Stype = 0.
- The VALIDATE statement will not revalidate sections that have been stored prior to this release, for example, sections that have been migrated from a previous release. These sections can only be revalidated by running the application to execute all the sections. An alternative is to recreate the module by preprocessing the application again. Thereafter, you can use the VALIDATE statement.
- For detailed information on modules refer to the section "Invalidation and Revalidation of Sections" in the "Maintenance" chapter of the *ALLBASE/SQL Database Administration Guide* and the "Using the Preprocessor" chapter in your ALLBASE/SQL application programming guide.
- For detailed information on procedures, refer to Chapter 4 , “Constraints, Procedures, and Rules.”
- When the WITH AUTOCOMMIT clause is used, a COMMIT WORK statement is executed automatically after each MODULE or PROCEDURE is validated. This can reduce both log space and shared memory requirements for the VALIDATE command.
- When the FORCE clause is used, all sections associated with the MODULE or PROCEDURE are revalidated, regardless of whether they are valid or invalid.
- When the FORCE clause is used with VALIDATE ALL MODULES and VALIDATE ALL PROCEDURES, every stored section in the database is forced to recompile using the latest release. These statements have essentially the same effect as preprocessing every program again that uses the database.

Authorization

You can execute this statement if you have OWNER or RUN authority on a module or you have OWNER or EXECUTE authority for a procedure or if you have DBA authority.

Examples

1. Validating sections in a module

   ALLBASE/SQL validates sections at preprocessing time and run time. To validate a section before running your application, you can use the VALIDATE statement. To find the names of modules with invalid sections, use ISQL to query the SYSTEM.SECTION view.

```
isql=> SELECT Name, Section FROM System.Section
> WHERE valid = 0 and stype = 0;

SELECT Name, Section FROM System.Section WHERE Valid=0 and Stype=0;
--------------------+---------------
NAME                |SECTION
--------------------+---------------
CEXP06              |1
CEXP06              |2
CEXP06              |3

------------------------------------------------------------------------------
   First 3 rows have been selected.
   U[p], d[own], l[eft], r[ight], t[op], b[ottom], pr[int] <n>,or e[nd]>
```

   Three sections of the module named CEX06 are invalid. Issue the VALIDATE statement to attempt validation.

```
isql => VALIDATE MODULE CEXP06;
```

2. Dropping SETOPT access plan

   The following SETOPT statement specifies that every table with an index is accessed with an index scan.

```
isql => SETOPT GENERAL INDEXSCAN;
```

   Validate the CEX09 module, but ignore the access plan specified in the preceding SETOPT statement.

```
isql => VALIDATE DROP SETOPTINFO MODULE CEXP09;
```

3. When the WITH AUTOCOMMIT clause is used, a COMMIT WORK statement is executed automatically after each module or procedure is validated.

```
VALIDATE ALL MODULES WITH AUTOCOMMIT;
VALIDATE ALL PROCEDURES WITH AUTOCOMMIT;
```

4. When the FORCE clause is used, all sections associated with the MODULE or PROCEDURE are revalidated regardless of whether they are valid or invalid.

```
VALIDATE FORCE ALL MODULES WITH AUTOCOMMIT;
VALIDATE FORCE ALL PROCEDURES WITH AUTOCOMMIT;
```

WHENEVER

WHENEVER is a directive used in an application program or a procedure to specify an action to be taken depending on the outcome of subsequent SQL statements.

Scope

Application Programs and Procedures Only

SQL Syntax

```
WHENEVER {SQLERROR
          SQLWARNING
          NOT FOUND  } {STOP
                        CONTINUE
                        GOTO [:]Label
                        GO TO [:]Label}
```

Parameters

SQLERROR	refers to a test for the condition SQLCODE < 0.
SQLWARNING	refers to a test for the condition SQLWARN0 = 'W'.
NOT FOUND	refers to a test for the condition SQLCODE = 100.
STOP	causes a ROLLBACK WORK statement and terminates the application program or procedure, whenever an SQL statement produces the specified condition.
CONTINUE	means no special action is taken automatically when a SQL statement produces the specified condition. Sequential execution will continue.
GOTO [:]*Label*	specifies a label to jump to whenever the condition is found to be true after executing a SQL statement. In an application, the label must conform to the SQL syntax rules for a basic name or any other legitimate label in the host language as well as the requirements of the host language. In a procedure, the label is an integer or a name which conforms to the SQL syntax rules for a basic name. You can optionally include a colon (:) before the label to conform to FIPS 127.1 flagger standards.

Description

- In an application, SQLCODE and SQLWARN0 are fields in the SQLCA or built-in variables. They are structures ALLBASE/SQL uses to return status information about SQL statements. In a procedure, ::sqlcode and ::sqlwarn0 are built-in variables. If the WHENEVER statement is not specified for a condition, the default action is CONTINUE.
- A WHENEVER directive affects all SQL statements that come after it in the source program listing or procedure, up to the next WHENEVER directive for the same condition.
- You can write code of your own to check the SQLCA for error or warning conditions,

whether or not you use the WHENEVER directive.

- This directive cannot be used interactively or with dynamic parameters.

Authorization

You do not need authorization to use the WHENEVER directive.

Example

Execution of the program terminates if the CONNECT TO statement cannot be executed successfully.

```
INCLUDE SQLCA
 .
 .
 .

WHENEVER SQLERROR STOP

CONNECT TO '.../sampledb/PartsDBE'
.
.
.
```

If a row does not qualify, control is passed to the statement labeled 9000.

```
INCLUDE SQLCA
.
.
.
WHENEVER NOT FOUND GO TO 9000

SELECT OrderDate
  FROM PurchDB.Orders
 WHERE OrderNumber = :OrdNum
```

WHILE

The WHILE statement is used to allow looping within a procedure.

Scope

Procedures only

SQL Syntax

```
WHILE Condition DO [Statement; [...]] ENDWHILE;
```

Parameters

Condition specifies anything that is allowed in a search condition except subqueries, column references, host variables, dynamic parameters, aggregate functions, string functions, date/time functions involving column references, long column functions, or TID functions. Local variables, built-in variables, and parameters may be included. See Chapter 9 , “Search Conditions.”

Statement is any SQL statement allowed in a procedure-- including a compound statement.

Description

- *Statement* is executed as long as the *Condition* evaluates to TRUE.
- WHILE statements can be nested.
- Local variables, built-in variables, and parameters can be used in a procedure as host variables are used in an application program.

Authorization

Anyone can use the WHILE statement.

Example

Create and execute a procedure to display all the quantities in the LargeOrders table for a given part:

```
CREATE PROCEDURE ShowOrders AS
   BEGIN
      DECLARE Quantity INTEGER;
      DECLARE PartName CHAR(16);

      DECLARE QtyCursor CURSOR FOR
         SELECT PartName, Quantity
         FROM LargeOrders;
```

```
        OPEN QtyCursor;
        WHILE ::sqlcode <> 100 DO
           FETCH QtyCursor INTO :PartName, :Quantity
           PRINT :PartName;
           PRINT :Quantity;
        ENDWHILE;
        CLOSE QtyCursor;
     END;
  EXECUTE PROCEDURE ShowOrders;
```

A SQL Syntax Summary

This listing of SQL syntax differs from the previous version in the following ways:

- The braces { } and brackets [ ] are shown in standard type size.
- Commands and keywords are shown in **BOLD UPPERCASE** characters

ADD DBEFILE

```
ADD DBEFILE DBEFileName TO DBEFILESET DBEFileSetName
```

ADD TO GROUP

```
ADD {DBEUserID
     GroupName
     ClassName  }[,...]TO GROUP TargetGroupName
```

ADVANCE

```
ADVANCE CursorName [USING [SQL] DESCRIPTOR {SQLDA
                                            AreaName}]
```

ALTER DBEFILE

```
ALTER DBEFILE DBEFileName SET TYPE = {TABLE
                                      INDEX
                                      MIXED }
```

ALTER TABLE

```
ALTER TABLE [Owner.]TableName {AddColumnSpecification
                               AddConstraintSpecification
                               DropConstraintSpecification
                               SetTypeSpecification
                               SetPartitionSpecification     }
```

AddColumnSpecification

```
ADD{ (ColumnDefinition [,...])
      ColumnDefinition        } [CLUSTERING ON CONSTRAINT [ConstraintID]]
```

AddConstraintSpecification

```
ADD CONSTRAINT ({UniqueConstraint
                 ReferentialConstraint
                 CheckConstraint       }[,...])
[CLUSTERING ON CONSTRAINT [ConstraintID1]]
```

DropConstraintSpecification

```
DROP CONSTRAINT {(ConstraintID [,...])
                  ConstraintID         }
```

SetTypeSpecification

```
SET TYPE {PRIVATE
          PUBLICREAD
          PUBLIC
          PUBLICROW  } [RESET AUTHORITY
                        PRESERVE AUTHORITY]
```

SetPartitionSpecification

```
SET PARTITION {PartitionName
               DEFAULT
               NONE          }
```

Assignment (=)

```
{:LocalVariable
 :ProcedureParameter}= Expression;
```

BEGIN

```
BEGIN [Statement;] [...] END;
```

BEGIN ARCHIVE

```
BEGIN ARCHIVE
```

BEGIN DECLARE SECTION

```
BEGIN DECLARE SECTION
```

BEGIN WORK

```
BEGIN WORK [Priority] [RR
                       CS
                       RC
                       RU] [LABEL {'LabelString'
                                   :HostVariable}] [[PARALLEL
                                                     NO       ] FILL]
```

CHECKPOINT

```
CHECKPOINT [:HostVariable
            :LocalVariable
            :ProcedureParameter]
```

CLOSE

```
CLOSE CursorName [USING {[SQL] DESCRIPTOR {SQLDA
                                           Areaname}
                         :HostVariable [[INDICATOR]:Indicator][,...] } ]
```

COMMIT ARCHIVE

```
COMMIT ARCHIVE
```

COMMIT WORK

```
COMMIT WORK [RELEASE]
```

CONNECT

```
CONNECT TO {'DBEnvironmentName'
            :HostVariable1      }[AS {'ConnectionName'
                                      :HostVariable2  }]
[USER {'UserID'
       :HostVariable3}[USING :HostVariable4]]
```

CREATE DBEFILE

```
CREATE DBEFILE DBEFileName WITH PAGES = DBEFileSize, NAME = 'SystemFileName'
[, INCREMENT = DBEFileIncrSize[, MAXPAGES = DBEFileMaxSize]]
[, TYPE = {TABLE
           INDEX
           MIXED }]
```

CREATE DBEFILESET

```
CREATE DBEFILESET DBEFileSetName
```

CREATE GROUP

```
CREATE GROUP [Owner.]GroupName
```

CREATE INDEX

```
CREATE [UNIQUE][CLUSTERING]INDEX [Owner.]Indexname ON
[Owner.]TableName ( {ColumnName [ASC
                                 DESC]}[,...])
```

CREATE PARTITION

```
CREATE PARTITION PartitionName WITH ID = PartitionNumber
```

CREATE PROCEDURE

```
CREATE PROCEDURE [Owner.]ProcedureName [LANG = ProcLangName]
[(ParameterDeclaration [, ParameterDeclaration][...])]
[WITH RESULT ResultDeclaration [, ResultDeclaration ][...]]
AS BEGIN [ProcedureStatement][...] END [IN DBEFileSetName]
```

ParameterDeclaration

```
ParameterName ParameterType [LANG = ParameterLanguage]
[DEFAULT DefaultValue][NOT NULL][OUTPUT[ONLY]]
```

ResultDeclaration

```
ResultType [LANG = ResultLanguage][NOT NULL]
```

CREATE RULE

```
CREATE RULE [Owner.]RuleName
AFTER StatementType [,...][ON
                           OF
                           FROM
                           INTO} [Owner.]TableName
[REFERENCING{OLD AS OldCorrellationName
             NEW AS NewCorrelationName}[...]] [WHERE FiringCondition]
EXECUTE PROCEDURE [OwnerName.]ProcedureName [(ParameterValue [,...])]
[IN DBEFileSetName]
```

CREATE SCHEMA

```
CREATE SCHEMA AUTHORIZATION AuthorizationName [TableDefinition
                                               ViewDefinition
                                               IndexDefinition
                                               ProcedureDefinition
                                               RuleDefinition
                                               CreateGroup
                                               AddToGroup
                                               GrantStatement      ][...]
```

CREATE TABLE

```
CREATE [PRIVATE
        PUBLICREAD
        PUBLIC
        PUBLICROW ]TABLE [Owner.]TableName
[LANG = TableLanguageName]
({ColumnDefinition
  UniqueConstraint
  ReferentialConstraint
  CheckConstraint       }[,...])
[UNIQUE HASH ON (HashColumnName [,...]) PAGES = PrimaryPages
 HASH ON CONSTRAINT [ConstraintID] PAGES = PrimaryPages
 CLUSTERING ON CONSTRAINT [ConstraintID]                    ]
[IN PARTITION {PartitionName
               DEFAULT
               NONE          }]
[IN DBEFileSetName1]
```

Column Definition

```
ColumnName{ColumnDataType
           LongColumnType [IN DBEFileSetName2]}
[LANG = ColumnLanguageName]
[[NOT] CASE SENSITIVE]
[DEFAULT{Constant
         USER
         NULL
         CurrentFunction}]
[NOT NULL [{UNIQUE
            PRIMARY KEY} [CONSTRAINT ConstraintID]]
 REFERENCES RefTableName [(RefColumnName)][CONSTRAINT ConstraintID]
 [...]

 CHECK (SearchCondition) [CONSTRAINT ConstraintID]
 [IN DBEFileSetName3]                                          ][...]
```

Unique Constraint (Table Level)

```
{UNIQUE
 PRIMARY KEY}(ColumnName [,...]) [CONSTRAINT ConstraintID]
```

Referential Constraint (Table Level)

```
FOREIGN KEY (FKColumnName [,...])
REFERENCES RefTableName [( RefColumnName [,...])] [CONSTRAINT ConstraintID]
```

Check Constraint (Table Level)

```
CHECK (SearchCondition) [CONSTRAINT ConstraintID] [IN DBEFileSetName3]
```

CREATE TEMPSPACE

```
CREATE TEMPSPACE TempSpaceName
WITH [MAXFILEPAGES = MaxTempFileSize,]LOCATION ='PhysicalLocation'
```

CREATE VIEW

```
CREATE VIEW [Owner.]ViewName [(ColumnName[,...])]
AS QueryExpression [IN DBEFileSetName]
[WITH CHECK OPTION [CONSTRAINT ConstraintID]]
```

DECLARE CURSOR

```
DECLARE CursorName [IN DBEFileSetName] CURSOR FOR
{ {QueryExpression
   SelectStatementName}[FOR UPDATE OF {ColumnName}[,...]
                        FOR READ ONLY                    ]
  ExecuteProcedureStatement
  ExecuteStatementName                                    }
```

DECLARE *Variable*

```
DECLARE { LocalVariable}[,...] VariableType {LANG = VariableLangName]
[DEFAULT {Constant
          NULL
          CurrentFunction}][NOT NULL]
```

DELETE

```
DELETE [WITH AUTOCOMMIT]FROM {[Owner.]TableName
                              [Owner.]ViewName} [WHERE SearchCondition]
```

DELETE WHERE CURRENT

```
DELETE FROM {[Owner.]TableName
             [Owner.]ViewName} WHERE CURRENT OF CursorName
```

DESCRIBE

```
DESCRIBE [OUTPUT
          INPUT
          RESULT] StatementName {INTO [[SQL] DESCRIPTOR]
                                 USING [SQL] DESCRIPTOR}{SQLDA
                                                         AreaName}
```

DISABLE AUDIT LOGGING

```
DISABLE AUDIT LOGGING
```

DISABLE RULES

```
DISABLE RULES
```

DISCONNECT

```
DISCONNECT {'ConnectionName'
            'DBEnvironmentName'
            :HostVariable
            ALL
            CURRENT             }
```

DROP DBEFILE

```
DROP DBEFILE DBEFileName
```

DROP DBEFILESET

```
DROP DBEFILESET DBEFileSetName
```

DROP GROUP

```
DROP GROUP GroupName
```

DROP INDEX

```
DROP INDEX [Owner.]IndexName [FROM [Owner.]TableName]
```

DROP MODULE

```
DROP MODULE [Owner.]ModuleName [PRESERVE]
```

DROP PARTITION

```
DROP PARTITION PartitionName
```

DROP PROCEDURE

```
DROP PROCEDURE [Owner.]ProcedureName [PRESERVE]
```

DROP RULE

```
DROP RULE [Owner.]RuleName [FROM TABLE [Owner.]TableName]
```

DROP TABLE

```
DROP TABLE [Owner.] TableName
```

DROP TEMPSPACE

```
DROP TEMPSPACE TempSpaceName
```

DROP VIEW

```
DROP VIEW [Owner.] ViewName
```

ENABLE AUDIT LOGGING

```
ENABLE AUDIT LOGGING
```

ENABLE RULES

```
ENABLE RULES
```

END DECLARE SECTION

```
END DECLARE SECTION
```

EXECUTE

```
EXECUTE{StatementName
        [Owner.]ModuleName [(SectionNumber)]}
[USING{[SQL] DESCRIPTOR{[INPUT]{SQLDA
                                AreaName1}
                          [AND OUTPUT {SQLDA
                                       AreaName2}]
                          OUTPUT {SQLDA
                                  AreaName}}
       [INPUT]HostVariableSpecification1
       [AND OUTPUT HostVariableSpecification2]
       OUTPUT HostVariableSpecification
       :Buffer [,:StartIndex[,:NumberOfRows]]         }]
```

HostVariableSpecification

```
:HostVariableName [[INDICATOR]:IndicatorVariable ] [,...]
```

EXECUTE IMMEDIATE

```
EXECUTE IMMEDIATE {'String'
                   :HostVariable}
```

EXECUTE PROCEDURE

```
EXECUTE PROCEDURE [:ReturnStatusVariable = ][Owner.]ProcedureName
[([ActualParameter][,[ActualParameter]][...]) ]
```

ActualParameter

```
[ParameterName = ]ParameterValue [OUTPUT[ONLY]]
```

FETCH

```
[BULK] FETCH CursorName{INTO HostVariableSpecification
                        USING{ [SQL] DESCRIPTOR {SQLDA
                                                 AreaName}
                               HostVariableSpecification   } }
```

BULK HostVariableSpecification

```
:Buffer [,:StartIndex [,:NumberOfRows]]
```

Non-BULK HostVariableSpecification

```
{:HostVariable [[INDICATOR]:Indicator ] } [,...]
```

GENPLAN

```
GENPLAN [WITH (HostVariableDefinition)] FOR
{SQLStatement
 MODULE SECTION [Owner.]ModuleName(Section Number)
 PROCEDURE SECTION [Owner.]ProcedureName(Section Number)}
```

GOTO

```
{GOTO
 GO TO}{Label
        Integer}
```

GRANT

```
GRANT {ALL [PRIVILEGES]
       {SELECT
        INSERT
        DELETE
        ALTER
        INDEX
        UPDATE [({ColumnName}[,...])]
        REFERENCES [({ColumnName}[,...])]}|,...|}
ON {[Owner.]TableName
    [Owner.]ViewName}TO {DBEUserID
                         GroupName
                         ClassName
                         PUBLIC    } [,...][WITH GRANT OPTION]
[BY {DBEUserID
     ClassName}]
```

Grant RUN or EXECUTE Authority

```
GRANT {RUN ON [Owner.]ModuleName
       EXECUTE ON PROCEDURE [Owner.]ProcedureName}TO
{{DBEUserID
  GroupName
  ClassName} [,...]
  PUBLIC            }
```

Grant CONNECT, DBA, INSTALL, MONITOR, or RESOURCE Authority

```
GRANT {CONNECT
       DBA
       INSTALL [AS OwnerID]
       MONITOR
       RESOURCE            } TO {DBEUserID
                                 GroupName
                                 ClassName } [,...]
```

Grant DBEFileSet Authority

```
GRANT {SECTIONSPACE
       TABLESPACE   } [,...] ON DBEFILESET DBEFileSetName TO
{DBEUserID
 GroupName
 ClassName
 PUBLIC    } [,...]
```

IF

```
IF Condition THEN [Statement;[...]]
[ELSEIF Condition THEN [Statement; [...]]]
[ELSE [Statement; [...]]] ENDIF;
```

INCLUDE

```
INCLUDE {SQLCA [[IS]EXTERNAL]
         SQLDA               }
```

INSERT - 1

```
[BULK]INSERT INTO { [Owner.]TableName
                    [Owner.]ViewName}
[({ColumnName}[,...])]
VALUES ({SingleRowValues
         BulkValues
         ?              })
```

SingleRowValues

```
{NULL
 USER
 :HostVariable [ [INDICATOR]:IndicatorVariable]
 ?
 :LocalVariable
 :ProcedureParameter
 ::Built-inVariable
 ConversionFunction
 CurrentFunction
 [+
  -]{Integer
     Float
     Decimal }
 'CharacterString'
 0xHexadeciamalString
 'LongColumnIOString'  }[,...]
```

LongColumnIOString

```
<{[PathName/]FileName
  %SharedMemoryAddress}
[{>
  >>
  >!}[PathName/]{FileName
                 CharSting$
                 CharString$ CharString}
  >%{SharedMemoryAddress
     $                  }               ]
```

BulkValues

:*Buffer* [,:*StartIndex* [, :*NumberOfRows*]]

Dynamic Parameter Substitution

(? [,...])

INSERT - 2

```
INSERT INTO {[Owner.]TableName
             [Owner.]ViewName}[(ColumnName [,...])] QueryExpression
```

Labeled Statement

Label: *Statement*

LOCK TABLE

```
LOCK TABLE [Owner.]TableName IN {SHARE [UPDATE]
                                 EXCLUSIVE      }MODE
```

LOG COMMENT

```
LOG COMMENT {'String'
             :HostVariable
             :ProcedureParameter
             :ProcedureLocalVariable
             ?                      }
```

OPEN

```
OPEN CursorName [KEEP CURSOR [WITH LOCKS
                              WITH NOLOCKS]]
[USING { [SQL]DESCRIPTOR {SQLDA
                          AreaName}
          HostVariableName[[INDICATOR]:IndicatorVariable][,...]} ]
```

PREPARE

```
PREPARE [REPEAT]{StatementName
                [Owner.]ModuleName [(SectionNumber)]}
[IN DBEFileSetName]FROM {'String'
                         :HostVariable}
```

PRINT

```
PRINT {'Constant'
       :LocalVariable
       :Parameter
       ::Built-inVariable};
```

RAISE ERROR

```
RAISE ERROR [ErrorNumber] [MESSAGE ErrorText]
```

REFETCH

```
REFETCH CursorName INTO {:HostVariable [[ INDICATOR] :Indicator]}[,...]
```

RELEASE

```
RELEASE
```

REMOVE DBEFILE

```
REMOVE DBEFILE DBEFileName FROM DBEFILESET DBEFileSetName
```

REMOVE FROM GROUP

```
REMOVE {DBEuserID
        GroupName
        ClassName}[,...]FROM GROUP [Owner.]TargetGroupName
```

RENAME COLUMN

```
RENAME COLUMN [Owner.]TableName.ColumnName TO NewColumnName
```

RENAME TABLE

```
RENAME TABLE [Owner.]TableName TO NewTableName
```

RESET

```
RESET {SYSTEM.ACCOUNT [FOR USER {*
                                 DBEUserID}]
       SYSTEM.COUNTER                       }
```

RETURN

```
RETURN [ReturnStatus];
```

REVOKE

Revoke Table or View Authority

```
REVOKE {ALL [PRIVILEGES]
        [SELECT
         INSERT
         DELETE
         ALTER
         INDEX
         UPDATE [({ColumnName}[,...])]
         REFERENCES [({ColumnName}[,...])]]|,...|}
ON {[Owner.]TableName
    [Owner.}ViewName  } FROM {DBEUserID
                              GroupName
                              ClassName
                              PUBLIC    }[,...][CASCADE]
```

Revoke RUN or EXECUTE Authority

```
REVOKE [RUN ON [Owner.]ModuleName
        EXECUTE ON PROCEDURE [Owner.] ProcedureName] FROM
{{DBEUserID
  GroupName
  ClassName}[,...]
 PUBLIC            }
```

Revoke CONNECT, DBA, INSTALL, MONITOR, or RESOURCE Authority

```
REVOKE {CONNECT
        DBA
        INSTALL [AS OwnerID]
        MONITOR
        RESOURCE             }FROM {DBEUserID
                                    GroupName
                                    ClassName}[,...]
```

SQL Syntax—Revoke DBEFileSet Authority

```
REVOKE {SECTIONSPACE
        TABLESPACE} |,...| ON DBEFILESET DBEFileSetName FROM
{ {DBEUserID
   GroupName
   ClassName}[,...]
  PUBLIC            }
```

ROLLBACK WORK

```
ROLLBACK WORK [TO {SavePointNumber
                   :HostVariable
                   :LocalVariable
                   :ProcedureParameter}
               RELEASE                 ]
```

SAVEPOINT

```
SAVEPOINT [:HostVariable
           :LocalVariable
           :ProcedureParameter]
```

SELECT

Select Statement Level

```
[BULK]QueryExpression [ORDER BY {ColumnID [ASC
                                           DESC]}[,...]]
```

Subquery Level

```
(QueryExpression)
```

Query Expression Level

```
{QueryBlock
 (QueryExpression)}[UNION [ALL]{QueryBlock
                                (QueryExpreession)}][...]
```

Query Block Level

```
SELECT [ALL
        DISTINCT] SelectList [INTO HostVariableSpecification]
FROM FromSpec [,...]
 [WHERE SearchCondition1]
 [GROUP BY GroupColumnList]
 [HAVING SearchCondition2]
```

SelectList

```
{*
 [Owner.]Table.*
 CorrelationName.*
 Expression
 [[Owner.]Table.]ColumnName
 CorrelationName.ColumnName}[,...]
```

HostVariableSpecification--With BULK Option

```
:Buffer [ ,:StartIndex [,:NumberOfRows] ]
```

HostVariableSpecification--Without BULK Option

```
{ :HostVariable [ [INDICATOR] :Indicator] ) [,...]
```

FromSpec

```
{TableSpec
 (FromSpec)
 FromSpec NATURAL [INNER
                   LEFT [OUTER]
                   RIGHT [OUTER]] JOIN {TableSpec
                                        (FromSpec) }
 FromSpec [INNER
           LEFT [OUTER]
           RIGHT [OUTER]] JOIN {TableSpec
                               (FromSpec)}{ON SearchCondition3
                                           USING (ColumnList) } }
```

TableSpec

```
[Owner.] TableName [CorrelationName]
```

SET CONNECTION

```
SET CONNECTION {'ConnectionName'
                :HostVariable   }
```

SET CONSTRAINTS

```
SET ConstraintType [,...]CONSTRAINTS {DEFERRED
                                      IMMEDIATE}
```

SET DEFAULT DBEFILESET

```
SET DEFAULT {SECTIONSPACE
             TABLESPACE   } TO DBEFILESET DBEFileSetName FOR PUBLIC
```

SET DML ATOMICITY

```
SET DML ATOMICITY AT {ROW
                      STATEMENT} LEVEL
```

SET MULTITRANSACTION

```
SET MULTITRANSACTION {ON
                      OFF}
```

SETOPT

```
SETOPT {CLEAR
        GENERAL {ScanAccess
                 JoinAlgorithm}[,...]
        BEGIN {GENERAL {ScanAccess
                        JoinAlgorithm} } [,...]END }
```

Scan Access

```
[NO]{SERIALSCAN
     INDEXSCAN
     HASHSCAN
     SORTINDEX  }
```

Join Algorithm

```
[NO] {NESTEDLOOP
      NLJ
      SORTMERGE
      SMJ        }
```

SET PRINTRULES

```
SET PRINTRULES [ON
                OFF]
```

SET SESSION

```
SET SESSION {ISOLATION LEVEL {RR
                              CS
                              RC
                              RU
                              REPEATABLE READ
                              SERIALIZABLE
                              CURSOR STABILITY
                              READ COMMITTED
                              READ UNCOMMITTED
                              :HostVariable1    }
             PRIORITY {Priority
                       :HostVariable2}
             LABEL {'LabelString'
                    :HostVariable3}
             ConstraintType [,...]CONSTRAINTS {DEFERRED
                                               IMMEDIATE}
             DML ATOMICITY AT {STATEMENT
                               ROW      } LEVEL
             ON {TIMEOUT
                 DEADLOCK}ROLLBACK {QUERY
                                    TRANSACTION}
             USER TIMEOUT [TO] {DEFAULT
                                MAXIMUM
                                TimeoutValue[{SECONDS
                                              MINUTES}]
                                :HostVariable4[{SECONDS
                                                MINUTES} ] }
             TERMINATION AT {SESSION
                             TRANSACTION
                             QUERY
                             RESTRICTED   }LEVEL
             [{PARALLEL
               NO       }]FILL                                  }[,...]
```

SET TRANSACTION

```
SET TRANSACTION {ISOLATION LEVEL {RR
                                  CS
                                  RC
                                  RU
                                  REPEATABLE READ
                                  SERIALIZABLE
                                  CURSOR STABILITY
                                  READ COMMITTED
                                  READ UNCOMMITTED
                                  :HostVariable1   }
                PRIORITY {Priority
                          :HostVariable2}
                LABEL {'LabelString'
                       :HostVariable3}
                ConstraintType [,...]CONSTRAINTS {DEFERRED
                                                  IMMEDIATE}
                DML ATOMICITY AT {STATEMENT
                                  ROW       }LEVEL
                ON {TIMEOUT
                    DEADLOCK}ROLLBACK {QUERY
                                       TRANSACTION}
                USER TIMEOUT [TO] {DEFAULT
                                   MAXIMUM
                                   TimeoutValue[{SECONDS
                                                 MINUTES}]
                                   :HostVariable4[{SECONDS
                                                   MINUTES}]
                TERMINATION AT {SESSION
                                TRANSACTION
                                QUERY
                                RESTRICTED  }LEVEL              }[,...]
```

SET USER TIMEOUT

```
SET USER TIMEOUT [TO] {{TimeoutValue
                        :HostVariable} [SECONDS
                                        MINUTES]
                      DEFAULT
                      MAXIMUM                   }
```

SQLEXPLAIN

```
SQLEXPLAIN :HostVariable
```

START DBE

```
START DBE 'DBEnvironmentName' [AS 'ConnectionName'][MULTI]
[BUFFER = (DataBufferPages, LogBufferPages)
 TRANSACTION = MaxTransactions
 MAXIMUM TIMEOUT = {TimeoutValue [SECONDS
                                  MINUTES]
                   NONE                  }
 DEFAULT TIMEOUT = {TimeoutValue [SECONDS
                                  MINUTES]
                   MAXIMUM               }
 RUN BLOCK = ControlBlockPages              ]|,...|
```

START DBE NEW

```
START DBE 'DBEnvironmentName' [AS 'ConnectionName'][MULTI]NEW
[{DUAL
  AUDIT}|...|LOG
 BUFFER = (DataBufferPages, LogBufferPages)
 LANG = LanguageName
 TRANSACTION = MaxTransactions
 MAXIMUM TIMEOUT = {TimeoutValue [SECONDS
                                  MINUTES]
                   NONE                   }
 DEFAULT TIMEOUT = {TimeoutValue [SECONDS
                                  MINUTES]
                   MAXIMUM                }
 RUN BLOCK = ControlBlockPages
 DEFAULT PARTITION = {DefaultPartitionNumber
                      NONE                    }
 COMMENT PARTITION = {CommentPartitionNumber
                      DEFAULT
                     NONE                     }
 MAXPARTITIONS = MaximumNumberOfPartitions
 AUDIT NAME = 'AuditName'
 {COMMENT
  DATA
  DEFINITION
  STORAGE
  AUTHORIZATION
  SECTION
  ALL             }|...|AUDIT ELEMENTS
 DBEFile0Definition
 DBELogDefinition                                ]|,...|
```

DBEFile0Definition

```
DBEFILE0 DBEFILE DBEDile0ID
WITH PAGES = DBEFile0Size
NAME = 'SystemFileName1'
```

DBELogDefinition

```
LOG DBEFILE DBELog1ID [AND DBELog2ID]
WITH PAGES = DBELogSize,
NAME = 'SystemFileName2' [AND 'SystemFileName3']
```

START DBE NEWLOG

```
START DBE 'DBEnvironmentName' [AS 'ConnectionName'][MULTI]NEWLOG
[{ARCHIVE
  DUAL
  AUDIT}|...|LOG
 BUFFER = (DataBufferPages, LogBufferPages)
 TRANSACTION = MaxTransactions
 MAXIMUM TIMEOUT = {TimeoutValue [SECONDS
                                  MINUTES]
                   NONE                  }
 DEFAULT TIMEOUT = {TimeoutValue [SECONDS
                                  MINUTES]
                   MAXIMUM               }
 RUN BLOCK = ControlBlockPages
 DEFAULT PARTITION = {DefaultPartitionNumber
                      NONE                   }
 COMMENT PARTITION = {CommentPartitionNumber
                      DEFAULT
                      NONE                   }
 MAXPARTITIONS = MaximumNumberOfPartitions
 AUDIT NAME = 'AuditName'
 {COMMENT
  DATA
  DEFINITION
  STORAGE
  AUTHORIZATION
  SECTION
  ALL           }|...|AUDIT ELEMENTS            ]|,...| NewLogDefinition
```

NewLogDefinition

```
LOG DBEFILE DBELog1ID [AND DBELog2ID]
WITH PAGES = DBELogSize,
NAME = 'SystemFileName1' [AND 'SystemFileName2']
```

STOP DBE

```
STOP DBE
```

TERMINATE QUERY

```
TERMINATE QUERY FOR {CID ConnectionID
                     XID TransactionID}
```

TERMINATE TRANSACTION

```
TERMINATE TRANSACTION FOR {CID ConnectionID
                           XID TransactionID}
```

TERMINATE USER

```
TERMINATE USER {DBEUserID
                SessionID
                CID ConnectionID}
```

TRANSFER OWNERSHIP

```
TRANSFER OWNERSHIP OF {[TABLE][Owner.]TableName
                       [VIEW][Owner.]ViewName
                       PROCEDURE [Owner.]ProcedureName
                       GROUP GroupName                  } TO NewOwnerName
```

TRUNCATE TABLE

```
TRUNCATE TABLE [Owner.]TableName
```

UPDATE

```
UPDATE {[Owner.]TableName
        [Owner.]ViewName  }
SET { ColumnName = { Expression
                     'LongColumnIOString'
                      NULL                  } } [,...]
[WHERE SearchCondition ]
```

LongColumnIOString

```
{ [<{[PathName/]FileName
     %SharedMemoryAddress}]
  [{>
    >>
    >!}[PathName/]{FileName
                   CharString$
                   CharString$ CharString}
   >% {SharedMemoryAddress
       $                                  } ] } |...|
```

UPDATE STATISTICS

```
UPDATE STATISTICS FOR TABLE {[Owner.]TableName
                             SYSTEM.SystemViewName}
```

UPDATE WHERE CURRENT

```
UPDATE {[Owner.]TableName
        [Owner.]ViewName}
SET { ColumnName = {Expression
                     'LongColumnIOString'
                     NULL                  }}[,...]
WHERE CURRENT OF CursorName
```

LongColumnIOString

```
{ [<{[PathName/]FileName
     %SharedMemoryAddress}]
  [{>
    >>
    >!}[PathName/]{FileName
                   CharString$
                   CharString$ CharString}
   >% {SharedMemoryAddress
       $                                  } ] } |...|
```

VALIDATE

```
VALIDATE [FORCE
          DROP SETOPTINFO]
{MODULE { {[Owner.]ModuleName} [,...]
          {SECTION [Owner.]ModuleName (SectionNumber)} [,...] }
 PROCEDURE { {[Owner.]ProcedureName} [,...]
             {SECTION [Owner.]ProcedureName (SectionNumber)} [,...] }
 ALL{MODULES
     PROCEDURES} [WITH AUTOCOMMIT]                                    }
```

WHENEVER

```
WHENEVER {SQLERROR
          SQLWARNING
          NOT FOUND   } {STOP
                         CONTINUE
                         GOTO [:]Label
                         GO TO [:]Label}
```

WHILE

```
WHILE Condition DO [Statement; [...]] ENDWHILE;
```

B ISQL Syntax Summary

ISQL is the interactive interface to ALLBASE/SQL. Some, but not all, ALLBASE/SQL statements can be entered interactively as ISQL commands.

CHANGE

```
C[HANGE]Delimiter OldString Delimiter NewString Delimiter [@]
```

DO

```
DO [CommandNumber
    CommandString]
```

EDIT

```
ED[IT][FileName]
```

END

```
EN[D]
```

ERASE

```
ER[ASE]FileName
```

EXIT

```
EX[IT]
```

EXTRACT

```
EXTRACT{ MODULE[Owner.]ModuleName[,...]
         SECTION [Owner.]ModuleName(SectionNumber) [,...]
         ALL MODULES                                        }
[NO SETOPTINFO]INTO FileName
```

HELP

```
HE[LP]{@
       SQLStatement
       ISQLCommand} [D[ESCRIPTION]
                     S[YNTAX]
                     E[XAMPLE]     ]
```

HOLD

```
HO[LD]{SQLStatement
       ISQLCommand}[EscapeCharacter;{SQLStatement
                                     ISQLCommand}][...]
```

INFO

```
IN[FO]{[Owner.]TableName
       [Owner.]ViewName}
```

INPUT

```
INP[UT]{[Owner.]TableName
        [Owner.]ViewName} (ColumnName[,ColumnName][...])
{ (Value[,Value][...])[ROLLBACK WORK
                       COMMIT WORK] } [...] E[ND]
```

INSTALL

```
IN[STALL]FileName[DROP][IN DBEFileSetName][NO OPTINFO]
```

LIST FILE

```
LI[ST]F[ILE]FileName
```

LIST HISTORY

```
LI[ST]H[ISTORY]{CommandNumber
                @             }
```

LIST INSTALL

```
LI[ST[I[NSTALL]FileName
```

LIST SET

```
LI[ST]S[ET]{Option
            @      }
```

LOAD

```
LO[AD][ P[ARTIAL]]FROM { E[XTERNAL]
                         I[NTERNAL]} InputFileName[AT StartingRow]
[FOR NumberOfRows]TO { [Owner.]TableName
                       [Owner.]ViewName}[ExternalInputSpec
                                         USING DescriptionFileName]
 {Y[ES] PatternLocation Pattern
  N[O]                          }
```

ExternalInputSpec

```
{ ColumnName StartingLocation Length [NullIndicator]
  [FormatType]                                        } [...]E[ND]
```

RECALL

```
REC[ALL]{C[URRENT]
        F[ILE]FileName
        H[ISTORY] CommandNumber}
```

REDO

```
RED[O][ CommandNumber
        CommandString]
```

Subcommands

B	Break
D	Delete
E	Exit
H	Help
I	Insert
L	List
R	Replace
X	Execute
+[*n*]	Forward *n*
-[*n*]	Backward *n*
Return	Next Line

RENAME

```
REN[AME]OldFileName NewFileName
```

SELECTSTATEMENT

```
SelectStatement;[PA[USE];][BrowseOption;][...]E[ND]
```

SET

```
SE[T]Option OptionValue
```

Options and Values

```
AUTOC[OMMIT] ON | OFF
AUTOL[OCK] ON | OFF
AUTOS[AVE] NumberofRows
C[ONTINUE] ON | OFF
CONV[ERT] ASCII | EBCDIC | OFF
EC[HO] ON | OFF
ECHO_[ALL] ON | OFF
EDITOR EditorName
```

```
ES[CAPE] Character
EXIT[_ON_DBERR] ON | OFF
EXIT_ON_DBWARN ON | OFF
FL[AGGER] FlaggerName
F[RACTION] Length
N[ULL] [Character]
OU[TPUT] FileName
OW[NER] OwnerName
LOAD_B[UFFER] BufferSize
PA[GEWIDTH] PageWidth
PR[OMPT] PromptString
```

SQLGEN

```
SQLG[EN]
```

SQLUTIL

```
SQLU[TIL]
```

START

```
STA[RT][CommandFileName][ (Value[,Value][...] ) ]
```

STORE

```
STO[RE] FileName [R[EPLACE] ]
```

SYSTEM

```
{S[YSTEM]
 !        }[HP-UXCommand]
```

UNLOAD

```
U[NLOAD] TO { E[XTERNAL]
              I[NTERNAL]}OutputFileName
FROM {[Owner.]TableName
      [Owner.]ViewName
      "SelectCommand"}ExternalOutputSpec
```

ExternalOutputSpec

```
DescriptionFileName{ OutputLength [FractionLength]
                     [NullIndicator]              }[...]
```

C Sample DBEnvironment

The DBEnvironment used in examples throughout the ALLBASE/SQL manual set is called *PartsDBE*. Your installation package includes the necessary files to create a working version of this DBEnvironment so that users can try the examples while learning about the features of ALLBASE/SQL. Also included is a set of sample applications that access PurchDB, the main database in PartsDBE.

This appendix presents the steps for setting up the sample DBEnvironment and then displays some important files and tables related to PartsDBE. It contains the following sections:

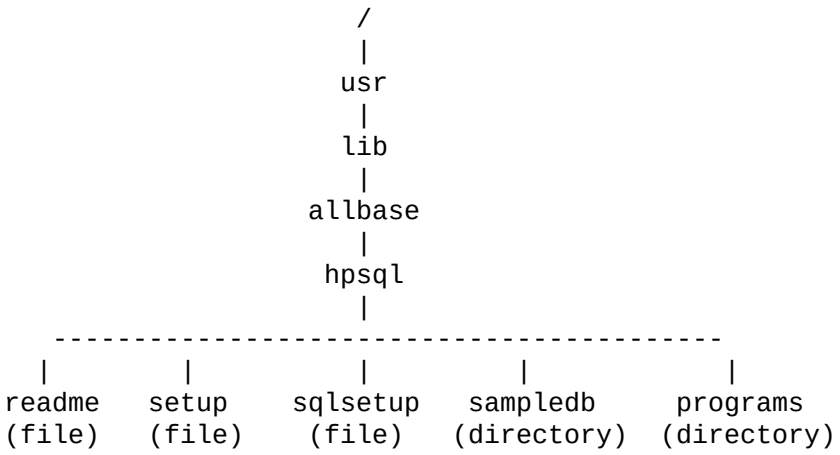
- Installing the Files for PartsDBE
- Setting Up PartsDBE
- ISQL command files for creating and loading PartsDBE
- Tables in the ManufDB, PurchDB, and RecDB Databases
- Sample Program Files

Before you can use PartsDBE, you must install the sample database files, then run a setup script to create and load PartsDBE. Optionally, you can preprocess and compile the sample application programs.

Installing the Files for PartsDBE

You can install the files for setting up PartsDBE when you install ALLBASE/SQL or at a later time. Refer to the pamphlet entitled *ALLBASE/SQL Release Notes* for installation instructions. This pamphlet accompanies the media for the software.

The files for PartsDBE are installed into the following directory structure:



The hpsql directory contains the files that are important for loading the sample DBEnvironment. Included are the following three files:

- readme, a file describing the files in the hpsql directory and how to use them
- sqlsetup, a C shell script that displays a menu of options for creating sample DBEnvironments (sqlsetup calls setup)
- setup, a script you use to copy the hpsql directory and subdirectories into your current working directory, create and load PartsDBE, and preprocess and compile the sample source code files

Also included are the two directories listed here:

- sampledb, which contains files for creating and loading PartsDBE
- programs, which contains source code files for sample ALLBASE/SQL C, COBOL, Pascal, and FORTRAN programs

Setting Up PartsDBE

Before beginning, change into the directory where you want to create PartsDBE. Use an empty directory if possible. Then choose one of the following two methods for setting up PartsDBE:

- Using SQLSetup
- Using Setup

SQLSetup is a sample database setup tool which simplifies the process of installing PartsDBE in your work space. Setup is a lower-level script called by SQLSetup.

Using SQLSetup

Run SQLSetup by issuing the proper command. From the C shell, issue the following command:

```
$ /usr/lib/allbase/hpsql/sqlsetup Return
```

From the K shell or Bourne shell, issue the following command:

```
$ csh /usr/lib/allbase/hpsql/sqlsetup Return
```

A menu like the one in <Undefined Cross-Reference> appears on your screen.

Figure C-1. SQLSetup Menu

```
        Options for Setting Up ALLBASE/SQL Sample DBEnvironments

    ================================================================

    Choose one:

    1. Create PartsDBE without sample programs
    2. Create PartsDBE, copy, preprocess and compile sample programs
    3. Copy, preprocess and compile sample programs only
    4. Generate a schema for PartsDBE
    5. Display schema for PartsDBE
    6. Purge PartsDBE and sample programs
    7. Help
    0. Exit

    ================================================================

    Enter your choice=>
```

From this menu, you select an option to create a copy of PartsDBE in your directory. Before choosing an option, examine each line on the menu. The first option simply creates a copy of PartsDBE. The second option, in addition to creating PartsDBE, copies a set of application programs into the current directory, then preprocesses and compiles them. (This is time-consuming.)

Option 3 creates just the sample program set. Option 4 creates a database schema by calling SQLGEN. Option 5 displays the schema once it has been created. Option 6 lets you purge the sample DBEnvironment and programs.

Choose the Help option to see more information about SQLSetup, or choose 0 to exit.

Creating PartsDBE

To create PartsDBE, choose option 1 from the SQLSetup menu. This option runs a set of ISQL command files that create the DBEnvironment, define all its tables, views, indexes and security structure, and then load it with data.

As the system creates PartsDBE, you see several messages displayed. At the end of the creation process, you see the following message:

```
Creation and Loading of PartsDBE is now complete!
```

When you return to the menu, choose 0 to exit.

Using Setup

The following is an alternate method for setting up PartsDBE.

Use the following command:

```
$ /usr/lib/allbase/hpsql/setup 2 Return
```

You will see a display of messages showing the progress of the setup script. For more information about setup, read the comments at the beginning of the file itself.

Listings of ISQL Command Files

Both SQLSetup and setup use a group of ISQL command files to create and load local copies of PartsDBE. These files, located in /usr/lib/allbase/hpsql/sampledb, are as follows:

- STARTDBE, an ISQL command file containing the `START DBE` command.
- CREATABS, an ISQL command file containing SQL commands. for creating DBEFileSets, DBEFiles, tables, and views
- LOADTABS, an ISQL command file containing ISQL and SQL commands for loading the two tables in the ManufDB database, the six tables in the PurchDB database, and the three tables in the RecDB database in the PartsDBE DBEnvironment. LOADTABS uses the following ASCII files, which contain sample data:
  - *`SupplyBa`* contains data for the ManufDB.SupplyBatches table.
  - *`TestData`* contains data for the ManufDB.TestData table.
  - *`Inventor`* contains data for the PurchDB.Inventory table.
  - *`OrderIte`* contains data for the PurchDB.OrderItems table.
  - *`Orders`* contains data for the PurchDB.Orders table.
  - *`Parts`* contains data for the PurchDB.Parts table.
  - *`Report1`* contains data for the PurchDB.Reports table.
  - *`SupplyPr`* contains data for the PurchDB.SupplyPrice table.
  - *`Vendors`* contains data for the PurchDB.Vendors table.
  - *`Members`* contains data for the RecDB.Members table.
  - *`Clubs`* contains data for the RecDB.Clubs table.
  - *`Events`* contains data for the RecDB.Events table.
- CREAINDX, an ISQL command file containing `CREATE INDEX` commands.
- CREASEC, an ISQL command file containing SQL commands for granting various authorities.

Listings of these files appear in the following sections.

STARTDBE Command File

```
   /*This file creates the PartsDBE DBEnvironment with MULTI user mode and dual
logging. */

   START DBE 'sampledb/PartsDBE' MULTI NEW
         DUAL LOG,
         TRANSACTION = 5,
         DBEFILE0 DBEFILE PartsDBE0
             WITH PAGES = 150,
             NAME = 'PartsF0',
         LOG DBEFILE PartsDBELog1 AND PartsDBELog2
             WITH PAGES = 256,
             NAME = 'PartsLG1' AND 'PartsLG2';
```

CREATABS Command File

```
   /* The following commands create the Purchasing Department's DBEFileSet
with two DBEFiles. */

   CREATE DBEFILESET PurchFS;

     CREATE DBEFILE PurchDataF1
          WITH PAGES = 50, NAME = 'PurchDF1',
          TYPE = TABLE;

     CREATE DBEFILE PurchIndxF1
          WITH PAGES = 50, NAME = 'PurchXF1',
          TYPE = INDEX;

     ADD DBEFILE PurchDataF1
          TO DBEFILESET PurchFS;

     ADD DBEFILE PurchIndxF1
          TO DBEFILESET PurchFS;

   /* The following commands create the Warehouse Department's DBEFileSet
with two DBEFiles. */

   CREATE DBEFILESET WarehFS;
     CREATE DBEFILE WarehDataF1
          WITH PAGES = 50, NAME = 'WarehDF1',
          TYPE = TABLE;
     CREATE DBEFILE WarehIndxF1
          WITH PAGES = 50, NAME = 'WarehXF1',
          TYPE = INDEX;
     ADD DBEFILE WarehDataF1
          TO DBEFILESET WarehFS;
     ADD DBEFILE WarehIndxF1
          TO DBEFILESET WarehFS;

   /* The following commands create the Receiving Department's DBEFileSet
with two DBEFiles. */

   CREATE DBEFILESET OrderFS;
     CREATE DBEFILE OrderDataF1
          WITH PAGES = 50, NAME = 'OrderDF1',
          TYPE = TABLE;
     CREATE DBEFILE OrderIndxF1
          WITH PAGES = 50, NAME = 'OrderXF1',
          TYPE = INDEX;
     ADD DBEFILE OrderDataF1
          TO DBEFILESET OrderFS;
     ADD DBEFILE OrderIndxF1
          TO DBEFILESET OrderFS;
```

```
   /* The following commands create a DBEFileSet with one DBEFile for
      storage of long field data in the PurchDB.Reports table     .*/

     CREATE DBEFILESET FileFS;

     CREATE DBEFILE FileData
          WITH PAGES=50, NAME='FileData',
          TYPE=TABLE;

     ADD DBEFILE FileData TO DBEFILESET FileFS;
   /* The following commands create the two tables that comprise the ManufDB
database. */

   CREATE PUBLIC TABLE ManufDB.SupplyBatches
     (VendPartNumber        CHAR(16)   NOT NULL,
      BatchStamp            DATETIME   DEFAULT CURRENT_DATETIME
                            NOT NULL
                            PRIMARY KEY,
      MinPassRate           FLOAT)
      IN WarehFS;

   CREATE PUBLIC TABLE ManufDB.TestData
     (BatchStamp   DATETIME  NOT NULL
                   REFERENCES ManufDB.SupplyBatches (BatchStamp),
      TestDate     DATE,
      TestStart    TIME,
      TestEnd      TIME,
      LabTime      INTERVAL,
      PassQty      INTEGER,
      TestQty      INTEGER)
      IN WarehFS;

   /* The following commands create the seven tables and two views
      that comprise the PurchDB database. */

   CREATE PUBLIC TABLE PurchDB.Parts
     (PartNumber        CHAR(16)          NOT NULL,
      PartName          CHAR(30),
      SalesPrice        DECIMAL(10,2) )
     IN WarehFS;

   CREATE PUBLIC TABLE PurchDB.Inventory
     (PartNumber        CHAR(16)          NOT NULL,
      BinNumber         SMALLINT          NOT NULL,
      QtyOnHand         SMALLINT,
      LastCountDate     CHAR(8),
      CountCycle        SMALLINT,
      AdjustmentQty     SMALLINT,
      ReorderQty        SMALLINT,
      ReorderPoint      SMALLINT )
     IN WarehFS;

   CREATE PUBLIC TABLE PurchDB.SupplyPrice
     (PartNumber        CHAR(16)          NOT NULL,
      VendorNumber      INTEGER           NOT NULL,
      VendPartNumber    CHAR(16)          NOT NULL,
      UnitPrice         DECIMAL(10,2),
      DeliveryDays      SMALLINT,
```

```
   DiscountQty       SMALLINT)
  IN PurchFS;

CREATE PUBLIC TABLE PurchDB.Vendors
  (VendorNumber      INTEGER           NOT NULL,
   VendorName        CHAR(30)          NOT NULL,
   ContactName       CHAR(30),
   PhoneNumber       CHAR(15),
   VendorStreet      CHAR(30)          NOT NULL,
   VendorCity        CHAR(20)          NOT NULL,
   VendorState       CHAR(2)           NOT NULL,
   VendorZipCode     CHAR(10)          NOT NULL,
   VendorRemarks     VARCHAR(60) )
  IN PurchFS;

CREATE PUBLIC TABLE PurchDB.Orders
  (OrderNumber       INTEGER           NOT NULL,
   VendorNumber      INTEGER,
   OrderDate         CHAR(8) )
  IN OrderFS;
CREATE PUBLIC TABLE PurchDB.OrderItems
  (OrderNumber       INTEGER           NOT NULL,
   ItemNumber        INTEGER           NOT NULL,
   VendPartNumber    CHAR(16),
   PurchasePrice     DECIMAL(10,2)     NOT NULL,
   OrderQty          SMALLINT,
   ItemDueDate       CHAR(8),
   ReceivedQty       SMALLINT )
  IN OrderFS;

CREATE PUBLIC TABLE PurchDB.Reports
  (ReportName        CHAR(20)          NOT NULL,
   ReportOwner       CHAR(20)          NOT NULL,
   FileData LONG VARBINARY(100000)IN FileFS NOT NULL)
  IN OrderFS;

CREATE VIEW PurchDB.PartInfo
     (PartNumber,
      PartName,
      VendorNumber,
      VendorName,
      VendorPartNumber,
      ListPrice,
      Quantity)  AS
   SELECT PurchDB.SupplyPrice.PartNumber,
          PurchDB.Parts.PartName,
          PurchDB.SupplyPrice.VendorNumber,
          PurchDB.Vendors.VendorName,
          PurchDB.Supplyprice.VendPartNumber,
          PurchDB.SupplyPrice.UnitPrice,
          PurchDB.SupplyPrice.DiscountQty
     FROM PurchDB.Parts,
          PurchDB.SupplyPrice,
          PurchDB.Vendors
    WHERE PurchDB.SupplyPrice.PartNumber =
          PurchDB.Parts.PartNumber
      AND PurchDB.SupplyPrice.VendorNumber =
          PurchDB.Vendors.VendorNumber;
```

```
CREATE VIEW PurchDB.VendorStatistics
     (VendorNumber,
      VendorName,
      OrderDate,
      OrderQuantity,
      TotalPrice)  AS
   SELECT PurchDB.Vendors.VendorNumber,
          PurchDB.Vendors.VendorName,
          OrderDate,
          OrderQty,
          OrderQty * PurchasePrice
     FROM PurchDB.Vendors,
          PurchDB.Orders,
          PurchDB.OrderItems
    WHERE PurchDB.Vendors.VendorNumber =
          PurchDB.Orders.VendorNumber
      AND PurchDB.Orders.OrderNumber =
          PurchDB.OrderItems.OrderNumber;

/* The following commands create the Recreation DBEFileSet
     with one DBEFile. */

CREATE DBEFILESET RecFS;

CREATE DBEFILE RecDataF1
     WITH PAGES = 50, NAME = 'RecDF1',
     TYPE = MIXED;

ADD DBEFILE RecDataF1
     TO DBEFILESET RecFS;

/* The following commands create three tables
     that comprise the RecDB database. */

CREATE PUBLIC TABLE RecDB.Clubs
  (ClubName CHAR(15) NOT NULL PRIMARY KEY CONSTRAINT Clubs_PK,
   ClubPhone         SMALLINT,
   Activity          CHAR(18) )
 IN RecFS;

CREATE PUBLIC TABLE RecDB.Members
  (MemberName        CHAR(20) NOT NULL,
   Club              CHAR(15) NOT NULL,
   MemberPhone       SMALLINT,
   PRIMARY KEY (MemberName, Club) CONSTRAINT Members_PK,
   FOREIGN KEY (Club)
   REFERENCES RecDB.Clubs (ClubName) CONSTRAINT Members_FK)
 IN RecFS;

CREATE PUBLIC TABLE RecDB.Events
  (SponsorClub       CHAR(15),
   Event             CHAR(30),
   Date              DATE DEFAULT CURRENT_DATE,
   Time              TIME,
   Coordinator       CHAR(20),
   FOREIGN KEY (Coordinator, SponsorClub)
   REFERENCES RecDB.Members (MemberName, Club) CONSTRAINT Events_FK)
 IN RecFS;
```

LOADTABS Command File

```
/* This file loads each of the two tables in the ManufDB    */
/*   database, the six tables in the PurchDB database,      */
/*   and the three tables in the RecDB database with data.  */

LOAD FROM EXTERNAL sampledb/SupplyBa TO ManufDB.SupplyBatches
VENDPARTNUMBER      1    16
BATCHSTAMP         18    23
MINPASSRATE        43     8   ?
END
N;
COMMIT WORK;
!echo Table SupplyBatches successfully loaded!;

LOAD FROM EXTERNAL sampledb/TestData TO ManufDB.TestData
BATCHSTAMP          1    23
TESTDATE           25    10   ?
TESTSTART          36     8   ?
TESTEND            45     8   ?
LABTIME            54    20   ?
PASSQTY            75     2   ?
TESTQTY            78     2   ?
ENDATA
N;
COMMIT WORK;
!echo Table TestData successfully loaded!;

LOAD FROM EXTERNAL sampledb/Parts TO PurchDB.Parts
PartNumber          1    16
PartName           17    30   ?
SalesPrice         47    12   ?
END
N;
COMMIT WORK;
!echo Table Parts successfully loaded!;

LOAD FROM EXTERNAL sampledb/Inventor TO PurchDB.Inventory
PartNumber          1    16
BinNumber          17     5
QtyOnHand          22     5   ?
LastCountDate      27     8   ?
CountCycle         35     5   ?
AdjustmentQty      40     5   ?
ReorderQty         45     5   ?
ReorderPoint       50     5   ?
END
N;
COMMIT WORK;
!echo Table Inventory successfully loaded!;

LOAD FROM EXTERNAL sampledb/SupplyPr TO PurchDB.SupplyPrice
PartNumber          1    16
VendorNumber       17    10
VendPartNumber     27    16
UnitPrice          43    12   ?
```

```
DeliveryDays       55     5   ?
DiscountQty        60     5   ?
END
N;
COMMIT WORK;
!echo Table SupplyPrice successfully loaded!;

LOAD FROM EXTERNAL sampledb/Orders TO PurchDB.Orders
OrderNumber         1    10
VendorNumber       11    10   ?
OrderDate          21     8   ?
END
N;
COMMIT WORK;
!echo Table Orders successfully loaded!;

LOAD FROM EXTERNAL sampledb/OrderIte TO PurchDB.OrderItems
OrderNumber         1    10
ItemNumber         11    10
VendPartNumber     21    16   ?
PurchasePrice      37    12
OrderQty           49     5   ?
ItemDueDate        54     8   ?
ReceivedQty        62     5   ?
END
N;
COMMIT WORK;
!echo Table OrderItems successfully loaded!;

LOAD FROM EXTERNAL sampledb/Vendors TO PurchDB.Vendors
VendorNumber        3     4
VendorName          7    30
ContactName        39    30   ?
PhoneNumber        71    12   ?
VendorStreet       88    30
VendorCity        120    20
VendorState       142     2
VendorZipCode     146     5
VendorRemarks     152    60   ?
END
N;
COMMIT WORK;
!echo Table Vendors successfully loaded!;

LOAD FROM EXTERNAL sampledb/Clubs TO RecDB.Clubs
ClubName            1    15
ClubPhone          25     4   ?
Activity           35    18   ?
END
N;
COMMIT WORK;
!echo Table Clubs successfully loaded!;

LOAD FROM EXTERNAL sampledb/Members TO RecDB.Members
MemberName          1    20
Club               25    15
MemberPhone        45     4   ?
END
```

```
N;
COMMIT WORK;
!echo Table Members successfully loaded!;

LOAD FROM EXTERNAL sampledb/Events TO RecDB.Events
SponsorClub         1    15   ?
Event              20    30   ?
Date               50    10   ?
Time               62     8   ?
Coordinator        71    20   ?
END
N;
COMMIT WORK;
!echo Table Events successfully loaded!;

INSERT INTO PURCHDB.REPORTS VALUES ('Report1', 'FREE',
'< sampledb/Report1>! Report1');
COMMIT WORK;
!echo Table Reports successfully loaded!;

!echo Loading of databases is now done!;
```

CREAINDEX Command File

```
/* This file creates the indexes for the PurchDB database  */
/* and then updates the statistics for each of the tables.    */

CREATE UNIQUE INDEX PartNumIndex
     ON PurchDB.Parts (PartNumber);
CREATE CLUSTERING INDEX PartToNumIndex
     ON PurchDB.SupplyPrice (PartNumber);
CREATE INDEX PartToVendIndex
     ON PurchDB.SupplyPrice (VendorNumber);
CREATE UNIQUE INDEX VendPartIndex
     ON PurchDB.SupplyPrice (VendPartNumber);
CREATE UNIQUE INDEX VendorNumIndex
     ON PurchDB.Vendors (VendorNumber);
CREATE UNIQUE CLUSTERING INDEX OrderNumIndex
     ON PurchDB.Orders (OrderNumber);
CREATE INDEX OrderVendIndex
     ON PurchDB.Orders (VendorNumber);
CREATE CLUSTERING INDEX OrderItemIndex
     ON PurchDB.OrderItems (OrderNumber);
CREATE UNIQUE INDEX InvPartNumIndex
     ON PurchDB.Inventory (PartNumber);

!echo Indexes have been created on tables in PurchDB!;

UPDATE STATISTICS FOR TABLE ManufDB.SupplyBatches;
UPDATE STATISTICS FOR TABLE ManufDB.TestData;
UPDATE STATISTICS FOR TABLE PurchDB.Parts;
UPDATE STATISTICS FOR TABLE PurchDB.Inventory;
UPDATE STATISTICS FOR TABLE PurchDB.SupplyPrice;
UPDATE STATISTICS FOR TABLE PurchDB.Vendors;
UPDATE STATISTICS FOR TABLE PurchDB.Orders;
UPDATE STATISTICS FOR TABLE PurchDB.OrderItems;
UPDATE STATISTICS FOR TABLE PurchDB.Reports;
UPDATE STATISTICS FOR TABLE RecDB.Members;
UPDATE STATISTICS FOR TABLE RecDB.Clubs;
UPDATE STATISTICS FOR TABLE RecDB.Events;

!echo Statistics have now been updated for all tables!;
```

CREASEC Command File

```
/* This file sets up authorities for the PurchDB and RecDB databases.*/
/* The DBA for the sampledb DBEnvironment is the DBEUserID John.     */

REVOKE ALL ON ManufDB.SupplyBatches FROM PUBLIC;
REVOKE ALL ON ManufDB.TestData FROM PUBLIC;
REVOKE ALL ON PurchDB.Parts FROM PUBLIC;
REVOKE ALL ON PurchDB.Inventory FROM PUBLIC;
REVOKE ALL ON PurchDB.SupplyPrice FROM PUBLIC;
REVOKE ALL ON PurchDB.Vendors FROM PUBLIC;
REVOKE ALL ON PurchDB.Orders FROM PUBLIC;
REVOKE ALL ON PurchDB.OrderItems FROM PUBLIC;
REVOKE ALL ON PurchDB.Reports FROM PUBLIC;
REVOKE ALL ON RecDB.Members FROM PUBLIC;
REVOKE ALL ON RecDB.Clubs FROM PUBLIC;
REVOKE ALL ON RecDB.Events FROM PUBLIC;

GRANT DBA TO John;


/* The following commands create the group for the Purchasing  */
/*   Department. This group has SELECT authority on all tables */
/*   and views of the PurchDB database. It also has INSERT     */
/*   and UPDATE authority for reports.                         */


CREATE GROUP PurchManagers;
ADD Margy TO GROUP PurchManagers;
ADD Ron TO GROUP PurchManagers;
ADD Sharon TO GROUP PurchManagers;

GRANT SELECT ON PurchDB.Parts TO PurchManagers;
GRANT SELECT ON PurchDB.Inventory TO PurchManagers;
GRANT SELECT ON PurchDB.SupplyPrice TO PurchManagers;
GRANT SELECT ON PurchDB.Vendors TO PurchManagers;
GRANT SELECT ON PurchDB.Orders TO PurchManagers;
GRANT SELECT ON PurchDB.OrderItems TO PurchManagers;
GRANT SELECT ON PurchDB.VendorStatistics TO PurchManagers;
GRANT SELECT ON PurchDB.PartInfo TO PurchManagers;
GRANT SELECT, INSERT, UPDATE ON PurchDB.Reports TO PurchManagers;

/* The following commands create the group that will maintain   */
/*   the database. This group has RESOURCE authority, and all   */
/*   table and view authorities for the tables and views of the */
/*   PurchDB database.                                          */

CREATE GROUP PurchDBMaint;

ADD Annie TO GROUP PurchDBMaint;
ADD Doug TO GROUP PurchDBMaint;
ADD David TO GROUP PurchDBMaint;

GRANT RESOURCE TO PurchDBMaint;
GRANT ALL ON PurchDB.Parts TO PurchDBMaint;
GRANT ALL ON PurchDB.Inventory TO PurchDBMaint;
```

```
GRANT ALL ON PurchDB.SupplyPrice TO PurchDBMaint;
GRANT ALL ON PurchDB.Vendors TO PurchDBMaint;
GRANT ALL ON PurchDB.Orders TO PurchDBMaint;
GRANT ALL ON PurchDB.Reports TO PurchDBMaint;
GRANT ALL ON PurchDB.OrderItems TO PurchDBMaint;
GRANT SELECT ON PurchDB.VendorStatistics TO PurchDBMaint;
GRANT SELECT ON PurchDB.PartInfo TO PurchDBMaint;


/* The following commands create the Purchasing Department's   */
/*   group. This group has SELECT, INSERT, DELETE, and UPDATE  */
/*   authority for the Inventory, SupplyPrice, Vendors, Orders,*/
/*   and OrderItems tables of the PurchDB database.            */

CREATE GROUP Purchasing;

ADD AJ TO GROUP Purchasing;
ADD Jorge TO GROUP Purchasing;
ADD Ragaa TO GROUP Purchasing;
ADD Greg TO GROUP Purchasing;
ADD Karen TO GROUP Purchasing;

GRANT SELECT,
      INSERT,
      DELETE,
      UPDATE
  ON PurchDB.Inventory
  TO Purchasing;

GRANT SELECT,
      INSERT,
      DELETE,
      UPDATE
  ON PurchDB.SupplyPrice
  TO Purchasing;

GRANT SELECT,
      INSERT,
      DELETE,
      UPDATE
  ON PurchDB.Vendors
  TO Purchasing;

GRANT SELECT,
      INSERT,
      DELETE,
      UPDATE
  ON PurchDB.Orders
  TO Purchasing;

GRANT SELECT,
      INSERT,
      DELETE,
      UPDATE
  ON PurchDB.OrderItems
  TO Purchasing;

/* The following commands create the Receiving Department's  */
```

```
/*   group. This group has SELECT, INSERT, DELETE, and UPDATE*/
/*   authority for the Orders and OrderItems tables of the   */
/*   PurchDB database.                                       */

CREATE GROUP Receiving;

ADD Al TO GROUP Receiving;
ADD Sue TO GROUP Receiving;
ADD Martha TO GROUP Receiving;

GRANT SELECT,
      INSERT,
      DELETE,
      UPDATE
  ON PurchDB.Orders
  TO Receiving;

GRANT SELECT,
      INSERT,
      DELETE,
      UPDATE
  ON PurchDB.OrderItems
  TO Receiving;

/* The following commands create the Warehouse Department's   */
/*   group. This group has SELECT, INSERT, DELETE, and UPDATE */
/*   authority for the Parts and Inventory tables of the      */
/*   PurchDB database.                                       */

CREATE GROUP Warehouse;
ADD Kelly TO GROUP Warehouse;
ADD Al TO GROUP Warehouse;
ADD Peter TO GROUP Warehouse;

GRANT SELECT,
      INSERT,
      DELETE,
      UPDATE
  ON PurchDB.Parts
  TO Warehouse;

GRANT SELECT,
      INSERT,
      DELETE,
      UPDATE
  ON PurchDB.Inventory
  TO Warehouse;

/* The following commands create the Accounts Payable Department's  */
/*   group. This group has SELECT, INSERT, DELETE, and UPDATE       */
/*   authority for the SupplyPrice, Vendors, Orders, and OrderItems */
/*   tables of the PurchDB database.                                */

CREATE GROUP AccountsPayable;

ADD Michele TO GROUP AccountsPayable;
ADD Jim TO GROUP AccountsPayable;
ADD Karen TO GROUP AccountsPayable;
```

```
ADD Stacey TO GROUP AccountsPayable;

GRANT SELECT,
      INSERT,
      DELETE,
      UPDATE
  ON PurchDB.SupplyPrice
  TO AccountsPayable;

GRANT SELECT,
      INSERT,
      DELETE,
      UPDATE
  ON PurchDB.Vendors
  TO AccountsPayable;

GRANT SELECT,
      INSERT,
      DELETE,
      UPDATE
  ON PurchDB.Orders
  TO AccountsPayable;

GRANT SELECT,
      INSERT,
      DELETE,
      UPDATE
  ON PurchDB.OrderItems
  TO AccountsPayable;


/* The following commands create the group called Purch.  All DBEUserIDs
or the groups to which they belong are made members of this group.
This group has CONNECT authority only to the PartsDBE DBEnvironment. */

CREATE GROUP Purch;

ADD PurchManagers TO GROUP Purch;
ADD PurchDBMaint TO GROUP Purch;
ADD Purchasing TO GROUP Purch;
ADD Receiving TO GROUP Purch;
ADD Warehouse TO GROUP Purch;
ADD AccountsPayable TO GROUP Purch;
ADD Tom@Wilkens TO GROUP Purch;

GRANT CONNECT TO Purch;

/* The following commands create the Manufacturing Department's  */
/*   group. This group has SELECT, INSERT, DELETE, and UPDATE    */
/*   authority for the TestData and SupplyBatches tables of the  */
/*   ManufDB database.                                           */

CREATE GROUP Manuf;
ADD Henry TO GROUP Manuf;
ADD Peter TO GROUP Manuf;

GRANT SELECT,
      INSERT,
```

```
      DELETE,
      UPDATE
  ON ManufDB.SupplyBatches
  TO Manuf;

GRANT SELECT,
      INSERT,
      DELETE,
      UPDATE
  ON ManufDB.TestData
  TO Manuf;

GRANT CONNECT TO Manuf;

/* The following commands GRANT specific authorities to  */
/*   specific DBEUserIDs.                                */

GRANT SELECT ON PurchDB.Vendors TO Tom;
GRANT SELECT ON PurchDB.VendorStatistics TO Tom;
GRANT SELECT ON PurchDB.PartInfo TO Tom;
GRANT UPDATE (BinNumber,QtyOnHand,LastCountDate)
          ON PurchDB.Inventory TO Kelly;
GRANT UPDATE (BinNumber,QtyOnHand,LastCountDate)
          ON PurchDB.Inventory TO Peter;
GRANT UPDATE (PhoneNumber,VendorStreet,VendorCity,
              VendorState,VendorZipCode)
          ON PurchDB.Vendors TO Karen;
GRANT UPDATE (PhoneNumber,VendorStreet,VendorCity,
              VendorState,VendorZipCode)
          ON PurchDB.Vendors TO Jim;

/* The following commands create a group called DBEUser  */
/*   for all other DBEUserIDs, and GRANTS specific       */
/*   authorities to this group.                          */

CREATE GROUP DBEUsers;

GRANT CONNECT TO DBEUsers;
GRANT RESOURCE TO DBEUsers;
GRANT SELECT,
      INSERT,
      DELETE,
      UPDATE
ON PurchDB.Parts
TO DBEUsers;

GRANT SELECT,
      INSERT,
      DELETE,
      UPDATE
ON PurchDB.Inventory
TO DBEUsers;

GRANT SELECT,
      INSERT,
      DELETE,
      UPDATE
ON PurchDB.Vendors
```

```
TO DBEUsers;

GRANT SELECT,
      INSERT,
      DELETE,
      UPDATE
ON PurchDB.Orders
TO DBEUsers;

GRANT SELECT,
      INSERT,
      DELETE,
      UPDATE
ON PurchDB.OrderItems
TO DBEUsers;

GRANT SELECT,
      INSERT,
      DELETE,
      UPDATE
ON PurchDB.VendorStatistics
TO DBEUsers;

GRANT SELECT,
      INSERT,
      DELETE,
      UPDATE
ON RecDB.Members
TO DBEUsers;

GRANT SELECT,
      INSERT,
      DELETE,
      UPDATE
ON RecDB.Clubs
TO DBEUsers;

GRANT SELECT,
      INSERT,
      DELETE,
      UPDATE
ON RecDB.Events
TO DBEUsers;
```

Data in the Sample DBEnvironment

There are three databases in the DBEnvironment PartsDBE-- ManufDB, PurchDB, and RecDB. Use the SELECT command to retrieve all the data in every table in each database, as shown in the following sections.

ManufDB.SupplyBatches Table

```
isql=> select * from manufdb.supplybatches;

select * from manufdb.supplybatches;
----------------+-------------------------+----------------
VENDPARTNUMBER  |BATCHSTAMP               |MINPASSRATE
----------------+-------------------------+----------------
7310            |1984-06-19 08:45:33.123  |            0.99
8113            |1984-06-14 11:13:15.437  |            0.93
790805          |1984-07-02 14:54:07.984  |            0.95
70250           |1984-07-22 09:06:23.319  |            0.97
9040            |1984-07-09 16:07:17.394  |            0.94
9050            |1984-07-13 09:25:53.183  |            0.97
29201           |1984-07-15 15:32:03.529  |            0.98
13350           |1984-07-25 10:15:58.159  |            0.97
549335          |1984-08-19 08:45:33.123  |            0.98
-------------------------------------------------------------------

Number of rows selected is 10
U[p], d[own], l[eft], r[ight], t[op], b[ottom], pr[int] <n>, or e[nd]>
```

ManufDB.TestData Table

```
   select * from manufdb.testdata;
   -------------------------+-----------+----------+----------+
   BATCHSTAMP               |TESTDATE   |TESTSTART |TESTEND   |
   -------------------------+-----------+----------+----------+
   1984-06-19 08:45:33.123  |1984-06-23 |08:12:19  |13:23:01  |
   1984-06-14 11:13:15.437  |1984-06-17 |08:05:02  |14:01:27  |
   1984-07-02 14:54:07.984  |1984-07-05 |14:03:21  |19:33:54  |
   1984-07-22 09:06:23.319  |1984-07-29 |14:01:28  |20:16:07  |
   1984-06-19 08:45:33.123  |1984-06-27 |08:02:29  |14:13:31  |
   1984-07-09 16:07:17.394  |1984-07-13 |08:43:16  |13:22:44  |
   1984-07-13 09:25:53.183  |1984-07-18 |14:07:01  |20:03:22  |
   1984-07-15 13:22:13.782  |1984-07-22 |09:01:48  |14:47:02  |
   1984-07-09 16:07:17.394  |1984-07-19 |08:13:26  |13:45:34  |
   1984-07-15 15:32:03.529  |1984-07-23 |14:02:34  |19:56:02  |
   1984-07-25 10:15:58.159  |1984-07-30 |08:25:11  |13:34:22  |
   1984-07-25 10:15:58.159  |1984-08-02 |08:01:13  |14:29:03  |
   1984-08-19 08:45:33.123  |1984-08-25 |08:12:19  |19:30:00  |
   ---------------------------------------------------------------
Number of rows selected is 13
U[p], d[own], l[eft], r[ight], t[op], b[ottom], pr[int] <n>, or e[nd] >

   -+----------------------+----------+----------
    |LABTIME               |PASSQTY   |TESTQTY
   -+----------------------+----------+----------
    |      0 05:10:42.000  |         49|         50
    |      0 05:56:25.000  |         47|         50
    |      0 05:30:33.000  |         48|         50
    |      0 06:14:39.000  |         50|         50
    |      0 06:11:02.000  |         49|         50
    |      0 04:39:28.000  |         46|         50
    |      0 05:56:21.000  |         49|         50
    |      0 05:45:14.000  |         50|         50
    |      0 05:32:08.000  |         49|         50
    |      0 05:53:28.000  |         49|         50
    |      0 05:09:11.000  |         48|         50
    |      0 06:27:50.000  |         47|         50
    |      5 04:23:00.000  |         49|         50
   ------------------------------------------------------------------------
   Number of rows selected is 13
   U[p], d[own], l[eft], r[ight], t[op], b[ottom], pr[int] <n>, or e[nd] >
```

PurchDB.Inventory Table

```
select partnumber,binnumber,qtyonhand,lastcountdate from purchdb.inventory;
   ---------------+---------+---------+-------------
   PARTNUMBER     |BINNUMBER|QTYONHAND|LASTCOUNTDATE
   ---------------+---------+---------+-------------
   1123-P-01      |     4003|        5|19841207
   1133-P-01      |     4007|       11|19841207
   1143-P-01      |     4016|        8|19841207
   1153-P-01      |     4027|        5|19841207
   1223-MU-01     |     5031|       12|19841207
   1233-MU-01     |     5036|       11|19841207
   1243-MU-01     |     5042|       15|19841207
   1323-D-01      |     3007|       12|19841207
   1333-D-01      |     3015|       47|19841207
   1343-D-01      |     3025|       18|19841207
   1353-D-01      |     3036|        6|19841207
   1423-M-01      |     2011|       10|19841207
   1433-M-01      |     2015|       18|19841207
   1523-K-01      |     1015|       16|19841207
   1623-TD-01     |     1095|       13|19841207
   1723-AD-01     |     6050|       25|19841207
   1733-AD-01     |     6055|       18|19841207
   1823-PT-01     |     7011|       10|19841207
   1833-PT-01     |     7035|       15|19841207
   1923-PA-01     |     7096|        7|19841207
   1933-FD-01     |     8016|        8|19841207
   1943-FD-01     |     9016|       23|19841207
   ------------------------------------------------------------------------
Number of rows selected is 22
U[p], d[own], l[eft], r[ight], t[op], b[ottom], pr[int] <n>, or e[nd]>

select countcycle,adjustmentqty,reorderqty,reorderpoint from
purchdb.inventory;
   ----------+-------------+----------+------------
   COUNTCYCLE|ADJUSTMENTQTY|REORDERQTY|REORDERPOINT
   ----------+-------------+----------+------------
           90|            4|        10|          30
           60|            7|        12|          14
           30|            1|        12|          10
           60|            4|        16|          20
           60|            4|        30|          60
           60|            3|        30|          30
           90|            4|        25|          30
           60|            7|        20|          20
           90|            8|        10|          50
           60|            9|         5|          15
           60|            3|        24|           6
           60|            5|        10|          10
           30|            4|         6|          16
           90|            2|         4|          16
```

```
          60|            1|        10|          10
          60|            3|        10|          20
          30|           -2|        10|          10
          60|           -5|        12|           8
          30|           -9|        10|          10
          90|           -5|        16|           8
          90|           -4|         6|          10
          60|            6|        10|          20
   ----------------------------------------------------------------------
Number of rows selected is 22
U[p], d[own], l[eft], r[ight], t[op], b[ottom], pr[int] <n>, or e[nd]>
```

PurchDB.OrderItems Table

```
select ordernumber,itemnumber,vendpartnumber from purchdb.orderitems;
-----------+-----------+---------------
ORDERNUMBER|ITEMNUMBER |VENDPARTNUMBER
-----------+-----------+---------------
      30507|          1|2310
      30507|          2|7310
      30508|          1|1110
      30508|          2|1115
      30508|          3|1113
      30508|          4|8113
      30509|          1|1533
      30509|          2|8113
      30510|          1|1001
      30510|          2|1005
      30511|          1|10175
      30511|          2|10675
      30511|          3|10975
      30512|          1|750001
      30512|          2|750101
      30512|          3|790115
      30512|          4|790805
      30513|          1|1010
      30513|          2|1050
      30514|          1|2310
      30514|          2|7310
      30515|          1|71705
      30515|          2|70150
      30515|          3|70250
      30515|          4|70500
      30515|          5|71755
      30516|          1|9040
      30516|          2|9050
      30516|          3|9060
      30516|          4|9080
      30516|          5|9090
      30517|          1|90015
      30517|          2|90035
      30517|          3|90045
      30518|          1|29201
      30518|          2|39201
      30518|          3|49201
      30518|          4|99201
      30519|          1|1010
      30519|          2|1050
      30520|          1|9375
      30520|          2|9105
      30520|          3|9135
      30521|          1|750001
      30521|          2|770105
```

```
         30521|          3|790805
         30522|          1|13350
   ------------------------------------------------------------------------
Number of rows selected is 47
U[p], d[own], l[eft], r[ight], t[op], b[ottom], pr[int] <n>, or e[nd]>

select purchaseprice,orderqty,itemduedate,receivedqty from
purchdb.orderitems;
   --------------+--------+-----------+-----------
   PURCHASEPRICE |ORDERQTY|ITEMDUEDATE|RECEIVEDQTY
   --------------+--------+-----------+-----------
          2000.00|       5|19840621   |          3
           565.00|      10|19840621   |         10
           450.00|       5|19840701   |          5
           180.00|       5|19840701   |          2
           210.00|       5|19840701   |          5
            70.00|      10|19840616   |          8
           435.00|       3|19840705   |          2
            70.00|       5|19840701   |          5
           345.00|       3|19840701   |          3
           195.00|       5|19840701   |          5
           180.00|       5|19840715   |          5
           195.00|       5|19840701   |          4
           195.00|       5|19840701   |          4
           475.00|       3|19840715   |          2
           175.00|       3|19840715   |          3
           450.00|       5|19840701   |          4
            80.00|      10|19840705   |         10
           335.00|       3|19840710   |          3
           650.00|       5|19840710   |          5
          2000.00|       5|19840715   |          3
           565.00|      10|19840715   |          7
           525.00|       3|19840710   |          3
           200.00|      10|19840726   |          8
           205.00|      10|19840726   |         10
          1985.00|       3|19840715   |          3
            70.00|      10|19840715   |          9
           190.00|      15|19840710   |         14
           200.00|      10|19840715   |         10
          1310.00|       5|19840715   |          5
          1650.00|       3|19840726   |          3
           240.00|       5|19840715   |          5
           200.00|      10|19840711   |          9
           220.00|       5|19840711   |          4
           645.00|       5|19840711   |          3
           195.00|      10|19840716   |         10
           180.00|      10|19840716   |         10
           210.00|       5|19840716   |          4
           590.00|       5|19840711   |          5
           335.00|       5|19840711   |          5
           650.00|       5|19840711   |          5
            95.00|      10|19840716   |          9
           450.00|       3|19840711   |          3
```

```
          1990.00|       3|19840711   |          3
           475.00|       5|19840727   |          4
          1295.00|       3|19840716   |          2
            80.00|      15|19840716   |         13
           200.00|      10|19840727   |         10
   ----------------------------------------------------------------------
Number of rows selected is 47
U[p], d[own], l[eft], r[ight], t[op], b[ottom], pr[int] <n>, or e[nd]>
```

PurchDB.Orders Table

```
isql=> select * from purchdb.orders;
-----------+------------+---------
ORDERNUMBER|VENDORNUMBER|ORDERDATE
-----------+------------+---------
      30507|        9001|19840601
      30508|        9002|19840601
      30509|        9002|19840615
      30510|        9006|19840615
      30511|        9004|19840615
      30512|        9008|19840615
      30513|        9010|19840626
      30514|        9001|19840626
      30515|        9012|19840626
      30516|        9015|19840626
      30517|        9003|19840627
      30518|        9009|19840627
      30519|        9010|19840627
      30520|        9013|19840627
      30521|        9008|19840627
      30522|        9014|19840627
      30523|        9014|19840628
-----------------------------------------------------------------------
Number of rows selected is 17
U[p], d[own], l[eft], r[ight], t[op], b[ottom], pr[int] <n>, or e[nd]>
```

PurchDB.Parts Table

```
select * from purchdb.parts;
----------------+------------------------------+-------------
PARTNUMBER      |PARTNAME                      |SALESPRICE
----------------+------------------------------+-------------
1123-P-01       |Central Processor             |        500.00
1133-P-01       |Communication Processor       |        200.00
1143-P-01       |Video Processor               |        180.00
1153-P-01       |Graphics Processor            |        220.00
1223-MU-01      |Cache Memory Unit             |         80.00
1233-MU-01      |Main Memory Unit              |        300.00
1243-MU-01      |Extended Memory Unit          |        100.00
1323-D-01       |Floppy Diskette Drive         |        200.00
1333-D-01       |Slimline Diskette Drive       |        200.00
1343-D-01       |Winchester Drive              |       2000.00
1353-D-01       |Standard Drive                |       1300.00
1423-M-01       |Video Monitor                 |        340.00
1433-M-01       |Graphics Monitor              |        650.00
1523-K-01       |Keyboard                      |        200.00
1623-TD-01      |Tape Drive                    |       1800.00
1723-AD-01      |Graphics Monitor Adapter      |        240.00
1733-AD-01      |Monochrome Displ/Prt Adapter  |        250.00
1823-PT-01      |Graphics Printer              |        450.00
1833-PT-01      |Color Printer                 |       1995.00
1923-PA-01      |Printer Adapter               |         75.00
1933-FD-01      |Fixed Disc Adapter            |        590.00
1943-FD-01      |Plotter Adapter               |        600.00
----------------------------------------------------------------------
Number of rows selected is 22
U[p], d[own], l[eft], r[ight], t[op], b[ottom], pr[int] <n>, or e[nd]>
```

PurchDB.Reports Table

```
   select reportname, reportowner, filedata from purchdb.reports;
   -------------------+---------------+----------------
   REPORTNAME         |REPORTOWNER    |FILEDATA
   -------------------+---------------+----------------
   Report1            |FREE           |>!Report1
   ------------------------------------------------------------------------
Number of rows selected is 1
U[p], d[own], l[eft], r[ight], t[op], b[ottom], pr[int] <n>, or e[nd]>
```

PurchDB.SupplyPrice Table

```
isql=> select partnumber, vendornumber,vendpartnumber from purchdb.suppl
   ----------------+------------+----------------
   PARTNUMBER      |VENDORNUMBER|VENDPARTNUMBER
   ----------------+------------+----------------
   1123-P-01       |        9002|1110
   1123-P-01       |        9003|90005
   1123-P-01       |        9007|35001
   1123-P-01       |        9008|750001
   1123-P-01       |        9009|19101
   1123-P-01       |        9012|71705
   1133-P-01       |        9002|1115
   1133-P-01       |        9003|90015
   1133-P-01       |        9007|35011
   1133-P-01       |        9009|29201
   1143-P-01       |        9004|10175
   1143-P-01       |        9007|35101
   1143-P-01       |        9008|750101
   1143-P-01       |        9009|39201
   1153-P-01       |        9002|1113
   1153-P-01       |        9003|90035
   1153-P-01       |        9007|35201
   1223-MU-01      |        9005|390121
   1223-MU-01      |        9013|9102
   1223-MU-01      |        9015|9010
   1233-MU-01      |        9005|390221
   1233-MU-01      |        9013|9115
   1233-MU-01      |        9015|9020
   1243-MU-01      |        9005|390321
   1243-MU-01      |        9013|9375
   1243-MU-01      |        9015|9030
   1323-D-01       |        9004|10675
   1323-D-01       |        9009|49201
   1323-D-01       |        9012|70150
   1323-D-01       |        9015|9040
   1333-D-01       |        9004|10975
   1333-D-01       |        9012|70250
   1333-D-01       |        9015|9050
   1343-D-01       |        9001|2310
   1343-D-01       |        9011|51050
   1353-D-01       |        9008|770105
   1353-D-01       |        9012|70350
   1353-D-01       |        9015|9060
   1423-M-01       |        9006|1001
   1423-M-01       |        9010|1010
   1433-M-01       |        9003|90045
   1433-M-01       |        9007|35801
   1433-M-01       |        9010|1050
   1523-K-01       |        9006|1005
   1523-K-01       |        9014|13350
```

```
   1623-TD-01      |        9011|55050
   1623-TD-01      |        9015|9080
   1723-AD-01      |        9004|10875
   1723-AD-01      |        9011|59050
   1723-AD-01      |        9012|70100
   1723-AD-01      |        9015|9090
   1733-AD-01      |        9004|10775
   1733-AD-01      |        9011|57050
   1733-AD-01      |        9012|70450
   1823-PT-01      |        9002|1533
   1823-PT-01      |        9008|790115
   1823-PT-01      |        9013|9105
   1833-PT-01      |        9012|70500
   1833-PT-01      |        9013|9135
   1923-PA-01      |        9002|8113
   1923-PA-01      |        9008|790805
   1923-PA-01      |        9012|71755
   1923-PA-01      |        9014|15550
   1933-FD-01      |        9001|7310
   1933-FD-01      |        9003|93715
   1933-FD-01      |        9007|35701
   1933-FD-01      |        9009|99201
   1933-FD-01      |        9014|16530
   1943-FD-01      |        9007|37502
   ------------------------------------------------------------------------
Number of rows selected is 69
U[p], d[own], l[eft], r[ight], t[op], b[ottom], pr[int] <n>, or e[nd]>

isql=> select unitprice,deliverydays,discountqty from purchdb.supplyp
   ------------------+------------+-----------
   UNITPRICE         |DELIVERYDAYS|DISCOUNTQTY
   ------------------+------------+-----------
              450.00|          30|          1
              475.00|          15|          5
              550.00|          15|          3
              475.00|          30|          5
              500.00|          20|          5
              525.00|          15|          2
              180.00|          30|          6
              200.00|          15|         10
              220.00|          15|         12
              195.00|          20|         15
              180.00|          30|         15
              185.00|          15|         12
              175.00|          30|          9
              180.00|          20|         10
              210.00|          30|         10
              220.00|          15|          8
              200.00|          15|          5
               75.00|          15|          3
               85.00|          30|          5
               80.00|          15|          5
              285.00|          15|          4
```

```
                295.00|          30|           3
                305.00|          15|          10
                100.00|          15|           8
                 95.00|          20|           9
                105.00|          15|          15
                195.00|          15|          10
                210.00|          20|          25
                200.00|          30|          20
                190.00|          15|          25
                195.00|          15|          21
                205.00|          30|          18
                200.00|          20|          17
               2000.00|          20|          15
               1950.00|          30|          18
               1295.00|          20|          20
               1300.00|          20|           5
               1310.00|          20|           3
                345.00|          15|           1
                335.00|          15|          19
                645.00|          15|          22
                700.00|          20|          15
                650.00|          15|          16
                195.00|          15|           5
                200.00|          30|           3
               1800.00|          15|          50
               1650.00|          30|          35
                260.00|          15|          10
                230.00|          15|          13
                250.00|          20|          11
                240.00|          20|          12
                250.00|          15|          18
                225.00|          20|          14
                255.00|          15|          19
                435.00|          20|           0
                450.00|          15|           2
                450.00|          15|           1
               1985.00|          20|           1
               1990.00|          15|           1
                 70.00|          15|           7
                 80.00|          20|           7
                 70.00|          20|           8
                 75.00|          10|           6
                565.00|          20|           5
                585.00|          15|           5
                600.00|          10|           5
                590.00|          15|           5
                585.00|          20|           3
                575.00|            |           3
   ------------------------------------------------------------------------
Number of rows selected is 69
U[p], d[own], l[eft], r[ight], t[op], b[ottom], pr[int] <n>, or e[nd]>
```

PurchDB.Vendors Table

```
   select vendornumber,vendorname,contactname from purchdb.vendors;
   ------------+-----------------------------+-------------------------
   VENDORNUMBER|VENDORNAME                   |CONTACTNAME
   ------------+-----------------------------+-------------------------
           9001|Remington Disk Drives        |Debra Thomason
           9002|Dove Computers               |Peter B. Galvin
           9003|Space Management Systems     |Stacey Wolf
           9004|Coupled Systems              |Micki Sue Ding
           9005|Underwood Inc.               |Diane Oliver
           9006|Pro-Litho Inc.               |Karen Thomas
           9007|Eve Computers                |Elisa Nissman
           9008|Jujitsu Microelectronics     |Adam D. Gerston
           9009|Latin Technology             |George Warrior
           9010|KellyCo Inc.                 |Celia Toledo
           9011|Morgan Electronics           |Tom Peterson
           9012|Semination al Co.            |Elizabeth Kramer
           9013|Seaside Microelectronics     |Geoff Grigsby
           9014|Educated Boards Inc.         |AJ White
           9015|Proulx Systems Inc.          |Michael Goldberg
           9016|Covered Cable Co.            |Phil Blank
           9017|SemiTech Systems             |Melissa Benson
           9018|Chocolate Chips              |Frederick Chung
   ---------------------------------------------------------------------
Number of rows selected is 18
U[p], d[own], l[eft], r[ight], t[op], b[ottom], pr[int] <n>, or e[nd]>

   select phonenumber,vendorstreet from purchdb.vendors;
   --------------+-----------------------------
   PHONENUMBER    |VENDORSTREET
   --------------+-----------------------------
   205 555 1234   |3006 Salvo St.
   303 234 5678   |123 Coyote Way
   408 456 7890   |3500 Scott Ave.
   206 677 2232   |1001 Island Way
   609 444 3579   |2001 Boardwalk
   408 765 2345   |17 Par Drive
   208 999 8642   |999 West 9th Street
   301 657 3579   |345 Black Boulevard
   408 555 9000   |280 Park Ave.
   617 333 0987   |555 Hillview Blvd.
   617 666 9182   |888 Industrial Parkway
   213 987 1423   |345 International Blvd.
   619 355 7565   |3210 Del Mar Blvd.
   602 987 0909   |4000 University Ave.
   408 290 5678   |404 Nosh Ave.
                  |777 Twisted Trail
                  |428 Tech Drive
                  |3425 Swirl Lane
   ---------------------------------------------------------------------
```

```
Number of rows selected is 18
U[p], d[own], l[eft], r[ight], t[op], b[ottom], pr[int] <n>, or e[nd]>

   select vendorcity,vendorstate,vendorzipcode from purchdb.vendors;
   --------------------+-----------+-------------
   VENDORCITY          |VENDORSTATE|VENDORZIPCODE
   --------------------+-----------+-------------
   Concord             |AL         |35567
   Littleton           |CO         |80123
   Santa Clara         |CA         |95033
   Puget Sound         |WA         |96122
   Atlantic City       |NJ         |10807
   Pebble Beach        |CA         |95012
   Snake River         |ID         |74503
   Bethesda            |MD         |20068
   San Jose            |CA         |95110
   Crabtree            |MA         |02135
   Braintree           |MA         |02088
   City of Industry    |CA         |92108
   Oceanside           |CA         |92078
   Phoenix             |AR         |60987
   Cupertino           |CA         |95035
   Bakersfield         |CA         |93662
   San Jose            |CA         |95130
   Lac du Choc         |MN         |32134
   ------------------------------------------------------------------------
Number of rows selected is 18
U[p], d[own], l[eft], r[ight], t[op], b[ottom], pr[int] <n>, or e[nd]>


   select vendorremarks from purchdb.vendors;
   ------------------------------------------------------------
   VENDORREMARKS
   ------------------------------------------------------------
   Slow shipping
   Discount rate 5%
   Slow shipping
   Discount rate 5.5%
   Discount rate 5%, slow shipping
   Poor service
   Discount rate 6%, purchase over $10,000
   No discount rate, fast shipping
   Often out of stock, fast shipping
   Discount rate 5.5%
   Fast shipping, 5% Discount
   Discount rate 6% for order over $15000
   Discount rate 10%, very slow shipping
   Discount rate 5%, fast shipping
   Discount 6%, fast shipping
   ------------------------------------------------------------------------
Number of rows selected is 18
U[p], d[own], l[eft], r[ight], t[op], b[ottom], pr[int] <n>, or e[nd]>
```

RecDB.Clubs Table

```
select * from recdb.clubs;
---------------+---------+------------------
CLUBNAME       |CLUBPHONE|ACTIVITY
---------------+---------+------------------
Energetics     |     1111|aerobics
Windjammers    |     2222|sailing
Downhillers    |     3333|skiing
Poker Faces    |     4444|cards
Spikers        |     5555|volleyball
Stingers       |     6666|soccer
Green Thumbs   |     7777|gardening
Crescendos     |     8888|music
Keys 'n Strings|     9999|music
---------------------------------------------------------------------
Number of rows selected is 9
U[p], d[own], l[eft], r[ight], t[op], b[ottom], pr[int] <n>, or e[nd]>
```

RecDB.Events Table

```
   select * from recdb.events;
   ---------------+-----------------------------+--------+------+------
SPONSORCLUB    |EVENT                          |DATE    |TIME  |COORDINAT
   ---------------+-----------------------------+--------+------+------
Energetics     |beginning exercises          |19861201|  1230|Martha Mi
Energetics     |advanced stretching          |19861204|  1530|Martha Mi
Windjammers    |holiday regatta              |19861226|   900|Bill Hale
Downhillers    |slalom race                  |19861231|   600|Karen Man
Poker Faces    |game                         |19861201|  2100|Al Krebbs
Poker Faces    |game                         |19861205|  1800|Marty Tho
Spikers        |winter play-offs             |19861206|   700|Nancy Cun
Stingers       |tournament round 1           |19861219|  1100|Jorge Pab
Stingers       |tournament round 2           |19861220|  1000|Stacey Va
Green Thumbs   |weed killing seminar         |19861227|   900|Annie And
Green Thumbs   |dwarf tree planting          |19861207|  1200|Sue Peter
Crescendos     |cantata rehearsal            |19861220|  1400|Karen Llo
Crescendos     |cantata                      |19861224|  2000|Mariann H
Keys 'n Strings|rehearsal for New Year's     |19861213|  1300|Karen Wal
Keys 'n Strings|New Year's Eve concert       |19861231|  1800|Wolfgang
   ---------------------------------------------------------------------
   Number of rows selected is 15
   U[p], d[own], l[eft], r[ight], t[op], b[ottom], pr[int] <n>, or e[nd]>
```

RecDB.Members Table

```
select * from recdb.members;
--------------------+---------------+-----------
MEMBERNAME          |CLUB           |MEMBERPHONE
--------------------+---------------+-----------
George Smith        |Poker Faces    |       1476
Darab Jones         |Stingers       |       2605
Annie Anderson      |Green Thumbs   |       1105
Annie Anderson      |Stingers       |       1105
Sue Peters          |Green Thumbs   |       7505
Al Krebbs           |Poker Faces    |       9615
John Ewing          |Crescendos     |       6925
John Ewing          |Energetics     |       6925
Wolfgang Ross       |Keys 'n Strings|       6255
MJ Kipper           |Keys 'n Strings|       3305
Jim Johnson         |Spikers        |       8625
Becky Gardner       |Spikers        |       5605
John Brown          |Downhillers    |       3605
Doug Griffith       |Downhillers    |       5915
Jorge Pablo         |Stingers       |       7655
Marguerite Harris   |Crescendos     |       1605
Bill Haley          |Windjammers    |       5505
Ragaa Morrow        |Keys 'n Strings|       4405
Ragaa Morrow        |Energetics     |       4405
Miranda Wong        |Windjammers    |       2105
Glen Stevens        |Windjammers    |       8005
Peter Crane         |Stingers       |       1205
David Loomis        |Windjammers    |       9505
Sharon Means        |Keys 'n Strings|       2305
Karen Manor         |Downhillers    |       7005
Marty Thomas        |Poker Faces    |       6305
Mariann Humphrey    |Crescendos     |       9105
Renee Ball          |Crescendos     |       3105
Diane Rizzo         |Green Thumbs   |       1715
Martha Mitchell     |Energetics     |       1605
Robert Klein        |Energetics     |       9005
Nancy Cunning       |Spikers        |       4605
May-Inn Kong        |Energetics     |       8505
Karen Walters       |Keys 'n Strings|       1665
Karen Lloyd         |Crescendos     |       1715
Tom Thumb           |Poker Faces    |       2715
Stacey Valley       |Stingers       |       3405
--------------------------------------------------------------------------
Number of rows selected is 37
U[p], d[own], l[eft], r[ight], t[op], b[ottom], pr[int] <n>, or e[nd]>
```

Sample Program Files

The following table contains a list of sample program files located in the /usr/lib/allbase/hpsql/programs directory.

All programs *except* those marked with an asterisk (*) are fully discussed in the ALLBASE/SQL application programming guides.

Table C-1. Sample Programs in /usr/lib/allbase/hpsql/programs

C	COBOL	FORTRAN	Pascal	Description
cex2	cobex2	forex2	pasex2	Single-row `SELECT` into host variables from PurchDB.Parts
cex5	cobex5	forex5	pasex5	Single-row `SELECT` into host variables from PurchDB.Parts with implicit and explicit error handling, including recovery from deadlock
cex7	cobex7	forex7	pasex7	Single-row `SELECT`, `INSERT`, `UPDATE`, and `DELETE` operations on the PurchDB.Vendors table
cex8	cobex8	forex8	pasex8	Cursor manipulations on the PurchDB.OrderItems table
cex8a*	cobex8a*	forex8a*	pasex8a*	DATE/TIME data types and join operations on the TestData and SupplyBatches tables.
cex9	cobex9		pasex9	BULK operations on the PurchDB.Orders and PurchDB.OrderItems tables
	cobex10a	forex9a		Dynamic non-query commands using `EXECUTE IMMEDIATE`
	cobex10b	forex9b		Dynamic non-query commands using `PREPARE` and `EXECUTE`
cex10a			pasex10a	Dynamic query commands with unknown Format
cex10b			pasex10b	Dynamic query commands with known Format
cex12*	cobex12*	forex12*	pasex12*	Using long fields in the PurchDB.Reports table

D Standards Flagging Support

Introduction

The United States government has adopted ANSI X3.135-1989, Database Language SQL, as the database language to be used by all federal departments and agencies. This SQL standard, known as Federal Information Processing Standard 127.1 (FIPSPUB 127.1), requires that an option be provided which flags all features or extensions that do not conform to the SQL language or are processed in a nonconforming manner. FIPS 127.1 also has added an optional integrity enhancement feature, addendum 1, to X3.135-1989. Addendum 1 includes referential integrity constraints, a check clause, and a default clause. A feature does not have to be flagged if it conforms to addendum 1.

The SQL standard does not contain functionality for many common categories, such as storage management and index creation. While many of these non-standard features are useful, they can reduce the portability of programs that use them. Most SQL implementations (including ALLBASE/SQL) support implementation-defined features that do not conform to FIPS 127.1. These non-standard implementation features are of concern to users who want to port programs and who need to identify features that do not conform to FIPS 127.1. In order to recognize features and extensions that do not conform to the SQL standard, FIPS 127.1 requires that a flagger capability be implemented that identifies any non-standard features. This flag can be implemented through software or in documentation. In addition to this appendix, ALLBASE/SQL provides flagger options for preprocessing and a SET FLAGGER command in ISQL. Refer to the *ALLBASE/SQL Advanced Application Programming Guide* and the *ALLBASE/ISQL Refernece Manual* respectively for related documentation.

Non-standard Statements and Extensions

The following tables contain ALLBASE/SQL statements and extensions and indicate whether they are compliant with FIPS 127.1. If the ALLBASE/SQL statement is not compliant, the extensions to that statement are not compliant and are therefore not included in the table. A compliant statement may also have non-compliant extensions. These extensions are shown as non-compliant in the table.

Table D-1. ALLBASE/SQL FIPS 127.1 Compliance

ALLBASE/SQL Statement	FIPS 127.1 Compliant Statement?	Extension to Statement	FIPS 127.1 Compliant Extension?
ADD DBEFILE	NO		
ADD TO GROUP	NO		
ADVANCE	NO		
ALTER DBEFILE	NO		
ALTER TABLE	NO		
BEGIN	NO		
BEGIN ARCHIVE	NO		
BEGIN DECLARE SECTION	YES		
BEGIN WORK	NO		
CHECKPOINT	NO		
CLOSE	YES	Using	NO
COMMIT ARCHIVE	NO		
COMMIT WORK	YES	RELEASE	NO
CONNECT	NO		
CREATE DBEFILE	NO		
CREATE DBEFILESET	NO		
CREATE GROUP	NO		
CREATE INDEX	NO		
CREATE PARTITION	NO		
CREATE PROCEDURE	NO		
CREATE RULE	NO		
CREATE SCHEMA	YES	*TableDefinition*	YES

Table D-1. ALLBASE/SQL FIPS 127.1 Compliance

ALLBASE/SQL Statement	FIPS 127.1 Compliant Statement?	Extension to Statement	FIPS 127.1 Compliant Extension?
		ViewDefinition	YES
		IndexDefinition	NO
		ProcedureDefinition	NO
		RuleDefinition	NO
		CreateGroup	NO
		AddToGroup	NO
		GrantStatement	YES
CREATE TABLE	Only when used in CREATE SCHEMA	PUBLIC	NO
		PUBLICREAD	NO
		PRIVATE	NO
		PUBLICROW	NO
		LANG=*TableLangName*	NO
		UNIQUE HASH ON	NO
		HASH ON CONSTRAINT	NO
		ConstraintID	NO
		CLUSTERING ON CONSTRAINT	NO
		IN *DBEFileSetName1*	NO
		ColumnDefinition	NO
		UniqueConstraint	NO
		ReferentialConstraint	NO
		CheckConstraint	NO
CREATE TABLE	Only when used in CREATE SCHEMA	*UniqueConstraint*	NO
		ReferentialConstraint	NO
		CheckConstraint	NO
		ColumnDataType	YES

Table D-1. ALLBASE/SQL FIPS 127.1 Compliance

ALLBASE/SQL Statement	FIPS 127.1 Compliant Statement?	Extension to Statement	FIPS 127.1 Compliant Extension?
		LongColumnType	NO
		LANG=*ColLangName*	NO
		DEFAULT	YES
		Constant	YES
		NULL	YES
		CurrentFunction	NO
		NOT NULL	YES
		UNIQUE	YES
		PRIMARY KEY REFERENCES	YES
		RefTableName	YES
		CONSTRAINT *ConstraintID*	NO
		CHECK	YES
		Case Sensitive	NO
		IN *DBEFileSetName3*	NO
CREATE TEMPSPACE	NO		
CREATE VIEW	Only when used in CREATE SCHEMA	*ColumnName*	YES
		WITH CHECK OPTION	YES
		CONSTRAINT *ConstraintID*	NO
		IN *DBEFileSetName*	YES
DECLARE CURSOR	YES	IN *DBEFileSetName*	NO
		FOR UPDATE OF *ColumnName*	NO
		FOR READ ONLY	NO
		QueryExpression	YES
		ExecuteProcedureName	NO
		ExecuteStatementName	NO

Table D-1. ALLBASE/SQL FIPS 127.1 Compliance

ALLBASE/SQL Statement	FIPS 127.1 Compliant Statement?	Extension to Statement	FIPS 127.1 Compliant Extension?
		SelectStatementName	
DELETE	YES	WITH AUTOCOMMIT	NO
DELETE WHERE CURRENT	YES		
DESCRIBE	NO		
DISABLE RULES	NO		
DISCONNECT	NO		
DROP DBEFILE	NO		
DROP DBEFILESET	NO		
DROP GROUP	NO		
DROP INDEX	NO		
DROP MODULE	NO		
DROP PARTITION	NO		
DROP PROCEDURE	NO		
DROP RULE	NO		
DROP TABLE	NO		
DROP TEMPSPACE	NO		
DROP VIEW	NO		
ENABLE AUDIT LOGGING	NO		
ENABLE RULES	NO		
END DECLARE SECTION	YES		
EXECUTE	NO		
EXECUTE IMMEDIATE	NO		
EXECUTE PROCEDURE	NO		
EXTRACT	NO		
FETCH	YES	BULK INTO *HostVariableSpec* USING *clause*	NO YES YES

Table D-1. ALLBASE/SQL FIPS 127.1 Compliance

ALLBASE/SQL Statement	FIPS 127.1 Compliant Statement?	Extension to Statement	FIPS 127.1 Compliant Extension?
		[SQL]DESCRIPTOR	YES
		HostVariableSpec	NO
		INDICATOR	YES
GENPLAN	NO		
GRANT	Only when used in CREATE SCHEMA	ALL (without the PRIVILEGE QUALIFIER)	NO
		SELECT	YES
		INSERT	YES
		DELETE	YES
		ALTER	NO
		INDEX	NO
		UPDATE *ColumnName*	YES
		REFERENCES *ColumnName*	YES
		TableName	YES
		TableView	YES
		DBEUserID	YES
		GroupName	NO
		ClassName	NO
		PUBLIC WITH GRANT OPTION	YES
		BY	NO
		RUN ON	NO
		EXECUTE ON PROCEDURE	NO
		CONNECT	NO
		DBA	NO
		RESOURCE	NO
		MONITOR	NO

Table D-1. ALLBASE/SQL FIPS 127.1 Compliance

ALLBASE/SQL Statement	FIPS 127.1 Compliant Statement?	Extension to Statement	FIPS 127.1 Compliant Extension?
		INSTALL	NO
		DBEFileSet	NO
INCLUDE	NO		
INSERT	YES	BULK	NO
		SingleRowValues	YES
BulkValues	NO		
INSERT SingleRowValues	YES	NULL	YES
		USER	NO
		HostVariable	YES
		INDICATOR	YES
		?	NO
		:LocalVariable	NO
		:ProcedureParameter	NO
		::Built-inVariable	NO
		ConversionFunction	NO
		CurrentFunction	NO
		+	YES
		-	YES
		Integer	YES
		Float	YES
		Decimal	YES
		'CharacterString'	YES
		0*xHexadecimalString*	NO
		LongColumnString	NO
LOCK TABLE	NO		
LOG COMMENT	NO		
OPEN	YES	KEEP CURSOR	NO
		WITH LOCKS,	NO

Table D-1. ALLBASE/SQL FIPS 127.1 Compliance

ALLBASE/SQL Statement	FIPS 127.1 Compliant Statement?	Extension to Statement	FIPS 127.1 Compliant Extension?
		WITH NOLOCKS	NO
		USING *clause*	NO
PREPARE	NO		
RAISE ERROR	NO		
REFETCH	NO		
RELEASE	NO		
REMOVE DBEFILE	NO		
REMOVE FROM GROUP	NO		
RENAME	NO		
RESET	NO		
REVOKE	NO		
ROLLBACK WORK	YES	TO	NO
		RELEASE	NO
SAVEPOINT	NO		
SELECT	YES	BULK	NO
		ORDER BY	YES
		ColumnID	YES
		ASC | DESC	YES
		QueryBlock	YES
		(QueryExpression)	YES
		UNION	YES
		ALL	YES
		DISTINCT	YES
		INTO	YES
		WHERE *SearchCondition1*	YES
		GROUP BY *GroupColumnList*	YES

Table D-1. ALLBASE/SQL FIPS 127.1 Compliance

ALLBASE/SQL Statement	FIPS 127.1 Compliant Statement?	Extension to Statement	FIPS 127.1 Compliant Extension?
		HAVING *SearchCondition2*	YES
		NATURAL JOIN	NO
		INNER JOIN	NO
		LEFT JOIN	NO
		RIGHT JOIN	NO
		OUTER JOIN	NO
SET CONNECTION	NO		
SET CONSTRAINTS	NO		
SET DEFAULT	NO		
SET DML ATOMICITY	NO		
SET MULTITRANSACTION	NO		
SET PRINTRULES	NO		
SET SESSION	NO		
SET TRANSACTION	NO		
SET USER TIMEOUT	NO		
SETOPT	NO		
SQLEXPLAIN	NO		
START DBE	NO		
START DBE NEW	NO		
START DBE NEWLOG	NO		
STOP DBE	NO		
STOREINFO	NO		
TERMINATE USER	NO		
TRANSFER OWNERSHIP	NO		
TRUNCATE TABLE	NO		
UPDATE	YES	*'LongColumnIOString'*	NO
UPDATE STATISTICS	NO		

Table D-1. ALLBASE/SQL FIPS 127.1 Compliance

ALLBASE/SQL Statement	FIPS 127.1 Compliant Statement?	Extension to Statement	FIPS 127.1 Compliant Extension?
UPDATE WHERE CURRENT	YES	*'LongColumnIOString'*	NO
		NULL	YES
WHENEVER	YES	SQLERROR	YES
		SQLWARNING	NO
		NOT FOUND	YES
		STOP	NO
		CONTINUE	YES
		GOTO *Label*	YES
		: (colon; not required)	NO

Non-Standard Data Types

The following data types are *not* FIPS compliant. They are used in CREATE TABLE and ALTER TABLE column definitions.

VARCHAR

DATE

TIME

DATETIME

INTERVAL

BINARY

VARBINARY

LONG BINARY

LONG VARBINARY

Non-Standard Expression Extensions

The following use of extensions in an expression is *not* FIPS compliant:

TID

DynamicParameters

OUTER JOIN

NATURAL JOIN

STRING_LENGTH

SUBSTRING

OUTPUT_DEVICE

OUTPUT_NAME

CURRENT_DATE

CURRENT_TIME

CURRENT_DATETIME

TO_CHAR

TO_DATE

TO_TIME

TO_DATETIME

TO_INTERVAL

ADD_MONTHS

TO_INTEGER

CAST

Concatenation (||)

Non-Standard Syntax Rules

ALLBASE/SQL supports certain non-FIPS compliant extensions to the standard FIPS syntax rules listed in the ANSI SQL/89 document. The section number, the status rule number, and the FIPS SQL syntax rule for each non-FIPS compliant extension are listed below.

Note that in some cases no flagger warning is generated for these exceptions.

Table D-2. FIPS Syntax Rules and ALLBASE/SQL Exceptions

Section # Status Rule #	FIPS SQL Syntax Rule	ALLBASE SQL Extension
5.3 SR 3	All identifiers must be no longer than 18 characters.	20 characters are allowed.
5.3 SR 4	No identifier may be the same as a keyword, noting that all keywords are specified in upper case.	Keywords are not case sensitive; keywords can be identifiers.
5.24 SR 8	There may only be one DISTINCT per subquery, not including any nested subqueries.	SELECT DISTINCT MAX (DISTINCT C1) FROM T1 is valid.
5.25 SR 5	There may only be one DISTINCT per query, not including any subqueries in that query.	SELECT DISTINCT MAX (DISTINCT C1) FROM T1 is valid.
5.25 SR 11b	Every <value expression> in the <select list> consists of a <column specification>, and no <column specification> appears more than once - updatability of a table/view.	It is possible to update a regular column in a view that contains a virtual column. (No flagger warning is generated).
6.1	DDL commands must be used in the CREATE SCHEMA statement.	DDL commands can be issued outside of a `CREATE SCHEMA` statement.
8.6 SR 3b	In a FETCH, only an exact numeric column or expression may be FETCHed into an exact numeric host variable.	Compatible data and truncation are allowed. (No flagger warning is generated).
8.6 SR 6a	An INSERT into a character column must be a character string of length less than or equal to the column.	Compatible data and truncation are allowed. (No flagger warning is generated).
8.7 SR 6b	An INSERT into an exact numeric column must be an exact numeric value.	Compatible data and truncation are allowed. (No flagger warning is generated).
8.10 SR 4b	In a SELECT ... INTO, only an exact numeric column or expression may be selected into an exact numeric host variable.	Compatible data and truncation are allowed. (No flagger warning is generated).

Table D-2. FIPS Syntax Rules and ALLBASE/SQL Exceptions

Section # Status Rule #	FIPS SQL Syntax Rule	ALLBASE SQL Extension
8.11 SR 8b	An UPDATE ... WHERE CURRENT of a character column must be a character string of length less than or equal to the column.	Compatible data and truncation are allowed. (No flagger warning is generated).
8.11 SR 8c	In an UPDATE ... WHERE CURRENT, only an exact numeric value or NULL may be put in an exact numeric column.	Compatible data and truncation are allowed. (No flagger warning is generated).
8.12 SR 6b	A searched UPDATE of a character column must be a character string of length less than or equal to the column.	Compatible data and truncation are allowed. (No flagger warning is generated).
8.12 SR 6c	In a searched UPDATE, only an exact numeric value (or NULL) may be put in an exact numeric column.	Compatible data and truncation are allowed. (No flagger warning is generated).
9.2 SR 1bc	COLON has to precede the host identifier in WHENEVER.	The absence of COLON in the front of the label is allowed.

NOTE There is one more exception to the syntax rules listed above:

No flagger warning is generated for a second reference to a non-standard extension within the first non-standard reference.

Index

A

access plan
  defined, 429
  modifying with SETOPT, 531
access to databases
  multiple connections, 95
  types, 42
active connection
  defined, 103
active set
  in DECLARE CURSOR, 372
  in FETCH, 424
  in OPEN, 464
  in REFETCH, 476
actual parameter
  using in procedures, 149
ADD DBEFILE
  syntax, 293
ADD TO GROUP
  syntax, 295
ADD_MONTHS
  in an expression, 228
adding
  ADD TO GROUP, 295
  column to table, 301
  constraint to table, 301
  DBEFiles, 293
  members to authorization group, 295
  rows, 81, 445
ADVANCE
  syntax, 297
aggregate functions
  in an expression, 228
  in NULL predicates, 275
ALL
  in quantified predicate, 278
  in SELECT, 505
all audit element
  in START DBE NEW, 555
  in START DBE NEWLOG, 563
ALLBASE/SQL
  components, 42
  data types, 207
  definition, 41
  message catalog, 550
  names, 201
  users, 51
allocating file space
  in CREATE DBEFILESET, 330
ALTER DBEFile
  syntax, 299
ALTER TABLE
  syntax, 301
  to change table locking, 301
  to set audit partition, 301
altering
  DBEFile type, 299
  tables, 301
ANY
  in quantified predicate, 278
application programming
  and SQL statements, 93
archive logging
  wrapperDBE, 92
archive mode
  definition, 560
  use of BEGIN ARCHIVE, 310
  use of COMMIT ARCHIVE, 322
archive record
  use of COMMIT ARCHIVE, 322
arithmetic operators
  in an expression, 228
asymmetric outer join
  defined, 127
atomicity
  setting in SET TRANSACTION, 542
audit
  DBE understanding, 91
  disabling logging, 389
  elements default, 91
  elements understanding, 91
  functionality definition, 91
  information wrapperDBE, 92
  log record, 91
  partition with ALTER TABLE, 301
  tool, 92
  transactions with SQLAudit, 92
audit logging
  enabling, 411
AUDIT NAME
  in START DBE NEW, 555
  in START DBE NEWLOG, 563
authorities
  and program development, 94
  and program use, 94
  defined, 45
  granting, 76, 436
  granting on DBEFileSET, 439
  how to obtain, 75
  OWNER, 75
  REFERENCES, 75
  revoking, 76, 489
  RUN, 75
  summary of types, 75
  table and view, 75
authorization
  audit element in START DBE NEWLOG, 563
  DBEFileSet in CREATE PROCEDURE, 344, 354, 373
  DBEFileSet in CREATE VIEW, 369
  DBEFileSet in PREPARE, 468
  DBEFileSet in REVOKE, 493
  for a select cursor, 373
  in START DBE NEW, 555
  long column in ALTER TABLE, 306, 307
  long column in CREATE TABLE, 362
  name, 204
  section in PREPARE, 468
  SECTIONSPACE in CREATE RULE, 349
authorization groups
  adding members to, 295
  advantages of, 79
  creating, 332
  dropping, 397
  removing members from, 482
  use of, 79
AUTOCOMMIT
  DELETE parameter, 378
  VALIDATE parameter, 592
automatic locking modes
  in CREATE TABLE, 64
autostart
  and CONNECT, 325
  mode, 560

B

base table
  defined, 67
basic names
  objects having, 201
  rules governing, 201
BEGIN
  syntax, 309
BEGIN ARCHIVE
  syntax, 310
BEGIN DECLARE SECTION
  syntax, 311
BEGIN WORK
  and MULTITRANSACTION, 529
  in a procedure, 147
  read committed isolation level, 312
  read uncommitted isolation level, 312
  repeatable read isolation level, 312
  syntax, 312
BETWEEN predicate
  in search condition, 262
  syntax, 264
BINARY

conversions rules, 504
long data type defined, 208
storage requirements, 210
built-in variable
differences from local variables, 151
similar to SQLCA elements, 151
using in procedures, 151
BULK operations
FETCH, 424
INSERT, 445
SELECT, 499
use of, 95

C

C preprocessor
defined, 42
caller of a procedure
recommended practices for, 156
Cartesian product
defined, 114
CASCADE
explained, 77
case sensitive
comparison predicate, 266
comparisons, 211
CATALOG
owner of catalog views, 206
catalog views
explained, 105
chain
of grants, 76
of rules, 159, 164
changing
connections, 529
data, 82
DBEFile type, 299
table locking, 301
CHAR
conversions rules, 504
defined, 208
native language data, 226
storage requirements, 210
check constraint
defined, 68, 139
in a view, 141
in ALTER TABLE, 301
in CREATE TABLE, 354, 359
search condition, 140
CHECKPOINT
record, 316
syntax, 316
use of, 316
checkpoint for STOP DBE, 571
class names
in REVOKE, 489, 491
rules for, 201
classes
creating, 80
use of, 80
clause
defined, 52
CLOSE
syntax, 319
closing cursors, 319
clustered indexes, 334
COBOL preprocessor
defined, 42
MICROFOCUS, 42
coding practices
for procedures, 156
column names
in ALTER TABLE, 301
in an expression, 228
in CREATE INDEX, 334
in CREATE TABLE, 357
in CREATE VIEW, 367
in INSERT, 445
in null predicates, 275
rules for, 201
columns
adding to tables, 301
and Cartesian product, 513
common columns in join, 512
defined, 44
defining, 355
definition, 65
maximum allowed in tables, 357
maximum allowed in views, 367
order of display, 512
comment audit element
in START DBE NEW, 555
in START DBE NEWLOG, 563
comment initiator
within SQL statements, 53
comment partition
in START DBE NEW, 555
in START DBE NEWLOG, 563
COMMIT ARCHIVE
syntax, 322
COMMIT WORK
in a procedure, 147
syntax, 323
common columns
in SELECT, 512
COMPARISON predicate
character collation sequence, 266
in search condition, 262
operators, 265
syntax, 265
complex queries
defined, 116
range of types, 116
compound identifiers
in names, 203
concatenate
data types, 231
strings, 230
concurrency
and table size, 178
control, 167
configuring a DBEnvironment
in START DBE NEW, 555
summarized, 60
CONNECT
syntax, 325
connection
and SET CONNECTION, 519
changing, 529
disconnecting, 391
initiating, 325
terminating, 103
to DBEnvironments, 96
use with timeouts, 95
connection name
in CONNECT, 325
in SET CONNECTION, 519
in START DBE, 552
in START DBE NEWLOG, 563
constant
in an expression, 228
in NULL predicates, 275
constraint
check constraint, 68, 137
defined, 137
defining, 354
error checking and SET statement, 521
error checking in SET TRANSACTION, 542
example, 141
constraint checking
setting in SET TRANSACTION, 542
control block
in START DBE, 552
in START DBE NEW, 555
in START DBE NEWLOG, 563
control flow statements
in procedures, 149
RETURN, 487
control language
in procedures, 442, 597
controlling
error checking level, 526
conversion rules
data in query expressions, 504
copying rows
to tables and views, 82
correlated subquery

explained, 126
CREATE DBEFILE
syntax, 327
CREATE DBEFILESET
syntax, 330
CREATE GROUP
syntax, 332
CREATE INDEX
syntax, 334
CREATE PARTITION
syntax, 337
CREATE PROCEDURE
explained, 146
syntax, 339
CREATE RULE
invoking a procedure through, 147
syntax, 346
using, 158
CREATE SCHEMA
syntax, 351
CREATE TABLE
LANG = clause, 67
syntax, 354
CREATE TEMPSPACE
syntax, 365
CREATE VIEW
syntax, 367
creating
audit DBE, 91
authorization groups, 332
constraints, 354
databases, 60
DBEFile, 327
DBEFileSets, 330
DBEnvironments, 60, 555
indexes, 334
partition, 91, 337
tables, 354
TempSpace, 365
views, 367
CS isolation level
explained, 175
in SET SESSION, 537
in SET TRANSACTION, 543
current
function in an expression, 228
language defined, 57
row in FETCH, 424
row in REFETCH, 476
timeout value in multi-transaction, 99
current connection
in CONNECT, 325
none after DISCONNECT CURRENT, 104
setting, 96
cursor names
in CLOSE, 319
in DECLARE CURSOR, 371
in DELETE WHERE CURRENT, 381
in FETCH, 424
in OPEN, 464
in UPDATE WHERE CURRENT statement, 587
rules for, 201
cursor stability (CS)
explained, 175
cursor stability isolation level
in SET SESSION, 536
in SET TRANSACTION, 542
cursors
active set, 424, 476
advancing, 297
closed, 571
closing, 319, 479, 495
current row, 424, 476
declaring, 371
deleting rows with, 372, 381
in procedures, 153
opening, 464
procedure cursor parameters in, 149
retrieving rows with, 372, 424, 476
updatability of, 372
updating data with, 371, 372
use of, 95, 371
using in procedures, 160
using multiple, 464
cycle
in chain of grants, 76

D

data
access, 48, 75
in native languages, 226
manipulation, 109
data audit element
in START DBE NEW, 555
in START DBE NEWLOG, 563
data buffer pages
in START DBE, 552
in START DBE NEW, 555
in START DBE NEWLOG, 563
data definition statements
in procedures, 164
data types
comparisons between, 211
conversion, 213
effect of, 207
of column added to existing table, 301
of columns in joins, 512
rules governing, 207
table of, 208
valid combinations, 213
database
administration activities, 105
administrator defined, 51
control of access to, 75
creation, 60
creation (CREATE SCHEMA), 351
logical definition, 44
physical definition, 45
statistics, 106
DATE
conversions rules, 504
defined, 209
operations with values, 217
storage requirements, 210
values in arithmetic expression, 218
date/time conversion functions
in an expression, 228
DATETIME
conversions rules, 504
defined, 209
operations with values, 217
storage requirements, 210
values in arithmetic expressions, 218
DBA
automatic grant of authority, 76
defined, 51
statements authorized to use, 76
DBE sessions
and autostart mode, 62
defined, 48
multiuser, 62
setting the current connection, 519
single-user, 62
starting, 62, 325, 552
terminating, 62, 479, 572, 574
DBECon file
creation, 555
defined, 47
naming conventions, 205
overriding parameters, 552
parameters, 560
DBECreator
authorization, 76
defined, 61
statements authorized to use, 76
DBEFile names
in ADD DBEFile, 293

in ALTER DBEFile, 299
in CREATE DBEFILE, 327
in DROP DBEFILE, 393
in REMOVE DBEFile, 480
rules for, 201
DBEFile type
in ALTER DBEFile, 299
in CREATE DBEFILE, 327
DBEFile0
defined, 47
naming, 559
DBEFiles
adding, 293
altering type of, 299
creating, 327
defined, 45
dropping, 393, 480
for data, 45
for indexes, 45
incrementing size, 327
purging, 393
relation to DBEFileset, 45
removing from DBEFileSet, 393, 395, 480
size range, 327
type, 327
using, 327
DBEFileSet
authorization in CREATE PROCEDURE, 344, 354, 373
authorization in CREATE VIEW, 369
authorization in PREPARE, 468
authorization in REVOKE, 493
creating, 330
defined, 45
dropping, 395
dropping default, 395
dynamic section, 468
dynamic statement, 468
for a check constraint, 354
for a long column, 354
for a table, 354
in DECLARE CURSOR, 371
relation to DBEFiles, 45
setting a default, 524
specifying for a cursor, 371
specifying for a view, 367
DBEFileSet names
in ADD DBEFile, 293
in CREATE TABLE, 360
in REMOVE DBEFile, 480
rules for, 201
DBELog1
defined, 47
DBELog2
defined, 47
DBEnvironment
components, 47
configuration, 60, 555
connecting to, 96
creating, 60
creating audit, 91
defined, 47
disconnecting from, 103
initial privileges, 61
naming conventions, 205
obtaining information on, 106
startup parameters, 60
statistics, 106
DBEnvironment name
in CONNECT, 325, 519
in DISCONNECT, 391
in START DBE, 552
in START DBE NEW, 555
in START DBE NEWLOG, 563
DBEUserID
defined, 203
in ADD TO GROUP, 295
in GRANT, 437
in REMOVE FROM GROUP, 482
in RESET, 486
in REVOKE, 489, 491
in TERMINATE QUERY, 572
in TERMINATE USER, 574
rules governing, 203
DDL Enabled flag
defined, 561
deadlock
avoidance of, 197
definition, 195
detection/resolution, 195
example, 195
in multi-transaction mode, 97
DECIMAL
conversions rules, 504
defined, 208
in operations, 216
storage requirements, 210
DECLARE
and local variables in a procedure, 150
DECLARE %%Variable%%
syntax, 376
DECLARE CURSOR
syntax, 371
declaring
cursors, 371
host variables, 311, 414
local variables in a procedure, 376
default
columns in tables, 301
ownership discussed, 78
default DBEFileSet
dropping, 395
setting, 524
default partition
in START DBE NEW, 555
in START DBE NEWLOG, 563
deferred error checking
constraint (SET CONSTRAINTS), 521
explained, 144
referential constraint, 144
defining objects
authorization groups, 332
DBEFile, 327
DBEFileSets, 330
DBEnvironments, 555
default columns in tables, 301
tables, 354
TempSpace, 365
views, 367
definition
of a column, 355
of audit functionality, 91
procedure cursor, 151
select cursor, 151
wrapperDBE, 92
definition audit element
in START DBE NEW, 555
in START DBE NEWLOG, 563
DELETE
displaying access plan, 133, 429
statement type in rules, 379, 382
syntax, 378
DELETE WHERE CURRENT
syntax, 381
deleting
all rows from tables, 578
authorization groups, 397
data, 82
DBEFiles, 393
DBEFileSets, 395
indexes, 399, 406
modules, 401
rows, 82, 378
rows using a cursor, 381
tables, 406
TempSpaces, 408
views, 406, 409
DESCRIBE
syntax, 384
describing
dynamic statements, 384
DISABLE AUDIT LOGGING
syntax, 389

DISABLE RULES
  syntax, 390
  using, 163
disabling
  audit logging, 389
DISCONNECT
  syntax, 391
DISCONNECT CURRENT
  no current connection following, 104
disconnecting
  from DBEnvironments, 103
displaying
  access plan, 133, 429
DISTINCT
  in SELECT, 499, 505
DML ATOMICITY
  setting in SET TRANSACTION, 542
DML atomicity
  setting, 526
DO
  in procedures, 597
DROP DBEFILE
  syntax, 393
DROP DBEFILESET
  syntax, 395
DROP GROUP
  syntax, 397
DROP INDEX
  syntax, 399
DROP MODULE
  syntax, 401
DROP PARTITION
  syntax, 403
DROP PROCEDURE
  syntax, 404
DROP RULE
  syntax, 405
DROP TABLE
  syntax, 406
DROP TEMPSPACE
  syntax, 408
DROP VIEW
  syntax, 409
dropping
  authorization groups, 397
  constraint, 301
  DBEFiles, 393, 480
  DBEFileSets, 395
  indexes, 399, 406
  modules, 401
  partition, 403
  procedures, 404
  rules, 405
  tables, 406
  TempSpaces, 408
  views, 406, 409
dual logging, 555
dynamic parameters
  example of usage, 232
  example with INSERT, 456
  in DECLARE CURSOR specifying, 373
  in EXECUTE PROCEDURE, 421
  syntax, 452
dynamic preprocessing
  defined, 93
  DESCRIBE, 384
  EXECUTE, 415
  EXECUTE IMMEDIATE, 420
  EXECUTE PROCEDURE, 421
  PREPARE, 466

E

ELSE
  in procedures, 442
ELSEIF
  in procedures, 442
embedding SQL statements
  explained, 93
ENABLE AUDIT LOGGING
  syntax, 411
ENABLE RULES
  syntax, 413
  using, 163
enabling
  audit logging, 411
END DECLARE SECTION
  syntax, 414
ENDIF
  in procedures, 442
ENDWHILE
  in procedures, 597
error checking
  explained, 56
  setting atomicity, 526
  transaction and statement level constraints, 521
  using constraints, 144
error handling
  4008, 4009, or -14024 or greater, 527
  built-in variables, 471
  error number and text, 474
  in procedures invoked by rules, 161
  in procedures not invoked by rules, 154
  RAISE ERROR, 474
exclusive lock
  defined, 179
EXCLUSIVE mode
  in LOCK TABLE, 460
EXECUTE
  syntax, 415
EXECUTE authority
  granting, 437
EXECUTE IMMEDIATE
  statements that cannot be used with, 420
  syntax, 420
EXECUTE PROCEDURE
  in ISQL, 153
  syntax, 421
  using, 147
executing
  a procedure, 421
  dynamic statements, 415, 420
EXISTS predicate
  explained, 124
  syntax, 267
explicit locking, 460
expression
  and null values, 232
  defined, 227
  in BETWEEN predicate, 264
  in COMPARISON predicate, 265
  in EXISTS predicate, 267
  in IN predicate, 268
  in LIKE predicate, 272
  in NULL predicates, 275
  order of evaluation of elements in, 232
  overflow, 232
  syntax, 228
  truncation, 232
  type conversion, 232
  underflow, 232
  use, 227
  use of parentheses, 232
  USER expression value, 269
extended characters
  comparison prediate, 266

F

FETCH
  syntax, 424
fetching rows, 424
file names
  explained, 205
file space management
  for tables and indexes, 63
FILL option
  setting in BEGIN WORK, 312
fixed-length strings
  defined, 208
FLOAT
  conversions rules, 504

defined, 208
storage requirements, 210
FORCE
VALIDATE parameter, 592
FOREIGN KEY
in CREATE TABLE, 354
formal parameter
using in procedures, 149
Fortran preprocessor
defined, 42
free log space
checkpoint host variable, 316
FROM
in simple queries, 111
fully qualified name, 204

G

generating
log comment, 462
GENPLAN
explained, 133
syntax, 429
with SYSTEM and CATALOG views, 433, 435
GRANT
MONITOR authority, 199
syntax, 436
WITH GRANT OPTION explained, 76
GRANT ON DBEFILESET
syntax, 439
grantable privileges
explained, 76
revoking, 77
grants
automatic, 354
explicit (GRANT), 437
grantable privileges, 76
implicit (CREATE TABLE), 354
issuing, 436
issuing for DBEFileSet, 439
revoking, 489
which authorities can be granted, 76
who can issue them, 76
granularity
of locking, 179
GROUP BY
in simple queries, 111
group names
in ADD TO GROUP, 295
in CREATE GROUP, 332
in REMOVE FROM GROUP, 482
in REVOKE, 489, 491
in TRANSFER OWNERSHIP, 576
rules for, 201
grouping rows
in SELECT, 511
groups
adding members to, 295
creating, 332
dropping, 397
in query result, 511
removing members from, 482

H

hash
in CREATE TABLE, 354
specifying with SETOPT, 531
HAVING
in SELECT, 507
in simple queries, 111
host variables
declaration of, 311, 414
differences from local variables, 150
free log space, 316
in an expression, 228
in CONNECT, 325
in EXECUTE IMMEDIATE, 420
in FETCH, 424
in INSERT, 446, 456
in LIKE predicates, 272
in PREPARE, 468
in ROLLBACK WORK, 495
in SAVEPOINT, 497
in SET CONNECTION, 519
in SQLEXPLAIN, 550
in UPDATE, 580
in UPDATE WHERE CURRENT, 587
input, 94
naming rules, 205
output, 94
procedure parameter, 316
procedure value, 316
use of, 94
HPODBSS
reserved owner name, 206
HPRDBSS
owner of system tables, 206
hyphen
as comment initiator, 53

I

IF
in procedures, 442
syntax, 442
IN predicate
explained, 123
syntax, 268
USER expression value, 269
INCLUDE
syntax, 444
incrementing
DBEFile size, 327
index
allocating storage for, 360
creating, 71, 334
defined, 71
dropping, 393, 399, 406
duplicate keys, 335
locking explained, 181
name in CREATE INDEX, 334
name in DROP INDEX, 399
name rules for, 201
null values in, 335
number of keys in, 334
order of entries, 335
restrictions in using, 71
specifying with SETOPT, 531
uses for, 71
INDEX DBEFiles, 327
indicator variables
example in predicates, 270
in expressions, 228, 232
use of, 94
INNER
in SELECT, 510
inner join
defined, 116, 126
syntax, 509
INSERT
statement type in rules, 451, 454, 455
syntax, 445, 453
use of, 81
inserting
rows in a table, 81
rows in INSERT, 445
values in constraint columns, 143
instance of partition, 91
INTEGER
conversions rules, 504
defined, 208
storage requirements, 210
integrity constraint
defined, 68, 137
example, 141
intention
exclusive lock, 179
share lock, 179
interactive database access
defined, 42
INTERVAL
conversions rules, 504
defined, 209

operations with values, 217
storage requirements, 210
values in arithmetic expressions, 218
IS lock
explained, 179
isolation level
defined, 185
setting in BEGIN WORK, 312
setting in SET TRANSACTION, 542
ISQL
defined, 42
EXECUTE PROCEDURE in, 153
using to issue statements, 87
IX lock
explained, 179

J

join
algorithm specified by SETOPT, 531
asymmetric, 127
in complex queries, 116
inner and outer, 509
natural, 512
nested loop, 531
not using explicit join syntax, 514
outer join, 130
sort merge, 531
symmetric, 127
three or more tables, 513
JOIN ON
in SELECT, 510
JOIN USING
in SELECT, 511, 513
joining tables
in SELECT, 511
in simple queries, 112

K

keys
index, 335

L

Labeled Statement, 458
LANG = clause
for columns and tables, 67
in ALTER TABLE, 301
in CREATE TABLE, 67, 354
in START DBE NEW, 60, 555, 556
LANG variable
setting and resetting, 58
language
current language, 58
DBEnvironment and current, 60
native language support, 57
setting and resetting, 58
LEFT JOIN
in SELECT, 513
LEFT OUTER JOIN
in SELECT, 510
left outer join
defined, 127
LIKE predicate
in search condition, 262
syntax, 272
local variable
and DECLARE %%Variable%%, 376
differences from host variables, 150
naming rules, 205
using in procedures, 150
LOCK TABLE
syntax, 460
locking
automatic, 181, 354
concurrency control, 167
deadlocks, 195
exclusive mode, 182, 460
explicit, 181, 460
granularity, 179
implicit, 181
in COMMIT WORK, 323
in LOCK TABLE, 460
isolation levels, 185
levels of, 181
mode, 64, 181, 182
mode associated authorities, 64
mode types of, 65
objects locked, 181
overriding automatic locking, 460
page and table compared, 177
PUBLIC, 183
PUBLICREAD, 183
release, 188
released in deadlock, 195
released in STOP DBE, 571
releasing in ROLLBACK WORK, 495
row and page compared, 177
SET USER TIMEOUT, 548
share, 179, 182, 460
share update mode, 460
locks
exclusive (X), 179
in RELEASE, 479
intention exclusive (IX), 179
intention share (IS), 179
share (S), 179
share intention exclusive (SIX), 179
waits and timeout, 194
log buffer pages
in START DBE, 552
in START DBE NEW, 555
in START DBE NEWLOG, 563
log buffers
flushing, 316
LOG COMMENT
syntax, 462
log comment
generating, 462
log file
creating new, 563
defined, 47
increasing or decreasing space, 567
orphaned, in wrapperDBE, 92
log file names
assigning, 560
in START DBE NEWLOG, 563
rules for, 201
logging
audit, 91, 555, 563
dual, 555, 563
rollback, 495
row level DML atomicity, 144
logical operators, 262
LONG
I/O string syntax, 222
LONG BINARY
defined, 208
storage requirements, 210
long column
authorization in ALTER TABLE, 306, 307
authorization in CREATE TABLE, 362
LONG data type
restricted from search condition, 263
syntax, defining column, 221
LONG VARBINARY
defined, 208
storage requirements, 210

M

maximum
columns, 357
columns for a view, 367
columns in a query, 512
concurrent transactions, 553, 556, 564

hash key size, 361
host variable names, 416
host variables, 465
items for DISTINCT option, 505
length of index key, 361
log file size, 567
number of partitions, 555, 557, 565
PrimaryPages, 356
tables per query, 506
TempSpace, 365
timeout, 553, 564, 568
maxpartitions
in START DBE NEW, 555
in START DBE NEWLOG, 563
MaxTransactions, 553
message
buffer in procedures, 155
catalog, 550
error number, 474
for PRINT statements, 472
number 5000, 472
MICROFOCUS
COBOL preprocessor, 42
minimum
PrimaryPages, 356
mixed DBEFiles, 327
module names
in DROP MODULE, 401
in PREPARE, 466
rules for, 201
modules
access plan for validation, 532
created by preprocessor, 93
dropping, 401
effect of DBEnvironment changes on, 94
extracting, 532
owner of, 468
validating, 592
MONITOR
authority, 199
multiple
connections example, 97
connections using, 95
DBEnvironments, 95
multitable operations
SELECT, 499
using, 112
multi-transaction mode
explained, 100
undetectable deadlocks, 97
use with one DBE, 101
multitransaction mode
in SET MULTITRANSACTION, 529
multiuser mode
defined, 48
in a DBE session, 62

N

names
basic, 201
used in ALLBASE/SQL, 201
naming
database objects, 201
DBEConfile, 205
DBEnvironment, 205
DBEUserID, 203
host variables, 205
owners, 203
system files, 205
with native language objects, 203
NATIVE CHAR
conversions rules, 504
native language
ALLBASE/SQL object names, 203
character data, 226
current language, 57
defaults, 57
in columns and tables, 67
in creating a DBEnvironment, 60
setting and resetting, 58
support overview, 58
tables in ALTER TABLE, 301
tables in CREATE TABLE, 354
NATIVE VARCHAR
conversions rules, 504
NATURAL
in SELECT, 510
natural inner join
defined, 116, 126
NATURAL JOIN
in SELECT, 512, 513
n-computer
and double quotes, 67
defined, 57
NOT NULL
in CREATE TABLE, 360
in defining a column, 66
NULL predicate
in search condition, 262
syntax, 275
null values
and altering tables, 301
and FETCH, 424
and INSERT, 446
and UPDATE, 580
and UPDATE WHERE CURRENT, 587
as index keys, 335
behavior in Cartesian product, 514
behavior in joins, 514
defined, 215
in an expression, 232
in Cartesian product, 114
in joins, 114
in search conditions, 263

O

objects
defined, 48
native language names in, 203
owner of, 77
OPEN
syntax, 464
operators
arithmetic, 228
comparison, 265
logical, 262
optimizer
displaying access plan, 133, 429
modifying plan with SETOPT, 531
ORDER BY
in simple queries, 111
specifying result columns, 512
order of evaluation
of elements in an expression, 232
orphaned log files
wrapperDBE, 92
outer join
defined, 126
syntax, 509
using UNION operator, 130
overflow
in expression, 232
of data, 212
owner
authorization group, 44
changing, 576
class, 44
how to become one, 77
individuals, 44
of modules, 468
of system section tables, 206
of system tables, 206
privileges, 79
use of name, 77
who can be one, 77
owner names
origin of, 77
rules governing, 203
specification of, 45
use of, 77
ownership

and dropping authorization groups, 397
creating objects, 77
how it is assigned, 77
of objects, 77
transferring, 77, 576

P

page buffers
flushing, 316
page level locking, 177
pages
deadlocks, 195
in DBEFiles, 327
in TempSpaces, 365
PARALLEL FILL option
setting in BEGIN WORK, 312
parameter
in procedures, 149
naming rules, 205
using in procedures, 146
partition
creating, 91, 337
dropping, 403
instance, 91
setting with ALTER TABLE, 301
understanding, 91
Pascal preprocessor
defined, 42
pattern matching
in LIKE predicate, 272
performance
in procedures and rules, 165
precision
defined, 208
in DECIMAL operations, 216
predicates
BETWEEN, 264
COMPARISON, 265
compatible data types in, 263
definition of, 262
EXISTS predicate, 124, 267
IN predicate, 123, 268
in search condition, 262
LIKE predicate, 272
NULL predicates, 275
null values in, 263
order of evaluation of, 263
quantified predicate, 120, 278
PREPARE
statements that cannot be prepared, 468
syntax, 466
preparing statements
interactively and programmatically, 466
preprocessor
defined, 42
tasks, 93
preserving authorization
and DROP MODULE, 401
primary
in an expression, 228
pages, 356
PRIMARY KEY
in ALTER TABLE, 301
in CREATE TABLE, 354
PRINT
in procedures, 471
priority
setting in SET TRANSACTION, 542
private
locking, 183
PRIVATE tables
in CREATE TABLE, 354
locking mode, 64
privilege
defined, 45
grantable, 76
procedure
and transaction management, 147
BEGIN, 309
built-in variables in, 151
caller recommended practices, 156
checkpoint host variable, 316
coding practices, 156
comments within, 156
control flow statements in, 149
control language, 442, 597
creating, 146, 339
defined, 137
executing, 147
explained, 145
local variables in, 150
parameters in, 149
PRINT statement, 155, 471
queries in, 151
RAISE ERROR, 474
recommended practices, 156
result set, 342
RETURN, 487
rule and non-rule invocation, 163
runtime errors, 155
SELECT in, 152
specifying in DECLARE CURSOR, 373
using DECLARE %%Variable%% in, 376
using with rules, 159
validating, 592
with single format multiple row sets, 342
procedure cursor
defined, 151, 342, 372
in ISQL, 153
parameters in, 149
query types, 151
procedure names
in GRANT, 437
procedures and rules
chains of, 159
using, 145, 157
programmatic database access
defined, 42
programs
effect of DBEnvironment changes on, 94
PUBLIC
special name, 206
PUBLIC tables
in CREATE TABLE, 354
in GRANT, 437
locking mode, 64
PUBLICREAD tables
in CREATE TABLE, 354
locking mode, 64
PUBLICROW tables
in CREATE TABLE, 354
locking mode, 65
purging DBEFiles
using DROP DBEFILE, 393

Q

quantified predicate
explained, 120
syntax, 278
queries
available with a procedure cursor, 151
defined, 49
displaying access plan, 133, 429
in procedures, 151
range of complex types, 116
simple types, 112
updatable, 136
query
block in SELECT, 499
blocks in expression, 116
complex, 116
expression, 116
expression in SELECT, 499
file space used, 511
modifying access plan, 531
processor defined, 42
result defined, 110
results, 499

syntax (SELECT), 499

R

RAISE ERROR
in procedures, 155
in procedures invoked by rules, 162
syntax, 474
RC isolation level
explained, 175
in SET SESSION, 537
in SET TRANSACTION, 543
read committed (RC)
explained, 175
read committed isolation level
in BEGIN WORK, 312
in SET SESSION, 537
in SET TRANSACTION, 542
read uncommitted (RU)
explained, 176
read uncommitted isolation level
in BEGIN WORK, 312
in SET SESSION, 537
in SET TRANSACTION, 543
REAL
conversions rules, 504
data type defined, 208
storage requirements, 210
recovery
rollback, 495
referenced table
defined, 138
referencing table
defined, 139
referential constraint
deferred error checking, 144
defined, 138
in ALTER TABLE, 301
in CREATE TABLE, 354
revoking, 489
slowing TRUNCATE TABLE, 578
referential integrity
using constraints, 68, 137
REFETCH
syntax, 476
relation
defined, 44
relational database
defined, 44
RELEASE
syntax, 479
release
DBE session, 479
in COMMIT WORK, 323
remote connections
establishment of, 95
REMOVE DBEFile
syntax, 480
REMOVE FROM GROUP
syntax, 482
removing DBEFiles, 480
RENAME COLUMN
syntax, 484
RENAME TABLE
syntax, 485
renaming
columns, 484
tables, 485
repeatable read (RR)
explained, 174
repeatable read isolation level
in BEGIN WORK, 312
in SET SESSION, 536
in SET TRANSACTION, 542
RESET
syntax, 486
resetting ALLBASE/SQL system data, 486
result
table defined, 49
result columns
in SELECT, 512
retrieving
data, 109
rows, 109, 499
RETURN
syntax, 487
return status
in DECLARE CURSOR specifying, 373
revalidating rows, 476
REVOKE
and CASCADE, 77
and grantable privilege, 77
syntax, 489
revoking
authorities using REVOKE, 489
grants using CASCADE, 77
RIGHT OUTER JOIN
in SELECT, 513
right outer join
defined, 127
roll forward
wrapperDBE, 92
rollback
recovery, 495
ROLLBACK WORK, 495
ROLLBACK WORK
in a procedure, 147
syntax, 495
row level
DML atomicity, 526
DML atomicity logging, 144
locking, 177
rows
defined, 44
fetching, 424
inserting, 445
joining, 511
selecting, 499
RR isolation level
explained, 174
in SET TRANSACTION, 543
RU isolation level
explained, 176
in SET SESSION, 537
in SET TRANSACTION, 543
rule
and transaction management, 163
creating, 158, 346
defined, 137
differences from integrity constraints, 166
enabling and disabling, 163
explained, 157
techniques for using procedures with, 159
rules and procedures
chains of, 159
CREATE PROCEDURE statement, 339
CREATE RULE statement, 346
DISABLE RULES statement, 390
DROP PROCEDURE statement, 404
DROP RULE statement, 405
ENABLE RULES statement, 413
TRUNCATE TABLE statement, 578
using, 145, 157
RUN authority
granting, 437
purpose, 94
run block
in START DBE, 552
in START DBE NEWLOG, 563
runtime control block pages
in START DBE, 552
in START DBE NEW, 555
in START DBE NEWLOG, 563
runtime errors
in a procedure, 155

S

S (SHARE) locks, 460
S lock
explained, 179

SAVEPOINT
in a procedure, 147
setting, 497
syntax, 497
using, 495
scale
defined, 208
in DECIMAL operations, 216
scoping
transaction and session attributes, 87
search condition
compatible predicates, 263
defined, 111
definition, 261
in complex queries, 116
in DELETE, 378
subquery in, 120
syntax, 262
type conversion in, 263
use for, 261
value extensions in, 263
SearchCondition
in CREATE TABLE, 359
section
authorization in PREPARE, 468
defined, 93
invalidation by TRUNCATE TABLE, 578
invalidation through procedures, 164
semi-permanent, 466
validating, 592
section audit element
in START DBE NEW, 555
in START DBE NEWLOG, 563
SECTIONSPACE
authorization in CREATE RULE, 349
security
of database, 75
SELECT
displaying access plan, 133, 429
in CREATE VIEW, 367
in DECLARE CURSOR, 371
in procedures, 152
syntax, 499, 505
use of, 109
with cursor in procedures, 153
select cursor
authorization in CREATE PROCEDURE, 373
defined, 151, 342, 372
within a procedure, 151
select list
defined, 110, 506
selecting data
discussed, 109
grouping rows, 511
maximum columns, 507
SELECT, 499
unique rows, 505
semi-permanent section
creating with PREPARE, 466
owner, 206
SEMIPREM owner, 206
serializable isolation level
in SET SESSION, 537
in SET TRANSACTION, 543
session
DBE, 62
SET CONNECTION
syntax, 519
set constraint type statement
explained, 144
SET CONSTRAINTS
syntax, 521
SET DEFAULT DBEFILESET
syntax, 524
SET DML ATOMICITY
syntax, 526
SET MULTITRANSACTION
explained, 99
syntax, 529
SET PRINTRULES
syntax, 534, 536
SET PRINTRULES ON
to trace rule chaining, 164
SET SESSION
CS isolation level, 537
cursor stability isolation level, 536
RC isolation level, 537
read committed isolation level, 537
read uncommitted isolation level, 537
repeatable read isolation level, 536
RU isolation level, 537
serializable isolation level, 537
SET TRANSACTION
CS isolation level, 543
cursor stability isolation level, 542
RC isolation level, 543
read committed isolation level, 542
read uncommitted isolation level, 543
repeatable read isolation level, 542
RR isolation level, 543
RU isolation level, 543
serializable isolation level, 543
syntax, 542
SET USER TIMEOUT
syntax, 548
SETOPT
syntax, 531
validating modules, 592
setting
constraint checking with SET TRANSACTION, 542
constraints to deferred, 521
current connection, 96
default DBEFileSet, 524
DML atomicity, 526
multiple-transaction mode, 100
options with BEGIN WORK, 312
savepoints, 497
the current connection, 519
timeout values, 97
transaction mode, 99
setting DML ATOMICITY
with SET TRANSACTION, 542
setting DML atomicity, 144
setting transaction attributes
with SET TRANSACTION, 542
share
intention exclusive lock, 179
lock defined, 179
SHARE mode
in LOCK TABLE, 460
share mode
locking, 182
SHARE UPDATE mode
in LOCK TABLE, 460
simple names
defined, 203
simultaneous transactions
with BEGIN WORK, 313
single-transaction mode
explained, 99
in SET MULTITRANSACTION, 529
single-user mode
defined, 48
in a DBE session, 62
SIX (SHARE INTENT EXCLUSIVE) locks, 460
SIX lock
explained, 179
SMALLINT
conversions rules, 504
defined, 208
storage requirements, 210
SOME
in quantified predicate, 278
sorting

using TempSpace, 63
space management
for tables and indexes, 63
in CREATE DBEFILESET, 330
special
authorities revoking, 491
names, 206
special predicates
EXISTS predicate, 267
IN predicate, 268
quantified predicate, 278
SQL
defined, 41
language structure, 52
naming rules, 201
usage, 41, 60
SQL statement
ADD DBEFile, 293
ADD TO GROUP, 295
ADVANCE, 297
ALTER DBEFILE, 299
ALTER TABLE, 301
BEGIN, 309
BEGIN ARCHIVE, 310
BEGIN DECLARE SECTION, 311
BEGIN WORK, 312
BULK FETCH, 424
BULK INSERT, 445
BULK SELECT, 499
CHECKPOINT, 316
CLOSE, 319
COMMIT ARCHIVE, 322
COMMIT WORK, 323
CONNECT, 325
CREATE DBEFILE, 327
CREATE DBEFILESET, 330
CREATE GROUP, 332
CREATE INDEX, 334
CREATE PARTITION, 337
CREATE PROCEDURE, 339
CREATE RULE, 346
CREATE SCHEMA, 351
CREATE TABLE, 354
CREATE TEMPSPACE, 365
CREATE VIEW, 367
DECLARE %%Variable%%, 376
DECLARE CURSOR, 371
DELETE, 378
DELETE WHERE CURRENT, 381
DESCRIBE, 384
DISABLE AUDIT LOGGING, 389
DISABLE RULES, 390
DISCONNECT, 391
DROP DBEFILE, 393
DROP DBEFILESET, 395
DROP GROUP, 397
DROP INDEX, 399
DROP MODULE, 401
DROP PARTITION, 403
DROP PROCEDURE, 404
DROP RULE, 405
DROP TABLE, 406
DROP TEMPSPACE, 408
DROP VIEW, 409
ENABLE AUDIT LOGGING, 411
ENABLE RULES, 413
END DECLARE SECTION, 414
EXECUTE, 415
EXECUTE IMMEDIATE, 420
EXECUTE PROCEDURE, 421
FETCH, 424
GENPLAN, 429
GRANT, 436
GRANT ON DBEFILESET, 439
IF, 442
INCLUDE, 444
INSERT, 445
Labeled Statement, 458
length, 283
LOCK TABLE, 460
LOG COMMENT, 462
OPEN, 464
PREPARE, 466
PRINT, 471
RAISE ERROR, 474
REFETCH, 476
RELEASE, 479
REMOVE DBEFILE, 480
REMOVE FROM GROUP, 482
RENAME COLUMN, 484
RENAME TABLE, 485
RESET, 486
RETURN, 487
REVOKE, 489
ROLLBACK WORK, 495
SAVEPOINT, 497
SELECT, 499, 505
SET CONNECTION, 519
SET CONSTRAINTS, 521
SET DEFAULT DBEFILESET, 524
SET DML ATOMICITY, 526
SET MULTITRANSACTION, 529
SET PRINTRULES, 534, 536
SET TRANSACTION, 542
SET USER TIMEOUT, 548
SETOPT, 531
SQLEXPLAIN, 550
START DBE, 552
START DBE NEW, 555
START DBE NEWLOG, 563
STOP DBE, 571
summary table, 283
TERMINATE QUERY, 572
TERMINATE TRANSACTION, 573
TERMINATE USER, 574
TRANSFER OWNERSHIP, 576
TRUNCATE TABLE, 578
UPDATE, 580
UPDATE STATISTICS, 585
UPDATE WHERE CURRENT, 587
VALIDATE, 592
WHENEVER, 595
WHILE, 597
SQL statements
categories, 54
within procedures, 148
SQLAudit
auditing transactions, 92
defined, 43
SQLCA
INCLUDE SQLCA, 444
used with SQLEXPLAIN, 550
used with WHENEVER, 595
SQLCheck
defined, 43
SQLDA
INCLUDE SQLDA, 444
used with FETCH, 424
SQLEXPLAIN
on returning from procedures, 156
syntax, 550
SQLGEN
defined, 43
SQLMigrate
defined, 43
SQLMON
authority, 199
defined, 43
grant authority to run, 438
monitoring locking, 199
monitoring transactions, 90
SQLUtil
defined, 43
setting transaction limits, 89
wrapdbe command, 92
SQLVer
defined, 43
START DBE
syntax, 552
START DBE NEW
syntax, 555
START DBE NEWLOG

syntax, 563
starting a DBE session
using CONNECT, 325
using START DBE, 552
startup parameters
defined in START DBE NEW, 560
in START DBE NEW, 60
statement level
constraint error checking, 521
DML atomicity, 526
error enforcement explained, 144
STOP DBE
syntax, 571
stopping
ALLBASE/SQL using STOP DBE, 571
session using DISCONNECT, 391
storage allocation
defined, 45
storage audit element
in START DBE NEW, 555
in START DBE NEWLOG, 563
Storage Manager
defined, 42
storage requirements
for specific data types, 210
stored modules
and DROP MODULE, 401
STOREDSECT
owner of system section tables, 206
Structured Query Language
defined, 41
subquery
as part of a predicate, 116
correlated, 126
defined, 120
in a quantified predicate, 122
in an EXISTS predicate, 124
in an IN predicate, 124
switching transactions
and MULTITRANSACTION, 529
symmetric outer join
and UNION, 130
defined, 127
using UNION operator, 131
SYSTEM
as owner, 77
owner of system views, 206
table locking, 181
view names in UPDATE STATISTICS, 585
system catalog
contents, 61, 105
defined, 47
system tables, 61
system views, 61
table of system views, 106
updating statistics, 585
system DBEFileset
defined, 47
system views
summary of, 106
SYSTEM.ACCOUNT
resetting, 486
SYSTEM.COLUMN
updating statistics for, 585
SYSTEM.COUNTER
resetting, 486
SYSTEM.DBEFILE
updating statistics for, 585
SYSTEM.DBEFILESET
updating statistics for, 585
SYSTEM.INDEX
updating statistics for, 585
SYSTEM.SECTION
validating modules, 594
SYSTEM.TABLE
monitoring for table size, 178
updating statistics for, 585

T

table
adding constraint to, 301
allocating storage for, 360
and check constraints, 139
changing locking, 301
creating, 354
defined, 44, 64
defining default columns in, 301, 357
deleting all rows from, 578
dropping, 393, 406
dropping constraint from, 301
explicit locking, 460
granting authorities, 436
implicit locking, 354
inserting rows into, 445
locking, 183
locking explained, 177
referenced, 138
referencing, 139
revoking authorities, 489
updating statistics, 585
TABLE DBEFiles, 327
table names
in ALTER TABLE, 301
in CREATE INDEX, 334
in CREATE TABLE, 354
in DELETE, 378
in DELETE WHERE CURRENT, 381
in DROP INDEX, 399
in DROP TABLE, 406
in GRANT, 437
in INSERT, 445
in LOCK TABLE, 460
in REVOKE, 489
in TRANSFER OWNERSHIP, 576
in TRUNCATE TABLE, 578
in UPDATE statements, 580
in UPDATE STATISTICS, 585
in UPDATE WHERE CURRENT, 587
rules for, 201
TableSpec
in SELECT, 500
TEMP
and modules not stored, 468
owner of modules, 206
temporary section
validating, 592
TempSpace
defined, 63
dropping, 408
names in CREATE TEMPSPACE, 365
names in DROP TEMPSPACE, 408
using, 365
TERMINATE QUERY
syntax, 572
TERMINATE TRANSACTION
syntax, 573
TERMINATE USER
syntax, 574
terminating
a DBE session, 62, 391, 479, 572, 574
transactions, 495, 573
TERMINATION LEVEL
setting with SET TRANSACTION, 542
THEN
in procedures, 442
TIME
conversions rules, 504
defined, 209
operations with values, 217
storage requirements, 210
values in arithmetic expression, 218
timeout
and BEGIN WORK, 314
and lock waits, 194

SET TRANSACTION statement, 542
SET USER TIMEOUT statement, 548
START DBE NEWLOG, 568
value setting, 97
values in DBECon file, 552
transaction attributes
setting in BEGIN WORK, 312
setting in SET TRANSACTION, 542
transaction level constraint checking
errors, 521
transactions
aborted, 571
and data consistency, 48
and multiple connections, 95
and recovery, 48
and timeouts, 95
automatic rollback of, 495
committing, 82
defined, 48, 82
effect of terminating, 323
implicit vs. explicit, 313
in a procedure, 147
in a procedure invoked by a rule, 163
in START DBE NEW, 556
locks released, 313
management, 82
maximum in START DBE, 552
maximum in START DBE NEWLOG, 564
mode setting, 99
priority, 195
SET USER TIMEOUT, 548
simultaneous with BEGIN WORK, 313
statements that must be in the same, 313
terminating, 313, 323, 495
TRANSFER OWNERSHIP
syntax, 576
transferring ownership, 576
when dropping authorization group, 397
TRUNCATE TABLE
syntax, 578
truncation
and native language data, 226
in expressions, 232
of data, 212
tuple
defined, 44
type conversion
and overflow, 212
and truncation, 212
defined, 213
in expressions, 232
in search conditions, 263

U

underflow
defined, 212
in expression, 232
undetectable deadlock
in multi-transaction mode, 97
UNION
and outer join, 130
character constants with, 119
in outer joins, 130
in queries, 117
in SELECT, 503
UNION ALL form, 117
unique
indexes, 334
rows, 505
unique constraint
defined, 138
in ALTER TABLE, 301
in CREATE TABLE, 354
updatability
of cursors, 372
rules, 136
UPDATE
displaying access plan, 133, 429
statement type in rules, 581, 588
syntax, 580
UPDATE STATISTICS
syntax, 585
UPDATE WHERE CURRENT
syntax, 587
updating
data, 82
system catalog statistics, 585
USER expression value
expressions, 269
IN predicate, 269
user mode
defined, 48
in CONNECT, 325
user table locking
explained, 181
user timeout value
in SET TRANSACTION, 542
in START DBE, 552
in START DBE NEW, 555
in START DBE NEWLOG, 563
using ALLBASE/SQL
ad hoc queries, 51
application programming, 51
database administration, 51
using SQL
summarized, 60

V

VALIDATE
syntax, 592
validity checking
of data, 68
of databases, 137
value
comparisons, 211
extensions in search conditions, 263
VARBINARY
conversions rules, 504
long data type defined, 208
storage requirements, 210
VARCHAR
conversions rules, 504
defined, 208
storage requirements, 210
variable-length strings
defined, 208
variables
BEGIN DECLARE SECTION, 311
END DECLARE SECTION, 414
indicator, 94
input, 94
output, 94
verb
defined, 52
view names
in CREATE VIEW, 367
in DELETE, 378
in DELETE WHERE CURRENT, 381
in DROP TABLE, 406
in DROP VIEW, 409
in INSERT, 445
in REVOKE, 489
in TRANSFER OWNERSHIP, 576
in UPDATE, 580
in UPDATE WHERE CURRENT, 587
rules for, 201
views
and check constraints, 141
base tables, 67
creating, 67, 367
defined, 44, 67
dropping, 406, 409
granting authorities, 436
inserting data, 446
restrictions in using, 67, 368
revoking authorities, 489

updatable, 136
uses for, 67
WITH CHECK OPTION, 141

W

WHENEVER
in procedures, 154
syntax, 595
WHERE
and joins, 514
in SELECT, 506
in simple queries, 111
WHILE
syntax, 597
WITH CHECK OPTION
in CREATE VIEW, 368
view, 141
WITH GRANT OPTION
explained, 76
syntax in GRANT, 436
wrapdbe command
wrapperDBE, 92
wrapperDBE
archive logging, 92
audit information, 92
definition, 92
roll forward, 92
wrapdbe command, 92
writer of a procedure
recommended practices, 156

X

X (EXCLUSIVE) locks, 460
explained, 179

Index