Administration Guide

PUBLIC

SAP Adaptive Server Enterprise 16.0
Document Version: 1.1 – March 15 2015

Transact-SQL Users Guide

Content

1 SQL Building Blocks. 20

1.1 Tables, Columns, and Rows. 20

1.2 Queries, Data Modification, and Commands. 21

1.3 Relational Operations. 21

1.4 Compiled Objects. 22

Save or Restore Source Text. 22

Verify and Encrypt Source Text. 23

Replacing Object Definitions. 23

1.5 Compliance to ANSI Standards. 24

Federal Information Processing Standards (FIPS) Flagger. 25

Chained Transactions and Isolation Levels. 25

Identifier Compliance to ANSI Standards. 25

SQL Standard-Style Comments. 26

Right Truncation of Character Strings. 26

Permissions Required for update and delete Statements. 26

Arithmetic Errors. 26

Synonymous Keywords. 27

Treatment of Nulls. 27

1.6 Data and Language Characters. 27

Naming Convention Identifiers. 29

1.7 Expressions in SAP ASE. 35

Arithmetic Operators. 35

Bitwise Operators. 36

The String Concatenation Operator. 38

The Comparison Operators. 38

Nonstandard Operators. 39

Character Expression Comparisons. 39

Empty Strings. 39

Quotation Marks 40

Relational and Logical Expressions. 41

1.8 Transact-SQL Extensions. 42

compute Clause. 42

Control-of-Flow Language. 42

Stored Procedures. 43

Extended Stored Procedures. 44

Triggers. 44

Defaults and Rules. . . . 45
Error Handling and set Options. . . . 45
Additional SAP ASE Extensions to SQL. . . . 45
1.9 SAP ASE Login Accounts. . . . 47
1.10 isql Utility. . . . 47
Default Databases. . . . 48
Network-Based Security Services with isql. . . . 48
1.11 Displaying SQL Text. . . . 49

2 Databases and Tables. . . . 50
2.1 Databases. . . . 50
2.2 Create a User Database . . . 51
The on Clause. . . . 52
The log on Clause. . . . 53
for load Option. . . . 53
2.3 Choose a Database. . . . 53
2.4 Permissions Within Databases. . . . 54
2.5 Initialize Databases Asynchronously. . . . 55
Determine If There is Space to be Initialized. . . . 55
Restrictions for Initializing Databases Asynchronously . . . 56
2.6 Drop Databases. . . . 57
2.7 Change the Database Size. . . . 57
2.8 Enforce Data Integrity in Databases. . . . 58
2.9 quiesce database Command. . . . 59
2.10 Tables. . . . 59
2.11 Designing and Creating a Table. . . . 61
Table Names. . . . 62
Create the User-Defined Datatypes. . . . 62
Choose Columns That Accept Null Values. . . . 63
Sample Table Design Sketch. . . . 63
2.12 Create Tables in Different Databases. . . . 64
2.13 Create New Tables from Query Results: select into. . . . 65
Check for Errors. . . . 68
2.14 Temporary Tables Usage. . . . 68
Unique Temporary Table Names . . . 69
Manipulate Temporary Tables in Stored Procedures. . . . 69
General Rules for Temporary Tables. . . . 70
2.15 Deferred Table Creation. . . . 71
Deferred Table Creation at the Database Level. . . . 71
Create Deferred Tables. . . . 71
Explicitly Materialize Deferred Tables. . . . 72
Identify Deferred Tables. . . . 72

Roll Back for Deferred Tables. . . . 73
Command Behavior in Deferred Tables. . . . 73
2.16 IDENTITY Columns Usage. . . . 74
Create IDENTITY Columns with User-Defined Datatypes. . . . 75
Reference IDENTITY Columns. . . . 75
Refer to IDENTITY Columns with syb_identity. . . . 76
Automatically Create “hidden” IDENTITY Columns. . . . 76
Using select into with IDENTITY Columns. . . . 77
2.17 Allow Null Values in a Column. . . . 79
Constraints and Rules Used with Null Values. . . . 80
Defaults and Null Values. . . . 80
Nulls Require Variable-Length Datatypes. . . . 81
text, unitext, and image Columns. . . . 82
2.18 Alter Existing Tables. . . . 82
Objects Using select * Do Not List Changes to Table. . . . 83
Use alter table on Remote Tables. . . . 83
Add Columns. . . . 84
Drop Columns. . . . 85
Modify Columns. . . . 88
Add IDENTITY Columns. . . . 92
Drop IDENTITY Columns. . . . 92
Modify IDENTITY Columns. . . . 93
Data Copying. . . . 93
Modifying Locking Schemes and Table Schema. . . . 95
Add, Drop, or Modify Columns with User-Defined Datatypes. . . . 95
Errors and Warnings from alter table. . . . 96
Rename Tables and Other Objects. . . . 98
2.19 Drop Tables. . . . 99
2.20 Manage Identity Gaps in Tables. . . . 100
Parameters for Controlling Identity Gaps. . . . 100
Comparison of identity burning set factor and identity_gap. . . . 101
Set the Table-Specific Identity Gap. . . . 102
Change the Table-Specific Identity Gap. . . . 103
Display Table-Specific Identity Gap Information. . . . 103
Gaps from Other Causes. . . . 104
IDENTITY Column Maximum Value. . . . 105
2.21 Define Integrity Constraints for Tables. . . . 105
Table and Column Level Constraints. . . . 106
Create Error Messages for Constraints. . . . 107
Check Constraints. . . . 107
Default Column Values. . . . 108

unique and primary key Constraints. . . . 109
Referential Integrity Constraints. . . . 110
2.22 Computed Columns. . . . 114
Computed Columns Usage. . . . 115
Indexes on Computed Columns. . . . 118
Deterministic Property. . . . 118
2.23 Retrieve Information About Databases and Tables. . . . 122
Help on Databases. . . . 123
Help on Database Objects. . . . 124

3 SQL-Derived Tables. . . . 129
3.1 SQL-Derived Tables and Optimization. . . . 130
3.2 SQL-Derived Table Syntax. . . . 130
Derived Column Lists. . . . 131
Correlated SQL-Derived Tables Are Not Supported. . . . 132
3.3 SQL-Derived Tables Usage. . . . 132
Nesting. . . . 132
Subqueries Using SQL-Derived Tables. . . . 133
Unions in Derived-Table Expressions 133
Unions in Subqueries. . . . 133
Rename Columns with SQL-Derived Tables. . . . 134
Constant Expressions. . . . 134
Aggregate Functions. . . . 135
Joins with SQL-Derived Tables. . . . 135
Create a Table From a SQL-Derived Table. . . . 136
Views with SQL-Derived Tables. . . . 136
Correlated Attributes. . . . 137

4 Partition Tables and Indexes. . . . 138
4.1 Partitioning Types. . . . 140
Range Partitioning. . . . 140
Hash Partitioning. . . . 140
List Partitioning 141
Round-Robin Partitioning. . . . 141
4.2 Partition Pruning. . . . 141
4.3 Composite Partitioning Keys. . . . 142
4.4 Indexes and Partitions. . . . 143
Global Indexes. . . . 143
Local Indexes. . . . 145
Guarantee a Unique Index. . . . 147
4.5 Create and Manage Partitions. . . . 148
Partitioning Tasks. . . . 148

Create a Range-Partitioned Table. . . . 150
Create a Hash-Partitioned Table. . . . 151
Create a List-Partitioned Table. . . . 151
Create a Round-Robin–Partitioned Table. . . . 152
Create Partitioned Indexes. . . . 152
Create a Partitioned Table From an Existing Table. . . . 153
4.6 Change Data Partitions. . . . 153
Split, Merge, and Move Partitions. . . . 154
Add Partitions to a Partitioned Table. . . . 158
Change the Partitioning Type or Key. . . . 159
Unpartition Round-Robin–Partitioned Tables. . . . 159
partition Parameter Usage. . . . 160
Change Partition-Key Columns. . . . 160
4.7 Configure Partitions. . . . 161
4.8 update, delete, and insert in Partitioned Tables. . . . 161
4.9 Update Values in Partition-Key Columns. . . . 162
4.10 Display Information About Partitions. . . . 163
Function Usage. . . . 163
4.11 Truncate a Partition. . . . 164
4.12 Using Partitions to Load Table Data. . . . 164
4.13 Update Partition Statistics. . . . 165
4.14 Improved Concurrency for Partition-Level Online Operations. . . . 165
Partition-Level Online Operation Syntax. . . . 166
Concurrency with Partition-Level Online Operations. . . . 167
Partition-Level Online Operations with Global Index. . . . 167

5 Virtually Hashed Tables. . . . 168
5.1 Structure of a Virtually Hashed Table. . . . 168
5.2 Create a Virtually Hashed Table. . . . 169
5.3 Limitations for Virtually Hashed Tables. . . . 171
5.4 Commands that Support Virtually Hashed Tables. . . . 172
5.5 Query Processor Support. . . . 172
5.6 Monitor Counter Support. . . . 173
5.7 System Procedure Support. . . . 173

6 Create Indexes on Tables. . . . 174
6.1 Guidelines for Using Indexes. . . . 175
6.2 Methods of Creating Indexes. . . . 175
6.3 Create Indexes. . . . 176
Issue create index in Parallel. . . . 177
Function-Based Indexes. . . . 179
Create Indexes Without Blocking Access to Data. . . . 179

Unique Indexes. . . . 180
IDENTITY Columns in Nonunique Indexes. . . . 181
Ascending and Descending Index-Column Values. . . . 182
Using fillfactor, max_rows_per_page, and reservepagegap. . . . 182
6.4 Indexes on Computed Columns. . . . 183
6.5 Clustered or Nonclustered Index Usage. . . . 184
Create Clustered Indexes on Segments. . . . 185
6.6 Index Options. . . . 185
ignore_dup_key Option . . . 186
ignore_dup_row and allow_dup_row . . . 186
sorted_data Option . . . 187
on segment_name Option. . . . 187
6.7 Drop Indexes. . . . 188
6.8 Identifying the Indexes on a Table. . . . 188
6.9 Update Statistics for Indexes. . . . 190

7 Datatypes. . . . 191
7.1 System-Supplied Datatypes. . . . 191
Exact Numeric Types: Integers. . . . 194
Exact Numeric Types: Decimal Numbers. . . . 195
Approximate Numeric Datatypes. . . . 195
Money Datatypes. . . . 196
Date and Time Datatypes. . . . 196
Character Datatypes. . . . 197
Binary Datatypes. . . . 201
bit Datatype. . . . 202
timestamp Datatype. . . . 203
sysname and longsysname Datatype. . . . 203
7.2 LOB Locators in Transact-SQL Statements. . . . 203
Implicitly Create a Locator. . . . 204
Explicitly Create a Locator. . . . 205
Convert the Locator Value to the LOB Value. . . . 206
Locator Scope. . . . 206
7.3 Convert Between Datatypes. . . . 207
7.4 Mixed-Mode Arithmetic and Datatype Hierarchy. . . . 207
Working with money Datatypes . . . 209
Determine Precision and Scale. . . . 209
7.5 User-Defined Datatypes. . . . 210
Length, Precision, and Scale. . . . 211
Null Type. . . . 211
Associate Rules and Defaults with User-Defined Datatypes. . . . 211
Create User-Defined Datatype with IDENTITY Property. . . . 212

Create IDENTITY Columns from User-Defined Datatypes. 212
Drop a User-Defined Datatype. 212
7.6 Datatype Entry Rules. 213
char, nchar, unichar, univarchar, varchar, nvarchar, unitext, and text. 213
Date and Time. 214
binary, varbinary, and image. 218
money and smallmoney. 218
float, real, and double precision. 219
decimal and numeric. 219
Integer Types and Their Unsigned Counterparts. 220
timestamp. 220
7.7 Get Information About Datatypes. 220

8 Queries: Selecting Data from a Table. 222
8.1 select Syntax. 222
Check for Identifiers in a select Statement. 223
8.2 Choose Columns Using the select Clause. 224
Choose all Columns Using select * 224
Choose Specific Columns. 225
Rearrange the Column Order. 225
Rename Columns in Query Results. 226
Expressions. 226
Select Text, Unitext, and Image Values. 231
select List Summary. 232
8.3 select for update. 233
Use select for update in Cursors and DML. 234
Concurrency Issues. 234
8.4 Eliminate Duplicate Query Results with Distinct. 235
8.5 Specify Tables with the from Clause. 236
8.6 Select Rows Using the where Clause. 237
Comparison Operators in where Clauses. 238
Ranges (between and not between). 239
Lists (in and not in). 240
Matching Character Strings: like. 242
“Unknown” Values: NULL. 248
Connect Conditions with Logical Operators. 253
8.7 Multiple select Items in a Nested exists Query. 255
8.8 Use a Column Alias in Nested select Statements. 256

9 Subqueries: Queries Within Other Queries. 257
9.1 Subquery Restrictions. 258
9.2 Qualify Column Names. 259

9.3 Subqueries with Correlation Names. . . . 260
9.4 Multiple Levels of Nesting. . . . 261
9.5 Using an Asterisk in Nested select Statements . . . 261
Use Table-Name Qualifiers. . . . 262
Use Nested Queries with group by. . . . 262
Usage and Examples of Asterisks in select Statements. . . . 263
9.6 Subqueries in update, delete, and insert Statements. . . . 265
9.7 Subqueries in Conditional Statements. . . . 265
9.8 Subqueries Instead of Expressions. . . . 266
9.9 Types of Subqueries. . . . 267
Expression Subqueries. . . . 267
Quantified Predicate Subqueries. . . . 270
Subqueries Used with in. . . . 274
Subqueries Used with not in. . . . 276
Subqueries Using not in with NULL. . . . 277
Subqueries Used with exists. . . . 277
Subqueries Used with not exists. . . . 279
Find Intersection and Difference with exists. . . . 280
Subqueries Using SQL Derived Tables. . . . 281
9.10 Correlated Subqueries. . . . 281
Correlated Subqueries with Correlation Names. . . . 282
Correlated Subqueries with Comparison Operators. . . . 283
Correlated Subqueries in a having Clause. . . . 284

10 Aggregates, Grouping, and Sorting. . . . 285
10.1 Aggregate Functions and Datatypes. . . . 286
10.2 count versus count (*). . . . 287
10.3 Aggregate Functions with distinct. . . . 288
10.4 Null Values and the Aggregate Functions. . . . 289
10.5 Using Statistical Aggregates. . . . 290
10.6 Organize Query Results into Groups: the group by Clause. . . . 290
group by and SQL Standards. . . . 292
Nest Groups with group by. . . . 292
Reference Other Columns in Queries Using group by. . . . 293
Expressions and group by. . . . 295
group by in Nested Aggregates . . . 296
Null Values and group by. . . . 296
where Clause and group by. . . . 297
group by and all. . . . 299
Aggregates Without group by. . . . 300
10.7 Select Groups of Data: the having Clause. . . . 301
Interactions between having, group by, and where Clauses . . . 302

having Without group by. . . . 304
10.8 Sort Query Results: the order by Clause. . . . 305
order by and group by. . . . 308
order by and group by Used with select distinct. . . . 308
10.9 Summarize Groups of Data: the compute Clause. . . . 309
Row Aggregates and compute. . . . 311
Specify More Than One Column After compute. . . . 313
Use More Than One compute Clause. . . . 313
Apply an Aggregate to More Than One Column. . . . 314
Use Different Aggregates in the Aame compute Clause. . . . 314
Generate Totals: compute Without by. . . . 315
10.10 Combine Queries: the union Operator. . . . 316
Guidelines for union Queries. . . . 317

11 Joins: Retrieve Data from Several Tables. . . . 320
11.1 Join Syntax. . . . 321
11.2 Joins and the Relational Model. . . . 321
11.3 How Joins are Structured. . . . 322
The from Clause. . . . 323
The where Clause. . . . 324
11.4 How Joins are Processed. . . . 326
11.5 Equijoins and Natural Joins. . . . 327
11.6 Joins with Additional Conditions. . . . 328
11.7 Joins Not Based on Equality. . . . 328
11.8 Self-Joins and Correlation Names. . . . 329
11.9 The Not-Equal Join. . . . 330
Not-Equal Joins and Subqueries. . . . 332
11.10 Join More Than Two Tables. . . . 333
11.11 Star Joins. . . . 334
11.12 Outer Joins. . . . 334
Inner and Outer Tables. . . . 335
Outer Join Restrictions. . . . 335
Views Used with Outer Joins. . . . 335
ANSI Inner and Outer Joins. . . . 336
ANSI outer joins. . . . 341
Transact-SQL Outer Joins. . . . 348
11.13 Relocated Joins. . . . 352
Configuring Relocated Joins. . . . 352
11.14 How Null Values Affect Joins. . . . 353
11.15 Determine Which Table Columns to Join. . . . 354

12 Managing Data. . . . 355

12.1 Referential Integrity. . . . 355
12.2 Transactions. . . . 356
12.3 Sample Databases. . . . 357
12.4 Add New Data. . . . 357
Add New Rows with Values. . . . 358
Insert Data into Specific Columns. . . . 358
Add New Rows with select. . . . 366
12.5 Create Nonmaterialized, Non-Null Columns. . . . 369
Add Nonmaterialized Columns. . . . 369
Tables That Already Have Nonmaterialized Columns. . . . 370
Nonmaterialized Column Storage. . . . 370
Alter Nonmaterialized Columns. . . . 371
Limitations for Nonmaterialized Columns 371
12.6 Change Existing Data. . . . 371
Use the set Clause with Update. . . . 372
Use the where Clause with update. . . . 373
Use the from Clause with update. . . . 373
Perform updates with joins. . . . 374
Update IDENTITY Columns. . . . 374
12.7 Change text, unitext, and image data. . . . 375
12.8 Truncate Trailing Zeros. . . . 376
12.9 Transfer Data Incrementally. . . . 379
Mark Tables for Incremental Transfer. . . . 379
Transfer Tables from a Destination File. . . . 380
Convert SAP ASE Datatypes to SAP IQ. . . . 380
Store Transfer Information. . . . 382
Exceptions and Errors. . . . 384
Sample Incremental Transfer. . . . 385
12.10 Delete Data. . . . 390
Use the from Clause with delete. . . . 390
Delete from IDENTITY Columns. . . . 391
12.11 Delete All Rows from a Table. . . . 391
truncate table Syntax. . . . 392

13 Views: Limit Access to Data. . . . 393
13.1 Advantages of Views. . . . 393
13.2 Security. . . . 394
13.3 Logical Data Independence. . . . 395
13.4 Create Views. . . . 395
create view Syntax. . . . 396
select Statement Usage with create view. . . . 397
Validate a View's Selection Criteria. . . . 401

13.5 Retrieve Data Through Views. . . . 403
View Resolution. . . . 404
Redefine Views. . . . 404
Rename Views. . . . 405
Alter or Drop Underlying Objects. . . . 406
13.6 Modify Data Through Views. . . . 406
Restrictions on Updating Views. . . . 407
13.7 Drop Views. . . . 410
13.8 Use Views as Security Mechanisms. . . . 410
13.9 Get Information About Views. . . . 411

14 Defining Defaults and Rules for Data. . . . 413
14.1 Create Defaults. . . . 413
Bind Defaults. . . . 414
Unbind Defaults. . . . 416
How Defaults Affect NULL Values. . . . 417
14.2 Drop Defaults. . . . 417
14.3 Create Rules. . . . 418
Bind Rules. . . . 419
Rules and NULL Values. . . . 421
Unbind Rules. . . . 421
14.4 Drop Rules. . . . 422
14.5 Retrieve Information About Defaults and Rules. . . . 423
14.6 Share Inline Defaults. . . . 423
Create an Inline Shared Default. . . . 423
Unbind a Shared Inline Default. . . . 424
Limitations for Shared Inline Defaults . . . 424

15 Precomputed Result Sets. . . . 426
15.1 Benefits of Precomputed Result Sets. . . . 426
15.2 Configuring SAP ASE for Precomputed Result Sets. . . . 427
15.3 Creating Precomputed Result Sets. . . . 428
15.4 Identifying Precomputed Result Sets. . . . 428
15.5 Refreshing Precomputed Result Sets. . . . 429
15.6 Altering Precomputed Result Sets. . . . 431
15.7 Dropping or Truncating Precomputed Result Sets. . . . 432
15.8 Configuring Staleness. . . . 433
15.9 Querying Precomputed Result Sets. . . . 434
15.10 Rewriting Queries. . . . 434
15.11 Replicating Precomputed Result Sets. . . . 435
15.12 Restrictions for Precomputed Result Sets . . . 435

16 Batches and Control-of-Flow Language. . . . 437

16.1 Rules Associated with Batches. . . . 437
Examples of Using Batches. . . . 438
Batches Submitted as Files. . . . 441
16.2 Control-of-Flow Language Usage. . . . 441
if...else. . . . 442
case Expression. . . . 444
begin...end. . . . 453
while and break...continue. . . . 454
declare and Local Variables. . . . 456
goto. . . . 456
return. . . . 456
print. . . . 457
raiserror. . . . 458
Create Messages for print and raiserror. . . . 459
waitfor. . . . 460
Comments. . . . 461
16.3 Local Variables. . . . 463
Local Variables and select Statements. . . . 463
Local Variables and update Statements. . . . 465
Local Variables and Subqueries. . . . 465
Local Variables and while Loops and if...else Blocks. . . . 465
Variables and Null Values. . . . 466
16.4 Global Variables. . . . 467
Transactions and Global Variables. . . . 468

17 Transact-SQL Functions. . . . 471
17.1 Built-In Functions. . . . 471
System Functions. . . . 471
String Functions. . . . 472
Text and Image Functions 475
Aggregate Functions. . . . 476
Statistical Aggregate Functions. . . . 480
Mathematical Functions. . . . 482
Date Functions. . . . 483
Datatype Conversion Functions. . . . 484
Security Functions. . . . 496
XML Functions. . . . 496
17.2 User-Created Functions. . . . 497

18 Stored Procedures 498
18.1 Examples. . . . 499
18.2 Permissions. . . . 501

18.3 Performance. . . . 501
18.4 Create and Execute Stored Procedures. . . . 502
Deferred Name Resolution Usage. . . . 502
Parameters. . . . 503
Default Parameters. . . . 505
Using Multiple Parameters. . . . 508
LOB Datatypes in Stored Procedures. . . . 509
Procedure Groups. . . . 510
with recompile in create procedure. . . . 510
with recompile in execute. . . . 511
Nesting Procedures. . . . 511
Temporary Tables in Stored Procedures. . . . 512
Set Options in Stored Procedures. . . . 513
Execution of Stored Procedures. . . . 515
18.5 Deferred Compilation in Stored Procedures. . . . 520
18.6 Information Returned From Stored Procedures. . . . 521
Return Status. . . . 521
Check Roles in Procedures. . . . 523
Return Parameters. . . . 524
18.7 Restrictions Associated with Stored Procedures. . . . 528
Qualify Names Inside Procedures. . . . 528
18.8 Rename Stored Procedures. . . . 529
Rename Objects Referenced by Procedures. . . . 529
18.9 Stored Procedures as Security Mechanisms. . . . 530
18.10 Dropping Stored Procedures. . . . 530
18.11 System Procedures. . . . 530
Execute System Procedures. . . . 531
Permissions on System Procedures. . . . 531
Types of System Procedures. . . . 531
Other SAP ASE-Supplied Stored Procedures. . . . 531
18.12 Get Information About Stored Procedures. . . . 532
Get a Report with sp_help. . . . 532
View the Source Text of a Procedure with sp_helptext. . . . 532
Identify Dependent Objects with sp_depends. . . . 533
Identify Permissions with sp_helprotect. . . . 534

19 Extended Stored Procedures Usage. . . . 535
19.1 XP Server. . . . 535
CIS RPC Mechanism 536
sybesp_dll_version 536
19.2 Dynamic Link Library Support. . . . 537
19.3 Open Server API. . . . 537

19.4 ESPs and Permissions. . . . 538
19.5 ESPs and Performance. . . . 539
19.6 Create Functions for ESPs. . . . 539
Files for ESP Development. . . . 540
Open Server Data Structures. . . . 540
Open Server Return Codes. . . . 540
Outline of a Simple ESP Function. . . . 541
ESP Function Example. . . . 541
Building the DLL. . . . 545
19.7 Registering ESPs. . . . 546
create procedure Usage. . . . 547
sp_addextendedproc Usage. . . . 547
19.8 Remove ESPs. . . . 548
Renaming ESPs. . . . 549
19.9 Execute ESPs. . . . 549
19.10 System ESPs. . . . 550
19.11 Get Information About ESPs. . . . 550
19.12 ESP Exceptions and Messages. . . . 551

20 Cursors: Accessing Data. . . . 552
20.1 Types of Cursors. . . . 553
20.2 Cursor Scope. . . . 553
20.3 Cursor Scans and the Cursor Result Set. . . . 554
20.4 Make Cursors Updatable. . . . 555
Determine Which Columns Can Be Updated. . . . 556
20.5 How SAP ASE Processes Cursors. . . . 557
20.6 Monitor Cursor Statements. . . . 559
20.7 declare cursor . . . 560
cursor_scrollability. . . . 561
Cursor Sensitivity. . . . 561
read_only Option. . . . 562
20.8 Open Cursors. . . . 562
20.9 Fetch Data Rows Using Cursors. . . . 562
fetch Syntax. . . . 563
into Clause Usage. . . . 564
Check Cursor Status. . . . 564
Get Multiple Rows With Each Fetch. . . . 566
Check the Number of Rows Fetched. . . . 567
20.10 Update and Delete Rows Using Cursors. . . . 568
Update Cursor Result Set Rows. . . . 568
Delete Cursor Result Set Rows. . . . 569
20.11 Close and Deallocate Cursors. . . . 570

20.12 Cursor Examples. . . . 570
20.13 Cursors in Stored Procedures. . . . 575
20.14 Cursors and Locking. . . . 577
Cursor-Locking Options. . . . 578
20.15 Transaction Support for Updatable Cursors. . . . 578
20.16 Get Information About Cursors. . . . 579
20.17 Browse Mode Versus Cursors. . . . 580

21 Triggers: Enforce Referential Integrity. . . . 583
21.1 Use Triggers Versus Integrity Constraints. . . . 584
21.2 Create Triggers. . . . 584
create trigger Syntax. . . . 585
21.3 Use Triggers to Maintain Referential Integrity. . . . 585
Test Data Modifications Against the Trigger Test Tables. . . . 586
Insert Trigger Example. . . . 588
Delete Trigger Examples. . . . 588
Update Trigger Examples. . . . 590
21.4 Multirow Considerations. . . . 594
Insert Trigger Example Using Multiple Rows. . . . 594
Delete Trigger Example Using Multiple Rows. . . . 595
Update Trigger Example Using Multiple Rows. . . . 596
Conditional Insert Trigger Example Using Multiple Rows. . . . 596
21.5 Roll Back Triggers. . . . 597
21.6 Global Login Triggers. . . . 599
21.7 Nesting Triggers. . . . 599
Trigger Self-Recursion. . . . 600
21.8 Rules Associated with Triggers. . . . 602
Triggers and Permissions. . . . 602
Trigger Restrictions. . . . 602
Implicit and Explicit Null Values. . . . 603
Triggers and Performance. . . . 604
set Commands in Triggers. . . . 604
Renaming and triggers. . . . 604
21.9 Disable Triggers. . . . 604
21.10 Drop Triggers. . . . 605
21.11 Multiple Triggers. . . . 605
Changing the Order of When a Trigger Is Fired. . . . 606
Order of Triggers in Merge Statements. . . . 606
Transactional Behavior with Multiple Triggers. . . . 607
Disabling and Reenabling Triggers. . . . 607
21.12 Get Information About Triggers. . . . 608
sp_help. . . . 609

sp_helptext. 609
sp_depends. 610
21.13 instead of Triggers. 610
Inserted and Deleted Logical Tables. 611
Triggers and Transactions. 612
Nesting. 612
Recursion. 613
instead of insert Triggers. 613
instead of update Trigger. 615
instead of delete Trigger. 616
Searched and Positioned update and delete. 616
Get Information About `instead of` Triggers. 618

22 In-Row Off-Row LOB. 620
22.1 In-Row LOB Columns Compression 620
22.2 Migrate Off-Row LOB Data to In-Row Storage 621
In-Row LOB Columns and Bulk Copy. 621
Methods for Migrating Existing Data. 622
Guidelines for Selecting the In-Row LOB Length. 630
Identifying In-Row LOB Length Selection. 630
22.3 Downgrading Tables Containing In-Row LOB Columns. 631

23 Transactions: Maintain Data Consistency and Recovery. 632
23.1 Transactions and Consistency. 633
23.2 Transactions and Recovery. 634
23.3 Transaction Usage. 634
Allow Data Definition Commands in Transactions. 634
System Procedures That Are Not Allowed in Transactions. 636
Begin and Commit Transactions. 636
Roll Back and Save Transactions. 637
Transaction States. 639
Nested Transactions. 641
Example of a Transaction. 641
23.4 Transaction Mode and Isolation Level. 642
Choose a Transaction Mode. 642
Choose an Isolation Level. 644
Compliance with SQL Standards. 650
Use the Lock Table Command to Improve Performance. 651
23.5 Transactions in Stored Procedures and Triggers. 651
Errors and Transaction Rollbacks. 652
Transaction Modes and Stored Procedures. 655
23.6 Use Cursors in Transactions. 657

23.7 Issues to Consider When Using Transactions. . . . 658
23.8 Backup and Recovery of Transactions. . . . 659

24 Locking Commands and Options. . . . 660
24.1 wait/nowait Option of the Lock Table Command. . . . 660
24.2 Session-Level Lock-Wait Limit. . . . 661
24.3 Server-Wide Lock-Wait Limit. . . . 662
24.4 Information on the Number of Lock-Wait Timeouts. . . . 662
24.5 Readpast Locking for Queue Processing. . . . 662
Incompatible Locks During readpast Queries. . . . 663
Allpages-Locked Tables and readpast Queries. . . . 663
Effects of Isolation Levels Select Queries with readpast. . . . 664
Data Modification Commands with readpast and Isolation Levels. . . . 665
text, unitext, and image columns and readpast. . . . 665

25 The pubs2 Database. . . . 666
25.1 Tables in the pubs2 Database. . . . 666
publishers Table. . . . 666
authors Table. . . . 667
titles Table. . . . 667
titleauthor Table. . . . 669
salesdetail Table. . . . 670
sales Table. . . . 671
stores Table. . . . 671
roysched Table. . . . 672
discounts Table. . . . 672
blurbs Table. . . . 673
au_pix Table. . . . 673
25.2 Diagram of the pubs2 Database. . . . 674

26 The pubs3 Database. . . . 675
26.1 Tables in the pubs3 Database. . . . 675
publishers Table. . . . 675
authors Table. . . . 676
titles Table. . . . 676
titleauthor Table. . . . 678
salesdetail Table. . . . 678
sales Table. . . . 679
stores Table. . . . 680
store_employees Table. . . . 680
roysched Table. . . . 680
discounts Table. . . . 681

blurbs Table. 681
26.2 Diagram of the pubs3 Database. 681

1 SQL Building Blocks

SQL includes commands for querying (retrieving data from) a database, and for creating new databases and database objects, adding new data, modifying existing data, and other functions.

Originally developed by the IBM San Jose Research Laboratory in the late 1970s, SQL (Structured Query Language) has been adopted by, and adapted for, many relational database management systems. It has been approved as the official relational query language by the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO).

Transact-SQL®, the SAP® extension of SQL, is compatible with IBM SQL and most other commercial implementations of SQL. It provides important extra capabilities and functions, such as summary calculations, stored procedures (predefined SQL statements), and error handling.

> **Note**
>
> If Java is enabled on your server, you can install and use Java classes in the database. You can invoke Java operations and store Java classes using standard Transact-SQL commands.

Related Information

The pubs2 Database [page 666]
The pubs3 Database [page 675]

1.1 Tables, Columns, and Rows

In relational database management systems, users access and modify data that is stored in tables. SQL is specifically designed for the relational model of database management.

Each row, or record, in a table describes one occurrence of a piece of data—a person, a company, a sale, or some other thing. Each column, or field, describes one characteristic of the data—a person's name or address, a company's name or president, quantity of items sold.

A relational database is made up of a set of tables that can be related to each other. The database usually contains many tables.

1.2 Queries, Data Modification, and Commands

In SQL, a query requests data using the `select` command.

For example, this query asks for authors who live in the state of California:

```
select au_lname, city, state
from authors
where state = "CA"
```

Data modification refers to adding, deleting, or changing data using the `insert`, `delete`, or `update` commands. For example:

```
insert into authors (au_lname, au_fname, au_id)
values ("Smith", "Gabriella", "999-03-2346")
```

Other SQL commands, such as dropping tables or adding users, perform administrative operations. For example:

```
drop table authors
```

Each command or SQL statement begins with a keyword, such as `insert`, that names the basic operation performed. Many SQL commands also have one or more keyword phrases, or clauses, that tailor the command to meet a particular need. When you run a query, Transact-SQL displays the results. If no data meets the criteria specified in the query, you see a message to that effect. Data modification statements and administrative statements do not retrieve data, and therefore, do not display results. Transact-SQL provides a message to let you know whether the data modification or other command has been performed.

1.3 Relational Operations

The basic query operations in a relational system include selection (also called restriction), projection, and join. These can all be combined in the SQL `select` command.

A selection is a subset of the rows in a table. Specify the limiting conditions in the `select` query. For example, to look only at the rows for all authors who live in California, enter:

```
select *
from authors
where state = "CA"
```

A projection is a subset of the columns in a table. For example, this query displays only the name and city of all authors, omitting the street address, the phone number, and other information:

```
select au_fname, au_lname, city
from authors
```

A join links the rows in two or more tables by comparing the values in specified fields. For example, suppose you have one table containing information about authors, including the columns `au_id` (author identification number) and `au_lname` (author's last name). A second table contains title information about books, including

a column that gives the ID number of the book's author (au_id). You might join the authors table and the titles table, testing for equality of the values in the au_id columns of each table. Whenever there is a match, a new row—containing columns from both tables—is created and appears as part of the result of the join. Joins are often combined with projections and selections so that only selected columns of selected matching rows appear.

```
select *
from authors, publishers
where authors.city = publishers.city
```

1.4 Compiled Objects

SAP® Adaptive Server® Enterprise (ASE) uses compiled objects to hold vital information about each database and to help you access and manipulate data.

A compiled object requires entries in the sysprocedures table and includes:

- Check constraints
- Defaults
- Rules
- Stored procedures
- Extended stored procedures
- Triggers
- Views
- Functions
- Computed columns
- Partition conditions

Compiled objects are created from source text, which are SQL statements that describe and define the compiled object. When a compiled object is created, SAP® ASE:

1. Parses the source text, catching any syntactic errors, to generate a parsed tree.
2. Normalizes the parsed tree to create a normalized tree, which represents the user statements in a binary tree format. This is the compiled object.
3. Stores the compiled object in the sysprocedures table.
4. Stores the source text in the syscomments table.

1.4.1 Save or Restore Source Text

If a compiled object does not have matching source text in the syscomments table, you can restore the source text to syscomments.

Use any of the following methods:

- Load the source text from a backup.
- Manually re-create the source text.

- Reinstall the application that created the compiled object.

1.4.2 Verify and Encrypt Source Text

Use these commands to verify the existence of source text and encrypt the source text of a compiled object.

- `sp_checkresource` – verifies that source text is present in `syscomments` for each compiled object.
- `sp_hidetext` – encrypts the source text of a compiled object in the `syscomments` table.
- `sp_helptext` – displays the source text if it is present in `syscomments`, or notifies you of missing source text.
- `dbcc checkcatalog` – notifies you of missing source text.

1.4.3 Replacing Object Definitions

Replace existing compiled object definitions with new definitions while preserving the original names, object IDs, security attributes—such as auditing options and permissions— and replication attributes.

The `create or replace` functionality creates a new object if it does not exist, or replaces an existing object with the same name. The `or replace` clause implicitly drops and re-creates an existing object of the same name and type within the database, changing the definition of the object, while preserving the existing security and replication attributes. If the text of the compiled object was hidden before it was replaced, it will remain hidden after being replaced.

The `or replace` functionality is supported only for objects that do not contain data. Check constraints, computed columns, and partition conditions cannot be replaced.

If the object is in use while being replaced, error 3702 is raised:

```
"Cannot drop or replace the %S_MSG '%.*s' because it is currently in use."
```

When an object is replaced, SAP ASE replaces its definition in the following system tables: `sysprocedures`, `syscomments`, `sysdepends`, and `syscolumns`. Some fields in the `sysobjects` table are also updated. The query tree for the object is normalized before being replaced in `sysprocedures`.

The replaced object may be used in other object definitions. SAP ASE recompiles the replaced object when it is used, however, in some cases, you may need to replace the calling object when the interface of the replaced object does not match with that used in the calling object. You can run `sp_depends` on the replaced object to verify whether there are calling objects and then replace them. For details, see *Objects Dependent on Replaced Objects*, in *Reference Manual: Commands:*

- *create procedure*
- *create view*
- *create function*

With granular permissions enabled or disabled, you must be the object owner to replace a compiled object. You cannot replace a compiled object by impersonating the object owner through an alias or `setuser`. However, if you are the owner through `set proxy`, you can replace a compiled object.

> **Note**
>
> The `create or replace` functionality performs an implicit `drop` followed by `create` in the same transaction. Because of this, additional transaction log space is required. If you use `create or replace` instead of dropping an object and then creating the object, you may need to increase the size of the transaction log.

1.5 Compliance to ANSI Standards

Certain behaviors defined by the SQL standards are incompatible with SAP ASE applications. Transact-SQL provides `set` options that allow you to toggle these behaviors.

By default, compliant behavior is enabled for all Embedded SQL™ precompiler applications.

Option	Setting
`ansi_permissions`	`on`
`ansinull`	`on`
`arithabort`	`off`
`arithabort numeric_truncation`	`on`
`arithignore`	`off`
`chained`	`on`
`close on endtran`	`on`
`fipsflagger`	`on`
`quoted_identifier`	`on`
`string_rtruncation`	`on`
`transaction isolation level`	`3`

1.5.1 Federal Information Processing Standards (FIPS) Flagger

For customers writing applications that must conform to the ANSI SQL standard, SAP ASE provides a `set fipsflagger` option.

When this option is turned on, all commands containing Transact-SQL extensions that are not allowed in entry-level ANSI SQL generate an informational message. This option does not disable the extensions. Processing completes when you issue the non-ANSI SQL command.

1.5.2 Chained Transactions and Isolation Levels

SAP ASE provides SQL standard-compliant "chained" transaction behavior as an option.

In chained mode, all data retrieval and modification commands (`delete`, `insert`, `open`, `fetch`, `select`, and `update`) implicitly begin a transaction. Since such behavior is incompatible with many Transact-SQL applications, Transact-SQL style (or "unchained") transactions remain the default.

You can initiate chained transaction mode using the `set chained` option. The `set transaction isolation level` option controls transaction isolation levels.

Related Information

Transactions: Maintain Data Consistency and Recovery [page 632]

1.5.3 Identifier Compliance to ANSI Standards

To be compliant with entry-level ANSI SQL, identifiers cannot begin with a pound sign (#), have more than 18 characters, or contain lowercase letters. SAP ASE supports delimited identifiers for table, view, and column names which can be used to avoid certain restrictions on object names.

Delimited identifiers are object names enclosed in double quotation marks. Use the `set quoted_identifier` option to recognize delimited identifiers. When this option is on, all characters enclosed in double quotes are treated as identifiers. Because this behavior is incompatible with many existing applications, the default setting for this option is off.

1.5.4 SQL Standard-Style Comments

In Transact-SQL, comments are delimited by "/*" and "*/", and can be nested. Transact-SQL also supports SQL standard-style comments, which consist of any string beginning with two connected minus signs, a comment, and a terminating new line.

```
select "hello" -- this is a comment
```

The Transact-SQL "/*" and "*/" comment delimiters are fully supported, but "--" within Transact-SQL comments is not recognized.

1.5.5 Right Truncation of Character Strings

The `string_rtruncation set` option controls silent truncation of character strings for SQL standard compatibility. Enable this option to prohibit silent truncation and enforce SQL standard behavior.

1.5.6 Permissions Required for update and delete Statements

The `ansi_permissions set` option determines permissions that are required for `delete` and `update` statements.

When enabled, SAP ASE uses the more stringent ANSI SQL permission requirements for these statements. By default, this option is disabled because this behavior is incompatible with many existing applications.

1.5.7 Arithmetic Errors

The `arithabort` and `arithignore` options for `set` allow compliance with the ANSI SQL standard.

- `arithabort arith_overflow` specifies behavior following a divide-by-zero error or a loss of precision. The default setting, `arithabort arith_overflow on`, rolls back the entire transaction in which the error occurs. If the error occurs in a batch that does not contain a transaction, `arithabort arith_overflow on` does not roll back earlier commands in the batch, but SAP ASE does not execute statements in the batch that follow the error-generating statement.
  If you set `arithabort arith_overflow off`, SAP ASE aborts the statement that causes the error but continues to process other statements in the transaction or batch.
- `arithabort numeric_truncation` specifies behavior following a loss of scale by an exact numeric type. The default setting, on, aborts the statement that causes the error but continues to process other statements in the transaction or batch. If you set `arithabort numeric_truncation off`, the query results is truncated and processing continues. For compliance with the ANSI SQL standard, enter `set arithabort numeric_truncation on`.
- `arithignore arith_overflow` determines whether SAP ASE displays a message after a divide-by-zero error or a loss of precision. The default setting, off, displays a warning message after these errors. Setting

`arithignore arith_overflow on` suppresses warning messages after these errors. For compliance to the ANSI SQL standard, enter `set arithignore off`.

1.5.8 Synonymous Keywords

Several keywords added for SQL standard compatibility are synonymous with existing Transact-SQL keywords.

Current Syntax	Additional Syntax
`commit tran`, `commit transaction`, `rollback tran`, `rollback transaction`	`commit work`, `rollback work`
`any`	`some`
`grant all`	`grant all privileges`
`revoke all`	`revoke all privileges`
`max (<expression>)`	`max ([all | distinct]) <expression>`
`min (<expression>)`	`min ([all | distinct]) <expression>`
`user_name <function>`	`user <keyword>`

1.5.9 Treatment of Nulls

The `set` option `ansinull` determines whether or not evaluation of null-valued operands in SQL equality (=) or inequality (!=) comparisons and aggregate functions is SQL-standard-compliant.

This option does not affect how null values are evaluated in other kinds of SQL statements such as `create table`.

1.6 Data and Language Characters

The characters recognized by SAP ASE are limited in part by the language of the installation and the default character set.

Therefore, the characters allowed in SQL statements and in the data contained in the server vary from installation to installation and are determined in part by definitions in the default character set.

SQL statements must follow precise syntactical and structural rules, and can contain operators, constants, SQL keywords, special characters, and identifiers. Identifiers are database objects within the server, such as

database names or table names. Naming conventions vary for some parts of the SQL statement. Operator, constants, SQL keywords, and Transact-SQL extensions must adhere to stricter naming restrictions than identifiers, which themselves cannot contain operators and special characters. However, identifiers, the data contained within the server, can be named following more permissive rules.

SQL Data Characters

The set of SQL data characters is the larger set from which both SQL language characters and identifier characters are taken. Any character in an SAP ASE character set, including both single-byte and multibyte characters, can be used for data values.

SQL Language Characters

SQL keywords, Transact-SQL extensions, and special characters, such as the comparison operators > and <, can be represented only by 7-bit ASCII values A – Z, a – z, 0 – 9, and certain ASCII characters.

These are the ASCII characters used in SQL:

Character	Description
;	(semicolon)
(	(open parenthesis)
)	(close parenthesis)
,	(comma)
:	(colon)
%	(percent sign)
-	(minus sign)
?	(question mark)
'	(single quote)
"	(double quote)
+	(plus sign)
_	(underscore)
*	(asterisk)
/	(slash)

Character	Description
	(space)
<	(less than operator)
>	(greater than operator)
=	(equals operator)
&	(ampersand)
|	(vertical bar)
^	(circumflex)
[	(left bracket)
]	(right bracket)
@	(at sign)
~	(tilde)
!	(exclamation point)
$	(dollar sign)
#	(number sign)
.	(period)

1.6.1 Naming Convention Identifiers

Conventions for naming database objects apply throughout SAP ASE software and documentation. Most user-defined identifiers can be up to 255 bytes in length; other identifiers can be only up to 30 bytes. In either case, the byte limit is independent of whether or not multibyte characters are used.

255-Byte-Limit Identifiers	30-Byte-Limit Identifiers
table name	cursor name
column name	server name
index name	host name
view name	login name
user-defined datatype	password

255-Byte-Limit Identifiers	30-Byte-Limit Identifiers
trigger name	host process identification
default name	application name
rule name	initial language name
constraint name	character set name
stored procedure name	user name
variable name	group name
JAR name	database name
Lightweight processes (LWPs) or dynamic statement name	cache name
function name	logical device name
time range name	segment name
function name	session name
application context name	execution class name
	engine name
	quiesce tag name

You must declare the first character of an identifier as an alphabetic character in the character set definition in use on SAP ASE. You can also use the @ sign or _ (underscore character). The @ sign as the first character of an identifier indicates a local variable.

Temporary table names must either begin with # (the pound sign), if they are created outside `tempdb`, or be preceded by "`tempdb`." If you create a temporary table with a name requiring fewer than 238 bytes, SAP ASE adds a 17-byte suffix to ensure that the table name is unique. If you create a temporary table with a name of more than 238 bytes, SAP ASE uses only the first 238 bytes, and then adds the 17-byte suffix.

After the first character, identifiers can include characters declared as alphabetic, numeric, or the character $, #, @, _, ¥ (yen), or £ (pound sterling). However, you cannot use two @@ symbols together at the beginning of a named object, as in "@@myobject." This naming convention is reserved for global variables, which are system-defined variables that are automatically updated.

Case sensitivity is set during server installation and can be changed only by a system administrator. To see the setting for your server, execute:

```
sp_helpsort
```

On a server that is not case-sensitive, the identifiers `MYOBJECT`, `myobject`, and `MyObject` (and all combinations of case) are considered identical. You can create only one of these objects, but you can use any combination of case to refer to that object.

You cannot use embedded spaces, or SQL reserved words in identifiers. Use `valid_name` to determine if an identifier you have created is acceptable to SAP ASE:

```
select valid_name ("@name", 255)
```

See *Reserved Words* and *Transact-SQL Functions*, in *Reference Manual: Building Blocks*

1.6.1.1 Multibyte Character Sets

In multibyte character sets, a wider range of characters is available for use in identifiers.

For example, on a server that has the Japanese language installed, you can use the following types of characters as the first character of an identifier: Zenkaku or Hankaku Katakana, Hiragana, Kanji, Romaji, Cyrillic, Greek, or ASCII.

Although Hankaku Katakana characters are allowed in identifiers on Japanese systems, SAP recommends that you do not use them in heterogeneous systems. These characters cannot be converted between the EUC-JIS and Shift-JIS character sets.

The same is true for some 8-bit European characters. For example, the character "Œ," the OE ligature, is part of the Macintosh character set (code point 0xCE), but does not exist in the ISO 8859-1 (iso_1) character set. If "Œ" exists in data being converted from the Macintosh to the ISO 8859-1 character set, it causes a conversion error.

If an object identifier contains a character that cannot be converted, the client loses direct access to that object.

1.6.1.2 Delimited Identifiers

Delimited identifiers are object names enclosed in double quotes. Using delimited identifiers allows you to avoid certain restrictions on object names.

You can use double quotes to delimit table, view, and column names; you cannot use them for other database objects.

Delimited identifiers can be reserved words, can begin with nonalphabetic characters, and can include characters that would not otherwise be allowed. They cannot exceed 253 bytes. A pound sign (#) is illegal as a first character of any quoted identifier.

Before you create or reference a delimited identifier, execute:

```
set quoted_identifier on
```

This allows SAP ASE to recognize delimited identifiers. Each time you use the quoted identifier in a statement, you must enclose it in double quotes. For example:

```
create table "1one"(col1 char(3))
select * from "1one"
create table "include spaces" (col1 int)
```

> **Note**
>
> You cannot use delimited identifiers with `bcp`, as these identifiers may not be supported by all front-end products, and may produce unexpected results when used with system procedures.

While the `quoted_identifier` option is turned on, use single quotes around character or date strings. Delimiting these strings with double quotes causes SAP ASE to treat them as identifiers. For example, to insert a character string into `<col1>` of `<1onetable>`, use:

```
insert "1one"(col1) values ('abc')
```

rather than:

```
insert "1one"(col1) values ("abc")
```

To insert a single quote into a column, use two consecutive single quotation marks. For example, to insert the characters “a’b” into `col1`, use:

```
insert "1one"(col1) values('a''b')
```

Syntax that Includes Quotes

When you set the `quoted_identifier` option to `on` for a session, use double quotes to delimit object names that may cause syntax errors. Use single quotes for character strings. When you set the `quoted_identifier` option to `off` for a session (the default), use double or single quotes to delimit character strings (you cannot quote identifiers).

This example creates table `1one`, which, because its name starts with a digit, fails the rules for identifiers and must be set in quotes:

```
set quoted identifier on
go
create table "1one" (c1 int)
```

Although `create table` and most other SQL statements require an identifier to name a table or other SQL object, some commands, functions, and so on require that you supply an object name as a string, whether or not you set the `quoted_identifier` option to `on`. For example

```
select object_id('1one')
```

```
-----------------------
                896003192
```

You can include an embedded double quote in a quoted identifier by doubling the quote. This creates a table named `embedded”quote`:

```
create table "embedded""quote" (c1 int)
```

However, you need not double the quote when the statement syntax requires the object name to be expressed as a string:

```
select object_id('embedded"quote')
```

Bracketed Delimited Identifiers

Bracketed identifiers are supported. The behavior is identical to that of quoted identifiers, with the exception that you need not set the `quoted_identifier` option to `on` to use them.

```
create table [bracketed identifier](c1 int)
```

Support for brackets with delimited identifiers increases platform compatibility.

1.6.1.3 Uniqueness and Qualification Conventions

The names of database objects need not be unique in a database. However, column names and index names must be unique within a table, and other object names must be unique for each owner within a database. Database names must be unique.

If you try to create a column using a name that is not unique in the table, or to create another database object, such as a table, a view, or a stored procedure, with a name that you have already used in the same database, SAP ASE responds with an error message.

You can uniquely identify a table or column by adding other names that qualify it. The database name, the owner's name, and, for a column, the table name or view name may be used to create a unique ID. Each of these qualifiers is separated from the next by a period.

For example, if the user "sharon" owns the `authors` table in the `pubs2` database, the unique identifier of the `city` column in that table is:

```
pubs2.sharon.authors.city
```

The same naming syntax applies to other database objects. You can refer to any object in a similar fashion:

```
pubs2.dbo.titleview
```

```
dbo.postalcoderule
```

If the `quoted_identifier` option of the `set` command is on, you can use double quotes around individual parts of a qualified object name. Use a separate pair of quotes for each qualifier that requires quotes. For example, use:

```
database.owner."table_name"."column_name"
```

rather than:

```
database.owner."table_name.column_name"
```

The full naming syntax is not always allowed in `create` statements because you cannot create a view, procedure, rule, default, or trigger in a database other than the one you are currently in. The naming conventions are indicated in the syntax as:

```
[[database.]owner.]object_name
```

or:

```
[owner.]object_name
```

The default value for `<owner>` is the current user, and the default value for `<database>` is the current database. When you reference an object in any SQL statement, other than a `create` statement, without qualifying it with the database name and owner name, SAP ASE first looks at all the objects you own, and then at the objects owned by the database owner. As long as there is enough information to identify an object, you need not type every element of its name. You can omit intermediate elements and indicate their positions with periods:

```
<database>..<table_name>
```

In the example above, you must include the starting element if you are using this syntax to create tables. If you omit the starting element, a table named `..mytable` is created. The naming convention prevents you from performing certain actions on such a table, such as cursor updates.

When qualifying a column name and a table name in the same statement, use the same naming abbreviations for each; they are evaluated as strings and must match, or an error is returned. Here are two examples with different entries for the column name. The second example is incorrect, and cannot execute, because the syntax for the column name does not match the syntax for the table name.

```
select pubs2.dbo.publishers.city
from pubs2.dbo.publishers
```

```
city
-----------------------
Boston
Washington
Berkeley
```

```
select pubs2.sa.publishers.city
from pubs2..publishers
```

```
The column prefix "pubs2.sa.publishers" does not match
a table name or alias name used in the query.
```

1.6.1.4 Remote Servers

You can execute stored procedures on a remote SAP ASE server. The results from the stored procedure appear on the terminal that calls the procedure.

The syntax for identifying a remote server and the stored procedure is:

```
[execute] server.[database].[owner].procedure_name
```

You can omit the `execute` keyword when the remote procedure call (RPC) is the first statement in a batch. If other SQL statements precede the RPC, you must use `execute` or `exec`. You must include both the server name and the stored procedure name. If you omit the database name, a search is performed for `procedure_name` in your default database. If you give the database name, you must also give the procedure owner's name, unless you own the procedure or the procedure is owned by the database owner.

The following statements execute the stored procedure `byroyalty` in the `pubs2` database located on the GATEWAY server:

Statement	Notes
`GATEWAY.pubs2.dbo.byroyalty` `GATEWAY.pubs2..byroyalty`	`byroyalty` is owned by the database owner.
`GATEWAY...byroyalty`	Use if `pubs2` is the default database.
`declare @var int exec GATEWAY...byroyalty`	Use when the statement is not the first statement in a batch.

See, *Managing Remote Servers*, in the *System Administration Guide: Volume 1* for information about configuring an SAP ASE server for remote access. A remote server name (GATEWAY in the previous example) must match a server name in your local `interfaces` file. If the server name in `interfaces` is in uppercase letters, you must also use uppercase letters in the RPC to match the server name.

1.7 Expressions in SAP ASE

An expression is a combination of one or more constants, literals, functions, column identifiers, and variables, separated by operators, that returns a single value.

Expressions can be of several types, including arithmetic, relational, logical (or Boolean), and character string. In some Transact-SQL clauses, a subquery can be used in an expression. A case expression can be used in an expression.

Use parentheses to group the elements in an expression. When you provide "`<expression>`" as a variable in a syntax statement, a simple expression is assumed. Use logical_expression when only a logical expression is acceptable.

1.7.1 Arithmetic Operators

SAP ASE uses certain arithmetic operators.

Operator	Meaning
+	Addition

Operator	Meaning
–	Subtraction
*	Multiplication
/	Division
%	Modulo (Transact-SQL extension)

Operators have certain precedence levels among arithmetic operators. In order of precedence (from lowest to highest):

1. unary (single argument) – + ~
2. * /%
3. binary (two argument) + – & | ^
4. not
5. and
6. or

Use addition, subtraction, division, and multiplication on exact numeric, approximate numeric, and money type columns.

A modulo operator, which can be used on exact numeric columns except money and numeric, finds the remainder after a division involving two numbers. For example, using integers: 21 % 11 = 10 because 21 divided by 11 equals 1, with a remainder of 10. You can obtain a noninteger result with numeric or decimal datatypes: 1.2 % 0.07 = 0.01 because 1.2 / 0.07 = 17 * 0.07 + 0.01. You receive similar results from float and real datatype calculations: 1.2e0 % 0.07 = 0.010000.

When you perform arithmetic operations on mixed datatypes (for example, float and int) SAP ASE follows specific rules for determining the type of the result.

Related Information

Datatypes [page 191]

1.7.2 Bitwise Operators

The bitwise operators are a Transact-SQL extension for use with the integer datatype.

These operators convert each integer operand into its binary representation and then evaluate the operands column by column. A value of 1 corresponds to true; a value of 0 corresponds to false. The following table summarize the results for operands of 0 and 1. If either operand is NULL, the bitwise operator returns NULL.

& (and)	1	0

1	1	0
0	0	0
`| (or)`	1	0
1	1	1
0	1	0
`^ (exclusive or)`	1	0
1	0	1
0	1	0
`~ (not)`		
1	FALSE	
0	0	

The following examples use two `tinyint` arguments: A = 170 (10101010 in binary form) and B = 75 (01001011 in binary form).

Operation	Binary Form	Result	Explanation

(A & B)	`10101010`
	`01001011`
	`----------`
	`00001010`
		10	Result column equals 1 if both A and B are 1. Otherwise, result column equals 0.

(A | B)	`10101010`
	`01001011`
	`----------`
	`11101011`
		235	Result column equals 1 if either A or B, or both, is 1. Otherwise, result column equals 0.

(A ^ B)	`10101010`
	`01001011`
	`----------`
	`11100001`
		225	Result column equals 1 if either A or B, but not both, is 1.

(~A)	`10101010`
	`----------`
	`01010101`
		85	All 1s are changed to 0s and all 0s to 1s.

1.7.3 The String Concatenation Operator

The string operator + can concatenate two or more character or binary expressions.

For example:

1. ```
   select Name = (au_lname + ", " + au_fname)
       from authors
   ```

   Displays author names under the column heading "Name" in last-name, first-name order, with a comma after the last name; for example, "Bennett, Abraham."

2. ```
   select "abc" + "" + "def"
   ```

   Returns the string "abc def". The empty string is interpreted as a single space in all `char`, `varchar`, `nchar`, `nvarchar`, and `text` concatenation, and in `varchar` insert and assignment statements.

When concatenating noncharacter, nonbinary expressions, use `convert`:

```
select "The date is " +
    convert(varchar(12), getdate())
```

1.7.4 The Comparison Operators

SAP ASE uses certain comparison operators.

Operator	Meaning
=	Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
<>	Not equal to
! =	Not equal to (Transact-SQL extension)
! >	Not greater than (Transact-SQL extension)
! <	Not less than (Transact-SQL extension)

In comparing character data, < means closer to the beginning of the server's sort order and > means closer to the end of the sort order. Uppercase and lowercase letters are equal in a sort order that is case-insensitive. Use `sp_helpsort` to see the sort order for your server. For comparison purposes, trailing blanks are ignored.

In comparing dates, < means earlier than and > means later than.

Put single or double quotes around all character and `date` and `time` data used with a comparison operator:

```
= "Bennet"
  "May 22 1947"
```

1.7.5 Nonstandard Operators

SAP ASE supports certain nonstandard operators that are Transact-SQL extensions.

- Modulo operator: `%`
- Negative comparison operators: `!>`, `!<`, `!=`
- Bitwise operators: `~`, `^`, `|`, `&`
- Join operators: `*=` and `=*`

1.7.6 Character Expression Comparisons

SAP ASE treats character constant expressions as `varchar`. If they are compared with non-`varchar` variables or column data, the datatype precedence rules are used in the comparison (that is, the datatype with lower precedence is converted to the datatype with higher precedence).

If implicit datatype conversion is not supported, you must use the `convert` function. See the *Reference Manual: Building Blocks* for more information on supported and unsupported conversions.

Comparison of a `char` expression to a `varchar` expression follows the datatype precedence rule; the "lower" datatype is converted to the "higher" datatype. All `varchar` expressions are converted to `char` (that is, trailing blanks are appended) for the comparison.

1.7.7 Empty Strings

An empty string ("") or ('') is interpreted as a single blank in `insert` or assignment statements on `varchar` data.

When `varchar`, `char`, `nchar`, or `nvarchar` data is concatenated, the empty string is interpreted as a single space. For example, this statement is stored as "abc def":

```
"abc" + "" + "def"
```

An empty string is never evaluated as NULL.

1.7.8 Quotation Marks

You can specify literal quotes by using an additional quote with a quote of the same type, or by enclosing a quote in the opposite kind of quotation mark.

The first method is to use an additional quote with a quote of the same type. This is called “escaping” the quote. For example, if you begin a character entry with a single quote, but you want to include a single quote as part of the entry, use two single quotes:

```
’I don’’t understand.’
```

Here is an example containing internal double and single quotes. The single quote does not have to be escaped, but the double quote does:

```
"He said, ""It’s not really confusing."""
```

The second method is to enclose a quote in the opposite kind of quotation mark. In other words, surround an entry containing a double quote with single quotes (or vice versa). Here are some examples:

```
’George said, "There must be a better way."’
"Isn’t there a better way?"
’George asked, "Isn”t there a better way?"’
```

To continue a character string that would go off the end of one line on your screen, enter a backslash (\) before going to the following line.

> **Note**
>
> If the `quoted_identifier` option is set to on, do not use double quotes around character or date data. You must use single quotes, or the data is treated as an identifier.

Related Information

Delimited Identifiers [page 31]

1.7.9 Relational and Logical Expressions

A logical expression or relational expression returns TRUE, FALSE, or UNKNOWN.

The general patterns are:

```
expression <comparison_operator> [any | all] <expression>
```

```
expression [not] in <expression>
```

```
[not] exists <expression>
```

```
expression [not] between <expression> and <expression>
```

```
expression [not] like "<match_string>" [escape "<escape_character>"]
```

```
not expression like "<match_string>" [escape "<escape_character>"]
```

```
<expression> is [not] null
```

```
not <logical_expression>
```

```
logical_expression {and | or} <logical_expression>
```

- `any` is used with <, >, =, and a subquery. It returns results when any value retrieved in the subquery matches the value in the `where` or `having` clause of the outer statement.
- `all` is used with < or > and a subquery. It returns results when all values retrieved in the subquery are less than (<) or greater than (>) the value in the `where` or `having` clause of the outer statement.
- `in` returns results when any value returned by the second expression matches the value in the first expression. The second expression must be a subquery or a list of values enclosed in parentheses. `in` is equivalent to = `any`.
- `and` connects two expressions and returns results when both are true.
- `or` connects two or more conditions and returns results when either condition is true.

When more than one logical operator is used in a statement, `and` is evaluated before `or`. Use parentheses to change the order of execution.

This truth table shows the results of logical operations, including those that involve null values:

and	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
NULL	UNKNOWN	FALSE	UNKNOWN
or	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN

NULL	TRUE	UNKNOWN	UNKNOWN
not			
TRUE	FALSE		
FALSE	TRUE		
NULL	UNKNOWN		

The result UNKNOWN indicates that one or more of the expressions evaluates to NULL, and that the result of the operation cannot be determined to be either TRUE or FALSE.

1.8 Transact-SQL Extensions

Transact-SQL enhances the power of SQL, and minimizes the occasions on which users must resort to a programming language to accomplish a desired task. Transact-SQL capabilities go beyond the ISO standards and the many commercial versions of SQL.

See, *Reference Manual: Commands* for the Transact-SQL extensions for each command.

1.8.1 compute Clause

The Transact-SQL `compute` clause extension is used with the row aggregate functions `sum`, `max`, `min`, `avg`, `count`, and `count_big` to calculate summary values.

Queries that include a `compute` clause display results with both detail and summary rows. These reports resemble those produced by almost any database management system (DBMS) with a report generator. `compute` displays summary values as additional rows in the results, instead of as new columns.

Related Information

Aggregates, Grouping, and Sorting [page 285]

1.8.2 Control-of-Flow Language

Transact-SQL provides control-of-flow language that you can use as part of any SQL statement or batch.

These constructs are available:

- `begin...end`
- `break`

- `continue`
- `declare`
- `goto label`
- `if...else`
- `print`
- `raiserror`
- `return`
- `waitfor`
- `while`.

You can define local variables with `declare` and assigned values. A number of predefined global variables are supplied by the system.

Transact-SQL also supports `case` expressions, which include the keywords `case`, `when`, `then`, `coalesce`, and `nullif`. `case` expressions replace the `if` statements of standard SQL. `case` expressions are allowed anywhere a value expression is used.

1.8.3 Stored Procedures

One of the most important Transact-SQL extensions is the ability to create stored procedures. A stored procedure is a collection of SQL statements and optional control-of-flow statements stored under a name.

The creator of a stored procedure can also define parameters to be supplied when the stored procedure is executed.

The ability to write your own stored procedures greatly enhances the power, efficiency, and flexibility of the SQL database language. Since the execution plan is saved after stored procedures are run, stored procedures can subsequently run much faster than standalone statements.

Stored procedures supplied by SAP ASE, called system procedures, aid in SAP ASE system administration.

You can execute stored procedures on remote servers. All Transact-SQL extensions support return values from stored procedures, user-defined return statuses from stored procedures, and the ability to pass parameters from a procedure to its caller.

Related Information

Stored Procedures [page 498]

1.8.4 Extended Stored Procedures

An extended stored procedure (ESP) uses the same interface as a stored procedure, but instead of containing SQL statements and control-of-flow statements, it executes procedural language code that has been compiled into a dynamic link library (DLL).

The procedural language in which an ESP function is written can be any language capable of calling C language functions and manipulating C datatypes.

ESPs allow SAP ASE to perform a task outside the relational database management system (RDBMS), in response to an event occurring within the database. For example, you could use an ESP to send an e-mail notification or network-wide broadcast in response to an event occurring within the RDBMS.

There are some SAP ASE-supplied ESPs, called system extended stored procedures. One of these, `xp_cmdshell`, allows you to execute an operating system command from within SAP ASE.

ESPs are implemented by an Open Server™ application called XP Server™, which runs on the same machine as your SAP ASE server. Remote execution of a stored procedure is called a remote procedure call (RPC). Your SAP ASE server and XP Server communicate through RPCs. XP Server is automatically installed with SAP ASE.

See, *System Extended Stored Procedures*, in the *Reference Manual: Procedures*.

Related Information

Extended Stored Procedures Usage [page 535]

1.8.5 Triggers

A trigger is a stored procedure that instructs the system to take one or more actions when a specific change is attempted. By preventing incorrect, unauthorized, or inconsistent changes to data, triggers help maintain the integrity of a database.

Triggers can also protect referential integrity—enforcing rules about the relationships among data in different tables. Triggers go into effect when a user attempts to modify data with an `insert`, `delete`, or `update` command.

Triggers can nest to a depth of 16 levels, and can call local or remote stored procedures or other triggers.

Related Information

Triggers: Enforce Referential Integrity [page 583]

1.8.6 Defaults and Rules

Transact-SQL provides keywords for maintaining entity integrity, which ensures that a value is supplied for every column that requires one, and domain integrity, which ensures that each value in a column belongs to the set of legal values for that column.

Defaults and rules define the integrity constraints that are used during data entry and modification. A default is a value linked to a particular column or datatype, and inserted by the system if no value is provided during data entry. Rules are user-defined integrity constraints linked to a particular column or datatype, and enforced at data entry time.

Related Information

Defining Defaults and Rules for Data [page 413]

1.8.7 Error Handling and set Options

Transact-SQL error-handling techniques include capturing return status from stored procedures, defining customized return values from stored procedures, passing parameters from a procedure to its caller, and retrieving reports from global variables.

`set` options customize the results display, show processing statistics, and provide other diagnostic aids for debugging your Transact-SQL programs.

The `raiserror` and `print` statements, in combination with control-of-flow language, can direct error messages to a Transact-SQL application. Developers can localize `print` and `raiserror` to use different languages.

All `set` options except `showplan` and `char_convert` take effect immediately.

See the *Reference Manual: Commands*.

1.8.8 Additional SAP ASE Extensions to SQL

SAP ASE includes additional features of Transact-SQL.

- The following extensions to SQL search conditions:
  - modulo operator (`%`)
  - negative comparison operators (`!>`
  - `!<`, and `!=`)
  - bitwise operators (`-`, `^`
  - `|`, and `&`)
  - join operators (`*=` and `=*`)

  - wildcard characters ([ ] and -)
  - `not` operator (^)
- Fewer restrictions on the `group by` clause and the `order by` clause.
- Subqueries, which can be used almost anywhere an expression is allowed.
- Temporary tables and other temporary database objects, which exist only for the duration of the current work session.
- User-defined datatypes built on SAP ASE-supplied datatypes.
- The ability to `insert` data from a table into that same table.
- The ability to extract data from one table and put it into another with the `update` command.
- The ability to remove data based on data in other tables using the join in a `delete` statement.
- A fast way to delete all rows in a specified table and reclaim the space they took up with the `truncate table` command.
- Identity columns, which provide system-generated values that uniquely identify each row within a table.
- Updates and selections through views. Unlike most other versions of SQL, Transact-SQL places no restrictions on retrieving data through views, and few restrictions on updating data through views.
- Dozens of built-in functions.
- Options to the `create index` command for fine-tuning aspects of performance determined by indexes, and controlling the treatment of duplicate keys and rows.
- Control over what happens when a user attempts to enter duplicate keys in a unique index, or duplicate rows in a table.
- Bitwise operators for use with `integer` and `bit` type columns.
- Support for `text` and `image` datatypes.
- The ability to gain access to both SAP and non-SAP databases. With Component Integration Services, you can access remote tables as if they were local, perform joins, transfer data between tables, maintain referential integrity, provide applications such as PowerBuilder® with transparent access to heterogeneous data, and use native remote server capabilities. For more information, see the *Component Integration Services Users Guide*.

Related Information

Aggregates, Grouping, and Sorting [page 285]
Subqueries: Queries Within Other Queries [page 257]
Databases and Tables [page 50]
Datatypes [page 191]
Defining Defaults and Rules for Data [page 413]
Managing Data [page 355]
Managing Data [page 355]
Managing Data [page 355]
Managing Data [page 355]
Managing Data [page 355]
Views: Limit Access to Data [page 393]
Transact-SQL Functions [page 471]
Create Indexes on Tables [page 174]

Create Indexes on Tables [page 174]
Bitwise Operators [page 36]
Datatypes [page 191]
Datatypes [page 191]

1.9 SAP ASE Login Accounts

Each SAP ASE user must have a login account that is established by a system security officer.

Login accounts have a login name, unique on that server, and a password. A login profile is applied to a set of login accounts. Login profiles define login characteristics, such as default roles or the login script associated with each login bound to the profile.

Use `sp_displaylogin` to view information about your own SAP ASE login account.

Use groups to grant and revoke permissions to more than one user at a time within a database. For example, if everyone who works in the Sales department needs access to certain tables, all of those users can be put into a group called "sales." The database owner can grant specific access permissions to that group rather than granting permissions individually. See, *Manage SAP ASE Logins and Database Users*, in the *Security Administration Guide*.

System security officers can use roles as a convenient way to grant and revoke server-wide permissions to several users simultaneously. For example, clerical staff may need to insert and select from tables in several databases, but they may not need to update them. A system security officer can define a role called "clerical_user_role" and grant the role to everyone in the clerical staff. Database object owners can then grant the required privileges to "clerical_user_role." See, *Create a User-Defined Role*, in the *Security Administration Guide*.

You can execute stored procedures on a remote SAP ASE server using remote procedure calls if you have been granted access to the remote server and an appropriate database on that server. See, *Managing Remote Servers*, in the *System Administration Guide: Volume 1*.

1.10 isql Utility

Use the standalone utility program `isql` to enter Transact-SQL statements directly from the operating system.

You must first set up an account, or login, on SAP ASE. To use `isql`, type a command similar to the following at your operating system prompt:

```
isql -U<user_name> -P<password> -S<server_name>
```

Once you are logged in, you see:

```
1>
```

> **Note**
>
> Do not use the `-P` option on the command line to access `isql`. Instead, to prevent another user seeing your password, wait for the `isql` password prompt.

Log out of `isql` by entering:

```
quit
```

or:

```
exit
```

For more information, see *isql* in the *Utility Guide*.

To connect to a non-SAP database using Component Integration Services, use the `connect to` command. See the *Component Integration Services User's Guide*. See also `connect to...disconnect` in the *Reference Manual: Commands*.

1.10.1 Default Databases

When your SAP ASE account was created, you may have been assigned a default database to which you are automatically connected when you log in.

For example, your default database might be `pubs2`, the sample database. If you were not assigned a default database, you are connected to the master database.

You can change your default database to any database that you have permission to use, or to any database that allows guests. Any user with an SAP ASE login can be a guest. To change your default database, use `alter login` or `alter login profile`, which are described in *Reference Manual: Commands*.

To change to the `pubs2` database, which is used for most examples in this manual, enter:

```
1> use pubs2
2> go
```

Enter the word "go" on a line by itself and do not precede it with blanks or tabs. It is the command terminator; it lets the server know that you have finished typing, and you are ready for your command to be executed.

In general, examples of Transact-SQL statements in this manual do not include the line prompts used by the `isql` utility, nor do they include the terminator `go`.

1.10.2 Network-Based Security Services with isql

Using network-based security services such as unified login allows you to authenticate with a security mechanism offered by a third-party provider and then log in to SAP ASE without specifying a login name or a password.

Use the `-V` option of `isql` to specify network-based user authentication such as unified login.

See *isql* in the *Utility Guide* and, *External Authentication*, in the *Security Administration Guide* for more information about the options you can specify to use network-based security.

1.11 Displaying SQL Text

`set show_sqltext` allows you to print the SQL text for ad hoc queries, stored procedures, cursors, and dynamic prepared statements.

You need not enable `set show_sqltext` before you execute the query (as you do with commands like `set showplan on`) to collect diagnostic information for a SQL session. Instead, you can enable it while the commands are running to help determine which query is performing poorly.

Before you enable `set show_sqltext`, enable `dbcc traceon` to send the command results to standard output (`stdout`):

```
dbcc traceon(3604)
```

The syntax for `set show_sqltext` is:

```
set show_sqltext {on | off}
```

For example, this enables `show_sqltext`:

```
set show_sqltext on
```

Once `set show_sqltext` is enabled, all SQL text is printed to `stdout` for each command or system procedure you enter. Depending on the command or system procedure you run, this output can be extensive.

To disable `show_sqltext`, enter:

```
set show_sqltext off
```

Restrictions for show_sqltext

- You must have the `sa_role` or `sso_role` to run `show_sqltext`.
- You cannot use `show_sqltext` to print the SQL text for triggers.
- You cannot use `show_sqltext` to show a binding variable or a view name.

2 Databases and Tables

A database stores information (data) in a set of database objects, such as tables, that relate to each other. A table is a collection of rows that have associated columns containing individual data items.

When you create databases and tables, you are deciding the organization of your data. This process is called data definition.

SAP ASE database objects include:

- Tables
- Rules
- Defaults
- Stored procedures
- Triggers
- Views
- Referential integrity constraints
- Check integrity constraints
- Functions
- Computed columns
- Partition conditions

Columns and datatypes define the type of data included in tables. Indexes describe how data is organized in tables. They are not considered database objects by SAP ASE and are not listed in `sysobjects`.

> **Note**
>
> To create databases, tables, and other database objects, as well as to execute certain commands and stored procedures, you must have the appropriate permissions.See, *Manage User Permissions*, in the *Security Administration Guide*.

2.1 Databases

A database is a collection of related tables and other database objects—views, indexes, and so on.

When you install SAP ASE, it contains these system databases:

- `master` – controls the user databases and the operation of SAP ASE as a whole.
- `sybsystemprocs` – contains the system stored procedures.
- `sybsystemdb` – contains information about distributed transactions.
- `tempdb` – stores temporary objects, including temporary tables created with the name prefix "`tempdb..`".
- `model` – is used by SAP ASE as a template for creating new user databases.

In addition, system administrators can install these optional databases:

- pubs2 – a sample database that contains data representing a publishing operation. You can use this database to test your server connections and learn Transact-SQL. Most of the examples in the SAP ASE documentation use the pubs2 database.
- pubs3 – a version of pubs2 that uses referential integrity examples. pubs3 has a table, store_employees, that uses a self-referencing column. pubs3 also includes an IDENTITY column in the sales table. Additionally, the primary keys in the pubs3 master tables use nonclustered unique indexes, and the titles table has an example of the numeric datatype.
- interpubs – similar to pubs2, but contains French and German data.
- jpubs – similar to pubs2, but contains Japanese data. Use it if you have installed the Japanese Language Module.

These optional databases are user databases. All of your data is stored in user databases. SAP ASE manages each database by means of system tables. The data dictionary tables in the master database and in other databases are considered system tables.

2.2 Create a User Database

You can create a new database if a system administrator has granted you permission to use create database. You must be using the master database when you create a new database.

In many enterprises, a system administrator creates all databases. The creator of a database is its owner. Another user who creates a database for you can use sp_changedbowner to transfer ownership of it.

The database owner is responsible for giving users access to the database and for granting and revoking certain other permissions to users. In some organizations, the database owner is also responsible for maintaining regular backups of the database and for reloading it in case of system failure. The database owner can use the setuser command to temporarily attain any other user's permissions on a database.

Because each database is allocated a significant amount of space, even if it contains only small amounts of data, you may not have permission to use create database.

The simplest form of create database is:

```
create database <database_name>
```

To create a new database called newpubs database, verify you are using the master database rather than pubs2, then enter:

```
use master
create database newpubs
drop database newpubs
use pubs2
```

A database name must be unique on SAP ASE, and must follow the rules for identifiers. SAP ASE can manage up to 32,767 databases. You can create only one database at a time. The maximum number of segments (a label that points to one or more database devices) for any database is 32.

SAP ASE creates a new database as a copy of the model database, which contains the system tables that belong in every user database.

The creation of a new database is recorded in the master database tables sysdatabases and sysusages.

See the *Reference Manual: Commands* and the *Reference Manual: Tables*.

The `with overide` option allows machines with limited space to maintain their logs on device fragments that are separate from their data. This is not recommend, but for machines with limited storage, it may be an option. For information about `with override`, see, *Creating and Managing User Databases*, the *System Administration Guide: Volume 2*.

Related Information

Naming Convention Identifiers [page 29]

2.2.1 The on Clause

Use the `on` clause to specify where to store a database and how much space, in megabytes, to allocate for the database.

If you use the keyword `default`, the database is assigned to an available database device in the pool of default database devices indicated in the `master` database table `sysdevices`. Use `sp_helpdevice` to see which devices are in the default list.

> **Note**
>
> A system administrator may have made certain storage allocations based on performance statistics and other considerations. Before creating databases, check with a system administrator.

To specify a size of 5MB for a database to be stored in this default location, use `on default = <size>`:

```
use master
create database newpubs
on default = 5
drop database newpubs
use pubs2
```

To specify a different location for the database, give the logical name of the database device where you want it stored. You can store a database on more than one database device, with different amounts of space on each.

This example creates the `newpubs` database and allocates 3MB to it on `pubsdata` and 2MB on `newdata`:

```
create database newpubs
on pubsdata = 3, newdata = 2
```

If you omit the `on` clause and the size, the database is created with 2MB of space from the pool of default database devices indicated in `sysdevices`.

A database allocation can range in size from 2MB to 2^{23}MB.

2.2.2 The log on Clause

Unless you are creating very small, noncritical databases, always use the `log on` clause with the `create database` command. Using this extension places the transaction logs on a separate database device.

Placing the logs on a separate device:

- Allows you to use `dump transaction` rather than `dump database`, thus saving time and tapes.
- Allows you to establish a fixed size for the log, keeping it from competing with other database activity for space.
- It improves performance.
- It ensures full recovery in the event of hard disk failures.

The following command places the log for `newpubs` on the logical device `pubslog`, with a size of 1MB:

```
create database newpubs
on pubsdata = 3, newdata = 2
log on pubslog = 1
```

> **Note**
>
> When you use the `log on` extension, you are placing the database transaction log on a segment named "logsegment." To add more space for an existing log, use `alter database` and, in some cases, `sp_extendsegment`. See the *Reference Manual: Commands*, *Reference Manual: Procedures*, or, *Creating and Using Segments*, in the *System Administration Guide: Volume 2* for details.

The size of the device required for the transaction log varies, according to the amount of update activity and the frequency of transaction log dumps. As a general guideline, allocate to the log between 10 and 25 percent of the space you allocate to the database.

2.2.3 for load Option

The optional `for load` clause invokes a streamlined version of `create database` that you can use only for loading a database dump.

Use the `for load` option for recovery from media failure or for moving a database from one machine to another. See the *Reference Manual: Commands* and, *Backing Up and Restoring User Databases*, in the *System Administration Guide: Volume 2*.

2.3 Choose a Database

The `use` command lets you access an existing database if you are a known user:

```
use <database_name>
```

For example, to access the `pubs2` database, enter:

```
use pubs2
```

It is likely that you are automatically connected to the `master` database when you log in to SAP ASE, so to use another database, issue the `use` command. Use `alter login` to specify the default database for a login. Only a system administrator can change the default database for another user.

2.4 Permissions Within Databases

Permissions or privileges you are granted determine the actions you can perform on databases and database objects.

Ordinarily, a system administrator or database owner sets up permissions for you, based on the kind of work you do and the functions you need. These permissions can be different for each user in an installation or database.

Determine what your permissions are by executing:

```
sp_helprotect <user_name>
```

where `<user_name>` is your SAP ASE login name.

The `pubs2` and `pubs3` databases have a guest user name in their `sysusers` system tables. The scripts that create `pubs2` and `pubs3` grant a variety of permissions to "guest."

The "guest" mechanism means that anyone who has a login on SAP ASE, that is, anyone who is listed in `master..syslogins`, has access to `pubs2` and `pub3`, and permission to create and drop such objects as tables, indexes, defaults, rules, procedures, and so on. The "guest" user name also allows you to use certain stored procedures, create user-defined datatypes, query the database, and modify the data in it.

To use the `pubs2` or `pubs3` database, issue the `use` command. SAP ASE checks whether you are listed under your own name in `pubs2.sysusers` or `pubs3..sysusers`. If not, you are admitted as a guest without any action on your part. If you are listed in the `sysusers` table for `pubs2` or `pubs3`, SAP ASE admits you as yourself, but may give you different permissions from those of "guest."

Most users can look at the system tables in the `master` database by using the "guest" mechanism. Users who are not recognized by name in the `master` database are allowed in and treated as a user named "guest." The "guest" user is added to the `master` database in the script that creates the `master` database when it is installed.

A database owner, "dbo," can add a "guest" user to any user database using `sp_adduser`. System administrators automatically become the database owner in any database they use. See, *Getting Started with Security Administration in SAP ASE*, in the *System Administration Guide: Volume 1*.

2.5 Initialize Databases Asynchronously

The `async_init` parameter for the `alter database` and `create database` commands lets you asynchronously initialize a database while it is being used.

That is, the database is immediately available when it is created or altered, not when the database initialization is complete. The initialization is transparent to the user.

Any task that uses a page of the database that is not yet initialized performs an initialization of the allocation unit on which the page resides.

The asynchronous initialization is performed by a service task that is started by the `create database` or `alter database` command. When it restarts, SAP ASE automatically starts a new service task that completes the initialization. In a clustered environment, if an instance running the service task fails or is shut down, the coordinating instance starts a new service task to complete the initialization.

Use can use the `enable async database init` configuration parameter to specify whether SAP ASE asynchronously creates or alters databases.

The `noasync_init` clause can also be use to indicate that you are extending a database, and that SAP ASE initializes the extended space synchronously.

The syntax to create databases asynchronously is:

```
create [temporary] database <database_name>
    [on {default | <database_device>} [= <size>]
. . .
    [with {override
    | default_location = "<pathname>" [,[no]async_init] }
    [for {load | proxy_update}]
```

`noasync_init` indicates the database is initialized synchronously. The syntax to alter a database asynchronously is:

```
alter database <database_name>
    [on {default | <database_device> } [= size]
. . .
    [with override [,[no]async_init]]
    [for load]
    [for proxy_update]
```

Using the `[no]async_init` clause for `create` or `alter database` overrides the settings for `enable async database init`.

2.5.1 Determine If There is Space to be Initialized

SAP ASE synchronizes a portion of the data and log segments synchronously before making the database available, allowing the initializer to work ahead of any commands that require space in the database.

However, you may occasionally see a performance impact to commands normally run against the database while SAP ASE is busy initializing the space. This occurs because a command that requires space that is not yet initialized must initialize the space before it proceeds.

Information about initialized space is stored in `sysattributes`.

To determine if there is space not yet initialized in the database (for example, if the initializer terminated prematurely and left part of the database uninitialized), issue a query similar to:

```
select lstart=object_info1, size=object_info2, segmap=object_info3
from master..sysattributes where class=42 and object=db_id("mydb")
```

```
lstart           size             segmap
 -----------     -----------  -----------
        1536         3584000            3
        5120           51200            4
```

If the query returns one or more rows, the database contains space not yet initialized (in this query, the `mydb` database). This query does not indicate if the asynchronous initialization service task is running, only that it is not finished (if it was finished, the result set would contain zero rows).

Use a query similar to the following to determine if the initializer is running on a specific database (in this query, the `test` database):

```
select spid from sysprocesses
where dbid=db_id("test") and cmd="CRDB AUINIT"
```

```
 spid
------
    22
```

SAP ASE prints this message to the error log once the asynchronous initialization service task is running:

```
Asynchronous initialization of database '<database_name>' has completed.
```

If the asynchronous initialization service task stops prematurely, SAP ASE prints this message to the error log:

```
Asynchronous database initialization terminated
prematurely for database '%.*s'. Use DBCC
DBREPAIR(%.*s, async_database_init, start) to restart
it if required as uninitialized pages will incur a small
performance penalty when they are first referenced.
```

2.5.2 Restrictions for Initializing Databases Asynchronously

You cannot initialize certain databases asynchronously, even if you explicitly use the `async_init` parameter.

- Databases such as:
  - All system databases
  - All temporary databases, system or user
  - Archive databases
  - Proxy databases
  - Any database created with the `for load` option
- These commands cannot be run in a database that it still undergoing initialization:
  - `unmount database`

- `alter database ... log off`
- You can put the database into single user mode during initialization. However, the initializer does not run while the database is in single user mode, and is automatically restarted to continue initialization when you take the database out of single user mode.

> **Note**
>
> You may notice a slight performance impact to DMLs that use the space in the database being initialized while the asynchronous initialization service task is running.

2.6 Drop Databases

Use the `drop database` command to remove a database. `drop database` deletes the database and all of its contents from SAP ASE, frees the storage space that had been allocated for it, and deletes references to it from the `master` database.

See the *Reference Manual: Commands*.

You cannot drop a database that is in use, that is, open for reading or writing by any user.

You can drop more than one database in a single command. For example:

```
drop database newpubs, newdb
```

You can remove damaged databases with `drop database`. If `drop database` does not work, use `dbcc dbrepair` to repair the damaged database before you drop it.

2.7 Change the Database Size

If a database has filled its allocated storage space, you cannot add new data or updates to it. Existing data is always preserved. If the space allocated for a database proves to be too small, the database owner can use the `alter database` command to increase it.

`alter database` permission defaults to the database owner, and cannot be transferred. You must be using the `master` database to use `alter database`.

The default increase is 2MB from the default pool of space. This statement adds 2MB to `newpubs` on the default database device:

```
alter database newpubs
```

See the *Reference Manual: Commands*.

The `on` clause in the `alter database` command is just like the `on` clause in `create database`. The `for load` clause is just like the `for load` clause in `create database` and can be used only on a database created with the `for load` clause.

To increase the space allocated for `newpubs` by 2MB on the database device `pubsdata`, and by 3MB on the database device `newdata`, type:

```
alter database newpubs
on pubsdata = 2, newdata = 3
```

When you use `alter database` to allocate more space on a device already in use by the database, all of the segments already on that device use the added space fragment. All the objects already mapped to the existing segments can now grow into the added space. The maximum number of segments for any database is 32.

When you use `alter database` to allocate space on a device that is not yet in use by a database, the `system` and `default` segments are mapped to the new device. To change this segment mapping, use `sp_dropsegment` to drop the unwanted segments from the device. See the *Reference Manual: Procedures*.

> **Note**
>
> Using `sp_extendsegment` automatically unmaps the system and default segments.

2.8 Enforce Data Integrity in Databases

Data integrity refers to the correctness and completeness of data within a database. To enforce data integrity, you can constrain or restrict the data values that users can insert, delete, or update in the database.

For example, the integrity of data in the `pubs2` and `pubs3` databases requires that a book title in the `titles` table must have a publisher in the `publishers` table. You cannot insert books that do not have a valid publisher into `titles`, because it violates the data integrity of `pubs2` or `pubs3`.

Transact-SQL provides several mechanisms for integrity enforcement in a database such as rules, defaults, indexes, and triggers. These mechanisms allow you to maintain these types of data integrity:

- Requirement – requires that a table column must contain a valid value in every row; it cannot allow null values. The `create table` statement allows you to restrict null values for a column.
- Check or validity – limits or restricts the data values inserted into a table column. You can use triggers or rules to enforce this type of integrity.
- Uniqueness – no two table rows can have the same non-null values for one or more table columns. You can use indexes to enforce this integrity.
- Referential – data inserted into a table column must already have matching data in another table column or another column in the same table. A single table can have up to 192 references.

As an alternative to using rules, defaults, indexes, and triggers, Transact-SQL provides a series of integrity constraints as part of the `create table` statement to enforce data integrity as defined by the SQL standards.

Related Information

Transactions: Maintain Data Consistency and Recovery [page 632]

2.9 quiesce database Command

The `quiesce database` command suspends and resumes updates to a specified list of databases.

```
quiesce database
```

This command both suspends and resumes updates to a specified list of databases. See the *Reference Manual: Commands* and *Suspending and Resuming Updates to Databases* in the *System Administration Guide: Volume 2*.

2.10 Tables

When you create a table, you name its columns and supply a datatype for each column. You can also specify whether a particular column can hold null values, or specify integrity constraints for columns in the table.

The `create table` command builds a new table in the currently open database.

There can be as many as 2,000,000,000 tables per database.

The limits for the length of object names or identifiers are 255 bytes for regular identifiers, and 253 bytes for delimited identifiers. This limit applies to most user-defined identifiers, including table name, column name, index name and so on.

For variables, "@" counts as 1 byte, and names can be up to 254 bytes long.

The maximum number of columns in a table depends on many factors, including, your server's logical page size and whether the tables are configured for allpages or data-only locking.

Use the `create table` command to define each column in a table.

`create table` also:

- Provides the column name and datatype and specifies how each column handles null values.
- Specifies which column, if any, has the IDENTITY property.
- Defines column-level integrity constraints and table-level integrity constraints. Each table definition can have multiple constraints per column and per table.

For example, the `create table` statement for the `titles` table in the `pubs2` database is:

```
create table titles
(title_id tid,
title varchar(80) not null,
type char(12),
pub_id char(4) null,
price money null,
advance money null,
royalty int null,
total_sales int null,
notes varchar(200) null,
pubdate datetime,
contract bit not null)
```

See the *Reference Manual: Commands*.

> **Note**
>
> The `on <segment_name>` extension to `create table` allows you to place your table on an existing segment. `<segment_name>` points to a specific database device or a collection of database devices. Before creating a table on a segment, see a system administrator or the database owner for a list of segments that you can use. Certain segments may be allocated to specific tables or indexes for performance reasons, or for other considerations.

Examples of Creating Tables

If you use these examples, be sure you first created a sample database first (such as `newpubs`) otherwise these changes will affect another database, like `pubs2` or `pubs3`.

The simplest form of `create table` is:

```
create table <table_name>
(<column_name> <datatype>)
```

For example, to create a table named `names` with one column named "some_name," and a fixed length of 11 bytes, enter:

```
create table names
(some_name char(11))
drop table names
```

If you have `set quoted_identifier on`, both the table name and the column names can be delimited identifiers. Column names must be unique within a table, but you can use the same column name in different tables in the same database.

There must be a datatype for each column. The word "char" after the column name in the example above refers to the datatype of the column—the type of value that column will contain.

The number in parentheses after the datatype determines the maximum number of bytes that can be stored in the column. You give a maximum length for some datatypes. Others have a system-defined length.

Put parentheses around the list of column names, and commas after each column definition. The last column definition does not need a comma after it.

> **Note**
>
> You cannot use a variable in a default if the default is part of a `create table` statement.

For complete documentation of `create table`, see the *Reference Manual: Commands*.

2.11 Designing and Creating a Table

Use the `create table` statement to create a practice table. If you do not have `create table` permission, see a system administrator or the owner of the database in which you are working.

Context

You can create a table, input some data, and work with it for a while before you create indexes, defaults, rules, triggers, or views. This allows you to see what kind of transactions are most common and what kind of data is frequently entered.

However, it is more efficient to design a table and the components that go with it at the same time. You might find it easiest to sketch your plans on paper before you actually create a table and its accompanying objects.

Procedure

1. Decide what columns you need in the table, and the datatype, length, precision, and scale, for each.
2. Create any new user-defined datatypes *before* you define the table where they are to be used.
3. Decide which column, if any, should be the IDENTITY column.
4. Determine which columns can and cannot accept null values.
5. Decide what integrity constraints or column defaults, if any, you need to add to the columns in the table. This includes deciding when to use column constraints and defaults instead of defaults, rules, indexes, and triggers.
6. Decide whether you need defaults and rules, and if so, where and what kind. Consider the relationship between the NULL and NOT NULL status of a column, and defaults and rules.
7. Decide what kind of indexes you need and where.

Results

Create the table and its associated objects:

1. Create the table and its indexes using `create table` and `create index`.
2. Create defaults and rules using `create default` and `create rule`.
3. Bind any defaults and rules using `sp_bindefault` and `sp_bindrule`. Any existing defaults or rules on a user-defined datatype already used in a `create table` statement, are automatically used.
4. Create triggers using `create trigger`.
5. Create views using `create view`.

Related Information

Create Indexes on Tables [page 174]
Defining Defaults and Rules for Data [page 413]
Stored Procedures [page 498]
Triggers: Enforce Referential Integrity [page 583]
Views: Limit Access to Data [page 393]

2.11.1 Table Names

Table names must be unique for each user.

You can create temporary tables either by preceding the table name in a `create table` statement with a pound sign (#), or by specifying the name prefix "`tempdb`".

You can use any tables or other objects that you have created without qualifying their names. You can also use objects created by the database owner without qualifying their names, as long as you have the appropriate permissions on them. These rules apply to all users, including the system administrator and the database owner.

Different users can create tables of the same name. For example, a user named "jonah" and a user named "sally" can each create a table named `info`. Users who have permissions on both tables must qualify them as `jonah.info` and `sally.info`. Sally must qualify references to Jonah's table as `jonah.info`, but she can refer to her own table simply as `info`.

Related Information

Temporary Tables Usage [page 68]

2.11.2 Create the User-Defined Datatypes

Before you create a table, create any user-defined datatypes.

The same is true of the `p#` datatype for the `phone` column:

```
execute sp_addtype nm, "varchar(30)"
execute sp_addtype p#, "char(10)"
```

The first two columns used in the sample table design are for the personal (first) name and surname. They are defined as the `nm` datatype.

The `nm` datatype allows for a variable-length character entry with a maximum of 30 bytes. The `p#` datatype allows for a `char` datatype with a fixed-length size of 10 bytes.

2.11.3 Choose Columns That Accept Null Values

Except for columns that are assigned user-defined datatypes, each column has an explicit NULL or NOT NULL entry. You do not need to specify NOT NULL in the table definition, because it is the default.

The sample table design specifies NOT NULL explicitly, for readability.

The NOT NULL default means that an entry is required for that column, for example, for the two name columns in this table. The other data is meaningless without the names. In addition, the `gender` column must be NOT NULL because you cannot use NULL with `bit` columns.

If a column is designated NULL and a default is bound to it, the default value, rather than NULL, is entered when no other value is given on input. If a column is designated NULL and a rule is bound to it that does not specify NULL, the column definition overrides the rule when no value is entered for the column. Columns can have both defaults and rules.

Related Information

Defining Defaults and Rules for Data [page 413]

2.11.4 Sample Table Design Sketch

A table called `friends_etc` is used to illustrate how to create indexes, defaults, rules, triggers, and so forth.

`friends_etc` hold names, addresses, telephone numbers, and personal information. It does not define any column defaults or integrity constraints.

If another user has already created the `friends_etc` table, check with a system administrator or the database owner if you plan to follow the examples and create the objects that go with `friends_etc`. The owner of `friends_etc` must drop its indexes, defaults, rules, and triggers so that there is no conflict when you create these objects.

This table shows the proposed structure of the `friends_etc` table and the indexes, defaults, and rules that go with each column.

Column	Datatype	Null?	Index	Default	Rule
`pname`	`nm`	NOT NULL	`nmind`(composite)		
`sname`	`nm`	NOT NULL	`nmind`(composite)		
`address`	`varchar(30)`	NULL			
`city`	`varchar(30)`	NOT NULL		`citydflt`	
`state`	`char(2)`	NOT NULL		`statedflt`	

Column	Datatype	Null?	Index	Default	Rule
zip	char(5)	NULL	zipind	zipdflt	ziprule
phone	p#	NULL			phonerule
age	tinyint	NULL			agerule
bday	datetime	NOT NULL		bdflt	
gender	bit	NOT NULL		gndrdflt	
debt	money	NOT NULL		gndrdflt	
notes	varchar(255)	NULL			

2.11.4.1 Define the Sample Table

Write the `create table` statement for the sample table called `friends_etc`.

For example:

```
create table friends_etc
(pname        nm             not null,
sname         nm             not null,
address       varchar(30)    null,
city          varchar(30)    not null,
state         char(2)        not null,
postalcode    char(5)        null,
phone         p#             null,
age           tinyint        null,
bday          datetime       not null,
gender        bit            not null,
debt          money          not null,
notes         varchar(255)   null)
```

You have now defined columns for the personal name and surname, address, city, state, postal code, telephone number, age, birthday, gender, debt information, and notes. Later, you will create rules, defaults, indexes, triggers, and views for this table.

2.12 Create Tables in Different Databases

You can create a table in a database other than the current one by qualifying the table name with the name of the other database.

However, you must be an authorized user of the database in which you are creating the table, and you must have `create table` permission in it.

If you are using pubs2 or pubs3 and there is another database called newpubs, you can create a table called newtab in newpubs like this:

```
create table newpubs..newtab (col1 int)
```

You cannot create other database objects—views, rules, defaults, stored procedures, and triggers—in a database other than the current one.

2.13 Create New Tables from Query Results: select into

The select into command lets you create a new table based on the columns specified in the select statement's select list and the rows specified in the where clause.

The into clause is useful for creating test tables, new tables as copies of existing tables, and for making several smaller tables out of one large table.

The select and select into clauses, as well as the delete and update clauses, enable TOP functionality. The TOP option is an unsigned integer that allows you to limit the number of rows inserted in the target table. It implements compatibility with other platforms. See the *Reference Manual: Commands*.

You can use select into on a permanent table only if the select into/bulkcopy/pllsort database option is set to on. A system administrator can turn on this option using sp_dboption. Use sp_helpdb to see if this option is on.

Here is what sp_helpdb and its results look like when the select into/bulkcopy/pllsort database option is set to on. This example uses a page size of 8K.

```
sp_helpdb pubs2
```

```
name      db_size    owner     dbid created        status
--------- ---------- --------- ---- -------------- --------------
pubs2     20.0 MB     sa           4 Apr 25, 2005   select
     into/bulkcopy/pllsort, trunc log on chkpt, mixed log and data
device_fragments    size          usage         created      free kbytes
------------------- ------------- ------------- ----------   ------------
master              10.0MB        data and log  Apr 13 2005          1792
```

```
pubs_2_dev         10.0MB        data and log  Apr 13 2005         9888
device                 segment
---------------------- ----------------------
master                 default
master                 logsegment
master                 system
pubs_2_dev             default
pubs_2_dev             logsegment
pubs_2_dev             system
pubs_2_dev             seg1
pubs_2_dev             seg2
```

sp_helpdb output indicates whether the option is set to on or off.

If the select into/bulkcopy/pllsort database option is on, you can use the select into clause to build a new permanent table without using a create table statement. You can select into a temporary table, even if the select into/bulkcopy/pllsort option is not on.

> **Note**
>
> Because `select into` is a minimally logged operation, use `dump database` to back up your database following a `select into`. You cannot dump the transaction log following a minimally logged operation.

Unlike a view that displays a portion of a table, a table created with `select into` is a separate, independent entity.

The new table is based on the columns you specify in the select list, the tables you name in the `from` clause, and the rows you specify in the `where` clause. The name of the new table must be unique in the database, and must conform to the rules for identifiers.

A `select` statement with an `into` clause allows you to define a table and put data into it, based on existing definitions and data, without going through the usual data definition process.

The following example shows a `select into` statement and its results. This example creates a table called `newtable`, using two of the columns in the four-column table `publishers`. Because this statement includes no `where` clause, data from all the rows (but only the two specified columns) of `publishers` is copied into `newtable`.

```
select pub_id, pub_name
into newtable
from publishers
```

```
(3 rows affected)
```

"3 rows affected" refers to the three rows inserted into `newtable`. `newtable` looks like this:

```
select *
from newtable
```

```
pub_id  pub_name
------  ------------------------------------
0736    New Age Books
0877    Binnet & Hardley
1389    Algodata Infosystems
```

The new table contains the results of the `select` statement. It becomes part of the database, just like its parent table.

You can create a skeleton table with no data by putting a false condition in the `where` clause. For example:

```
select *
into newtable2
from publishers
where 1=2
```

```
(0 rows affected)
```

```
select *
from newtable2
```

```
pub_id     pub_name         city      state
-------    --------------    --------  ------
```

No rows are inserted into the new table, because 1 never equals 2.

You can also use `select into` with aggregate functions to create tables with summary data:

```
select type, "Total_amount" = sum(advance)
into #whatspent
from titles
group by type
```

```
(6 rows affected)
```

```
select * from #whatspent
```

```
 type         Total_amount
 ------------ ------------------------
 UNDECIDED                       NULL
 business                   25,125.00
 mod_cook                   15,000.00
 popular_comp               15,000.00
 psychology                 21,275.00
 trad_cook                   19,000.00
```

Always supply a name for any column in the `select into` result table that results from an aggregate function or any other expression. Examples are:

- Arithmetic aggregates, for example, `<amount> * 2`
- Concatenation, for example, `lname + fname`
- Functions, for example, `lower(lname)`

Here is an example of using concatenation:

```
select au_id,
    "Full_Name" = au_fname + ' ' + au_lname
into #g_authortemp
from authors
where au_lname like "G%"
```

```
(3 rows affected)
```

```
select * from #g_authortemp
```

```
 au_id       Full_Name
 ----------- -------------------------
 213-46-8915 Marjorie Green
 472-27-2349 Burt Gringlesby
 527-72-3246 Morningstar Greene
```

Because functions allow null values, any column in the table that results from a function other than `convert` or `isnull` allows null values.

Related Information

Views: Limit Access to Data [page 393]

2.13.1 Check for Errors

If a `select into` statement fails after creating a new table, the table is not automatically dropped or the first data page deallocated. This means that any rows inserted on the first page before the error occurred remain on the page. Check the value of the `<@@error>` global variable after a `select into` statement to be sure that no error occurred.

`select into` is a two-step operation. The first step creates the new table and the second step inserts the specified rows into the table. Because `select into` operations are not logged, you cannot issue them within user-defined transactions, and you cannot roll them back.

If an error occurs from a `select into` operation, use `drop table` to remove the new table, then reissue the `select into` statement.

2.14 Temporary Tables Usage

Temporary tables are created in the `tempdb` database. To create a temporary table, you must have `create table` permission in `tempdb`; this permission defaults to the database owner.

There are two kinds of temporary tables:

- Tables that can be shared among SAP ASE sessions
  Create a shareable temporary table by specifying `tempdb` as part of the table name in the `create table` statement. For example, the following statement creates a temporary table that can be shared among SAP ASE sessions:

  ```
  create table tempdb..authors
  (au_id char(11))
  drop table tempdb..authors
  ```

  SAP ASE does not change the names of temporary tables created this way. While hash temporary tables exist until the current session or scope is exited, shared temporary tables exist until they are explicitly dropped using `drop table`.
- Tables that are accessible only by the current SAP ASE session or procedure
  Create a nonshareable temporary table by specifying a pound sign (#) before the table name in the `create table` statement. For example:

  ```
  create table #authors
  (au_id char (11))
  ```

  While hash temporary tables exist until the current session or scope is exited, shared temporary tables exist until they are explicitly dropped.

If you do not use the pound sign or "`tempdb..`" before the table name, and you are not currently using `tempdb`, the table is created as a permanent table. A permanent table stays in the database until it is explicitly dropped by its owner.

This statement creates a nonshareable temporary table:

```
create table #myjobs
(task char(30),
```

```
start datetime,
stop datetime,
notes varchar(200))
```

You can use this table to keep a list of today's chores and errands, along with a record of when you start and finish, and any comments you may have. This table and its data is automatically deleted at the end of the current work session. Temporary tables are not recoverable.

You can associate rules, defaults, and indexes with temporary tables, but you cannot create views on temporary tables or associate triggers with them. You can use a user-defined datatype when creating a temporary table only if the datatype exists in `tempdb..systypes`.

To add an object to `tempdb` for the current session only, execute `sp_addtype` while using `tempdb`. To add an object permanently, execute `sp_addtype` in `model`, then restart SAP ASE so `model` is copied to `tempdb`.

2.14.1 Unique Temporary Table Names

To ensure that a temporary table name is unique for the current session, SAP ASE truncates the table name to 238 bytes, including the pound sign (#), if necessary and appends a 17-digit numeric suffix that is unique for each SAP ASE session.

The following example shows a table created as `#temptable` and stored as `#temptable00000050010721973`:

```
use pubs2
go
create table #temptable (task char(30))
go
use tempdb
go
select name from sysobjects where name like
    "#temptable%"
go
```

```
name
------------------------------
#temptable00000050010721973

(1 row affected)
```

2.14.2 Manipulate Temporary Tables in Stored Procedures

Stored procedures can reference temporary tables that are created during the current session.

Temporary Table Names Beginning with "#"

Temporary tables with names beginning with "#" that are created within stored procedures are not saved when the procedure exits.

A single procedure can:

- Create a temporary table
- Insert data into the table
- Run queries on the table
- Call other procedures that reference the table

1. Use `create table` to create the temporary table.
2. Create the procedures that access the temporary table, but do not create the procedure that creates the table.
3. Drop the temporary table.
4. Create the procedure that creates the table and calls the procedures created in step 2.

Temporary Table Names Beginning with tempdb..

You can use `create table <tempdb...tablename>` from inside a stored procedure to create temporary tables without the # prefix. These tables persist when the procedure completes, so they can be referenced by independent procedures.

Follow the steps for temporary table names beginning with "#" to create these tables.

> ⚠ Caution
>
> Create temporary tables with the "tempdb.." prefix from inside a stored procedure only if you intend to share the table among users and sessions. Stored procedures that create and drop a temporary table should use the "#" prefix to avoid inadvertent sharing.

2.14.3 General Rules for Temporary Tables

Temporary tables with names that begin with # are subject to certain restrictions.

The restrictions are:

- You cannot create views on these tables.
- You cannot associate triggers with these tables.
- From within a stored procedure, you cannot:
  1. Create a temporary table
  2. Drop it
  3. Create a new temporary table with the same name.
- You cannot tell which session or procedure has created these tables.

These restrictions do not apply to shareable, temporary tables that are created in `tempdb`.

Rules that apply to both types of temporary tables:

- You can associate rules, defaults, and indexes with temporary tables. Indexes created on a temporary table disappear when the temporary table disappears.

- System procedures, such as `sp_help`, work on temporary tables only if you invoke them from `tempdb`.
- You cannot use user-defined datatypes in temporary tables unless the datatypes exist in `tempdb`; that is, unless the datatypes have been explicitly created in `tempdb` since the last time SAP ASE was restarted.
- You do not have to set the `select into/bulkcopy` option `on` to select into a temporary table.

2.15 Deferred Table Creation

The `with deferred_allocation` parameter for the `create table` command lets you defer page allocation for a table. The use of deferred tables can improve performance for applications that create numerous tables, but use only a small number of them. Tables are called "deferred" until SAP ASE allocates their pages.

System tables include entries for deferred tables. These entries allow you to create objects associated with deferred tables such as views, procedures, triggers, and so on.

SAP ASE performs page allocation for deferred tables when it inserts the first row (called table materialization). Access to the table before the first `insert`, such as selects, deletes or updates, functions that report space usage, or enforce referential integrity constraints during DML on other tables, behave as if the table is empty. That is, a `select` against a deferred table produces an empty result set. Although you can create indexes on deferred tables, the page allocation for these indexes is deferred until SAP ASE materializes the table.

2.15.1 Deferred Table Creation at the Database Level

Use the `'deferred table allocation'` database option to configure the database to defer page allocation for all subsequently created user tables.

The syntax is:

```
sp_dboption <database_name>, "deferred table allocation", true
```

You cannot enable `deferred table allocation` for any system databases (such as `master`, `sybsystemprocs`, `sybsystemdb`) or temporary databases.

2.15.2 Create Deferred Tables

Use the `create table . . . with deferred_allocation` parameter to create deferred tables.

The syntax is:

```
create table <table_name> . . . with deferred_allocation
```

For example, to create a table named `im_not_here_yet`, enter:

```
create table im_not_here_yet (
col_1 int,
col_2 varchar(20)
)
with deferred_allocation
```

`sp_dboption 'deferred table allocation'` need not be enabled to create deferred tables.

Use `create table . . . with immediate_allocation` to create tables that are not deferred when `sp_dboption 'deferred table allocation'` is enabled. The syntax is:

```
create table <table_name> . . . with immediate_allocation
```

2.15.3 Explicitly Materialize Deferred Tables

Use `alter table . . . immediate_allocation` to explicitly materialize a deferred table.

The syntax is:

```
alter table <table_name> immediate_allocation
```

Once you materialize the table, SAP ASE allocates pages for all data and index partitions.

For example, to materialize the table `im_not_here_yet`, enter:

```
alter table im_not_here_yet immediate allocation
```

2.15.4 Identify Deferred Tables

`sp_help` includes information about deferred tables in the `object_status` column.

This example shows a partial `sp_help` output for the `im_not_here_yet` deferred table:

```
sp_help im_not_here_yet
```

```
Name            Owner  Object_type  Object_status        Create_date
---------------  -----  -----------  -------------------  -------------------
im_not_here_yet    dbo   user table  deferred allocation    Apr 9 2012 2:09PM
```

`sysobjects` includes the 0x80 status bit in the `sysstat3` column to indicate that a table is deferred.

2.15.5 Roll Back for Deferred Tables

If the materialization of a deferred table is part of a transaction that gets rolled back, SAP ASE does not roll back any page allocations that have been performed for the deferred table.

For example:

```
create table im_not_here_yet with deferred_allocation
go
begin tran t1
go
insert into deferred table ...
go
rollback tran t1
```

`insert` materializes the `im_not_here_yet`, table, then inserts a value. Although the `rollback tran` removes the value from the table, the page allocation is not rolled back, so the table remains materialized and is no longer a deferred table.

2.15.6 Command Behavior in Deferred Tables

Most commands work similarly on deferred tables and empty tables.

Command	Action on Deferred Table
`insert`	Materializes the table; execute `insert`
`select`	0 rows selected
`update`	0 rows affected
`delete`	0 rows affected
`alter table`	Materialize the table; execute `alter table`
`drop table`	Drop table
`create view`, `trigger` or, `procedure`	Creates view, trigger, or procedure
`create index`	Creates indexes without page allocations
`drop index`	Drops index
`reorg` subcommands	None
`update statistics`	None
`truncate table`	None
`dbcc checktable`	None

Command	Action on Deferred Table
`dbcc checkcatalog`	Skips indexes on deferred tables

2.16 IDENTITY Columns Usage

An IDENTITY column contains a value for each row, generated automatically by SAP ASE, that uniquely identifies the row within the table.

Each table can have only one IDENTITY column. You can define an IDENTITY column when you create a table with a `create table` or `select into` statement, or add it later with an `alter table` statement.

Define an IDENTITY column by specifying the keyword `identity`, instead of `null` or `not null`, in the `create table` statement. IDENTITY columns must have a datatype of `numeric` and scale of 0, or any integer type. Define the IDENTITY column with any desired precision, from 1 to 38 digits, in a new table:

```
create table <table_name>
    (<column_name> numeric(<precision> ,0) identity)
```

The maximum possible column value is $10^{precision} - 1$. For example, this command creates a table with an IDENTITY column that allows a maximum value of $10^5 - 1$, or 9999:

```
create table sales_daily
    (sale_id numeric(5,0) identity,
    stor_id char(4) not null)
```

Once an IDENTITY column reaches its maximum value, all further `insert` statements return an error that aborts the current transaction

You can create automatic IDENTITY columns by using the `auto identity` database option and the `size of auto identity` configuration parameter. To include IDENTITY columns in nonunique indexes, use the `identity in nonunique index` database option.

> **Note**
>
> By default, SAP ASE begins numbering rows with the value 1, and continues numbering rows consecutively as they are added. Some activities, such as manual insertions, deletions, or transaction rollbacks, and server shutdowns or failures, can create gaps in IDENTITY column values.

Related Information

Manage Identity Gaps in Tables [page 100]

2.16.1 Create IDENTITY Columns with User-Defined Datatypes

You can use user-defined datatypes to create IDENTITY columns. The user-defined datatype must have an underlying type of `numeric` and a scale of 0, or any integer type.

If the user-defined datatype was created with the IDENTITY property, you do not have to repeat the `identity` keyword when creating the column.

This example shows a user-defined datatype with the IDENTITY property:

```
sp_addtype ident, "numeric(5)", "identity"
```

This example shows an IDENTITY column based on the `ident` datatype:

```
create table sales_monthly
    (sale_id ident, stor_id char(4) not null)
```

If the user-defined type was created as `not null`, you must specify the `identity` keyword in the `create table` statement. You cannot create an IDENTITY column from a user-defined datatype that allows null values.

2.16.2 Reference IDENTITY Columns

When you create a table column that references an IDENTITY column, as with any referenced column, make sure it has the same datatype definition as the IDENTITY column.

For example, in the `pubs3` database, the `sales` table is defined using the `ord_num` column as an IDENTITY column:

```
create table sales
(stor_id char(4) not null
     references stores(stor_id),
ord_num numeric(6,0) identity,
date datetime not null,
unique nonclustered (ord_num))
```

The `ord_num` IDENTITY column is defined as a unique constraint, which it needs to reference the `ord_num` column in `salesdetail`. `salesdetail` is defined as follows:

```
create table salesdetail
(stor_id char(4) not null
     references storesz(stor_id),
ord_num numeric(6,0)
     references salesz(ord_num),
title_id tid not null
     references titles(title_id),
qty smallint not null,
discount float not null)
```

An easy way to insert a row into `salesdetail` after inserting a row into `sales` is to use the `<@@identity>` global variable to insert the IDENTITY column value into `salesdetail`. The `@@identity` global variable stores the most recently generated IDENTITY column value. For example:

```
begin tran
insert sales values ("6380", "04/25/97")
insert salesdetail values ("6380", @@identity, "TC3218", 50, 50)
commit tran
```

This example is in a transaction because both inserts depend on each other to succeed. For example, if the `sales` insert fails, the value of `<@@identity>` is different, resulting in an erroneous row being inserted into `salesdetail`. Because the two inserts are in a transaction, if one fails, the entire transaction is rejected.

Related Information

Retrieve IDENTITY Column Values with @@identity [page 363]
Transactions: Maintain Data Consistency and Recovery [page 632]

2.16.3 Refer to IDENTITY Columns with syb_identity

Once you have defined an IDENTITY column, you need not remember the actual column name. You can use the `syb_identity` keyword, qualified by the table name where necessary, in a `select`, `insert`, `update`, or `delete` statement on the table.

For example, this query selects the row for which `<sale_id>` equals 1:

```
select * from sales_monthly
    where syb_identity = 1
```

2.16.4 Automatically Create “hidden” IDENTITY Columns

System administrators can use the `auto identity` database option to automatically include a 10-digit IDENTITY column in new tables.

To turn this feature on in a database, use:

```
sp_dboption <database_name>, "auto identity", "true"
```

Each time a user creates a new table without specifying a primary key, a unique constraint, or an IDENTITY column, SAP ASE automatically defines an IDENTITY column. The IDENTITY column is invisible when you use `select *` to retrieve all columns from the table. You must explicitly include the column name, `SYB_IDENTITY_COL` (all uppercase letters), in the select list. If Component Integration Services is enabled, the automatic IDENTITY column for proxy tables is called `OMNI_IDENTITY_COL`.

To set the precision of the automatic IDENTITY column, use the `size of auto identity` configuration parameter. For example, to set the precision of the IDENTITY column to 15 use:

```
sp_configure "size of auto identity", 15
```

2.16.5 Using select into with IDENTITY Columns

There are special rules for using the `select into` command with tables containing IDENTITY columns.

2.16.5.1 Select an IDENTITY Column into a New Table

To select an existing IDENTITY column into a new table, include the column name (or the `syb_identity` keyword) in the `select` statement's `<column_list>`.

The syntax is:

```
select <column_list>
     into <table_name>
     from <table_name>
```

The following example creates a new table, `stores_cal_pay30`, based on columns from the `stores_cal` table:

```
select record_id, stor_id, stor_name
into stores_cal_pay30
from stores_cal
where payterms = "Net 30"
```

The new column inherits the IDENTITY property, unless any of the following conditions is true:

- The IDENTITY column is selected more than once.
- The IDENTITY column is selected as part of an expression.
- The `select` statement contains a `group by` clause, aggregate function, `union` operator, or join.

2.16.5.2 Select the IDENTITY Column More Than Once

A table cannot have more than one IDENTITY column. If an IDENTITY column is selected more than once, it is defined as NOT NULL in the new table. It does not inherit the IDENTITY property.

In the following example, the `record_id` column, which is selected once by name, and once by the `syb_identity` keyword, is defined as NOT NULL in `stores_cal_pay60`:

```
select syb_identity, record_id, stor_id, stor_name
into stores_cal_pay60
from stores_cal
where payterms = "Net 60"
```

2.16.5.3 Add a New IDENTITY Column with select into

To define a new IDENTITY column in a `select into` statement, add the column definition before the `into` clause.

The definition includes the column's precision but not its scale:

```
select <column_list>
     <identity_column_name> = identity(<precision>)
     into <table_name>
     from <table_name>
```

The following example creates a new table, `new_discounts`, from the `discounts` table and adds a new IDENTITY column, `id_col`:

```
select *, id_col=identity(5)
into new_discounts
from discounts
```

If the `<column_list>` includes an existing IDENTITY column, and you add a description of a new IDENTITY column, the `select into` statement fails.

You cannot use the `union` operator with an `identity` function in a `select into`.

2.16.5.4 Define a Column for Which the Value Must Be Computed

IDENTITY column values are generated by SAP ASE. New columns that are based on IDENTITY columns, and for which the values must be computed rather than generated, cannot inherit the IDENTITY property.

If a table's `select` statement includes an IDENTITY column as part of an expression, the resulting column value must be computed. The new column is created as NULL if any column in the expression allows a NULL value. Otherwise, it is NOT NULL.

In the following example, the `new_id` column, which is computed by adding 1000 to the value of `record_id`, is created NOT NULL:

```
select new_id = record_id + 1000, stor_name
into new_stores
from stores_cal
```

Column values are also computed if the `select` statement contains a `group by` clause or aggregate function. If the IDENTITY column is the argument of the aggregate function, the resulting column is created NULL. Otherwise, it is NOT NULL.

2.16.5.5 IDENTITY Columns Selected into Tables with Unions or Joins

The value of the IDENTITY column uniquely identifies each row in a table. However, if a table's `select` statement contains a union or join, individual rows can appear multiple times in the result set.

An IDENTITY column that is selected into a table with a union or join does not retain the IDENTITY property. If the table contains the union of the IDENTITY column and a NULL column, the new column is defined as NULL. Otherwise, it is NOT NULL.

Related Information

IDENTITY Columns Usage [page 74]
Update IDENTITY Columns [page 374]

2.17 Allow Null Values in a Column

If you omit `null` or `not null` in the `create table` statement, SAP ASE uses the null mode defined for the database (by default, NOT NULL). Use `sp_dboption` to set the `allow nulls by default` option to `true`.

You must make an entry in a column defined as NOT NULL; otherwise, SAP ASE displays an error message.

Defining columns as NULL provides a placeholder for data you may not yet know. For example, in the `titles` table, `price`, `advance`, `royalty`, and `total_sales` are set up to allow NULL.

However, `title_id` and `title` are not, because the lack of an entry in these columns would be meaningless and confusing. A price without a title makes no sense, whereas a title without a price simply means that the price has not been set yet or is not available.

In `create table`, use `not null` when the information in the column is critical to the meaning of the other columns.

Related Information

"Unknown" Values: NULL [page 248]

2.17.1 Constraints and Rules Used with Null Values

You cannot define a column to allow null values, and then override this definition with a constraint or a rule that prohibits null values. For example, if a column definition specifies NULL and a rule specifies:

```
@val in (1,2,3)
```

An implicit or explicit NULL does not violate the rule. The column definition overrides the rule, even a rule that specifies:

```
@val is not null
```

Related Information

Define Integrity Constraints for Tables [page 105]
Defining Defaults and Rules for Data [page 413]

2.17.2 Defaults and Null Values

You can use defaults, that is, values that are supplied automatically when no entry is made, with both NULL and NOT NULL columns.

A default counts as an entry. However, you cannot designate a NULL default for a NOT NULL column. You can specify null values as defaults using the `default` constraint of `create table` or using `create default`.

If you specify NOT NULL when you create a column and do not create a default for it, an error message occurs when a user fails to make an entry in that column during an insert. In addition, the user cannot insert or update such a column with NULL as a value.

This table illustrates the interaction between a column's default and its null type when a user specifies no column value, or explicitly enters a NULL value. The three possible results are a null value for the column, the default value for the column, or an error message.

Column Definition	User Entry	Result
Null and default defined	Enters no value Enters NULL value	Default used NULL used
Null defined, no default defined	Enters no value Enters NULL value	NULL used NULL used
Not null, default defined	Enters no value Enters NULL value	Default used NULL used

Column Definition	User Entry	Result
Not null, no default defined	Enters no value Enters NULL value	Error Error

Related Information

Defining Defaults and Rules for Data [page 413]

2.17.3 Nulls Require Variable-Length Datatypes

Only columns with variable-length datatypes can store null values. When you create a NULL column with a fixed-length datatype, SAP ASE converts it to the corresponding variable-length datatype. SAP ASE does not inform you of the type change.

This table lists the fixed-length datatypes and the variable-length datatypes to which SAP ASE converts them. Certain variable-length datatypes, such as `moneyn`, are reserved; you cannot use them to create columns, variables, or parameters.

Original Fixed-Length Datatype	Converted to
`char`	`varchar`
`nchar`	`nvarchar`
`unichar`	`univarchar`
`binary`	`varbinary`
`datetime`	`datetimn`
`float`	`floatn`
`bigint`, `int`, `smallint`, `tinyint`,	`intn`
`unsigned bigint`, `unsigned int`, and `unsigned smallint`	`uintn`
`decimal`	`decimaln`
`numeric`	`numericn`
`money` and `smallmoney`	`moneyn`

Data entered into `char`, `nchar`, `unichar`, and `binary` columns follows the rules for variable-length columns, rather than being padded with spaces or zeros to the full length of the column specification.

2.17.4 text, unitext, and image Columns

`text`, `unitext`, and `image` columns created with `insert` and NULL are not initialized and contain no value. They do not use storage space and cannot be accessed with `readtext` or `writetext`.

When a NULL value is written in a `text`, `unitext`, or `image` column with `update`, the column is initialized, a valid text pointer to the column is inserted into the table, and a 2K data page is allocated to the column. Once the column is initialized, it can be accessed by `readtext` and `writetext`. See the *Reference Manual: Commands*.

2.18 Alter Existing Tables

Use the `alter table` command to change the structure of an existing table.

You can:

- Add columns and constraints
- Change column default values
- Add NULL and NOT NULL columns
- Drop columns and constraints
- Change the locking scheme
- Partition or unpartition tables
- Convert column datatypes
- Convert the `null` default value of existing columns
- Increase or decrease column length

You can also change a table's partitioning attributes.

For example, by default, the `au_lname` column of the `authors` table uses a `varchar(50)` datatype. To alter the `au_lname` to use a `varchar(60)`, enter:

```
alter table authors
modify au_lname varchar(60)
```

> **Note**
>
> You cannot use a variable as the argument to a default that is part of an `alter table` statement.

Dropping, modifying, and adding non-null columns may perform a data copy, which has implications for required space and the locking scheme.

The modified table's page chains inherits the table's current configuration options (for example, if `fillfactor` is set to 50 percent, the new pages use the same `fillfactor`).

> **Note**
>
> SAP ASE performs partial logging (of page allocations) for `alter table` operations. However, because `alter table` is performed as a transaction, you cannot dump the transaction log after running `alter`

table; you must dump the database to ensure it is recoverable. If the server encounters any problems during the alter table operation, SAP ASE rolls back the transaction.

alter table acquires an exclusive table lock while it is modifying the table schema. This lock is released as soon as the command has finished.

alter table does not fire any triggers.

Related Information

Partition Tables and Indexes [page 138]
Data Copying [page 93]

2.18.1 Objects Using select * Do Not List Changes to Table

If a database has any objects (stored procedures, triggers, and so on) that perform a select * on a table from which you have dropped a column, an error message lists the missing column.

This occurs even if you create the objects using the with recompile option. For example, if you dropped the postalcode column from the authors table, any stored procedure that performed a select * on this table issues this error message:

```
Msg 207, Level 16, State 4:
Procedure 'columns', Line 2:
Invalid column name 'postalcode'.
(return status = -6)
```

This message does not appear if you add a new column and then run an object containing a select *; in this case, the new column does not appear in the output.

You must drop and re-create any objects that reference a dropped column.

2.18.2 Use alter table on Remote Tables

You can use alter table to modify a remote table using Component Integration Services (CIS).

Before you modify a remote table, make sure that CIS is running by entering:

```
sp_configure "enable cis"
```

If CIS is enabled, the output of this command is "1." By default, CIS is enabled when you install SAP ASE.

See, *Setting Configuration Parameters*, in the *System Administration Guide: Volume 1*, and the *Component Integration Services Users Guide*.

2.18.3 Add Columns

Use the `alter table` command to add a column to an existing table.

This is an example adding on a non-null column named `author_type`, which includes the constant "primary_author" as the default value, and a null column named `au_publisher` to the `authors` table.

```
alter table authors
add author_type varchar(20)
default "primary_author" not null,
au_publisher varchar(40) null
```

2.18.3.1 Add Columns Appends Column IDs

`alter table` adds a column to the table with a column ID that is one greater than the current maximum column ID.

For example, this table lists the default column IDs of the `salesdetail` table:

Column Name	stor_id	ord_num	title_id	qty	discount
Col ID	1	2	3	4	5

This command appends the `store_name` column to the end of the `salesdetail` table with a column ID of 6:

```
alter table salesdetail
add store_name varchar(40)
default
"unknown" not null
```

If you add another column, it will have a column ID of 7.

> **Note**
>
> Because a table's column IDs change as columns are added and dropped, your applications should never rely on them.

2.18.3.2 Add NOT NULL Columns

You can add a NOT NULL column to a table. This means that a constant expression, and not a null value, is placed in the column when the column is added. This also ensures that, for all existing rows, the new column is populated with the specified constant expression when the table is created.

SAP ASE issues an error message if a user fails to enter a value for a NOT NULL column.

The following adds the column `owner` to the `stores` table with a default value of "unknown:"

```
alter table stores
add owner_lname varchar(20)
default "unknown" not null
```

The default value can be a constant expression when adding NULL columns, but it can be a constant value only when adding a NOT NULL column (as in the example above).

2.18.3.3 Add Constraints

Use `alter table` to add a constraint to an existing column.

For example, to add a constraint to the `titles` table that does not allow an advance in excess of 10,000:

```
alter table titles
add constraint advance_chk
check (advance < 10000)
```

If a user attempts to insert a value greater than 10,000 into the `titles` table, SAP ASE produces an error message similar to this:

```
Msg 548, Level 16, State 1:
Line 1:Check constraint violation occurred,
dbname = 'pubs2',table name= 'titles',
constraint name = 'advance_chk'.
Command has been aborted.
```

Adding a constraint does not affect the existing data. Also, if you add a new column with a default value and specify a constraint on that column, the default value is not validated against the constraint.

Related Information

Drop Constraints [page 87]

2.18.4 Drop Columns

Use `alter table` to drop a column from an existing table.

You can drop any number of columns using a single `alter table` statement. However, you cannot drop the last remaining column from a table (for example, if you drop four columns from a five-column table, you cannot then drop the remaining column).

For example, to drop the `advance` and the `contract` columns from the `titles` table:

```
alter table titles
drop advance, contract
```

`alter table` rebuilds all indexes on the table when it drops a column.

2.18.4.1 Drop Columns Renumbers the Column ID

`alter table` renumbers column IDs when you drop a column from a table. Columns with IDs higher than the number of the dropped column move down one column ID to fill the gap that the dropped column leaves behind.

For example, the `titleauthor` table contains these column names and column IDs:

Column Name	au_id	title_id	au_ord	royaltyper
Column ID	1	2	3	4

If you drop the `au_ord` column from the table:

```
alter table titleauthor drop au_ord
```

`titleauthor` now has these column names and column IDs:

Column Name	au_id	title_id	royaltyper
Column ID	1	2	3

The `royaltyper` column now has the column ID of 3. The nonclustered index on both `title_id` and `royaltyper` are also rebuilt when `au_ord` is dropped. Also, all instances of column IDs in different system catalogs are renumbered.

Users generally will not notice the renumbering of column IDs.

> **Note**
>
> Because a table's column IDs are renumbered as columns are added and dropped, your applications should never rely on them. If you have stored procedures or applications that depend on column IDs, rewrite them so they access the correct column IDs.

2.18.4.2 Drop Columns Without Performing a Data Copy

The `no datacopy` parameter to the `alter table drop column` allows you to drop columns from a table without performing a data copy, and reduces the amount of time required for `alter table drop column` to run.

The syntax is:

```
alter table [[<database>.][<owner>].<table_name>
    {add <column_name datatype>}
. . . }
    modify <column_name>
```

```
    drop {<column_name> [, <column_name>]...
        with exp_row_size=num_bytes
            | transfer table [on | off]}
            | no datacopy
```

Instead of immediately removing the columns from the table, `no datacopy` updates the system tables, indicating the affected rows will be reformatted the next time you run `reog rebuild` or another datacopy operation (the space allocated for dropped columns (including large objects) is not freed until the next time you run `reorg rebuild`).

This example drops the `total_sales` column from the titles table without a data copy:

```
alter table titles
drop total_sales
with no datacopy
```

2.18.4.2.1 Restrictions for no datacopy Parameter

You cannot use the `no datacopy` parameter in certain scenarios.

You cannot use it in:

- Materialized or virtual computed columns
- Encrypted columns
- XML columns
- IDENTITY columns
- Java columns
- Proxy tables
- Columns using these datatypes:
  - `timestamp`
  - `bit`
- You cannot change the locking scheme of a table:
  - That has been affected by a `no datacopy` operation
  - For which you have not yet executed a `reorg rebuild` or datacopy operation since the last `drop column with no datacopy`

  You must run `reorg rebuild` before changing the locking scheme of a table.

2.18.4.3 Drop Constraints

Use `alter table` to drop a constraint.

For example:

```
alter table titles
drop constaint advance_chk
```

Related Information

Use sp_helpconstraint to Find Table Constraint Information [page 126]

2.18.5 Modify Columns

Use `alter table` to modify an existing column. You can modify any number of columns in a single `alter table` statement.

For example, this command changes the datatype of the `type` column in the `titles` table from `char`(12) to `varchar`(20) and makes it nullable:

```
alter table titles
modify type varchar(20) null
```

> ⚠ Caution
>
> You may have objects (stored procedures, triggers, and so on) that depend on a column having a particular datatype. Before you modify a column, use `sp_depends` to determine a table's dependent objects, and to make sure that any objects that reference these objects can run successfully after the modification.

2.18.5.1 Convert Datatypes

You can convert only datatypes that are either implicitly or explicitly convertible to the new datatype, or if there is an explicit conversion function in Transact-SQL.

See, *System and User-Defined Datatypes*, in the *Reference Manual: Building Blocks* for a list of the supported datatype conversions. If you attempt an illegal datatype modification, SAP ASE raises an error message and the operation is aborted.

> ℹ Note
>
> You cannot convert an existing column datatype to the `timestamp` datatype, nor can you modify a column that uses the `timestamp` datatype to any other datatype.

If you issue the same `alter table...modify` command more than once, SAP ASE issues a message similar to this:

```
Warning: ALTER TABLE operation did not affect column 'au_lname'.
Msg 13905, Level 16, State 1:
Server 'SAP1', Line 1:
Warning: no columns to drop, add or modify. ALTER TABLE 'authors' was aborted.
```

2.18.5.2 Modifying Tables and Using Bulk Copy

Modifying either the length or datatype of a column may prevent you from successfully using bulk-copy to copy in older dumps of the table.

The older table schema may not be compatible with the new table schema. Before you modify a column's length or datatype, verify that doing so does not prevent you from copying in a previous dump of the table.

2.18.5.3 Decreased Column Length May Truncate Data

If you decrease the length of a column, make sure the reduced column length does not result in truncated data.

For example, although you can use `alter table` to reduce the length of the `title` column of the `titles` table from a `varchar(80)` to a `varchar(2)`, doing so renders the data meaningless:

```
select title from titles
```

```
title
-----
Bu
Co
Co
Em
Fi
Is
Li
Ne
On
Pr
Se
Si
St
Su
Th
Th
Th
Yo
```

SAP ASE issues error messages about truncating the column data only if the `set string_rtruncation` option is turned on. If you need to truncate character data, set the appropriate string-truncation option and modify the column to decrease its length.

2.18.5.4 Modify datetime Columns

If you modify a column from a `char` datatype to `datetime`, `smalldatetime`, or `date`, you can neither specify the order that the month, day, and year appear in the output, nor specify the language used in the output. Instead, both of these settings are assigned a default value.

However, you can use `set dateformat` or `set language` to alter the output to match the setting of the information stored in the column. Also, SAP ASE does not support modifying a column from `smalldatetime` to `char` datatype. See the *Reference Manual: Commands*.

2.18.5.5 Modify the NULL Default Value of a Column

If you are changing only the NULL default value of a column, you need not specify the column's datatype.

For example, this command modifies the `address` column in the `authors` table from NULL to NOT NULL:

```
alter table authors
modify address not null
```

If you modify a column and specify the datatype as NOT NULL, the operation succeeds as long as none of the rows have NULL values. If, however, any of the rows have a NULL value, the operation fails and any incomplete transactions are rolled back. For example, the following statement fails because the `titles` table contains NULL values for the *The Psychology of Computer Cooking*:

```
alter table titles
modify advance numeric(15,5) not null
```

```
Attempt to insert NULL value into column 'advance', table 'pubs2.dbo.titles';
column does not allow nulls. Update fails.
Command has been aborted.
```

To run this command successfully, update the table to change all NULL values of the modified column to NOT NULL, then reissue the command.

2.18.5.6 Check Columns That Have Precision or Scale

Before you modify a column's scale, check the length of your data.

If an `alter table` command causes a column value to lose precision (for example, from `numeric`(10,5) to `numeric`(5,5)), SAP ASE aborts the statement. If this statement is part of a batch, the batch is aborted if the `arithabort arithignore arith_overflow` option is turned on.

If an `alter table` command causes a column value to lose scale (say from `numeric(10, 5)` to `numeric(10,3)`, the rows are truncated without warning. This occurs whether `arithabort numeric_truncation` is on or off.

If `arithignore arith_overflow` is on and `alter table` causes a numeric overflow, SAP ASE issues a warning. However, if `arithabort arithignore arith_overflow` is off, SAP ASE does not issue a warning

if `alter table` causes a numeric overflow. By default, `arithignore arith_overflow` is off when you install SAP ASE.

> **Note**
>
> Make sure you review the data truncation rules and fully understand their implications before issuing commands that may truncate the length of the columns in your production environment. First perform the commands on a set of test columns.

2.18.5.7 Modify text, unitext, and image Columns

You can modify the `text`, `unitext`, and `image` columns with certain restrictions.

`text` columns can be converted to:

- `[n]char`
- `[n]varchar`
- `unichar`
- `univarchar`

`unitext` columns can be converted to:

- `[n]char`
- `[n]varchar`
- `unichar`
- `univarchar`
- `binary`
- `varbinary`

`image` columns can be converted to:

- `varbinary`
- `binary`

You cannot modify `char`, `varchar`, `unichar`, and `univarchar` datatype columns to `text` or `unitext` columns. If you are converting from `text` or `unitext` to `char`, `varchar`, `unichar`, or `univarchar`, the maximum length of the column is governed by page size. If you do not specify a column length, `alter table` uses the default length of one byte. If you are modifying a multibyte character `text`, `unitext`, or `image` column, and you do not specify a column length that is sufficient to contain the data, SAP ASE truncates the data to fit the column length.

2.18.6 Add IDENTITY Columns

You can add IDENTITY columns only with a default value of NOT NULL. You cannot specify a default clause for a new IDENTITY column.

To add an IDENTITY column to a table, specify the `identity` keyword in the `alter table` statement:

```
alter table <table_name> add column_name
     numeric(<precision> ,0) identity not null
```

The following example adds an IDENTITY column, `record_id`, to the `stores` table:

```
alter table stores
    add record_id numeric(5,0) identity not null
```

When you add an IDENTITY column to a table, SAP ASE assigns a unique sequential value, beginning with the value 1, to each row. If the table contains a large number of rows, this process can be time consuming. If the number of rows exceeds the maximum value allowed for the column (in this case, 10^5 - 1, or 99,999), the `alter table` statement fails.

You can create IDENTITY columns with user-defined datatypes. The user-defined datatype must be a `numeric` type with a scale of 0.

2.18.7 Drop IDENTITY Columns

You can drop IDENTITY columns just like any other column.

For example:

```
alter table stores
drop record_id
```

These are the restrictions for dropping an IDENTITY column:

- If `sp_dboption "identity in nonunique index"` is turned on in the database, you must first drop all indexes, then drop the IDENTITY column, and then re-create the indexes.
  If the IDENTITY column is hidden, you must first identify it using the `syb_identity` keyword.
- To drop an IDENTITY column from a table that has `set identity_insert` turned on first, issue `sp_helpdb` to determine if `set identity_insert` is turned on.
  Next, turn off the `set identity_insert` option:

  ```
  set identity_insert table_name off
  ```

  Drop the IDENTITY column, then add the new IDENTITY column, and turn on the `set identity_insert` option:

  ```
  set identity_insert <table_name> on
  ```

Related Information

Refer to IDENTITY Columns with syb_identity [page 76]

2.18.8 Modify IDENTITY Columns

You can modify the size of an IDENTITY column to increase its range. This might be necessary if either your current range is too small, or the range has been used up because of a server shutdown.

For example, you can increase the range of `record_id` by entering:

```
alter table stores
modify record_id numeric(9,0)
```

You can decrease the range by specifying a smaller precision for the target datatype. If the IDENTITY value in the table is too large for the range of the target IDENTITY column, an arithmetic conversion is raised and `alter table` aborts the statement.

You cannot add a non-null IDENTITY column to a partitioned table using `alter table` commands that require a data copy. Data copying is performed in parallel for partitioned tables, and cannot guarantee unique IDENTITY values.

2.18.9 Data Copying

SAP ASE performs a data copy only if it must temporarily copy data out of a table before it changes the table's schema. If the table has any indexes, SAP ASE rebuilds the indexes when the data copy finishes.

> **Note**
>
> If `alter table` is performing a data copy, the database that contains the table must have `select into/bulkcopy/pllsort` turned on. See the *Reference Manual: Commands*.

SAP ASE performs a data copy when you:

- Drop a column.
- Modify any of these properties of a column:
  - The datatype (except when you increase the length of `varchar`, `varbinary`, NULL `char`, or NULL `binary` columns).
  - From NULL to NOT NULL, or vice-versa.
  - Decrease length. If you decrease a column's length, you may not know beforehand if all the data will fit in the reduced column length. For example, if you decrease `au_lname` to `varchar`(30), it may contain a name that requires a `varchar`(35). When you decrease a column's data length, SAP ASE first performs a data copy to ensure that the change in the column length is successful.
- Increase the length of a number column (for example, from `tinyint` to `int`). SAP ASE performs data copying in case one row has a NOT NULL value for this column.

- Add a NOT NULL column.

`alter table` does not perform a data copy when you change the:

- Length of either a `varchar` or a `varbinary` column.
- User-defined datatype ID, but not the physical datatype. For example, if your site has two datatypes `mychar1` and `mychar2` that have different user-defined datatypes but the same physical datatype, there is no data copy performed if you change `mychar1` to `mychar2`.
- NULL default value of a variable-length column from NOT NULL to NULL.

To identify if `alter table` performs a data copy:

1. Set `showplan` on to report whether SAP ASE will perform a data copy.
2. Set `noexec` on to ensure that no work will be performed.
3. Perform the `alter table` command if no data copy is required; only catalog updates are performed to reflect the changes made by the `alter table` command.

> **Note**
>
> See *Create Nonmaterialized, Non-Null Columns* for information about adding non NULL columns that do not require data copy.

Related Information

Create Nonmaterialized, Non-Null Columns [page 369]

2.18.9.1 Change exp_row_size

If you perform a data copy, you can also change the `exp_row_size`, which allows you to specify how much space to allow per row.

You can change the `exp_row_size` only if the modified table schema contains variable-length columns, and only to within the range specified by the `maxlen` and `minlen` values in `sysindexes` for the modified table schema. The `maxlen` and `minlen` values do not include the overhead for the row ID.

If the column has fixed-length columns, you can change the `exp_row_size` to only 0 or 1. If you drop all the variable-length columns from a table, you must specify an `exp_row_size` of 0 or 1. Also, if you do not supply an `exp_row_size` with the `alter table` command, the old `exp_row_size` is used. SAP ASE raises an error if the table contains only fixed-length columns and the old `exp_row_size` is incompatible with the modified schema.

You cannot use the `exp_row_size` clause with any of the other `alter table` subclauses (for example, defining a constraint, changing the locking scheme, and so on). You can also use `sp_chgattribute` to change the `exp_row_size`. See the *Reference Manual: Commands*.

2.18.10 Modifying Locking Schemes and Table Schema

When using `alter table` to modify data, you can also include the `lock` command to change the locking scheme of a table.

Context

For example, to modify the `au_lname` column of the `authors` table and change the locking scheme of the table from `allpages` locking to `datarows` locking:

```
alter table authors
modify au_lname varchar(10)
lock datarows
```

Procedure

1. Drop the index.
2. Modify the table schema and change the locking scheme in the same statement (if the change in the table schema also includes a data copy).
3. Rebuild the clustered index.

Results

Alternately, you can issue an `alter table` command to change the locking scheme, then issue another `alter table` command to change the table's schema.

2.18.11 Add, Drop, or Modify Columns with User-Defined Datatypes

The syntax to add, drop, or modify a column to include user-defined datatypes is the same as with a system-defined datatype.

Add a Column with User-Defined Datatypes

To add a column to the `authors` table of `pubs2` using the `usertype` datatype:

```
alter table titles
```

```
add newcolumn usertype not null
```

The NULL or NOT NULL default you specify takes precedence over the default specified by the user-defined datatype. That is, if you add a column and specify NOT NULL as the default, the new column has a default of NOT NULL even if the user-defined datatype specifies NULL. If you do not specify NULL or NOT NULL, the default specified by the user-defined datatype is used.

You must supply a default clause when you add columns that are `not null`, unless the user-defined datatype already has a default bound to it.

If the user-defined datatype specifies IDENTITY column properties (precision and scale), the column is added as an IDENTITY column.

Modify a Column with User-Defined Datatypes

To modify the `au_lname` of the `authors` table to use the user-defined `newtype` datatype:

```
alter table authors
modify au_lname newtype(60) not null
```

If you do not specify either NULL or NOT NULL as the default, columns use the default specified by the user-defined datatype.

Modifying the table does not affect any current rules or defaults bound to the column. However, if you specify new rules or defaults, any old rules or defaults bound to the user-defined datatype are dropped. If there are no previous rules or defaults bound to the column, any user-defined rules and defaults are applied.

You cannot modify an existing column to an IDENTITY column. You can only modify an existing IDENTITY column with user-defined datatypes that have IDENTITY column properties (precision and scale).

Drop a Column with User-Defined Datatypes

Drop a column with a user-defined datatype in the same way that you drop a column with a system-defined datatype.

2.18.12 Errors and Warnings from alter table

Most errors you encounter when running `alter table` inform you of schema constructs that prevent the requested command (for example, if you try to drop a column that is part of an index).

To report error conditions:

1. Set `showplan` on.
2. Set `noexec` on.
3. Perform the `alter table` command.

After you have changed the command to address any reported errors, set `showplan` and `noexec` to off so that SAP ASE actually performs the work.

All runtime data-dependent errors (for example, errors of numeric overflow, character truncation, and so on) can be identified only when the statement executes. Either change the command to fit the data available, or fix the data values to work with the required target datatypes the statement specifies. To identify these errors, run the command with `noexec` disabled.

2.18.12.1 Errors and Warnings Generated by alter table modify

Certain errors are generated only by the `alter table modify` command. Although `alter table modify` converts columns to compatible datatypes, `alter table` may issue errors if the columns you are converting have certain restrictions.

> **Note**
>
> Make sure you understand the implications of modifying a datatype before you issue the command. Generally, use `alter table modify` only to implicitly convert between convertible datatypes. This ensures that any hidden conversions required during processing of `insert` and `update` statements do not fail because of datatype incompatibility.

For example, if you add a `second_advance` column to the `titles` table with a datatype of `int`, and create a clustered index on `second_advance`, you cannot then modify this column to a `char` datatype. This would cause the `int` values to be converted from integers (1, 2, 3) to strings ('1', '2', '3'). When the index is rebuilt with sorted data, the data values are expected to be in sorted order. But in this example, the datatype has changed from `int` to `char` and is no longer in sorted order for the `char` datatype's ordering sequence. So, the `alter table` command fails during the index rebuild phase.

Be very cautious when choosing a new datatype for columns that are part of index-key columns of clustered indexes. `alter table modify` must specify a target datatype that will not violate the ordering sequence of the modified data values after its data copy phase.

`alter table modify` also issues a warning message if you modify the datatype to an incompatible datatype in a column that contains a constraint. For example, if you try to modify from datatype `char` to datatype `int`, and the column includes a constraint, `alter table modify` issues this warning:

```
Warning: a rule or constraint is defined on column 'new_col' being modified.
Verify the validity of rules and constraints after this ALTER TABLE operation.
```

The `modify` operation is flexible, but must be used with caution. In general, modifying to an implicitly convertible datatype works without errors. Modifying to an explicitly convertible datatype may lead to inconsistencies in the table schema. Use `sp_depends` to identify all column-level dependencies before modifying a column's datatype.

2.18.12.2 Scripts Generated by if exists()...alter table

Scripts that include certain constructs may produce errors if the table described in the script does not include the specified column.

For example:

```
if exists (select 1 from syscolumns
    where id = object_id("some_table")
        and name = "some_column")
    begin
        alter table some_table drop some_column
    end
```

In this example, `some_column` must exist in `some_table` for the batch to succeed.

If `some_column` exists in `some_table`, the first time you run the batch, `alter table` drops the column. On subsequent executions, the batch does not compile.

SAP ASE raises these errors while preprocessing this batch, which are similar to those that are raised when a normal `select` tries to access a nonexistent column. These errors are raised when you modify a table's schema using clauses that require a data copy. If you add a null column, and use the above construct, SAP ASE does not raise these errors.

To avoid such errors when you modify a table's schema, include `alter table` in an `execute immediate` command:

```
if exists (select 1 from syscolumns
       where id = object_id("some_table")
         and name = "some_column")
begin
    exec ("alter table some_table drop
        some_column")
end
```

Because the `execute immediate` statement is run only if the `if exists()` function succeeds, SAP ASE does not raise any errors when it compiles this script.

You must also use the `execute immediate` construct for other uses of `alter table`, for example, to change the locking scheme, and for any other cases when the command does not require data copy.

2.18.13 Rename Tables and Other Objects

Use `sp_rename` to rename tables and other database objects: columns, constraints, datatypes, views, indexes, rules, defaults, procedures, and triggers.

You must own an object to rename it. You cannot change the name of system objects or system datatypes. The database owner can change the name of any user's objects. Also, the object for which you are changing the name must be in the current database.

To rename the database, use `sp_renamedb`. See the *Reference Manual: Procedures*.

For example, to change the name of `friends_etc` to `infotable`:

```
sp_rename friends_etc, infotable
```

To rename a column, use:

```
sp_rename "<table.column>", <newcolumnname>
```

Do not include the table name prefix in the new column name, or the new name is not accepted.

To change the name of an index, use:

```
sp_rename "<table.index>", <newindexname>
```

Do not include the table name in the new name.

To change the name of the user datatype `tid` to `t_id`, use:

```
exec sp_rename tid, "t_id"
```

2.18.13.1 Rename Dependent Objects

When you rename objects, you must also change the text of any dependent procedure, trigger, or view, to reflect the new object name.

The original object name continues to appear in query results until you change the name of, and compile the procedure, trigger, or view. The safest course is to change the definitions of any dependent objects when you execute `sp_rename`. You can use `sp_depends` to get a list of dependent objects.

You can use the `defncopy` utility program to copy the definitions of procedures, triggers, rules, defaults, and views into an operating system file. Edit this file to correct the object names, then use `defncopy` to copy the definition back into SAP ASE. See the *Utility Guide*.

2.19 Drop Tables

Use `drop table` to remove specified tables from the database, together with their contents and all indexes and privileges associated with them. Rules or defaults that are bound to the table are no longer bound, but are otherwise not affected.

You must be the owner of a table to drop it. However, no one can drop a table while it is being read or written to by a user or application. You cannot use `drop table` on any system tables, either in the `master` database or in a user database.

You can drop a table in another database if you are the table owner.

If you `delete` all the rows in a table or use `truncate table` on it, the table exists until you `drop` it.

`drop table` and `truncate table` permission cannot be transferred to other users.

2.20 Manage Identity Gaps in Tables

The IDENTITY column contains a unique ID number, generated by SAP ASE, for each row in a table.

Because of the way the server generates ID numbers by default, you may occasionally have large gaps in the ID numbers. The `identity_gap` parameter gives you control over ID numbers, and potential gaps in them, for a specific table.

By default, SAP ASE allocates a block of ID numbers in memory instead of writing each ID number to disk as it is needed, which requires more processing time. The server writes the highest number of each block to the table's object allocation map (OAM) page. This number is used as the starting point for the next block after the currently allocated block of numbers is used or "burned." The other numbers of the block are held in memory, but are not saved to disk. Numbers are considered burned when they are allocated to memory, then deleted from memory, either because they were assigned to a row, or because they were erased from memory due to some abnormal occurrence such as a system failure.

Allocating a block of ID numbers improves performance by reducing contention for the table. However, if the server fails or is shut down with `no wait` before all the ID numbers are assigned, the unused numbers are burned. When the server is running again, it starts numbering with the next block of numbers based on the highest number of the previous block that the server wrote to disk. Depending on how many allocated numbers were assigned to rows before the failure, you may have a large gap in the ID numbers.

Identity gaps can also result from dumping and loading an active database. When dumping, database objects are saved to the OAM page. If an object is currently being used, the `maximum used identity value` is not in the OAM page and, therefore, is not dumped.

2.20.1 Parameters for Controlling Identity Gaps

SAP ASE provides parameters that allow you to control gaps in identity numbers.

Parameter name	Scope	Used with	Description
`identity_gap`	Table-specific	`create table` or `select into`	Creates ID number blocks of a specific size for a specific table. Overrides `identity burning set factor` for the table. Works with `identity grab size`.

Parameter name	Scope	Used with	Description
`identity burning set factor`	Server-wide	`sp_configure`	Indicates a percentage of the total available ID numbers to allocate for each block. Works with `identity grab size`. If the `identity_gap` for a table is set to 1 or higher, `identity burning set factor` has no effect on that table. The burning set factor is used for all tables for which `identity_gap` is set to 0. When you set `identity burning set factor`, express the number in decimal form, and then multiply it by 10,000,000 (10^7) to get the correct value to use with `sp_configure`. For example, to release 15 percent (.15) of the potential IDENTITY column values at one time, specify a value of .15 times 10^7 (or 1,500,000): `sp_configure "identity burning set factor", 1500000`
`identity grab size`	Server-wide	`sp_configure`	Reserves a block of contiguous ID numbers for each process. Works with `identity burning set factor` and `identity_gap`.

2.20.2 Comparison of identity burning set factor and identity_gap

The `identity_gap` parameter controls the size of identity gaps for a particular table.

For example, if you create a table named `books` that includes all the books in a bookstore, each book must have a unique ID number, which SAP ASE automatically generates. `books` includes an IDENTITY column that uses the default numeric value of (18, 0), providing a total of 999,999,999,999,999,999 ID numbers. The `identity burning set factor` configuration parameter uses the default setting of 5000 (.05 percent of 999,999,999,999,999,999), which means that SAP ASE allocates blocks of 500,000,000,000,000 numbers.

The server allocates the first 500,000,000,000,000 numbers in memory and stores the highest number of the block (500,000,000,000,000) on the table's OAM page. When all the numbers are assigned to rows or burned, SAP ASE takes the next block of numbers (the next 500,000,000,000,000), starting with 500,000,000,000,001, and stores the number 1,000,000,000,000,000 as the highest number of the block.

If the server fails after row number 500,000,000,000,022, only numbers 1 through 500,000,000,000,022 were used as ID numbers for `books`. Numbers 500,000,000,000,023 through 1,000,000,000,000,000 are burned. When SAP ASE starts again, it creates ID numbers starting from the highest number stored on the table's OAM page plus one (1,000,000,000,000,001), which leaves a gap of 499,999,999,999,978 ID numbers.

Reduce the Identity Number Gap

Creating the `books` table with an `identity_gap` value of 1000, overrides the server-wide `identity burning set factor` setting that resulted in blocks of 500,000,000,000,000 ID numbers. Instead, ID numbers are allocated in memory in blocks of 1000

The server allocates the first 1000 numbers and stores the highest number of the block (1000) to disk. When all the numbers are used, SAP ASE takes the next 1000 numbers, starting with 1001, and stores the number 2000 as the highest number.

If SAP ASE fails after row number 1002, it uses the numbers 1000 through 1002: numbers 1003 through 2000 are lost. When you restart SAP ASE, it creates ID numbers starting from the highest number stored on the table's OAM page plus one (2000), which leaves a gap of only 998 numbers.

You can significantly reduce the gap in ID numbers by setting the `identity_gap` for a table instead of using the server-wide `table burning set factor`. However, if you set this value too low, each time the server must write the highest number of a block to disk, which affects performance. For example, if `identity_gap` is set to 1, which means you are allocating one ID number at a time, the server must write the new number every time a row is created, which may reduce performance because of page lock contention on the table. You must find the best setting to achieve the optimal performance with the lowest gap value acceptable for your situation.

2.20.3 Set the Table-Specific Identity Gap

Set the table-specific identity gap when you create a table using either `create table` or `select into`.

This statement creates a table named `mytable` with an `identity` column:

```
create table mytable (IdNum numeric(12,0) identity)
with identity_gap = 10
```

The identity gap is set to 10, which means ID numbers are allocated in memory in blocks of ten. If the server fails or is shut down with no wait, the maximum gap between the last ID number assigned to a row and the next ID number assigned to a row is ten numbers.

If you are creating a table in a `select into` statement from a table that has a specific identity gap setting, the new table does not inherit the identity gap setting from the parent table. Instead, the new table uses the `identity burning set factor` setting. To give the new table a specific `identity_gap` setting, specify the identity gap in the `select into` statement. You can give the new table an identity gap that is the same as or different from the parent table.

For example, to create a new table (`newtable`) from the existing table (`mytable`) with an identity gap:

```
select IdNum into newtable
with identity_gap = 20
from mytable
```

2.20.4 Change the Table-Specific Identity Gap

To change the identity gap for a specific table, use `sp_chgattribute`.

```
sp_chgattribute "<table_name>", "identity_gap", <set_number>
```

For example:

```
sp_chgattribute "mytable", "identity_gap", 20
```

To change `mytable` to use the `identity burning set factor` setting instead of the `identity_gap` setting, set `identity_gap` to `0`:

```
sp_chgattribute "mytable", "identity_gap", 0
```

See the *Reference Manual: Procedures*.

2.20.5 Display Table-Specific Identity Gap Information

To see the `identity_gap` setting for a table, use `sp_help`.

For example, the zero value in the `identity_gap` column (towards the end of the output) indicates that no table-specific identity gap is set. `mytable` uses the server-wide `identity burning set factor` value.

```
sp_help mytable
```

```
Name    Owner    Object_type    Create_date
------- -------- -------------- ------------------------------
mytable      dbo     user table            Nov 29 2004 1:30PM
(1 row affected)
. . .
exp_row_size reservepagegap fillfactor max_rows_per_page identity_gap
------------ -------------- ---------- ----------------- ------------
           1              0          0                 0            0
```

If you change the `identity_gap` of `mytable` to 20, `sp_help` output for the table shows 20 in the `identity_gap` column. This setting overrides the server-wide `identity burning set factor` value.

```
sp_help mytable
```

```
Name    Owner    Object_type    Create_date
------- -------- -------------- ------------------------------
mytable dbo      user table     Nov 29 2004 1:30PM
(1 row affected)
. . .
exp_row_size reservepagegap fillfactor max_rows_per_page identity_gap
------------ -------------- ---------- ----------------- ------------
           1              0          0                 0           20
```

2.20.6 Gaps from Other Causes

Manually inserting values into the IDENTITY column, deleting rows, setting the `identity grab size` value, and rolling back transactions can create gaps in IDENTITY column values. Setting the `identity burning set factor` does not affect these gaps.

For example, assume that you have an IDENTITY column with these values:

```
select syb_identity from stores_cal
```

```
 id_col
 -------
      1
      2
      3
      4
      5
```

You can delete all rows for which the IDENTITY column falls between 2 and 4, leaving gaps in the column values:

```
delete stores_cal
where syb_identity between 2 and 4
select syb_identity from stores_cal
```

```
id_col
------
      1
      5
```

After setting `identity_insert on` for the table, the table owner, database owner, or system administrator can manually insert any legal value greater than 5. For example, inserting a value of 55 would create a large gap in IDENTITY column values:

```
insert stores_cal
(syb_identity, stor_id, stor_name)
values (55, "5025", "Good Reads")
select syb_identity from stores_cal
```

```
id_col
 -------
      1
      5
     55
```

If `identity_insert` is then set to `off`, SAP ASE assigns an IDENTITY column value of 55 + 1, or 56, for the next insertion. If the transaction that contains the `insert` statement is rolled back, SAP ASE discards the value 56 and uses a value of 57 for the next insertion.

2.20.7 IDENTITY Column Maximum Value

The maximum number of rows you can insert into a table depends on the precision that is set for the IDENTITY column.

If a table reaches that limit, you can either re-create the table with a larger precision or, if the table's IDENTITY column is not used for referential integrity, use `bcp` to remove the gaps.

Related Information

Maximum Value of the IDENTITY Column [page 365]

2.21 Define Integrity Constraints for Tables

To maintain data integrity in a database, you can either define rules, defaults, indexes, and triggers; or, you can define `create table` integrity constraints.

The method select depends on your requirements. Integrity constraints offer the advantages of defining integrity controls in one step during the table creation process (as defined by the SQL standards) and of simplifying the process to create those integrity controls. However, integrity constraints are more limited in scope and less comprehensive than defaults, rules, indexes, and triggers.

For example, triggers provide more complex handling of referential integrity than those declared in `create table`. The integrity constraints defined by a `create table` are specific to that table; you cannot bind them to other tables, and you can only drop or change them using `alter table`. Constraints cannot contain subqueries or aggregate functions, even on the same table.

The two methods are not mutually exclusive. You can use integrity constraints along with defaults, rules, indexes, and triggers. This gives you the flexibility to choose the best method for your application. This section describes the `create table` integrity constraints. Defaults, rules, indexes, and triggers are described in later chapters.

You can create these types of constraints:

- `unique` and `primary key` constraints require that no two rows in a table have the same values in the specified columns. In addition, a `primary key` constraint does not allow a null value in any row of the column.
- Referential integrity (`references`) constraints require that data being inserted in specific columns already has matching data in the specified table and columns. Use `sp_helpconstraint` to find a table's referenced tables.
- `check` constraints limit the values of data inserted into columns.

You can also enforce data integrity by restricting the use of null values in a column (the `null` or `not null` keywords) and by providing default values for columns (the `default` clause).

> **Caution**
>
> Do not define or alter the definitions of constraints for system tables.

Related Information

Allow Null Values in a Column [page 79]
Use sp_helpconstraint to Find Table Constraint Information [page 126]

2.21.1 Table and Column Level Constraints

You can declare integrity constraints at the table or column level.

Although the difference is rarely noticed by users, column-level constraints are verified only if a value in the column is being modified, while the table-level constraints are verified whenever there is any modification to a row, regardless of whether or not it changes the column in question.

Place column-level constraints after the column name and datatype, but before the delimiting comma. Enter table-level constraints as separate comma-delimited clauses. SAP ASE treats table-level and column-level constraints the same way; both ways are equally efficient

However, you must declare constraints that operate on more than one column as table-level constraints. For example, the following `create table` statement has a `check` constraint that operates on two columns, `pub_id` and `pub_name`:

```
create table my_publishers
(pub_id      char(4),
pub_name     varchar(40),
constraint my_chk_constraint
    check (pub_id in ("1389", "0736", "0877")
        or pub_name not like "Bad News Books"))
```

You can optionally declare constraints that operate on a single column as column-level constraints. For example, if the above `check` constraint uses only one column (`pub_id`), you can place the constraint on that column:

```
create table my_publishers
(pub_id     char(4) constraint my_chk_constraint
        check (pub_id in ("1389", "0736", "0877")),
pub_name     varchar(40))
```

On either column-level or table-level constraints, the `constraint` keyword and accompanying `<constraint_name>` are optional.

> **Note**
>
> You cannot issue `create table` with a check constraint and then insert data into the table in the same batch or procedure. Either separate the `create` and `insert` statements into two different batches or procedures, or use `execute` to perform the actions separately.

Related Information

Check Constraints [page 107]

2.21.2 Create Error Messages for Constraints

You can use `sp_addmessage` to create error messages, then use `sp_bindmsg` to bind them to constraints.

For example:

```
sp_addmessage 25001,
     "The publisher ID must be 1389, 0736, or 0877"
sp_bindmsg my_chk_constraint, 25001
insert my_publishers values
     ("0000", "Reject This Publisher")
```

```
Msg 25001, Level 16, State 1:
Server 'snipe', Line 1:
The publisher ID must be 1389, 0736, or 0877
Command has been aborted.
```

To change the message for a constraint, bind a new message. The new message replaces the old message.

Unbind messages from constraints using `sp_unbindmsg`; drop user-defined messages using `sp_dropmessage`.

For example:

```
sp_unbindmsg my_chk_constraint
sp_dropmessage 25001
```

To change the text of a message but keep the same error number, unbind it, drop it with `sp_dropmessage`, add it again with `sp_addmessage`, and bind it with `sp_bindmsg`.

2.21.3 Check Constraints

You can declare a `check` constraint to limit the values users insert into a column in a table.

Check constraints are useful for applications that check a limited, specific range of values. A `check` constraint specifies a `<search_condition>` that any value must pass before it is inserted into the table. A `<search_condition>` can include:

- A list of constant expressions introduced with `in`
- A range of constant expressions introduced with `between`
- A set of conditions introduced with `like`, which may contain wildcard characters

An expression can include arithmetic operations and Transact-SQL built-in functions. The `<search_condition>` cannot contain subqueries, a set function specification, or a target specification.

For example, this statement ensures that only certain values can be entered for the `pub_id` column:

```
create table my_new_publishers
(pub_id      char(4)
        check (pub_id in ("1389", "0736", "0877",
             "1622", "1756")
        or pub_id like "99[0-9][0-9]"),
pub_name     varchar(40),
city         varchar(20),
state        char(2))
```

Column-level check constraints can reference only the column on which the constraint is defined; they cannot reference other columns in the table. Table-level check constraints can reference any columns in the table. `create table` allows multiple check constraints in a column definition.

Because check constraints do not override column definitions, you cannot use a check constraint to prohibit null values if the column definition permits them. If you declare a check constraint on a column that allows null values, you can insert NULL into the column, implicitly or explicitly, even though NULL is not included in the `<search_condition>`. For example, suppose you define the following check constraint on a table column that allows null values:

```
check (pub_id in ("1389", "0736", "0877", "1622", "1756"))
```

You can still insert NULL into that column. The column definition overrides the check constraint because the following expression always evaluates to true:

```
col_name != null
```

When you create a check constraint, source text, which describes the check constraint, is stored in the `text` column of the `syscomments` system table.

> ⚠ **Caution**
>
> Do not remove this information from `syscomments`; doing so can cause problems for future upgrades of SAP ASE.

If you have security concerns, encrypt the text in `syscomments` by using `sp_hidetext`, described in the *Reference Manual: Procedures*.

2.21.4 Default Column Values

Before you define any column-level integrity constraints, you can use the `default` clause to assign a default value to a column as part of the `create table` statement. When you do not enter a value for a column, the default value is inserted.

You can use the following values with the `default` clause:

- `<constant_expression>` – specifies a constant expression to use as a default value for the column. The constant expression cannot include the name of any columns or other database objects, but you can include built-in functions that do not reference database objects. This default value must be compatible with the datatype of the column.

- `user` – specifies that the user name is inserted as the default. The datatype of the column must be either `char(30)` or `varchar(30)` to use this default.
- `null` – specifies that the null value is inserted as the default. You cannot use the not null keyword to define this default for columns that do not allow null values.

For example, this `create table` statement defines two column defaults:

```
create table my_titles
(title_id        char(6),
title            varchar(80),
price            money       default null,
total_sales      int         default 0)
```

You can include only one `default` clause per column in a table.

Using the `default` clause to assign defaults is simpler than the two-step Transact-SQL method. In Transact-SQL, you can use `create default` to declare the default value, and then use `sp_bindefault` to bind it to the column.

2.21.5 unique and primary key Constraints

You can declare `unique` or `primary key` constraints to ensure that no two rows in a table have the same values in the specified columns.

Both constraints create unique indexes to enforce this data integrity. However, `primary key` constraints are more restrictive than `unique` constraints. Columns with `primary key` constraints cannot contain a NULL value. You normally use a table's `primary key` constraint with referential integrity constraints defined on other tables.

The definition of `unique` constraints in the SQL standard specifies that the column definition shall not allow null values. By default, SAP ASE defines the column as not allowing null values (if you have not changed this using `sp_dboption`) when you omit `null` or `not null` keywords in the column definition. In Transact-SQL, you can define the column to allow null values along with the `unique` constraint, since the unique index used to enforce the constraint allows you to insert a null value.

> **Note**
>
> Do not confuse the unique and primary key integrity constraints with the information defined by `sp_primarykey`, `sp_foreignkey`, and `sp_commonkey`. The unique and primary key constraints actually create indexes to define unique or primary key attributes of table columns. `sp_primarykey`, `sp_foreignkey`, and `sp_commonkey` define the logical relationship of keys (in the `syskeys` table) for table columns, which you enforce by creating indexes and triggers.

`unique` constraints create unique nonclustered indexes by default; `primary key` constraints create unique clustered indexes by default. You can declare either clustered or nonclustered indexes with either type of constraint.

For example, the following `create table` statement uses a table-level `unique` constraint to ensure that no two rows have the same values in the `stor_id` and `ord_num` columns:

```
create table my_sales
```

```
(stor_id     char(4),
ord_num      varchar(20),
date         datetime,
unique clustered (stor_id, ord_num))
```

There can be only one clustered index on a table, so you can specify only one `unique clustered` or `primary key clustered` constraint.

You can use the `unique` and `primary key` constraints to create unique indexes (including the `with fillfactor`, `with max_rows_per_page`, and `on <segment_name>` options) when enforcing data integrity. However, indexes provide additional capabilities.

Related Information

Create Indexes on Tables [page 174]

2.21.6 Referential Integrity Constraints

Referential integrity refers to the methods used to manage the relationships between tables. When you create a table, you can define constraints to ensure that the data inserted into a particular column has matching values in another table.

There are three types of references you can define in a table: references to another table, references from another table, and self-references, that is, references within the same table. The referential integrity constraints in these examples are defined at the column level, using the `references` keyword in the `create table` statement.

The following two tables from the `pubs3` database illustrate how declarative referential integrity works. The first table, `stores`, is a "referenced" table:

```
create table stores
(stor_id     char(4) not null,
stor_name    varchar(40) null,
stor_address varchar(40) null,
city         varchar(20) null,
state        char(2) null,
country      varchar(12) null,
postalcode   char(10) null,
payterms     varchar(12) null,
unique nonclustered (stor_id))
```

The second table, `store_employees`, is a "referencing table" because it contains a reference to the `stores` table. It also contains a self-reference:

```
create table store_employees
(stor_id     char(4) null
       references stores(stor_id),
emp_id       id not null,
mgr_id       id null
       references store_employees(emp_id),
emp_lname    varchar(40) not null,
emp_fname    varchar(20) not null,
```

```
phone         char(12) null,
address       varchar(40) null,
city          varchar(20) null,
state         char(2) null,
country       varchar(12) null,
postalcode    varchar(10) null,
unique nonclustered (emp_id))
```

The references defined in the `store_employees` table enforce these restrictions:

- Any store specified in the `store_employees` table must be included in the `stores` table. The `references` constraint enforces this by verifying that any value inserted into the `stor_id` column in `store_employees` must already exist in the `stor_id` column in `my_stores`.
- All managers must have employee identification numbers. The `references` constraint enforces this by verifying that any value inserted into the `mgr_id` column must already exist in the `emp_id` column.

2.21.6.1 Table and Column Level Referential Integrity Constraints

You can define referential integrity constraints at the column level or the table level.

When you define table-level referential integrity constraints, include the `foreign key` clause, and a list of one or more column names. `foreign key` specifies that the listed columns in the current table are foreign keys for which the target keys are the columns listed the following `references` clause. For example:

```
constraint sales_detail_constr
     foreign key (stor_id, ord_num)
     references my_salesdetail(stor_id, ord_num)
```

The `foreign key` syntax is permitted only for table-level constraints, and not for column-level constraints.

After defining referential integrity constraints at the column level or the table level, you can use `sp_primarykey`, `sp_foreignkey`, and `sp_commonkey` to define the keys in the `syskeys` system table.

> **Note**
>
> The maximum number of references allowed for a table is 192.

Related Information

Table and Column Level Constraints [page 106]

2.21.6.2 Using Create Schema for Cross-Referencing Constraints

You cannot create a table that references a table that does not yet exist. To create two or more tables that reference each other, use `create schema`.

A schema is a collection of objects owned by a particular user, and the permissions associated with those objects. If any of the statements within a `create schema` statement fail, the entire command is rolled back as a unit, and none of the commands take effect.

The `create schema` syntax is:

```
create schema authorization <authorization name>
<create_object_statement>
     [create_object_statement ...]
[<permission_statement> ...]
```

For example:

```
create schema authorization dbo
    create table list1
        (col_a char(10) primary key,
          col_b char(10) null
        references list2(col_A))
    create table list2
        (col_A char(10) primary key,
          col_B char(10) null
        references list1(col_a))
```

2.21.6.3 General Rules for Creating Referential Integrity Constraints

You should follow certain guidelines when creating referential integrity constraints.

When you define referential integrity constraints in a table:

- Make sure you have `references` permission on the referenced table. See, *Managing User Permissions*, in the *Security Administration*.
- Make sure that the referenced columns are constrained by a unique index in the referenced table. You can create that unique index using either the `unique` or `primary key` constraint, or the `create index` statement. For example, the referenced column in the `stores` table is defined as:

  ```
  stor_id char(4) primary key
  ```

- Make sure the columns used in the references definition have matching datatypes. For example, the `stor_id` columns in both `my_stores` and `store_employees` were created using the `char(4)` datatype. The `mgr_id` and `emp_id` columns in `store_employees` were created with the `id` datatype.
- You can omit column names in the `references` clause only if the columns in the referenced table are designated as a primary key through a `primary key` constraint.
- You cannot delete rows or update column values from a referenced table that match values in a referencing table. Delete or update from the referencing table first, and then delete from the referenced table.

Similarly, you cannot use `truncate table` on a referenced table. Truncate the referencing table first, then truncate the referenced table.

- You must drop the referencing table before you drop the referenced table; otherwise, a constraint violation occurs.
- Use `sp_helpconstraint` to find a table's referenced tables.

Referential integrity constraints provide a simpler way to enforce data integrity than triggers. However, triggers provide additional capabilities to enforce referential integrity between tables.

Related Information

Triggers: Enforce Referential Integrity [page 583]

2.21.6.4 Designing Applications That Use Referential Integrity

You should follow certain guidelines when designing applications that use referential integrity.

When you design applications that use referential integrity features:

- Do not create unnecessary referential constraints. The more referential constraints a table has, the slower a statement requiring referential integrity runs on that table.
- Use as few self-referencing constraints on a table as possible.
- Use `check` constraint rather than `references` constraint for applications that check a limited, specific range of values. Using `check` constraint eliminates the need for SAP ASE to scan other tables to complete the query, since there are no references. Therefore, queries on such tables run faster than on tables using references.
  For example, this table uses a `check` constraint to limit the authors to California:

  ```
  create table cal_authors
  (au_id id not null,
  au_lname varchar(40) not null,
  au_fname varchar(20) not null,
  phone char(12) null,
  address varchar(40) null,
  city varchar(20) null,
  state char(2) null
       check(state = "CA"),
  country varchar(12) null,
  postalcode char(10) null)
  ```

- Bind commonly scanned foreign-key indexes to their own caches, to optimize performance. Unique indexes are automatically created on primary-key columns. These indexes are usually selected to scan the referenced table when their corresponding foreign keys are updated or inserted.
- Keep multirow updates of candidate keys at a minimum.
- Put referential integrity queries into procedures that use constraint checks. Constraint checks are compiled into the execution plan; when a referential constraint is altered, the procedure that has the constraint compiled is automatically recompiled when that procedure is executed.

- If you cannot embed referential integrity queries in a procedure and you must frequently recompile referential integrity queries in an ad hoc batch, bind the system catalog `sysreferences` to its own cache. This improves performance when SAP ASE recompiles referential integrity queries.
- To test a table that has referential constraints, use `set showplan, noexec on` before running a query using the table. The `showplan` output indicates the number of auxiliary scan descriptors required to run the query; scan descriptors manage the scan of a table whenever queries are run on it. If the number of auxiliary scan descriptors is very high, either redesign the table so it uses fewer scan descriptors, or increase the value of the `number of auxiliary scan descriptors` configuration parameter.

2.22 Computed Columns

Computed columns, computed column indexes, and function-based indexes provide convenient data manipulation and faster data access.

- Computed columns are defined by an expression, whether from regular columns in the same row, or functions, arithmetic operators, XML path queries, and so forth.
  The expression can be either deterministic or nondeterministic. The deterministic expression always returns the same results from the same set of inputs.
- You can create indexes on materialized computed columns as if they were regular columns.

Computed columns and function-based indexes similarly allow you to create indexes on expressions.

Computed columns and function-based indexes differ in some respects:

- A computed column provides both shorthand for an expression and indexability, while a function-based index provides no shorthand.
- Function-based indexes allow you to create indexes directly on expressions, while to create an index on a computed column, you must create the computed column first.
- A computed column can be either deterministic or nondeterministic, but a function-based index must be deterministic. “Deterministic” means that if the input values in an expression are the same, the return values must also be the same.
- You can create a clustered index on a computed column, but not a clustered function-based index.

These are the differences between materialized and not materialized computed columns:

- Computed columns can be materialized or not materialized. Columns that are materialized are preevaluated and stored in the table when base columns are inserted or updated. The values associated with them are stored in both the data row and the index row. Any subsequent access to a materialized column does not require reevaluation; its preevaluated result is accessed. Once a column is materialized, each access to it returns the same value.
- Columns that are not materialized are sometimes called virtual columns; virtual columns become materialized when they are accessed. If a column is virtual, or not materialized, its result value must be evaluated each time the column is accessed. This means that if the virtual computed column expression is based on a nondeterministic expression, or calls one, it may return different values each time you access it.You may also encounter runtime exceptions, such as domain errors, when you access virtual computed columns.

> **Note**
>
> A computed column cannot reference a user-defined function that belongs to a different database.

Related Information

Deterministic Property [page 118]
Function-Based Indexes [page 179]

2.22.1 Computed Columns Usage

Computed columns allow you to create a shorthand term for an expression, such as “Pay” for “Salary + Commission,” and to make that column indexable, as long as its datatype is indexable.

Nonindexable datatypes include:

- `text`
- `unitext`
- `image`
- `Java class`
- `bit`

Computed columns are intended to improve application development and maintenance efficiency. By centralizing expression logics in the table definition, and giving expressions meaningful aliases, computed columns make greatly simplified and readable queries. You can change expressions by simply modifying the computed column definitions.

Computed columns are particularly useful when you must index a column for which the defining expression is either a nondeterministic expression or function, or which calls a nondeterministic expression or function. For example, `getdate` always returns the current date, so it is nondeterministic. To index a column using `getdate`, build a materialized computed column and then index it:

```
create table rental
      (cust_id int, start_date as getdate()materialized, prod_id int)
create index ind_start_date on rental (start_date)
```

Composing and Decomposing Datatypes

An important feature of computed columns is that you can use them to compose and decompose complex datatypes (for example, `XML`, `text`, `unitext`, `image`, and Java classes). You can use computed columns either to make a complex datatype from simpler elements (compose), or to extract one or more elements from a complex datatype (decompose). Complex datatypes are usually composed of individual elements or fragments. You can define automatic decomposing or composing of these complex datatypes when you define the table. For example, suppose you want to store XML “order” documents in a table, along with some relational elements: `<order_no>`, `<part_no>`, and `<customer>`. Using `create table` with the `compute` and `materialized` parameters, you can define an extraction with computed columns:

```
create table orders(xml_doc image,
order_no compute xml_extract("order_no", xml_doc)materialized,
part_no compute xml_extract ("part_no", xml_doc)materialized,
```

```
customer compute xml_extract("customer", xml_doc)materialized)
```

Each time you insert a new XML document into the table, the document's relational elements are automatically extracted into the computed columns.

Or, to present the relational data in each row as an XML document, specify mapping the relational data to an XML document using a computed column in the table definition. For example, define a table:

```
create table orders
(order_no int,part_no int, quantity smallint, customer varchar(50))
```

Later, to return an XML representation of the relational data in each row, add a computed column using `alter table`:

```
alter table orders
add order_xml compute order_xml(order_no, part_no, quantity, customer)
```

Then use a `select` statement to return each row in XML format:

```
select order_xml from orders
```

User-Defined Ordering

Computed columns support `comparison`, `order by`, and `group by` ordering of complex datatypes, such as `XML`, `text`, `unitext`, `image`, and Java classes. You can use computed columns to extract relational elements of complex data, which you can use to define ordering.

You can also use computed columns to transform data into different formats, to customize data presentations for data retrieval. This is called user-defined sort order. For example, this query returns results in the order of the server's default character set and sort order, usually ASCII alphabetical order:

```
select name, part_no, listPrice from parts_table order by name
```

Use computed columns to present your query result in a case-insensitive format, such as ordering based on special-case acronyms, as in the ordering of stock market symbols, or using system sort orders other than the default. To transform data into a different format, use either the built-in function `sortkey`, or a user-defined sort-order function.

For example, to add a computed column called `<name_in_myorder>` with the user-defined function `Xform_to_myorder()`:

```
alter table parts_table add name_in_myorder compute
     Xform_to_myorder(name)materialized
```

To return the result in the customized format:

```
select name, part_no, listPrice from parts_table order by name_in_myorder
```

This approach allows you to materialize the transformed ordering data and create indexes on it.

If you prefer, you can do the same thing using data manipulation language (DML):

```
select name, part_no, listPrice from parts_table
```

```
       order by Xform_to_myorder(name)
```

However, using the computed column approach allows you to materialize the transformed ordering data and create indexes on it, which improves the performance of the query.

Decision-Support Systems (DSS)

Typical decision-support system applications require intensive data manipulation, correlation, and collation data analysis. Such applications frequently use expressions and functions in queries, and special user-defined ordering is often required. Using computed columns and function-based indexes simplifies the tasks necessary in such applications, and improves performance.

2.22.1.1 Computed Columns Example

A computed column is defined by an expression. You can build the expression by combining regular columns in the same row. Expressions may contain functions, arithmetic operators, case expressions, other columns from the same table, global variables, Java objects, and path expressions.

In the example below:

- `part_no` is a `Java object` column that represents the specified part number.
- `desc` is a `text` column that contains a detailed description of the specified parts.
- `spec` is an `image` column that stores the parsed XML stream object.
- name_order is a computed column defined by the user-defined function `XML()`.
- `version_order` is a computed column defined by the Java class.
- `descr_index` is a computed column defined by `des_index()`, which generates an index key on the `text` data.
- `spec_index` is a computed column defined by `xml_index()`, which generates an index key on the XML document.
- `total_cost` is a computed column defined by an arithmetical expression.

```
create table parts_table
  (part_no Part.Part_No, name char(30),
  descr text, spec image, listPrice money,
      quantity int,
   name_order compute name_order(part_no)
   version_order compute part_no version,
   descr_index compute des_index(descr),
   spec_index compute xml_index(spec)
   total_cost compute quantity*listPrice
)
```

2.22.2 Indexes on Computed Columns

You can create indexes on computed columns, as long as the datatype of the result can be indexed. Computed column indexes and function-based indexes provide a way to create indexes on complex datatypes like `XML`, `text`, `unitext`, `image`, and Java classes.

For example, the following code sample creates a clustered index on the computed columns:

```
CREATE CLUSTERED INDEX name_index on part_table(name_order)
CREATE INDEX adt_index on parts_table(version_order)
CREATE INDEX xml_index on parts_table(spec_index)
CREATE INDEX text_index on parts_table(descr_index)
```

SAP ASE evaluates the computed columns and uses the results to build or update indexes when you create or update an index.

2.22.3 Deterministic Property

All expressions and functions are deterministic or nondeterministic, which means they may or may not return the same results each time they are evaluated.

- Deterministic expressions and functions always return the same result, as long as they are evaluated with the same set of input values. This expression is deterministic:

  ```
  c1 * c2
  ```

- Nondeterministic expressions of functions may return different results each time they are evaluated, even when they are called with the same set of input values. The function `getdate` is nondeterministic because it always returns the current date.

An expression's deterministic property defines a computed column or a function-based index key, and thus defines the computed column or function-based index key itself.

The deterministic property depends on whether the expression contains any nondeterministic elements, such as various system functions, user-defined functions, and global variables.

Whether a function is deterministic or nondeterministic depends on the function coding:

- If the function calls nondeterministic functions, it may be nondeterministic itself.
- If a function's return value depends on factors other than input values, the function is probably nondeterministic.

2.22.3.1 Effects of Deterministic Property on Computed Columns

Virtual and materialized are two types of computed columns in SAP ASE.

A virtual computed column is referenced by a query, and is evaluated each time a query accesses it.

A materialized computed column's result is stored in the table when a data row is inserted, or when any base columns are updated. When a materialized computed column is referenced in a query, it is not reevaluated. Its preevaluated result is used.

- A nonmaterialized, or virtual, computed column becomes a materialized computed column if it is used as an index key.
- A materialized computed column is reevaluated only if one of its base columns is updated.

2.22.3.2 Effects of Deterministic Property on Materialized Computed Columns

SAP ASE guarantees repeatable reads on materialized computed columns, regardless of their deterministic property, because they are not reevaluated when you reference them in a query. Instead, SAP ASE uses the preevaluated values.

Deterministic materialized computed columns always have the same values, no matter how often they are reevaluated.

Nondeterministic materialized computed columns must adhere to these rules:

- Each evaluation of the same computed column may return a different result, even using the same set of inputs.
- References to nondeterministic preevaluated computed columns use the preevaluated results, which may differ from current evaluation results. In other words, historical rather than current data is used in nondeterministic preevaluated computed columns.

In the first example from *Examples of Nondeterministic Computed Columns*, `Start_Date` is a nondeterministic materialized computed column. Its results differ, depending on what day you insert the row. For instance, if the rental period begins on "02/05/04," "02/05/04" is inserted into the column, and later references to `Start_Date` use this value. If you reference this value later, on 06/05/04, the query continues to return "02/05/04," not "06/05/04," as you would expect if the expression was evaluated every time you query the column.

2.22.3.3 Effects of Deterministic Property on Virtual Computed Columns

SAP ASE guarantees repeatable reads on deterministic virtual computed columns, even though, by definition, a virtual computed column is evaluated each time it is referenced.

For example, this statement always returns the same result, if the data in the table does not change:

```
select Cust_ID, Property_ID from Renting
     where Formatted_Name ='RICHARD HUANG'
```

SAP ASE does not guarantee repeatable reads on nondeterministic virtual computed columns. For example, in this query, the column `Rent_Due` returns different results on different days; the column has a serial time property, for which the value is a function of the amount of time that passes between rent payments:

```
select Cust_Name, Rent_Due from renting
    where Cust_Name= 'RICHARD HUANG'
```

The nondeterministic property is useful here, but use it with caution. For instance, if you inadvertently defined `Start_Date` as a virtual computed column and entered the same query, you would rent all your properties for nothing: `Start_Date` is always evaluated to the current date, so in this query, the number of `Rental_Days` is always 0.

Likewise, if you mistakenly define the nondeterministic computed column `Rent_Due` as a preevaluated column, either by declaring it materialized or by using it as an index key, you would rent your properties for nothing. It is evaluated only once, when the record is inserted, and the number of rental days is 0. This value is returned each time the column is referenced.

2.22.3.4 Effects of Deterministic Property on Function-Based Indexes

Unlike computed columns, function-based index keys must be deterministic. A computed column is still conceptually a column, which, once evaluated and stored, does not require reevaluation. A function or expression, however, must be reevaluated upon each appearance in a query.

You cannot use preevaluated data, such as index data, unless the function always evaluates to the same results with the same input set .

- SAP ASE internally represents function-based index keys as hidden materialized computed columns. The value of a function-based index key is stored on both a data row and an index page, and therefore assumes all the properties of a materialized computed column.
- SAP ASE assumes all function- or expression-based index keys to be deterministic. When these index keys are referenced in a query, the preevaluated results that are already stored in the index page are used; the index keys are not reevaluated.
- Preevaluated results are updated only when the base columns of the function-based index key are updated.
- Do not use a nondeterministic function as an index, as in Example 2. The results can be unexpected.

Related Information

Function-Based Indexes [page 179]

2.22.3.5 Examples of Nondeterministic Computed Columns

Examples are provide to illustrate both the usefulness and the dangers of using nondeterministic computed columns and index keys.

Example

The table `Renting` in this example stores rental information on various properties. It contains these fields:

- `Cust_ID` – ID of the customer
- `Cust_Name` – name of the customer
- `Formatted_Name` – customer's name
- `Property_ID` – ID of the property rented
- `Property_Name` – name of the property in a standard format
- `Start_Date` – starting day of the rent
- `Rent_Due` – amount of rent due today

```
create table Renting
     (Cust_ID int, Cust_Name varchar(30),
     Formatted_Name compute format_name(Cust_Name),      Property_ID
int,Property_Name compute
     get_pname(Property_ID), start_date compute
     today_date()materialized, Rent_due compute
     rent_calculator(Property_ID, Cust_ID,      Start_Date))
```

`Formatted_Name`, `Property_Name`, `Start_Date`, and `Rent_Due` are defined as computed columns.

- `Formatted_Name` – virtual computed column that transforms the customer name to a standard format. Since its output depends solely on the input `Cust_Name`, `Formatted_Name` is deterministic.
- `Property_Name` – virtual computed column that retrieves the name of the property from another table, `Property`, which is defined as:

  ```
  create table Property
       (Property_ID int, Property_Name varchar(30),       Address
  varchar(50), Rate int)
  ```

  To get the property name based on the input ID, the function `get_pname` invokes a JDBC query:

  ```
  select Property_Name from Property where Property_ID=input_ID
  ```

  The computed column `Property_Name` looks deterministic, but it is actually nondeterministic, because its return value depends on the data stored in the table `Property` as well as on the input value `Property_ID`.
- `Start_Date` – a nondeterministic user-defined function that returns the current date as a `varchar(15)`. It is defined as materialized. Therefore, it is reevaluated each time a new record is inserted, and that value is stored in the `Renting` table.
- `Rent_Due` – a virtual nondeterministic computed column, which calculates the current rent due, based on the rent rate of the property, the discount status of the customer, and the number of rental days.

Example

Using the table created in Example 1, `Renting`: If you create an index on the virtual computed column `Property_Name`, it becomes a materialized computed column.

If you then inserted a new record:

```
Property_ID=10
```

This new record calls `get_pname(10)` from the table `Property`, executing this JDBC query:

```
select Property_Name from Property where Property_ID=10
```

The query returns "Rose Palace," which is stored in the data row. This all works, unless someone changes the name of the property by issuing:

```
update Property set Property_Name ='Red Rose Palace'
     where Property_ID = 10
```

The query returns "Red Rose Palace," so SAP ASE stores "Red Rose Palace." This `update` command on the table `Property` invalidates the stored value of `Property_Name` in the table `Renting`, which must also be updated to "Red Rose Palace." Because `Property_Name` is defined on the column `Property_ID` in the table `Renting`, not on the column `Property_Name` in the table `Property`, it is not automatically updated. Future reference to `Property_Name` may produce incorrect results.

To avoid this situation, create a trigger on the table `Property`:

```
CREATE TRIGGER my_t ON Property FOR UPDATE AS
     IF UPDATE(Property_Name)
     BEGIN
     UPDATE Renting SET Renting.Property_ID=Property.Property_ID
          FROM Renting, Property
          WHERE Renting.Property_ID=Property.Property_ID
     END
```

When this trigger updates the column `Property_Name` in the table `Property`, it also updates the column `Renting.Property_ID`, the base column of `Property_Name`. This automatic update triggers SAP ASE to reevaluate `Property_Name` and update the value stored in the data row. Each time SAP ASE updates the column `Property_Name` in the table `Property`, the materialized computed column `Property_Name` in the table `Renting` is refreshed, and shows the correct value.

2.23 Retrieve Information About Databases and Tables

SAP ASE includes several procedures and functions you can use to get information about databases, tables, and other database objects.

See the *Reference Manual: Procedures* and also the *Reference Manual: Building Blocks* for information about the procedures and functions for additional information.

2.23.1 Help on Databases

`sp_helpdb` can report information about a specified database or about all SAP ASE databases.

```
sp_helpdb [<dbname>]
```

This example displays a report on `pubs2` on a server using a page size of 8K.

```
sp_helpdb pubs2
```

```
name      db_size    owner     dbid created        status
--------- ---------- --------- ---- -------------- --------------
pubs2     20.0 MB     sa           4 Apr 25, 2005   select
     into/bulkcopy/pllsort, trunc log on chkpt, mixed log and data
device_fragments    size          usage         created      free kbytes
------------------- ------------- ------------- ----------   ------------
master              10.0MB        data and log  Apr 13 2005          1792
```

```
pubs_2_dev         10.0MB        data and log  Apr 13 2005         9888
device                 segment
---------------------- ----------------------
master                 default
master                 logsegment
master                 system
pubs_2_dev             default
pubs_2_dev             logsegment
pubs_2_dev             system
pubs_2_dev             seg1
pubs_2_dev             seg2
```

`sp_databases` lists all the databases on a server. For example:

```
sp_databases
```

```
database_name     database_size  remarks
----------------- -------------  ------------
master                    5120  NULL
model                     2048  NULL
pubs2                     2048  NULL
pubs3                     2048  NULL
sybsecurity               5120  NULL
sybsystemprocs           30720  NULL
tempdb                    2048  NULL

(7 rows affected, return status = 0)
```

To find out who owns a database, use `sp_helpuser`:

```
sp_helpuser dbo
```

```
Users_name    ID_in_db Group_name   Login_name
------------- -------- ------------ ------------
dbo                  1 public       sa
```

To identify the current database, use `db_id` and `db_name`:

```
select db_name(), db_id()
```

```
------------------------------ ------
master                              1
```

2.23.2 Help on Database Objects

SAP ASE provides system procedures, catalog stored procedures, and built-in functions that return helpful information about database objects, such as tables, columns, and constraints.

2.23.2.1 sp_help Usage on Database Objects

Use `sp_help` to display information about a specified database object (that is, any object listed in `sysobjects`), a specified datatype (listed in `systypes`), or all objects and datatypes in the current database.

```
sp_help [<objname>]
```

This is `sp_help` output for the `publishers` table:

```
Name                Owner        Object_type     Create_date
----------------    -----------  -------------   ------------------------------
publishers          dbo          user table      Nov 9 2004 9:57AM

(1 row affected)
Column_name Type      Length   Prec  Scale   Nulls   Default_name   Rule_name
----------- -------   ------   ----- ------- ------- -------------- ----------
pub_id      char           4    NULL  NULL        0  NULL           pub_idrule
pub_name    varchar       40    NULL  NULL        1  NULL           NULL
city        varchar       20    NULL  NULL        1  NULL           NULL
state       char           2    NULL  NULL        1  NULL           NULL
Access_Rule_name    Computed_Column_object     Identity
------------------- -------------------------  ------------
NULL                NULL                                  0
NULL                NULL                                  0
NULL                NULL                                  0
NULL                NULL                                  0

Object has the following indexes
index_name index_keys  index_description  index_max_rows_per_page
---------- ----------  ---------- -------- -------------
pubind     pub_id      clustered, unique              0

index_fill_factor  index_reservepagegap   index_created      index_local
------------------ ----------------------- ----------------   ----------------
                0                       0  Nov 9 2004 9:58AM  Global Index
(1 row affected)
index_ptn_name   index_ptn_segment
---------------- --------------------
pubind_416001482  default
(1 row affected)
keytype  object      related_object  related_keys
-------  ----------  --------------  ------------
```

```
primary  publishers  -- none --      pub_id, *, *, *, *, *
foreign  titles      publishers      pub_id, *, *, *, *, *

(1 row affected)
name         type        partition_type     partitions     partition_keys
--------     --------    ------------------ -------------- ------------------
publishers   base table  roundrobin                     1   NULL
partition_name        partition_id  pages   segment     Create_date
------------------    ------------  ------  ----------- -----------------------
publishers_416001482  416001482         1   default     Nov 9 2004 9:58AM
Partition_Conditions
----------------------
NULL
Avg_pages   Max_pages   Min_pages   Ratio

(return status = 0)
No defined keys for this object.
name    type       partition_type   partitions   partition_keys
------- ---------- ---------------- ------------ ---------------
mytable base table roundrobin                  1 NULL
partiton_name       partition_id   pages     segment   create_date
-----------------   -------------- --------- --------- ------------------------
mytable_1136004047  1136004047            1  default     Nov 29 2004 1:30PM
partition_conditions
--------------------
NULL
Avg_pages   Max_pages   Min_pages   Ratio(Max/Avg)        Ration(Min/Avg)
----------- ----------- ----------- --------------------- -------------------
         1           1           1              1.000000               1.000000
Lock scheme Allpages
The attribute 'exp_row_size' is not applicable to tables with
allpages lock scheme.
The attribute 'concurrency_opt_threshold' is not applicable to
tables with allpages lock scheme.
exp_row_size reservepagegap fillfactor max_rows_per_page identity_gap
 ------------ -------------- ---------- ----------------- ------------
            1              0          0                 0            0
(1 row affected)
concurrency_opt_threshold   optimistic_index_lock   dealloc_first_txtpg
--------------------------- ------------------------ --------------------
                          0                        0                    0
(return status = 0)
```

If you execute `sp_help` without supplying an object name, the resulting report shows each object in `sysobjects`, along with its name, owner, and object type. Also shown is each user-defined datatype in `systypes` and its name, storage type, length, whether null values are allowed, and any defaults or rules bound to it. The report also notes if any primary or foreign-key columns have been defined for a table or view.

`sp_help` lists any indexes on a table, including those created by defining unique or primary key constraints. However, it does not include information about the integrity constraints defined for a table.

2.23.2.2 Use sp_helpconstraint to Find Table Constraint Information

sp_helpconstraint reports information about the declarative referential integrity constraints that are specified for a table, including the constraint name and definition of the default, unique or primary key constraint, referential, or check constraint.

sp_helpconstraint also reports the number of references associated with the specified tables.

```
sp_helpconstraint [<objname>] [, detail]
```

<objname> is the name of the table being queried. If you do not include a table name, sp_helpconstraint displays the number of references associated with each table in the current database. With a table name, sp_helpconstraint reports the name, definition, and number of integrity constraints associated with the table. The detail option also returns information about the constraint's user or error messages.

For example, sp_helpconstraint output on the store_employees table in pubs3 looks similar to:

```
name                          defn
----------------------------  --------------------------------
store_empl_stor_i_272004000   store_employees FOREIGN KEY
                              (stor_id) REFERENCES stores(stor_id)
store_empl_mgr_id_288004057   store_employees FOREIGN KEY
                              (mgr_id) SELF REFERENCES
                              store_employees(emp_id)
store_empl_2560039432         UNIQUE INDEX( emp_id) :
                              NONCLUSTERED, FOREIGN REFERENCE

(3 rows affected)

Total Number of Referential Constraints: 2
Details:
-- Number of references made by this table: 2
-- Number of references to this table: 1
-- Number of self references to this table: 1

Formula for Calculation:
Total Number of Referential Constraints
= Number of references made by this table
+ Number of references made to this table
- Number of self references within this table
```

To find the largest number of referential constraints associated with any table in the current database, run sp_helpconstraint without specifying a table name:

```
sp_helpconstraint
```

```
id          name                      Num_referential_constraints
----------- ------------------------- ---------------------------
   80003316 titles                                              4
   16003088 authors                                             3
  176003658 stores                                              3
  256003943 salesdetail                                         3
  208003772 sales                                               2
  336004228 titleauthor                                         2
  896006223 store_employees                                     2
   48003202 publishers                                          1
  128003487 roysched                                            1
  400004456 discounts                                           1
  448004627 au_pix                                              1
```

```
  496004798 blurbs                                          1

(11 rows affected)
```

This report shows that the `titles` table has the largest number of referential constraints in the `pubs3` database.

2.23.2.3 Determining Much Space a Table Uses

`sp_spaceused` computes and displays the number of rows and data pages used by a table or a clustered or nonclustered index.

To find out how much space a table uses, enter:

```
sp_spaceused [<objname>]
```

To display a report on the space used by the `titles` table:

```
sp_spaceused titles
```

```
name    rows   reserved  data  index_size  unused
------- -----  --------- ----- ---------   ------
titles  18     48 KB     6 KB  4 KB        38 KB

(0 rows affected)
```

If you do not include an object name, `sp_spaceused` displays a summary of space used by all database objects.

2.23.2.4 List Tables, Columns, and Datatypes

Catalog stored procedures retrieve information from the system tables in tabular form. You can supply wildcard characters for some parameters.

`sp_tables` lists all user tables in a database when used in the following format:

```
sp_tables @table_type = "'TABLE'"
```

`sp_columns` returns the datatype of any or all columns in one or more tables in a database. You can use wildcard characters to get information about more than one table or column.

For example, the following command returns information about all columns that include the string “id” in all the tables with “sales” in their name:

```
sp_columns "%sales%", null, null, "%id%"
```

```
table_qualifier table_owner
       table_name     column_name
       data_type type_name  precision  length  scale radix  nullable
       remarks
```

```
ss_data_type colid
---------------  -----------
      ----------       -----------
      ---------  --------- ---------  ------  ----- -----  --------
      -------

------------ -----
pubs2              dbo
       sales           stor_id
       1          char       4           4       NULL  NULL    0
NULL

47            1
pubs2            dbo
       salesdetail     stor_id
       1         char        4           4       NULL  NULL    0
NULL

4             1
pubs2          dbo
        salesdetail     title_id
        12      varchar      6           6       NULL  NULL 0
    NULL
39            3
(3 rows affected, return status = 0)
```

2.23.2.5 Find an Object Name and ID

To identify the ID and name of an object, use `object_id()` and `object_name()`.

The syntax is:

```
select object_id("titles")
```

```
----------
  208003772
```

Object names and IDs are stored in the `sysobjects` system table.

3 SQL-Derived Tables

A SQL-derived table is defined by one or more tables through the evaluation of a query expression, used in the query expression in which it is defined, and exists only for the duration of the query. It is not described in system catalogs or stored on disk.

SQL-derived tables are not the same as abstract-plan-derived tables. A table derived from an abstract plan is used for query optimization and execution, and differs from a SQL-derived table in that it exists only as part of an abstract plan and is invisible to the end user.

A SQL-derived table is created from an expression consisting of a nested `select` statement, as in the following example, which returns a list of cities in the `publishers` table of the `pubs2` database:

```
select city from (select city from publishers) cities
```

The SQL-derived table is named `cities` and has one column titled `city`. The SQL-derived table is defined by the nested select statement and persists only for the duration of the query, which returns:

```
city
--------------------
Boston
Washington
Berkeley
```

This example shows the advantages of SQL-derived tables.

If you are interested in viewing only the titles of books written in Colorado, you might create a view like this:

```
create view vw_colorado_titles as
     select title
     from titles, titleauthor, authors
     where titles.title_id = titleauthor.title_id
     and titleauthor.au_id = authors.au_id
     and authors.state ="CO"
```

You can repeatedly use the view `vw_colorado_titles`, stored in memory, to display its results:

```
select * from vw_colorado_titles
```

Drop the view when it is no longer needed:

```
drop view vw_colorado_titles
```

If the query results are only needed once, you might instead use a SQL-derived table:

```
select title
from (select title
      from titles, titleauthor, authors
      where titles.title_id = titleauthor.title_id
      and titleauthor.au_id = authors.au_id and
      authors.state = "CO") dt_colo_titles
```

The SQL-derived table created is named `dt_colo_titles`. The SQL-derived table persists only for the duration of the query, in contrast with a temporary table, which exists for the entire session.

In the previous example for query results that are only needed once, a view is less desirable than a SQL-derived table query because the view is more complicated, requiring both `create` and `drop` statements in addition to a `select` statement. The benefits of creating a view for only one query are additionally offset by the overhead of administrative tasks such as dealing with system catalogs. SQL-derived tables spontaneously create nonpersistent tables which require no administrative tasks. A SQL-derived table used multiple times performs comparably to a query using a view with a cached definition.

3.1 SQL-Derived Tables and Optimization

Queries that are expressed as a single SQL statement make more efficient use of the optimizer than queries that are expressed in two or more SQL statements.

SQL-derived tables use a single step for what might otherwise require several SQL statements and temporary tables, especially where intermediate aggregate results must be stored. For example, this single SQL statement obtains aggregate results from the SQL-derived tables `dt_1` and `dt_2`, and computes a join between the two tables:

```
select dt_1.* from
   (select sum(total_sales)
       from titles_west group by total_sales)
             dt_1(sales_sum),
   (select sum(total_sales)
       from titles_east group by total_sales)
             dt_2(sales_sum)
where dt_1.sales_sum = dt_2.sales_sum
```

3.2 SQL-Derived Table Syntax

The query expression for a SQL-derived table is specified in the `from` clause of the `select` or `select into` command.

The syntax is:

```
<from_clause> ::=
    from <table_reference> [,<table_reference>]...
```

```
<table_reference> ::=
    <table_view_name> | <ANSI_join>
```

```
<table_view_name> ::=
    {<table_view_reference> | <derived_table_reference>}
    [holdlock | noholdlock]
    [readpast]
    [shared]
```

```
<table_view_reference> ::=
    [[<database>.]<owner>.] {<table_name> | <view_name>}
    [[as] <correlation_name>]
    [index {<index_name> | <table_name> }]
    [parallel [<degree_of_parallelism>]]
    [prefetch size ]
    [lru | mru]
```

```
<derived_table_reference> ::=
    <derived_table> [as] <correlation_name>
    ['(' <derived_column_list>')']
```

```
<derived_column_list> ::= <column_name> [',' <column_name>] ...
```

```
<derived_table> ::= '(' <select> ')'
```

A derived-table expression is similar to the `select` in a `create view` statement and follows the same rules, except:

- Temporary tables are permitted in a derived-table expression except when it is part of a `create view` statement.
- A local variable is permitted in a derived-table expression except when it is part of a `create view` statement. You cannot assign a value to a variable within a derived-table expression.
- You cannot use variables in derived table syntax as part of a `create view` statement where the derived table is referenced in cursors.
- A `correlation_name`, which must follow the derived-table expression to specify the name of the SQL-derived table, may omit a derived column list, whereas a view cannot have unnamed columns:

  ```
  select * from
     (select sum(advance) from total_sales) dt
  ```

See "Restrictions on views" in the Usage section of `create view` in the *Reference Manual: Commands*.

3.2.1 Derived Column Lists

If a derived column list is not included in a SQL-derived table, the names of the SQL-derived table columns must match the names of the columns specified in the target list of the derived-table expression.

If a column name is not specified in the target list of the derived-table expression, as in the case where a constant expression or an aggregate is present in the target list of the derived-table expression, the resulting column in the SQL-derived table has no name. The server returns error 11073, `A derived-table expression may not have null column names...`

If a derived column list is included in a SQL-derived table, it must specify names for all columns in the target list of the derived-table expression. These column names must be used in the query block instead of the natural column names of the SQL-derived table. The columns must be listed in the order in which they occur in the derived-table expression, and a column name cannot be specified more than once in the derived column list.

3.2.2 Correlated SQL-Derived Tables Are Not Supported

Transact-SQL does not support correlated SQL-derived tables, which are also not ANSI standard.

For example, the following query is not supported because it references the SQL-derived table `dt_publishers2` inside the derived-table expression for `dt_publishers1`:

```
select * from
   (select * from titles where titles.pub_id =
      dt_publishers2.pub_id) dt_publishers1,
   (select * from publishers where city = "Boston")
      dt_publishers2
where dt_publishers1.pub_id = dt_publishers2.pub_id
```

Similarly, the following query is not supported because the derived-table expression for `dt_publishers` references the `publishers_pub_id` column, which is outside the scope of the SQL-derived table:

```
select * from publishers
  where pub_id in (select pub_id from
                      (select pub_id from titles
                      where pub_id = publishers.pub_id)
                   dt_publishers)
```

The following query illustrates proper referencing and is supported:

```
select * from publishers
  where pub_id in (select pub_id from
                      (select pub_id from titles)
                   dt_publishers
                     where pub_id = publishers.pub_id)
```

3.3 SQL-Derived Tables Usage

You can use SQL-derived tables to form part of a larger integrated query that uses assorted SQL clauses and operators.

3.3.1 Nesting

A query can use numerous nested derived-table expressions, which are SQL expressions that define a SQL-derived table.

In the following example, the innermost derived-table expression defines SQL-derived table `dt_1`, the `select from` forming the derived-table expression that defines SQL-derived table `dt_2`.

```
select postalcode
   from (select postalcode
            from (select postalcode
                    from authors) dt_1) dt_2
```

The degree of nesting is limited to 25.

3.3.2 Subqueries Using SQL-Derived Tables

You can use a SQL-derived table in a subquery `from` clause.

For example, this query finds the names of the publishers who have published business books:

```
select pub_name from publishers
   where "business" in
      (select type from
         (select type from titles, publishers
             where titles.pub_id = publishers.pub_id)
       dt_titles)
```

`dt_titles` is the SQL-derived table defined by the innermost `select` statement.

SQL-derived tables can be used in the `from` clause of subqueries anywhere subqueries are legal.

Related Information

Subqueries: Queries Within Other Queries [page 257]

3.3.3 Unions in Derived-Table Expressions

A `union` clause is allowed within a derived-table expression.

For example, the following query yields the contents of the `stor_id` and `ord_num` columns of both the `sales` and `sales_east` tables:

```
select * from
   (select stor_id, ord_num from sales
    union
    select stor_id, ord_num from sales_east)
dt_sales_info
```

The union of two `select` operations defines the SQL-derived table `dt_sales_info`.

3.3.4 Unions in Subqueries

A `union` clause is allowed in a subquery inside a derived-table expression.

The following example uses a `union` clause in a subquery within a SQL-derived table to list the titles of books sold at stores listed in the `sales` and `sales_east` tables:

```
select title_id from salesdetail
   where stor_id in
      (select stor_id from
         (select stor_id from sales
          union
```

```
          select stor_id from sales_east)
       dt_stores)
```

3.3.5 Rename Columns with SQL-Derived Tables

If a derived column list is included for a SQL-derived table, it follows the name of the SQL-derived table and is enclosed in parentheses.

For example:

```
select dt_b.book_title, dt_b.tot_sales
   from (select title, total_sales
            from titles) dt_b (book_title, tot_sales)
   where dt_b.book_title like "%Computer%"
```

The column names `title` and `total_sales` in the derived-table expression are respectively renamed to `book_title` and `tot_sales` using the derived column list. The `book_title` and `tot_sales` column names are used in the rest of the query.

> **Note**
>
> SQL-derived tables cannot have unnamed columns.

3.3.6 Constant Expressions

If a column name is not specified in the target list of the derived-table expression, as in the case where a constant expression is used for the column name, the resulting column in the SQL-derived table has no name.

For example:

```
1> select * from
2> (select title_id, (lorange + hirange)/2
3> from roysched) as dt_avg_range
4> go
title_id
--------- -----------
BU1032    2500
BU1032    27500
PC1035    1000
PC1035    2500
```

You can specify column names for the target list of a derived-table expression using a derived column list:

```
1> select * from
2> (select title_id, (lorange + hirange)/2
3> from roysched) as dt_avg_range (title, avg_range)
4> go
title     avg_range
--------- -----------
BU1032    2500
BU1032    27500
PC1035    1000
```

```
PC1035    2500
```

Alternately, you can specify column names by renaming the column in the target list of the derived-table expression:

```
1> select * from
2> (select title_id, (lorange + hirange)/2 avg_range
3> from roysched) as dt_avg_range
4> go
title     avg_range
--------- -----------
BU1032    2500
BU1032    27500
PC1035    1000
PC1035    2500
```

> **Note**
>
> If you specify column names in both a derived column list and in the target list of the derived-table expression, the resulting columns are named by the derived column list. The column names in a derived column list take precedence over the names specified in the target list of the derived-table expression.

If you use a constant expression within a `create view` statement, you must specify a column name for the constant expression results.

3.3.7 Aggregate Functions

Derived-table expressions can use aggregate functions, such as `sum`, `avg`, `max`, `min`, `count_big`, and `count`.

The following example selects columns `pub_id` and `adv_sum` from the SQL-derived table `dt_a`.

The second column is created in the derived-table expression using the `sum` function over the `advance` column of the `titles` table.

```
select dt_a.pub_id, dt_a.adv_sum
   from (select pub_id, sum(advance) adv_sum
         from titles group by pub_id) dt_a
```

If you use an aggregate function within a `create view` statement, you must specify a column name for the aggregate results.

3.3.8 Joins with SQL-Derived Tables

You can use the `where` clause to join a SQL-derived table and an existing table.

In this example, the two tables joined are `dt_c`, which is a SQL-derived table, and `publishers`, which is an existing table in the `pubs2` database.

```
select dt_c.title_id, dt_c.pub_id
   from (select title_id, pub_id from titles) as dt_c,
        publishers
```

```
   where dt_c.pub_id = publishers.pub_id
```

The following example illustrates a join between two SQL-derived tables. The two tables joined are `dt_c` and `dt_d`.

```
select dt_c.title_id, dt_c.pub_id
   from (select title_id, pub_id from titles)
         as dt_c,
        (select pub_id from publishers)
         as dt_d
   where dt_c.pub_id = dt_d.pub_id
```

You can also use outer joins with SQL-derived tables. SAP supports both left and right outer joins. The following example illustrates a left outer join between two SQL-derived tables.

```
select dt_c.title_id, dt_c.pub_id
   from (select title_id, pub_id from titles)
         as dt_c,
        (select title_id, pub_id from publishers)
         as dt_d
   where dt_c.title_id *= dt_d.title_id
```

The following example illustrates a left outer join within a derived-table expression:

```
select dt_titles.title_id
   from (select * from titles, titleauthor
         where titles.title_id *= titleauthor.title_id)
dt_titles
```

3.3.9 Create a Table From a SQL-Derived Table

Data obtained from a SQL-derived table can be inserted into a new table.

For example:

```
select pubdate into pub_dates
   from (select pubdate from titles) dt_e
            where pubdate = "450128 12:30:1PM"
```

Data from the SQL-derived table `dt_e` is inserted into the new table `pub_dates`.

3.3.10 Views with SQL-Derived Tables

You can create a view using a SQL-derived table.

The following example creates a view, `view_colo_publishers`, using a SQL-derived table, `dt_colo_pubs`, to display publishers based in Colorado:

```
create view view_colo_publishers (Pub_Id, Publisher,
City, State)
as select pub_id, pub_name, city, state
from
(select * from publishers where state="CO")
```

```
dt_colo_pubs
```

You can insert data through a view that contains a SQL-derived table if the `insert` rules and permission settings for the derived-table expression follow the `insert` rules and permission settings for the `select` part of the `create view` statement. For example, the following `insert` statement inserts a row through the `view_colo_publishers` view into the `publishers` table on which the view is based:

```
insert view_colo_publishers
values ('1799', 'Gigantico Publications', 'Denver',
'CO')
```

You can also update existing data through a view that uses a SQL-derived table:

```
update view_colo_publishers
set Publisher = "Colossicorp Industries"
where Pub_Id = "1699"
```

> **Note**
>
> Specify the column names of the view definition, not the column names of the underlying table.

Views that use a SQL-derived table are dropped in the standard manner:

```
drop view view_colo_publishers
```

3.3.11 Correlated Attributes

You cannot reference correlated attributes that exceed the scope of a SQL-derived table from a SQL-derived-table expression.

For example, the following query results in an error:

```
select * from publishers
   where pub_id in
      (select pub_id from
         (select pub_id from titles
          where pub_id = publishers.pub_id)
       dt_publishers)
```

Here, the column `publishers.pub_id` is referenced in the SQL-derived-table expression, but it is outside the scope of the SQL-derived table `dt_publishers`.

4 Partition Tables and Indexes

Use partitioning to manage large tables and indexes by dividing them into smaller, more manageable pieces. Partitions, like a large-scale index, provide faster and easier access to data.

Each partition can reside on a separate segment. Partitions are database objects and can be managed independently. You can, for example, load data and create indexes at the partition level. Yet partitions are transparent to the end user, who can select, insert, and delete data using the same DML commands whether the table is partitioned or not.

SAP ASE supports horizontal partitioning, in which a selection of table rows can be distributed among disk devices. Individual table or index rows are assigned to a partition according to a partitioning strategy.

Partitioning is the basis for parallel processing, which can significantly improve performance.

> **Note**
>
> Semantics-based partitioning is licensed separately. To enable semantic partitioning at a licensed site, set the value of the `enable semantic partitioning` configuration parameter to 1. See, *Setting Configuration Parameters*, in the *System Administration Guide: Volume 1*.

Partitioning:

- Improves scalability.
- Improves performance – concurrent multiple I/O on different partitions, and multiple threads on multiple CPUs working concurrently on multiple partitions.
- Provides faster response time.
- Provides partition transparency to applications.
- Supports very large database (VLDB) – concurrent scanning of multiple partitions of very large tables.
- Provides range partitioning to manage historical data.

> **Note**
>
> By default, SAP ASE creates tables with a single partition and uses a round-robin partitioning strategy. These tables are described as “unpartitioned” to distinguish between tables created or modified without partitioning syntax (the default) and those created with partitioning syntax.

Data Partitions

A data partition is an independent database object with a unique partition ID. It is a subset of a table, and shares the column definitions and referential and integrity constraints of the base table.

To maximize I/O parallelism, SAP recommends that you bind each partition to a different segment, and bind each segment to a different storage device.

Partition Keys

Each semantically partitioned table has a partition key that determines how individual data rows are distributed to different partitions. The partition key may consist of a single partition-key column or multiple key columns. The values in the key columns determine the actual partition distribution.

Range- and hash-partitioned tables can have as many as 31 key columns in the partition key. List partitions can have one key column in the partition key. Round-robin partitioned tables do not have a partition key.

You can specify partitioning-key columns of any type except:

- `text`, `image`, and `unitext`
- `bit`
- Java classes
- Computed columns

You can partition tables containing columns of these datatypes, but the partitioning key columns must be of supported datatypes.

Index Partitions

Indexes, like tables, can be partitioned. You can create local as well as global indexes.

An index partition is an independent database object identified with a unique combination of index ID and partition ID; it is a subset of an index, and resides on a segment or other storage device.

SAP ASE supports local and global indexes.

- A local index – spans data in exactly one data partition. For semantically partitioned tables, a local index has partitions that are equipartitioned with their base table; that is, the table and index share the same partitioning key and partitioning type.
  For all partitioned tables with local indexes, each local index partition has one and only one corresponding data partition.
- A global index – spans all data partitions in a table. SAP supports only unpartitioned global indexes. All unpartitioned indexes on unpartitioned tables are global.

You can mix partitioned and unpartitioned indexes with partitioned tables:

- A partitioned table can have partitioned and unpartitioned indexes.
- An unpartitioned table can have only unpartitioned, global indexes.

Partition IDs

A partition ID is a pseudorandom number similar to object ID. Partition IDs and object IDs are allocated from the same number space. An index or data partition is identified with a unique combination of index ID and partition ID.

Locks and Partitions

Partition locks increases data availability by creating finer locking granularity, which allows access to other partitions for concurrent DDL and DML statements.

See *Performance and Tuning Series: Locking and Concurrency Control*.

4.1 Partitioning Types

SAP ASE supports range partitioning, hash partitioning, list partitioning, and round-robin partitioning.

4.1.1 Range Partitioning

Rows in a range-partitioned table or index are distributed among partitions according to values in the partitioning key columns.

The partitioning column values of each row are compared with a set of upper and lower bounds to determine the partition to which the row belongs.

- Every partition has an inclusive upper bound, which is specified by the `values <=` clause when the partition is created.
- Every partition except the first has a noninclusive lower bound, which is specified implicitly by the `values <=` clause on the next-lower partition.

Range partitioning is particularly useful for high-performance applications in both OLTP and decision-support environments. Select ranges carefully so that rows are assigned equally to all partitions—knowledge of the data distribution of the partition-key columns is crucial to balancing the load evenly among the partitions.

Range partitions are ordered; that is, each succeeding partition must have a higher bound than the previous partition.

4.1.2 Hash Partitioning

In hash partitioning, SAP ASE uses a hash function to specify the partition assignment for each row. You select the partitioning key columns, but SAP ASE chooses the hash function that controls the partition assignment.

Hash partitioning is a good choice for:

- Large tables with many partitions, particularly in decision-support environments
- Efficient equality searches on hash key columns
- Data with no particular order, for example, alphanumeric product code keys

If you choose an appropriate partition key, hash partitioning distributes data evenly across all partitions. However, if you choose an inappropriate key—for example, a key that has the same value for many rows—the result may be an unbalanced distribution of rows among the partitions.

4.1.3 List Partitioning

As with range partitioning, list partitioning distributes rows semantically; that is, according to the actual value in the partitioning key column.

A list partition has only one key column. The value in the partitioning key column is compared with sets of user-supplied values to determine the partition to which each row belongs. The partition key must match exactly one of the values specified for a partition.

The value list for each partition must contain at least one value, and value lists must be unique across all partitions. You can specify as many as 250 values in each list partition. List partitions are not ordered.

4.1.4 Round-Robin Partitioning

In round-robin partitioning, SAP ASE assigns rows in a round-robin manner to each partition so that each partition contains a more or less equal number of rows and load balancing is achieved. Because there is no partition key, rows are distributed randomly across all partitions.

In addition, round-robin partitioning offers:

- Multiple insertion points for future inserts
- A way to enhance performance using parallelism
- A way to perform administrative tasks, such as updating statistics and truncating data on individual partitions

4.2 Partition Pruning

Partition pruning, or partition elimination, can save considerable time and resources during execution.

Semantics-based partitioning allow SAP ASE to eliminate certain partitions when performing a search. Range-based partitions, for example, contain rows for which partitioning keys are discrete value sets. When a query predicate—a `where` clause—is based on those partitioning keys, SAP ASE can quickly ascertain whether rows in a particular partition can satisfy the query. This behavior is called partition pruning.

- For range and list partitioning – SAP ASE can apply partition pruning on equality (=) and range (>, >=, <, and <=) predicates on partition-key columns on a single table.
- For hash partitioning – SAP ASE can apply partitioning pruning only on equality predicates on a single table.
- For range, list, and hash partitioning – SAP ASE cannot apply partition pruning on predicates with “not equal to” (!=) clauses or to complex predicates that have expressions on the partitioning column.

For example, suppose the `roysched` table in `pubs2` is partitioned on `hirange` and `royalty`. SAP ASE can use partitioning pruning on this query:

```
select avg(royalty) from roysched
where hirange <= 10000 and royalty < 9
```

The partition pruning process identifies `p1` and `p2` as the only partitions to qualify for this query. Thus, the `p3` partition need not be scanned, and SAP ASE can return query results more efficiently because it needs to scan only `p1` and `p2`.

In these examples, SAP ASE does not use partition pruning:

```
select * from roysched
where hirange != 5000
```

```
select * from roysched
where royalty*0.15 >= 45
```

> **Note**
>
> In serial execution mode, partition pruning applies only to scans, inserts, deletes, and updates; partition pruning does not apply to other operators. In parallel execution mode, partition pruning applies to all operators.

Related Information

Composite Partitioning Keys [page 142]

4.3 Composite Partitioning Keys

Semantically partitioned tables have one partition key per table or index. For range- or hash-partitioned tables, the partition key can be a composite key with as many as 31 key columns.

If a hash-partitioned table has a composite partitioning key, SAP ASE takes the values in all partitioning key columns and hashes the resultant data stream with a system-supplied hash function.

When a range-partitioned table has more than one partitioning key column, SAP ASE compares values of corresponding partitioning key columns in each data row with each partition upper and lower bound. Each partition bound is a list of one or more values, one for each partitioning key column.

SAP ASE compares partitioning key values with bounds in the order specified when the table was first created. If the first key value satisfies the assignment criteria of a partition, the row is assigned to that partition and no other key values are evaluated. If the first key value does not satisfy the assignment criteria, succeeding key values are evaluated until the assignment criteria is satisfied. Thus, SAP ASE may evaluate as few as one partitioning key value or as many as all keys values to determine a partition assignment.

For example, suppose `key1` and `key2` are partitioning columns for `my_table`. The table is made up of three partitions: `p1`, `p2`, and `p3`. The declared upper bounds are (a, b) for `p1`, (c, d) for `p2`, and (e, f) for `p3`.

```
if key1 < a, then the row is assigned to p1
if key1 = a, then
     if key2 < b or key2 = b, then the row is assigned to p1
if key1 > a or (key1 = a and key2 > b), then
     if key1 < c, then the row is assigned to p2
```

```
          if key1 = c, then
               if key2 < d or key2 = d, then the row is assigned to p2
          if key1 > c or (key1 = c and key2 > d), then
               if key1 < e, then the row is assigned to p3
               if key1 = e, then
                    if key2 < f or key2 = f, then the row is assigned to p3
                    if key2 > f, then the row is not assigned
```

Suppose the `roysched` table in `pubs2` is partitioned by range. The partitioning columns are high range (`hirange`) and royalty (`royalty`). There are three partitions: `p1`, `p2`, and `p3`. The upper bounds are (5000, 14) for `p1`, (10000, 10) for `p2`, and (100000, 25) for `p3`.

You can create partitions in the `roysched` table using `alter table`:

```
alter table roysched partition
        by range (hirange, royalty)
    (p1 values <= (5000, 14),
    p2 values <= (10000, 10),
    p3 values <= (100000, 25))
```

SAP ASE partitions the rows in this way:

- Rows with these partitioning key values are assigned to `p1`: (4001, 12), (5000, 12), (5000, 14), (3000, 18).
- Rows with these partitioning key values are assigned to `p2`: (6000, 18), (5000, 15), (5500, 22), (10000, 10), (10000, 9).
- Rows with these partitioning key values are assigned to `p3`: (10000, 22), (80000, 24), (100000, 2), (100000, 16).

SAP ASE evaluates tables with more than two partitioning key columns in a similar manner.

4.4 Indexes and Partitions

Indexes speed data retrieval by pointing to the location of a table column's data on disk. You can create global indexes and local indexes, each of which can also be clustered or nonclustered.

In clustered indexes, the physical data is stored in the same order as the index, and the bottom level of the index contains the actual data pages. In nonclustered indexes, the physical data is not stored in the same order as the index, and the bottom level of the index contains pointers to the rows on the data pages.

Clustered indexes on semantically partitioned tables are always local indexes—whether or not you specify "local" index in the `create index` command. Clustered indexes on round-robin tables can be either global or local.

4.4.1 Global Indexes

Global indexes span data in one or more partitions, which are not equipartitioned with the base table. Because SAP ASE supports only unpartitioned global indexes, a global index spans all partitions.

Global indexes are supported for compatibility with earlier versions of SAP ASE, and because they are particularly useful in OLTP environments.

SAP ASE supports these types of global indexes:

- Clustered indexes on round-robin and unpartitioned tables
- Nonclustered indexes on all types of tables

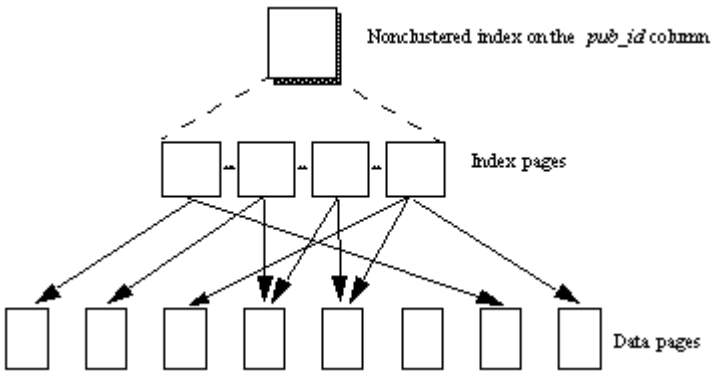
Global Nonclustered Index on Unpartitioned Table

This is an example of creating a global nonclustered index on an unpartitioned table.

The figure below shows the default nonclustered index configuration.

To create this index on the unpartitioned `publishers` table, enter:

```
create nonclustered index publish5_idx on
    publishers(pub_id)
```

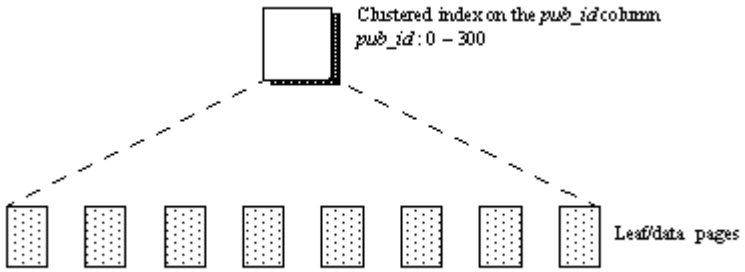


Global Clustered Index on Unpartitioned Table

This is an example figure that shows a default clustered index configuration. The table and index are unpartitioned.

To create this table on an unpartitioned `publishers` table, enter:

```
create clustered index publish4_idx
    on publishers(pub_id)
```



4.4.2 Local Indexes

All local indexes are equipartitioned with the base table's data partitions; that is, they inherit the partitioning type and partition key of the base table. Each local index spans just one data partition.

You can create local indexes on range-, hash-, list-, and round-robin–partitioned tables. Local indexes allow multiple threads to scan each data partition in parallel, which can greatly improve performance.

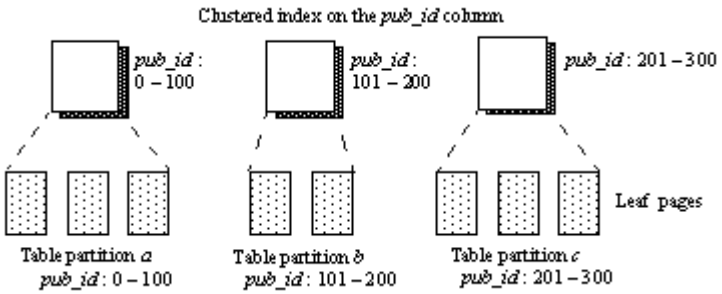
Local Clustered Indexes

When a table is partitioned, rows are assigned to a partition based on value, but the data is not sorted. When a local index is created, each partition is sorted separately.

This figure shows an example of partitioned clustered indexes on a partitioned table. The index is created on the `pub_id` column, and the table is indexed on `pub_id`. This example can enforce uniqueness on the `pub_id` column.

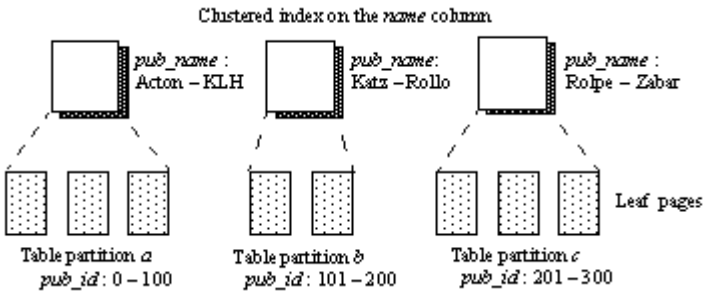
To create this table on the range-partitioned `publishers` table, enter:

```
create clustered index publish6_idx
    on publishers(pub_id)
    local index p1, p2, p3
```



To create this example on the range-partitioned `publishers` table, enter:

```
create clustered index publish7_idx
    on publishers(pub_name)
    local index p1, p2, p3
```



The information in each data partition conforms to the boundaries established when the partitions were created, which means you can enforce unique index keys across the entire table.

Local Nonclustered Indexes

You can define local nonclustered indexes on any set of indexable columns.

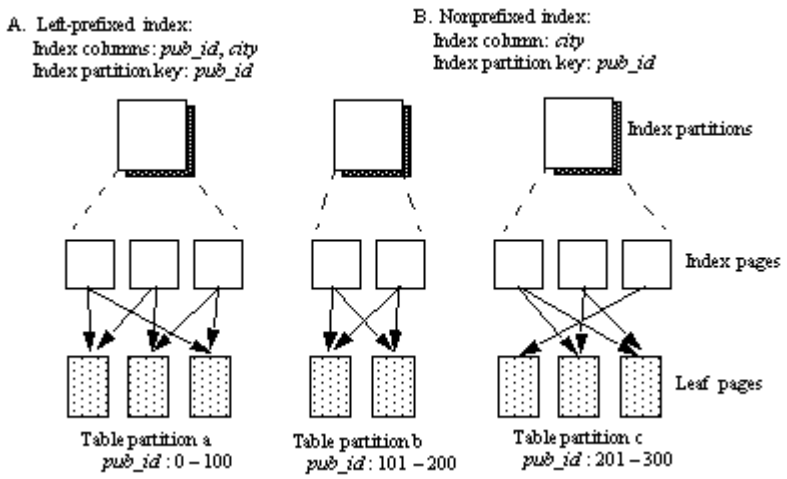
Using the `publishers` table partitioned by range on the `pub_id` column as in , create a partitioned, nonclustered index on the `pub_id` and `city` columns:

```
create nonclustered index publish8_idx (A)
    on publishers(pub_id, city)
    local index p1, p2, p3
```

You can also create a partitioned, nonclustered index on the `city` column:

```
create nonclustered index publish9_idx (B)
    on publishers(city)
    local index p1, p2, p3
```

This example shows both examples of nonclustered local indexes. The graphic description of each is identical. However, you can enforce uniqueness on example A; you cannot enforce uniqueness on example B.



4.4.3 Guarantee a Unique Index

A unique index ensures that no two rows have the same index value, including NULL. The system checks for duplicate values when the index is created, if data already exists, and checks each time data is added or modified with an `insert` or `update`.

You can easily enforce uniqueness—using the `unique` keyword—on global indexes because they are not partitioned. Local indexes are partitioned; enforcing uniqueness requires additional constraints.

To enforce uniqueness on local indexes, the partition keys must:

- Be a subset of the index keys
- Have the same sequence as the index keys

For example, you can impose uniqueness in these instances:

- A table partitioned by hash, list, or range on `column1`, with a local index with index key on `column1`.
- A table partitioned by hash, list, or range on `column1`, with a local index with index keys on `column1` and `column2`.
- A table is partitioned by hash, list, or range on `column1` and `column3`. A local index has these index keys:
  - `column1`, `column3`, or
  - `column1`, `column2`, `column3`, or
  - `column0`, `column1`, `column3`, `column4`.

  An index with these index keys cannot enforce uniqueness: `column3` or `column1`, `column3`.

You cannot enforce uniqueness on round-robin partitioned tables with local indexes.

Related Information

Create Indexes on Tables [page 174]

4.5 Create and Manage Partitions

The `sp_configure` option `enable semantic paritioning` turns on semantic paritioning. Round-robin partitioning is always available, and is unaffected by the value of `enable semantic partitioning`.

To enable semantic partitioning at your licensed site, enter:

```
sp_configure 'enable semantic partitioning', 1
```

Enable semantic partitioning to perform commonly used administrative and maintenance operations such as:

- Creating and truncating tables – `create table`, `truncate table`
- Altering tables to change locks or modify schema – `alter table`
- Creating indexes – `create index`
- Updating statistical information – `update statistics`
- Reorganizing table pages to conform to clustered indexes and best use of space – `reorg rebuild`

4.5.1 Partitioning Tasks

Before you partition a table or index, you must prepare the disk devices and the segments or other storages devices that you will use for the partitions.

Context

You can assign multiple partitions to a segment, but a partition can be assigned to only one segment. Assigning a single partition to each segment, with devices bound to individual segments, ensures the most benefit from parallelization and partitioning.

Procedure

1. Use `disk init` to initialize a new database device. `disk init` maps a physical disk device or operating system file to a logical database device name. For example:

   ```
   use master
   ```

   ```
   disk init
   name ="pubs_dev1",
   physname = "SYB_DEV01/pubs_dev",
   size = "50M"
   ```

   See, *Initializing Database Devices*, in the *System Administration Guide: Volume 1*.

2. Use `alter database` to assign the new device to the database containing the table or index to partition. For example:

```
use master
```

```
alter database pubs2 on pubs_dev1
```

3. (Optional) Use `sp_addsegment` to define the segments in the database. This example assumes that `pubs_dev2`, `pubs_dev3`, and `pubs_dev4` have been created in a similar manner to `pubs_dev1`.

```
use pubs2
```

```
sp_addsegment seg1, pubs2, pubs_dev1
sp_addsegment seg2, pubs2, pubs_dev2
sp_addsegment seg3, pubs2, pubs_dev3
sp_addsegment seg4, pubs2, pubs_dev4
```

4. Drop all indexes from the table to partition. For example:

```
use pubs2
```

```
drop index salesdetail.titleidind,
        salesdetail.salesdetailind
```

5. Use `sp_dboption` to enable the bulk-copy of table or index data to the new partitions. For example:

```
use master
```

```
sp_dboption pubs2,"select into", true
```

6. Use `alter table` to repartition a table or `create table` to create a new table with partitions; use `create index` to create a new, partitioned index; or use `select into` to create a new, partitioned table from an existing table.

   For example, to repartition the `salesdetail` table in `pubs2`:

```
use pubs2
```

```
alter table salesdetail partition by range (qty)
        (smsales values <= (1000) on seg1,
        medsales values <= (5000) on seg2,
        lgsales values <= (10000) on seg3)
```

7. Re-create indexes on the partitioned table. For example, on the `salesdetail` table:

```
use pubs2
```

```
create nonclustered index titleidind
        on salesdetail (title_id)
```

```
create nonclustered index salesdetailind
        on salesdetail (stor_id)
```

4.5.2 Create a Range-Partitioned Table

For best performance, each partition of a range-partitioned table resides on a separate segment.

This example creates a range-partitioned table called `fictionsales`; it has four partitions, one for each quarter of the year:

```
create table fictionsales
    (store_id int not null,
    order_num int not null,
    date datetime not null)
partition by range (date)
    (q1 values <= ("3/31/2004") on seg1,
    q2 values <= ("6/30/2004") on seg2,
    q3 values <= ("9/30/2004") on seg3,
    q4 values <= ("12/31/2004") on seg4)
```

The partitioning-key column is `date`. The `q1` partition resides on `seg1`, and includes all rows with `date` values through 3/31/2004. The `q2` partition resides on `seg2`, and includes all rows with date values of 4/1/2004 through 6/30/2004. `q3` and `q4` are partitioned similarly.

Attempting to insert `date` values later than "12/31/2004" causes an error, and the insert fails. In this way, the range conditions act as a check constraint on the table by limiting the rows that can be inserted into the table.

To make sure that all values, up to the maximum value for a datatype, are included, use the `MAX` keyword as the upper bound for the last-created partition. For example:

```
create table pb_fictionsales
    (store_id int not null,
    order_num int not null,
    date datetime not null)
partition by range (order_num)
    (low values <= (1000) on seg1,
    mid values <= (5000) on seg2,
    high values <= (MAX) on seg3)
```

4.5.2.1 Restrictions on Partition Keys and Bound Values for Range-Partitioned Tables

Partition bounds must be in ascending order according to the order in which the partitions were created. That is, the upper bound for the second partition must be higher than for the first partition, and so on.

In addition, partition bound values must be compatible with the corresponding partition-key column datatype. For example, `varchar` is compatible with `char`. If a bound value has a different datatype than that of its corresponding partition-key column, SAP ASE converts the bound value to the datatype of the partition-key column, with these exceptions:

- Explicit conversions are not allowed. This example attempts an illegal conversion from `varchar` to `int`:

  ```
  create table employees(emp_names varchar(20))
              partition by range(emp_name)
          (p1 values <=(1),
          p2 values <= (10))
  ```

- Implicit conversions that result in data loss are not allowed. In this example, rounding assumptions may lead to data loss if SAP ASE converts the bound values to `integer` values. The partition bounds are not compatible with the partition-key datatype.

```
create table emp_id (id int)
            partition by range(id)
        (p1 values <= (10.5),
        p2 values <= (100.5))
```

  In this example, the partitions bounds and the partition-key datatype are compatible. SAP ASE converts the bound values directly to `float` values. No rounding is required, and conversion is supported.

```
create table id_emp (id float)
            partition by range(id)
        (p1 values <= (10),
        p2 values <= (100))
```

- Conversions from nonbinary datatypes to binary datatypes are not allowed. For example, this conversion is not allowed:

```
create table newemp (name binary)
            partition by range(name)
        (p1 values <= ("Maarten"),
        p2 values <= ("Zymmerman"))
```

4.5.3 Create a Hash-Partitioned Table

With hash partitioning, all rows are guaranteed to belong to some partition. There is no possibility that inserts and updates will fail to find a partition, which is not the case for range- or list-partitioned tables.

This example creates a table with three hash partitions:

```
create table mysalesdetail
    (store_id char(4) not null,
    ord_num varchar(20) not null,
    title_id tid not null,
    qty smallint not null,
    discount float not null)
partition by hash (ord_num)
(p1 on seg1, p2 on seg2, p3 on seg3)
```

Hash-partitioned tables are easy to create and maintain. SAP ASE chooses the hash function and attempts to distribute the rows equally among the partitions.

4.5.4 Create a List-Partitioned Table

List partitioning controls how individual rows map to specific partitions. List partitions are not ordered and are useful for low cardinality values. Each partition value list must have at least one value, and no value can appear in more than one list.

This example creates a table with two list partitions:

```
create table my_publishers
```

```
(pub_id char(4) not null,
pub_name varchar(40) null,
city varchar(20) null,
state char(2) null)
partition by list (state)
(west values ('CA', 'OR', 'WA') on seg1,
east value ( 'NY', 'NJ') on seg2)
```

An attempt to insert a row with a value in the `state` column other than one provided in the list fails. Similarly, an attempt to update an existing row with a key column value other than one provided in the list fails. As with range-partitioned tables, the values in each list act as a check constraint on the entire table.

4.5.5 Create a Round-Robin–Partitioned Table

This partitioning strategy is random as no partitioning criteria are used. Round-robin-partitioned tables have no partition keys.

This example specifies round-robin partitioning:

```
create table currentpublishers
(pub_id char(4) not null,
pub_name varchar(40) null,
city varchar(20) null,
state char(2) null)
partition by roundrobin 3 on (seg1)
```

All partition-aware utilities and administrative tasks are available for round-robin partitioned tables—whether or not semantic partitioning has been licensed or configured.

4.5.6 Create Partitioned Indexes

Indexes can be created in serial or parallel mode, however, you can create global indexes on round-robin-partitioned tables only in parallel mode.

SAP ASE supports local clustered indexes and local nonclustered indexes on all types of partitioned tables. A local index inherits the partition types, partitioning columns, and partition bounds of the base table.

For range-, hash-, and list-partitioned tables, SAP ASE always creates local clustered indexes, whether or not you include the keywords `local index` in the `create index` statement.

This example creates a local, clustered index on the partitioned `mysalesdetail` table. In a clustered index, the physical order of index rows must be the same as that of the data rows; you can create only one clustered index per table.

```
create clustered index clust_idx
    on mysalesdetail(ord_num) local index
```

This example creates a local, nonclustered index on the partitioned `mysalesdetail` table. The index is partitioned by `title_id`. You can create as many as 249 nonclustered indexes per table.

```
create nonclustered index nonclust_idx
        on mysalesdetail(title_id)
```

```
    local index p1 on seg1, p2 on seg2, p3 on seg3
```

Create Clustered Indexes on Partitioned Tables

You can create a clustered index on a partitioned table if the `select into/bulkcopy/pllsort` database option is true, and if there are as many worker threads available as there are partitions.

To speed recovery, dump the database after creating the clustered index.

See a system administrator or the database owner before creating a clustered index on a partitioned table.

4.5.7 Create a Partitioned Table From an Existing Table

To create a partitioned table from an existing table, use the `select into` command.

You can use `select` with the `into_clause` to create range-, hash-, list-, or round-robin–partitioned tables. The table from which you select can be partitioned or unpartitioned. See the *Reference Manual: Commands*.

> **Note**
>
> You can create temporary partitioned tables in `tempdb` using `select` with the `into_clause` in your applications.

For example, to create the partitioned `sales_report` table from the `salesdetail` table, enter:

```
select * into sales_report partition by range (qty)
(smallorder values <= (500) on seg1,
bigorder values <= (5000) on seg2)
from salesdetail
```

4.6 Change Data Partitions

The `alter table` alters data partitions.

You can use the `alter table` command to:

- Change an unpartitioned table to a multipartitioned table
- Split, merge, or move partitions
- Add one or more partitions to a list- or range-partitioned tables
- Repartition a table for a different partitioning type
- Repartition a table for a different partitioning key or bound
- Repartition a table for a different number of partitions
- Repartition a table to assign partitions to different segments

The general procedures for repartitioning a table are:

1. If the partition key or type is to change during the repartition process, drop all indexes on the table.
2. Use `alter table` to repartition the table.
3. If the partition key or type changed during the repartition process, re-create the indexes on the table.

Change an Unpartitioned Table to a Partitioned Table

An example of an unpartitioned `titles` table changing to a table with three range partitions.

```
alter table titles partition by range (total_sales)
(smallsales values <= (500) on seg1,
mediumsales values <= (5000) on seg2,
bigsales values <= (25000) on seg3)
```

4.6.1 Split, Merge, and Move Partitions

Over time, a partition's data distribution might become skewed, or the manner in which the data was originally partitioned may no longer suit the current business requirements. Use `alter table` to merge, split, or move partitions to redistribute the data and revive performance benefits using partitions.

For example, a company may split partitions to better access its data according to four regions —North, South, East and West. The split partitions allow customer representatives fast and efficient access to their regions' customers, independent of other regions. If sales increase in the Southern region and the customer base has expanded significantly, frequent queries involving partition scans and maintenance operations may cause the South partition to be slow and inefficient, losing out on the benefits of partitioning the customer data. In this situation, splitting the data in the South partition into two partitions, South-East and South-West, may revive performance without affecting the data in other partitions.

A company may merge partitions for better performance because their sales data is partitioned into the four yearly quarters—partitions Q1, Q2, Q3, and Q4. At the end of the year, the company merges the data for the year and archives it. Merging the partitions is efficient because the sales data for a past year is accessed infrequently, and the older data is most likely to be read but not updated.

> **Note**
>
> During a partition split, merge, or move, SAP ASE takes an exclusive lock on the table on which it performs the operation, and the system table entries corresponding to the table.

4.6.1.1 Partition Schemes Available for Splitting or Merging

`alter table` allows you to split, merge, or move range and list partitioning schemes.

Because the location for the data in round-robin partitioned tables is distributed randomly among the data partitions, there is no need to split or merge round-robin partitions.

For hash-partitioned tables, use the repartition utility to redistribute the data.

4.6.1.2 Split Partitions

Use the `alter table ... split partition` parameter to redistribute data to two or more partitions.

The syntax is:

```
alter table <table_name>
split partition <partition_name>
into <partition_condition_clause>
```

where:

- `<partition_name>` – the partition you are splitting.
- `<partition_condition_clause>` – conditions that specify how to split the source partition data. Typically, conditions consist of a numerical range or a data range. The partition conditions should cover all, and only, the data in the source partition.
  `<partition_condition_clause>` may be on the same segment as the source partition, or on a new segment. If you do not specify destination partition segments, SAP ASE creates the new partitions on the segment on which the source partition resides.

See *Reference Manual: Commands*.

You must enable `select into/bulkcopy` to issue `alter table ... split partition`. By default, `alter table ... split partition` rebuilds the section of the local or global index on the partitioned table affected by the split operation.

Except for the step that rebuilds the index, `alter table ... split partition` is not a logged operation. SAP recommends that you perform a database dump after running the `alter table ... split partition` command.

This example creates the `orders` table and then splits its partitions to redistribute the data:

```
create table orders (orderid int, amount float, orderdate datetime)
partition by range (amount)
( P1 values <= (10000) on seg1,
  P2 values <= (50000) on seg2,
  P3 values <= (100000) on seg3,
  P4 values <= (MAX) on seg4)
create clustered index ind_orderid
on orders(orderid) local index (i1 on seg1, i2 on seg2, i3 on seg3, i4 on seg4)
alter table orders
split partition P2
into
( P5 values <= (25000) on seg2,
```

```
  P6 values <= (50000) on seg3)
alter table orders
split partition P3
into
( P7 values <= (50000) on seg2,
  P8 values <= (100000) on seg3)
alter table orders
split partition P4
into
( P9 values <= (200000),
  P10 values <= (MAX))
```

4.6.1.3 Merge Partitions

Use `alter table ... merge partition` to combine the data from two or more merge-compatible (that is, available for the merge) partitions into a single partition.

Whether partitions are merge compatible depends on how they are partitioned:

- For list-partitioned tables, any two partitions are merge-compatible
- For range-partitioned tables, partitions must be adjacent to be merge-compatible

The syntax is:

```
alter table <table_name>
merge partition  {<partition_name>  [{, <partition_name>}…]}
into <destination_partition_name> [on <segment_name>]
```

where:

- `<partition_name>` – the source partitions you are merging. All source partitions must be on the same segment.
- `<destination_partition_name>` – a new or existing partition. If `<destination_partition_name>` is an existing partition, it cannot be any of the source partitions you are merging.
  The partition condition for the merged destination partition is derived from the partition conditions of all the source data partitions being merged, so the destination partition includes all the data residing in the source data partitions being merged. For example, for a list-partitioned table, the new partition condition for the merged partition is the union of all the values that form the source data partition conditions.

See *Reference Manual: Commands*.

You must enable `select into/bulkcopy` to issue `alter table ... merge partition`.

`alter table ... merge partition` is fully logged. Use the transaction dump to recover from a server failure.

This example creates the `sales` table and then merges its partitions to consolidate the data:

```
create table sales(salesmanid int, salesdate datetime, salesregion varchar(10))
partition by range(salesdate)
( Q1 values <= ('31 Mar 2007'),
  Q2 values <= ('30 Jun 2007'),
  Q3 values <= ('30 Sep 2007'),
  Q4 values <= ('31 Dec 2007'))
create index ind_region on sales(salesregion)
alter table sales
merge partition Q3
```

```
into Q4
alter table sales
merge partition Q1, Q2, Q3, Q4
into Y2007
```

4.6.1.4 Move Partitions

Use `alter table ... move partition` to move a partition (and its index) to a specified segment.

The syntax is:

```
alter table <table_name>
move partition <partition_name>
to <destination_segment_name>
```

where:

- `<partition_name>` – the partition you are moving.
- `<destination_segment_name>` – a new or existing segment to which you are moving the partition. You cannot specify "default" as the `<destination_segment_name>`.

See the *Reference Manual: Commands*.

You must enable `select into/bulkcopy` to issue `alter table ... move partition`.

4.6.1.5 Effect of Split or Merged Partitions on Indexes

SAP ASE rebuilds all affected indexes when you perform a `split`, `merge`, or `move partition` on a table with indexes.

> **Note**
>
> Merging or splitting partitions removes statistics from `systabstats` and `sysstatistics`. SAP recommends that you run `update statistics` after merging or splitting partitions.

Table 1: Splitting and Merging Partitions on Indexes

Command	Global Nonclustered Index	Local Index	Local Clustered Index	Local Nonclustered Index
`split partition`	Index is rebuilt	All affected index partitions are rebuilt	Rebuilds all index partitions	Rebuilt on default segment
`merge partition`	No effect if the source and destination segments are the same	All affected index partitions are rebuilt	Rebuilds all index partitions	Rebuilt on default segment

If the table you are splitting or merging includes indexes on separate segments, the segments on which the newly rebuilt indexes reside depend on the type of index.

Table 2: Effect of Split Partition on Index Segments

Type of Index	After the Split or Merge Operation
Global nonclustered index	The index remains on the same segment as prior to the operation.
Local nonclustered index	• New index partitions (corresponding to the split or merge destination data partitions) are placed on the segment specified at the index level. • The indexes are placed on the default segment if you do not specify an index segment. • All unaffected index partitions (corresponding to other data partitions that were not involved in the split or merge) remain on the same segment as prior to the split or merge operation
Local clustered index	• New index partitions are placed on the same segment on which the corresponding destination data partition is placed. • The unaffected index partitions (corresponding to other data partitions not involved in the split or merge) remain on the same segment as prior to the split or merge operation.

4.6.2 Add Partitions to a Partitioned Table

You can add partitions to list- or range-partitioned tables, but you cannot add partitions to a hash- or round-robin–partitioned table.

This example adds a new partition to a range-partitioned table using the existing partition-key column:

```
alter table titles add partition
(vbigsales values <= (40000) on seg4)
```

> **Note**
>
> You can add partitions only to the high end of existing range-based partitions. If you have defined `values <= (MAX)` on a partition, you cannot add new partitions.

Adding a partition to list- or range-partitioned tables does not involve a data copy. The newly created partition is empty.

4.6.3 Change the Partitioning Type or Key

Before changing the partition type or key, you must drop all indexes.

Change the Partition Type

This example repartitions `titles` by hash on the `title_id` column:

```
alter table titles partition by hash(title_id)
    3 on (seg1, seg2, seg3)
```

Repartition `titles` again by range on the `total_sales` column.

```
alter table titles partition
        by range (total_sales)
    (smallsales values <= (500) on seg1,
    mediumsales values <= (5000) on seg2,
    bigsales values <= (25000) on seg3)
```

Change the Partition Key

This example changes the partition key but not the partitioning type:

```
alter table titles partition by range(pubdate)
    (q1 values <= ("3/31/2006"),
    q2 values <= ("6/30/2006"),
    q3 values <= ("9/30/2006"),
    q1 values <= ("12/31/2006"))
```

Related Information

Add Partitions to a Partitioned Table [page 158]

4.6.4 Unpartition Round-Robin–Partitioned Tables

You can create an unpartitioned round-robin table from a partitioned round-robin table using `alter table` with the `unpartition` clause—as long as all partitions are on the same segment, and there are no indexes on the table.

This capability is useful when you are loading large amounts of data into a table that will eventually be used as an unpartitioned table.

Related Information

Using Partitions to Load Table Data [page 164]

4.6.5 partition Parameter Usage

You can use the `partition <number_of_partitions>` parameter to change an unpartitioned round-robin table to a round-robin-partitioned table that has a specified number of partitions.

SAP ASE places all existing data in the first partition. The remaining partitions are created empty; they are placed on the same segment as the first existing partition. Data inserted later is distributed among all partitions according to the round-robin strategy.

If a local index is present on the initial partition, SAP ASE builds empty local indexes on the new partitions. If, when you created the table, you declared a segment, SAP ASE places the new partitions on that segment; otherwise, the partitions are placed on the default segment specified at the table and index level.

For example, you can use `partition <number_of_partitions>` on the `discounts` table in `pubs2` to create three round-robin partitions:

```
alter table discounts partition 3
```

> **Note**
>
> `alter table` with the `partition` clause is supported only for the creation of round-robin partitions. It is not supported for the creation of other types of partitions.

4.6.6 Change Partition-Key Columns

Certain rules apply when you modify partition-key columns.

These rules are:

- You cannot drop a column that is part of the partition key. You can drop columns that are not part of the partition key.
- If you change the datatype of a column that is part of the partition key for a range-partitioned table, the bound of that partition is converted to the new datatype, with these exceptions:
  - Explicit conversions
  - Implicit conversions that result in data loss
  - Conversions from nonbinary datatypes to binary datatypes

In certain cases, if you modify the datatype of a partition-key column or columns, data may redistribute among the partitions:

- For range partitions – if some partition-key values are close to the partition bounds, a datatype conversion may cause those rows to migrate to another partition.

For example, suppose the original datatype of the partition key is `float`, and it is converted to `integer`. The partition bounds are: `p1 values <= (5), p2 values <= (10)`. A row with a partition key of 5.5 is converted to 5, and the row migrates from `p2` to `p1`.

- For range partitions – if the sort order changes because the partition-key datatype changes, all data rows are repartitioned according to the new sort order. For example, the sort order changes if the partition-key datatype changes from `varchar` to `datetime`.
  `alter table` fails if you attempt to alter the datatype of a partition-key column, and, after conversion, the new bound does not maintain the necessary ascending order, or not all rows fit in the new partitions. See *Handling suspect partitions*, in, *Configuring Character Sets, Sort Orders, and Languages*, in the *System Administration Guide: Volume 1*.
- For hash partitions – both the data value and the storage size of the partition-key datatype are used to generate the hash value. As a consequence, changing the datatype of the hash partition key may result in data redistribution.

Related Information

Restrictions on Partition Keys and Bound Values for Range-Partitioned Tables [page 150]

4.7 Configure Partitions

You can configure partitions to improve performance.

The configuration parameters for partitions are:

- `number of open partitions` – specifies the number of partitions that SAP ASE can access simultaneously. The default value is 500.
- `partition spinlock ratio` – specifies the number of spinlocks used to protect against concurrent access of open partitions. The default value is 10.

See, *Setting Configuration Parameters*, the *System Administration Guide: Volume 1*.

4.8 update, delete, and insert in Partitioned Tables

The syntax to update, insert, and delete data in partitioned tables is the same as for unpartitioned tables. You cannot specify a partition in `update`, `insert`, and `delete` statements.

In a partitioned table, data resides on the partitions, and the table becomes a logical union of partitions. The exact partition on which a particular data row is stored is transparent to the user. SAP ASE determines which partitions are to be accessed through a combination of internal logic and the table's partitioning strategy.

SAP ASE aborts any transaction that attempts to insert a row that does not qualify for any of the table's partitions. In a round-robin– or hash-partitioned table, every row qualifies. In a range- or list-partitioned table, only those rows that meet the partitioning criteria qualify.

- For range-partitioned tables – insertions of data rows with values that exceed the upper range defined for the table abort unless the MAX range is specified. If the MAX range is specified, all rows qualify at the upper end.
- For list-partitioned tables – insertions of data rows with partition column values that do not match the partitioning criteria fail.

If the partition-key column for a data row is updated so that the key column value no longer satisfies the partitioning criteria for any partition, the update aborts.

Related Information

Update Values in Partition-Key Columns [page 162]

4.9 Update Values in Partition-Key Columns

For semantically partitioned tables, updating the value in a partition-key column can move the data row from one partition to another.

SAP ASE updates partition-key columns in deferred mode when a data row must move to another partition. A deferred update is a two-step procedure in which the row is deleted from the original partition and then inserted in the new partition.

Such an operation on data-only-locked tables causes the row ID (RID) to change, and may result in scan anomalies. For example, a table may be created and partitioned by range on column `a`:

```
create table test_table (a int) partition by range (a)
    (partition1 <= (1),
    partition2 <= (10))
```

The table has a single row located in `partition2`. The partition-key column value is 2. `partition1` is empty. Assume the following:

```
Transaction T1:
    begin tran
    go
    update table set a = 0
    go
```

```
Transaction T2:
    select count(*) from table isolation level 1
    go
```

Updating T1 causes the single row to be deleted from `partition2` and inserted into `partition1`. However, neither the `delete` nor the `insert` is committed at this point. Therefore, `select count(*)` in T2 does not block on the uncommitted `insert` in `partition1`. Rather, it blocks on the uncommitted `delete` in `partition2`. If T1 commits, T2 does not see the committed `delete`, and returns a count value of zero (0).

This behavior can be seen in `inserts` and `deletes` on data-only-locked tables that do not use partitions. It exists for `updates` only when the partition-key values are updated such that the row moves from one partition

to another. See, *Controlling Physical Data Placement*, in the *Performance and Tuning Series: Physical Database Tuning* and, *Indexes*, in the *Performance and Tuning Series: Locking and Concurrency Control*.

4.10 Display Information About Partitions

Use `sp_helpartition` to view information about partitions.

For example, to view information about the `p1` partition in `publishers`, enter:

```
sp_helpartition publishers, null, p1
```

See the *Reference Manual: Procedures*.

4.10.1 Function Usage

There are several functions you can use to display partition information.

See the *Reference Manual: Building Blocks* for complete syntax and usage information.

- `data_pages` – returns the number of pages used by a table, index, or partition.
- `reserved_pages` – returns the number of pages reserved for a table, index, or partition.
- `row_count` – estimates the number of rows in a table or partition.
- `used_pages` – returns the number of pages used by a table, index, or partition. Unlike `data_pages`, `used_pages` includes pages used by internal structures.
- `partition_id` – returns the partition ID of a specified partition for specified index.
- `partition_name` – returns the partition name that corresponds to the specified index and partition IDs.

Examples

This example returns the number of pages used by the object with an ID of 31000114 in the specified database. The number of pages includes those for indexes.

```
data_pages(5, 31000114)
```

This example returns the partition ID corresponding to the `testtable_ptn1` partition.

```
select partition_id("testtable", testtable_ptn1")
```

This example returns the partition name for the partition ID 1111111111 belonging to the base table with an index ID of 0.

```
select partition_name(0, 1111111111)
```

4.11 Truncate a Partition

You can delete all the information in a partition without affecting information in other partitions.

For example, to delete all rows from the `q1` and `q2` partitions of the `fictionsales` table, enter:

```
truncate table fictionsales partition q1
```

```
truncate table fictionsales partition q2
```

See the *Reference Manual: Commands*.

4.12 Using Partitions to Load Table Data

You can use partitioning to expedite loading large amounts of table data, even if the table will eventually be used as an unpartitioned table.

Procedure

1. Create an empty table, and partition it `<n>` ways:

   ```
   create table currentpublishers
   (pub_id char(4) not null,
   pub_name varchar(40) null,
   city varchar(20) null,
   state char(2) null)
   partition by roundrobin 3 on (seg1)
   ```

2. Run `bcp in` using the `<partition_id>` option. Copy presorted data into each partition. For example, to copy `datafile1.dat` into the first partition of `currentpublishers`, enter:

   ```
   bcp pubs2..currentpublishers:1 in datafile1.dat
   ```

3. Unpartition the table:

   ```
   alter table currentpublishers unpartition
   ```

4. Create a clustered index:

   ```
   create clustered index pubnameind
           on currentpublishers(pub_name)
   ```

   ```
           with sorted_data
   ```

Results

When the partitions are created, SAP ASE places an entry for each one in the `syspartitions` table. `bcp in` with the `<partition_id>` option loads data into each partition in the order listed in `syspartitions`. You unpartitioned the table before creating the clustered index to maintain this order.

4.13 Update Partition Statistics

The SAP ASE query processor uses statistics about the tables, indexes, partitions, and columns in a query to estimate query costs. The query processor chooses the access method that it determines to be the least expensive. But to do so, it must have accurate statistics.

Some statistics are updated during query processing. Others are updated only when you run the `update statistics` command or create indexes.

`update statistics` helps SAP ASE make the best decisions by creating histograms for each major attribute of the local indexes for a partition, and creating densities for the composite attributes. Use `update statistics` when a large amount of data in a partitioned table has been added, changed, or deleted.

Permission to issue `update statistics` and `delete statistics` defaults to the table owner and is not transferable. `update statistics` commands let you update statistics for individual data and index partitions. `update statistics` commands that yield information on partitions include:

- `update statistics`
- `update table statistics`
- `update all statistics`
- `update index statistics`
- `delete statistics`

For example, to update statistics for the `smallvalues` partition of the `titles` table created previosuly, enter:

```
update statistics titles partition smallvalues
```

See the *Reference Manual: Commands*.

4.14 Improved Concurrency for Partition-Level Online Operations

Certain partition-level operations can concurrently operate on different partitions of a table. DML can also concurrently operate on the table while the partition-level online operation is running.

These include:

- `alter table … split partition`

- `alter table … merge partition`
- `alter table … move partition`
- `alter table … drop partition`
- `truncate partition`
- `dbcc checkindex`
- `dbcc checktable`
- `dbcc tablealloc`
- `dbcc indexalloc`

4.14.1 Partition-Level Online Operation Syntax

To allow for greater data availability, `alter table` and `truncate partition` commands include the `with online` subclause in the partition clause.

Context

Partition locking must first be enabled.

The syntax for `alter table` is:

```
alter table <table_name >
{merge | drop | move | split} partition <partition_name>
<existing_clause>
with online
```

where:

`<table_name>` – is the name of the table to modify.

`<partition_name>` – specifies the name of the partition to merge, drop, move, or split.

`<existing_clause>` – is the partition subclause, depending on the partition action specified.

`with online` – executes in an online mode. Enables concurrent access to the table.

The syntax for `truncate partition` is:

```
truncate table table_name
   [partition <partition_name>]
   [with online]
```

where:

`<table_name>` – is the name of the table to truncate.

`<partition_name>` – specifies the name of the partition to truncate.

`with online` – executes in an online mode. Enables concurrent access to the table.

4.14.2 Concurrency with Partition-Level Online Operations

Multiple partition-level online operations can be executed concurrently on different partitions of a table.

All DMLs — that is, `select` (but not `select into`), `insert`, `update`, and `delete` — can operate on a table while partition-level online operations are in progress.

DMLs can operate on all the partitions other than the partitions being operated by partition-level online operations.

DMLs with appropriate use of predicates leads to SAP ASE making use of the partition elimination technique. This may lead to DMLs operating on only minimal required set of partitions.

Without partition elimination, DMLs operate on all partitions of table.

DMLs will be aborted when it operates on the partition that is concurrently being operated by partition-level operations.

> **Note**
>
> To avoid concurrent DMLs leading to errors, DMLs can be written using appropriate predicates that makes use of partition elimination technique. This results in DMLs operating on a minimal set of partitions.

For partition-level online operations, such as splitting a partition or moving a partition, the table must have a local unique index. The move command is allowed for concurrent DMLs to all partitions in the table, including ones being operated by split.

4.14.3 Partition-Level Online Operations with Global Index

Concurrent DML access to a table does not use global indexes while a partition-level operation is in progress on the table. SAP ASE uses alternate local index scan or table scan for concurrent DML scans.

Multiple partition-level operations on different partitions can operate concurrently. Each one does not perform global index rebuild. Only the last committed partition-level operation performs global index rebuild.

The last partition-level operation leading to abort may result in global index marked as suspect. The global index in such case has to be explicitly rebuilt.

5 Virtually Hashed Tables

You can perform hash-based index scans using nonclustered indexes or clustered indexes on data-only-locked tables.

> **Note**
>
> Virtually hashed tables are available on IBM Linux pSeries and Linux AMD64.

During the scan, each worker process navigates the higher levels of the index and reads the leaf-level pages of the index. Each worker process then hashes on either the data page ID or the value in a separate hash table to determine which data pages or data rows to process.

A virtually hashed table can be an efficient way to organize a table because it does not require a separate hash table. Instead, it stores the rows so that, using the hash key, the query processor can determine the row ID (based on the row's ordinal number) and the location of the data. Because it does not use a separate hash table to hold the information, it is called a "virtually" hashed table.

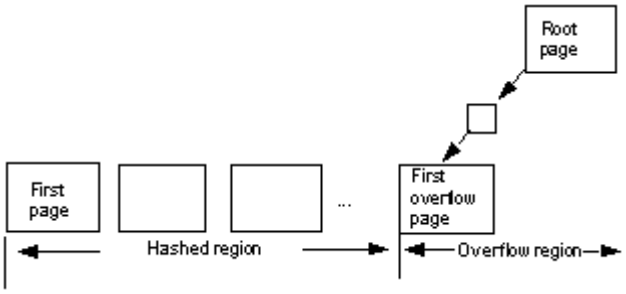
For systems that require efficient CPU usage, the virtually hashed table is a good option.

Clustered and nonclustered indexes are expensive for tables that are used for lookups, or for tables in which the row position does not change. With recent advancements in L2 and L3 CPU architectures, you must utilize the cache to take advantage of the real CPU computing power. If you do not utilize the cache, the CPU spends needless cycles waiting for available memory. For clustered or nonclustered indexes, the server misses rows every time it accesses the index-level search, which consumes many CPU cycles. Virtually hashed tables access row-location patterns by computing the hash-key value instead of performing a search.

5.1 Structure of a Virtually Hashed Table

A virtually hashed table contains a "hashed" region and an "overflow" region. The hashed region stores the hashed rows, and the overflow region stores the remaining rows. You can access the overflow region with a regular clustered index scan using a B-tree clustered index.

This figure demonstrates accessing the overflow region with a root page.



The first data page, the root page, and the first overflow page of a virtually hashed table are created when you create the table. `SYSINDEXES.indroot` is the root page for the overflow clustered region. The first leaf page under this page is the first overflow page. `SYSINDEXES.indfirst` points to the first data page, so a table scan starts at the beginning of the table and scans the entire table.

5.2 Create a Virtually Hashed Table

To create a virtually hashed table, specify the maximum value for the hash region.

This is the partial syntax for `create table`; the parameters for virtually hashed tables are shown in bold:

```
create table [database.[owner].]<table_name>
. . .
             | {unique | primary key}
             using clustered
             (<column_name> [asc | desc] [{, <column_name> [asc | desc]}...])=
             (hash_factor [{, hash_factor}...])
            with max num_hash_values key
```

where:

- `using clustered` – indicates you are creating a virtually hashed table. The list of columns are treated as key columns for this table.
- `<column_name>` [asc | desc] – because rows are placed based on their hash function, you cannot use [asc | desc] for the hash region. If you provide an order for the key columns of virtually hashed tables, it is used only in the overflow clustered region.
- `<hash_factor>` – required for the hash function for virtually hashed tables. For the hash function, a hash factor is required for every key column. These factors are used with key values to generate hash value for a particular row.
- `with max <num_hash_values> key` – the maximum number of hash values that you can use. Defines the upper bound on the output of this hash function.

Determining Values for hash_factor

You can keep the hash factor for the first key as 1. The hash factor for all the remaining key columns is greater than the maximum value of the previous key allowed in the hash region multiplied by its hash factor.

SAP ASE allows tables with hash factors greater than 1 for the first key column to have fewer rows on a page. For example, if a table has a hash factor of 5 for the first key column, after every row in a page, space for the next four rows is kept empty. To support this, SAP ASE requires five times the amount of table space.

If the value of a key column is greater than or equal to the hash factor of the next key column, the current row is inserted in the overflow clustered region to avoid collisions in the hash region.

For example, `t` is a virtually hashed table with key columns `id` and `age`, and corresponding hash factors of (10,1). Because the hash value for rows (5, 5) and (2, 35) is 55, this may result in a hash collision.

However, because the value 35 is greater than or equal to 10 (the hash factor for the next key column, `id`), SAP ASE stores the second row in the overflow clustered region, avoiding collisions in the hash region.

In another example, if `u` is a virtually hashed table with a primary index and hash factors of (id1, id2, id3) = (125, 25, 5) and a `<max hash_value>` of 200:

- Row (1,1,1) has a hash value of 155 and is stored in the hash region.
- Row (2,0,0) has a hash value 250 and is stored in overflow clustered region.
- Row (0,0,6) has a hash factor of 6 x 5, which is greater than or equal to 25, so it is stored in the overflow clustered region.
- Row (0,7,0) has a hash factor of 7 x 25, which is greater than or equal to 125, so it is stored in the overflow clustered region

This example illustrates how the number of rows in the hash region, row length, and the number of rows per page affect the page layout of hash and overflow regions. It creates a virtually hashed table named `orders` on the `pubs2` database on the `order_seg` segment:

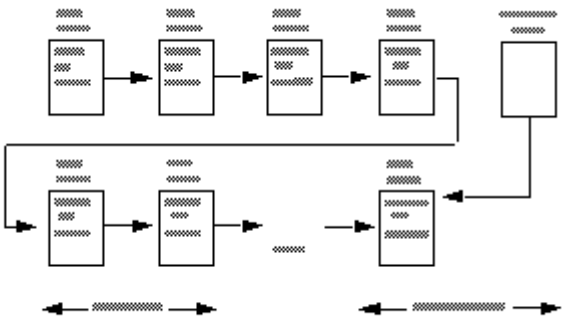
```
create table orders(
id int,
age int,
primary key using clustered (id,age) = (10,1) with max 1000 key)
on order_seg
```

The layout for the data is:

- The `order_seg` segment starts on page ID 51200.
- The logical page size is 2048 bytes
- The ID for the first data object allocation map (OAM) page is 51201.
- For the logical page size of 2048 bytes, the maximum rows per page is 168.
- The row size is 10.
- The root index page of the overflow clustered region is 51217.

In this example:

- The row size is 10 bytes
- 1000 rows fit in the hash region, with key values ranging from (0,0) to (99,9)
- The total number of pages in the hash region is 6, with 168 rows per page in the hash region and a maximum of 1000 keys (ceiling(1000/168) = 6). The last page (the sixth) has some unused space. Assuming the segment starts at page 51200 and the first extent is reserved for the OAM page, the first data page starts from 51208, so pages in the hash region range from 51208 to 51213.
  The page after the last page in hash region (page number 51214) is the first page of the overflow region and is governed by a clustered index, so the root page, 51217, points to page number 51214.



For this page layout, the number of rows per page is 168. Since the hash factors for `id` and `age` are 10 and 1, respectively, the maximum value for column `age` that qualifies for the hash region is 9. The range of key values of (`id` and `age`) combination that qualify for the the hash region (1000 keys in total) is:

- (0, 0) – (0, 9) – for a total of 10
- (1, 0) – (1,9) – for a total of 10
- (2, 0) – (2, 9) – for a total of 10
- . . .
- (99, 0) – (99, 9) – for a total of 10

From these keys, the first 168 keys—(0, 0) to (16, 7)—are mapped to the first data page, 51208. The next range of 168 keys—(16, 8) to (33, 5)—are mapped to the second data page, 51209, and so on.

5.3 Limitations for Virtually Hashed Tables

Virtually hashed tables have certain limitations.

These are the limitations:

- `truncate table` is not supported. Use `delete from table <table_name>` instead.
- SQL92 does not allow two unique constraints on a relation to have the same key columns, so SAP ASE does not support primary-key or unique-key constraint on the same key columns as the key columns of the virtually hashed table.
- Because you cannot create a virtually hashed clustered index after you create a table, you also cannot drop a virtually hashed clustered index.
- You must create a virtually hashed table on an exclusive segment. You cannot share disk devices you assign to the segments for creating a virtually hashed table with other segments. In other words, you must create a special device first, and then create an exclusive segment on the device.
- Virtually hashed tables must have unique rows. Virtually hashed tables do not allow multiple rows with the same key column values because SAP ASE cannot keep one row in the hash region and another with the same key column value in the overflow clustered region.
- You cannot create two virtually hashed tables on the same exclusive segment. SAP ASE supports 32 different segments per database. Three segments are reserved for the default, system, and log segments, so the maximum number of virtually hashed tables per database is 29.
- You cannot use the `alter table` or `drop clustered index` commands on virtually hashed tables.
- Virtually hashed tables must use all-pages locking.
- The key columns and hash factors of a virtually hashed table must use the `int` datatype.
- You cannot include `text` or `image` columns in virtually hashed tables, or columns with datatypes based on the `text` or `image` datatypes.
- You cannot create a partitioned virtually hashed table.

Do not create virtually hashed tables that:

- Have frequent inserts and updates.
- Are partitioned.
- Use frequent table scans.
- Have more data rows in the overflow region than in the hash region. In this situation use a B-tree instead of a virtually hashed table.

5.4 Commands that Support Virtually Hashed Tables

The `dbcc checktable` and `dbcc checkstorage` commands support virtually hashed tables.

- `dbcc checktable` – in addition to the regular checks it performs, `checktable` verifies that the layout of data and OAM pages in the hash region is correct.
  - Data pages are not allocated in an extent reserved for OAM pages as per the layout.
  - The OAM pages are allocated only in the first extent of an allocation unit.
- `dbcc checkstorage` – reports a soft fault if any data page that is not the first data page is empty for tables that are not hashed. However, `dbcc checkstorage` does not report this soft fault for the hashed region of a virtually hashed table. Any data page in the hashed region of a virtually hashed table can be empty.

Related Information

Create a Virtually Hashed Table [page 169]

5.5 Query Processor Support

The query processor uses a virtually hashed index only if you include search arguments that include an equality qualifier (for example, `where id=2`) on all key columns.

If the query processor uses the virtually hashed index, it includes a line similar to this in the `showplan` output:

```
Using Virtually Hashed Index.
```

The query processor includes lines similar to this in the index selection output if it selects a virtually hashed index:

```
Unique virtually hashed index found, returns 1 row, 1 pages
```

5.6 Monitor Counter Support

The `am_srch_hashindex` monitor counter counts the number of times SAP ASE performs a search using a virtually hashed clustered index.

5.7 System Procedure Support

Virtually hashed tables are supported through system procedures.

- `sp_addsegment` – you cannot create a segment on a device that already has an exclusive segment.
- `sp_extendsegment` – you cannot extend a segment on a device that already has an exclusive segment, and you cannot extend an exclusive segment on a device that has another segment.
- `sp_placeobject` – you cannot use `sp_placeobject` on a virtually hashed table, and you cannot place other objects on an exclusive segment.
- `sp_chgattribute` – does not allow you to change attributes for virtually hashed tables.
- `sp_help` – for virtually hashed table, reports:
  - That a table is virtually hashed
  - The `hash_key_factors` for the table
    For example:

```
attribute_class         attribute       int_value
char_value                                comments

---------------------   ---------------  ------------------------
--------------------------------------    -----------
misc table info         hash key factors                     NULL
id:10.0, id2:1.0, max_hash_key=1000.0           NULL
```

6 Create Indexes on Tables

An index provides quick access to data in a table, based on the values in specified columns. A table can have more than one index. Indexes are transparent when accessing data from that table; SAP ASE automatically determines when to use the indexes.

Indexes speed data retrieval by pointing to the location of a table column's data on disk.

```
create index stor_id_ind
on stores (stor_id)
```

The `stor_id_ind` index goes into effect automatically the next time you query the `stor_id` column in `stores`. In other words, indexes are transparent to users. SQL includes no syntax for referring to an index in a query. You can only create or drop indexes from a table; SAP ASE determines whether to use the indexes for each query submitted for that table. As the data in a table changes over time, SAP ASE may change the table's indexes to reflect those changes. Again, these changes are transparent to users.

SAP ASE supports these types of indexes:

- Composite indexes – these indexes involve more than one column. Use this type of index when two or more columns are best searched as a unit because of their logical relationship.
- Unique indexes – these indexes do not permit any two rows in the specified columns to have the same value. SAP ASE checks for duplicate values when the index is created (if data already exists) and each time data is added.
- Clustered or nonclustered indexes – clustered indexes force SAP ASE to continually sort and re-sort the rows of a table so that their physical order is always the same as their logical (or indexed) order. You can have only one clustered index per table. Nonclustered indexes do not require the physical order of rows to be the same as their indexed order. Each nonclustered index can provide access to the data in a different sort order.
- local indexes – local indexes are an index subtree that indexes only one data partition. They can be partitioned, and they are supported on all types of partitioned tables.
- Global indexes – global indexes index span all data partitions in a table. Nonpartitioned, global clustered indexes are supported on round-robin-partitioned tables, and nonclustered global indexes are supported on all types of partitioned tables. You cannot partition global indexes.Clustered and nonclustered global indexes on partitioned tables can only be created using syntax supported in SAP ASE version 12.5.x and earlier.

For information on how you can design indexes to improve performance, see the *Performance and Tuning Series: Locking and Concurrency Control*.

Related Information

Partition Tables and Indexes [page 138]

6.1 Guidelines for Using Indexes

Placing an index on a column often makes the difference between a quick response to a query and a long wait. However, building an index takes time and storage space.

For example, nonclustered indexes are automatically re-created when a clustered index is rebuilt.

Additionally, inserting, deleting, or updating data in indexed columns takes longer than in unindexed columns. However, this cost is usually outweighed by the extent to which indexes improve retrieval performance.

When determining whether or not to create an index, following these general guidelines:

- If you plan to make manual insertions into the IDENTITY column, create a unique index to ensure that the inserts do not assign a value that has already been used.
- A column that is often accessed in sorted order, that is, specified in the `order by` clause, should genereally be indexed, so that SAP ASE can take advantage of the indexed order.
- Columns that are regularly used in joins should always be indexed, since the system can perform the join faster if the columns are in sorted order.
- The column that stores the primary key of the table often has a clustered index, especially if it is frequently joined to columns in other tables. Remember, there can be only one clustered index per table.
- A column that is often searched for ranges of values is often a good choice for a clustered index. Once the row with the first value in the range is found, rows with subsequent values are guaranteed to be physically adjacent. A clustered index does not offer as much of an advantage for searches on single values.

In some cases, indexes are not useful:

- Columns that are seldom or never referenced in queries do not benefit from indexes, since the system seldom has to search for rows on the basis of values in these columns.
- Columns that have many duplicates, and few unique values relative to the number of rows in the table, receive no real advantage from indexing.

If the system does have to search an unindexed column, it does so by looking at the rows one by one. The length of time it takes to perform this kind of scan is directly proportional to the number of rows in the table.

6.2 Methods of Creating Indexes

You can create indexes on tables either by using the `create index` statement, or by using the `unique` or `primary key` integrity constraints of the `create table` command.

However, integrity constraints are limited in the following ways:

- You cannot create nonunique indexes.
- You cannot use the options provided by the `create index` command to tailor how indexes work.
- You can only drop these indexes as a constraint using the `alter table` statement.

If your application requires these features, you should create your indexes using `create index`. Otherwise, the `unique` or `primary key` integrity constraints offer a simpler way to define an index for a table.

Related Information

Databases and Tables [page 50]

6.3 Create Indexes

You must be the owner of a table to `create` or `drop` an index. The owner of a table can `create` or `drop` an index at any time, whether or not there is data in the table. Indexes can be created on tables in another database by qualifying the table name.

Before executing `create index`, turn on `select into`.

The syntax is:

```
sp_dboption,'select into', true
```

The simplest form of `create index` is:

```
create index <index_name>
     on <table_name> (<column_name>)
```

To create an index on the `au_id` column of the `authors` table, execute:

```
create index au_id_ind
on authors(au_id)
```

The index name must conform to the rules for identifiers. The column and table name specify the column you want indexed and the table that contains it.

You cannot create indexes on columns with `bit`, `text`, or `image` datatypes.

> **Note**
>
> The `on <segment _name>` extension to `create index` allows you to place your index on a segment that points to a specific database device or a collection of database devices. Before creating an index on a segment, see a system administrator or the database owner for a list of segments that you can use. Certain segments may already be allocated to specific tables or indexes for performance reasons or for other considerations.

6.3.1 Issue create index in Parallel

SAP ASE includes a parallel form of `create index` that uses the query execution engine to more efficiently execute the command.

6.3.1.1 Configuring enhanced parallel create index

Enhanced parallel `create index` is disabled by default, and is part of the `enable functionality group` configuration parameter.

Context

Procedure

1. Enable the `enable functionality group` configuration parameter:

   ```
   sp_configure "enable functionality group", 1
   ```

2. Set the database option `select into/bulkcopy/pllsort` to true:

   ```
   sp_dboption database_name, "select into", true
   ```

3. Set the following configuration parameter according to your hardware environment:
   - `number of worker processes` – set to the maximum number of concurrently executing parallel threads used by all users
   - `max parallel degree` – set to the `maximum parallel degree` used for an individual user, but not higher than `number of worker processes`
   - `max online engines` – SAP recommends that you configure a sufficient `number of engines` when configuring parallel threads, depending on hardware avaialbility and other workloads. `number of worker processes` is typically set higher than `number of engines`

6.3.1.2 Enhanced Parallel create index Usage

Once SAP ASE is configured for parallel `create index`, it determines if using parallel execution to execute `create index` is the best choice.

If SAP ASE determines that a serial query plan is the most efficient, it does not use a parallel query plan. If SAP ASE determines that a parallel query plan is the most efficient, it selects an enhanced parallel query plan if:

- The table upon which the index is to be created:
  - Uses a data-only-locked format, and
  - Is not partitioned, and
  - The table is not empty
- The index you are creating is a non-clustered index, and
- The leading column of the index has at least two distinct values

Use the `create index ... with consumers = <N>` to force SAP ASE to use parallel query plans when it would typically use a serial query plan. For example, SAP ASE uses parallel query plans for the following even though if the table contains too few rows:

```
create index i1 on t1(c1, c3) with consumers = 3
```

If you use `with consumers` to force a parallel `create index`, and SAP ASE does not select an enhanced parallel query plan, SAP ASE uses a parallel `create index` query plan from a version of SAP ASE earlier than 15.7 ESD #2.

6.3.1.3 View Parallel create index Commands with showplan

If SAP ASE is configured for parallel `create index` and chooses an enhanced parallel `create index` query plan, `showplan` displays information about the `create index` commands below the `PLL CREATE INDEX COORDINATOR` operator and `CREATE INDEX` operators.

For example:

```
create index i1 on t5(c1) with consumers = 3
. . .
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 3 worker processes.
STEP 1
  The type of query is CREATE INDEX.
  5 operator(s) under root
|ROOT:EMIT Operator (VA = 5)
|
|  |PLL CREATE INDEX COORDINATOR Operator
|  |
|  |  |EXCHANGE Operator (VA = 3) (Merged)
|  |  |Executed in parallel by 3 Producer and 1 Consumer processes.
|  |  |
|  |  |  |EXCHANGE:EMIT Operator (VA = 2)
|  |  |  |
|  |  |  |  |CREATE INDEX Operator
|  |  |  |  |
|  |  |  |  |  |SCAN Operator (VA = 0)
|  |  |  |  |  |  FROM TABLE
|  |  |  |  |  |  t5
|  |  |  |  |  |  Table Scan.
|  |  |  |  |  |  Forward Scan.
|  |  |  |  |  |  Positioning at start of table.
|  |  |  |  |  |  Executed in parallel with a 3-way range repartitioning scan.
|  |  |  |  |  |  Using I/O Size 16 Kbytes for data pages.
|  |  |  |  |  |  With MRU Buffer Replacement Strategy for data pages.
```

6.3.2 Function-Based Indexes

Function-based indexes contain one or more expressions as index keys.You can create indexes directly on functions and expressions.

Like computed columns, function-based indexes are helpful for user-defined ordering and decision-support system (DSS) applications, which frequently require intensive data manipulation. Function-based indexes simplify the tasks in these applications and improve performance.

Function-based indexes are similar to computed columns in that they both allow you to create indexes on expressions.

However, there are significant differences:

- A function-based index allows you to index the expression directly. It does not first create the column.
- A function-based index must be deterministic and cannot reference global variables, unlike a computed column.
- You can create a clustered computed column index, but not a clustered function-based index.

Before you can execute `create index`, you must enable the database option `select into`:

```
sp_dboption <dbname>, 'select into', true
```

See the *Reference Manual: Commands* and the *Reference Manual: Procedures*.

Related Information

Computed Columns [page 114]

6.3.3 Create Indexes Without Blocking Access to Data

Use the `create index ... online` parameter to create indexes without blocking access to the data you are indexing.

The syntax is:

```
create [unique] [clustered | nonclustered] index index_name
    on database.]owner.]table_name
    [with {...
            online,
    ...}
```

For example, to create the index `pub_dates_ix` on the `titles` table with the `online` parameter, use:

```
create index pub_dates_ix
on titles (pub_id asc, pubdate desc)
with online
```

Except for the `sorted_data` parameter, SAP ASE processes other `create index` parameters the same way, both with or without the `online` parameter. For example, if you include the `reservepagegap` parameter with

the `online` parameter, SAP ASE reserves the pages while creating the new data layer. However, if you create the index using the `sorted_data` option, SAP ASE creates the index on the existing data layer.

Restrictions

- User tables must include a unique index to use the `create clustered index ... online` command (creating nonclustered indexes does not have this restriction).
- You can run `create index ... online` with a `pll sort` only on round robin partitioned tables
- If you issue an `insert`, `delete`, `update`, or `select` command while `create index … online` or `reorg … online` are in the logical synchronization blocking phase:
  - The `insert`, `delete`, `update`, or `select` commands may wait and execute after `create index … online` or `reorg … online` are finished
  - SAP ASE may issue error message 8233.
- You cannot:
  - Run `dbcc` commands and utility commands, such as `reog rebuild`, on the same table while you are simultaneously running `create index ... online`.
  - Run more than one iteration of `create index ... online` simultaneously.
  - Perform a `dump transaction` after running `create index ... online`. Instead, you can:
    - Run `create index ... online`, then dump the database, or
    - Run a blocking `create index`, then issue `dump transaction`.
  - Run `create index ... online` within a multistatement transaction.
  - Create a functional index using the `online` parameter.

> **Note**
>
> Because `create index ... online` increments the schema count in the `sysobjects` row that reflects the table's state change, concurrent activity waiting for `create index ... online` to commit may encounter error 540 after `create index ... online` commits.

6.3.4 Unique Indexes

A unique index does not allow any two rows to have the same index value, including NULL. If data already exists, the system checks for duplicate values when the index is created and subsequently checks each time data is added or modified with an `insert` or `update`.

Specifying a unique index makes sense only when uniqueness is a characteristic of the data itself. For example, you would not want a unique index on a `last_name` column, because there is likely to be more than one “Smith” or “Wong” in tables of even a few hundred rows.

However, a unique index on a column holding social security numbers is a good idea. Uniqueness is a characteristic of the data—each person has a different social security number. Furthermore, a unique index

serves as an integrity check. For instance, a duplicate social security number probably reflects some kind of error in data entry or on the part of the government.

If you try to create a unique index on data that includes duplicate values, the command is aborted, and SAP ASE displays an error message that gives the first duplicate. You cannot create a unique index on a column that contains null values in more than one row; these are treated as duplicate values for indexing purposes.

If you attempt to insert the same row during two concurrent sessions on a DOL table with a unique index, the first session fails with error number 2601. SAP ASE fails the `insert` to avoid blocking, and leaves the database in a consistent state. Additionally, a blocked second session could impact concurrency.

If you try to change data on which there is a unique index, the results depend on whether you have used the `ignore_dup_key` option.

You can use the `unique` keyword on composite indexes.

Related Information

ignore_dup_key Option [page 186]

6.3.5 IDENTITY Columns in Nonunique Indexes

The `identity in nonunique index` database option automatically includes an IDENTITY column in a table's index keys so that all indexes created on the table are unique.

This option makes logically nonunique indexes internally unique and allows them to process updatable cursors and isolation level 0 reads.

To enable `identity in nonunique indexes`, enter:

```
sp_dboption pubs2, "identity in nonunique index", true
```

The table must already have an IDENTITY column, either from a `create table` statement or by setting the `auto identity` database option to true before creating the table.

Use `identity in nonunique index` to use cursors and isolation level 0 reads on tables with nonunique indexes. A unique index ensures that the cursor is positioned at the correct row the next time a `fetch` is performed on that cursor.

For example, after setting `identity in nonunique index` and `auto identity` to `true`, suppose you create the following table, which has no indexes:

```
create table title_prices
(title varchar(80)  not null,
price  money          null)
```

`sp_help` shows that the table contains an IDENTITY column, `SYB_IDENTITY_COL`, which is automatically created by the `auto identity` database option. If you create an index on the `title` column, use `sp_helpindex` to verify that the index automatically includes the IDENTITY column.

6.3.6 Ascending and Descending Index-Column Values

You can use the `asc` (ascending) and `desc` (descending) keywords to assign a sort order to each column in an index. By default, sort order is ascending.

Creating indexes so that columns are in the same order specified in the `order by` clauses of queries eliminates sorting the columns during query processing. The following example creates an index on the Orders table. The index has two columns, the first is `customer_ID`, in ascending order, the second is `date`, in descending order, so that the most recent orders are listed first:

```
create index nonclustered cust_order_date
     on Orders
     (customer_ID asc,
     date desc)
```

6.3.7 Using fillfactor, max_rows_per_page, and reservepagegap

`fillfactor`, `max_rows_per_page`, and `reservepagegap` are space-management properties that apply to tables, and indexes and affect the way physical pages are filled with data.

Table 3: Summary of Space-Management Properties for Indexes

Property	Description	Use	Comments
`fillfactor`	Specifies the percent of space on a page that can be filled when the index is created. A `fillfactor` less than 100% leaves space for inserts into a page without immediately causing page splits. Benefits: • Initially, fewer page splits. • Reduced contention for pages, because there are more pages and fewer rows on a page.	Applies only to a clustered index on a data-only-locked table.	The `fillfactor` percentage is used only when an index is created on a table with existing data. It does not apply to pages and inserts after a table is created. If no `fillfactor` is specified, the system-wide default `fillfactor` is used. Initially, this is set to 100%, but can be changed using `sp_configure`.
`max_rows_per_page`	Specifies the maximum number of rows allowed per page. Benefit: • Can reduce contention for pages by limiting the number of rows per page and increasing the number of pages.	Applies only to all-pages-locked tables. The maximum value that you can set this property to is 256.	`max_rows_per_page` applies at all times, from the creation of an index, onward. If not specified, the default is as many rows as will fit on a page.

Property	Description	Use	Comments
`reservepagegap`	Determines the number of pages left empty when extents are allocated. For example, a `reservepagegap` of 16 means that 1 page of the 16 pages in 2 extents is left empty when the extents are allocated. Benefits: • Can reduce row forwarding and lessen the frequency of maintenance activities such as running `reorg rebuild` and re-creating indexes.	Applies to pages in all locking schemes.	If `reservepagegap` is not specified, no pages are left empty when extents are allocated.

This statement sets the `fillfactor` for an index to 65% and sets the `reservepagegap` to one empty page for each extent allocated:

```
create index postalcode_ind2
     on authors (postalcode)
     with fillfactor = 10, reservepagegap = 8
```

6.4 Indexes on Computed Columns

You can create indexes on computed columns as though they were regular columns, as long as the datatype of the result can be indexed. Computed column indexes provide a way to create indexes on complex datatypes like `XML`, `text`, `image`, and Java classes.

For example, the following code sample creates a clustered index on the computed columns as though they were regular columns:

```
CREATE CLUSTERED INDEX name_index on parts_table(name_order)
CREATE INDEX adt_index on parts_table(version_order)
CREATE INDEX xml_index on parts_table(spec_index)
CREATE INDEX text_index on parts_table(descr_index)
```

SAP ASE evaluates the computed columns and uses the results to build or update indexes when you create or update an index.

6.5 Clustered or Nonclustered Index Usage

With a clustered index, SAP ASE sorts rows on an ongoing basis so that their physical order is the same as their logical (indexed) order. The bottom or leaf level of a clustered index contains the actual data pages of the table. With a nonclustered index, the physical order of the rows is not the same as their indexed order.

Create the clustered index before creating any nonclustered indexes, since nonclustered indexes are automatically rebuilt when a clustered index is created.

There can be only one clustered index per table. It is often created on the primary key—the column or columns that uniquely identify the row.

Logically, the database's design determines a primary key. You can specify primary key constraints with the `create table` or `alter table` statements to create an index and enforce the primary key attributes for table columns. You can display information about constraints with `sp_helpconstraint`.

Also, you can explicitly define primary keys, foreign keys, and common keys (pairs of keys that are frequently joined) by using `sp_primarykey`, `sp_foreignkey`, and `sp_commonkey`. However, these procedures do not enforce key relationships.

You can display information about defined keys with `sp_helpkey` and about columns that are likely join candidates with `sp_helpjoins`. See the *Reference Manual: Procedures*.

For a nonclustered index, the leaf level contains pointers to rows on data pages. More precisely, each leaf page contains an indexed value and a pointer to the row with that value. In other words, a nonclustered index has an extra level between the index structure and the data itself.

Each of the up to 249 nonclustered indexes permitted on a table can provide access to the data in a different sorted order.

Finding data using a clustered index is almost always faster than using a nonclustered index. In addition, a clustered index is advantageous when many rows with contiguous key values are being retrieved—that is, on columns that are often searched for ranges of values. Once the row with the first key value is found, rows with subsequent indexed values are guaranteed to be physically adjacent, and no further searches are necessary.

If neither the `clustered` nor the `nonclustered` keyword is used, SAP ASE creates a nonclustered index.

Here is how the `titleidind` index on the `title_id` column of the `titles` table is created. To try this command, first drop the index:

```
drop index titles.titleidind
```

Then, create the clustered index:

```
create clustered index titleidind
on titles(title_id)
```

If you think you will often want to sort the people in the `friends_etc` table, which you created earlier by postal code, create a nonclustered index on the `postalcode` column:

```
create nonclustered index postalcodeind
on friends_etc(postalcode)
```

A unique index does not make sense here, since some of your contacts are likely to have the same postal code. A clustered index would not be appropriate either, since the postal code is not the primary key.

The clustered index in `friends_etc` should be a composite index on the personal name and surname columns, for example:

```
create clustered index nmind
on friends_etc(pname, sname)
```

Related Information

Triggers: Enforce Referential Integrity [page 583]
Databases and Tables [page 50]

6.5.1 Create Clustered Indexes on Segments

The `create index` command allows you to create the index on a specified segment.

Since the leaf level of a clustered index and its data pages are the same by definition, creating a `clustered` index and using the `on segment_name` extension moves a table from the device on which it was created to the named segment.

See a system administrator or the database owner before creating tables or indexes on segments; certain segments may be reserved for performance reasons.

6.6 Index Options

The index options `ignore_dup_key`, `ignore_dup_row`, and `allow_dup_row` control what happens when a duplicate key or duplicate row is created with `insert` or `update`.

This table shows which option to use, based on the type of index.

Index Type	Options
Clustered	`ignore_dup_row|allow_dup_row`
Unique clustered	`ignore_dup_key`
Nonclustered	None
Unique nonclustered	None

6.6.1 ignore_dup_key Option

If you need to insert a duplicate value, you can use `ignore_dup_key` option with a unique index.

The unique index can be either clustered or nonclustered. When you begin data entry, any attempt to insert a duplicate key is canceled with an error message. After the cancellation, any transaction that was active at the time may continue as though the `update` or `insert` had never taken place. Nonduplicate keys are inserted normally.

You cannot create a unique index on a column that already includes duplicate values, whether or not `ignore_dup_key` is set. If you attempt to do so, SAP ASE prints an error message and a list of the duplicate values. You must eliminate duplicates before you create a unique index on the column.

Here is an example of using the `ignore_dup_key` option:

```
create unique clustered index phone_ind
on friends_etc(phone)
with ignore_dup_key
```

6.6.2 ignore_dup_row and allow_dup_row

Use `ignore_dup_row` and `allow_dup_row` to create a nonunique, clustered index.

These options are not relevant when creating a nonunique, nonclustered index. Since an SAP ASE nonclustered index attaches a unique row identification number internally, duplicate rows are never an issue—even for identical data values.

`ignore_dup_row` and `allow_dup_row` are mutually exclusive.

A nonunique clustered index allows duplicate keys, but does not allow duplicate rows unless you specify `allow_dup_row`.

If `allow_dup_row` is set, you can create a new nonunique, clustered index on a table that includes duplicate rows, and you can subsequently `insert` or `update` duplicate rows.

If any index in the table is unique, the requirement for uniqueness—the most stringent requirement—takes precedence over the `allow_dup_row` option. Thus, `allow_dup_row` applies only to tables with nonunique indexes. You cannot use this option if a unique clustered index exists on any column in the table.

The `ignore_dup_row` option eliminates duplicates from a batch of data. When you enter a duplicate row, SAP ASE ignores that row and cancels that particular `insert` or `update` with an informational error message. After the cancellation, any transaction that may have been active at the time continues as though the insert or update had never taken place. Nonduplicate rows are inserted normally.

The `ignore_dup_row` applies only to tables with nonunique indexes: you cannot use this keyword if a unique index exists on any column in the table.

This table illustrates how `allow_dup_row` and `ignore_dup_row` affect attempts to create a nonunique, clustered index on a table that includes duplicate rows, and to enter duplicate rows into a table.

Option	Has Duplicates	Enter Duplicates
Neither option set	`create index` command fails.	Command fails.
`allow_dup_row` set	Command completes.	Command completes.
`ignore_dup_row` set	Index created but duplicate rows deleted; error message.	Duplicates not inserted/updated; error message; transaction completes.

6.6.3 sorted_data Option

The `sorted_data` option of `create index` speeds index creation when the data in the table is already in sorted order, for example, when you have used `bcp` to copy sorted data into an empty table.

The speed increase becomes significant on large tables and increases to several times faster in tables larger than 1GB.

If you specify `sorted_data`, but data is not in sorted order, an error message appears and the command is aborted.

`sorted_data` speeds indexing only for clustered indexes or unique nonclustered indexes. Creating a nonunique nonclustered index is, however, successful, unless there are rows with duplicate keys. If there are rows with duplicate keys, an error message appears and the command is aborted.

Certain other `create index` options require a sort even if `sorted_data` is specified. See the *Reference Manual: Commands*.

6.6.4 on segment_name Option

A nonclustered index can be created on a different segment than the data pages. The `on segment_name` clause specifies a database segment name on which the index is to be created.

For example:

```
create index titleind
on titles(title)
on seg1
```

If you use `segment_name` when creating a clustered index, the table containing the index moves to the segment you specify. See a system administrator or the database owner before creating tables or indexes on segments; certain segments may be reserved for performance reasons.

6.7 Drop Indexes

The `drop index` command removes an index from the database, reclaiming their storage space

The `drop index` command cannot be used on any of the system tables in the `master` database or in the user database.

You might want to drop an index if it is not used for most or all of your queries.

For example, to drop the index `phone_ind` in the `friends_etc` table:

```
drop index friends_etc.phone_ind
```

Use `sp_post_xpload` to check and rebuild indexes after a cross-platform `load database` where the endian types are different.

6.8 Identifying the Indexes on a Table

Use `sp_helpindex` to see the indexes that exist on a table.

Here is a report on the `friends_etc` table:

```
sp_helpindex friends_etc
```

```
index_name      index_keys    index_description    index_max_rows_per_page
-------------   ----------   ----------------- --------------------------
nmind           pname,sname   clustered                                  0
postalcodeind   postalcode    nonclustered                               0
index_fillfactor  index_reservepagegap  index_created        index_local
----------------- --------------------  ---------------      ------------
              0                      0  May 24 2005 1:49PM   Global Index
              0                      0  May 24 2005 1:49PM   Global Index
(2 rows affected)
index_ptn_name            index_ptn_seg
------------------        ---------------
nmind_1152004104          default
postalcodeind_1152004104  default
(2 rows affected)
```

`sp_help` runs `sp_helpindex` at the end of its report.

`sp_statistics` returns a list of indexes on a table. For example:

```
sp_statistics friends_etc
```

```
table_qualifier               table_owner
         table_name                       non_unique
         index_qualifier                  index_name
         type   seq_in_index column_name                    collation
         cardinality pages
-------------------------------- ---------------------------------
        -------------------------------- ----------
        -------------------------------- -------------------------------
```

```
        ------ ------------ ------------------------------- ---------
        ----------- -----------
pubs2                           dbo
        friends_etc                           NULL
        NULL                             NULL
             0          NULL NULL                            NULL
                   0           1
pubs2                             dbo
         friends_etc                                1
         friends_etc                         nmind
            1           1 pname                        A
                   0           1
pubs2                               dbo
         friends_etc                                 1
         friends_etc                          nmind
               1            2 sname                          A
                   0            1
pubs2                               dbo
         friends_etc                                  1
         friends_etc                         postalcodeind
             3           1  postalcode                        A
                NULL       NULL

(4 rows affected)
table_qualifier table_owner table_name index_qualifier index_name
       non_unique_type seq_in_index column_name collation index_id
       cardinality pages status status2
---------------- ---------  ---------- --------------- -----------
       ---------------- ------------ ----------- --------- ---------
       ----------- ----- ------ --------
pubs2            dbo         friends_etc friends_etc     nmind
                   1     1              1 pname      A
                    0          1        16       0
pubs2            dbo         friends_etc friends_etc     nmind
                   1     1              2 sname      A
                   0          1        16       0
pubs2            dbo         friends_etc friends_etc     postalcodeind
                   1     3              1 postalcode A
                NULL          NULL       0       0
pubs2             dbo         friends_etc NULL            NULL
                NULL     0      NULL NULL    NULL    0
                    0          1           0       0
(4 rows affected)
(return status = 0)
```

In addition, if you follow the table name with “1”, `sp_spaceused` reports the amount of space used by a table and its indexes. For example:

```
sp_spaceused friends_etc, 1
```

```
index_name           size       reserved   unused
-------------------- ---------- ---------- ----------
nmind                2 KB       32 KB      28 KB
postalcodeind        2 KB       16 KB      14 KB
name         rowtotal   reserved     data    index_size      unused
------------ ---------- ------------ ------- --------------- --------
friends_etc  1          48 KB        2 KB    4 KB            42 KB

(return status = 0)
```

6.9 Update Statistics for Indexes

Use `update statistics` when a large amount of data in an indexed column has been added, changed, or deleted.

This command helps SAP ASE make the best decisions about which indexes to use when it processes a query, by keeping it up to date about the distribution of the key values in the indexes.

When Component Integration Services is enabled, `update statistics` can generate accurate distribution statistics for remote tables. See the *Component Integration Users Guide*.

Permission to issue the `update statistics` command defaults to the table owner and cannot be transferred. Its syntax is:

```
update statistics <table_name> [<index_name>]
```

If you do not specify an index name, the command updates the distribution statistics for all the indexes in the specified table. Giving an index name updates statistics only for that index.

Use `sp_helpindex` to find the names of indexes. See the *Reference Manual: Procedures*.

To update the statistics for all of the indexes, enter the name of the table:

```
update statistics authors
```

To update the statistics only for the index on the `au_id` column, enter:

```
update statistics authors auidind
```

Because Transact-SQL does not require index names to be unique in a database, you must give the name of the table with which the index is associated. SAP ASE automatically runs `update statistics` when you create an index on existing data.

You can set `update statistics` to run automatically during the time that best suits your site, and avoid running it at times that hamper your system.

7 Datatypes

In Transact-SQL, datatypes specify the type of information, size, and storage format of table columns, stored procedure parameters, and local variables.

You can use SAP ASE system datatypes when you are defining columns, or you can create user-defined datatypes.

For example, the `int` (integer) datatype stores whole numbers in the range of plus or minus 2^{31}, and the `tinyint` (tiny integer) datatype stores whole numbers between 0 and 255 only.

SAP ASE supplies several system datatypes, and two user-defined datatypes, `timestamp` and `sysname`. Use `sp_addtype` to build user-defined datatypes based on the system datatypes.

You must specify a system datatype or user-defined datatype when declaring a column, local variable, or parameter. The following example uses the system datatypes `char`, `numeric`, and `money` to define the columns in the `create table` statement:

```
create table sales_daily
    (stor_id char(4),
    ord_num numeric(10,0),
    ord_amt money)
```

The next example uses the `bit` system datatype to define the local variable in the `declare` statement:

```
declare @switch bit
```

Use `sp_help` to determine which datatypes have been defined for columns of existing tables.

7.1 System-Supplied Datatypes

You can use either a system datatype or user-defined datatype when declaring a column, local variable, or parameter.

The system datatypes are printed in lowercase characters, although SAP ASE allows you to enter them in either uppercase or lowercase. Most SAP ASE-supplied datatypes are not reserved words and can be used to name other objects. To build user-defined datatypes that are based on the system datatypes, use `sp_addtype`.

This table shows the SAP ASE system-supplied datatypes:

Datatypes by Category	Synonyms	Range	Bytes of Storage
Exact numeric: integers			

Datatypes by Category	Synonyms	Range	Bytes of Storage
`bigint`		Whole numbers between 2^{63} and $-2^{63} - 1$ (from -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807, inclusive	8
`int`	`integer`	2^{31} -1 (2,147,483,647) to -2^{31} (-2,147,483,648)	4
`smallint`		2^{15} -1 (32,767) to -2^{15} (-32,768)	2
`tinyint`		0 to 255 (negative numbers are not permitted)	1
`unsigned bigint`		Whole numbers between 0 and 18,446,744,073,709,551,615	8
`unsigned int`		Whole numbers between 0 and 4,294,967,295	4
`unsigned smallint`		Whole numbers between 0 and 65535	2
Exact numeric: decimals			
`numeric (precision, scale)`		10^{38} -1 to -10^{38} + 1	2 to 17
`decimal (precision, scale)`	`dec`	10^{38} -1 to -10^{38} + 1	2 to 17
Approximate numeric			
`float (precision)`		Machine-dependent	4 for `default precision` < 16, 8 for `default precision` >= 16
`double precision`		Machine-dependent	8
`real`		Machine-dependent	4
Money			
`smallmoney`		214,748.3647 to -214,748.3648	4

Datatypes by Category	Synonyms	Range	Bytes of Storage
money		922,337,203,685,477.5807 to -922,337,203,685,477.5808	8
Date/time			
smalldatetime		January 1, 1900 to June 6, 2079	4
datetime		January 1, 1753 to December 31, 9999	8
date		January 1, 0001 to December 31, 9999	4
time		12:00:00 a.m to 11:59:59:999 p.m.	4
bigdatetime		January 1, 0001 to December 31, 9999 and 12:00:00.000000 a.m. to 11:59:59.999999 p.m.	8
bigtime		12:00:00.000000 a.m 11:59:59.999999 p.m.	8
Character			
char(n)	character	page size	n
varchar(n)	character varying, char varying	page size	Actual entry length
unichar	Unicode character	page size	<n> * <@@unicharsize> (<@@unicharsize> equals 2)
univarchar	Unicode character varying, char varying	page size	actual number of characters * <@@unicharsize>
nchar(n)	national character, national char	page size	<n> * <@@ncharsize>
nvarchar(n)	nchar varying, national char varying, national character varying	page size	<@@ncharsize> * number of characters
text		2^{31} -1 (2,147,483,647) bytes or fewer	0 when uninitialized; multiple of 2K after initialization

Datatypes by Category	Synonyms	Range	Bytes of Storage
`unitext`		1,073,741,823 Unicode characters or fewer	0 when uninitialized; multiple of 2K after initialization
Binary			
`binary(n)`		pagesize	`<n>`
`varbinary(n)`		pagesize	actual entry length
`image`		2^{31} -1 (2,147,483,647) bytes or fewer	0 when uninitialized; multiple of 2K after initialization
Bit			
`bit`		0 or 1	1 (one byte holds up to 8 `bit` columns)

7.1.1 Exact Numeric Types: Integers

SAP ASE provides various datatypes for storing integers (whole numbers). These types are exact numeric types, which means they preserve their accuracy during arithmetic operations.

Choose among the integer types based on the expected size of the numbers to be stored. Internal storage size varies by datatype.

Implicit conversion from any integer type to a different integer type is supported only if the value is within the range of the type being converted to.

Unsigned integer datatypes allow you to extend the range of the positive numbers for the existing integer types without increasing the required storage size. That is, the signed versions of these datatypes extend both in the negative direction and the positive direction (for example, from -32 to +32). However, the unsigned versions extend only in the positive direction.

Datatype	Range of Signed Datatypes	Datatype	Range of Unsigned Datatypes
`bigint`	Whole numbers between -2^{63} and 2^{63} - 1 (from -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807, inclusive	`unsigned bigint`	Whole numbers between 0 and 18,446,744,073,709,551,615
`int`	Whole numbers between -2^{31} and 2^{31} - 1 (-2,147,483,648 and 2,147,483,647), inclusive	`unsigned int`	Whole numbers between 0 and 4,294,967,295
`smallint`	Whole numbers between -2^{15} and 2^{15} -1 (-32,768 and 32,767), inclusive	`unsigned smallint`	Whole numbers between 0 and 65535

7.1.2 Exact Numeric Types: Decimal Numbers

Use the exact numeric types, `numeric` and `decimal`, for numbers that include decimal points. Data stored in these columns is packed to conserve disk space, and preserves its accuracy to the least significant digit after arithmetic operations.

The `numeric` and `decimal` types are identical in all respects but one: only `numeric` types with a scale of 0 can be used for the identity column.

The exact numeric types accept two optional parameters, `precision` and `scale`, enclosed within parentheses and separated by a comma:

```
<datatype> [(<precision>  [, <scale> ])]
```

SAP ASE defines each combination of `precision` and `scale` as a distinct datatype. For example, `numeric`(10,0) and `numeric`(5,0) are two separate datatypes. The precision and scale determine the range of values that can be stored in a `decimal` or `numeric` column:

- `precision` specifies the maximum number of decimal digits that can be stored in the column. It includes all digits to the right or left of the decimal point. You can specify a precision of 1 – 38 digits, or use the default precision of 18 digits.
- `scale` specifies the maximum number of digits that can be stored to the right of the decimal point. The scale must be less than or equal to `precision`. You can specify a scale of 0 – 38 digits, or use the default scale of 0 digits.

Exact numeric types with a scale of 0 display without a decimal point. You cannot enter a value that exceeds either the precision or the scale for the column.

The storage size for a `numeric` or `decimal` column depends on its precision. The minimum storage requirement is 2 bytes for a 1- or 2-digit column. Storage size increases by 1 byte for each additional 2 digits of precision, to a maximum of 17 bytes.

7.1.3 Approximate Numeric Datatypes

The numeric types `float`, `double precision`, and `real` store numeric data that can tolerate rounding during arithmetic operations.

Approximate numeric datatypes store, as binary tractions, slightly inaccurate representations of real numbers, stored as binary fractions. Anytime an approximate numeric value is shown, printed, transferred between hosts, or used in calculations, the numbers lose precision. `isql` displays only six significant digits after the decimal point, and rounds the remainder. See *System and User-Defined Datatypes*, in the *Reference Manual: Building Blocks*.

Use the approximate numeric types for data that covers a wide range of values. They support all aggregate functions and all arithmetic operations.

The `real` and `double precision` types are built on types supplied by the operating system. The `float` type accepts an optional precision in parentheses. `float` columns with a precision of 1 – 15 are stored as `real`; those with higher precision are stored as `double precision`. The range and storage precision for all three types is machine-dependent.

7.1.4 Money Datatypes

The money datatypes, `money` and `smallmoney`, store monetary data.

You can use these datatypes for U.S. dollars and other decimal currencies, although SAP ASE provides no means to convert from one currency to another. You can use all arithmetic operations except `modulo`, and all aggregate functions, with `money` and `smallmoney` data.

Both `money` and `smallmoney` are accurate to one ten-thousandth of a monetary unit, but round values up to two decimal places for display purposes. The default print format places a comma after every three digits.

7.1.5 Date and Time Datatypes

Use the `datetime` and `smalldatetime` datatypes to store date and time information from January 1, 1753 through December 31, 9999. Use `date` for dates from January 1, 0001 to December 31, 9999 or `time` for 12:00:00 a.m. to 11:59:59:999.

Dates outside this range must be entered, stored, and manipulated as `char` or `varchar` values.

- `datetime` columns hold dates between January 1, 1753 and December 31, 9999. `datetime` values are accurate to 1/300 second on platforms that support this level of granularity. Storage size is 8 bytes: 4 bytes for the number of days since the base date of January 1, 1900 and 4 bytes for the time of day.
- `smalldatetime` columns hold dates from January 1, 1900 to June 6, 2079, with accuracy to the minute. Its storage size is 4 bytes: 2 bytes for the number of days after January 1, 1900, and 2 bytes for the number of minutes after midnight.
- `bigdatetime` columns hold dates from January 1, 0001 to December 31, 9999 and 12:00:00.000000 a.m. to 11:59:59.999999 p.m.. Its storage size is 8 bytes. `bigdatetime` values are accurate to a microsecond. The internal representation of `bigdatetime` is a 64-bit integer containing the number of microseconds since 01/01/0000.
- `bigtime` columns hold times from 12:00:00.000000 a.m. to 11:59:59.999999 p.m.. Its storage size is 8 bytes. The `bigtime` values are accurate to a microsecond. The internal representation of `bigtime` is a 64-bit integer containing the number of microseconds since midnight.
- `date` is a literal value consisting of a date portion in single or double quotes. This column can hold dates between January 1, 0001 to December 31, 9999. Storage size is 4 bytes.
- `time` is a literal value consisting of a time portion enclosed in single or double quotes. This column holds time from 12:00:00a.m. to 11:59:59:999p.m.. Storage size is 4 bytes.

Enclose date and time information in single or double quotes. You can enter it in either uppercase or lowercase letters and include spaces between data parts. SAP ASE recognizes a wide variety of data entry formats. However, SAP ASE rejects values such as 0 or 00/00/00, which are not recognized as dates.

The default display format for dates is “Apr 15 1987 10:23p.m.”. You can use the `convert` function for other formats. You can also perform some arithmetic calculations on `datetime` values with the built-in date functions, although SAP ASE may round or truncate millisecond values, unless you use the `time` datatype.

For `bigdatetime` and `bigtime`, the value that appears reflects microsecond precision. `bigdatetime` and `bigtime` have default display formats that accomodate this increased precision.

- hh:mi:ss.zzzzzzAM or PM

- hh:mi:ss.zzzzzz
- mon dd yyyy hh:mi:ss.zzzzzz
- yyyy-mm-dd hh:mi:ss.zzzzzz

Related Information

Managing Data [page 355]

7.1.6 Character Datatypes

Use the character datatypes to store strings of letters, numbers, and symbols entered within single or double quotes.

Use the `like` keyword to search these strings for particular characters, and the built-in string functions to manipulate their contents. Use the `convert` function to convert strings consisting of numbers to exact and approximate numeric datatypes, which can then be used for arithmetic.

The `char(<n>)` datatype stores fixed-length strings, and the `varchar(<n>)` datatype stores variable-length strings, in single-byte character sets such as English. Their international character counterparts, `nchar(<n>)` and `nvarchar(<n>)`, store fixed- and variable-length strings in multibyte character sets such as Japanese.The `unichar` and `univarchar` datatypes store Unicode characters, which are a constant size. You can specify the maximum number of characters with `n` or use the default column length of one character. For strings larger than the page size, use the `text` datatype.

Datatype	Stores
`char(n)`	Fixed-length data, such as social security numbers or postal codes
`varchar(n)`	Data, such as names, that is likely to vary greatly in length
`unichar`	Fixed-length Unicode data, comparable to `char`
`univarchar`	Unicode data that is likely to vary greatly in length, comparable to `varchar`
`nchar(n)`	Fixed-length data in multibyte character sets
`nvarchar(n)`	Variable-length data in multibyte character sets
`text`	Up to 2,147,483,647 bytes of printable characters on linked lists of data pages
`unitext`	Up to 1,073,741,823 Unicode characters on linked lists of data pages

SAP ASE truncates entries to the specified column length without warning or error, unless you set `string_rtruncation on`. See the *Reference Manual: Commands*. The empty string, “ ”or ‘ ’, is stored as a single space rather than as NULL. Thus, “abc” + “ ” + “def” is equivalent to “abc def”, not to “abcdef”.

Fixed- and variable-length columns behave somewhat differently:

- Data in fixed-length columns is blank-padded to the column length. For `char` and `unichar` datatypes, storage size is `<n>` bytes, (`unichar` = `n*@@unicharsize`); for `nchar`, `<n>` times the average national character length (`<@@ncharsize>`). When you create a `char`, `unichar`, or `nchar` column that allows nulls, SAP ASE converts it to a `varchar`, `univarchar`, or `nvarchar` column and uses the storage rules for those datatypes. This is not true of `char` and `nchar` variables and parameters.
- Data in variable-length columns is stripped of trailing blanks; storage size is the actual length of the data. For `varchar` or `univarchar` columns, this is the number of characters; for `nvarchar` columns, it is the number of characters times the average character length. Variable-length character data may require less space than fixed-length data, but is accessed somewhat more slowly.

7.1.6.1 unichar Datatype

The `unichar` and `univarchar` datatypes support the UTF-16 encoding of Unicode in SAP ASE. These datatypes are independent of the `char` and `varchar` datatypes, but mirror their behavior.

For example, the built-in functions that operate on `char` and `varchar` also operate on `unichar` and `univarchar`. However, `unichar` and `univarchar` store only UTF-16 characters and have no connection to the default character set ID or the default sort order ID, as `char` and `varchar` do.

Each `unichar/univarchar` character requires two bytes of storage. The declaration of a `unichar/univarchar` column is the number of 16-bit Unicode values. The following example creates a table with one `unichar` column for 10 Unicode values, requiring 20 bytes of storage:

```
create table unitbl (unicol unichar(10))
```

The length of a `unichar/univarchar` column is limited by the size of a data page, as is the length of `char/varchar` columns.

Unicode surrogate pairs use the storage of two 16-bit Unicode values (in other words, four bytes). Be aware of this when declaring columns intended to store Unicode surrogate pairs (a pair of 16 bit values that represent a character in the range [0x010000..0x10FFFF]). By default, SAP ASE correctly handles surrogates, and does not split the pair. Truncation of Unicode data is handled in a manner similar to that of `char` and `varchar` data.

You can use `unichar` expressions anywhere `char` expressions are used, including comparison operators, joins, subqueries, and so forth. However, mixed-mode expressions of both `unichar` and `char` are performed as `unichar`. The number of Unicode values that can participate in such operations is limited to the maximum size of a `unichar` string.

The normalization process modifies Unicode data so there is only a single representation in the database for a given sequence of abstract character (see *Introduction to the Basics*, in the *Performance and Tuning Series: Basics* for a discussion of normalization). Often, characters followed by combined diacritics are replaced by pre-combined forms. This allows significant performance optimizations. By default, the server assumes all Unicode data should be normalized.

7.1.6.1.1 Relational Expressions

All relational expressions involving at least one expression of `unichar` or `univarchar`, are based on the default Unicode sort order. If one expression is `unichar` and the other is `varchar` (`nvarchar`, `char`, or `nchar`), the latter is implicitly converted to `unichar`.

The following table shows which expressions are most often used in `where` clauses, and where they may be combined with logical operators.

When comparing Unicode character data, "less than" means closer to the beginning of the default Unicode sort order, and "greater than" means closer to the end. "Equality" means the Unicode default sort order makes no distinction between two values (although they need not be identical). For example, the precomposed character ê must be considered equal to the combining sequence consisting of the letter e followed by U +0302. (A precomposed character is a Unicode character you can decompose into an equivalent string of several other characters.) If the Unicode normalization feature is turned on (the default), Unicode data is automatically normalized and the server never sees unnormalized data.

`expr1 op_compare [any | all] (subquery)`	The use of `any` or `all` with comparison operators and subquery `expr2` , implicitly invokes `min` or `max`. For instance, "`expr1> any expr2`" means, in effect, "`expr1> min(expr2)`".
`expr1 [not] in (expression list)` `expr1 [not] in (subquery)`	The `in` operator checks for equality with each of the elements in `expr2`, which can be a list of constants, or the results of a subquery.
`expr1 [not] between expr2 and expr3`	The `between` operator specifies a range. It is, in effect, shorthand for "`expr1 = expr2` and `expr1<= expr3`".
`expr1 [not] like "match_string" [escape"esc_char"]`	The `like` operator specifies a pattern to be matched. The semantics for pattern matching with Unicode data are the same as for regular character data. If `expr1` is a `unichar` column name, then "`match_string`" may be either a `unichar` string or a `varchar` string. In the latter case, an implicit conversion takes place between `varchar` and `unichar`

7.1.6.1.2 Join Operators

Join operators appear in the same manner as comparison operators. You can use any comparison operator in a join.

Expressions involving at least one expression of type `unichar` are based on the default Unicode sort order. If one expression is of type `unichar` and the other type `varchar` (`nvarchar`, `char`, or `nchar`), the latter is implicitly converted to `unichar`.

7.1.6.1.3 Union Operators

The `union` operator operates with `unichar` data much like it does with `varchar` data. Corresponding columns from individual queries must be implicitly convertible to `unichar`, or explicit conversion must be used.

7.1.6.1.4 Clauses and Modifiers

When `unichar` and `univarchar` columns are used in `group by` and `order by` clauses, equality is judged according to the default Unicode sort order. This is also true when using the `distinct` modifier.

7.1.6.2 text Datatype

The `text` datatype stores up to 2,147,483,647 bytes of printable characters on linked lists of separate data pages. Each page stores a maximum of 1800 bytes of data.

To save storage space, define `text` columns as NULL. When you initialize a `text` column with a non-null `insert` or `update`, SAP ASE assigns a text pointer and allocates an entire 2K data page to hold the value.

If you are using databases connected with Component Integration Services, there are several differences in the way `text` datatypes are handled. See the *Component Integration Services Users Guide*.

Related Information

Change text, unitext, and image data [page 375]

7.1.6.3 unitext Datatype

The variable-length `unitext` datatype can hold up to 1,073,741,823 Unicode characters (2,147,483,646 bytes). You can use `unitext` anywhere you use the `text` datatype, with the same semantics. unitext columns are stored in UTF-16 encoding, regardless of the SAP ASE default character set.

The `unitext` datatype uses the same storage mechanism as `text`. To save storage space, define `unitext` columns as NULL. When you initialize a `unitext` column with a non-null `insert` or `update` clause, SAP ASE assigns a text pointer and allocates an entire 2K data page to hold the value.

The benefits of `unitext` include:

- Large Unicode character data. Together with `unichar` and `univarchar` datatypes, SAP ASE provides complete Unicode datatype support, which is best for incremental multilingual applications.
- unitext stores data in UTF-16, which is the native encoding for Windows and Java environments.

Related Information

Change text, unitext, and image data [page 375]

7.1.7 Binary Datatypes

Binary datatypes store raw binary data, such as pictures, in a hexadecimal-like notation.

Binary data begins with the characters “0x” and includes any combination of digits and the uppercase and lowercase letters A – F. The two digits following “0x” in `binary` and `varbinary` data indicate the type of number: “00” represents a positive number and “01” represents a negative number.

If the input value does not include “0x,” SAP ASE assumes that the value is an ASCII value and converts it.

> **Note**
>
> SAP ASE manipulates the binary types in a platform-specific manner. For true hexadecimal data, use the `hextoint` and `inttohex` functions.

Use the `binary(n)` and `varbinary(n)` datatypes to store data up to 255 bytes in length. Each byte of storage holds 2 binary digits. Specify the column length with `<n>`, or use the default length of 1 byte. If you enter a value longer than `<n>`, SAP ASE truncates the entry to the specified length without warning or error.

- Use the fixed-length binary type, `binary(n)`, for data in which all entries are expected to have a similar length. Because entries in `binary` columns are zero-padded to the column length, they may require more storage space than those in `varbinary` columns, but they are accessed somewhat faster.
- Use the variable-length binary type, `varbinary(n)`, for data that is expected to vary greatly in length. Storage size is the actual size of the data values entered, not the column length. Trailing zeros are truncated.

When you create a `binary` column that allows nulls, SAP ASE converts it to a `varbinary` column and uses the storage rules for that datatype.

You can search binary strings with the `like` keyword and operate on them with the string functions.

> **Note**
>
> Because the exact form in which you enter a particular value depends upon the hardware you are using, calculations involving binary data may produce different results on different platforms.

Related Information

Transact-SQL Functions [page 471]

7.1.7.1 image Datatype

Use the `image` datatype to store larger blocks of binary data on external data pages. An `image` column can store up to 2,147,483,647 bytes of data on linked lists of data pages separate from other data storage for the table.

When you initialize an `image` column with a non-null `insert` or `update`, SAP ASE assigns a text pointer and allocates an entire 2K data page to hold the value. Each page stores a maximum of 1800 bytes.

To save storage space, define `image` columns as NULL. To add `image` data without saving large blocks of binary data in your transaction log, use `writetext`. See the *Reference Manual: Commands*.

You cannot use the `image` datatype:

- For parameters to stored procedures, as values passed to these parameters, or for local variables
- For parameters to remote procedure calls (RPCs)
- In `order by`, `compute`, `group by`, or `union` clauses
- In an index
- In subqueries or joins
- In a `where` clause, except with the keyword `like`
- With the + concatenation operator
- In the `if update` clause of a trigger

If you are using databases connected with Component Integration Services, there are several differences in the way `image` datatypes are handled. See the *Component Integration Services Users Guide*.

Related Information

Change text, unitext, and image data [page 375]

7.1.8 bit Datatype

Use `bit` columns for true/false or yes/no types of data. `bit` columns hold either 0 or 1.

Integer values other than 0 or 1 are interpreted as 1. Storage size is 1 byte. Multiple `bit` datatypes in a table are collected into bytes. For example, 7-`bit` columns fit into 1 byte; 9-`bit` columns take 2 bytes.

Columns of datatype `bit` cannot be NULL and cannot have indexes on them. The `status` column in the `syscolumns` system table indicates the unique offset position for `bit` columns.

7.1.9 timestamp Datatype

The `timestamp` user-defined datatype is used for columns in tables that are browsed in Open Client™ DB-Library applications.

Every time a row containing a `timestamp` column is inserted or updated, the `timestamp` column is automatically updated. A table can have only one column of the `timestamp` datatype. A column named `timestamp` automatically has the system datatype `timestamp`. Its definition is:

```
varbinary(8) "NULL"
```

Because `timestamp` is a user-defined datatype, you cannot use it to define other user-defined datatypes. You must enter it as "timestamp" in all lowercase letters.

7.1.10 sysname and longsysname Datatype

`sysname` and `longsysname` are user-defined datatypes used in the system tables.

`sysname` is defined as:

```
varchar(30) "NOT NULL"
```

`longsysname` is defined as:

```
varchar(255) "NOT NULL"
```

You can declare a column, parameter, or variable to be of type `sysname` or `longsysname`. Alternately, you can also create a user-defined datatype with a base type of `sysname` or `longsysname`.

You can then use this user-defined datatype to create columns.

Related Information

User-Defined Datatypes [page 210]

7.2 LOB Locators in Transact-SQL Statements

Large object (LOB) locators let you reference an LOB in Transact-SQL statements rather than referencing the LOB itself.

Because the size of a `text`, `unitext`, or `image` LOB can be many megabytes, using an LOB locator in Transact-SQL statements reduces network traffic between the client and SAP ASE, and reduces the amount of memory otherwise needed by the client to process the LOB.

SAP ASE lets client applications send and receive locators as host variables and parameter markers.

When you create an LOB locator, SAP ASE caches the LOB value in memory and generates an LOB locator to reference it.

A LOB locator remains valid for the duration of the transaction in which it was created. SAP ASE invalidates the locator when the transaction commits or is rolled back.

LOB locators use three different datatypes:

- `text_locator` – for `text` LOBs.
- `unitext_locator` – for `unitext` LOBs.
- `image_locator` – for `image` LOBs.

You can declare local variables for the locator datatypes. For example:

```
declare @v1 text_locator
```

Because LOBs and locators are stored only in memory, you cannot use locator datatypes as column datatypes for user tables or views, or in constraints or defaults.

You can create a LOB locator explicitly or implicitly.

In general, when used in a Transact-SQL statement, locators are implicitly converted to the LOB they reference. That is, when a locator is passed to a Transact-SQL function, the function operates on the LOB that is referenced by the locator.

Any changes you make to the LOB referenced by the locator are not reflected in the source LOB in the database—unless you explicitly save them. Similarly, any changes you make to the LOB stored in the database are not reflected in the LOB referenced by the locator.

> **Note**
>
> Locators are best used in Transact-SQL statements that return only a few rows, or in cursor statements. This allows locators and associated LOBs to be processed and released in a manner that conserves memory. You may need to increase available memory if several LOBs are created in a single transaction.

7.2.1 Implicitly Create a Locator

If you assign one locator to another, a new locator value is assigned to the new variable. Each locator has a unique locator value.

In this example, the third statement assigns a new LOCATOR which is created by copying the LOB value referenced by `@v` to `@w`. For example:

```
declare @v text_locator, @w text_locator
select @v = create_locator(text_locator, textvol)
from my_table where id = 5
select @w = @v
```

You can use the `set send_locator on` command to implicitly create a locator by specifying that all LOB values in a result set are to be converted to the relevant locator type and then sent as such to the client. For example:

```
set send_locator on
select textcol from my_table where id = 5
```

Because `send_locator` is on, SAP ASE creates a locator for each row value of `textcol`, and sends the resulting locators to the client. If `send_locator` is off (the default), SAP ASE sends the actual text value.

7.2.2 Explicitly Create a Locator

Use the `create_locator` function to explicitly create a locator.

- To create a locator for a `text` LOB, enter:

  ```
  select create_locator(text_locator, convert(text, "<some_text_value>"))
  ```

- To create a locator for an `image` LOB, enter:

  ```
  select create_locator(image_locator, <image_col>) from <table_name>
  ```

  For example, to create a locator for an `image` LOB stored in the `image_column` column of `my_table`, enter:

  ```
  select create_locator(image_locator, image_column) from my_table where id=7
  ```

Both examples create and return a LOB locator that references a LOB value stored in SAP ASE memory.

> **Note**
>
> When explicitly creating a locator, SAP ASE always sends a locator, regardless of the value of `send_locator`.

Using a `select` statement to create a locator is most useful when the client application stores the received locator for use in subsequent Transact-SQL statements. In an `isql` session, the locator is assigned to a local variable. For example:

```
declare @v text_locator
select @v = create_locator(text_locator, textcol) from
my_table where id = 10
```

> **Note**
>
> You can also create a locator that references an empty LOB.

7.2.3 Convert the Locator Value to the LOB Value

After using a locator in a Transact-SQL statement, you can convert (dereference) the locator to the corresponding LOB.

To explicitly convert a locator, use the `return_lob` function. For example, to return the LOB value for `<@w>`, enter:

```
declare @w text_locator
select return_lob(text, @w)
```

You can also implicitly convert the locator. For example, to insert the actual LOB value for `<@w>` into the `textcol` column of `my_table`, enter:

```
insert my_table(textcol) values (@w)
```

The `return_lob` command overrides the `set sent_locator on` command. `return_lob` always returns the LOB.

Parameter Markers

You can explicitly convert a locator when using parameter markers in Transact-SQL statements. For example:

```
insert my_table (textcol)) values(return_lob(text,?))
```

Use the `locator_literal` function to identify the locator:

```
insert my_table (imagecol) values (locator_literal(image_locator,
<binary_locator_value>))
```

7.2.4 Locator Scope

In general, a locator is valid for the duration of a transaction. Use the `deallocate locator` function to override the default scoping, and deallocate the locator within the transaction.

`deallocate locator` can be especially useful in saving memory when you need to create many locators within a transaction. For example:

```
begin tran
declare @v text_locator
select @v = textcol from my_table where id=5
deallocate locator @v
...
commit
```

`deallocate locator` removes the LOB value for `<@v>` from SAP ASE memory before the transaction commits, and marks the locator as invalid.

7.3 Convert Between Datatypes

SAP ASE automatically handles many conversions from one datatype to another. These are called implicit conversions. Use the `convert`, `inttohex`, and `hextoint` functions to explicitly request other conversions.

Some conversions cannot be performed, either explicitly or automatically, because of incompatibilities between the datatypes.

For example, SAP ASE automatically converts `char` expressions to `datetime` for the purposes of a comparison, both expressions can be interpreted as `datetime` values. However, for display purposes, you must use the `convert` function to convert `char` to `int`. Similarly, you must use `convert` on integer data for SAP ASE to treat it as character data so that you can use the `like` keyword with it.

The syntax for the `convert` function is:

```
convert (<datatype>, <expression>, [<style>])
```

In the following example, `convert` displays the `total_sales` column using the `char` datatype, showing all sales beginning with the digit 2:

```
select title, total_sales
from titles
where convert (char(20), total_sales) like "2%"
```

Use the optional `style` parameter to convert `datetime` values to `char` or `varchar` datatypes to get a wide variety of date display formats.

Related Information

Transact-SQL Functions [page 471]

7.4 Mixed-Mode Arithmetic and Datatype Hierarchy

When you perform arithmetic on values with different datatypes, SAP ASE must determine the datatype and, in some cases, the length and precision, of the result.

Each system datatype has a datatype hierarchy, which is stored in the `systypes` system table. User-defined datatypes inherit the hierarchy of the system type on which they are based.

The SAP ASE datatype hierarchy applies only to computations or expressions involving numeric datatypes.

When comparing different datatypes related to date or time (for example, `datetime` versus `date`), only components that are present in both datatypes are compared. For example, SAP ASE considers the `datetime` value “20-Nov-2012 23:24:25” equal to the `date` value “20-Nov-2012” since it compares only the date component (in this case, the string “20-Nov-2012”). This is complaint with the ANSI SQL standard.

The following query ranks the datatypes in a database by hierarchy. In addition to the information shown below, query results include information about any user-defined datatypes in the database:

```
select name, hierarchy
from systypes
order by hierarchy
```

```
name           hierarchy
---------- ---------
floatn             1
float              2
datetimn           3
datetime           4
real               5
numericn           6
numeric            7
decimaln           8
decimal            9
moneyn            10
money             11
smallmoney        12
smalldatet        13
intn              14
uintn             15
bigint            16
ubigint           17
int               18
uint              19
smallint          20
usmallint         21
tinyint           22
bit               23
univarchar        24
unichar           25
unitext           26
sysname           27
varchar           27
nvarchar          27
longsysnam        27
char              28
nchar             28
timestamp         29
varbinary         29
binary            30
text              31
image             32
date              33
time              34
daten             35
timen             36
bigdatetim        37
bigtime           38
bigdatetim        39
bigtimen          40
extended t        99
```

Note

`u<integer_type>` (for example, `usmallint`) is an internal representation. The correct syntax for unsigned types is `unsigned {int | integer | bigint | smallint }`.

The datatype hierarchy determines the results of computations using values of different datatypes. The result value is assigned the datatype that is closest to the top of the list.

In the following example, `qty` from the `sales` table is multiplied by `royalty` from the `roysched` table. `qty` is a `smallint`, which has a hierarchy of 20; `royalty` is an `int`, which has a hierarchy of 18. Therefore, the datatype of the result is an `int`.

```
smallint(qty) * int(royalty) = int
```

This example multiplies an `int`, which has a hierarchy of 18; with an `unsigned int`, which has a hierarchy of 19, and the datatype of the result is a `int`:

```
int(10) * unsigned int(5) = int(50)
```

> **Note**
>
> Unsigned integers are always promoted to a signed datatype when you use a mixed-mode expression. If the unsigned integer value is not in the signed integer range, SAP ASE issues a conversion error.

See the *Reference Manual: Building Blocks* for more information about the datatype hierarchy.

7.4.1 Working with money Datatypes

Combine money datatype with literals or variables, or with a `float` or `numeric` datatype.

If you are combining `money` and literals or variables, and you need results of `money` type, use `money` literals or variables.

```
create table mytable
(moneycol money,)
insert into mytable values ($10.00)
select moneycol * $2.5 from mytable
```

If you are combining money with a `float` or `numeric` datatype from column values, use the `convert` function:

```
select convert (money, moneycol * percentcol)
    from debits, interest
drop table mytable
```

7.4.2 Determine Precision and Scale

For the `numeric` and `decimal` types, each combination of precision and scale is a distinct SAP ASE datatype.

If you perform arithmetic on two `numeric` or `decimal` values, `n1` with precision `p1` and scale `s1`, and `n2` with precision `p2` and scale `s2`, SAP ASE determines the precision and scale of the results as in the following table.

Operation	Precision	Scale
n1 + n2	max(s1, s2) + max(p1 - s1, p2 - s2) + 1	max(s1, s2)
n1 - n2	max(s1, s2) + max(p1 - s1, p2 - s2) + 1	max(s1, s2)
n1 * n2	s1 + s2 + (p1 - s1) + (p2 - s2) + 1	s1 + s2
n1 / n2	max(s1 + p2 + 1, 6) + p1 - s1 + s2	max(s1 + p2 + 1, 6)

7.5 User-Defined Datatypes

A Transact-SQL enhancement to SQL allows you to design your own datatypes to supplement the system datatypes. A user-defined datatype is defined in terms of system datatypes.

> **Note**
>
> To use a user-defined datatype in more than one database, create the datatype in the `model` database. You can then use the user-defined datatype definition in any new databases you create.

Once you define a datatype, you can use it as the datatype for any column in the database. For example, `tid` is used as the datatype for columns in several `pubs2` tables: `titles.title_id`, `titleauthor.title_id`, `sales.title_id`, and `roysched.title_id`.

The advantage of user-defined datatypes is that you can bind rules and defaults to them for use in several tables.

Use `sp_addtype` to create user datatypes. It takes as parameters the name of the datatype being created, the SAP ASE-supplied datatype from which it is being built, and an optional null, not null, or identity specification.

You can build a user-defined datatype using any system datatype other than `timestamp`. User-defined datatypes have the same datatype hierarchy as the system datatypes on which they are based. Unlike SAP ASE-supplied datatypes, user-defined datatype names are case-sensitive.

For example, to define datatype `tid`:

```
sp_addtype tid, "char(6)", "not null"
```

You must enclose a parameter within single or double quotes if it includes a blank or some form of punctuation, or if it is a keyword other than `null` (for example, `identity` or `sp_helpgroup`). In this example, quotes are required around `char(6)` because of the parentheses, but around “not null” because of the blank. They are not required around `tid`.

Related Information

Defining Defaults and Rules for Data [page 413]

7.5.1 Length, Precision, and Scale

For some user-defined datatypes, you must specify additional parameters for lengh, precision, and scale.

- The `char`, `nchar`, `varchar`, `nvarchar`, `binary`, and `varbinary` datatypes expect a length in parentheses. If you do not supply one, SAP ASE assumes the default length of 1 character.
- The `float` datatype expects a precision in parentheses. If you do not supply one, SAP ASE uses the default precision for your platform.
- The `numeric` and `decimal` datatypes expect a precision and scale, in parentheses and separated by a comma. If you do not supply them, SAP ASE uses a default precision of 18 and scale of 0.

You cannot change a user-defined datatype's length, precision, or scale when you include it in a `create table` statement.

7.5.2 Null Type

The null type determines how the user-defined datatype treats nulls. You can create a user-defined datatype with a null type of "null", "NULL", "nonull", "NONULL", "not null", or "NOT NULL". `bit` and identity types do not allow null values.

If you omit the null type, SAP ASE uses the null mode defined for the database (by default, NOT NULL). For compatibility with SQL standards, use `sp_dboption` to set the `allow nulls by default` option to true.

You can override the null type when you include the user-defined datatype in a `create table` statement.

7.5.3 Associate Rules and Defaults with User-Defined Datatypes

Once you have created a user-defined datatype, use `sp_bindrule` and `sp_bindefault` to associate rules and defaults with the datatype. Use `sp_help` to print a report that lists the rules, defaults, and other information associated with the datatype.

Related Information

Defining Defaults and Rules for Data [page 413]

7.5.4 Create User-Defined Datatype with IDENTITY Property

Use `sp_addtype` to create a user-defined datatype with the IDENTITY property.

The new type must be based on a physical type of `numeric` with a scale of 0 or any integer type:

```
sp_addtype <typename>, "numeric (<precision>, 0)", "identity"
```

The following example creates a user-defined type, `IdentType`, with the IDENTITY property:

```
sp_addtype IdentType, "numeric(4,0)", "identity"
```

When you create a column from an IDENTITY type, you can specify either `identity` or `not null`—or neither one—in the `create` or `alter table` statement. The column automatically inherits the IDENTITY property.

Here are three different ways to create an IDENTITY column from the `IdentType` user-defined type:

```
create table new_table (id_col IdentType)
drop table new_table
create table new_table (id_col IdentType identity)
drop table new_table
create table new_table (id_col IdentType not null)
drop table new_table
```

> **Note**
>
> If you try to create a column that allows nulls from an IDENTITY type, the `create table` or `alter table` statement fails.

7.5.5 Create IDENTITY Columns from User-Defined Datatypes

You can create IDENTITY columns from user-defined datatypes that do not have the IDENTITY property.

The user-defined types must have a physical datatype of `numeric` or with a scale of 0, or any integer type, and must be defined as `not null`.

7.5.6 Drop a User-Defined Datatype

Use `sp_droptype` to drop a user-defined datatype.

See the *Reference Manual: Procedures*.

> **Note**
>
> You cannot drop a datatype that is in use in any table.

7.6 Datatype Entry Rules

Several of the SAP ASE-supplied datatypes have special rules for entering and searching for data.

Related Information

Databases and Tables [page 50]

7.6.1 char, nchar, unichar, univarchar, varchar, nvarchar, unitext, and text

Certain rules apply when using character data types.

All `character` and `text` data must be enclosed in single or double quotes when you enter it as a literal.

Use single quotes if the `quoted_identifier` option of the `set` command is set `on`. If you use double quotes, SAP ASE treats the text as an identifier.

Character literals may be any length, whatever the logical page size of the database. If the literal is wider than 16KB (16384 bytes), SAP ASE treats it as `text` data, which has restrictive rules regarding implicit and explicit conversion to other datatypes. See, *System and User-Defined Datatypes*, in the *Reference Manual: Building Blocks* for a discussion of the different behavior of `character` and `text` datatypes.

When you insert character data into a `char`, `nchar`, `unichar`, `univarchar`, `varchar`, or `nvarchar` column for which the specified length is less than the length of the data, the entry is truncated. Set the `string_rtruncation` option `on` to receive a warning message when this occurs.

> **Note**
>
> This truncation rule applies to all character data, whether it resides in a column, a variable, or a literal string.

There are two ways to specify literal quotes within a character entry:

- Use two quotes. For example, if you begin a character entry with a single quote and you want to include a single quote as part of the entry, use two single quotes: 'I don't 'understand.' ' For double quotes: “He said, “ “It's not really confusing.” ”
- Enclose the quoted material in the opposite kind of quotation mark. In other words, surround an entry containing a double quote with single quotes, or vice versa. For example: “George said, 'There must be a better way.'”

To enter a character string that is longer than the width of your screen, enter a backslash (\) before going to the next line.

Use the `like` keyword and wildcard characters to search for `character`, `text`, and `datetime` data.

See, *System and User-Defined Datatypes*, in the *Reference Manual: Building Blocks* for details on inserting `text` data and information about trailing blanks in `character` data.

7.6.2 Date and Time

The SAP ASE datatypes used to store date and time information include `datetime`, `smalldatetime`, `date`, `time`, `bigdatetime`, and `bigtime`.

All `date` and `time` data must be enclosed in single or double quotes when you enter it as a literal.

Display and entry formats for date and time data provide a wide range of date output formats, and recognize a variety of input formats. The display and entry formats are controlled separately. The default display format provides output that looks like "Apr 15 2003 10:23PM." The `convert` command provides options to display seconds and milliseconds and to display the date with other date-part ordering.

SAP ASE recognizes a wide variety of data entry formats for dates. Case is always ignored, and spaces can occur anywhere between date parts. When you enter `datetime` and `smalldatetime` values, always enclose them in single or double quotes. Use single quotes if the `quoted_identifier` option is on; if you use double quotes, SAP ASE treats the entry as an identifier.

SAP ASE recognizes the two date and time portions of the data separately, so the time can precede or follow the date. Either portion can be omitted, in which SAP ASE uses the default. The default date and time is January 1, 1900, 12:00:00:000AM.

For `datetime`, the earliest date you can use is January 1, 1753; the latest is December 31, 9999. For `smalldatetime`, the earliest date you can use is January 1, 1900; the latest is June 6, 2079. For `bigdatetime`, the earliest date you can enter is January 1, 0001 and the latest is December 31, 9999. For `date`, the earliest date you can use is January 1, 0001; the latest is December 31, 9999. Dates earlier or later than these dates must be entered, stored, and manipulated as `char`, or `unichar`; or `varchar` or `univarchar` values. SAP ASE rejects all values it cannot recognize as dates between those ranges.

For `time`, the earliest time is 12:00AM; the latest is 11:59:59:999. For `bigtime`, the earliest time is 12:00:00.000000AM; the latest is 11:59:59.999999PM.

Related Information

Transact-SQL Functions [page 471]

7.6.2.1 Enter Times

The order of time components is significant. Enter the hours first; then minutes; then seconds; then milliseconds; then AM (or am) or PM (pm).

12AM is midnight; 12PM is noon. To be recognized as time, a value must contain either a colon or an AM or PM signifier. `smalldatetime` is accurate only to the minute. `time` is accurate to the millisecond.

Milliseconds can be preceded by either a colon or a period. If preceded by a colon, the number means thousandths of a second. If preceded by a period, a single digit means tenths of a second, two digits mean hundredths of a second, and three digits mean thousandths of a second.

For example, “12:30:20:1” means 20 and one-thousandth of a second past 12:30; “12:30:20.1” means 20 and one-tenth of a second past 12:30.

Among the acceptable formats for time data are:

```
14:30
14:30[:20:999]
14:30[:20.9]
4am
4 PM
[0]4[:30:20:500]AM
```

Display and entry formats for `bigdatetime` and `bigtime` include microseconds. The time for these datatypes must be specified as:

- `hours[:minutes[:seconds[.microseconds]] [<AM> | <PM>]`
- `hours[:minutes[:seconds[number of milliseconds]] [<AM> | <PM>]`

Use 12 AM for midnight and 12 PM for noon. A `bigtime` value must contain either a colon or an AM or PM signifier. AM or PM can be entered in uppercase, lowercase, or mixed case.

The seconds specification can include either a decimal portion preceded by a point, or a number of milliseconds preceded by a colon. For example, “12:30:20:1” means twenty seconds and one millisecond past 12:30; “12:30:20.1” means twenty and one-tenth of a second past.

To store a `bigdatetime` or `bigtime` time value that includes microseconds, specify a string literal using a point. “00:00:00.1” means one tenth of a second past midnight and “00:00:00.000001” means one millionth of a second past midnight. Any value after the colon specifying fractional seconds will continue to refer to a number of milliseconds. Such as “00:00:00:5” means 5 milliseconds.

7.6.2.2 Enter Dates

The `set dateformat` command specifies the order of the date parts (month, day, and year) when dates are entered as strings of numbers with separators.

`set language` can also affect the format for dates, depending on the default date format for the language you specify. The default language is us_english, and the default date format is `<mdy>`. See the *Reference Manual: Commands*.

> **Note**
>
> `dateformat` affects only the dates entered as numbers with separators, such as “4/15/90” or “20.05.88”. It does not affect dates where the month is provided in alphabetic format, such as “April 15, 1990” or where there are no separators, such as “19890415”.

7.6.2.2.1 Date Formats

SAP ASE recognizes three basic date formats. Each format must be enclosed in quotes and can be preceded or followed by a time specification.

- The month is entered in alphabetic format.
  - Valid formats for specifying the date alphabetically are:

```
Apr[il] [15][,] 1997
Apr[il] 15[,] [19]97
Apr[il] 1997 [15]
```

```
[15] Apr[il][,] 1997
15 Apr[il][,] [19]97
15 [19]97 apr[il]
[15] 1997 apr[il]
```

```
1997 APR[IL] [15]
1997 [15] APR[IL]
```

  - Month can be a three-character abbreviation, or the full month name, as given in the specification for the current language.
  - Commas are optional.
  - Case is ignored.
  - If you specify only the last two digits of the year, values of less than 50 are interpreted as "20yy," and values of 50 or more are interpreted as "19yy."
  - Type the century only when the day is omitted or when you need a century other than the default.
  - If the day is missing, SAP ASE defaults to the first day of the month.
  - When you specify the month in alphabetic format, the `dateformat` setting is ignored (see the *Reference Manual: Commands*).
- The month is entered in numeric format, in a string with a slash (/), hyphen (-), or period (.) separator.
  - The month, day, and year must be specified.
  - The strings must be in the form:

```
<num> <sep> <num> <sep> <num> [ <time spec> ]
```

or:

```
[ <time spec> ]  <num> <sep> <num> <sep> <num>
```

  - The interpretation of the values of the date parts depends on the `dateformat` setting. If the ordering does not match the setting, either the values are not interpreted as dates, because the values are out of range, or the values are misinterpreted. For example, "12/10/08" can be interpreted as one of six different dates, depending on the `dateformat` setting. See the *Reference Manual: Commands*.
  - To enter "April 15, 1997" in `<mdy>` `dateformat`, you can use any of these formats:

```
[0]4/15/[19]97
[0]4-15-[19]97
[0]4.15.[19]97
```

  - The other entry orders are shown below with "/" as a separator; you can also use hyphens or periods:

```
15/[0]4/[19]97 (dmy)
1997/[0]4/15 (ymd)
```

```
1997/15/[0]4 (ydm)
[0]4/[19]97/15 (myd)
15/[19]97/[0]4 (dym)
```

- The date is given as an unseparated four-, six-, or eight-digit string, or as an empty string, or only the time value, but no date value, is given.
  - The `dateformat` is always ignored with this entry format.
  - If four digits are given, the string is interpreted as the year, and the month is set to January, the day to the first of the month. The century cannot be omitted.
  - Six- or eight-digit strings are always interpreted as `<ymd>`; the month and day must always be two digits. This format is recognized: [19]960415.
  - An empty string (“”) or missing date is interpreted as the base date, January 1, 1900. For example, a time value like “4:33” without a date is interpreted as “January 1, 1900, 4:33AM''.

The `set datefirst` command specifies the day of the week (Sunday, Monday, and so on) when `weekday` or `dw` is used with `datename`, and a corresponding number when used with `datepart`. Changing the language with `set language` can also affect the format for dates, depending on the default first day of the week value for the language. For the default language of `us_english`, the default `datefirst` setting is Sunday=1, Monday=2, and so on; others produce Monday=1, Tuesday=2, and so on. Use `set datefirst` to change default behavior on a per-session basis. See the *Reference Manual: Commands*.

Related Information

Enter Times [page 214]

7.6.2.3 Search Dates and Times

You can use the `like` keyword and wildcard characters with `datetime`, `smalldatetime`, `bigdatetime`, `bigtime`, `date`, and time data, as well as with `char`, `unichar`, `nchar`, `varchar`, `univarchar`, `nvarchar`, `text`, and `unitext`.

When you use `like` with `date` and `time` values, SAP ASE first converts the dates to the standard `date/time` format, then converts them to `varchar` or `univarchar`. Since the standard display formats for `datetime` and `smalldatetime` do not include seconds or milliseconds, you cannot search for seconds or milliseconds with `like` and a match pattern. Use the type conversion function, `convert`, to search for seconds and milliseconds.

Use `like` when you search for `datetime, bigtime, bigdatetime` or `smalldatetime` values, because these types of data entries may contain a variety of date parts. For example, if you insert the value “9:20” into a column named `arrival_time`, the following clause would not find it, because SAP ASE converts the entry to “Jan 1, 1900 9:20AM”:

```
where arrival_time = "9:20"
```

However, this clause would find it:

```
where arrival_time like "%9:20%"
```

This applies to `date` and `time` datatypes as well.

If you are using `like`, and the day of the month is less than 10, you must insert two spaces between the month and day to match the `varchar` conversion of the `datetime` value. Similarly, if the hour is less than 10, the conversion places two spaces between the year and the hour. The clause `like May 2%`, with one space between "May" and "2", finds all dates from May 20 through May 29, but not May 2. You need not insert the extra space with other date comparisons, only with `like`, since the `datetime` values are converted to `varchar` only for the `like` comparison.

7.6.3 binary, varbinary, and image

When you enter `binary`, `varbinary`, or `image` data as literals, you must precede the data with "0x". For example, to enter "FF", type "0xFF". Do not, however, enclose data beginning with "0x" with quotation marks.

Binary literals can be of any length, regardless of the logical page size of the database. If the length of the literal is less than 16KB (16384 bytes), SAP ASE treats the literal as `varbinary` data. If the length of the literal is greater than 16KB, SAP ASE treats it as `image` data.

When you insert `binary` data into a column for which the specified length is less than the length of the data, the entry is truncated without warning.

A length of 10 for a `binary` or `varbinary` column means 10 bytes, each storing 2 hexadecimal digits.

When you create a default on a `binary` or `varbinary` column, precede it with "0x".

See, *System and User-Defined Datatypes* in the *Reference Manual: Building Blocks* for the different behaviors of `binary` datatypes and `image` datatypes and for information about trailing zeros in hexadecimal values.

7.6.4 money and smallmoney

Monetary values entered with the E notation are interpreted as `float`. This may cause an entry to be rejected or to lose some of its precision when it is stored as a `money` or `smallmoney` value.

`money` and `smallmoney` values can be entered with or without a preceding currency symbol such as the dollar sign ($), yen sign (¥), or pound sterling sign (£). To enter a negative value, place the minus sign after the currency symbol. Do not include commas in your entry.

You cannot enter `money` or `smallmoney` values with commas, although the default print format for `money` or `smallmoney` data places a comma after every three digits. When `money` or `smallmoney` values appear, they are rounded up to the nearest cent. All the arithmetic operations except modulo are available with `money`.

7.6.5 float, real, and double precision

Enter approximate numeric types—`float`, `real`, and `double precision`—as a mantissa followed by an optional exponent.

The mantissa can include a positive or negative sign and a decimal point. The exponent, which begins after the character “e” or “E”, can include a sign but not a decimal point.

To evaluate approximate numeric data, SAP ASE multiplies the mantissa by 10 raised to the given exponent. This table shows examples of `float`, `real`, and `double precision` data:

Data Entered	Mantissa	Exponent	Value
10E2	10	2	$10 * 10^2$
15.3e1	15.3	1	$15.3 * 10^1$
-2.e5	-2	5	$-2 * 10^5$
2.2e-1	2.2	-1	$2.2 * 10^{-1}$
+56E+2	56	2	$56 * 10^2$

The column’s binary precision determines the maximum number of binary digits allowed in the mantissa. For `float` columns, you can specify a precision of up to 48 digits; for real and double precision columns, the precision is machine-dependent. If a value exceeds the column’s binary precision, SAP ASE flags the entry as an error.

7.6.6 decimal and numeric

The exact numeric types—`dec`, `decimal`, and `numeric`—begin with an optional positive or negative sign and can include a decimal point.The value of exact numeric data depends on the column’s decimal `<precision>` and `<scale>`.

The syntax is:

```
<datatype> [(<precision>  [, <scale> ])]
```

SAP ASE treats each combination of precision and scale as a distinct datatype. For example, `numeric (10,0)` and `numeric (5,0)` are two separate datatypes. The precision and scale determine the range of values that can be stored in a `decimal` or `numeric` column:

- The precision specifies the maximum number of decimal digits that can be stored in the column. It includes all digits to the right and left of the decimal point. You can specify a precision ranging from 1 to 38 digits, or use the default precision of 18 digits.
- The scale specifies the maximum number of digits that can be stored to the right of the decimal point. The scale must be less than or equal to the precision. You can specify a scale ranging from 0 to 38 digits, or use the default scale of 0 digits.

If a value exceeds the column’s precision or scale, SAP ASE flags the entry as an error. Here are some examples of valid `dec` and `numeric` data:

Table 4: Valid precision and scale for numeric data

Data entered	Datatype	Precision	Scale	Value
12.345	`numeric(5,3)`	5	3	12.345
-1234.567	`dec(8,4)`	8	4	-1234.567

The following entries result in errors because they exceed the column's precision or scale:

Table 5: Invalid precision and scale for numeric data

Data entered	Datatype	Precision	Scale
1234.567	`numeric(3,3)`	3	3
1234.567	`decimal(6)`	6	1

7.6.7 Integer Types and Their Unsigned Counterparts

You can insert numeric values into `bigint`, `int`, `smallint`, `tinyint`, `unsigned bigint`, `unsigned int`, and `unsigned smallint` columns with the E notation.

7.6.8 timestamp

You cannot insert data into a `timestamp` column. You must either insert an explicit null by typing **`NULL`** in the column, or use an implicit null by providing a column list that skips the `timestamp` column.

SAP ASE updates the `timestamp` value after each insert or update.

Related Information

Insert Data into Specific Columns [page 358]

7.7 Get Information About Datatypes

Use `sp_help` to display information about the properties of a system datatype or a user-defined datatype.

The output from `sp_help` includes the base type from which the datatype was created, whether it allows nulls, the names of any rules and defaults bound to the datatype, and whether it has the IDENTITY property.

The following examples show information about the `money` system datatype and the `tid` user-defined datatype:

```
sp_help money
```

```
Type_name  Storage_type Length Prec  Scale
---------- ------------ ------ ----- -----
money      money           8 NULL  NULL
Nulls   Default_name  Rule_name   Identity
-----   ------------  ---------   --------
    1   NULL          NULL            NULL
(return status = 0)
sp_help tid
Type_name  Storage_type Length Prec  Scale
---------- ------------ ------ ----- -----
tid        varchar           6 NULL  NULL
Nulls   Default_name  Rule_name   Identity
-----   ------------  ---------   --------
    0   NULL          NULL               0
(return status = 0)
```

8 Queries: Selecting Data from a Table

An SQL query requests data from the database and receives the results. This process, also known as data retrieval, is expressed using the `select` statement.

A query has three main parts: the `select` clause, the `from` clause, and the `where` clause.

The query process, also known as data retrieval, is used for selections, which retrieve a subset of the `rows` in one or more tables, or for projections, which retrieve a subset of the `columns` in one or more tables.

A simple example of a `select` statement is:

```
select <select_list>
from <table_list>
where <search_conditions>
```

The `select` clause specifies the columns you want to retrieve. The `from` clause specifies the tables to search. The `where` clause specifies which rows in the tables you want to see. For example, the following `select` statement finds, in the `pubs2` database, the first and the last names of writers living in Oakland from the `authors` table:

```
select au_fname, au_lname
from authors
where city = "Oakland"
```

Results of this query appear in columnar format:

```
au_fname        au_lname
--------------  -----------
Marjorie        Green
Dick            Straight
Dirk            Stringer
Stearns         MacFeather
Livia           Karsen

(5 rows affected)
```

8.1 select Syntax

A simple `select` statement contains only the `select` clause; the `from` clause is almost always included, but is necessary only in `select` statements that retrieve data from tables.

All other clauses, including the `where` clause, are optional.

The full syntax of the `select` statement is in the *Reference Manual: Commands*.

`TOP <unsigned_integer>` lets you limit the number of rows in a result set; specify the number of rows you want to view. `TOP` is also used in the `delete` and `update` commands, for the same purpose.

If the statement includes a `group by` clause and an `order by` clause, the `group by` clause must precede the `order by` clause.

Qualify the names of database objects if there is ambiguity about the object referred to. If several columns in multiple tables are called "name," you may need to qualify "name" with the database name, owner name, or table name. For example:

```
select au_lname from pubs2.dbo.authors
```

The `holdlock`, `noholdlock`, and `shared` keywords (which are for locking in SAP ASE) and the `index` clause are described in, *Using Locking Commands* in the *Performance and Tuning Series: Locking and Concurrency Control* guide. For information about the `for read only` and `for update` clauses, see the `declare cursor` command in *Reference Manual: Commands*.

> **Note**
>
> The `for browse` clause is used only in DB-Library applications. See the *Open Client DB-Library/C Reference Manual* for details.

Related Information

Aggregates, Grouping, and Sorting [page 285]
Databases and Tables [page 50]
Transactions: Maintain Data Consistency and Recovery [page 632]
Browse Mode Versus Cursors [page 580]

8.1.1 Check for Identifiers in a select Statement

The `syscomments` system table contains entries for each view, rule, default, trigger, table constraint, and procedure.When the source text of a stored procedure or trigger is stored in `syscomments`, a query using `select *` is also stored in `syscomments` expanding the column list referenced in the `select *`.

For example, a `select *` from a table containing the columns `col1` and `col2` is stored as:

```
select <table>.col1, <table>.col2 from <table>
```

The column list verifies that identifiers (table names, column names and so on) comply with the rules for identifiers.

For example, if a table includes the columns `col1` and `2col`, the second column name starts with a number, which can be included only by using brackets in the `create table` statement.

When performing a `select *` in a stored procedure or trigger from this table, the text in `syscomments` looks similar to:

```
select <table>.col1, <table>[2col] from <table>
```

For all identifiers used in the text that extends a `select *`, brackets are added when the identifier does not comply with the rules for identifiers.

8.2 Choose Columns Using the select Clause

The items in the `select` clause make up the select list. When the select list consists of a column name, a group of columns, or the wildcard character (*), the data is retrieved in the order in which it is stored in the table (`create table` order).

8.2.1 Choose all Columns Using select *

The asterisk (*) selects all the column names in all the tables specified by the `from` clause.

Use the asterisk to save typing time and errors when you want to see all the columns in a table. When you use the asterisk in a query, data is retrieved in `create table` order.

The syntax for selecting all the columns in a table is:

```
select *
from <table_list>
```

The following statement retrieves all columns in the `publishers` table and displays them in `create table` order. This statement retrieves all rows since it contains no `where` clause:

```
select *
from publishers
```

The results look like this:

```
pub_id  pub_name              city          state
-----   --------------        ---------     -----
0736    New Age Books         Boston        WA
0877    Binnet & Hardley      Washington    DC
1389    Algodata Infosystems  Berkeley      CA

(3 rows affected)
```

If you listed all the column names in the table in order after the `select` keyword, you would get exactly the same results:

```
select pub_id, pub_name, city, state
from publishers
```

You can also use "*" more than once in a query:

```
select *, *
from publishers
```

This query displays each column name and each piece of column data twice. Like a column name, you can qualify an asterisk with a table name. For example:

```
select publishers.*
from publishers
```

However, because `select *` finds all the columns currently in a table, changes in the structure of a table such as adding, removing, or renaming columns automatically modify the results of `select *`. Listing columns individually gives you more precise control over the results.

8.2.2 Choose Specific Columns

You can select specific columns in a table by separating column names with commas.

To select only specific columns in a table, use:

```
select <column_name>[, <column_name>]...
     from <table_name>
```

For example:

```
select au_lname, au_fname
from authors
```

8.2.3 Rearrange the Column Order

The order in which you list column names in the `select` clause determines the order in which the columns appear in the query results.

The examples that follow show how to specify column order, displaying publisher names and identification numbers from all three rows in the `publishers` table. The first example prints `pub_id` first, followed by `pub_name`; the second reverses that order. The information is the same but the organization changes.

```
select pub_id, pub_name
from publishers
```

```
pub_id   pub_name
-----    ---------------
0736     New Age Books
0877     Binnet & Hardley
1389     Algodata Infosystems

(3 rows affected)
```

```
select pub_name, pub_id
from publishers
```

```
pub_name                     pub_id
---------------------        ------
New Age Books                0736
```

```
Binnet & Hardley              0877
Algodata Infosystems          1389

(3 rows affected)
```

8.2.4 Rename Columns in Query Results

The default heading for each column is the name given to the query results when it was created. You can rename a column heading for display purposes.

To rename a column heading, use one of these options instead of the column name in a select list:

- `<column_heading> = <column_name>`
- `<column_name> <column_heading>`
- `<column_name> as <column_heading>`

This provides a substitute name for the column. For example, to change `pub_name` to "Publisher" in the previous query, type any of the following statements:

- `select Publisher = pub_name, pub_id from publishers`
- `select pub_name Publisher, pub_id from publishers`
- `select pub_name as Publisher, pub_id from publishers`

The results of any of these statements look like this:

```
Publisher                 pub_id
----------------------    ------
New Age Books             0736
Binnet & Hardley          0877
Algodata Infosystems      1389

(3 rows affected)
```

8.2.5 Expressions

The `select` statement can also include one or more expressions, which allow you to manipulate the data that is retrieved.

```
select <expression> [, <expression>]...
     from <table_list>
```

An expression is any combination of constants, column names, functions, subqueries, or `case` expressions, connected by arithmetic or bitwise operators and parentheses. Expressions can include nested items (you can nest up to 32 functions in an expression).

If any table or column name in the list does not conform to the rules for valid identifiers, set the `quoted_identifier` option on and enclose the identifier in double quotes.

8.2.5.1 Quoted Strings in Column Headings

You can include any characters—including blanks—in a column heading if you enclose the entire heading in quotation marks.

You need not set the `quoted_identifier` option on. If the column heading is not enclosed in quotation marks, it must conform to the rules for identifiers. Both of these queries produce the same result:

```
select "Publisher's Name" = pub_name from publishers
```

```
select pub_name "Publisher's Name" from publishers
```

```
Publisher’s Name
----------------
New Age Books
Binnet & Hardley
Algodata Infosystems

(3 rows affected)
```

You can also use Transact-SQL reserved words in quoted column headings. For example, the following query, using the reserved word `sum` as a column heading, is valid:

```
select "sum" = sum(total_sales) from titles
```

Quoted column headings cannot be more than 255 bytes long.

> **Note**
>
> Before using quotes around a column name in a `create table`, `alter table`, `select into`, or `create view` statement, you must `set quoted_identifier on`.

8.2.5.2 Character Strings in Query Results

You can write queries so that the results contain strings of characters. Enclose the string in single or double quotation marks, and separate it from other elements in the select list with a comma.

Use double quotation marks if there is an apostrophe in the string—otherwise, the apostrophe is interpreted as a single quotation mark.

Here is a query with a character string:

```
select "The publisher’s name is", Publisher = pub_name
from publishers
```

```
                              Publisher
------------------------      --------------------
The publisher’s name is       New Age Books
The publisher’s name is       Binnet & Hardley
The publisher’s name is       Algodata Infosystems

(3 rows affected)
```

8.2.5.3 Computed Values in the select List

You can perform certain arithmetic operations on `date/time` columns using the date functions.

You can use all of these operators in the select list with column names and numeric constants in any combination. For example, to see what a projected sales increase of 100 percent for all the books in the `titles` table looks like, enter:

```
select title_id, total_sales, total_sales * 2
from titles
```

Here are the results:

```
title_id     total_sales
--------     -----------       ---------
BU1032              4095            8190
BU1111              3876            7752
BU2075             18722           37444
BU7832              4095            8190
MC2222              2032            4064
MC3021             22246           44492
MC3026              NULL            NULL
PC1035              8780           17560
PC8888              4095            8190
PC9999              NULL            NULL
PS1372               375             750
PS2091              2045            4090
PS2106               111             222
PS3333              4072            8144
PS7777              3336            6672
TC3218               375             750
TC4203             15096           30192
TC7777              4095            8190

(18 rows affected)
```

Notice the null values in the `total_sales` column and the computed column. Null values have no explicitly assigned values. When you perform any arithmetic operation on a null value, the result is NULL. You can give the computed column a heading, “proj_sales” for example, by entering:

```
select title_id, total_sales,
     proj_sales = total_sales * 2
from titles
```

```
title_id     total_sales     proj_sales
---------    -----------     -----------
BU1032             4095            8190
 ....
```

Try adding character strings such as “Current sales =” and “Projected sales are” to the `select` statement. The column from which the computed column is generated does not have to appear in the select list. The `total_sales` column, for example, is shown in these sample queries only for comparison of its values with the values from the `total_sales` * 2 column. To see only the computed values, enter:

```
select title_id, total_sales * 2
from titles
```

Arithmetic operators also work directly with the data values in specified columns, when no constants are involved. For example:

```
select title_id, total_sales * price
from titles
```

```
title_id
--------     ----------
BU1032        81,859.05
BU1111        46,318.20
BU2075        55,978.78
BU7832        81,859.05
MC2222        40,619.68
MC3021        66,515.54
MC3026             NULL
PC1035       201,501.00
PC8888        81,900.00
PC9999             NULL
PS1372         8,096.25
PS2091        22,392.75
PS2106           777.00
PS3333        81,399.28
PS7777        26,654.64
TC3218         7,856.25
TC4203       180,397.20
TC7777        61,384.05

(18 rows affected)
```

Computed columns can also come from more than one table. The joining and subqueries chapters in this manual include information on multitable queries.

As an example of a join, this query multiplies the number of copies of a psychology book sold by an outlet (the `qty` column from the `salesdetail` table) by the price of the book (the `price` column from the `titles` table).

```
select salesdetail.title_id, stor_id, qty * price
from titles, salesdetail
where titles.title_id = salesdetail.title_id
and titles.title_id = "PS2106"
```

```
title_id         stor_id
---------------- -----------  ------
PS2106           8042         210.00
PS2106           8042         350.00
PS2106           8042         217.00

(3 rows affected)
```

Related Information

Transact-SQL Functions [page 471]

8.2.5.4 Arithmetic Operator Precedence

When there is more than one arithmetic operator in an expression; multiplication, division, and modulo are calculated first, followed by subtraction and addition.

If all arithmetic operators in an expression have the same level of precedence, the order of execution is left to right. Expressions in parentheses take precedence over all other operations.

For example, the following `select` statement multiplies the total sales of a book by its price to calculate a total dollar amount, then subtracts from that one half of the author's advance.

```
select title_id, total_sales * price - advance / 2
from titles
```

The product of `total_sales` and `price` is calculated first, because the operator is multiplication. Next, the advance is divided by 2, and the result is subtracted from `total_sales` * `price`.

To avoid misunderstandings, use parentheses. The following query has the same meaning and gives the same results as the previous one, but it is easier to understand:

```
select title_id,(total_sales * price) - (advance /2)
from titles
```

```
title_id
--------     ----------
 BU1032       79,359.05
 BU1111       43,818.20
 BU2075       50,916.28
 BU7832       79,359.05
 MC2222       40,619.68
 MC3021       59,015.54
 MC3026            NULL
 PC1035      198,001.00
 PC8888       77,900.00
 PC9999            NULL
 PS1372        4,596.25
 PS2091       21,255.25
 PS2106       -2,223.00
 PS3333       80,399.28
 PS7777       24,654.64
 TC3218        4,356.25
 TC4203      178,397.20
 TC7777       57,384.05

(18 rows affected)
```

Use parentheses to change the order of execution; calculations inside parentheses are handled first. If parentheses are nested, the most deeply nested calculation has precedence. For example, the result and meaning of the preceding example is changed if you use parentheses to force evaluation of the subtraction before the division:

```
select title_id, (total_sales * price - advance) /2
from titles
```

```
title_id
--------   -----------------------
BU1032                  38,429.53
BU1111                  20,659.10
BU2075                  22,926.89
```

```
BU7832                  38,429.53
MC2222                  20,309.84
MC3021                  25,757.77
MC3026                       NULL
PC1035                  97,250.50
PC8888                  36,950.00
PC9999                       NULL
PS1372                     548.13
PS2091                  10,058.88
PS2106                  -2,611.50
PS3333                  39,699.64
PS7777                  11,327.32
TC3218                     428.13
TC4203                  88,198.60
TC7777                  26,692.03

(18 rows affected)
```

8.2.6 Select Text, Unitext, and Image Values

`text`, `unitext`, and `image` values can be quite large. When a select list includes the values for these datatypes, the limit on the length of the data returned depends on the setting of the `<@@textsize>` global variable.

The default setting for `<@@textsize>` depends on the software you use to access SAP ASE; the default value for for `isql` is 32K. To change the value, use the `set` command:

```
set textsize 2147483648
```

With this setting of `<@@textsize>`, a `select` statement that includes a `text` column displays only the first 2 gigabytes of the data.

> **Note**
>
> When you select `image` data, the returned value includes the characters “0x”, which indicates that the data is hexadecimal. These two characters are counted as part of `<@@textsize>`.

To reset `<@@textsize>` to the SAP ASE default value, use:

```
set textsize 0
```

If the actual length of returned data is less than `textsize`, the entire data string appears.

Related Information

Datatypes [page 191]

8.2.6.1 readtext Usage

The `readtext` command provides a way to retrieve `text`, `unitext`, and `image` values and retrieve only a selected portion of a column's data.

`readtext` requires the name of the table and column, the text pointer, a starting offset within the column, and the number of characters or bytes to retrieve. This example finds six characters in the `copy` column in the `blurbs` table:

```
declare @val binary(16)
select @val = textptr(copy) from blurbs
where au_id = "648-92-1872"
readtext blurbs.copy @val 2 6 using chars
```

In the example, after the `<@val>` local variable has been declared, `readtext` displays characters 3 – 8 of the `copy` column, since the offset was 2.

Instead of storing potentially large `text`, `unitext`, and `image` data in the table, SAP ASE stores it in a special structure. A text pointer (`textptr`) which points to the page where the data is actually stored is assigned. When you retrieve data using `readtext`, you actually retrieve `textptr`, which is a 16-byte `varbinary` string. To avoid this, declare a local variable to hold `textptr`, and then use the variable with `readtext`, as in the example above.

Related Information

Text and Image Functions [page 475]

8.2.7 select List Summary

The `select` list can include * (all columns in `create table` order), a list of column names in any order, character strings, column headings, and expressions (including arithmetic operators).

For example:

```
select titles.*
from titles
```

```
select Name = au_fname, Surname = au_lname
from authors
```

```
select Sales = total_sales * price,
ToAuthor = advance,
ToPublisher = (total_sales * price) - advance
from titles
```

```
select "Social security #", au_id
from authors
```

```
select this_year = advance, next_year = advance
   + advance/10, third_year = advance/2,
    "for book title #", title_id
from titles
```

```
select "Total income is",
Revenue = price * total_sales,
"for", Book# = title_id
from titles
```

Related Information

Aggregates, Grouping, and Sorting [page 285]

8.3 select for update

`select for update` exclusively locks rows in datarows-locked tables for subsequent updates within the same transaction, and for updatable cursors.

functionality is automatically available to clients when the for update clause is added to a select statement and to any updatable cursors opened within the clients. `select for update` is supported at isolation levels 1, 2, and 3.

`select for update` can be issued as a language statement outside of a cursor context. With both language statements and cursors, you must execute `select for update` within a `begin transaction` command or in chained mode.

If you run `select for update` within a cursor context, the cursor `open` and `fetch` statements must be within the context of a transaction.

The synax is:

```
select <col-list> from … where …
    [for update[ of <col-list> ]
```

> **Note**
>
> To obtain exclusive locks, you must set the `select for update` configuration parameter to 1, and include the `for update` clause.

8.3.1 Use select for update in Cursors and DML

`select for update` behavior is based on the value of the configuration parameter `select for update`.

- 0 – `select for update` is available only through cursors.
- 1 – you can use `select for update` at the language level, outside of a cursor context.

- `select for update` is supported for language statements and for cursors.
- Exclusive locks are acquired for qualifying rows of `select for update`, thereby blocking other readers and writers provided that you are:
  - Using datarows-locked tables
  - Using the command within a transaction-context, or in chained mode
- `select for update` can have an `order by` clause for both language statements and cursors. Using an `order by` clause with a `for update` clause allows the cursor to be updatable.

8.3.2 Concurrency Issues

If a session has an open transaction that is executing `select for update` at isolation levels 1, 2, or 3, a second concurrent session issuing data manipulation language (DML) statements on the same table may be blocked, depending on the type of transaction issued by the second session and its isolation level.

The states of transactions in second concurrent session are:

Transaction	Isolation Level			
	0	1	2	3
`select` qualified rows	Not blocked	Not blocked[1]	Blocked	Blocked
`select` unqualified rows	Not blocked	Not blocked	Not blocked[2]	Not blocked[2]
`update` qualified rows	Blocked[3]	Blocked	Blocked	Blocked
`update` unqualified rows	Not blocked[2,3]	Not blocked[2]	Not blocked[2]	Not blocked[2]
`select for update` qualified rows	N/A[4]	Blocked	Blocked	Blocked
`select for update` unqualified rows	N/A[4]	Not blocked[2]	Not blocked[2]	Not blocked[2]
`delete` qualified rows	Blocked[3]	Blocked	Blocked	Blocked
`delete` unqualified rows	Not blocked[2,3]	Not blocked[2]	Not blocked[2]	Not blocked[2]
`insert`	Not blocked[2,3]	Not blocked[2]	Not blocked[2]	Not blocked[2]

[1]SAP ASE does not block select commands at isolation level 1 unless the select list contains a large object (LOB) such as text, image, or unitext.

[2] If the first session issues select for update at isolation level 3, SAP ASE prevents "phantom rows" by exclusively locking more rows than just the qualifying rows. In this case, SAP ASE blocks the second session on these additional unqualified rows.

[3] Although the second session issues these DMLs at isolation level 0, SAP ASE executes them at isolation level 2.

[4]SAP ASE does not support select for update at isolation level 0.

8.4 Eliminate Duplicate Query Results with Distinct

The `distinct` keyword eliminates duplicate rows from the default results of a `select` statement.

For compatibility with other implementations of SQL, the use of `all` is allowed to explicitly query all rows. The default for the `select` statements is `all`. If you do not specify `distinct`, by default, all rows are returned, including duplicates.

For example, here is the result of searching for all the author identification codes in the `titleauthor` table without `distinct`:

```
select au_id
from titleauthor
```

```
au_id
-----------
172-32-1176
213-46-8915
213-46-8915
238-95-7766
267-41-2394
267-41-2394
274-80-9391
409-56-7008
427-17-2319
472-27-2349
486-29-1786
486-29-1786
648-92-1872
672-71-3249
712-45-1867
722-51-5454
724-80-9391
724-80-9391
756-30-7391
807-91-6654
846-92-7186
899-46-2035
899-46-2035
998-72-3567
998-72-3567

(25 rows affected)
```

There are some duplicate listings. Use `distinct` to eliminate them.

```
select distinct au_id
```

```
from titleauthor
```

```
au_id
-----------
172-32-1176
213-46-8915
238-95-7766
267-41-2394
274-80-9391
409-56-7008
427-17-2319
472-27-2349
486-29-1786
648-92-1872
672-71-3249
712-45-1867
722-51-5454
724-80-9391
756-30-7391
807-91-6654
846-92-7186
899-46-2035
998-72-3567

(19 rows affected)
```

`distinct` treats multiple null values as duplicates. In other words, when `distinct` is included in a `select` statement, only one NULL is returned, no matter how many null values are encountered.

When used with the `order by` clause, `distinct` can return multiple values.

Related Information

order by and group by Used with select distinct [page 308]

8.5 Specify Tables with the from Clause

The `from` clause is required in every `select` statement that retrieves data from tables or views. Use it to list all the tables and views containing columns included in the select list and in the `where` clause.

If the `from` clause includes more than one table or view, separate them with commas.

At most, a query can reference 250 tables and 46 worktables (such as those created by aggregate functions). The 250-table limit includes:

- Tables (or views on tables) listed in the `from` clause
- Each instance of multiple references to the same table (self-joins)
- Tables referenced in subqueries
- Tables being created with `into`
- Base tables referenced by the views listed in the `from` clause

For more information, see *Reference Manual: Commands*.

- Table names can be 1 – 255 bytes long.
- You can use a letter, @, #, or _ as the first character.
- The characters that follow can be digits, letters, or @, #, $, _, ¥, or £.
- Temporary table names must begin either with "#" (pound sign), if they are created outside `tempdb`, or with "`tempdb`".
- Temporary table names cannot be longer than 238 bytes, as SAP ASE adds an internal numeric suffix of 17 bytes to ensure that the name is unique.

The full naming syntax for tables and views is always permitted in the `from` clause:

```
<database>.<owner>.<table_name>
     <database>.<owner>.<view_name>
```

However, the full naming syntax is required only if there is potential confusion about the name.

To save typing, you can assign table names correlation names. Assign the correlation name in the `from` clause by giving the correlation name after the table name, like this:

```
select p.pub_id, p.pub_name
from publishers p
```

All other references to that table (for example, in a `where` clause) must also use the correlation name. Correlation names cannot begin with a numeral.

Related Information

Databases and Tables [page 50]

8.6 Select Rows Using the where Clause

The `where` clause in a `select` statement specifies the search conditions that determine which rows are retrieved.

The general format is:

```
select <select_list>
     from <table_list>
     where <search_conditions>
```

Search conditions, or qualifications, in the `where` clause include:

- Comparison operators (=, <, >, and so on)

  ```
  where advance * 2 > total_sales * price
  ```

- Ranges (`between` and `not between`)

  ```
  where total_sales between 4095 and 12000
  ```

- Lists (`in, not in`)

```
where state in ("CA", "IN", "MD")
```

- Character matches (`like` and `not like`)

```
where phone not like "415%"
```

- Unknown values (`is null` and `is not null`)

```
where advance is null
```

- Combinations of search conditions (`and`, `or`)

```
where advance < 5000 or total_sales between 2000
  and 2500
```

The `where` keyword can also introduce:

- Join conditions
- Subqueries

> **Note**
>
> The only `where` condition that you can use on `text` columns is `like` (or `not like`).

SAP ASE does not necessarily evaluate and execute predicates in left-to-right order. Instead, SAP ASE can evaluate and execute predicates in any order. For example, for this query:

```
where x != 0 and y = 10 or z = 100
```

SAP ASE may not evaluate and execute `x != 0` first.

For more information on search conditions, see the *Reference Manual: Commands*.

Related Information

Joins: Retrieve Data from Several Tables [page 320]
Subqueries: Queries Within Other Queries [page 257]

8.6.1 Comparison Operators in where Clauses

Place apostrophes or quotation marks around all `char`, `nchar`, `unichar`, `unitext`, `varchar`, `nvarchar`, `univarchar`, `text`, and `date/time` data.

For the purposes of comparison, trailing blanks are ignored. For example, “Dirk” is the same as “Dirk ”. In comparing dates, < means earlier than, and > means later than.

```
select *
from titleauthor
```

```
where royaltyper < 50
```

```
select authors.au_lname, authors.au_fname
from authors
where au_lname > "McBadden"
```

```
select au_id, phone
from authors
where phone != "415 658-9932"
```

```
select title_id, newprice = price * 1.15
from  pubs2..titles
where advance > 5000
```

`not` negates an expression. Either of the following two queries finds all business and psychology books that have advances of less than 5500. Note the difference in position between the negative logical operator (`not`) and the negative comparison operator (`!>`).

```
select title_id, type, advance
from titles
where (type = "business" or type = "psychology")
and not advance >5500
```

```
select title_id, type, advance
from titles
where (type = "business" or type = "psychology")
and advance !>5500
```

Both return the same result set:

```
title_id  type             advance
--------  ------------    --------
BU1032    business        5,000.00
BU1111    business        5,000.00
BU7832    business        5,000.00
PS2091    psychology      2,275.00
PS3333    psychology      2,000.00
PS7777    psychology      4,000.00

(6 rows affected)
```

Related Information

Managing Data [page 355]

8.6.2 Ranges (between and not between)

Use the `between` keyword to specify an inclusive range.

For example, to find all the books with sales between and including 4095 and 12,000, use:

```
select title_id, total_sales
```

```
from titles
where total_sales between 4095 and 12000
```

```
title_id  total_sales
------    -----------
BU1032           4095
BU7832           4095
PC1035           8780
PC8888           4095
TC7777           4095

(5 rows affected)
```

You can specify an exclusive range using the greater than (>) and less than (<) operators:

```
select title_id, total_sales
from titles
where total_sales > 4095 and total_sales < 12000
```

```
title_id  total_sales
------    -----------
PC1035           8780

(1 row affected)
```

`not between` finds all rows outside the specified range. To find all the books with sales outside the 4095 to 12,000 range, enter:

```
select title_id, total_sales
from titles
where total_sales not between 4095 and 12000
```

```
title_id     total_sales
--------     -----------
BU1111              3876
BU2075             18722
MC2222              2032
MC3021             22246
PS1372               375
PS2091              2045
PS2106               111
PS3333              4072
PS7777              3336
TC3218               375
TC4203             15096

(11 rows affected)
```

8.6.3 Lists (in and not in)

The `in` keyword allows you to select values that match any one of a list of values. `not in` finds rows that do not match the values in the list.

The expression can be a constant or a column name, and the values list can be a set of constants or a subquery. Separate items following the `in` keyword by commas, and enclose then entire list of values in parentheses. Use single or double quotes around `char`, `varchar`, `unichar`, `unitext`, `univarchar`, and `datetime` values.

For example, to list the names and states of all authors who live in California, Indiana, or Maryland, use:

```
select au_lname, state
from authors
where state in ("CA", "IN", "MD")
```

```
 au_lname         state
 -----------      -----
 White            CA
 Green            CA
 Carson           CA
 O'Leary          CA
 Straight         CA
 Bennet           CA
 Dull             CA
 Gringlesby       CA
 Locksley         CA
 Yokomoto         CA
 DeFrance         IN
 Stringer         CA
 MacFeather       CA
 Karsen           CA
 Panteley         MD
 Hunter           CA
 McBadden         CA
```

Using the `in` keyword in the query produces the same result set as the following, longer query:

```
select au_lname, state
from authors
where state = "CA" or state = "IN" or state = "MD"
```

Perhaps the most important use for the `in` keyword is in nested queries, which are also called subqueries.

For example, suppose you want to know the names of the authors who receive less than 50 percent of the total royalties on the books they coauthor. The `authors` table stores author names and the `titleauthor` table stores royalty information. By putting the two tables together using `in`, but without listing the two tables in the same `from` clause, you can extract the information you need. The following query:

- Searches the `titleauthor` table for all `au_id`s of authors making less than 50 percent of the royalty on any one book.
- Selects from the `authors` table all the author names with `au_id`s that match the results from the `titleauthor` query. The results show that several authors fall into the less than 50 percent category.

```
select au_lname, au_fname
from authors
where au_id in
  (select au_id
   from titleauthor
   where royaltyper <50)
```

```
au_lname          au_fname
--------------    ------------
Green             Marjorie
O'Leary           Michael
Gringlesby        Burt
Yokomoto          Akiko
MacFeather        Stearns
Ringer            Anne

(6 rows affected)
```

`not in` finds the authors that do not match the items in the list. The following query finds the names of authors who do not make less than 50 percent of the royalties on at least one book.

```
select au_lname, au_fname
from authors
where au_id not in
  (select au_id
   from titleauthor
   where royaltyper <50)
```

```
au_lname          au_fname
---------------   ------------
White             Johnson
Carson            Cheryl
Straight          Dick
Smith             Meander
Bennet            Abraham
Dull              Ann
Locksley          Chastity
Greene            Morningstar
Blotchet-Halls    Reginald
del Castillo      Innes
DeFrance          Michel
Stringer          Dirk
Karsen            Livia
Panteley          Sylvia
Hunter            Sheryl
McBadden          Heather
Ringer            Albert
Smith             Gabriella

(18 rows affected)
```

Related Information

Subqueries: Queries Within Other Queries [page 257]

8.6.4 Matching Character Strings: like

The `like` keyword searches for a character string that matches a pattern.

`like` is used with:

- `char`
- `varchar`
- `nchar`
- `nvarchar`
- `unichar`
- `unitext`
- `univarchar binary`
- `varbinary`

- text,
- date/time

The syntax for like is:

```
{where | having} [not]
     <column_name> [not] like  "<match_string>"
```

These are the special symbols for matching character strings:

Symbols	Meaning
%	Matches any string of zero or more characters.
_	Matches a single character.

[<specifier>]	Brackets enclose ranges or sets, such as [a – f] or [abcdef]. <specifier> can take two forms:
	• <rangespec1> – <rangespec2>:
	  ○ <rangespec1> indicates the start of a range of characters.
	  ○ <rangespec2> indicates the end of a range of characters.
	  ○ The symbol – is a special character, indicating a range.
	• <set> can be composed of any discrete set of values, in any order, such as [a2bR].The range [a – f], and the sets [abcdef] and [fcbdae] return the same set of values.
	Specifiers are case-sensitive.

[^<specifier>]	A caret (^) preceding a specifier indicates noninclusion. [^a – f] means "not in the range a – f"; [^a2bR] means "not a, 2, b, or R."

You can match the column data to constants, variables, or other columns that contain the wildcard characters. When using constants, enclose the match strings and character strings in quotation marks. For example, using like with the data in the authors table:

- like "Mc%" searches for every name that begins with ''Mc'' (McBadden).
- like "%inger" searches for every name that ends with ''inger'' (Ringer, Stringer).
- like "%en%" searches for every name containing ''en'' (Bennet, Green, McBadden).
- like "_heryl" searches for every six-letter name ending with ''heryl'' (Cheryl).
- like "[CK]ars[eo]n" searches for ''Carsen,'' ''Karsen,'' ''Carson,'' and ''Karson'' (Carson).
- like "[M-Z]inger" searches for all names ending with ''inger'' that begin with any single letter from M to Z (Ringer).
- like "M[^c]%" searches for all names beginning with ''M'' that do not have ''c'' as the second letter.

This query finds all the phone numbers in the authors table that have an area code of 415:

```
select phone
from authors
where phone like "415%"
```

The only where condition you can use on text columns is like. This query finds all the rows in the blurbs table where the copy column includes the word "computer":

```
select * from blurbs
```

```
where copy like "%computer%"
```

SAP ASE interprets wildcard characters used without `like` as literals rather than as a pattern; they represent exactly their own values. The following query attempts to find any phone numbers that consist of the four characters “415%” only. It does not find phone numbers that start with 415.

```
select phone
from authors
where phone = "415%"
```

When you use `like` with `datetime` values, SAP ASE converts the values to the standard `datetime` format, and then to `varchar` or `univarchar`. Since the standard storage format does not include seconds or milliseconds, you cannot search for seconds or milliseconds with `like` and a pattern.

Use `like` when you search for `date` and `time` values, since these datatype entries may contain a variety of date parts. For example, if you insert “9:20” into a `datetime` column named `arrival_time`, the query below does not find the value, because SAP ASE converts the entry into “Jan 1 1900 9:20AM”:

```
where arrival_time = "9:20"
```

However, this query finds the 9:20 value:

```
where arrival_time like "%9:20%"
```

You can also use the `date` and `time` datatypes for `like` transactions.

8.6.4.1 not like Usage

You can use the same wildcard characters with `not like` that you can use with `like`.

For example, to find all the phone numbers in the `authors` table that do *not* have 415 as the area code, use either of these queries:

```
select phone
from authors
where phone not like "415%"
```

```
select phone
from authors
where not phone like "415%"
```

8.6.4.2 Different Results Using not like and ^

You cannot always duplicate `not like` patterns with `like` and the negative wildcard character [^]. Match strings with negative wildcard characters are evaluated in steps, one character at a time. If the match fails at any point in the evaluation, it is eliminated.

For example, this query finds the system tables in a database that have names beginning with “sys”:

```
select name
from sysobjects
where name like "sys%"
```

If you have a total of 32 objects and `like` finds 13 names that match the pattern, `not like` finds the 19 objects that do not match the pattern.

```
where name not like "sys%"
```

However, this pattern may produce different results:

```
like [^s][^y][^s]%
```

Instead of 19, you might get only 14, with all the names that begin with “s” *or* have “y” as the second letter *or* have “s” as the third letter eliminated from the results, as well as the system table names.

8.6.4.3 Use Wildcard Characters as Literal Characters

You can search for wildcard characters by using the escape character and searching for them as literals. There are two ways to use the wildcard characters as literals in a `like` match string: square brackets and the `escape` clause.

The match string can also be a variable or a value in a table that contains a wildcard character.

Square Brackets (Transact-SQL Extension)

Use square brackets for the percent sign, the underscore, and right and left brackets. To search for a dash, rather than using it to specify a range, use the dash as the first character inside a set of brackets.

like Clause	Searches for
`like "5%"`	5 followed by any string of 0 or more characters
`like "5[%]"`	5%
`like "_n"`	an, in, on, and so forth
`like "[_]n"`	_n

like Clause	Searches for
`like "[a-cdf]"`	a, b, c, d, or f
`like "[-acdf]"`	-, a, c, d, or f
`like "[ [ ]"`	[
`like "[ ] ]"`	]

escape Clause (SQL-Compliant)

Use the `escape` clause to specify an escape character in the `like` clause. An escape character must be a single-character string. Any character in the server's default character set can be used.

like clause	Searches for
`like "5@%" escape "@"`	5%
`like "*_n" escape "*"`	_n
`like "%80@%%" escape "@"`	String containing 80%
`like "*_sql**%" escape "*"`	String containing _sql*
`like "%#####_#%%" escape "#"`	String containing ##_%

8.6.4.4 Interaction of Wildcard Characters and Square Brackets

An escape character retains its special meaning within square brackets, unlike wildcard characters.

Do not use existing wildcard characters as escape characters in the escape clause, for these reasons:

- If you specify square brackets or percent sign ("_" or "%") as an escape character, it loses its special meaning within that `like` clause and acts only as an escape character.
- If you specify a left or right square bracket ("[" or "]") as an escape character, the Transact-SQL meaning of the bracket is disabled within that `like` clause.
- If you specify a hyphen or carrot ("-" or "^") as an escape character, it loses the special meaning that it normally has within square brackets and acts only as an escape character.

8.6.4.5 Use Trailing Blanks and %

SAP ASE truncates trailing blanks following “%” in a `like` clause to a single trailing blank. `like` ‘‘% ’’ (percent sign followed by 2 spaces) matches ‘‘X ’’ (one space); ‘‘X ’’ (two spaces); ‘‘X ’’ (three spaces), or any number of trailing spaces.

8.6.4.6 Use Wildcard Characters in Columns

You can use wildcard characters for columns and column names. You might want to create a table called `special_discounts` in the `pubs2` database to run a price projection for a special sale:

```
create table special_discounts
id_type char(3), discount int)
insert into special_discounts
values("BU%", 10)
...
```

The table should contain the following data:

```
id_type discount
------- -----------
BU%              10
PS%              12
MC%              15
```

The following query uses wildcard characters in `id_type` in the `where` clause:

```
select title_id, discount, price, price - (price*discount/100)
from special_discounts, titles
where title_id like id_type
```

Here are the results of that query:

```
 title_id   discount      price
 -------- ----------- -------------- --------------
 BU1032            10          19.99          17.99
 BU1111            10          11.95          10.76
 BU2075            10           2.99           2.69
 BU7832            10          19.99          17.99
 PS1372            12          21.59          19.00
 PS2091            12          10.95           9.64
 PS2106            12           7.00           6.16
 PS3333            12          19.99          17.59
 PS7777            12           7.99           7.03
 MC2222            15          19.99          16.99
 MC3021            15           2.99           2.54
 MC3026            15           NULL           NULL

(12 rows affected)
```

This type of example permits sophisticated pattern matching without having to construct a series of `or` clauses.

8.6.5 “Unknown” Values: NULL

A NULL value in a column means no entry has been made in that column. A data value for the column is “unknown” or “not available.”

NULL is not synonymous with “zero” or “blank.” Rather, null values allow you to distinguish between a deliberate entry of zero for numeric columns (or blank for character columns) and a non-entry, which is NULL for both numeric and character columns.

In a column where null values are permitted:

- If you do not enter any data, SAP ASE automatically enters “NULL”.
- Users can explicitly enter the word “NULL” or “null” without quotation marks.

If you type the word ‘‘NULL’’ in a character column and include quotation marks, it is treated as data, rather than a null value.

Query results display the word NULL. For example, the `advance` column of the `titles` table allows null values. By inspecting the data in that column, you can tell whether a book had no advance payment by agreement (the row MC2222 has zero in the advance column) or whether the advance amount was not known when the data was entered (the row MC3026 has NULL in the advance column).

```
select title_id, type, advance
from titles
where pub_id = "0877"
```

```
title_id     type             advance
--------     ----------       ---------
MC2222       mod_cook              0.00
MC3021       mod_cook         15,000.00
MC3026       UNDECIDED             NULL
PS1372       psychology        7,000.00
TC3218       trad_cook         7,000.00
TC4203       trad_cook         4,000.00
TC7777       trad_cook         8,000.00

(7 rows affected)
```

8.6.5.1 SQL Standard for NULL Concatenation

Use `set sqlnull on` to implement SQL standard for NULL concatenation.

Standard SQL requires that string concatenation involving a NULL generates a NULL output. SAP ASE evaluates a string concatenated with NULL to the value of the string. A string concatenation involving a NULL is treated as a string with a 0 length and an empty string ("") is interpreted as a single space.

SAP ASE allows you to use the `set sqlnull` option to implement SQL standard for NULL concatenation.

This example, based on the table `staff_profile`, demonstrates the different output generated using the `sqlnull` option:

```
create table staff_profile(id int, firstname char(10) NULL,
surname char(10) NULL, city varchar(10) NULL, country varchar(10) NULL )
go
```

```
insert staff_profile values(001, 'Tom', 'Griffin', 'Dublin', 'US')
insert staff_profile values(002, 'Kumar', NULL, 'Pune', 'India')
insert staff_profile values(003,  NULL , 'Kobe', 'Tokyo', NULL)
insert staff_profile values(004, 'Steve', 'Lewis', 'London', 'UK')
insert staff_profile values(005, 'Hana', 'SAP', NULL, 'Germany')
insert staff_profile values(006, 'Wei', 'Ming', 'Shanghai', 'China')
insert staff_profile values(007, 'city-state', '  ', 'Singapore', '')
```

Output with the default value of `set sqlnull off`:

```
set sqlnull off
select id, rtrim(firstname) + '' + rtrim(surname) name, rtrim(city) + '' +
rtrim(country)
location from staff_profile
id           name           location
-------- ------------ ---------------------
1 Tom Griffin             Dublin US
2 Kumar                   Pune India
3 Kobe                    Tokyo
4 Steve Lewis             London UK
5 Hana SAP                Germany
6 Wei Ming                Shanghai China
7 city-state              Singapore
(6 rows affected)
```

Output with `set sqlnull on`:

```
set sqlnull on
select id, rtrim(firstname) + '' + rtrim(surname) name, rtrim(city) + '' +
rtrim(country)
location from staff_profile

id           name           location
-------- ------------ ---------------------
1 Tom Griffin             Dublin US
2 NULL                    Pune India
3 NULL                    NULL
4 Steve Lewis             London UK
5 Hana SAP                NULL
6 Wei Ming                Shanghai China
7 city-state              Singapore
(6 rows affected)
```

8.6.5.2 Test a Column for Null Values

Use `is null` in `where`, `if`, and `while` clauses to compare column values to NULL, and to select them or perform a particular action based on the results of the comparison.

Only columns that return a value of true are selected or result in the specified action; those that return false or unknown do not.

The following example selects only rows for which `advance` is less than 5000 or NULL:

```
select title_id, advance
from titles
where advance < 5000 or advance is null
```

SAP ASE treats null values in different ways, depending on the operators that you use and the type of values you are comparing. In general, the result of comparing null values is unknown, since it is impossible to

determine whether NULL is equal (or not equal) to a given value or to another NULL. The following cases return true when <expression> is any column, variable or literal, or combination of these, which evaluates as NULL:

- <expression> is null
- <expression> = null
- <expression> = @<x> where @<x> is a variable or parameter containing NULL. This exception facilitates writing stored procedures with null default parameters.
- <expression> != <n> where <n> is a literal not containing NULL and <expression> evaluates to NULL.

The negative versions of these expressions return true when the expression does not evaluate to NULL:

- <expression> is not null
- <expression> != null
- <expression> != @<x>

When the keywords like and not like are used instead of the operators = and !=, the opposite occurs. This comparison returns true:

- <expression> not like null

While this comparison returns false:

- <expression> like null

The far-right side of these expressions is a literal null, or a variable or parameter containing NULL. If the far-right side of the comparison is an expression (such as @<nullvar> + 1), the entire expression evaluates to NULL.

Null column values do not join with other null column values. Comparing null column values to other null column values in a where clause always returns unknown, regardless of the comparison operator, and the rows are not included in the results. For example, this query returns no result rows where column1 contains NULL in both tables (although it may return other rows):

```
select <column1>
from <table1>, <table2>
where <table1>.<column1> = <table2>.<column1>
```

These operators return results when used with a NULL:

- = returns all rows that contain NULL.
- != or <> returns all rows that do *not* contain NULL.

When set ansinull is on for SQL compliance, the = and != operators do not return results when used with a NULL. Regardless of the set ansinull option value, the following operators never return values when used with a NULL: <, <=, !<, >, >=, !>.

SAP ASE can determine that a column value is NULL. Thus, this is considered true:

```
<column1> =  NULL
```

However, the following comparisons can never be determined, since NULL means “having an unknown value:”

```
where <column1> > null
```

There is no reason to assume that two unknown values are the same.

This logic also applies when you use two column names in a `where` clause, that is, when you join two tables. A clause like "where column1 = column2" does not return rows where the columns contain null values.

You can also find null values or non-null values with this pattern:

```
where <column_name> is [not] null
```

For example:

```
where advance < 5000 or advance is null
```

Some of the rows in the `titles` table contain incomplete data. For example, a book called `The Psychology of Computer Cooking` (`title_id` = MC3026) has been proposed and its title, title identification number, and probable publisher have undetermined, null values in the `price`, `advance`, `royalty`, `total_sales`, and `notes` columns. Because null values do not match anything in a comparison, a query for all the title identification numbers and advances for books with advances of less than 5000 does not include `The Psychology of Computer Cooking`.

```
select title_id, advance
from titles
where advance < 5000
```

```
title_id  advance
--------  ----------
MC2222         0.00
PS2091     2,275.00
PS3333     2,000.00
PS7777     4,000.00
TC4203     4,000.00

(5 rows affected)
```

Here is a query for books with an advance of less than 5000 *or* a null value in the `advance` column:

```
select title_id, advance
from titles
where advance < 5000
  or advance is null
```

```
title_id  advance
--------  ----------
MC2222         0.00
MC3026         NULL
PC9999         NULL
PS2091      2,275.00
PS3333      2,000.00
PS7777      4,000.00
TC4203      4,000.00

(7 rows affected)
```

Related Information

Batches and Control-of-Flow Language [page 437]

Databases and Tables [page 50]
Managing Data [page 355]

8.6.5.3 Difference Between False and Unknown

There is an important logical difference between false and unknown: the opposite of false ("not false") is true, while the opposite of unknown is still unknown.

For example, "1 = 2" evaluates to false and its opposite, "1 != 2", evaluates to true. But "not unknown" is still unknown. If null values are included in a comparison, you cannot negate the expression to get the opposite set of rows or the opposite truth value.

8.6.5.4 Substitute a Value for NULLs

Use the `isnull` built-in function to substitute a particular value for nulls. The substitution is made only for display purposes; actual column values are not affected.

The syntax is:

```
isnull(<expression>, <value>)
```

For example, use the following statement to select all the rows from `titles`, and display all the null values in column `notes` with the value unknown.

```
select isnull(notes, "unknown")
from titles
```

8.6.5.5 Expressions that Evaluate to NULL

An expression with an arithmetic or bitwise operator evaluates to NULL if any of the operands is null.

For example, this evaluates to NULL if `column1` is NULL:

```
1 + column1
```

8.6.5.6 Concatenate Strings and NULL

If you concatenate a string and NULL, the expression evaluates to the string.

For example:

```
select "abc" + NULL + "def"
```

```
-----
abcdef
```

8.6.5.7 System-Generated NULLs

In Transact-SQL, system-generated NULLs, such as those that result from a system function like `convert`, behave differently than user-assigned NULLs.

For example, in the following statement, a not equals comparison of the user-provided NULL and 1 returns true:

```
if (1 != NULL) print "yes" else print "no"
```

```
yes
```

The same comparison with a system-generated NULL returns unknown:

```
if (1 != convert(integer, NULL))
print "yes" else print "no"
```

```
no
```

For more consistent behavior, enable `set ansinull` (set to `on`), so both system-generated and user-provided NULLs cause the comparison to return unknown.

8.6.6 Connect Conditions with Logical Operators

The logical operators `and`, `or`, and `not` connect search conditions in `where` clauses.

The syntax is:

```
{where | having} [not]
     <column_name> <join_operator> <column_name>
```

where `<join_operator>` is a comparison operator and `<column_name>` is the column used in the comparison. Qualify the name of the column if there is any ambiguity.

`and` joins two or more conditions and returns results only when *all* of the conditions are true. For example, the following query finds only the rows in which the author's last name is Ringer and the author's first name is Anne. It does not find the row for Albert Ringer.

```
select *
from authors
where au_lname = "Ringer" and au_fname = "Anne"
```

`or` also connects two or more conditions, but it returns results when *any* of the conditions is true. The following query searches for rows containing Anne or Ann in the `au_fname` column.

```
select *
from authors
where au_fname = "Anne" or au_fname = "Ann"
```

You can specify as many as 252 `and` and `or` conditions.

`not` negates the expression that follows it. The following query selects all the authors who do not live in California:

```
select * from authors
where not state = "CA"
```

When more than one logical operator is used in a statement, `and` operators are normally evaluated before `or` operators. You use parentheses to change the order of execution. For example:

```
select * from authors
where (city = "Oakland" or city = "Berkeley") and state = "CA"
```

8.6.6.1 Logical Operator Precedence

Arithmetic and bitwise operators are handled before logical operators. When more than one logical operator is used in a statement, `not` is evaluated first, then `and`, and finally `or`.

For example, the following query finds *all* the business books in the `titles` table, no matter what their advances are, as well as all psychology books that have an advance of more than 5500. The advance condition pertains only to psychology books because the `and` is handled before the `or`.

```
select title_id, type, advance
from titles
where type = "business" or type = "psychology"
  and advance > 5500
```

```
title_id  type        advance
--------  ----------  ----------
BU1032    business     5,000.00
BU1111    business     5,000.00
BU2075    business    10,125.00
BU7832    business     5,000.00
PS1372    psychology   7,000.00
PS2106    psychology   6,000.00

(6 rows affected)
```

You can change the meaning of the query by adding parentheses to force evaluation of the `or` first. This query finds all business and psychology books with advances of more than 5500:

```
select title_id, type, advance
from titles
where (type = "business" or type = "psychology")
  and advance > 5500
```

```
title_id  type         advance
--------  ----------   ---------
BU2075    business     10,125.00
PS1372    psychology    7,000.00
PS2106    psychology    6,000.00

(3 rows affected)
```

Related Information

Bitwise Operators [page 36]

8.7 Multiple select Items in a Nested exists Query

You can use multiple select items in nested queries.

The use of multiple columns in this example is the same as selecting a single `c1` or `c2` column in the nested `exists` query:

```
1> create table t1(c1 int, c2 int)
2> go
1> create table t2(c1 int, c2 int)
2> go
1> select * from t1 where exists (select c1, c2
   from t2)
2> go
```

You cannot mix an asterisk with other select items, such as:

```
1> select * from t1
    where exists (select t2.*, c1 from t2)
2> go
```

```
Msg 102, Level 15, State 1:
Line 1:
Incorrect syntax near ',',.
```

8.8 Use a Column Alias in Nested select Statements

You can use an column alias in the `select` list of nested `select` statements.

The column alias uses one of these forms:

- `column_heading = <expression>`
- `<expression> column_heading`
- `<expression> as column_heading`

For example, this example is equivalent to a `select` statement with the `as tableid` clause removed:

```
1> select *
2> from syscolumns c
3> where c.id in (
4> select o.id as tableid
5> from sysobjects o
6> where o.name like '%attr%')
```

SAP ASE ignores the alias (the allowed column heading) in this example.

9 Subqueries: Queries Within Other Queries

A subquery is a `select` statement that is nested inside another `select`, `insert`, `update`, or `delete` statement, inside a conditional statement, or inside another subquery.

You can also express subqueries as join operations.

Subqueries, also called inner queries, appear within a `where` or `having` clause of another SQL statement, or in the select list of a statement.

You can use subqueries to handle query requests that are expressed as the results of other queries. A statement that includes a subquery operates on rows from one table, based on its evaluation of the subquery's `select` list, which can refer either to the same table as the outer query, or to a different table. In Transact-SQL, you can use a subquery almost anywhere an expression is allowed, if the subquery returns a single value. A `case` expression can also include a subquery.

For example, this subquery lists the names of all authors whose royalty split is more than $75:

```
select au_fname, au_lname
from authors
where au_id in
   (select au_id
    from titleauthor
    where royaltyper > 75)
```

`select` statements that contain one or more subqueries are sometimes called nested queries or nested select statements.

The result of a subquery that returns no values is NULL. If a subquery returns NULL, the query failed.

Subquery Example

This is an example of how to find the books that have the same price as *Straight Talk About Computers*.

First find the price of *Straight Talk*:

```
select price
from titles
where title = "Straight Talk About Computers"
```

```
price
-------------
       $19.99

(1 row affected)
```

Use the results of the first query in a second query to find all the books that cost the same as *Straight Talk*:

```
select title, price
```

```
from titles
where price = $19.99
```

```
title                                          price
-----------------------------------------      -----
The Busy Executive's Database Guide            19.99
Straight Talk About Computers                  19.99
Silicon Valley Gastronomic Treats              19.99
Prolonged Data Deprivation: Four Case Studies  19.99
```

You can use a subquery to receive the same results in only one step:

```
select title, price
from titles
where price =
   (select price
    from titles
    where title = "Straight Talk About Computers")
```

```
title                                          price
---------------------------------------        -----
The Busy Executive's Database Guide            19.99
Straight Talk About Computers                  19.99
Silicon Valley Gastronomic Treats              19.99
Prolonged Data Deprivation: Four Case Studies  19.99
```

Related Information

Joins: Retrieve Data from Several Tables [page 320]

9.1 Subquery Restrictions

A subquery is subject to certain restrictions.

- The `<subquery_select_list>` can consist of only one column name, except in the `exists` subquery, where an `(*)` is usually used in place of the single column name. You can use an asterisk (`*`) in a nested `select` statement that is not an `exists` subquery.
  Do not specify more than one column name. Qualify column names with table or view names if there is ambiguity about the table or view to which they belong.
- Subqueries can be nested inside the `where` or `having` clause of an outer `select`, `insert`, `update`, or `delete` statement, inside another subquery, or in a select list. Alternatively, you can write many statements that contain subqueries as joins; SAP ASE processes such statements as joins.
- In Transact-SQL, a subquery can appear almost anywhere an expression can be used, if it returns a single value. SQL derived tables can be used in the `from` clause of a subquery wherever the subquery is used
- You cannot use subqueries in an `order by`, `group by`, or `compute by` list.
- You cannot include a `for browse` clause in a subquery.
- You cannot include a `union` clause in a subquery unless it is part of a derived table expression within the subquery.

- The select list of an inner subquery introduced with a comparison operator can include only one expression or column name, and the subquery must return a single value. The column you name in the `where` clause of the outer statement must be join-compatible with the column you name in the subquery select list.
- `You cannot include text`, `unitext`, or `image` datatypes in subqueries.
- Subqueries cannot manipulate their results internally, that is, a subquery cannot include the `order by` clause, the `compute` clause, or the `into` keyword.
- Correlated (repeating) subqueries are not allowed in the `select` clause of an updatable cursor defined by `declare cursor`.
- There is a limit of 50 nesting levels.
- The maximum number of subqueries on each side of a union is 250.
- The `where` clause of a subquery can contain an aggregate function only if the subquery is in a `having` clause of an outer query and the aggregate value is a column from a table in the `from` clause of the outer query.
- The result expression from a subquery is subject to the same limits as for any expression. The maximum length of an expression is 16K. See, "Expressions, Identifiers, and Wildcard "Characters," in the *Reference Manual: Building Blocks*.

Related Information

SQL-Derived Tables [page 129]

9.2 Qualify Column Names

Column names in a statement are implicitly qualified by tables that are referenced in the `from` clause at the same level.

In the following example, the table name `publishers` implicitly qualifies the `pub_id` column in the `where` clause of the outer query. The reference to `pub_id` in the select list of the subquery is qualified by the subquery's `from` clause—that is, by the `titles` table:

```
select pub_name
from publishers
where pub_id in
   (select pub_id
    from titles
    where type = "business")
```

This is what the query looks like with the implicit assumptions spelled out:

```
select pub_name
from publishers
where publishers.pub_id in
   (select titles.pub_id
    from titles
    where type = "business")
```

It is never wrong to state the table name explicitly, and you can override implicit assumptions about table names by using explicit qualifications.

9.3 Subqueries with Correlation Names

Table correlation names are required in self-joins because the table being joined to itself appears in two different roles. You can use correlation names in nested queries that refer to the same table in both an inner query and an outer query.

For example, to find authors who live in the same city as Livia Karsen:

```
select au1.au_lname, au1.au_fname, au1.city
from authors au1
where au1.city in
   (select au2.city
    from authors au2
    where au2.au_fname = "Livia"
    and au2.au_lname = "Karsen")
```

```
au_lname     au_fname     city
-----------  ---------    -------
Green        Marjorie     Oakland
Straight     Dick         Oakland
Stringer     Dirk         Oakland
MacFeather   Stearns      Oakland
Karsen       Livia         Oakland
```

Explicit correlation names make it clear that the reference to `authors` in the subquery is not the same as the reference to `authors` in the outer query.

Without explicit correlation, the subquery is:

```
select au_lname, au_fname, city
from authors
where city in
   (select city
    from authors
    where au_fname = "Livia"
    and au_lname = "Karsen")
```

Alternatively, state the above query, as well as other statements in which the subquery and the outer query refer to the same table, as self-joins:

```
select au1.au_lname, au1.au_fname, au1.city
from authors au1, authors au2
where au1.city = au2.city
and au2.au_lname = "Karsen"
and au2.au_fname = "Livia"
```

A subquery restated as a join may not return the results in the same order; additionally, the join may require the `distinct` keyword to eliminate duplicates.

Related Information

Joins: Retrieve Data from Several Tables [page 320]

9.4 Multiple Levels of Nesting

A subquery can include one or more subqueries. You can nest up to 250 subqueries in a statement.

To find the names of authors who have participated in writing at least one popular computing book, enter:

```
select au_lname, au_fname
from authors
where au_id in
   (select au_id
    from titleauthor
    where title_id in
       (select title_id
        from titles
        where type = "popular_comp") )
```

```
au_lname                au_fname
---------------------- ------------
Carson                  Cheryl
Dull                    Ann
Locksley                Chastity
Hunter                  Sheryl

(4 rows affected)
```

The outermost query selects all author names. The next query finds the authors' IDs, and the innermost query returns the title ID numbers PC1035, PC8888, and PC9999.

You can also express this query as a join:

```
select au_lname, au_fname
from authors, titles, titleauthor
where authors.au_id = titleauthor.au_id
and titles.title_id = titleauthor.title_id
and type = "popular_comp"
```

9.5 Using an Asterisk in Nested select Statements

You can use an asterisk (`*`) in a nested `select` statement that is not an `exists` subquery as long as the asterisk follows certain conditions.

The asterisk:

- Is the only item in the `select` statement.
- Resolves to a single table column for the nested query.

In addition, you can:

- Restrict the selected columns in your nested query to only those belonging to a specific table by using the `<qualifier>.*` format, where `<qualifier>` is one of the tables in the `from` clause.
- Use the asterisk in a nested query that includes a `group by` clause.

When an asterisk resolves to a single table column for the nested query, the query is equivalent to explicitly using a single table column.

This is a valid nested query, because `t2` only has one column:

```
1> create table t1(c1 int, c2 int)
2> create table t2(c1 int)
3>go
1>select * from t1 where c1 in (select * from t2)
2>go
```

The nested `select` statement is equivalent to:

```
1> select * t1 where c1 in (select c1 from t2)
2> go
```

9.5.1 Use Table-Name Qualifiers

You can use an asterisk in the form of `<qualifier>.*` (`<qualifier> <period> asterisk`), to select only those columns that are in your specified table.

For example:

```
1> create table t1(c1 int, c2 int)
2> go
1> create table t2(c1 int)
2> go
1> select * from t1
2> where c1 in (select t2.* from t1, t2)
3> go
```

The nested `select` statement is equivalent to:

```
1> select * from t1
2> where c1 in (select t2.c1 from t1, t2)
3> go
```

9.5.2 Use Nested Queries with group by

You can use an asterisk in a nested `group by` query as long as the group-by table has a single column.

Such as:

```
1> select * from t1
2> where c1 in (select * from t2 group by c1)
3> go
```

The nested `group by` query example is equivalent to:

```
1> select * from t1
2> where c1 in (select c1 from t2 group by c1)
3> go
```

9.5.3 Usage and Examples of Asterisks in select Statements

SAP ASE automatically replaces asterisks in queries with actual column names before saving new stored procedures, views, and triggers.

This replacement persists even if you alter a table to add columns. SAP ASE does not allow more than one column, however, when the replacement of the asterisk introduces an additional column. This incorrect behavior persists until you drop and re-create the text. For example:

```
1> create table t1(c1 int, c2 int)
2> go
1> create table t2(c1 int)
2> go
1> create proc p1
2> as
3> select * from t1 where c1 in (select * from t2)
4> go
1> exec p1
2> go
 c1          c2
 ----------- -----------
(0 rows affected)
(return status = 0)
1> sp_helptext p1
2> go
# Lines of Text
 ---------------
               2
(1 row affected)
text
------------------------------------------------------------------------------------
---------------------------
create proc p1
as/* SAP ASE has expanded all '*' elements in the following statement */
select t1.c1, t1.c2
    from t1 where c1 in (select t2.c1 from t2)
(2 rows affected)
(return status = 0)
1> alter table t2 add c2 int null
2> go
1> exec p1
2> go
 c1          c2
 ----------- -----------
(0 rows affected)
(return status = 0)
1> exec p1 with recompile
2> go
 c1          c2
 ----------- -----------
(0 rows affected)
(return status = 0)
1> drop proc p1
2> go
1> create proc p1
```

```
2> as
3> select * from t1 where c1 in (select * from t2)
4> go
Msg 299, Level 16, State 1:
Procedure 'p1', Line 4:
The symbol '*' can only be used for a non-EXISTS
subquery select list when the subquery is on a single
table with a single column.
```

SAP ASE expects the asterisk to resolve to a single column, and generates an error when it encounters more than one column after it converts the asterisk.

Examples

Examples of using an asterisk in nested `select` statements.

This example deletes any discount from `stores` that have no sales, or that have discounts greater than 10:

```
create view store_with_nosales(stor_id)
as
select stores.stor_id
from stores left join sales
     on stores.stor_id = sales.stor_id
where sales.stor_id IS NULL
go
delete from discounts
where (stor_id in (select *
          from store_with_nosales)
     or discount > 10.0)
go
```

This example returns an error because there is more than one column in the join between `stores` and `sales`:

```
create view store_with_nosales(stor_id)
as
select stores.stor_id
from stores left join sales
     on stores.stor_id = sales.stor_id
where (stor_id in (select *
       from stores left join sales
            on stores.stor_id = sales.stor_id
        where sales.stor_id IS NULL)
  or discount > 10.0)
go
delete from discounts
where (stor_id in (select *
          from store_with_nosales)
     or discount > 10.0)
go
```

```
Msg 299, Level 16, State 1:
Line 1:
The symbol '*' can only be used for a subquery select
list when the subquery is introduced with EXISTS or NOT
EXISTS or the subquery references a single table and
column.
```

9.6 Subqueries in update, delete, and insert Statements

You can nest subqueries in `update`, `delete`, and `insert` statements as well as in `select` statements.

> **Note**
>
> Running the sample queries in this section changes the `pubs2` database. If you require the original `pubs2` database after you have run these queries, ask a system administrator to reload the `pubs2` database.

The following query doubles the price of all books published by New Age Books. The statement updates the `titles` table; its subquery references the `publishers` table.

```
update titles
set price = price * 2
where pub_id in
   (select pub_id
    from publishers
    where pub_name = "New Age Books")
```

An equivalent `update` statement using a join is:

```
update titles
set price = price * 2
from titles, publishers
where titles.pub_id = publishers.pub_id
and pub_name = "New Age Books"
```

Remove all records of sales of business books with this nested `select` statement:

```
delete salesdetail
where title_id in
   (select title_id
    from titles
    where type = "business")
```

An equivalent `delete` statement using a join is:

```
delete salesdetail
from salesdetail, titles
where salesdetail.title_id = titles.title_id
and type = "business"
```

9.7 Subqueries in Conditional Statements

You can use subqueries in conditional statements.

Rewrite the subquery that removes all records of sales of business books, as shown in the next example, to check for the records before deleting them:

```
if exists (select title_id
    from titles
```

```
    where type = "business")
begin
    delete salesdetail
    where title_id in
      (select title_id
       from titles
       where type = "business")
end
```

9.8 Subqueries Instead of Expressions

In Transact-SQL, you can substitute a subquery almost anywhere you can use an expression in a `select`, `update`, `insert`, or `delete` statement.

You cannot use a subquery in an `order by` list, or as an expression in the `values` list in an `insert` statement.

The following statement shows how to find the titles and types of books that have been written by authors living in California and that are also published there:

```
select title, type
from titles
where title in
   (select title
    from titles, titleauthor, authors
    where titles.title_id = titleauthor.title_id
    and titleauthor.au_id = authors.au_id
    and authors.state = "CA")
and title in
   (select title
    from titles, publishers
    where titles.pub_id = publishers.pub_id
    and publishers.state = "CA")
```

```
title                                  type
-----------------------------------    ----------
The Busy Executive's Database Guide    business
Cooking with Computers:
     Surreptitious Balance Sheets      business
Straight Talk About Computers          business
But Is It User Friendly?               popular_comp
Secrets of Silicon Valley              popular_comp
Net Etiquette                          popular_comp

(6 rows affected)
```

The following statement selects the book titles that have had more than 5000 copies sold, lists their prices, and the price of the most expensive book:

```
select title, price,
    (select max(price) from titles)
    from titles
    where total_sales > 5000
```

```
title                                 price
-----------------------------------  -----  ------
You Can Combat Computer Stress!       2.99   22.95
The Gourmet Microwave                 2.99   22.95
```

```
But Is It User Friendly?             22.95   22.95
Fifty Years in Buckingham Palace
    Kitchens                         11.95   22.95

(4 rows affected)
```

9.9 Types of Subqueries

There are two basic types of subqueries: expression and quantified predicate.

- Expression subqueries are introduced with an unmodified comparison operator, must return a single value, and can be used almost anywhere an expression is allowed in SQL.
- Quantified predicate subqueries operate on lists that are introduced with `in` or with a comparison operator that is modified by `any` or `all`. Quantified predicate subqueries return zero or more values. This type is also used as an existence test (which checks whether a subquery produces any rows), introduced with `exists`.

Subqueries of either type are either noncorrelated or correlated (repeating).

- A noncorrelated subquery can be evaluated as if it were an independent query. Conceptually, the results of the subquery are substituted in the main statement, or outer query. This is not how SAP ASE actually processes statements with subqueries. Noncorrelated subqueries can alternatively be stated as joins and are processed as joins by SAP ASE.
- A correlated subquery cannot be evaluated as an independent query, but can reference columns in a table listed in the `from` list of the outer query.

9.9.1 Expression Subqueries

Expression subqueries include subqueries in a `select` list (introduced with `in`) and in a `where` or `having` clause connected by a comparison operator (=, !=, > , > =, <, <=)

This is the general form of expression subqueries:

[Start of `select`, `insert`, `update`, `delete` statement or subquery]

```
where <expression> <comparison_operator> (<subquery>)
```

[End of `select`, `insert`, `update`, `delete` statement or subquery]

An expression consists of a subquery or any combination of column names, constants, and functions connected by arithmetic or bitwise operators.

The `<comparison_operator>` is one of:

Operator	Meaning
=	Equal to

Operator	Meaning
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
!=	Not equal to
<>	Not equal to
!>	Not greater than
!<	Not less than

If you use a column name in the `where` or `having` clause of the outer statement, make sure a column name in the `<subquery_select_list>` is join-compatible with it.

A subquery that is introduced with an unmodified comparison operator (that is, a comparison operator that is not followed by `any` or `all`) must resolve to a single value. If such a subquery returns more than one value, SAP ASE returns an error message.

For example, suppose that each publisher is located in only one city. To find the names of authors who live in the city where Algodata Infosystems is located, write a statement with a subquery that is introduced with the comparison operator =:

```
select au_lname, au_fname
from authors
where city =
   (select city
    from publishers
    where pub_name = "Algodata Infosystems")
```

```
au_lname       au_fname
-------------- --------------
Carson         Cheryl
Bennet         Abraham
```

9.9.1.1 Use Scalar Aggregate Functions to Guarantee a Single Value

Subqueries that are introduced with unmodified comparison operators often include scalar aggregate functions, which return a single value.

For example, to find the names of books that are priced higher than the current minimum price:

```
select title
from titles
```

```
where price >
   (select min(price)
    from titles)
```

```
title
--------------------------------------------------
The Busy Executive's Database Guide
Cooking with Computers: Surreptitious Balance
     Sheets
Straight Talk About Computers
Silicon Valley Gastronomic Treats
But Is It User Friendly?
Secrets of Silicon Valley
Computer Phobic and Non-Phobic Individuals:
     Behavior Variations
Is Anger the Enemy?
Life Without Fear
Prolonged Data Deprivation: Four Case Studies
Emotional Security: A New Algorithm
Onions, Leeks, and Garlic: Cooking Secrets of the
     Mediterranean
Fifty Years in Buckingham Palace Kitchens
Sushi, Anyone?
```

9.9.1.2 Use group by and having in Expression Subqueries

Because subqueries that are introduced by unmodified comparison operators must return a single value, they cannot include `group by` and `having` clauses unless you know that the `group by` and `having` clauses will return a single value.

For example, this query finds the books that are priced higher than the lowest priced book in the `trad_cook` category:

```
select title
from titles
where price >
   (select min(price)
    from titles
    group by type
    having type = "trad_cook")
```

9.9.1.3 Use distinct with Expression Subqueries

Subqueries that are introduced with unmodified comparison operators often include the `distinct` keyword to ensure the return of a single value.

For example, without `distinct`, this subquery fails because it returns more than one value:

```
select pub_name from publishers
    where pub_id =
        (select distinct pub_id
        from titles
        where pub_id  = publishers.pub_id)
```

9.9.2 Quantified Predicate Subqueries

Quantified predicate subqueries, which return a list of zero or more values, are subqueries in a `where` or `having` clause that are connected by `any`, `all`, `in`, or `exists`. The `any` or `all` subquery operators modify comparison operators.

There are three types of quantified predicate subqueries:

- `any/all` subqueries. Subqueries introduced with a modified comparison operator, which may include a `group by` or `having` clause, take this general form:
  [Start of `select`, `insert`, `update`, `delete` statement; or subquery]

  ```
  where <expression> <comparison_operator> [any | all]
      (<subquery>)
  ```

  [End of `select`, `insert`, `update`, `delete` statement; or subquery]
- `in/not in` subqueries. Subqueries introduced with `in` or `not in` take this general form:
  [Start of `select`, `insert`, `update`, `delete` statement; or subquery]

  ```
  where <expression> [not] in (<subquery>)
  ```

  [End of `select`, `insert`, `update`, `delete` statement; or subquery]
- `exists/not exists` subqueries. Subqueries introduced by `exists` or `not exists` are existence tests which take this general form:
  [Start of `select`, `insert`, `update`, `delete` statement; or subquery]

  ```
  where [not] exists (<subquery>)
  ```

  [End of `select`, `insert`, `update`, `delete` statement; or subquery]

Although SAP ASE allows the keyword `distinct` in quantified predicate subqueries, it always processes the subquery as if `distinct` were not included.

9.9.2.1 Subqueries with any and all

The keywords `all` and `any` can modify comparison operators that introduce a subquery.

When `any` is used with <, >, or = with a subquery, it returns results when any value retrieved in the subquery matches the value in the `where` or `having` clause of the outer statement.

When `all` is used with < or > in a subquery, it returns results when all values retrieved in the subquery match the value in the `where` or `having` clause of the outer statement.

The syntax for `any` and `all` is:

```
{where | having} [not]
    <expression> <comparison_operator> {any | all} (<subquery>)
```

Using the > comparison operator as an example:

- `> all` means greater than every value, or greater than the maximum value. For example, `> all` (1, 2, 3) means greater than 3.

- `> any` means greater than at least one value, or greater than the minimum value. Therefore, `> any` (1, 2, 3) means greater than 1.

If you introduce a subquery with `all` and a comparison operator does not return any values, the entire query fails.

`all` and `any` can be tricky. For example, you might ask "Which books commanded an advance greater than any book published by New Age Books?"

You can paraphrase this question to make its SQL "translation" more clear: "Which books commanded an advance greater than the largest advance paid by New Age Books?" The `all` keyword, *not* the `any` keyword, is required here:

```
select title
from titles
where advance > all
   (select advance
    from publishers, titles
    where titles.pub_id = publishers.pub_id
    and pub_name = "New Age Books")
```

```
title
----------------------------------------
The Gourmet Microwave
```

For each title, the outer query gets the titles and advances from the `titles` table, and it compares these to the advance amounts paid by New Age Books returned from the subquery. The outer query looks at the largest value in the list and determines whether the title being considered has commanded an even greater advance.

9.9.2.1.1 > all Means Greater Than All Values

The `> all` operator means that, for a row to satisfy the condition in the outer query, the value in the column that introduces the subquery must be greater than each of the values returned by the subquery.

For example, to find the books that are priced higher than the highest-priced book in the `mod_cook` category:

```
select title from titles where price > all
    (select price from titles
    where type = "mod_cook")
```

```
title
--------------------------------------------------
But Is It User Friendly?
Secrets of Silicon Valley
Computer Phobic and Non-Phobic Individuals:
    Behavior Variations
Onions, Leeks, and Garlic: Cooking Secrets of
    the Mediterranean

(4 rows affected)
```

However, if the set returned by the inner query contains a NULL, the query returns 0 rows. This is because NULL stands for "value unknown," and it is impossible to tell whether the value you are comparing is greater than an unknown value.

For example, try to find the books that are priced higher than the highest-priced book in the `popular_comp` category:

```
select title from titles where price > all
    (select price from titles
    where type = "popular_comp")
```

```
title
--------------------------------------------------

(0 rows affected)
```

No rows are returned because the subquery finds that one of the books, *Net Etiquette*, has a null price.

9.9.2.1.2 = all Means Equal to Every Value

The `= all` operator means that for a row to satisfy the outer query the value in the column that introduces the subquery must be the same as each value in the list of values returned by the subquery.

For example, the following query identifies the authors who live in the same city at the postal code:

```
select au_fname, au_lname, city
from authors
where city = all
     (select city
     from authors
     where postalcode like "946%")
```

9.9.2.1.3 > any Means Greater Than at Least One Value

`> any` means that, for a row to satisfy the outer query, the value in the column that introduces the subquery must be greater than at least one of the values in the list returned by the subquery.

The following example is introduced with a comparison operator modified by `any`. It finds each title that has an advance larger than any advance amount paid by New Age Books.

```
select title
from titles
where advance > any
   (select advance
    from titles, publishers
    where titles.pub_id = publishers.pub_id
    and pub_name = "New Age Books")
```

```
title
--------------------------------------------------
The Busy Executive’s Database Guide
Cooking with Computers: Surreptitious Balance
    Sheets
You Can Combat Computer Stress!
Straight Talk About Computers
The Gourmet Microwave
```

```
But Is It User Friendly?
Secrets of Silicon Valley
Computer Phobic and Non-Phobic Individuals:
    Behavior Variations
Is Anger the Enemy?
Life Without Fear
Emotional Security: A New Algorithm
Onions, Leeks, and Garlic: Cooking Secrets of
    the Mediterranean
Fifty Years in Buckingham Palace Kitchens
Sushi, Anyone?
```

For each title selected by the outer query, the inner query finds a list of advance amounts paid by New Age Books. The outer query looks at all the values in the list and determines whether the title being considered has commanded an advance that is larger than any of those values. In other words, this example finds titles with advances as large as or larger than the lowest value paid by New Age Books.

If the subquery does not return any values, the entire query fails.

9.9.2.1.4 = any Means Equal to Some Value

The `= any` operator is an existence check; it is equivalent to `in`.

For example, to find authors that live in the same city as any publisher, you can use either `= any` or `in`:

```
select au_lname, au_fname
from authors
where city = any
   (select city
    from publishers)
select au_lname, au_fname
from authors
where city in
   (select city
    from publishers)
```

```
au_lname       au_fname
-------------- --------------
Carson         Cheryl
Bennet         Abraham
```

However, the `!= any` operator is different from `not in`. The `!= any` operator means "not = a *or* not = b *or* not = c"; `not in` means "not = a *and* not = b *and* not = c".

For example, to find the authors who live in a city where no publisher is located:

```
select au_lname, au_fname
from authors
where city != any
   (select city
    from publishers)
```

The results include all 23 authors. This is because every author lives in some city where no publisher is located, and each author lives in only one city.

The inner query finds all the cities in which publishers are located, and then, for each city, the outer query finds the authors who do not live there.

Here is what happens when you substitute `not in` in the same query:

```
select au_lname, au_fname
from authors
where city not in
   (select city
    from publishers)
```

```
au_lname                au_fname
--------------          ------------
White                   Johnson
Green                   Marjorie
O'Leary                 Michael
Straight                Dick
Smith                   Meander
Dull                    Ann
Gringlesby              Burt
Locksley                Chastity
Greene                  Morningstar
Blotchet-Halls          Reginald
Yokomoto                Akiko
del Castillo            Innes
DeFrance                Michel
Stringer                Dirk
MacFeather              Stearns
Karsen                  Livia
Panteley                Sylvia
Hunter                  Sheryl
McBadden                Heather
Ringer                  Anne
Ringer                  Albert
```

These are the results you want. They include all the authors except Cheryl Carson and Abraham Bennet, who live in Berkeley, where Algodata Infosystems is located.

You get the same results if you use `!=all`, which is equivalent to `not in`:

```
select au_lname, au_fname
from authors
where city != all
   (select city
    from publishers)
```

9.9.3 Subqueries Used with in

Subqueries that are introduced with the keyword `in` return a list of zero or more results.

For example, this query finds the names of the publishers who have published business books:

```
select pub_name
from publishers
where pub_id in
   (select pub_id
    from titles
    where type = "business")
```

```
pub_name
----------------------------------------
New Age Books
```

```
Algodata Infosystems
```

This statement is evaluated in two steps. The inner query returns the identification numbers of the publishers who have published business books, 1389 and 0736. These values are then substituted in the outer query, which finds the names that go with the identification numbers in the `publishers` table. The query looks like this:

```
select pub_name
from publishers
where pub_id in ("1389", "0736")
```

Another way to formulate this query using a subquery is:

```
select pub_name
from publishers
where "business" in
   (select type
    from titles
    where pub_id = publishers.pub_id)
```

The expression following the `where` keyword in the outer query can be a constant as well as a column name. You can use other types of expressions, such as combinations of constants and column names.

The preceding queries, like many other subqueries, can be alternatively formulated as a join query:

```
select distinct pub_name
from publishers, titles
where publishers.pub_id = titles.pub_id
and type = "business"
```

Both this query and the subquery versions find publishers who have published business books. All are equally correct and produce the same results, though you may need to use the `distinct` keyword to eliminate duplicates.

However, one advantage of using a join query rather than a subquery is that a join query shows columns from more than one table in the result. For example, to include the titles of the business books in the result, use the join version:

```
select pub_name, title
from publishers, titles
where publishers.pub_id = titles.pub_id
and type = "business"
```

```
pub_name               title
--------------------   ----------------------------------------
Algodata Infosystems   The Busy Executive’s Database Guide
Algodata Infosystems   Cooking with Computers: Surreptitious
                       Balance Sheets
New Age Books          You Can Combat Computer Stress!
Algodata Infosystems   Straight Talk About Computers
```

Here is another example of a statement that you can formulate with either a subquery or a join query: “Find the names of all second authors who live in California and receive less than 30 percent of the royalties on a book.” Using a subquery, the statement is:

```
select au_lname, au_fname
from authors
where state = "CA"
```

```
and au_id in
   (select au_id
    from titleauthor
    where royaltyper < 30
    and au_ord = 2)
```

```
au_lname                 au_fname
------------------------ ------------
MacFeather               Stearns
```

The outer query produces a list of the 15 authors who live in California. The inner query is then evaluated, producing a list of the IDs of the authors who meet the qualifications.

More than one condition can be included in the `where` clause of both the inner and the outer query.

Using a join, the query is expressed like this:

```
select au_lname, au_fname
from authors, titleauthor
where state = "CA"
  and authors.au_id = titleauthor.au_id
  and royaltyper < 30
  and au_ord = 2
```

A join can always be expressed as a subquery. A subquery can often be expressed as a join.

9.9.4 Subqueries Used with not in

Subqueries that are introduced with the keyword phrase `not in` also return a list of values that are zero (0) and greater.

`not in` means "not = a *and* not = b *and* not = c."

This query finds the names of the publishers who have *not* published business books:

```
select pub_name from publishers
where pub_id not in
   (select pub_id
    from titles
    where type = "business")
```

```
pub_name
----------------------------------------
Binnet & Hardley
```

The query is the same as the previous one except that `not in` is substituted for `in`. However, you cannot convert this statement to a join; the "not equal" join finds the names of publishers who have published *some* book that is not a business book.

Related Information

Subqueries Used with in [page 274]

Joins: Retrieve Data from Several Tables [page 320]

9.9.5 Subqueries Using not in with NULL

A subquery that uses `not in` returns a set of values for each row in the outer query.

If the value in the outer query is not in the set returned by the inner query, the `not in` evaluates to TRUE, and the outer query puts the record being considered in the results.

However, if the set returned by the inner query contains no matching value, but it does contain a NULL, the `not in` returns UNKNOWN. This is because NULL stands for "value unknown," and it is impossible to tell whether the value you are looking for is in a set containing an unknown value. The outer query discards the row. For example:

```
select pub_name
    from publishers
    where $100.00 not in
        (select price
         from titles
         where titles.pub_id = publishers.pub_id)
```

```
pub_name
------
New Age Books
```

New Age Books is the only publisher that does not publish any books that cost $100. Binnet & Handley and Algodata Infosystems were not included in the query results because each publishes a book for which the price is undecided.

9.9.6 Subqueries Used with exists

Use the `exists` keyword with a subquery to test for the existence of some result from the subquery.

The syntax is:

```
{where | having} [not] exists (<subquery>)
```

That is, the `where` clause of the outer query tests for the existence of the rows returned by the subquery. The subquery does not actually produce any data, but returns a value of TRUE or FALSE.

For example, this query finds the names of all the publishers who publish business books:

```
select pub_name
from publishers
where exists
   (select *
    from titles
    where pub_id = publishers.pub_id
    and type = "business")
```

```
pub_name
```

```
----------------------------------------
New Age Books
Algodata Infosystems
```

To conceptualize the resolution of this query, consider each publisher's name in turn. Does this value cause the subquery to return at least one row? In other words, does it cause the existence test to evaluate to TRUE?

In the results of the preceding query, the second publisher's name is Algodata Infosystems, which has an identification number of 1389. Are there any rows in the `titles` table for which the `pub_id` is 1389 and `type` is business? If so, "Algodata Infosystems" should be one of the values selected. The same process is repeated for each of the other publisher's names.

A subquery that is introduced with `exists` is different from other subqueries, in these ways:

- The keyword `exists` is not preceded by a column name, constant, or other expression.
- The subquery `exists` evaluates to TRUE or FALSE rather than returning any data.
- The select list of the subquery usually consists of the asterisk (*). You need not specify column names, since you are simply testing for the existence of rows that meet the conditions specified in the subquery. Otherwise, the select list rules for a subquery introduced with `exists` are identical to those for a standard select list.

The `exists` keyword is very important, because there is often no alternative non-subquery formulation. In practice, a subquery introduced by `exists` is always a correlated subquery.

Although you cannot express some queries formulated with `exists` in any other way, you can express all queries that use `in` or a comparison operator modified by `any` or `all` with `exists`. Some examples of statements using `exists` and their equivalent alternatives follow.

Here are two ways to find authors that live in the same city as a publisher:

```
select au_lname, au_fname
from authors
where city = any
   (select city
    from publishers)
```

```
select au_lname, au_fname
from authors
where exists
   (select *
    from publishers
    where authors.city = publishers.city)
```

```
au_lname          au_fname
--------------    --------------
Carson            Cheryl
Bennet            Abraham
```

Here are two queries that find titles of books published by any publisher located in a city that begins with the letter "B":

```
select title
from titles
where exists
   (select *
    from publishers
    where pub_id = titles.pub_id
```

```
    and city like "B%")
```

```
select title
from titles
where pub_id in
   (select pub_id
    from publishers
    where city like "B%")
```

```
title
--------------------------------------------------
You Can Combat Computer Stress!
Is Anger the Enemy?
Life Without Fear
Prolonged Data Deprivation: Four Case Studies
Emotional Security: A New Algorithm
The Busy Executive's Database Guide
Cooking with Computers: Surreptitious Balance
    Sheets
Straight Talk About Computers
But Is It User Friendly?
Secrets of Silicon Valley
Net Etiquette
```

Related Information

Correlated Subqueries [page 281]

9.9.7 Subqueries Used with not exists

`not exists` is just like `exists` except that the `where` clause in which it is used is satisfied when no rows are returned by the subquery.

For example, to find the names of publishers who do *not* publish business books, the query is:

```
select pub_name
from publishers
where not exists
   (select *
    from titles
    where pub_id = publishers.pub_id
    and type = "business")
```

```
pub_name
----------------------------------------
Binnet & Hardley
```

This query finds the titles for which there have been no sales:

```
select title
from titles
where not exists
   (select title_id
    from salesdetail
```

```
    where title_id = titles.title_id)
```

```
title
----------------------------------------
The Psychology of Computer Cooking
Net Etiquette
```

9.9.8 Find Intersection and Difference with exists

You can use subqueries that are introduced with `exists` and `not exists` for two set theory operations: intersection and difference.

The intersection of two sets contains all elements that belong to both of the original sets. The difference contains the elements that belong only to the first set.

The intersection of `authors` and `publishers` over the `city` column is the set of cities in which both an author and a publisher are located:

```
select distinct city
from authors
where exists
  (select *
   from publishers
   where authors.city = publishers.city)
```

```
city
--------------------
Berkeley
```

The difference between `authors` and `publishers` over the `city` column is the set of cities where an author lives but no publisher is located, that is, all the cities except Berkeley:

```
select distinct city
from authors
where not exists
  (select *
   from publishers
   where authors.city = publishers.city)
```

```
city
--------------------
Gary
Covelo
Oakland
Lawrence
San Jose
Ann Arbor
Corvallis
Nashville
Palo Alto
Rockville
Vacaville
Menlo Park
Walnut Creek
San Francisco
Salt Lake City
```

9.9.9 Subqueries Using SQL Derived Tables

You can use SQL derived tables in subquery `from` clauses.

For example, this query finds the names of the publishers who have published business books:

```
select pub_name from publishers
   where "business" in
      (select type from
         (select type from titles, publishers
             where titles.pub_id = publishers.pub_id)
       dt_titles)
```

Here, `dt_titles` is the SQL derived table defined by the innermost `select` statement.

You can use SQL derived tables in the `from` clause of subqueries wherever subqueries are legal.

Related Information

SQL-Derived Tables [page 129]

9.10 Correlated Subqueries

In queries that include a repeating subquery, or correlated subquery, the subquery depends on the outer query for its values. The subquery is executed repeatedly, once for each row that is selected by the outer query.

This example finds the names of all authors who earn 100 percent royalty on a book:

```
select au_lname, au_fname
from authors
where 100 in
   (select royaltyper
    from titleauthor
    where au_id = authors.au_id)
```

```
au_lname        au_fname
--------------  ----------
White           Johnson
Green           Marjorie
Carson          Cheryl
Straight        Dick
Locksley        Chastity
Blotchet-Hall   Reginald
del Castillo    Innes
Panteley        Sylvia
Ringer          Albert
(9 rows affected)
```

The subquery in this statement cannot be resolved independently of the main query. It needs a value for `authors.au_id`, but this value is variable—it changes as SAP ASE examines different rows of the `authors` table.

This is how the preceding query is evaluated: Transact-SQL considers each row of the `authors` table for inclusion in the results, by substituting the value in each row in the inner query. For example, suppose Transact-SQL first examines the row for Johnson White. Then, `authors.au_id` takes the value "172-32-1176," which Transact-SQL substitutes for the inner query:

```
select royaltyper
from titleauthor
where au_id = "172-32-1176"
```

The result is 100, so the outer query evaluates to:

```
select au_lname, au_fname
from authors
where 100 in (100)
```

Since the `where` condition is true, the row for Johnson White is included in the results. If you go through the same procedure with the row for Abraham Bennet, you can see how that row is not included in the results.

This query uses a correlated variable as the outer member of a Transact-SQL outer join:

```
select t2.b1, (select t2.b2 from t1 where t2.b1 *= t1.a1) from t2
```

9.10.1 Correlated Subqueries with Correlation Names

You can use a correlated subquery to find the types of books that are published by more than one publisher.

```
select distinct t1.type
from titles t1
where t1.type in
   (select t2.type
    from titles t2
    where t1.pub_id != t2.pub_id)
```

```
type
--------------------
business
psychology
```

Correlation names are required in the following query to distinguish between the two roles in which the `titles` table appears. This nested query is equivalent to the self-join query:

```
select distinct t1.type
from titles t1, titles t2
where t1.type = t2.type
and t1.pub_id != t2.pub_id
```

9.10.2 Correlated Subqueries with Comparison Operators

Expression subqueries can be correlated subqueries.

For example, to find the sales of psychology books where the quantity is less than average for sales of that title:

```
select s1.ord_num, s1.title_id, s1.qty
from salesdetail s1
where title_id like "PS%"
and s1.qty <
   (select avg(s2.qty)
    from salesdetail s2
    where s2.title_id = s1.title_id)
```

```
ord_num            title_id     qty
------------------ --------     ---
91-A-7             PS3333        90
91-A-7             PS2106        30
55-V-7             PS2106        31
AX-532-FED-452-2Z7 PS7777       125
BA71224            PS7777       200
NB-3.142           PS2091       200
NB-3.142           PS7777       250
NB-3.142           PS3333       345
ZD-123-DFG-752-9G8 PS3333       750
91-A-7             PS7777       180
356921             PS3333       200
```

The outer query selects the rows of the `sales` table (or `s1` one by one). The subquery calculates the average quantity for each sale being considered for selection in the outer query. For each possible value of `s1`, Transact-SQL evaluates the subquery and includes the record being considered in the results, if the quantity is less than the calculated average.

Sometimes a correlated subquery mimics a `group by` statement. To find titles of books that have prices higher than average for books of the same type, the query is:

```
select t1.type, t1.title
from titles t1
where t1.price >
   (select avg(t2.price)
    from titles t2
    where t1.type = t2.type)
```

```
type         title
---------    -------------------------------------
business     The Busy Executive’s Database Guide
business     Straight Talk About Computers
mod_cook     Silicon Valley Gastronomic Treats
popular_comp But Is It User Friendly?
psychology   Computer Phobic and Non-Phobic
             Individuals: Behavior Variations
psychology   Prolonged Data Deprivation: Four Case
             Studies
trad_cook    Onions, Leeks, and Garlic: Cooking
             Secrets of the Mediterranean
```

For each possible value of `t1`, Transact-SQL evaluates the subquery and includes the row in the results if the price value of that row is greater than the calculated average. You need not group explicitly by type, because the rows for which the average price is calculated are restricted by the `where` clause in the subquery.

9.10.3 Correlated Subqueries in a having Clause

Quantified predicate subqueries can be correlated subqueries.

This example of a correlated subquery in the `having` clause of an outer query finds the types of books in which the maximum advance is more than twice the average within a given group:

```
select t1.type
from titles t1
group by t1.type
having max(t1.advance) >= any
   (select 2 * avg(t2.advance)
   from titles t2
   where t1.type = t2.type)
```

```
type
----------
mod_cook
```

The subquery above is evaluated once for each group that is defined in the outer query, that is, once for each type of book.

10 Aggregates, Grouping, and Sorting

You can use aggregate functions let you summarize the data retrieved in a query.

The aggregate functions are: `sum, avg, count, min, max, count_big, count(*),` and `count_big(*)..`

If your SAP ASE server is not case sensitive, see `group by and having clauses` and `compute clause` in the *Reference Manual: Commands* for examples on how case sensitivity affects the data returned by these clauses.

To find out the number of books sold in the `titles` table of the `pubs2` database, enter:

```
select sum(total_sales)
from titles
```

```
-------------
       97746
```

There is no column heading for the aggregate column in the example.

An aggregate function takes as an argument the column name on which values it operates. You can apply aggregate functions to all the rows in a table, to a subset of the table specified by a `where` clause, or to one or more groups of rows in the table. From each set of rows to which an aggregate function is applied, SAP ASE generates a single value.

Here is the syntax of the aggregate function:

```
<aggregate_function> ( [all | distinct] <expression>)
```

`expression` is usually a column name. However, it can also be a constant, a function, or any combination of column names, constants, and functions connected by arithmetic or bitwise operators. You can also use a `case` expression or subquery in an expression.

For example, with this statement, you can calculate the average price of all books if prices were doubled:

```
select avg(price * 2)
from titles
```

```
-------------
        29.53

(1 row affected)
```

Use the optional keyword `distinct` with `sum`, `avg`, `count`, `min`, and `max` to eliminate duplicate values before the aggregate function is applied. `all`, which performs the operation on all rows, is the default.

This is the syntax of the aggregate functions and the results they produce:

Aggregate Function	Result
sum([all | distinct] <expression>)	Total of the (distinct) values in the expression
avg([all | distinct] <expression>)	Average of the (distinct) values in the expression
count([all | distinct] <expression>)	Number of (distinct) non-null values in the expression returned as an integer
count_big ([all | distinct] <expression>)	Number of (distinct) non-null values in the expression returned as a bigint
count(*)	Number of selected rows as an integer
count_big(*)	Number of selected rows as a bigint
max(<expression>)	Highest value in the expression
min(<expression>)	Lowest value in the expression

You can use the aggregate functions in a select list, as shown in the previous example, or in the `having` clause.

You cannot use aggregate functions in a `where` clause, but most `select` statements with an aggregate function in the select list include a `where` clause that restricts the rows to which the aggregate is applied. In the examples given earlier in this section, each aggregate function produced a single summary value for the entire table.

If a `select` statement includes a `where` clause, but not a `group by` clause, an aggregate function produces a single value for the subset of rows, called a scalar aggregate. However, a `select` statement can also include a column in its select list (a Transact-SQL extension), that repeats the single value for each row in the result table.

This query returns the average advance and the sum of sales for only business books, preceded by a column named “advance and sales:”

```
select "advance and sales", avg(advance), sum(total_sales)
from titles
where type = "business"
```

```
-----------------  ----------------- -----------
advance and sales          6,281.25       30788
(1 row affected)
```

10.1 Aggregate Functions and Datatypes

You can use the aggregate functions with any type of column, with a few exceptions.

The exceptions are:

- You can use `sum` and `avg` with numeric columns only—`bigint`, `int`, `smallint`, `tinyint`, `unsigned bigint`, `unsigned int`, `unsigned smallint`, `decimal`, `numeric`, `float`, and `money`.
- You cannot use `min` and `max` with `bit` datatypes.
- You cannot use aggregate functions other than `count(*)` and `count_big(*)` with `text` and `image` datatypes.

For example, you can use `min` (minimum) to find the lowest value—the one closest to the beginning of the alphabet—in a character type column:

```
select min(au_lname)
from authors
```

```
--------------------------
Bennet

(1 row affected)
```

However, you cannot average the contents of a text column:

```
select avg(au_lname)
from authors
```

```
Msg 257, Level 16, State 1:
--------------------------
(1 row affected)
Line 1:
Implicit conversion from datatype 'VARCHAR' to 'INT' is not allowed.  Use the
CONVERT function to run this query.
```

10.2 count versus count (*)

`count` finds the number of non-null values in an expression; `count(*)` finds the total number of rows in a table.

This statement finds the total number of books:

```
select count(*)
from titles
```

```
------------------
                18

(1 row affected)
```

`count(*)` returns the number of rows in the specified table without eliminating duplicates. It counts each row, including those containing null values.

Like other aggregate functions, you can combine `count(*)` with other aggregates in the select list, with `where` clauses, and so on:

```
select count(*), avg(price) from titles
```

```
where advance > 1000
```

```
---------- ---------
        15     14.42

(1 row affected)
```

10.3 Aggregate Functions with distinct

You can use the optional keyword `distinct` only with `sum`, `avg`, `count_big`, and `count`. When you use `distinct`, SAP ASE eliminates duplicate values before performing calculations.

If you use `distinct`, you cannot include an arithmetic expression in the argument. The argument can use only a column name. `distinct` appears inside the parentheses and before the column name. For example, to find the number of different cities in which there are authors, enter:

```
select count(distinct city)
from authors
```

```
-------------
           16

(1 row affected)
```

For an accurate calculation of the average price of all business books, omit `distinct`. The following statement returns the average price of all business books:

```
select avg(price)
from titles
where type = "business"
```

```
-------------
        13.73

(1 row affected)
```

However, if two or more books have the same price and you use `distinct`, the shared price is included only once in the calculation:

```
select avg(distinct price)
from titles
where type = "business"
```

```
-------------
        11.64

(1 row affected)
```

10.4 Null Values and the Aggregate Functions

SAP ASE ignores any null values in the column on which an aggregate function is operating for the purposes of the function (except `count`(*) and `count_big(*)`, which includes them).

If you have set `ansinull` to on, SAP ASE returns an error message whenever a null value is ignored. See the *Reference Manual: Commands*.

For example, the `count` of advances in the `titles` table is not the same as the `count` of title names, because of the null values in the `advance` column:

```
select count(advance)
from titles
```

```
-------------
           16

(1 row affected)
```

```
select count(title)
from titles
```

```
-------------
           18

(1 row affected)
```

If all the values in a column are null, `count` returns 0. If no rows meet the conditions specified in the `where` clause, `count` returns 0. The other functions all return NULL. Here are examples:

```
select count(distinct title)
from titles
where type = "poetry"
```

```
-------------
            0

(1 row affected)
```

```
select avg(advance)
from titles
where type = "poetry"
```

```
-------------
         NULL

 (1 row affected)
```

10.5 Using Statistical Aggregates

SAP ASE supports statistical aggregate functions, which permit statistical analysis of numeric data. Statistical aggregate functions include `stddev`, `stddev_samp`, `stddev_pop`, `variance`, `var_samp`, and `var_pop`.

Simple aggregate functions, such as `sum`, `avg`, `max`, `min`, `count_big`, and `count` are allowed only in the select list and in the `having` and `order by` clauses as well as the `compute` clause of a `select` statement.

These functions, including `stddev` and `variance`, are true aggregate functions in that they can compute values for a group of rows as determined by the query's `group by` clause. As with other basic aggregate functions such as `max` or `min`, their calculations ignore null values in the input. All variance and standard deviation computation uses IEEE double-precision floating-point standard.

If the input to any variance or standard deviation function is an empty set, each aggregate function returns a null value. If the input to any variance or standard deviation function is a single value, then each function returns 0 as its result.

The statistical aggregate functions (and their aliases) are:

- `stddev_pop` (also `stdevp`) – standard deviation of a population. Computes the population standard deviation of the provided value expression evaluated for each row of the group (if `distinct` was specified, then each row that remains after duplicates have been eliminated), defined as the square root of the population variance.
- `stddev_samp` (also `stdev`, `stddev`) – standard deviation of a sample. Computes the population standard deviation of the provided value expression evaluated for each row of the group (if `distinct` was specified, then each row that remains after duplicates have been eliminated), defined as the square root of the sample variance.
- `var_pop` (also `varp`) – variance of a population. Computes the population variance of value expression evaluated for each row of the group (if `distinct` was specified, then each row that remains after duplicates have been eliminated), defined as the sum of squares of the difference of value expression from the mean of value expression, divided by the number of rows in the group.
- `var_samp` (also `var`, `variance`) – variance of a sample. Computes the sample variance of value expression evaluated for each row of the group (if `distinct` was specified, then each row that remains after duplicates have been eliminated), defined as the sum of squares of the difference from the mean of the value expression, divided by one less than the number of rows in the group.

10.6 Organize Query Results into Groups: the group by Clause

The `group by` clause divides the output of a query into groups. You can group by one or more column names, or by the results of computed columns using numeric datatypes in an expression.

When used with aggregates, `group by` retrieves the calculations in each subgroup, and may return multiple rows.

The maximum number of group by columns (or expressions) is not explicitly limited. The only limit of group by results is that the width of the group by columns plus the aggregate results cannot be larger than 64K.

> **Note**
>
> You cannot use `group by` with columns of `text`, `unitext`, or `image` datatypes.

While you can use `group by` without aggregates, such a construction has limited functionality and may produce confusing results. The following example groups results by title type:

```
select type, advance
     from titles
group by type
```

```
type              advance
------------         ---------
popular comp          7,000.00
popular comp          8,000.00
popular comp              NULL
business              5,000.00
business              5,000.00
business             10,125.00
mod_cook                  0.00
mod_cook             15,000.00
trad_cook             7,000.00
trad_cook             4,000.00
trad_cook             8,000.00
UNDECIDED                 NULL
psychology            7,000.00
psychology            2,275.00
psychology            6,000.00
psychology            2,000.00
psychology            4,000.00

(18 rows affected)
```

With an aggregate for the `advance` column, the query returns the sum for each group:

```
select type, sum(advance)
   from titles
group by type
```

```
type
------------ ------------------------
popular_comp                15,000.00
business                    25,125.00
mod_cook                    15,000.00
trad_cook                   19,000.00
UNDECIDED                        NULL
psychology                  21,275.00
(6 rows affected)
```

The summary values in a `group by` clause using aggregates are called vector aggregates, as opposed to scalar aggregates, which result when only one row is returned.

See the *Reference Manual:Commands*.

10.6.1 group by and SQL Standards

The SQL standards for `group by` are more restrictive than the default SAP ASE standard.

The SQL standard requires that:

- The columns in a select list to be in the `group by` expression or to be arguments of aggregate functions.
- A `group by` expression can contain only column names in the select list, but not those used only as arguments for vector aggregates.

Several Transact-SQL extensions (described in the following sections) relax these restrictions. However, complex result sets may be more difficult to understand. If you set the `fipsflagger` option as follows, you will receive a warning message stating that Transact-SQL extensions are used:

```
set fipsflagger on
```

For more information about the `fipsflagger` option, see the `set` command in the *Reference Manual: Commands*.

10.6.2 Nest Groups with group by

Nest groups by including more than one column in the `group by` clause. Once the sets are established with `group by`, the aggregates are applied.

This statement finds the average price and the sum of book sales, grouped first by publisher identification number, then by type:

```
select pub_id, type, avg(price), sum(total_sales)
from titles
group by pub_id, type
pub_id type
------ ------------ ------ -------
0736 business 2.99 18,722
```

```
0736 psychology 11.48 9,564
0877 UNDECIDED NULL NULL
0877 mod_cook 11.49 24,278
0877 psychology 21.59 375
0877 trad_cook 15.96 19,566
1389 business 17.31 12,066
1389 popular_comp 21.48 12,875(8 rows affected)
```

You can nest groups within groups. The maximum number of `group by` columns (or expressions) is not explicitly limited.

10.6.3 Reference Other Columns in Queries Using group by

SQL standards state that the `group by` clause must contain items from the `select` list. However, Transact-SQL allows you to specify any valid column name in either the `group by` or `select` list, whether they employ aggregates or not.

SAP ASE lifts restrictions on what you can include or omit in the `select` list of a query that includes `group by`:

- The columns in the select list are not limited to the grouping columns and columns used with the vector aggregates.
- The columns specified by `group by` are not limited to nonaggregate columns in the select list.

A vector aggregate must be accompanied by a `group by` clause. The SQL standards require nonaggregate columns in the `select` list to match the `group by` columns. However, the first bulleted item above allows you to specify additional, extended columns in the `select` list of the query.

For example, many versions of SQL do not allow the inclusion of the extended `title_id` column in the `select` list, but Transact-SQL: allows this:

```
select type, title_id, avg(price), avg(advance)
from titles
group by type
type title_id
```

```
------------ -------- ----- -------
business BU1032 13.73 6,281.25
business BU1111 13.73 6,281.25
business BU2075 13.73 6,281.25
business BU7832 13.73 6,281.25
mod_cook MC2222 11.49 7,500.00
mod_cook MC3021 11.49 7,500.00
UNDECIDED MC3026 NULL NULL
popular_comp PC1035 21.48 7,500.00
popular_comp PC8888 21.48 7,500.00
popular_comp PC9999 21.48 7,500.00
psychology PS1372 13.50 4,255.00
psychology PS2091 13.50 4,255.00
psychology PS2106 13.50 4,255.00
psychology PS3333 13.50 4,255.00
psychology PS7777 13.50 4,255.00
trad_cook TC3218 15.96 6,333.33
trad_cook TC4203 15.96 6,333.33
trad_cook TC7777 15.96 6,333.33
(18 rows affected)
```

The above example still aggregates the `price` and `advance` columns based on the `type` column, but its results also display the `title_id` for the books included in each group.

The second bullet item described above allows you to group columns that are not specified as columns in the select list of the query. These columns do not appear in the results, but the vector aggregates still compute their summary values. For example:

```
select state, count(au_id)
from authors
group by state, city
```

```
state
```

```
----- --------
CA 2
CA 1
CA 5
CA 5
CA 2
CA 1
CA 1
CA 1
CA 1
IN 1
KS 1
MD 1
MI 1
OR 1
TN 1
UT 2
(16 rows affected)
```

This example groups the vector aggregate results by both state and city, even though it does not display which city belongs to each group. Therefore, results are potentially misleading.

You may think the following query should produce similar results to the previous query, since only the vector aggregate seems to tally the number of each city for each row:

```
select state, count(au_id)
from authors
group by city
```

However, its results are much different. By not using `group by` with both the state and city columns, the query tallies the number of each city, but it displays the tally for each row of that city in authors rather than grouping them into one result row per city.

```
state
----- -----------
CA 1
CA 5
CA 2
CA 1
CA 5
KS 1
CA 2
CA 2
CA 1
CA 1
TN 1
OR 1
CA 1
MI 1
IN 1
CA 5
CA 5
CA 5
MD 1
CA 2
CA 1
UT 2
UT 2
(23 rows affected)
```

When you use the Transact-SQL extensions in complex queries that include the `where` clause or joins, the results may become even more difficult to understand. To avoid confusing or misleading results with group by, SAP suggests that you use the `fipsflagger` option to identify queries that contain Transact-SQL extensions.

Related Information

group by and SQL Standards [page 292]

10.6.4 Expressions and group by

Use `group by` for an expression that does not include aggregate functions.

For example:

```
select avg(total_sales), total_sales * price
from titles
group by total_sales * price
```

```
---------      -------------
     2045          22,392.75
     2032          40,619.68
     4072          81,399.28
     NULL               NULL
     4095          61,384.05
    18722          55,978.78
      375           7,856.25
    15096         180,397.20
     3876          46,318.20
      111             777.00
     3336          26,654.64
     4095          81,859.05
    22246          66,515.54
     8780         201,501.00
      375           8,096.25
     4095          81,900.00

(16 rows affected)
```

The expression "`total_sales * price`" is allowed.

You cannot use `group by` on a column heading, also known as an alias, although you can still use one in your select list. This statement produces an error message:

```
select Category = type, title_id, avg(price), avg(advance)
from titles
group by Category
------------------
Msg 207, Level 16, State 4:
Line 1:
Invalid column name 'Category'
Msg 207, Level 16, State 4:
Line 1:
Invalid column name 'Category'
```

The `group by` clause should be "`group by type,`" not "`group by Category`."

```
select Category = type, title_id, avg(price), avg(advance)
from titles
group by type
--------------
         21.48
         13.73
```

```
         11.49
         15.96
           NULL
13.50
(6 rows affected)
```

10.6.5 group by in Nested Aggregates

Use `group by` to nest a vector aggregate inside a scalar aggregate.

For example, to find the average price of all types of books using a nonnested aggregate, enter:

```
select avg(price)
from titles
group by type
```

```
---------------
 NULL
13.73
11.49
21.48
13.50
15.96

(6 rows affected)
```

Nesting the average price inside the `max` function produces the highest average price of a group of books, grouped by type:

```
select max(avg(price))
from titles
group by type
```

```
-------------
        21.48

(1 row affected)
```

By definition, the `group by` clause applies to the innermost aggregate—in this case, `avg`.

10.6.6 Null Values and group by

If the grouping column contains a null value, that row becomes its own group in the results. If the grouping column contains more than one null value, all null values form a single group.

This example uses `group by` and the `advance` column, which contains some null values:

```
select advance, avg(price * 2)
from titles
group by advance
```

```
advance
```

```
------------------  -----------------
             NULL                NULL
             0.00               39.98
          2000.00               39.98
          2275.00               21.90
          4000.00               19.94
          5000.00               34.62
          6000.00               14.00
          7000.00               43.66
          8000.00               34.99
         10125.00                5.98
         15000.00                5.98

(11 rows affected)
```

If you are using the `count(<column_name>)` aggregate function, grouping by a column that contains null values returns a count of zero for the grouping row, since `count(<column_name>)` does not include null values. In most cases, use `count(*)` instead. This example groups and counts on the `price` column from the `titles` table, which contains null values, and shows `count(*)` for comparison:

```
select price, count(price), count(*)
from titles
group by price
```

```
price
------------- ----- -----
        NULL     0     2
        2.99     2     2
        7.00     1     1
        7.99     1     1
       10.95     1     1
       11.95     2     2
       14.99     1     1
       19.99     4     4
       20.00     1     1
       20.95     1     1
       21.59     1     1
       22.95     1     1

(12 rows affected)
```

10.6.7 where Clause and group by

You can use a `where` clause in a statement with `group by`.

Rows that do not satisfy the conditions in the `where` clause are eliminated before any grouping is done:

```
select type, avg(price)
from titles
where advance > 5000
group by type
```

```
type
-------------    --------
business            2.99
mod_cook            2.99
popular_comp       21.48
psychology         14.30
```

```
trad_cook          17.97

(5 rows affected)
```

Only the rows with advances of more than $5000 are included in the groups that are used to produce the query results.

The way SAP ASE handles extra columns in the `select` list and the `where` clause may seem contradictory. For example:

```
select type, advance, avg(price)
from titles
where advance > 5000
group by type
```

```
type           advance
-------------  ---------  --------
business        5,000.00      2.99
business        5,000.00      2.99
business       10,125.00      2.99
business        5,000.00      2.99
mod_cook            0.00      2.99
mod_cook       15,000.00      2.99
popular_comp    7,000.00     21.48
popular_comp    8,000.00     21.48
popular_comp        NULL     21.48
psychology      7,000.00     14.30
psychology      2,275.00     14.30
psychology      6,000.00     14.30
psychology      2,000.00     14.30
psychology      4,000.00     14.30
trad_cook       7,000.00     17.97
trad_cook       4,000.00     17.97
trad_cook       8,000.00     17.97

(17 rows affected)
```

When you look at the results for the `advance` (extended) column, it may seem as though the query is ignoring the `where` clause. SAP ASE still computes the vector aggregate using only those rows that satisfy the `where` clause, but it also displays all rows for any extended columns that you include in the select list. To further restrict these rows from the results, use a `having` clause.

Related Information

Select Groups of Data: the having Clause [page 301]

10.6.8 group by and all

The keyword `all` in the `group by` clause is a Transact-SQL enhancement. It is meaningful only if the `select` statement in which it is used also includes a `where` clause.

If you use `all`, the query results include all the groups produced by the `group by` clause, even if some groups do not have any rows that meet the search conditions. Without `all`, a `select` statement that includes `group by` does not show groups for which no rows qualify.

Here is an example:

```
select type, avg(advance)
from titles
where advance > 1000 and advance < 10000
group by type
```

```
type
------------  ------------------------
business                     5,000.00
popular_comp                 7,500.00
psychology                   4,255.00
trad_cook                    6,333.33

(4 rows affected)
```

```
select type, avg(advance)
from titles
where advance > 1000 and advance < 10000
group by all type
```

```
type
------------  ------------------------
UNDECIDED                        NULL
business                     5,000.00
mod_cook                         NULL
popular_comp                 7,500.00
psychology                   4,255.00
trad_cook                    6,333.33

(6 rows affected)
```

The first statement produces groups only for those books that commanded advances of more than $1000 but less than $10,000. Since no modern cooking books have an advance within that range, there is no group in the results for the `mod_cook` type.

The second statement produces groups for all types, including modern cooking and "UNDECIDED," even though the modern cooking group does not include any rows that meet the qualification specified in the `where` clause. SAP ASE returns a NULL result for all groups that lack qualifying rows.

10.6.9 Aggregates Without group by

By definition, scalar aggregates apply to all rows in a table, producing a single value for the entire table for each function.

This Transact-SQL extension allows you to include extended columns with vector aggregates also allows you to include extended columns with scalar aggregates. For example, look at the `publishers` table:

```
pub_id pub_name             city           state
------ ------------------   -------------- -----
0736   New Age Books        Boston         MA
0877   Binnet & Hardley     Washington     DC
1389   Algodata Infosystems Berkeley       CA
```

It contains three rows. The following query produces a three-row scalar aggregate based on each row of the table:

```
select pub_id, count(pub_id)
from publishers
```

```
pub_id
---------- ---------
0736               3
0877               3
1389               3

(3 rows affected)
```

Because SAP ASE treats `publishers` as a single group, the scalar aggregate applies to the (single-group) table. The results display every row of the table for each column you include in the select list, in addition to the scalar aggregate.

The `where` clause behaves the same way for scalar aggregates as with vector aggregates. The `where` clause restricts the columns included in the aggregate summary values, but it does not affect the rows that appear in the results for each extended column you specify in the select list. For example:

```
select pub_id, count(pub_id)
from publishers
where pub_id < "1000"
```

```
pub_id
-------------- -----------
0736                    2
0877                    2
1389                    2

(3 rows affected)
```

Like the other Transact-SQL extensions to `group by`, this extension provides results that may be difficult to understand, especially for queries on large tables, or queries with multitable joins.

10.7 Select Groups of Data: the having Clause

Use the `having` clause to display or reject rows defined by the `group by` clause.

The `having` clause sets conditions for the `group by` clause in the same way `where` sets conditions for the `select` clause, except `where` cannot include aggregates, while `having` often does. This example is allowed:

```
select title_id
from titles
where title_id like "PS%"
having avg(price) > $2.0
```

But this example is not:

```
select title_id
from titles
where avg(price) > $20
--------------------
Msg 147, Level 15, State 1
Line 1:
An aggregate function may not appear in a WHERE clause unless it is in a
subquery that is in a HAVING clause, and the column being aggregated is in a
table named in a FROM clause outside of the subquery.
```

`having` clauses can reference any of the items that appear in the select list.

This statement is an example of a `having` clause with an aggregate function. It groups the rows in the `titles` table by type, but eliminates the groups that include only one book:

```
select type
from titles
group by type
having count(*) > 1
```

```
type
----------------
business
mod_cook
popular_comp
psychology
trad_cook

(5 rows affected)
```

Here is an example of a `having` clause without aggregates. It groups the `titles` table by type and returns only those types that start with the letter “p”:

```
select type
from titles
group by type
having type like "p%"
```

```
type
------------
popular_comp
psychology

(2 rows affected)
```

When you include more than one condition in the `having` clause, combine the conditions with `and`, `or`, or `not`. For example, to group the `titles` table by publisher, and to include only those publishers who have paid more than $15,000 in total advances, whose books average less than $18 in price, and for which the book identification numbers (`pub_id`) are greater than 0800, the statement is:

```
select pub_id, sum(advance), avg(price)
from titles
group by pub_id
having sum(advance) > 15000
  and avg(price) < 18
  and pub_id > "0800"
```

```
pub_id
------  ---------------- ----------------
0877          41,000.00            15.41

(1 row affected)
```

10.7.1 Interactions between having, group by, and where Clauses

When you include the `having`, `group by`, and `where` clauses in a query, the sequence in which each clause affects the rows determines the final results.

- The `where` clause excludes rows that do not meet its search conditions.
- The `group by` clause collects the remaining rows into one group for each unique value in the `group by` expression.
- Aggregate functions specified in the select list calculate summary values for each group.
- The `having` clause excludes rows from the final results that do not meet its search conditions.

The following query illustrates the use of `where`, `group by`, and `having` clauses in one `select` statement containing aggregates:

```
select stor_id, title_id, sum(qty)
from salesdetail
where title_id like "PS%"
group by stor_id, title_id
having sum(qty) > 200
```

```
stor_id  title_id
-------  --------  -----------
5023     PS1372            375
5023     PS2091          1,845
5023     PS3333          3,437
5023     PS7777          2,206
6380     PS7777            500
7067     PS3333            345
7067     PS7777            250

(7 rows affected)
```

The query executed in this order:

1. The `where` clause identified only rows with `title_id` beginning with “PS” (psychology books),

2. `group by` collected the rows by common `stor_id` and `title_id`,
3. The `sum` aggregate calculated the total number of books sold for each group, and
4. The `having` clause excluded from the final results the groups for which the book totals do not exceed 200 books.

All of the previous `having` examples in this section adhere to the SQL standards, which specify that columns in a `having` expression must have a single value, and must be in the select list or `group by` clause. However, the Transact-SQL extensions to `having` allow columns or expressions not in the select list and not in the `group by` clause.

The following example determines the average price for each title type, but excludes those types that do not have more than $10,000 in total sales, even though the `sum` aggregate does not appear in the results.

```
select type, avg(price)
from titles
group by type
having sum(total_sales) > 10000
```

```
type
------------  ----------
business           13.73
mod_cook           11.49
popular_comp       21.48
trad_cook          15.96

(4 rows affected)
```

The extension behaves as if the column or expression were part of the select list but not part of the results. If you include a nonaggregated column with `having`, but it is not part of the select list or the `group by` clause, the query produces results similar to the "extended" column extension. For example:

```
select type, avg(price)
from titles
group by type
having total_sales > 4000
```

```
type
------------  ----------
business           13.73
business           13.73
business           13.73
mod_cook           11.49
popular_comp       21.48
popular_comp       21.48
psychology         13.50
trad_cook          15.96
trad_cook          15.96

(9 rows affected)
```

Unlike an extended column, the `total_sales` column does not appear in the final results, yet the number of displayed rows for each type depends on the `total_sales` for each title. The query indicates that three business, one mod_cook, two popular_comp, one psychology, and two trad_cook titles exceed $4000 in total sales.

As mentioned earlier, the way SAP ASE handles extended columns may seem as if the query is ignoring the `where` clause in the final results. To make the `where` conditions affect the results for the extended column, repeat the conditions in the `having` clause. For example:

```
select type, advance, avg(price)
from titles
where advance > 5000
group by type
having advance > 5000
```

```
type           advance
-------------  ---------  --------
business       10,125.00      2.99
mod_cook       15,000.00      2.99
popular_comp    7,000.00     21.48
popular_comp    8,000.00     21.48
psychology      7,000.00     14.30
psychology      6,000.00     14.30
trad_cook       7,000.00     17.97
trad_cook       8,000.00     17.97

(8 rows affected)
```

10.7.2 having Without group by

A query with a `having` clause should also have a `group by` clause. If you omit `group by`, all the rows not excluded by the `where` clause return as a single group.

Because no grouping is performed between the `where` and `having` clauses, they cannot act independently of each other. `having` acts like `where` because it affects the rows in a single group rather than groups, except the `having` clause can still use aggregates.

This example uses the `having` clause averages the price, excludes from the results titles with advances greater than $4,000, and produces results where price is less than the average price:

```
select title_id, advance, price
from titles
where advance < 4000
having price > avg(price)
```

```
title_id        advance    price
-------------  ---------  --------
BU1032          5,000.00     19.99
BU7832          5,000.00     19.99
MC2222              0.00     19.99
PC1035          7,000.00     22.95
PC8888          8,000.00     20.00
PS1372          7,000.00     21.59
PS3333          2,000.00     19.99
TC3218          7,000.00     20.95

(8 rows affected)
```

You can also use the `having` clause with the Transact-SQL extension that allows you to omit the `group by` clause from a query that includes an aggregate in its select list. These scalar aggregate functions calculate values for the table as a single group, not for groups within the table.

In this example, the `group by` clause is omitted, which makes the aggregate function calculate a value for the entire table. The `having` clause excludes non-matching rows from the result group.

```
select pub_id, count(pub_id)
from publishers
having pub_id < "1000"
```

```
pub_id
------ ----------------
0736                  3
0877                  3

(2 rows affected)
```

10.8 Sort Query Results: the order by Clause

The `order by` clause allows you to sort query results by up to as many as 400 columns. Each sort is either ascending (`asc`) or descending (`desc`). If neither is specified, `asc` is the default.

The following query orders results by `pub_id`:

```
select pub_id, type, title_id
from titles
order by pub_id
```

```
pub_id  type            title_id
------  ------------    --------
0736    business        BU2075
0736    psychology      PS2091
0736    psychology      PS2106
0736    psychology      PS3333
0736    psychology      PS7777
0877    UNDECIDED       MC3026
0877    mod_cook        MC2222
0877    mod_cook        MC3021
0877    psychology      PS1372
0877    trad_cook       TC3218
0877    trad_cook       TC4203
0877    trad_cook       TC7777
1389    business        BU1032
1389    business        BU1111
1389    business        BU7832
1389    popular_comp    PC1035
1389    popular_comp    PC8888
1389    popular_comp    PC9999

(18 rows affected)
```

Multiple Columns

If you name more than one column in the `order by` clause, SAP ASE nests the sorts. The following statement sorts the rows in the `stores` table first by `stor_id` in descending order, then by `payterms` (in ascending

order, since `desc` is not specified), and finally by `country` (also ascending). SAP ASE sorts null values first within any group.

```
select stor_id, payterms, country
from stores
order by stor_id desc, payterms
```

```
stor_id payterms     country
------- ------------ ------------
8042    Net 30       USA
7896    Net 60       USA
7131    Net 60       USA
7067    Net 30       USA
7066    Net 30       USA
6380    Net 60       USA
5023    Net 60       USA
(7 rows affected)
```

Column Position Numbers

You can use the position number of a column in a select list instead of the column name. You can mix column names and select list numbers. Both of the following statements produce the same results as the preceding one.

```
select pub_id, type, title_id
from titles
order by 1 desc, 2, 3
```

```
select pub_id, type, title_id
from titles
order by 1 desc, type, 3
```

Most versions of SQL require that `order by` items appear in the select list, but Transact-SQL has no such restriction. You can order the results of the preceding query by `title`, although that column does not appear in the select list.

> **Note**
>
> You cannot use `order by` on `text`, `unitext`, or `image` columns.

Aggregate Functions

Aggregate functions are permitted in an `order by` clause, but they must follow a syntax that avoids ambiguity about which `order by` column is subject to the `union` expression. However, the name of columns in a union is derived from the first (leftmost) part of the union. This means that the `order by` clause uses only column names specified in the first part of the union.

For example, the following syntax works, because the column identified by the `order by` key is clearly specified:

```
select id, min(id) from tab
union
select id, max(id) from tab
ORDER BY 2
```

However, this example produces an error message:

```
select id+2 from sysobjects
union
select id+1 from sysobjects
order by id+1
------------
Msg 104, Level 15, State1:
Line 3:
Order-by items must appear in the select list if the statement contains set
operators.
```

If you rearrange the statement by trading the `union` sides, it executes correctly:

```
select id+1 from sysobjects
union
select id+2 from sysobjects
order by id+1
```

Null Values

With `order by`, null values come before all others.

Mixed-Case Data

The effects of an `order by` clause on mixed-case data depend on the sort order installed on your SAP ASE. The basic choices are binary, dictionary order, and case-insensitive. `sp_helpsort` displays the sort order for your server. See *System Administration Guide: Volume 1 > Configuring Character Sets, Sort Orders, and Languages*.

Limitations

SAP ASE does not allow subqueries or variables in the `order by` list.

You cannot use `order by` on `text`, `unitext`, or `image` columns.

10.8.1 order by and group by

You can use an `order by` clause to order the results of a `group by` in a particular way.

Put the `order by` clause after the `group by` clause. For example, to find the average price of each type of book and order the results by average price, the statement is:

```
select type, avg(price)
from titles
group by type
order by avg(price)
```

```
type
---------- ------------
UNDECIDED           NULL
mod_cook          11.49
psychology        13.50
business          13.73
trad_cook         15.96
popular_comp      21.48

(6 rows affected)
```

10.8.2 order by and group by Used with select distinct

A `select distinct` query with `order by` or `group by` can return duplicate values if the `order by` or `group by` column is not in the `select` list.

For example, the following query orders results by `pub_id`:

```
select distinct pub_id
from titles
order by type
```

```
pub_id
------
0877
0736
1389
0877
1389
0736
0877
0877

(8 rows affected)
```

If a query has an `order by` or `group by` clause that includes columns not in the `select` list, SAP ASE adds those columns as hidden columns in the columns being processed. The columns listed in the `order by` or `group by` clause are included in the test for distinct rows. To comply with ANSI standards, include the `order by` or `group by` column in the `select` list. For example:

```
select distinct pub_id, type
from titles
```

```
order by type
```

```
pub_id type
------ ------------
0877   UNDECIDED
0736   business
1389   business
0877   mod_cook
1389   popular_comp
0736   psychology
0877   psychology
0877   trad_cook

(8 rows affected)
```

10.9 Summarize Groups of Data: the compute Clause

The `compute` clause is a Transact-SQL extension. Use it with row aggregates to produce reports that show subtotals of grouped summaries.

Such reports, usually produced by a report generator, are called control-break reports, since summary values appear in the report under the control of the groupings ("breaks") you specify in the `compute` clause.

These summary values appear as additional rows in the query results, unlike the aggregate results of a `group by` clause, which appear as new columns.

A `compute` clause allows you to see detail and summary rows with a single `select` statement. You can calculate summary values for subgroups and you can calculate more than one row aggregate for the same group.

The general syntax for `compute` is:

```
compute <row_aggregate>(<column_name>)
     [, <row_aggregate>(<column_name>)]...
     [by <column_name> [, <column_name>]...]
```

The row aggregates you can use with `compute` are `sum`, `avg`, `min`, `max`, and `count`, and `count_big`. You can use `sum` and `avg` only with numeric columns. Unlike the `order by` clause, you cannot use the positional number of a column from the select list instead of the column name.

> **Note**
>
> You cannot use `text`, `unitext`, or `image` columns in a `compute` clause.

A system test may fail because there are too many aggregates in the `compute` clause of a query. The number of aggregates that each `compute` clause can accommodate is limited to 127, and if a `compute` clause contains more than 127 aggregates, the system generates an error message when you try to execute the query.

Each `avg` aggregate counts as two aggregates when you are counting toward the limit of 127, because an `avg` aggregate is actually a combination of a `sum` aggregate and a `count` aggregate.

Following are two queries and their results. The first one uses `group by` and aggregates. The second uses `compute` and row aggregates. Notice the difference in the results.

```
select type, sum(price), sum(advance)
from titles
group by type
```

```
type
------------   -------    ----------
UNDECIDED         NULL         NULL
business         54.92    25,125.00
mod_cook         22.98    15,000.00
popular_comp     42.95    15,000.00
psychology       67.52    21,275.00
trad_cook        47.89    19,000.00

(6 rows affected)
```

```
select type, price, advance
from titles
order by type
compute sum(price), sum(advance) by type
```

```
type         price                     advance
------------ ------------------------- --------
UNDECIDED    NULL                      NULL

Compute Result:
------------------------ ------------------------
                   NULL                      NULL

type         price                 advance
------------ --------------------- ----------
business     2.99                  10,125.00
business    11.95                   5,000.00
business    19.99                   5,000.00
business    19.99                   5,000.00

Compute Result:
------------------------ ------------------------
                   54.92                 25,125.00
type         price                    advance
------------ ------------------------ ---------
mod_cook      2.99                    15,000.00
mod_cook     19.99                         0.00

Compute Result:
------------------------ ------------------------
                   22.98                 15,000.00

type          price                 advance
------------- --------------------- ------------
popular_comp  NULL                        NULL
popular_comp  20.00                    8,000.00
popular_comp  22.95                    7,000.00

Compute Result:
------------------------ ------------------------
                   42.95                 15,000.00

type         price                     advance
------------ ------------------------- --------
psychology    7.00                     6,000.00
psychology    7.99                     4,000.00
```

```
psychology   10.95                    2,275.00
psychology   19.99                    2,000.00
psychology   21.59                    7,000.00

Compute Result:
----------------------- -----------------------
                    67.52               21,275.00

type         price                   advance
------------ ----------------------- --------
trad_cook    11.95                   4,000.00
trad_cook    14.99                   8,000.00
trad_cook    20.95                   7,000.00

Compute Result:
----------------------- -----------------------
                    47.89               19,000.00

(24 rows affected)
```

Each summary value is treated as a row.

Related Information

Row Aggregates and compute [page 311]

10.9.1 Row Aggregates and compute

Row aggregates can be used with `compute` statement.

This table describes how aggregates are used with a compute statement:

Row Aggregates	Result
`sum`	Total of the values in the expression
`avg`	Average of the values in the expression
`max`	Highest value in the expression
`min`	Lowest value in the expression
`count`	Number of selected rows as an `integer`
`count_big`	Number of selected rows as a `bigint`

These row aggregates are the same aggregates that can be used with `group by`, except there is no row aggregate function that is the equivalent of `count(*)`. To find the summary information produced by `group by` and `count(*)`, use a `compute` clause without the `by` keyword.

10.9.1.1 Rules for compute Clauses

There are several rules to consider when using the `compute` clause.

- SAP ASE does not allow the `distinct` keyword with the row aggregates.
- The columns in a `compute` clause must also be in the select list.
- You cannot use `select into` in the same statement as a `compute` clause because statements that include `compute` do not generate normal rows.
- You cannot use a `compute` clause in a `select` statement within an `insert` statement, for the same reason: statements that include `compute` do not generate normal rows.
- If you use `compute` with the `by` keyword, you must also use an `order by` clause. The columns listed after `by` must be identical to, or a subset of, those listed after `order by`, and must be in the same left-to-right order, start with the same expression, and cannot skip any expressions.
  For example, suppose the `order by` clause is:

  ```
  order by <a, b>, <c>
  ```

  The `compute` clause can be any or all of these:

  ```
  compute <row_aggregate> (<column_name>) by <a, b, c >
  ```

  ```
  compute <row_aggregate> (<column_name>) by <a, b >
  ```

  ```
  compute <row_aggregate> (<column_name>) by <a>
  ```

  The `compute` clause cannot be any of these:

  ```
  compute <row_aggregate> (<column_name>) by <b, c >
  ```

  ```
  compute <row_aggregate> (<column_name>) by <a, c>
  ```

  ```
  compute <row_aggregate> (<column_name>) by <c>
  ```

  You must use a column name or an expression in the `order by` clause; you cannot sort by a column heading.
- You can use the `compute` keyword without `by` to generate grand totals, grand counts, and so on. `order by` is optional if you use the `compute` keyword without `by`.

Related Information

Databases and Tables [page 50]
Generate Totals: compute Without by [page 315]

10.9.2 Specify More Than One Column After compute

In a query, listing more than one column after the `by` keyword breaks a group into subgroups, and applies the specified row aggregate to each level of grouping.

For example, this query finds the sum of the prices of psychology books from each publisher:

```
select type, pub_id, price
from titles
where type = "psychology"
order by type, pub_id, price
compute sum(price) by type, pub_id
```

```
type        pub_id  price
----------- ------- -------------
psychology    0736           7.00
psychology    0736           7.99
psychology    0736          10.95
psychology    0736          19.99
Compute Result:
---------------
          45.93
type        pub_id  price
----------- ------- -------------
psychology    0877          21.59

Compute Result:
---------------
          21.59

(7 rows affected)
```

10.9.3 Use More Than One compute Clause

You can use different aggregates in the same statement by including more than one `compute` clause.

The following query is similar to the preceding one but adds the sum of the prices of psychology books by publisher:

```
select type, pub_id, price
from titles
where type = "psychology"
order by type, pub_id, price
compute sum(price) by type, pub_id
compute sum(price) by type
```

```
type        pub_id  price
----------- ------- --------------
psychology    0736           7.00
psychology    0736           7.99
psychology    0736          10.95
psychology    0736          19.99

Compute Result:
---------------
          45.93

type        pub_id  price
```

```
 ---------- ------- --------------
 psychology    0877         21.59

Compute Result:
---------------
          21.59

Compute Result:
---------------
           67.52

(8 rows affected)
```

10.9.4 Apply an Aggregate to More Than One Column

One `compute` clause can apply the same aggregate to several columns.

This query finds the sums of prices and advances for each type of cookbook:

```
select type, price, advance
from titles
where type like "%cook"
order by type
compute sum(price), sum(advance) by type
```

```
type      price            advance
--------- ---------------- ---------------
mod_cook              2.99       15,000.00
mod_cook             19.99            0.00

Compute Result:
--------------- ---------------
          22.98       15,000.00

type      price            advance
--------- ---------------- ---------------
trad_cook            11.95        4,000.00
trad_cook            14.99        8,000.00
trad_cook            20.95        7,000.00

Compute Result:
--------------- ---------------
          47.89       19,000.00

(7 rows affected)
```

Remember, the columns to which the aggregates apply must also be in the select list.

10.9.5 Use Different Aggregates in the Aame compute Clause

You can use different aggregates in the same `compute` clause.

```
select type, pub_id, price
from titles
```

```
where type like "%cook"
order by type, pub_id
compute sum(price), max(pub_id) by type
```

```
type        pub_id  price
----------- ------- --------------
mod_cook    0877             2.99
mod_cook    0877            19.99

Compute Result:
--------------- ----
          22.98 0877

type        pub_id  price
----------- ------- --------------
trad_cook   0877            11.95
trad_cook   0877            14.99
trad_cook   0877            20.95

Compute Result:
--------------- ----
          47.89 0877

(7 rows affected)
```

10.9.6 Generate Totals: compute Without by

You can use the `compute` keyword without `by` to generate grand totals, grand counts, and so on.

This statement finds the grand total of the prices and advances of all types of books that cost more than $20:

```
select type, price, advance
from titles
where price > $20
compute sum(price), sum(advance)
```

```
type         price            advance
------------ ----------------  -------------
popular_comp            22.95      7,000.00
psychology              21.59      7,000.00
trad_cook               20.95      7,000.00

Compute Result:
--------------- ---------
          65.49 21,000.00

(4 rows affected)
```

You can use a `compute` with `by` and a `compute` without `by` in the same query. The following query finds the sums of prices and advances by type and then computes the grand total of prices and advances for all types of books.

```
select type, price, advance
from titles
where type like "%cook"
order by type
compute sum(price), sum(advance) by type
```

```
compute sum(price), sum(advance)
```

```
type         price                advance
-----------  -----------------    ------------
mod_ cook                 2.99       15,000.00
mod_cook                 19.99            0.00

Compute Result:
--------------- ---------
          22.98 15,000.00

type         price                advance
-----------  -----------------    ------------
trad_cook                11.95        4,000.00
trad_cook                14.99        8,000.00
trad_cook                20.95        7,000.00

Compute Result:
--------------- ---------
          47.89 19,000.00

Compute Result:
--------------- ---------
          70.87 34,000.00

(8 rows affected)
```

10.10 Combine Queries: the union Operator

The `union` operator combines the results of two or more queries into a single result set.

The Transact-SQL extension to `union` allows you to:

- Use `union` in the `select` clause of an `insert` statement.
- Specify new column headings in the `order by` clause of a `select` statement when `union` is present in the `select` statement.

See the *Reference Manual: Commands*.

These tables, T1 and T2. T1 shows two columns, “a, `char(4)`,” and “b,”`char(4)`. T2 contains two columns, “a `char(4)`,” and “b, `int`.” Each table has three rows: in T1, Row 1 shows “abc” in the “a” column and “1” in the “b” column. T1 Row 2 shows “def” in the “a” column, and “2” in the “b” column. Row 3 shows “ghi” in the “a” column, and “3” in the “b `int`” column. Table T4, Row 1, shows “ghi” in the “a” column and “1” in the “b” column; Row 2 shows “jkl” in the “a” column and “2” in the “b” column; Row 3 shows “mno” in the “a” column and “3” in the “b(`int`)” column.

Table T1	
a char(4)	b int
abc	1
def	2
ghi	3

Table T2	
a char(4)	b int
ghi	3
jkl	4
mno	5

The following query creates a `union` between the two tables:

```
create table T1 (a char(4), b int)
insert T1 values ("abc", 1)
insert T1 values ("def", 2)
insert T1 values ("ghi", 3)
create table T2 (a char(4), b int)
insert T2 values ("ghi", 3)
insert T2 values ("jkl", 4)
insert T2 values ("mno", 5)
select * from T1
union
select * from T2
```

```
a     b
----  ---------
abc           1
def           2
ghi           3
jkl           4
mno           5

(5 rows affected)
```

By default, the `union` operator removes duplicate rows from the result set. Use the `all` option to include duplicate rows. Notice also that the columns in the result set have the same names as the columns in `T1`. You can use any number of `union` operators in a Transact-SQL statement. For example:

```
<x >union <y> union <z>
```

By default, SAP ASE evaluates a statement containing `union` operators from left to right. You can use parentheses to specify a different evaluation order.

For example, the following two expressions are not equivalent:

```
<x> union all (<y> union <z>)
(<x> union all <y>) union <z>
```

In the first expression, duplicates are eliminated in the union between `<y>` and `<z>`. Then, in the union between that set and `<x>`, duplicates are *not* eliminated. In the second expression, duplicates are included in the union between `<x>` and `<y>`, but are then eliminated in the subsequent union with `<z>`; `all` does not affect the final result of this statement.

10.10.1 Guidelines for union Queries

SAP ASE provides recommendations and guidelines when using `union` statements.

When you use `union` statements:

- All select lists in the `union` statement must have the same number of expressions (such as column names, arithmetic expressions, and aggregate functions). The following statement is invalid because the first select list is longer than the second:

  ```
  create table stores_east
  ```

```
(stor_id char(4) not null,
stor_name varchar(40) null,
stor_address varchar(40) null,
city varchar(20) null,
state char(2) null,
country varchar(12) null,
postalcode char(10) null,
payterms varchar(12) null)
select stor_id, city, state from stores
union
select stor_id, city from stores_east
drop table stores_east
```

- Corresponding columns in all tables, or any subset of columns used in individual queries, must be of the same datatype, or an implicit data conversion must be possible between the two datatypes, or an explicit conversion must be supplied. For example, a `union` between a column of the `char` datatype and one of the `int` datatype is possible, unless an explicit conversion is supplied. However, a `union` between a column of the `money` datatype and one of the `int` datatype is possible. See `union` in the *Reference Manual: Commands* and, "System and User-Defined Datatypes," in the *Reference Manual: Building Blocks*.
- You must place corresponding columns in the individual queries of a `union` statement in the same order, because `union` compares the columns one to one in the order given in the query. For example, suppose you have the following tables:

Table T3	
a	b
int	char(4)
1	abc
2	def
3	ghi

Table T4	
a	b
char(4)	int
abc	1
def	2
ghi	3

The above table shows two tables, T3 and T4. T3 has two columns, "a," `int`, and "b,"`char(4)`. T4 contains two columns, "a" `char(4)`, and "b," `int`. Each table has three rows: Row 1 shows "1" in the "a" column, and "abc" in the "b" column. Row 2 shows "2" in the "a" column, and "def" in the "b" column. Row 3 shows "3" in the "a" column, and "ghi" in the "b `char`" column. Table T4, Row 1, shows "abc" in the "a" column, "1" in the "b" column; Row 2 shows "def" in the "a" column, "2" in the "b" column; Row 3 shows "ghi" in the "a" column and "3" in the "b(`int`)" column.
Enter this query:

```
select a, b from T3
union
select b, a from T4
```

The query produces:

```
a          b
---------  ---
        1  abc
        2  def
        3  ghi

(3 rows affected)
```

The following query, however, results in an error message, because the datatypes of corresponding columns are incompatible:

```
select a, b from T3
```

```
union
select a, b from T4
drop table T3
drop table T4
```

When you combine different (but compatible) datatypes such as `float` and `int` in a `union` statement, SAP ASE converts them to the datatype with the most precision.

- SAP ASE takes the column names in the table resulting from a `union` from the first individual query in the `union` statement. Therefore, to define a new column heading for the result set, do so in the first query. In addition, to refer to a column in the result set by a new name, for example, in an `order by` statement, refer to it in that way in the first `select` statement.
  The following query is correct:

```
select Cities = city from stores
union
select city from authors
order by Cities
```

Follow these guidelines when you use `union` statements with other Transact-SQL commands:

- The first query in the `union` statement may contain an `into` clause that creates a table to hold the final result set. For example, the following statement creates a table called `results` that contains the union of tables `publishers`, `stores`, and `salesdetail`:

```
use mastersp_dboption pubs2, "select into", true
use pubs2
checkpoint
select pub_id, pub_name, city into results
from publishers
union
select stor_id, stor_name, city from stores
union
select stor_id, title_id, ord_num from salesdetail
```

  You can use the `into` clause only in the first query; if it appears anywhere else, you get an error message.
- You can use `order by` and `compute` clauses only at the end of the `union` statement to define the order of the final results or to compute summary values. You cannot use them within the individual queries that make up the `union` statement. Specifically, you cannot use `compute` clauses within an `insert...select` statement.
- You can use `group by` and `having` clauses within individual queries only; you cannot use them to affect the final result set.
- You can also use the `union` operator within an `insert` statement. For example:

```
create table tour (city varchar(20), state char(2))
insert into tour
    select city, state from stores
    union
    select city, state from authors
drop table tour
```

- You can use the `union` operator within a `create view` statement.
- You cannot use the `union` operator on `text` and `image` columns.
- You cannot use the `for browse` clause in statements involving the `union` operator.

11 Joins: Retrieve Data from Several Tables

A join operation compares two or more tables (or views) by specifying a column from each, comparing the values in those columns row by row, and linking the rows that have matching values. It then displays the results in a new table.

The tables specified in the join can be in the same database or in different databases.

You can state many joins as subqueries, which also involve two or more tables. When you join two or more tables, the columns being compared must have similar values—that is, values using the same or similar datatypes.

There are several types of joins, such as equijoins, natural joins, and outer joins. The most common join, the equijoin, is based on equality. The following join finds the names of authors and publishers located in the same city:

```
select au_fname, au_lname, pub_name
from authors, publishers
where authors.city = publishers.city
```

```
au_fname      au_lname      pub_name
--------      --------      --------------------
Cheryl        Carson        Algodata Infosystems
Abraham       Bennet        Algodata Infosystems

(2 rows affected)
```

Because the query draws on information contained in two separate tables, `publishers` and `authors`, you need a join to retrieve the requested information. This statement joins the `publishers` and `authors` tables using the `city` column as the link:

```
where authors.city = publishers.city
```

When Component Integration Services is enabled, you can perform joins across remote servers. See the *Component Integratin Services Users Guide*.

Related Information

Subqueries: Queries Within Other Queries [page 257]

11.1 Join Syntax

You can embed a join in a `select`, `update`, `insert`, `delete`, or subquery. Other join restrictions and clauses may follow the join conditions.

Joins use this syntax:

```
<start of select, update, insert, delete, or subquery>
     from {<table_list> | <view_list>}
     where [not]
          [<table_name>. | <view_name>.]<column_name> <join_operator>
          [<table_name>. | <view_name>.]<column_name>
     [{and | or} [not]
          [<table_name>.|<view_name>.]<column_name ><join_operator>
           [<table_nam>e.|<view_name>.]<column_name>]...
```

```
<End of select, update, insert, delete, or subquery>
```

11.2 Joins and the Relational Model

The join operation is the hallmark of the relational model of database management. More than any other feature, the join distinguishes relational database management systems from other types of database management systems.

In structured database management systems, often known as network and hierarchical systems, relationships between data values are predefined. Once a database has been set up, it is difficult to make queries about unanticipated relationships among the data.

In a relational database management system, relationships among data values are left unstated in the definition of a database. They become explicit when the data is manipulated—when you query the database, not when you create it. You can ask any question that comes to mind about the data stored in the database, regardless of what was intended when the database was set up.

According to the rules of good database design, called normalization rules, each table should describe one kind of entity—a person, place, event, or thing. That is why, when you want to compare information about two or more kinds of entities, you need the join operation. Relationships among data stored in different tables are discovered by joining them.

A corollary of this rule is that the join operation gives you unlimited flexibility in adding new kinds of data to your database. You can always create a new table that contains data about a different kind of entity. If the new table has a field with values similar to those in some field of an existing table or tables, it can be linked to those other tables by joining.

11.3 How Joins are Structured

A join statement, like a select statement, starts with the keyword `select`. The columns named after the `select` keyword are the columns to be included in the query results, in their desired order.

This example specifies the columns that contained the authors' names from the `authors` table, and publishers' names from the `publishers` tables:

```
select au_fname, au_lname, pub_name
from authors, publishers
```

You do not have to qualify the columns `au_fname`, `au_lname`, and `pub_name` by a table name because there is no ambiguity about the table to which they belong. But the `city` column used for the join comparison does need to be qualified, both the `authors` and `publishers` tables include columns with that name:

```
select au_fname, au_lname, pub_name
from authors, publishers
where authors.city = publishers.city
```

Though neither of the `city` columns is printed in the results, SAP ASE needs the table name to perform the comparison.

To specify that all the columns of the tables involved in the query be included in the results, use an asterisk (`*`) with `select`. For example, to include all the columns in `authors` and `publishers` in the preceding join query, the statement is:

```
select *
from authors, publishers
where authors.city = publishers.city
```

```
au_id       au_lname au_fname phone        address
city       state postalcode contract pub_id pub_name
city       state
----------- -------- -------- ------------ ---------------------
---------- ----- ---------- -------- ------ --------------------
---------- -----
238-95-7766 Carson   Cheryl   415 548-7723 589 Darwin Ln.
Berkeley   CA    94705      1        1389   Algodata Infosystems
Berkeley   CA
409-56-7008 Bennet   Abraham  415 658-9932 223 Bateman St
Berkeley   CA    94705      1        1389   Algodata Infosystems
Berkeley   CA

(2 rows affected)
```

The output shows a total of 2 rows with 13 columns each. Because of the length of the rows, each takes up multiple horizontal lines. Whenever "*" is used, the columns in the results appear in the order in which they were stated in the `create` statement that created the table.

The select list and the results of a join need not include columns from both of the tables being joined. For example, to find the names of the authors that live in the same city as one of the publishers, your query need not include any columns from `publishers`:

```
select au_lname, au_fname
from authors, publishers
where authors.city = publishers.city
```

Just as in any `select` statement, column names in the select list and table names in the `from` clause must be separated by commas.

11.3.1 The from Clause

Use the `from` clause to specify the tables and views to use in a join statement. .

This is the clause that indicates to SAP ASE that a join is desired. You can list the tables or views in any order. The order of tables affects the results that appear only when you use `select *` to specify the select list.

At most, a query can reference 250 tables and 46 worktables (such as those created by aggregate functions). The 250-table limit includes:

- Tables (or views on tables) listed in the `from` clause
- Each instance of multiple references to the same table (self-joins)
- Tables referenced in subqueries
- Tables being created with `into`
- Base tables referenced by the views listed in the `from` clause

The following example joins columns from the `titles` and `publishers` tables, doubling the price of all books published in California:

```
begin tran
```

```
update titles
  set price = price * 2
  from titles, publishers
  where titles.pub_id = publishers.pub_id
  and publishers.state = "CA"
```

```
rollback tran
```

Table or view names can be qualified by the names of the owner and database, and can be given correlation names for convenience. For example:

```
select au_lname, au_fname
from pubs2.blue.authors, pubs2.blue.publishers
where authors.city = publishers.city
```

Related Information

Join More Than Two Tables [page 333]
Views: Limit Access to Data [page 393]

11.3.2 The where Clause

Use the `where` clause to determine which rows are included in the results of a join statement.

`where` specifies the connection between the tables and views named in the `from` clause. Qualify column names if there is ambiguity about the table or view to which they belong. For example:

```
where authors.city = publishers.city
```

This `where` clause gives the names of the columns to be joined, qualified by table names if necessary, and the join operator—often equality, sometimes "greater than" or "less than."

> **Note**
>
> You will get unexpected results if you omit the `where` clause of a join. Without a `where` clause, any of the join queries discussed so far produces 69 rows instead of 2.

The `where` clause of a join statement can include conditions other than the one that links columns from different tables. In other words, you can include a join operation and a `select` operation in the same SQL statement.

Related Information

How Joins are Processed [page 326]
How Joins are Processed [page 326]

11.3.2.1 Join Operators

Joins that match columns on the basis of equality are called equijoins.

Equijoins use the following comparison operators:

Operator	Meaning
=	Equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
! =	Not equal to

Operator	Meaning
!>	Less than or equal to
!<	Greater than or equal to

Joins that use the relational operators are collectively called theta joins.

These set of join operators are used for outer joins:

Operator	Action
*=	Include in the results all the rows from the first table, not just the ones where the joined columns match.
=*	Include in the results all the rows from the second table, not just the ones where the joined columns match.

Related Information

Equijoins and Natural Joins [page 327]

11.3.2.2 Datatypes in Join Columns

The columns being joined must have the same or compatible datatypes. Use the `convert` function when comparing columns that are assigned datatypes that cannot be implicitly converted. Columns being joined need not have the same name, although they often do.

If the datatypes used in the join are compatible, SAP ASE automatically converts them. For example, SAP ASE converts among any of the numeric type columns:

- `bigint`
- `int`
- `smallint`
- `tinyint`
- `unsigned bigint`
- `unsigned int`
- `unsigned smallint`
- `decimal`
- or `float`

Among any of the character type and date columns, SAP ASE converts:

- `char`
- `varchar`

- unichar
- univarchar
- nchar
- nvarchar
- datetime
- date
- time

Related Information

Transact-SQL Functions [page 471]

11.3.2.3 Joins and Text and Image Columns

You cannot use joins for columns containing text or image values. You can, however, compare the lengths of text columns from two tables with a where clause.

For example:

```
where datalength(textab_1.textcol) >
datalength(textab_2.textcol)
```

11.4 How Joins are Processed

Knowing how joins are processed helps to understand them—and to figure out why, when you incorrectly state a join, you sometimes get unexpected results.

The first step in processing a join is to form the Cartesian product of the tables—all the possible combinations of the rows from each of the tables. The number of rows in a Cartesian product of two tables is equal to the number of rows in the first table multiplied by the number of rows in the second table.

The Cartesian product of the authors table and the publishers table is 69 (23 authors multiplied by 3 publishers). You can have a look at a Cartesian product with any query that includes columns from more than one table in the select list, more than one table in the from clause, and no where clause. For example, if you omit the where clause from the join used in any of the previous examples, SAP ASE combines each of the 23 authors with each of the 3 publishers, and returns all 69 rows.

```
select au_lname, au_fname
from authors, publishers
```

This Cartesian product does not contain any particularly useful information. It is actually misleading because it implies that every author in the database has a relationship with every publisher in the database—which is not true.

Including a `where` clause in the join specifies the columns to be matched and the basis on which to match them. It may also include other restrictions. Once SAP ASE forms the Cartesian product, it eliminates the rows that do not satisfy the join by using the conditions in the `where` clause.

For example, the `where` clause in the example cited (the Cartesian product of the `authors` table and the `publishers` table) eliminates from the results all rows in which the author's city is not the same as the publisher's city:

```
where authors.city = publishers.city
```

11.5 Equijoins and Natural Joins

Joins that are based on equality (=) are called equijoins. Equijoins compare the values in the columns being joined for equality and then include all the columns in the tables being joined in the results.

This query is an example of an equijoin:

```
select *
from authors, publishers
where authors.city = publishers.city
```

In the results of this statement, the `city` column appears twice. By definition, the results of an equijoin contain two identical columns. Because there is usually no point in repeating the same information, one of these columns can be eliminated by restating the query. The result is called a natural join.

The query that results in the natural join of `publishers` and `authors` on the `city` column is:

```
select publishers.pub_id, publishers.pub_name,
  publishers.state, authors.*
from publishers, authors
where publishers.city = authors.city
```

The column `publishers.city` does not appear in the results.

Another example of a natural join is:

```
select au_fname, au_lname, pub_name
from authors, publishers
where authors.city = publishers.city
```

You can use more than one join operator to join more than two tables, or to join more than two pairs of columns. These "join expressions" are usually connected with `and`, although `or` is also legal.

Following are two examples of joins connected by `and`. The first lists information about books (type of book, author, and title), ordered by book type. Books with more than one author have multiple listings, one for each author.

```
select type, au_lname, au_fname, title
from authors, titles, titleauthor
where authors.au_id = titleauthor.au_id
and titles.title_id = titleauthor.title_id
order by type
```

The second finds the names of authors and publishers that are located in the same city and state:

```
select au_fname, au_lname, pub_name
from authors, publishers
where authors.city = publishers.city
and authors.state = publishers.state
```

11.6 Joins with Additional Conditions

In a join query, both the `where` clause and the join condition can include selection criteria.

For example, to retrieve the names and publishers of all the books for which advances of more than $7500 were paid, use:

```
select title, pub_name, advance
from titles, publishers
where titles.pub_id = publishers.pub_id
and advance > $7500
```

```
title                           pub_name              advance
-----------------------------   --------------------  ---------
You Can Combat Computer Stress!  New Age Books        10,125.00
The Gourmet Microwave            Binnet & Hardley     15,000.00
Secrets of Silicon Valley        Algodata Infosystems  8,000.00
Sushi, Anyone?                   Binnet & Hardley      8,000.00

(4 rows affected)
```

The columns being joined (`pub_id` from `titles` and `publishers`) need not appear in the select list and, therefore, do not show up in the results.

You can include as many selection criteria as you want in a join statement. The order of the selection criteria and the join condition has no effect on the result.

11.7 Joins Not Based on Equality

The condition for joining the values in two columns does not need to be equality. You can use any of the other comparison operators: not equal (`!=`), greater than (>), less than (<), greater than or equal to (>=), and less than or equal to (<=).

Transact-SQL also provides the operators `!>` and `!<`, which are equivalent to <= and >=, respectively.

This example of a greater-than join finds New Age authors who live in states that come after New Age Books' state, Massachusetts, in alphabetical order.

```
select pub_name, publishers.state,
   au_lname, au_fname, authors.state
from publishers, authors
where authors.state > publishers.state
```

```
and pub_name = "New Age Books"
```

```
pub_name         state   au_lname       au_fname         state
-------------    ------  -------------- -----------      -----
New Age Books    MA      Greene         Morningstar      TN
New Age Books    MA      Blotchet-Halls Reginald         OR
New Age Books    MA      del Castillo   Innes            MI
New Age Books    MA      Panteley       Sylvia           MD
New Age Books    MA      Ringer         Anne             UT
New Age Books    MA      Ringer         Albert           UT

(6 rows affected)
```

The following example uses a >= join and a < join to look up the correct `royalty` from the `roysched` table, based on the book's total sales.

```
select t.title_id, t.total_sales, r.royalty
from titles t, roysched r
where t.title_id = r.title_id
and t.total_sales >= r.lorange
and t.total_sales < r.hirange
```

```
title_id       total_sales      royalty
--------       -----------      -------
BU1032               4095           10
BU1111               3876           10
BU2075               1872           24
BU7832               4095           10
MC2222               2032           12
MC3021              22246           24
PC1035               8780           16
PC8888               4095           10
PS1372                375           10
PS2091               2045           12
PS2106                111           10
PS3333               4072           10
PS7777               3336           10
TC3218                375           10
TC4203              15096           14
TC7777               4095           10

(16 rows affected)
```

11.8 Self-Joins and Correlation Names

Joins that compare values within the same column of one table are called self-joins. To distinguish the two roles in which the table appears, use aliases, or correlation names.

For example, you can use a self-join to find out which authors in Oakland, California, live in the same postal code area. Since this query involves a join of the `authors` table with itself, the `authors` table appears in two roles. To distinguish these roles, you can temporarily and arbitrarily give the `authors` table two different correlation names—such as `au1` and `au2`—in the `from` clause. These correlation names qualify the column names in the rest of the query. The self-join statement looks like this:

```
select au1.au_fname, au1.au_lname,
au2.au_fname, au2.au_lname
```

```
from authors au1, authors au2
where au1.city = "Oakland" and au2.city = "Oakland"
and au1.state = "CA" and au2.state = "CA"
and au1.postalcode = au2.postalcode
```

```
au_fname     au_lname    au_fname    au_lname
---------    ----------- --------    --------
Marjorie     Green       Marjorie    Green
Dick         Straight    Dick        Straight
Dick         Straight    Dirk        Stringer
Dick         Straight    Livia       Karsen
Dirk         Stringer    Dick        Straight
Dirk         Stringer    Dirk        Stringer
Dirk         Stringer    Livia       Karsen
Stearns      MacFeather  Stearns     MacFeather
Livia        Karsen      Dick        Straight
Livia        Karsen      Dirk        Stringer
Livia        Karsen      Livia       Karsen

(11 rows affected)
```

List the aliases in the `from` clause in the same order as you refer to them in the select list, as in this example. Depending on the query, the results may be ambiguous if you list them in a different order.

To eliminate the rows in the results where the authors match themselves, and are identical except that the order of the authors is reversed, you can make this addition to the self-join query:

```
select au1.au_fname, au1.au_lname,
au2.au_fname, au2.au_lname
from authors au1, authors au2
where au1.city = "Oakland" and au2.city = "Oakland"
and au1.state = "CA" and au2.state = "CA"
and au1.postalcode = au2.postalcode
and au1.au_id < au2.au_id
```

```
au_fname    au_lname     au_fname    au_lname
---------   -----------  ---------   ---------
Dick        Straight     Dirk        Stringer
Dick        Straight     Livia       Karsen
Dirk        Stringer     Livia       Karsen

(3 rows affected)
```

It is now clear that Dick Straight, Dirk Stringer, and Livia Karsen all have the same postal code.

11.9 The Not-Equal Join

A not-equal join is useful for restricting the rows returned by a self-join.

In the following example, a not-equal join and a self-join find the categories in which there are two or more inexpensive (less than $15) books of different prices:

```
select distinct t1.type, t1.price
from titles t1, titles t2
where t1.price < $15
and t2.price < $15
and t1.type = t2.type
```

```
and t1.price != t2.price
```

```
type        price
----------  -----
business     2.99
business    11.95
psychology   7.00
psychology   7.99
psychology  10.95
trad_cook   11.95
trad_cook   14.99

(7 rows affected)
```

The expression "not <column_name> = <column_name>" is equivalent to "<column_name> != <column_name>."

The following example uses a not-equal join combined with a self-join. It finds all the rows in the `titleauthor` table where there are two or more rows with the same `title_id`, but different `au_id` numbers, that is books that have multiple authors.

```
select distinct t1.au_id, t1.title_id
from titleauthor t1, titleauthor t2
where t1.title_id = t2.title_id
and t1.au_id != t2.au_id
order by t1.title_id
```

```
au_id           title_id
-----------     --------
213-46-8915     BU1032
409-56-7008     BU1032
267-41-2394     BU1111
724-80-9391     BU1111
722-51-5454     MC3021
899-46-2035     MC3021
427-17-2319     PC8888
846-92-7186     PC8888
724-80-9391     PS1372
756-30-7391     PS1372
899-46-2035     PS2091
998-72-3567     PS2091
267-41-2394     TC7777
472-27-2349     TC7777
672-71-3249     TC7777

(15 rows affected)
```

For each book in `titles`, the following example finds all other books of the same type that have a different price:

```
select t1.type, t1.title_id, t1.price, t2.title_id, t2.price
from titles t1, titles t2
where t1.type = t2.type
and t1.price != t2.price
```

11.9.1 Not-Equal Joins and Subqueries

Sometimes a not-equal join query is insufficiently restrictive and must be replaced by a subquery.

For example, suppose you want to list the names of authors who live in a city where no publisher is located. For the sake of clarity, let us also restrict this query to authors whose last names begin with “A”, “B”, or “C”. A not-equal join query might be:

```
select distinct au_lname, authors.city
from publishers, authors
where au_lname like "[ABC]%"
and publishers.city != authors.city
```

The results are not an answer to the question that was asked:

```
au_lname             city
----------------     ------------
Bennet               Berkeley
Carson               Berkeley
Blotchet-Halls       Corvallis

(3 rows affected)
```

The system interprets this query as: “find the names of authors who live in a city where a publisher is not located.” Only the authors who live in Berkeley and Corvalis qualify – towns that do not have publishers.

In this case, the way that the system handles joins (first finding every eligible combination before evaluating other conditions) causes this query to return undesirable results. Use a subquery to get the results you want. A subquery can eliminate the ineligible rows first and then perform the remaining restrictions.

Here is the correct statement:

```
select distinct au_lname, city
from authors
where au_lname like "[ABC]%"
and city not in
(select city from publishers
where authors.city = publishers.city)
```

Now, the results are what you want:

```
au_lname             city
-------------        ------------
Blotchet-Halls       Corvallis

(1 row affected)
```

Related Information

Subqueries: Queries Within Other Queries [page 257]

11.10 Join More Than Two Tables

You can join more than two tables or join more than two pairs of columns in the same statement.

To find the titles of all the books of a particular type and the names of their authors, use:

```
select au_lname, au_fname, title
from authors, titles, titleauthor
where authors.au_id = titleauthor.au_id
and titles.title_id = titleauthor.title_id
and titles.type = "trad_cook"
```

```
au_lname       au_fname     title
-------------- -----------  ------------------------
Panteley       Sylvia       Onions, Leeks, and Garlic: Cooking
                            Secrets of the Mediterranean
Blotchet-Halls Reginald     Fifty Years in Buckingham Palace
                            Kitchens
O'Leary        Michael      Sushi, Anyone?
Gringlesby     Burt         Sushi, Anyone?
Yokomoto       Akiko        Sushi, Anyone?

      (5 rows affected)
```

One of the tables in the `from` clause, `titleauthor`, does not contribute any columns to the results. Nor do any of the columns that are joined—`au_id` and `title_id`—appear in the results. Nonetheless, this join is possible only by using `titleauthor` as an intermediate table.

You can also join more than two pairs of columns in the same statement. For example, here is a query that shows the `title_id`, its total sales and the range in which they fall, and the resulting royalty:

```
select titles.title_id, total_sales, lorange, hirange, royalty
from titles, roysched
where titles.title_id = roysched.title_id
and total_sales >= lorange
and total_sales < hirange
```

```
title_id    total_sales   lorange hirange  royalty
--------    -----------   ------- -------  -------
 BU1032            4095         0    5000       10
 BU1111            3876         0    4000       10
 BU2075           18722     14001   50000       24
 BU7832            4095         0    5000       10
 MC2222            2032      2001    4000       12
 MC3021            2224     12001   50000       24
 PC1035            8780      4001   10000       16
 PC8888            4095         0    5000       10
 PS1372             375         0   10000       10
 PS2091            2045      1001    5000       12
 PS2106             111         0    2000       10
 PS3333            4072         0    5000       10
 PS7777            3336         0    5000       10
 TC3218             375         0    2000       10
 TC4203           15096      8001   16000       14
 TC7777            4095         0    5000       10

(16 rows affected)
```

When there is more than one join operator in the same statement, either to join more than two tables or to join more than two pairs of columns, the "join expressions" are almost always connected with `and`, as in the earlier examples. However, it is also legal to connect them with `or`.

11.11 Star Joins

A star join is a commonly used data warehouse query that runs against a star schema database, which consists of a large table (also known as a fact table) surrounded by dimension tables.

Fact tables typically include two types of columns: one that includes measurements, metrics, or facts about the business process, and another column that includes foreign keys to dimension tables. Dimension tables usually include descriptive attributes. For example, a fact table may include information about the number of hiking books sold by a particular author on a particular day. The corresponding dimension table includes the address, age, and gender of the author.

See *Performance and Tuning Series: Query Processing and Abstract Plans > Joins: Retrieve Data from Several Tables > Query Plan Optimization with Star Joins*.

11.12 Outer Joins

Joins that include all rows, regardless of whether there is a matching row, are called outer joins.

For example, the following query joins the `titles` and the `titleauthor` tables on their `title_id` column:

```
select *
from titles, titleauthor
where titles.title_id *= titleauthor.title_id
```

SAP supports both Transact-SQL and ANSI outer joins. Transact-SQL outer joins use the `*=` command to indicate a left outer join and the `=*` command to indicate a right outer join. Transact-SQL outer joins were created by SAP as part of the Transact-SQL language.

ANSI outer joins use the keywords `left join` and `right join` to indicate a left and right join, respectively. SAP implemented the ANSI outer join syntax to fully comply with the ANSI standard. This is the previous example rewritten as an ANSI outer join:

```
select *
from titles left join titleauthor
on titles.title_id = titleauthor.title_id
```

Related Information

Transact-SQL Outer Joins [page 348]
ANSI Inner and Outer Joins [page 336]

11.12.1 Inner and Outer Tables

The terms outer table and inner table describe the placement of tables in an outer join.

- In a *left join*, the outer table and inner table are the left and right tables respectively. The outer table and inner table are also referred to as the row-preserving and null-supplying tables, respectively.
- In a *right join*, the outer table and inner table are the right and left tables, respectively.

For example, in the queries below, `T1` is the outer table and `T2` is the inner table:

```
T1 left join T2
T2 right join T1
```

Or, using Transact-SQL syntax:

```
T1 *= T2
T2 =* T1
```

11.12.2 Outer Join Restrictions

If a table is an inner member of an outer join, it cannot participate in both an outer join clause and a regular join clause.

The following query fails because the `salesdetail` table is part of both the outer join and a regular join clause:

```
select distinct sales.stor_id, stor_name, title
from sales, stores, titles, salesdetail
where qty > 500
and salesdetail.title_id =* titles.title_id
and sales.stor_id = salesdetail.stor_id
and sales.stor_id = stores.stor_id
```

```
Msg 303, Level 16, State 1:
Server 'FUSSY', Line 1:
The table 'salesdetail' is an inner member of an outer-join clause. This is not
allowed if the table also participates in a regular join clause.
```

If you want to know the name of the store that sold more than 500 copies of a book, you must use a second query. If you submit a query with an outer join and a qualification on a column from the inner table of the outer join, the results may not be what you expect. The qualification in the query does not restrict the number of rows returned, but rather affects which rows contain the null value. For rows that do not meet the qualification, a null value appears in the inner table's columns of those rows.

11.12.3 Views Used with Outer Joins

If you define a view with an outer join, then query the view with a qualification on a column from the inner table of the outer join, the results may not be what you expect. The query returns all rows from the inner table.

Rows that do not meet the qualification show a null value in the appropriate columns of those rows.

The following rules determine the types of updates you can make to columns through join views:

- `delete` statements are not allowed on join views.
- `insert` statements are not allowed on join views created `with check option`.
- `update` statements are allowed on join views `with check option`. The update fails if any of the affected columns appear in the `where` clause, in an expression that includes columns from more than one table.
- If you insert or update a row through a join view, all affected columns must belong to the same base table.

11.12.4 ANSI Inner and Outer Joins

Use ANSI join syntax to write either inner joins or outer joins, or to join views.

This is the ANSI syntax for joining tables:

```
<left_table> [inner | left [outer] | right [outer]] join <right_table>
on <left_column_name>  = <right_column_name>
```

The result of the join between the left and the right tables is called a joined table. Joined tables are defined in the `from` clause. For example:

```
select titles.title_id, title, ord_num, qty
from titles left join salesdetail
on titles.title_id = salesdetail.title_id
```

```
title_id title                ord_num                 qty
----------------------------- -------------------- -----
BU1032  The Busy Executive   AX-532-FED-452-2Z7   200
BU1032  The Busy Executive   NF-123-ADS-642-9G3   1000
. . .
TC7777  Sushi, Anyone?       ZD-123-DFG-752-9G8   1500
TC7777  Sushi, Anyone?       XS-135-DER-432-8J2   1090
(118 rows affected)
```

ANSI join syntax allows you to write either:

- Inner joins, in which the joined table includes only the rows of the inner and outer tables that meet the conditions of the `on` clause. The result set of a query that includes an inner join does not include any null-supplied rows for the rows of the outer table that do not meet the conditions of the `on` clause.
- Outer joins, in which the joined table includes all the rows from the outer table, whether or not they meet the conditions of the `on` clause. If a row does not meet the conditions of the `on` clause, values from the inner table are stored in the joined table as null values. The `where` clause of an ANSI outer join restricts the rows that are included in the query result.

SAP ANSI join syntax does not support the `using` clause.

Related Information

ANSI Inner Joins [page 338]
ANSI outer joins [page 341]

11.12.4.1 Correlation Name and Column Referencing Rules for ANSI Joins

SAP ASE provides correlation name and column reference recommendations specifically for ANSI joins.

- If a table or view uses a correlation name reference to a column or view, it must always use the same correlation name, rather than the table name or view name. That is, you cannot name a table in a query with a correlation name and then use its table name later. The following example correctly uses the correlation name `t` to specify the table where its `pub_id` column is specified:

  ```
  select title, t.pub_id, pub_name
  from titles t left join publishers p
  on t.pub_id = p.pub_id
  ```

  However, the following example incorrectly uses the table name instead of the correlation name for the `titles` table (`t.pub_id`) in the `on` clause of the query, and produces the subsequent error message:

  ```
  select title, t.pub_id, pub_name
  from titles t left join publishers p
  on titles.pub_id = p.pub_id
  ```

  ```
  Msg 107, Level 15, State 1:
  Server 'server_name', Line 1:
  The column prefix 't' does not match with a table name or alias name used in
  the query. Either the table is not specified in the FROM clause or it has a
  correlation name which must be used instead.
  ```

- The restriction specified in the `on` clause can reference:
  - Columns that are specified in the joined table's reference
  - Columns that are specified in joined tables that are contained in the ANSI join (for example, in a nested join)
  - Correlation names in subqueries for tables specified in outer query blocks
- The condition specified in the `on` clause cannot reference columns that are introduced in ANSI joins that contain another ANSI join (typically when the joined table produced by the second join is joined with the first join).
  Here is an example of an illegal column reference that produces an error:

  ```
  select *
  from titles left join titleauthor
  on titles.title_id=roysched.title_id    /*join #1*/
  left join roysched
  on titleauthor.title_id=roysched.title_id    /*join #2*/
  where titles.title_id != "PS7777"
  ```

  The first left join cannot reference the `roysched.title_id` column because this column is not introduced until the second join. You can correctly rewrite this query as:

  ```
  select *
  from titles
  left join (titleauthor
  left join roysched
  on titleauthor.title_id = roysched.title_id) /*join #1*/
  on titles.title_id = roysched.title_id      /*join #2*/
  where titles.title_id != "PS7777"
  ```

And another example:

```
select title, price, titleauthor.au_id, titleauthor.title_id, pub_name,
publishers.city
from roysched, titles
left join titleauthor
on roysched.title_id=titleauthor.title_id
left join authors
on titleauthor.au_id=roysched.au_id, publishers
```

In this query, neither the `roysched` table nor the `publishers` table are part of either left join. Because of this, neither left join can refer to columns in either the `roysched` or `publishers` tables as part of their `on` clause condition.

Related Information

Self-Joins and Correlation Names [page 329]

11.12.4.2 ANSI Inner Joins

Joins that produce a result set that includes only the rows of the joining tables that meet the restriction are called inner joins. Rows that do not meet the join restriction are not included in the joined table.

If you require the joined table to include all the rows from one of the tables, regardless of whether they meet the restriction, use an outer join.

SAP ASE supports the use of both Transact-SQL inner joins and ANSI inner joins. Queries using Transact-SQL inner joins separate the tables being joined by commas and list the join comparisons and restrictions in the `where` clause. For example:

```
select au_id, titles.title_id, title, price
from titleauthor, titles
where titleauthor.title_id = titles.title_id
and price > $15
```

ANSI-standard `inner join` syntax is:

```
select <select_list>
     from <table1> inner join <table2
     >on <join_condition>
```

For example, the following use of `inner join` is equivalent to the Transact-SQL join above:

```
select au_id, titles.title_id, title, price
from titleauthor inner join titles
on titleauthor.title_id = titles.title_id
and price > 15
```

```
au_id         title_id  title                       price
----------    --------  ------------------------    -----
213-46-8915   BU1032    The Busy Executive's Datab  19.99
409-56-7008   BU1032    The Busy Executive's Datab  19.99
```

```
. . .
172-32-1176   PS3333    Prolonged Data Deprivation  19.99
807-91-6654   TC3218    Onions, Leeks, and Garlic:  20.95
(11 rows affected)
```

The two methods of writing joins, ANSI or Transact-SQL, are equivalent. For example, there is no difference between the result sets produced by the following queries:

```
select title_id, pub_name
from titles, publishers
where titles.pub_id = publishers.pub_id
```

and:

```
select title_id, pub_name
from titles left join publishers
on titles.pub_id = publishers.pub_id
```

An inner join can be part of an `update` or `delete` statement. For example, the following query multiplies the price for all the titles published in California by 1.25:

```
begin tran
```

```
update titles
  set price = price * 1.25
  from titles inner join publishers
  on titles.pub_id = publishers.pub_id
  and publishers.state = "CA"
```

Related Information

ANSI outer joins [page 341]
How Joins are Structured [page 322]

11.12.4.2.1 The Join Table of an Inner Join

An ANSI join specifies which tables or views to join in the query. The table references that are specified in the ANSI join make up the joined table.

For example, the join table of the following query includes the `title`, `price`, `advance`, and `royaltyper` columns:

```
select title, price, advance, royaltyper
from titles inner join titleauthor
on titles.title_id = titleauthor.title_id
```

```
title          price     advance      royaltyper
-----------    -------   ----------   ----------
The Busy...    19.99     5,000.00     40
The Busy...    19.99     5,000.00     60
. . .
```

```
Sushi, A...    14.99     8,000.00     30
Sushi, A...    14.99     8,000.00     40
(25 rows affected)
```

If a joined table is used as a table reference in an ANSI inner join, it becomes a nested inner join. ANSI nested inner joins follow the same rules as ANSI outer joins.

A query can reference a maximum of 250 user tables (or 46 worktables) on each side of a union, including:

- Base tables or views listed in the `from` clause
- Each correlated reference to the same table (self-join)
- Tables referenced in subqueries
- Base tables referenced by the views or nested views
- Tables being created with `into`

11.12.4.2.2 The on Clause of an ANSI Inner Join

The `on` clause of an ANSI inner join specifies the conditions used when the tables or views are joined. Although you can use a join on any column of a table, performance may be better if the columns are indexed.

Often, you must use qualifiers (table or correlation names) to uniquely identify the columns and the tables to which they belong. For example:

```
from titles t left join titleauthor ta
on t.title_id = ta.title_id
```

This `on` clause eliminates rows from both tables where there is no matching `title_id`.

The `on` clause often compares the ANSI joins tables, as in the third and fourth line of this query:

```
select title, price, pub_name
from titles inner join publishers
on titles.pub_id = publishers.pub_id
and total_sales > 300
```

The join restriction specified in this `on` clause removes all rows from the join table that do not have sales greater than 300. The `on` clause can include an `and` qualifier to furthor specify search arguments, as illustrated in the fourth line of the query.

ANSI inner joins restrict the result set similarly whether the condition is placed in the `on` clause or the `where` clause (unless they are nested in an outer join). That is, these two queries produce the same result sets:

```
select stor_name, stor_address, ord_num, qty
from salesdetail inner join stores
on salesdetail.stor_id = stores.stor_id
where qty > 3000
```

and:

```
select stor_name, stor_address, ord_num, qty
from salesdetail inner join stores
on salesdetail.stor_id = stores.stor_id
and qty > 3000
```

A query is usually more readable if the restriction is placed in the `where` clause; this explicitly tells users which rows of the join table are included in the result set.

Related Information

Self-Joins and Correlation Names [page 329]

11.12.5 ANSI outer joins

Joins that produce a joined table that includes all rows from the outer table, regardless of whether or not the `on` clause produces matching rows, are called outer joins.

Inner joins and equijoins produce a result set that includes only the rows from the tables where there are matching values in the join clause. There are times, however, when you want to include not only the matching rows, but also the rows from one of the tables where there are no matching rows in the second table. This type of join is an outer join. In an outer join, rows that do not meet the `on` clause conditions are included in the joined table with null-supplied values for the inner table of the outer join. The inner table is also referred to as the null-supplying member.

SAP recommends that your applications use ANSI outer joins because they unambiguously specify whether the `on` or `where` clause contains the predicate.

> **Note**
>
> Queries that contain ANSI outer joins cannot contain Transact-SQL outer joins, and vice versa. However, a query with ANSI outer joins can reference a view that contains a Transact-SQL outer join, and vice versa.

ANSI outer join syntax is:

```
select <select_list>
     from <table1> {left | right} [outer] join <table2>
     on <predicate>
     [<join restriction>]
```

Left joins retain all the rows of the table reference listed on the left of the join clause; right joins retain all the rows of the table reference on the right of the join clause. In left joins, the left table reference is referred to as the outer table, or row-preserving table.

The following example determines the authors who live in the same city as their publishers:

```
select au_fname, au_lname, pub_name
from authors left join publishers
on authors.city = publishers.city
```

```
au_fname   au_lname    pub_name
---------  ----------  -------------
Johnson    White       NULL
Marjorie   Green       NULL
Cheryl     Carson      Algodata Infosystems
. . .
```

```
Abraham    Bennet      Algodata Infosystems
. . .
Anne       Ringer      NULL
Albert     Ringer      NULL
(23 rows affected)
```

The result set contains all the authors from the `authors` table. The authors who do not live in the same city as their publishers produce null values in the `pub_name` column. Only the authors who live in the same city as their publishers, Cheryl Carson and Abraham Bennet, produce a non-null value in the `pub_name` column.

You can rewrite a left outer join as a right outer join by reversing the placement of the tables in the `from` clause. Also, if the `select` statement specifies "select *", you must write an explicit list of all the column names, otherwise, the columns in the result set may not be in the same order for the rewritten query.

Here is the previous example rewritten as a right outer join, which produces the same result set as the left outer join above:

```
select au_fname, au_lname, pub_name
from publishers right join authors
on authors.city = publishers.city
```

Related Information

Transact-SQL Outer Joins [page 348]

11.12.5.1 Placement of the Predicate in the on or where Clause

The result set of an ANSI outer join depends on whether you place the restriction in the `on` or the `where` clause.

The `on` clause defines the result set of a joined table and which rows of this joined table have null-supplied values; the `where` clause defines which rows of the joined table are included in the result set.

Whether you use an `on` or a `where` clause in your join condition depends on what you want your result set to include. The following examples may help you decide whether to place the predicate in the `on` or the `where` clause.

Predicate Restrictions on an Outer Table

The following query places a restriction on the outer table in the `where` clause. Because the restriction is applied to the result of the outer join, it removes all the rows for which the condition is not true:

```
select title, titles.title_id, price, au_id
from titles left join titleauthor
on titles.title_id = titleauthor.title_id
```

```
where titles.price > $20.00
```

```
title                title_id price      au_id
------------------- -------- ---------- -----------------
But Is It User F... PC1035        22.95 238-95-7766
Computer Phobic ... PS1372        21.59 724-80-9391
Computer Phobic ... PS1372        21.59 756-30-7391
Onions, Leeks, a... TC3218        20.95 807-91-6654
(4 rows affected)
```

Four rows meet the criteria and only these rows are included in the result set.

However, if you move this restriction on the outer table to the `on` clause, the result set includes all the rows that meet the `on` clause condition. Rows from the outer table that do not meet the condition are null-extended:

```
select title, titles.title_id, price, au_id
from titles left join titleauthor
on titles.title_id = titleauthor.title_id
and titles.price > $20.00
```

```
title                   title_id   price   au_id
--------------------    ---------  ------  ---------------
The Busy Executive's    BU1032     19.99   NULL
Cooking with Compute    BU1111     11.95   NULL
You Can Combat Compu    BU2075      2.99   NULL
Straight Talk About     BU7832     19.99   NULL
Silicon Valley Gastro   MC2222     19.99   NULL
The Gourmet Microwave   MC3021      2.99   NULL
The Psychology of Com   MC3026      NULL   NULL
But Is It User Friend   PC1035     22.95   238-95-7766
Secrets of Silicon Va   PC8888     20.00   NULL
Net Etiquette           PC9999      NULL   NULL
Computer Phobic and     PS1372     21.59   724-80-9391
Computer Phobic and     PS1372     21.59   756-30-7391
Is Anger the Enemy?     PS2091     10.95   NULL
Life Without Fear       PS2106      7.00   NULL
Prolonged Data Depri    PS3333     19.99   NULL
Emotional Security:     PS7777      7.99   NULL
Onions, Leeks, and Ga   TC3218     20.95   807-91-6654
Fifty Years in Buckin   TC4203     11.95   NULL
Sushi, Anyone?          TC7777     14.99   NULL
(19 rows affected)
```

Moving the restriction to the `on` clause added 15 null-supplied rows to the result set.

Generally, if your query uses a restriction on an outer table, and you want the result set to remove only the rows for which the restriction is false, you should probably place the restriction in the `where` clause to limit the rows of the result set. Outer table predicates are not used for index keys if they are in the `on` clause.

Whether you place the restriction on an outer table in the `on` or `where` clause ultimately depends on the information you need the query to return. If you want the result set to include only the rows for which the restriction is true, place the restriction in the `where` clause. However, if the result set must include all the rows of the outer table, regardless of whether they satisfy the restriction, place the restriction in the `on` clause.

Restrictions on an Inner Table

The following query includes a restriction on an inner table in the `where` clause:

```
select title, titles.title_id, titles.price, au_id
from titleauthor left join titles
on titles.title_id = titleauthor.title_id
where titles.price > $20.00
```

```
title            title_id   price    au_id
-------------    --------   -----    -----------
But Is It U...   PC1035     22.95    238-95-7766
Computer Ph...   PS1372     21.59    724-80-9391
Computer Ph...   PS1372     21.59    756-30-7391
Onions, Lee...   TC3218     20.95    807-91-6654
(4 rows affected)
```

Because the restriction of the `where` clause is applied to the result set after the join is made, all the rows for which the restriction is not true are removed from the result set. In other words, the `where` clause is not true for all null-supplied values and removes them. A join that places its restriction in the `where` clause is effectively an inner join.

However, if you move the restriction to the `on` clause, it is applied during the join and is utilized in the production of the joined table. In this case, the result set includes all the rows of the inner table for which the restriction is true, plus all the rows of the outer table, which are null-extended if they do not meet the restriction criteria:

```
select title, titles.title_id, price, au_id
from titleauthor left join titles
on titles.title_id = titleauthor.title_id
and price > $20.00
```

```
title      title_id   price           au_id
---------  ---------  -----------     -----------
NULL       NULL       NULL            172-32-1176
NULL       NULL       NULL            213-46-8915
. . .
Onions,    TC3218     20.95           807-91-6654
. . .
NUL        NULL       NULL            998-72-3567
NULL       NULL       NULL            998-72-3567
(25 rows affected)
```

This result set includes 21 rows that the previous example did not include.

Generally, if your query requires a restriction on an inner table (for example “and price > $20.00” in query above), place the condition in the `on` clause; this preserves the rows of the outer table. If you include a restriction for an inner table in the `where` clause, the result set might not include the rows of the outer table.

Like the criteria for the placement of a restriction on an outer table, whether you place the restriction for an inner table in the `on` or `where` clause ultimately depends on the result set you want. If you are interested only in the rows for which the restriction is true, place the restriction in the `where` clause. However, if you require the result set to include all the rows of the outer table, regardless of whether they satisfy the restriction, place the restriction in the `on` clause.

Restrictions Included in Both an Inner and Outer Tables

The restriction in the `where` clause of the following query includes both the inner and outer tables:

```
select title, titles.title_id, price, price*qty, qty
from salesdetail left join titles
on titles.title_id = salesdetail.title_id
where price*qty > $30000.00
```

```
title                title_id   price  qty
-----------------    --------   -----  ---------  -----
Silicon Valley Ga    MC2222     19.99  40,619.68  2032
But Is It User Fr    PC1035     22.95  45,900.00  2000
But Is It User Fr    PC1035     22.95  45,900.00  2000
But Is It User Fr    PC1035     22.95  49,067.10  2138
Secrets of Silico    PC8888     20.00  40,000.00  2000
Prolonged Data De    PS3333     19.99  53,713.13  2687
Fifty Years in Bu    TC4203     11.95  32,265.00  2700
Fifty Years in Bu    TC4203     11.95  41,825.00  3500
(8 rows affected)
```

Placing the restriction in the `where` clause eliminates:

- The rows for which the restriction "price*qty>$30000.0" is false
- The rows for which the restriction "price*qty>$30000.0" is unknown because `price` is null

To keep the unmatched rows of the outer table, move the restriction into the `on` clause:

```
select title, titles.title_id, price, price*qty, qty
from salesdetail left join titles
on titles.title_id = salesdetail.title_id
and price*qty > $30000.00
```

```
title                title_id   price  qty
-----------------    --------   -----  ---------  -----
NULL                 NULL       NULL   NULL       75
NULL                 NULL       NULL   NULL       75
. . .
Secrets of Silico    PC8888     20.00  40,000.00  2000
. . .
NULL                 NULL       NULL   NULL       300
NULL                 NULL       NULL   NULL       400
(116 rows affected)
```

This query retains all 116 rows of the `salesdetail` table in the result set, and null-extends the rows that do not meet the restriction.

Where you place the restriction that includes both the inner and outer table depends on the result set you want. If you are interested in only the rows for which the restriction is true, place the restriction in the `where` clause. However, to include all the rows of the outer table, regardless of whether they satisfy the restriction, place the restriction in the `on` clause.

11.12.5.2 Nested ANSI Outer Joins

Nested outer joins use the result set of one outer join as the table reference for another.

For example:

```
select t.title_id, title, ord_num, sd.stor_id, stor_name
from (salesdetail sd
left join titles t
on sd.title_id = t.title_id)   /*join #1*/
left join stores
on sd.stor_id = stores.stor_id /*join #2*/
```

```
title_id  title         ord_num  stor_id  stor_name
--------  ------------  -------  -------  ------------------------
TC3218    Onions, L...  234518   7896     Fricative Bookshop
TC7777    Sushi, An...  234518   7896     Fricative Bookshop
. . .
TC4203    Fifty Yea...  234518   6380     Eric the Read Books
MC3021  The Gourmet...  234518   6380     Eric the Read Books
(116 rows affected)
```

In this example, the joined table between the `salesdetail` and `titles` tables is logically produced first and is then joined with the columns of the `stores` table where `salesdetail.stor_id` equals `stores.stor_id`. Semantically, each level of nesting in a join creates a joined table and is then used for the next join.

In the query above, because the first outer join becomes an operator of the second outer join, this query is a left-nested outer join.

This example shows a right-nested outer join:

```
select stor_name, qty, date, sd.ord_num
from salesdetail sd left join (stores /*join #1 */
left join sales on stores.stor_id = sales.stor_id) /*join #2 */
on stores.stor_id = sd.stor_id
where date > "1/1/1990"
```

```
stor_name      qty  date                 ord_num
------------   ---- ------------------   --------------------
News & Brews   200  Jun 13 1990 12:00AM  NB-3.142
News & Brews   250  Jun 13 1990 12:00AM  NB-3.142
News & Brews   345  Jun 13 1990 12:00AM  NB-3.142
. . .
Thoreau Read   1005 Mar 21 1991 12:00AM  ZZ-999-ZZZ-999-0A0
Thoreau Read   2500 Mar 21 1991 12:00AM  AB-123-DEF-425-1Z3
Thoreau Read   4000 Mar 21 1991 12:00AM  AB-123-DEF-425-1Z3
```

In this example, the second join (between the `stores` and the `sales` tables) is logically produced first, and is joined with the `salesdetail` table. Because the second outer join is used as the table reference for the first outer join, this query is a right-nested outer join.

If the `on` clause for the first outer join (“from salesdetail. . .”) fails, it supplies null values to both the `stores` and `sales` tables in the second outer join.

Parentheses in Nested Outer Joins

Nested outer joins produce the same result set with or without parenthesis. Large queries with many outer joins can be much more readable for users if the joins are structured using parentheses.

The on Clause in Nested Outer Joins

The placement of the `on` clause in a nested outer join determines which join is logically processed first. Reading from left to right, the first `on` clause is the first join to be defined.

In this example, the position of the `on` clause in the first join (in parentheses) indicates that it is the table reference for the second join, so it is defined first, producing the table reference to be joined with the `authors` table:

```
select title, price, au_fname, au_lname
from (titles left join titleauthor
on titles.title_id = titleauthor.title_id)   /*join #1*/
left join authors
on titleauthor.au_id = authors.au_id      /*join #2*/
and titles.price > $15.00
```

```
title            price      au_fname      au_lname
---------------  ---------  ------------  -------------
The Busy Exe...  19.99      Marjorie      Green
The Busy Exe...  19.99      Abrahame      Bennet
. . .
Sushi, Anyon...  14.99      Burt          Gringlesby
Sushi, Anyon...  14.99      Akiko         Yokomoto
(26 rows affected)
```

However, if the `on` clauses are in different locations, the joins are evaluated in a different sequence, but still produce the same result set (this example is for explanatory purposes only; if joined tables are logically produced in a different order, it is unlikely that they will produce the same result set):

```
select title, price, au_fname, au_lname
from titles left join
(titleauthor left join authors
on titleauthor.au_id = authors.au_id)        /*join #2*/
on titles.title_id = titleauthor.title_id    /*join #1*/
and au_lname like"Yokomoto"
```

```
title                     price        au_fname          au_lname
----------------------    -------      -----------       -----------
The Busy Executive’s      19.99        Marjorie          Green
The Busy Executive’s      19.99        Abraham           Bennet
. . .
Sushi, Anyone?            14.99        Burt              Gringlesby
Sushi, Anyone?            14.99        Akiko             Yokomoto
(26 rows affected)
```

The position of the `on` clause of the first join (the last line of the query) indicates that the second left join is a table reference of the first join, so it is performed first. That is, the result of the second left join is joined with the `titles` table.

11.12.6 Transact-SQL Outer Joins

Transact-SQL includes syntax for both left and right outer joins. The left outer join, `*=`, selects from the first table all rows that meet the statement's restrictions. The second table generates values if there is a match on the join condition. Otherwise, the second table generates null values.

For example, this left outer join lists all authors and finds the publishers (if any) in their city:

```
select au_fname, au_lname, pub_name
from authors, publishers
where authors.city *= publishers.city
```

The right outer join, `=*`, selects all rows from the second table that meet the statement's restrictions. The first table generates values if there is a match on the join condition. Otherwise, the first table generates null values.

> **Note**
>
> You cannot include a Transact-SQL outer join in a `having` clause.

A table is either an inner or an outer member of an outer join. If the join operator is `*=`, the second table is the inner table; if the join operator is `=*`, the first table is the inner table. You can compare a column from the inner table to a constant as well as using it in the outer join. For example, to find out which `title` has sold more than 4000 copies:

```
select qty, title from salesdetail, titles
where qty > 4000
and titles.title_id *= salesdetail.title_id
```

However, the inner table in an outer join cannot also participate in a regular join clause.

An earlier example used a join to find the names of authors who live in the same city as a publisher returned two names: Abraham Bennet and Cheryl Carson. To include all the authors in the results, regardless of whether a publisher is located in the same city, use an outer join. Here is what the query and the results of the outer join look like:

```
select au_fname, au_lname, pub_name
from authors, publishers
where authors.city *= publishers.city
```

```
au_fname     au_lname         pub_name
---------    --------------   ---------------
Johnson      White            NULL
Marjorie     Green            NULL
Cheryl       Carson           Algodata Infosystems
Michael      O'Leary          NULL
Dick         Straight         NULL
Meander      Smith            NULL
Abraham      Bennet           Algodata Infosystems
Ann          Dull             NULL
Burt         Gringlesby       NULL
Chastity     Locksley         NULL
Morningstar  Greene           NULL
Reginald     Blotche-Halls    NULL
Akiko        Yokomoto         NULL
Innes        del Castillo     NULL
Michel       DeFrance         NULL
Dirk         Stringer         NULL
Stearns      MacFeather       NULL
```

```
Livia        Karsen           NULL
Sylvia       Panteley         NULL
Sheryl       Hunter           NULL
Heather      McBadden         NULL
Anne         Ringer           NULL
Albert       Ringer           NULL

(23 rows affected)
```

The comparison operator `*=` distinguishes the outer join from an ordinary join. This left outer join tells SAP ASE to include all the rows in the `authors` table in the results, whether or not there is a match on the `city` column in the `publishers` table. The results show no matching data for most of the authors listed, so these rows contain NULL in the `pub_name` column.

The right outer join is specified with the comparison operator `=*`, which indicates that all the rows in the second table are to be included in the results, regardless of whether there is matching data in the first table.

Substituting this operator in the outer join query shown earlier gives this result:

```
select au_fname, au_lname, pub_name
from authors, publishers
where authors.city =* publishers.city
```

```
au_fname     au_lname    pub_name
---------    ---------   ---------------
NULL         NULL        New Age Books
NULL         NULL        Binnet & Hardley
Cheryl       Carson      Algodata Infosystems
Abraham      Bennet      Algodata Infosystems

(4 rows affected)
```

You can further restrict an outer join by comparing it to a constant. This means that you can narrow it to precisely the values you want to see and use the outer join to list the rows that do not include the specified values. Look at the equijoin first and compare it to the outer join. For example, to find out which titles had a sale of more than 500 copies from any store, use this query:

```
select distinct salesdetail.stor_id, title
from titles, salesdetail
where qty > 500
and salesdetail.title_id = titles.title_id
```

```
stor_id
title

------- ---------------------------------------------
5023    Sushi, Anyone?
5023    Is Anger the Enemy?
5023    The Gourmet Microwave
5023    But Is It User Friendly?
5023    Secrets of Silicon Valley
5023    Straight Talk About Computers
5023    You Can Combat Computer Stress!
5023    Silicon Valley Gastronomic Treats
5023    Emotional Security: A New Algorithm
5023    The Busy Executive’s Database Guide
5023    Fifty Years in Buckingham Palace Kitchens
5023    Prolonged Data Deprivation: Four Case Studies
5023    Cooking with Computers: Surreptitious Balance Sheets
7067    Fifty Years in Buckingham Palace Kitchens
```

```
(14 rows affected)
```

You can also use an outer join query to show the titles that did not have a sale of more than 500 copies in any store:

```
select distinct salesdetail.stor_id, title
from titles, salesdetail
where qty > 500
and salesdetail.title_id =* titles.title_id
```

```
stor_id     title

-------   -------------------------------------------
NULL    Net Etiquette
NULL    Life Without Fear
5023    Sushi, Anyone?
5023    Is Anger the Enemy?
5023    The Gourmet Microwave
5023    But Is It User Friendly?
5023    Secrets of Silicon Valley
5023    Straight Talk About Computers
NULL    The Psychology of Computer Cooking
5023    You Can Combat Computer Stress!
5023    Silicon Valley Gastronomic Treats
5023    Emotional Security: A New Algorithm
5023    The Busy Executive's Database Guide
5023    Fifty Years in Buckingham Palace Kitchens
7067    Fifty Years in Buckingham Palace Kitchens
5023    Prolonged Data Deprivation: Four Case Studies
5023    Cooking with Computers: Surreptitious Balance Sheets
NULL    Computer Phobic and Non-Phobic Individuals:
        Behavior Variations
NULL    Onions, Leeks, and Garlic: Cooking Secrets of the
        Mediterranean

 (19 rows affected)
```

You can restrict an inner table with a simple clause. The following example lists the authors who live in the same city as a publisher, but excludes the author Cheryl Carson, who would normally be listed as an author living in a publisher's city:

```
select au_fname, au_lname, pub_name
from authors, publishers
where authors.city =* publishers.city
and authors.au_lname != "Carson"
```

```
au_fname     au_lname    pub_name
---------    ---------   ---------------
NULL         NULL        New Age Books
NULL         NULL        Binnet & Hardley
Abraham      Bennet      Algodata Infosystems

(3 rows affected)
```

11.12.6.1 Outer Joins and Aggregate Extended Columns

An outer join and an aggregate extended column, if you use them together, and if the aggregate extended column is a column from the inner table of the outer join, cause the result set of a query to equal the result set of the outer join.

An outer join connects columns in two tables by using the SAP outer join operator, =* or *=. These symbols are Transact-SQL extension syntax. They are not ANSI SQL symbols, and "outer join" is not a keyword in Transact-SQL. This section refers only to SAP ASE syntax.

The column specified on the side of the asterisk is the outer column from the outer table, for the purposes of the outer join.

An aggregate extended column, although it uses aggregate functions (`max`, `min`), is not included in the `group by` clause of a query.

For example, to create an outer join for which the result contains a null-supplied row, enter:

```
select publishers.pub_id, titles.price
from publishers, titles
where publishers.pub_id *= titles.pub_id
and titles.price > 20.00
pub_id price
------ ----------------- ------
0736 NULL
0877 20.95
0877 21.59
1389 22.95
(4 rows affected)
```

Similarly, to create an outer join and aggregate column for which the result contains a null-supplied row, enter:

```
select publishers.pub_id, max(titles.price)
from publishers, titles
where publishers.pub_id *= titles.pub_id
and titles.price > 20.00
group by publishers.pub_id
pub_id
--------------- --------
0736 NULL
0877 21.59
1389 22.95
(3 rows affected)
```

To create an outer join and aggregate column with an aggregate extended column, for which the result contains a null-supplied row, enter:

```
select publishers.pub_id, titles.title_id,
max(titles.price)
from publishers, titles
where publishers.pub_id *= titles.pub_id
and titles.price > 20.00
group by publishers.pub_id
--------------- --------
....
(54 rows affected)
```

11.13 Relocated Joins

Joins between local and remote tables can be relocated to a remote server. The remote system executes the join using a dynamically created proxy table referring back to the local table, avoiding a significant amount of network traffic.

This is an example of how to use relocated join:.

A join between local table `ls1` and remote table `rl1` results in this query being sent to the remote server:

```
select a,b,c
from localserver.testdb.dbo.ls1 t1, rl1 t2
where t1.a = t2.a
```

The statement sent to the remote server contains a fully qualified reference back to the local table on the local system. The remote server either dynamically creates a temporary proxy table definition for this table, or uses an existing proxy table with a matching mapping. The remote server then executes the join and returns the result set back to the local server.

See the *Performance and Tuning Series: Query Processing and Abstract Plans*.

11.13.1 Configuring Relocated Joins

You must specifically enable relocated joins for each remote server involved. The remote server must be able to connect back to the local server using Component Integration Services (CIS).

Context

To configure relocated joins:

Procedure

1. Use `sp_serveroption` to enable relocated joins to be sent to that remote server:

   ```
   sp_serveroption servername, "relocated joins",true
   ```

2. On the remote server, verify that:
   - The remote server has an interface entry for the local server.
   - There is a `sysservers` entry (added via `sp_addserver`).
   - External logins have been configured.

3. If you are using dynamically created proxy tables, they are created in `tempdb` when a relocated join is received. Enable `ddl in tran` to ensure that `tempdb` allows proxy tables:

```
sp_dboption tempdb,"ddl in tran",true
```

11.14 How Null Values Affect Joins

Null values in tables or views being joined never match each other. Since `bit` columns do not permit null values, a value of 0 appears in an outer join when there is no match for a `bit` column in the inner table.

The result of a join of null with any other value is null. Because null values represent unknown or inapplicable values, Transact-SQL has no basis to match one unknown value to another.

You can detect the presence of null values in a column from one of the tables being joined only by using an outer join. Each table has a null in the column that will participate in the join. A left outer join displays the null value in the first table.

Table t1	
a	b
1	one
NULL	three
4	join4

Table t2	
c	d
NULL	two
4	four

Table `t1` has two columns, “a” and “b.” Table `t2` has two columns, “c” and “d,” and so on. Here is the left outer join:

```
select *
from t1, t2
where a *= c
```

```
a            b       c            d
-----------  ------  -----------  ------
           1  one            NULL  NULL
        NULL  three          NULL  NULL
           4  join4             4  four

(3 rows affected)
```

The results make it difficult to distinguish a null in the data from a null that represents a failure to join. When null values are present in data being joined, it is usually preferable to omit them from the results by using a regular join.

11.15 Determine Which Table Columns to Join

Use `sp_helpjoins` to obtain a list of the columns in two tables or views that are likely join candidates.

The syntax is:

```
sp_helpjoins <table1>,< table2>
```

For example, to find the likely join columns between `titleauthor` and `titles`, use:

```
sp_helpjoins titleauthor, titles
```

```
first_pair
----------------------------- --------------------
title_id                       title_id
```

The column pairs that `sp_helpjoins` shows come from two sources. First, `sp_helpjoins` checks the `syskeys` table in the current database to see if any foreign keys have been defined on the two tables with `sp_foreignkey`, and then checks to see if any common keys have been defined on the two tables with `sp_commonkey`. If it does not find any common keys there, the procedure applies less restrictive criteria to identify any keys that may be reasonably joined. It checks for keys with the same user datatypes, and if that fails, for columns with the same name and datatype.

See the *Reference Manual: Procedures*.

12 Managing Data

After you create a database, tables, and indexes, you can put data into the tables and work with it—adding, changing, transferring, and deleting as necessary.

The commands you use to add, change, or delete data are called data modification statements. These commands include:

- `insert` – adds new rows to a table.
- `update` – changes existing rows in a table.
- `writetext` – adds or changes `text`, `unitext`, and `image` data without writing lengthy changes in the system's transaction log.
- `delete` – removes specific rows from a table.
- `truncate table` – removes all rows from a table.

See the *Reference Manual: Commands*.

You can also add data to a table by transferring it from a file using the bulk-copy utility program, `bcp`. See the *Utility Guide*.

You can use `insert`, `update`, or `delete` to modify data in one table per statement. Transact-SQL enhancements to these commands let you base modifications on data in other tables, and even other databases.

The data modification commands also work on views, with some restrictions.

Database owners and the owners of database objects can use the `grant` and `revoke` commands to specify the users who are allows to execute data modification commands.Permissions or privileges can be granted to individual users, groups, or the public for any combination of the data modification commands. Permissions are discussed in, *Managing User Permissions*, in the *System Administration Guide: Volume 1*.

Related Information

Views: Limit Access to Data [page 393]

12.1 Referential Integrity

Keeping data modifications consistent throughout all tables in a database is called referential integrity.

`insert`, `update`, `delete`, `writetext`, and `truncate table` allow you to change data without changing related data in other tables, however, disparities may develop.

One way to manage data consistency is to define referential integrity constraints for the table. Another way is to create special procedures called triggers that take effect when you give `insert`, `update`, and `delete`

commands for particular tables or columns (the `truncate table` command is not caught by triggers or referential integrity constraints).

For example, if you change the `au_id` entry for Sylvia Panteley in the `authors` table, you must also change it in the `titleauthor` table and in any other table in the database that has a column containing that value. If you do not, you cannot find information such as the names of Ms. Panteley's books, because you cannot make joins on her `au_id` column.

To delete data from referential integrity tables, change the referenced tables first and then the referencing table.

Related Information

Triggers: Enforce Referential Integrity [page 583]
Databases and Tables [page 50]

12.2 Transactions

A transaction is data modification that is performed on a database. A copy of the old and new state of each row affected by each data modification statement is written to the transaction log.

This means that if you begin a transaction by issuing the `begin transaction` command, realize you have made a mistake, and roll the transaction back, the database is restored to its previous condition.

> **Note**
>
> You cannot roll back changes made on a remote SAP ASE by means of a remote procedure call (RPC).

However, if `select/into bulkcopy` database option is set to `false`, the default mode of `writetext` does not log the transactions. This prevents the transaction log from filling with the extremely long blocks of data that `text`, `unitext`, and `image` fields may contain. To log changes made with `writetext`, use the `with log` option.

Related Information

Transactions: Maintain Data Consistency and Recovery [page 632]

12.3 Sample Databases

SAP ASE includes the `pubs2` and `pubs3` sample databases, which are used for most examples in the documentation.

SAP suggests that you start with a clean copy of the `pubs2` or `pubs3` database and return it to that state when you are finished. See a system administrator for help in getting a clean copy of either of these databases.

You can prevent any changes you make from becoming permanent by enclosing all the statements you enter inside a transaction, and then aborting the transaction when you are finished. For example, start the transaction by typing:

```
begin tran modify_pubs2
```

This transaction is named `modify_pubs2`. You can cancel the transaction at any time and return the database to the condition it was in before you began the transaction by typing:

```
rollback tran modify_pubs2
```

12.4 Add New Data

Use the `insert` command to add rows to a database.

To add rows using `insert`:

- Use the `values` keyword to specify values for some or all of the columns in a new row. A simplified version of the syntax for the `insert` command using the `values` keyword is:

  ```
  insert <table_name>
       values (<constant1>, <constant2>, ...)
  ```

- You can use a `select` statement in an `insert` statement to pull values from one or more tables (up to a limit of 250 tables, including the table into which you are inserting). A simplified version of the syntax for the `insert` command using a `select` statement is:

  ```
  insert <table_name>
       select <column_list>
       from <table_list>
       where <search_conditions>
  ```

  > **Note**
  >
  > You cannot use a `compute` clause in a `select` statement that is inside an `insert` statement, because statements that include `compute` do not generate normal rows.

  When you add `text`, `unitext`, or `image` values with `insert`, all data is written to the transaction log. You can use the `writetext` command to add these values without logging the long chunks of data that may comprise `text`, `unitext`, or `image` values.

Related Information

Insert Data into Specific Columns [page 358]
Change text, unitext, and image data [page 375]

12.4.1 Add New Rows with Values

Use the `insert` statement to add rows and values for columns in the row.

This `insert` statement adds a new row to the publishers table, giving a value for every column in the row.

```
insert into publishers
values ("1622", "Jardin, Inc.", "Camden", "NJ")
```

The data values are typed in the same order as the column names in the original `create table` statement, that is, first the ID number, then the name, then the city, and, finally, the state. The `values` data is surrounded by parentheses, and all character data is enclosed in single or double quotes.

Use a separate `insert` statement for each row you add.

12.4.2 Insert Data into Specific Columns

You can add data to some columns in a row by specifying only those columns and their data. All other columns that are not included in the column list must be defined to allow null values. The skipped columns can accept defaults. If you skip a column that has a default bound to it, the default is used.

You may want to use this form of the `insert` command to insert all of the values in a row except the `text`, `unitext`, or `image` values, and then use `writetext` to insert the long data values so that these values are not stored in the transaction log. You can also use this form of the command to skip over `timestamp` data.

Adding data in only two columns, for example, `pub_id` and `pub_name`, requires a command like this:

```
insert into publishers (pub_id, pub_name)
values ("1756", "The Health Center")
```

The order in which you list the column names must match the order in which you list the values. The following example produces the same results as the previous one:

```
insert publishers (pub_name, pub_id)
values("The Health Center", "1756")
```

Either of the `insert` statements places “1756” in the identification number column and “The Health Center” in the publisher name column. Since the `pub_id` column in `publishers` has a unique index, you cannot execute both of these `insert` statements; the second attempt to insert a `pub_id` value of “1756” produces an error message.

The following `select` statement shows the row that was added to `publishers`:

```
select *
from publishers
where pub_name = "The Health Center"
```

```
pub_id  pub_name             city    state
------- -----------------    ------  -------
1756    The Health Center    NULL    NULL
```

SAP ASE enters null values in the `city` and `state` columns because no value was given for these columns in the `insert` statement, and the `publisher` table allows null values in these columns.

12.4.2.1 Restrict Column Data: Rules

You can create a rule and bind it to a column or user-defined datatype. Rules govern the kind of data that can or cannot be added.

For example, a rule called `pub_idrule`, which specifies acceptable publisher identification numbers, is bound to the `pub_id` column of the `publishers` table. The acceptable IDs are “1389,” “0736,” “0877,” “1622,” “1756,” or any four-digit number beginning with “99.” If you enter any other number, you see an error message.

When you get this kind of error message, you may want to use `sp_helptext` to look at the definition of the rule:

```
sp_helptext pub_idrule
```

```
---------
        1

(1 row affected)

text
--------------------------------------------------
create rule pub_idrule
as @pub_id in ("1389", "0736", "0877", "1622", "1756")
or @pub_id like "99[0-9][0-9]"

(1 row affected)
```

For more general information on a specific rule, use `sp_help`, or, to find out if any of the columns has a rule, use `sp_help` with a table name as a parameter .

Related Information

Defining Defaults and Rules for Data [page 413]

12.4.2.2 The NULL Character String

Only columns for which NULL has been specified in the `create table` statement and into which you have explicitly entered **NULL**, or into which no data has been entered can contain null values.

Do not enter the character string “NULL” (with quotes) as data for a character column. Use “N/A” or “none” or a similar value instead.

To explicitly insert NULL into a column, use:

```
values({<expression> | null}
     [, {<expression> | null}]...)
```

The following example shows two equivalent `insert` statements. In the first statement, the user explicitly inserts a NULL into column `t1`. In the second, SAP ASE provides a NULL value for `t1` because the user has not specified an explicit column value:

```
create table test
(t1 char(10) null, t2 char(10) not null)
insert test
values (null, "stuff")
insert test (t2)
values ("stuff")
```

NULL Is Not an Empty String

The empty string (“ ”or ‘ ’) is always stored as a single space in variables and column data. This concatenation statement is equivalent to “abc def”, not “abcdef”:

```
"abc" + "" + "def"
```

An empty string is never evaluated as NULL.

12.4.2.3 Insert NULLs into Columns That Do Not Allow Them

To insert data with `select` from a table that has null values in some fields into a table that does not allow null values, you must provide a substitute value for any NULL entries in the original table.

For example, to insert data into an `advances` table that does not allow null values, this example substitutes “0” for the NULL fields:

```
insert advances
select pub_id, isnull(advance, 0) from titles
```

Without the `isnull` function, this command inserts all the rows with non-null values into `advances` and produces error messages for all the rows where the `advance` column in `titles` contains NULL.

If you cannot make this kind of substitution for your data, you cannot insert data containing null values into columns with a NOT NULL specification.

12.4.2.4 Add Rows Without Values in All Columns

You can specify values for only some of the columns in a row. Certain rules will apply when adding rows without values for all columns.

- If a default value exists for the column or user-defined datatype of the column, it is entered.
- If NULL was specified for the column when the table was created and no default value exists for the column or datatype, NULL is entered. See `insert` in the *Reference Manual: Commands*.
- If the column has the IDENTITY property, a unique, sequential value is entered.
- If NULL was not specified for the column when the table was created and no default exists, SAP ASE rejects the row and displays an error message.

Default Exists for Column or Datatype	Column Defined NOT NULL	Column Defined to Allow NULL	Column is IDENTITY
Yes	The default	The default	Next sequential value
No	Error message	NULL	Next sequential value

Use `sp_help` to display a report on a specified table or default, or on any other object listed in the system table `sysobjects`. To see the definition of a default, use `sp_helptext`.

Related Information

Defining Defaults and Rules for Data [page 413]

12.4.2.5 Change a Column's Value to NULL

To set a column value to NULL, use the `update` statement.

```
set <column_name> = {<expression> | null}
[, <column_name> = {<expression> | null}]...
```

For example, to find all rows in which the `title_id` is TC3218 and replace the `advance` with NULL:

```
update titles
set advance = null
where title_id = "TC3218"
```

12.4.2.6 SAP ASE-generated values for IDENTITY columns

When you insert a row into a table with an IDENTITY column, SAP ASE automatically generates the column value. Do not include the name of the IDENTITY column in the column list or its value in the values list.

This `insert` statement adds a new row to the `sales_daily` table. The column list does not include the IDENTITY column, `row_id`:

```
insert sales_daily (stor_id)
values ("7896")
```

Note

You can omit the column name `stor_id`. The server can identify an IDENTITY column and `insert` the next identity value, without the user entering the column name. For example, this table has three columns, but the `insert` statement gives values for two columns, and no column names:

```
create table idtext (a int, b numeric identity, c char(1))
-------------------
(1 row affected)
```

```
insert idtext values(98,"z")
-------------------
(1 row affected
```

```
insert idtest values (99, "v"))
--------------------
```

```
(1 row affected)
```

```
select * from idtest
---------------------
98     1        z
99     2         v
```

```
(2 rows affected)
```

The following statement shows the row that was added to `sales_daily`. SAP ASE automatically generates the next sequential value, 2, for `row_id`:

```
select * from sales_daily
where stor_id = "7896"
```

```
sale_id      stor_id
-------      -------
      1      7896

(1 row affected)
```

12.4.2.7 Explicitly Insert Data into an IDENTITY Column

At times, you may want to insert a specific value into an IDENTITY column. For example, you may want the first row inserted into the table to have an IDENTITY value of 101, rather than 1. Or you may need to reinsert a row that was deleted by mistake.

The table owner can explicitly insert a value into an IDENTITY column. The database owner and system administrator can explicitly insert a value into an IDENTITY column if they have been granted explicit permission by the table owner, or if they are acting as the table owner.

Before inserting the data, set the `identity_insert` option on for the table. You can set `identity_insert` on for only one table at a time in a database within a session.

This example specifies a "seed" value of 101 for the IDENTITY column:

```
set identity_insert sales_daily on
insert sales_daily (syb_identity, stor_id)
values (101, "1349")
```

The `insert` statement lists each column, including the IDENTITY column, for which a value is specified. When the `identity_insert` option is set to on, each `insert` statement for the table must specify an explicit column list. The values list must specify an IDENTITY column value, since IDENTITY columns do not allow null values.

After you set `identity_insert` off, you can insert IDENTITY column values automatically, without specifying the IDENTITY column. Subsequent insertions use IDENTITY values based on the value explicitly specified after you set `identity_insert` on. For example, if you specify 101 for the IDENTITY column, subsequent insertions are 102, 103, and so on.

> **Note**
>
> SAP ASE does not enforce the uniqueness of the inserted value. You can specify any positive integer within the range allowed by the column's declared precision. To ensure that only unique column values are accepted, create a unique index on the IDENTITY column before inserting any rows.

12.4.2.8 Retrieve IDENTITY Column Values with @@identity

Use the `<@@identity>` global variable to retrieve the last value inserted into an IDENTITY column.

The value of `<@@identity>` changes each time an `insert` or `select into` attempts to insert a row into a table. `<@@identity>` does not revert to its previous value if the `insert` or `select into` statement fails, or if the transaction that contains it is rolled back. If the statement affects a table without an IDENTITY column, `<@@identity>` is set to 0.

If the statement inserts multiple rows, `<@@identity>` reflects the last value inserted into the IDENTITY column.

The value for <@@identity> within a stored procedure or trigger does not affect the value outside the stored procedure or trigger. For example:

```
select @@identity
```

```
---------------------------------------
                                   101
```

```
create procedure reset_id as
    set identity_insert sales_daily on
    insert into sales_daily (syb_identity, stor_id)
         values (102, "1349")
    select @@identity
select @@identity
execute reset_id
```

```
---------------------------------------
                                  102
```

```
select @@identity
```

```
---------------------------------------
                                  101
```

12.4.2.9 Reserve a Block of IDENTITY Column Values

The `identity grab size` configuration parameter allows each SAP ASE process to reserve a block of IDENTITY column values for inserts into tables that have an IDENTITY column.

This configuration parameter reduces the number of times an SAP ASE engine must hold an internal synchronization structure when inserting implicit identity values. For example, to set the number of reserved values to 20:

```
sp_configure "identity grab size", 20
```

When a user performs an insert into a table containing an IDENTITY column, SAP ASE reserves a block of 20 IDENTITY column values for that user. Therefore, during the current session, the next 20 rows the user inserts into the table have sequential IDENTITY column values. If a second user inserts rows into the same table while the first user is performing inserts, SAP ASE reserves the next block of 20 IDENTITY column values for the second user.

For example, suppose the following table containing an IDENTITY column has been created, and the `identity grab size` is set to 10:

```
create table my_titles
(title_id   numeric(5,0)    identity,
title          varchar(30)     not null)
```

User 1 inserts these rows into the `my_titles` table:

```
insert my_titles (title)
values ("The Trauma of the Inner Child")insert my_titles (title)
```

```
values ("A Farewell to Angst")
insert my_titles (title)
values ("Life Without Anger")
```

SAP ASE allows user 1 a block of 10 sequential IDENTITY values, for example, `title_id` numbers 1–10.

While user 1 is inserting rows to `my_titles`, user 2 begins inserting rows into `my_titles`. SAP ASE grants user 2 the next available block of reserved IDENTITY values, that is, values 11–20.

If user 1 enters only three titles and then logs off SAP ASE, the remaining seven reserved IDENTITY values are lost. The result is a gap in the table's IDENTITY values. To avoid large gaps in the IDENTITY column, do not set the `identity grab size` too high.

12.4.2.10 Maximum Value of the IDENTITY Column

The maximum value that you can insert into an IDENTITY column is $10^{precision} - 1$. If you do not specify a precision for the IDENTITY column, SAP ASE uses the default precision for `numeric` columns (18 digits).

Once an IDENTITY column reaches its maximum value, `insert` statements return an error that aborts the current transaction. When this happens, use one of the methods discussed below to remedy the problem.

12.4.2.10.1 Modify the Maximum Value of the IDENTITY Column

Alter the maximum value of any IDENTITY column with a modify operation in the `alter table` command.

```
alter table my_titles
modify title_id, numeric (10,0)
```

This operation performs a data copy on a table and rebuilds all the table indexes.

12.4.2.10.2 Creating a New Table with a Larger Precision

You can create a new table that is identical to the old one, except with a larger precision value for the IDENTITY column. If the table contains IDENTITY columns that are used for referential integrity, retain the current numbers for the IDENTITY column values.

Procedure

1. Use `create table` entering a larger precision value for the IDENTITY column.
2. Use `insert into` to copy the data from the old table into to the new one.

12.4.2.10.3 Renumbering the Table IDENTITY Columns with bcp

If the table does not contain IDENTITY columns used for referential integrity, and if there are gaps in the numbering sequence, you can renumber the IDENTITY column to eliminate gaps, which allows more room for insertions.

Procedure

1. From the operating system command line, use `bcp` to copy out the data:

   ```
   bcp pubs2..mytitles out my_titles_file -N -c
   ```

   where `-N` instructs `bcp` not to copy the IDENTITY column values from the table to the host file, and `-c` instructs `bcp` to use character mode.
2. In SAP ASE, create a new table that is identical to the old table.
3. From the operating system command line, use `bcp` to copy the data into the new table:

   ```
   bcp pubs2..mynewtitles in my_titles_file -N -c
   ```

   where `-N` instructs `bcp` to have SAP ASE assign the IDENTITY column values when loading data from the host file, and `-c` instructs `bcp` to use character mode.
4. In SAP ASE, drop the old table, and use `sp_rename` to change the new table name to the old table name.

Results

If the IDENTITY column is a primary key for joins, you may need to update foreign keys in other tables.

By default, when you bulk-copy data into a table with an IDENTITY column, `bcp` assigns each row a temporary IDENTITY column value of 0. As it inserts each row into the table, the server assigns it a unique, sequential IDENTITY column value, beginning with the next available value. To enter an explicit IDENTITY column value for each row, specify the `-E` flag. See the *Utility Guide*.

12.4.3 Add New Rows with select

To add values to a table from one or more other tables, use a `select` clause in the `insert` statement. The `select` clause can insert values into some or all of the columns in a row.

Inserting values for only some columns may be convenient when you want to take some values from an existing table. You can then use `update` to add the values for the other columns.

Before inserting values for some, but not all, columns in a table, make sure that a default exists, or that NULL has been specified for the columns for which you are not inserting values. Otherwise, SAP ASE returns an error message.

When you insert rows from one table into another, the two tables must have compatible structures—that is, the matching columns must be either the same datatypes or datatypes between which SAP ASE automatically converts.

> **Note**
>
> You cannot insert data from a table that allows null values into a table that does not, if any of the data being inserted is null.

If the columns are in the same order in their `create table` statements, you need not specify column names in either table. Suppose you have a table named `newauthors` that contains some rows of author information in the same format as in `authors`. To add to `authors` all the rows in `newauthors`:

```
insert authors
select *
from newauthors
```

To insert rows into a table based on data in another table, the columns in the two tables need not be listed in the same sequence in their respective `create table` statements. You can use either the `insert` or the `select` statement to order the columns so that they match.

For example, suppose the `create table` statement for the `authors` table contained the columns `au_id`, `au_fname`, `au_lname`, and `address`, in that order, and `newauthors` contained `au_id`, `address`, `au_lname`, and `au_fname`. The column sequence must match in the `insert` statement. You could do this by using:

```
insert authors (au_id, address, au_lname, au_fname)
select * from newauthors
```

or:

```
insert authors
select au_id, au_fname, au_lname, address
    from newauthors
```

If the column sequence in the two tables fails to match, SAP ASE either cannot complete the `insert` operation, or completes it incorrectly, putting data in the wrong column. For example, you might get address data in the `au_lname` column.

12.4.3.1 Use Computed Columns

You can use computed columns in a `select` statement inside an `insert` statement.

For example, imagine that a table named `tmp` contains some new rows for the `titles` table, which contains out-of-date data—the `price` figures need to be doubled. A statement to increase the prices and insert the `tmp` rows into `titles` looks like:

```
insert titles
```

```
select title_id, title, type, pub_id, price*2,
    advance, total_sales, notes, pubdate, contract
from tmp
```

You cannot use the `select *` syntax when you perform computations on a column; each column must be named individually in the select list.

12.4.3.2 Insert Data into Some Columns

You can use the `select` statement to add data to some, but not all, columns in a row. Use the `insert` clause to specify the columns to which you want to add data.

For example, some authors in the `authors` table do not have titles and, therefore, do not have entries in the `titleauthor` table. To pull their `au_id` numbers out of the `authors` table and insert them into the `titleauthor` table as placeholders, try this statement:

```
insert titleauthor (au_id)
select au_id
    from authors
    where au_id not in
    (select au_id from titleauthor)
```

This statement is illegal, because a value is required for the `title_id` column. Null values are not permitted and no default is specified. You can enter the dummy value “xx1111” for `titles_id` by using a constant, as follows:

```
insert titleauthor (au_id, title_id)
select au_id, "xx1111"
    from authors
    where au_id not in
    (select au_id from titleauthor)
```

The `titleauthor` table now contains four new rows with entries for the `au_id` column, dummy entries for the `title_id` column, and null values for the other two columns.

12.4.3.3 Insert Data from the Same Table

You can insert data into a table based on other data in the same table. Essentially, this means copying all or part of a row.

For example, you can insert a new row in the `publishers` table that is based on the values in an existing row in the same table. Make sure you follow the rule on the `pub_id` column:

```
insert publishers
select "9999", "test", city, state
    from publishers
```

```
    where pub_name = "New Age Books"
```

```
 (1 row affected)
```

```
select * from publishers
```

```
 pub_id  pub_name              city        state
------- --------------------- -------     ------
0736    New Age Books         Boston      MA
0877    Binnet & Hardley      Washington  DC
1389    Algodata Infosystems  Berkeley    CA
9999    test                  Boston      MA

(4 rows affected)
```

The example inserts the two constants (“9999” and “test”) and the values from the `city` and `state` columns in the row that satisfied the query.

12.5 Create Nonmaterialized, Non-Null Columns

Nonmaterialized columns exist virtually, but are not physically stored in the row. Use nonmaterialized columns the same as any other column, selecting, updating, and referring to them in SQL queries, or using them as index keys.

SAP ASE treats nonmaterialized columns the same way it treats null columns: if a column is not physically present in the row, SAP ASE supplies a default. The default for a nullable column is `null`, but the default for a nonmaterialized column is a user-defined non-NULL value.

Converting a nonmaterialized column to a physically present column is called “instantiating” the column.

12.5.1 Add Nonmaterialized Columns

Use `alter table . . . not materialized` to create nonmaterialized columns.

For example:

```
alter table <table_name>
add <column_name> <datatype> default <constant_expression>
not null [not materialized]
```

See the *Reference Manual: Commands*.

> **Note**
>
> You cannot use the `not materialized` with the `null` parameter.

For example, to add the nonmaterialized column `alt_title` to the titles table, with a default of `aaaaa`, enter:

```
alter table titles
```

```
add alt_title varchar(24) default 'aaaaa'
not null not materialized
```

SAP ASE creates a default for column `<column_name>` using the specified value (if one does not already exist), and inserts an entry in `syscolumns` for the new column, associating that default with the column.

SAP ASE does not alter the table's physical data.

You can combine `alter table` clauses that specify nonmaterialized columns with clauses creating other nonmaterialized columns or creating nullable columns. You cannot combine `alter table` to create nonmaterialized columns with clauses that perform a full data copy (such as `alter table` to drop a column or to add a non-nullable column).

When used in an `alter table` statement, `<constant_expression>` must be a constant (for example, 6), and cannot be an expression. `<constant_expression>` cannot be an expression such as "6+4", a function (such as `getdate`), or the keyword "user" for a column specified in an `alter table` command.

Nonmaterialized columns require a default, and the default cannot be NULL. You can include the default by:

- Explicitly specifying it in the command (for example, `int default 0`), or
- Implicitly supplying it with a user-defined datatype that has a bound default

> **Note**
>
> If you supply an invalid default value that cannot be converted to that column's correct datatype, `alter table` raises an error

Use the `column default cache size` to configure the memory pool for column defaults.

12.5.2 Tables That Already Have Nonmaterialized Columns

SAP ASE immediately instantiates all nonmaterialized columns if you specify a command that includes clauses that require a full data copy to the table. This includes commands such as `reorg rebuild` and `alter table` to add a non-nullable column.

12.5.3 Nonmaterialized Column Storage

Once nonmaterialized columns are instantiated, SAP ASE stores them in the same way as any other row in the table.

When SAP ASE instantiates a fixed-length, nonmaterialized column, the row occupies more space than an equivalent fixed-length column that is materialized, and instantiated nonmaterialized columns require more space than equivalent fixed-length columns. For dataonly-locked (DOL) tables, the column overhead is 2 bytes. For allpages- locked tables, the column overhead is 1 or more bytes, depending on the column's length and its physical placement in the row.

Columns are nonmaterialized only in rows that were present in the table when you altered the table to include nonmaterialized columns. As you update rows, SAP ASE instantiates any columns that are nonmaterialized.

Nonmaterialized columns can be followed by null columns or other nonmaterialized columns, but cannot be followed by materialized data. When you instantiate a column within a row, SAP ASE instantiates nonmaterialized columns that appear before that row. Nonmaterialized columns are also instantiated in new rows added to the table.

When you instantiate a nonmaterialized column, the row becomes larger. When this happens in DOL tables, SAP ASE forwards the row if the containing page does not have enough reserved space to accommodate the expanded row (this is equivalent to replacing a null column with a non-NULL value).

12.5.4 Alter Nonmaterialized Columns

Use `alter table .... replace` to changed the defaults of nonmaterialized columns. However, this command does not change the defaults for columns using the previous default. Instead, `alter table .... replace` changes the defaults for columns that you add or update.

12.5.5 Limitations for Nonmaterialized Columns

`alter table … modify` does not support converting existing materialized columns to nonmaterialized columns. Running `alter table ... modify` against a column instantiates all nonmaterialized columns in the table.

You cannot:

- Use `bit`, `text`, `image`, `unitext`, and Java datatypes in nonmaterialized columns.
- Use nonmaterialized columns as IDENTITY columns.
- Include the `constraint`, `primary key`, `unique`, or `references` clauses when you use `alter table` to add nonmaterialized columns.
- Encrypt nonmaterialized columns.
- Downgrade an SAP ASE server using data rows with nonmaterialized, non-NULL columns to versions earlier than 15.7. You must first run `reorg rebuild` on tables containing these columns to convert them to regular, non-nullable columns, before downgrading your SAP ASE server.

12.6 Change Existing Data

Use the `update` command to change single rows, groups of rows, or all rows in a table. As in all data modification statements, you can change the data in only one table at a time.

`update` specifies the row or rows you want changed and the new data. The new data can be a constant or an expression that you specify, or data pulled from other tables.

If an `update` statement violates an integrity constraint, the update does not take place and an error message is generated. The update is canceled, for example, if it affects the table's IDENTITY column, or if one of the values being added is the wrong datatype, or if it violates a rule that has been defined for one of the columns or datatypes involved.

SAP ASE does not prevent you from issuing an `update` command that updates a single row more than once. However, because of the way that `update` is processed, updates from a single statement do not accumulate. That is, if an `update` statement modifies the same row twice, the second update is not based on the new values from the first update but on the original values. The results are unpredictable, since they depend on the order of processing.

> **Note**
>
> The `update` command is logged. If you are changing large blocks of `text`, `unitext`, or `image` data, try using the `writetext` command, which is not logged. Also, you are limited to approximately 125K per `update` statement.

See the *Reference Manual: Commands*.

Related Information

Views: Limit Access to Data [page 393]
Change text, unitext, and image data [page 375]

12.6.1 Use the set Clause with Update

The `set` clause specifies columns and changed values. The `where` clause determines which rows are to be updated. If you do not include the `where` clause, the specified columns of all the rows are updated with the values provided in the `set` clause.

> **Note**
>
> Before trying the examples in this section, make sure you know how to reinstall the `pubs2` database. See the *Installation Guide* and *Configuration Guide* for your platform for instructions.

For example, if all the publishing houses in the `publishers` table move their head offices to Atlanta, Georgia, update the table using:

```
update publishers
set city = "Atlanta", state = "GA"
```

In the same way, you can change the names of all the publishers to NULL with:

```
update publishers
set pub_name = null
```

You can use computed column values in an update. To double all the prices in the `titles` table, use:

```
update titles
set price = price * 2
```

Since there is no `where` clause, the change in prices is applied to every row in the table.

12.6.1.1 Assign Variables in the set Clause

You can assign variables in the `set` clause of an `update` statement, in the same way you can assign them in a `select` statement. Using variables with `update` reduces lock contention and CPU consumption that can occur when extra `select` statements are used with `update`.

This example uses a declared variable to update the `titles` table:

```
declare @price money
select @price = 0
update titles
    set total_sales = total_sales + 1,
    @price = price
    where title_id = "BU1032"
select @price, total_sales
    from titles
    where title_id = "BU1032"
```

```
                         total_sales
 ------------------------ -----------
                   19.99        4096

(1 row affected)
```

Related Information

Local Variables [page 463]

12.6.2 Use the where Clause with update

The `where` clause specifies which rows are to be updated.

For example, if the author Heather McBadden changes her name to Heather MacBadden:

```
update authors
set au_lname = "MacBadden"
where au_lname = "McBadden"
and au_fname = "Heather"
```

12.6.3 Use the from Clause with update

Use the `from` clause to add data from one or more tables into the table you are updating.

In this example, an author by the name of Dirk Stringer writes a book called *The Psychology of Computer Cooking*. The data was created by inserting new rows into the `titleauthor` table for authors without titles,

filling in the au_id column, and using dummy or null values for the other columns. You can modify his row in the titleauthor table by adding a title identification number for him:

```
update titleauthor
set title_id = titles.title_id
from titleauthor, titles, authors
    where titles.title =
    "The Psychology of Computer Cooking"
    and authors.au_id = titleauthor.au_id
    and au_lname = "Stringer"
```

An update without the au_id join changes all the title_ids in the titleauthor table so that they are the same as *The Psychology of Computer Cooking* 's identification number. If two tables are identical in structure, except one has NULL fields and some null values, and the other has NOT NULL fields, you cannot insert the data from the NULL table into the NOT NULL table with a select. In other words, a field that does not allow NULL cannot be updated by selecting from a field that does, if any of the data is NULL.

As an alternative to the from clause in the update statement, you can use a subquery, which is ANSI-compliant.

12.6.4 Perform updates with joins

You can use update to join columns when the columns being joined have the same or compatible datatypes.

This is an example that joins columns from the titles and publishers tables, doubling the price of all books published in California.

```
update titles
  set price = price * 2
  from titles, publishers
  where titles.pub_id = publishers.pub_id
  and publishers.state = "CA"
```

12.6.5 Update IDENTITY Columns

You can use the syb_identity keyword, qualified by the table name, where necessary, to update an IDENTITY column.

For example, this update statement finds the row in which the IDENTITY column equals 1 and changes the name of the store to “Barney’s”:

```
update stores_cal
set stor_name = "Barney’s"
where syb_identity = 1
```

12.7 Change text, unitext, and image data

Use `writetext` to change `text, unitext`, or `image` values when you do not want to store long text values in the database transaction log.

Generally, do not use the `update` command, because `update` commands are always logged. In its default mode, `writetext` commands are not logged.

> **Note**
>
> To use `writetext` in its default, non-logged state, a system administrator must use `sp_dboption` to set `select into/bulkcopy/pllsort on`. This permits the insertion of non-logged data. After using `writetext`, you must dump the database. You cannot use `dump transaction` after making unlogged changes to the database.

The `writetext` command overwrites any data in the column it affects. The column must already contain a valid text pointer.

Use the `textvalid()` function to check for a valid pointer:

```
select textvalid("blurbs.copy", textptr(copy))
from blurbs
```

There are two ways to create a text pointer:

- `insert` actual data into the `text`, `unitext`, or `image` column
- `update` the column with data or a NULL

An "initialized" `text` column uses 2K of storage, even to store a couple of words. SAP ASE saves space by not initializing text columns when explicit or implicit null values are placed in `text` columns with `insert`. The following code fragment inserts a value with a null text pointer, checks for the existence of a text pointer, and then updates the `blurbs` table. Explanatory comments are embedded in the text:

```
/* Insert a value with a text pointer. This could
** be done in a separate batch session. */
insert blurbs (au_id) values ("267-41-2394")
/* Check for a valid pointer in an existing row.
** Use textvalid in a conditional clause; if no
** valid text pointer exists, update 'copy' to null
** to initialize the pointer. */
if (select textvalid("blurbs.copy", textptr(copy))
    from blurbs
    where au_id = "267-41-2394") = 0
begin
    update blurbs
       set copy = NULL
       where au_id = "267-41-2394"
end
/*
** use writetext to insert the text into the
** column. The next statements put the text
** into the local variable @val, then writetext
** places the new text string into the row
** pointed to by @val. */
declare @val varbinary(16)
select @val = textptr(copy)
    from blurbs
```

```
    where au_id = "267-41-2394"
writetext blurbs.copy @val
    "This book is a must for true data junkies."
```

Related Information

Batches and Control-of-Flow Language [page 437]

12.8 Truncate Trailing Zeros

The `disable varbinary truncation` configuration parameter controls whether trailing zeros are included at the end of `varbinary` and `binary` null data.

By default, `disable varbinary truncation` is off for the server.

When SAP ASE is set to truncate trailing zeros, tables you subsequently create store the `varbinary` data after truncating the trailing zeros. For example, if you create the `test1` table with `disable varbinary truncation` set to 0:

```
create table test1(col1 varbinary(5))
```

Then insert some `varbinary` data with trailing zeros:

```
insert into test1 values(0x12345600)
```

SAP ASE truncates the zeros:

```
select * from test1
```

```
col1
------------
0x123456
```

However, if you drop and re-create table `test1` and set `disable varbinary truncation` to 1 (on) and perform the same steps, SAP ASE does not truncate the zeros:

```
select * from test1
```

```
col1
------------
0x12345600
```

SAP ASE considers data with or without trailing zeros as equal for comparisons (that is, 0x1234 is the same as 0x123400).

Because SAP ASE stores data according to how `disable varbinary truncation` is currently set, tables may have a mix of data with or without trailing zeros, although the datatype does not change:

- If you perform a `select into` to copy data from one table to another, SAP ASE copies the data as it is stored (that is, if `disable varbinary truncation` is turned off, the trailing zeros are truncated). For example, using the tables from the examples above, if you disable varbinary truncation, then select data from table `test1` into table `test2`:

  ```
  sp_configure "disable varbinary truncation", 1
  select * into test2 from test1
  ```

  Then reinsert the same data again:

  ```
  insert into test2 select * from test1
  ```

  Table `test2` does not truncate the trailing zeros because you ran `select into` with `disable varbinary truncation` set to 1, and the target table does not inherit the property from the source table. The data in the target table is truncated or preserved, depending on how the configuration parameters were setn when you ran `select into`:

  ```
  select * from test2
  ```

  ```
  col1
  ------------
  0x12345600
  0x12345600
  ```

- Bulk copy (`bcp`) inserts data according to how `disable varbinary truncation` is set on the column when created.
- You cannot use `alter table` to change the truncation behavior for a specific column. However, columns you add with `alter table` either truncate or preserve trailing zeros according to the value of `disable varbinary truncation`.
  For example, if you create table `test3` and column `c1` with truncate trailing zeros disabled:

  ```
  sp_configure "disable varbinary truncation", 1
  create table test3(c1 varbinary(5))
  insert into test3 values(0x123400)
  ```

  `c1` retains the trailing zeros:

  ```
  select * from test3
  c1
  ----------
  0x123400
  ```

  However, if you enable truncated trailing zeros and use `alter table` to add a new column, `c2`:

  ```
  sp_configure "disable varbinary truncation", 0
  alter table test3 add c2 varbinary(5) null
  insert into test3 values(0x123400, 0x123400)
  ```

  `c2` truncates the trailing zeros:

  ```
  select * from test3
  c1           c2
  ----------   ----------
  0x123400           NULL
  0x123400         0x1234
  ```

Trailing zeros are preserved in:

- Worktables (after `disable varbinary truncation` is set to 1). The first example below includes a worktable in which the trailing zeros are retained, but in the second example, the worktable stores only the first 6 digits:

```
select 0x12345600 union select 0x123456
-------------
0x12345600
select 0x123456 union select 0x12345600
-------------
0x123456
```

- Concatenations. For example:

```
select 0x12345600 + col1, col1 from test1
```

```
                      col1
--------------------      ------------
0x123456001234560000      0x1234560000
0x1234560001234560        0x01234560
0x1234560012345600        0x12345600
0x123456000123456700      0x0123456700
```

- Functions. For example:

```
select bintostr(0x12340000)
```

```
------------
     1234000
```

- `order by` and `group by` queries. For example:

```
select col1 from
(select 0x123456 col1 union all
select 0x12345600 col1) temp1 order by col1
```

```
col1
----------
0x123456
0x12345600
```

> **Note**
>
> If a query includes worktables, you must enable the `disable varbinary truncation` configuration parameter before running the query to ensure SAP ASE performs no truncation.

- Subqueries – trailing zeros are preserved unless the query involves a worktable, in which case the truncation depends on the value of `disable varbinary truncation`.
- Dumps and loads – if the table data you dump includes trailing zeros, the trailing zeros are preserved when you load that data, regardless of the value of `disable varbinary truncation` in the target database.
- Unions (see example for worktables, above).
- `convert`. For example:

```
select convert(binary(5), 0x0000001000)
------------
0x0000001000
```

12.9 Transfer Data Incrementally

The `transfer` command allows you to transfer data incrementally, and, if required, to a different product.

> **Note**
>
> SAP ASE enables the data transfer feature when you purchase, install, and register the in-memory database license, or when you install the RAP product.

Incremental data transfer:

- Lets you export data, including only the data that has changed since a prior transmittal, from SAP ASE tables that are marked for incremental transfer.
- Allows table data to be read without obtaining the usual locks, without guaranteeing any row retrieval order, and without interfering with other ongoing reads or updates.
- Lets you write selected rows to an output file (which can be a named pipe) formatted for a defined receiver: SAP® IQ®, SAP ASE,`bcp` file, or character-coded output. All selected rows are transmitted without encryption, and, by default, any encrypted columns within the row are decrypted before transmittal. The file to which you are writing must be visible to the machine on which SAP ASE is running (the file can be an NFS file that SAP ASE can open as a local file).
- Maintains a history of transmittals for eligible tables, and lets you remove transmittal history when it is no longer wanted.
- Exports data from tables not declared eligible for incremental transfer, subject to certain restrictions.
- Transfers entire rows from indicated tables. You cannot currently select certain columns, select a partition within a table, or transfer results from SQL queries.

12.9.1 Mark Tables for Incremental Transfer

You must mark tables as eligible to participate in incremental transfer.

Any table may be so designated, except system tables and worktables. You can designate eligibility either when you create a table, or at a later time by using `alter table`. You can also use `alter table` to remove a table's eligibility.

In eligible tables:

- A row is transferred if it has changed since the most recent previous transfer, and if any transactions that changed an existing row—or inserted a new row—were committed before the transfer began.
  This requires extra storage for every row, which is implemented by a hidden 8-byte column within the row.
- Additional information is kept for each table transfer. This information includes identifying information for the set and number of rows transmitted, the starting and ending times of the transfer, data formatting for the transfer, and the full path to the destination file.

Removing a table's eligibility removes any per-row changes added to support incremental transfer, and deletes any saved transfer history for that table.

12.9.2 Transfer Tables from a Destination File

Use the `transfer table` command to load data into SAP ASE from a table contained in an external file.

The table you load does not require a unique primary index, unless you are loading data that has changed from data already in the table (you can load new data without any restrictions). However, loading data becomes an issue when a row duplicates data already in the table, and you do not want the data duplicated. To prevent this, a unique primary index allows SAP ASE to find and remove the old row the new row replaces.

The table you load must have a unique index as its primary key (either a clustered index for allpages-locked tables, or a placement index for data-only-locked tables). A unique index allows `transfer` to detect attempts to insert a duplicate key, and converts the internal `insert` command to an `update` command. Without that index, SAP ASE cannot detect duplicate primary keys. Inserting an updated row causes:

- Some or all of the transfer operation to fail if the table has any other unique index and the row being inserted duplicates a key within that index
- The insert to succeed, but the table erroneously contains two or more rows with this primary key

This example transfers the `pubs2.titles` table from the external `titles.tmp` file located in `/SAP/data` into SAP ASE:

```
transfer table titles from '/SAP/data/titles.tmp' for ase
```

You cannot use all the parameters for the `transfer table...from` that you use with `transfer table. . .to`. Parameters that are inappropriate for loading data from a file produce errors and the transfer command stops. Earlier versions of SAP ASE include parameters for the `from` parameter that are reserved for future use, but `transfer` ignores these parameters if you include them with your syntax. The parameters for the `from` parameter are:

- `column_order=<option>` (does not apply to a load using `for ase`; reserved for future use)
- `column_separator=<string>` (does not apply to a load using `for ase`; reserved for future use)
- `encryption={true | false}` (does not apply to a load using `for ase`; reserved for future use)
- `progress=<nnn>`
- `row_separator=<string>` (does not apply to a load using `for ase`; reserved for future use)

12.9.3 Convert SAP ASE Datatypes to SAP IQ

When transferring datatypes to SAP IQ, SAP ASE makes the necessary transformations to convert the data to the IQ format.

SAP ASE		SAP IQ	
Datatype	Size (in bytes)	Datatype	Size (in bytes)
`bigint unsigned bigint`	8	`bigint unsigned bigint`	8
`int unsigned int`	4	`int unsigned int`	4

SAP ASE		SAP IQ	
smallint	2	smallint	2
unsigned smallint	2	int	2
tinyint	1	tinyint	1
numeric(P,S) decimal(P,S)	2 – 17	numeric(P,S) decimal(P,S)	2 – 26
double precision	8	double	8
real	4	real	4
float(P)	4, 8	float(P)	4, 8
money	8	money IQ stores this as numeric(19,4)	16
smallmoney	4	smallmoney IQ stores this as numeric(10,4)	8
bigdatetime datetime	8	datetime	8
smalldatetime	4	smalldatetime	8
date	4	date	4
time	4	time	8
bigtime	8	time	8
char(N)	1 – 16296	char(N)	1 – 16296
char(N) (null)	1 – 16296	char(N) (null)	1 – 16296
varchar(N) (null)	1 – 16296	varchar(N) (null)	1 – 16296
unichar(N)	1– 8148	binary(N*2)	1 – 16296
unichar(N) nullunivarchar(N) (null)	1– 8148	varbinary(N*2) (null)	1 – 16296
binary(N)	1 – 16296	binary(N)	1 – 16296
varbinary(N)	1 – 16296	varbinary(N)	1 – 16296
binary(N) nullvarbinary(N) (null)	1 – 16296	varbinary(N) (null)	1 – 16296

SAP ASE		SAP IQ	
bit	1	bit	1
timestamp	8	varbinary(8) null	8

Consider the following when converting SAP ASE datatypes to datatypes for IQ:

- Define precision and scale the same on IQ and SAP ASE.
- The storage size for `float` is 4 or 8 bytes, depending on precision. If you do not supply a value for precision, SAP ASE stores `float` as `double precision`, but IQ stores it as `real`. SAP ASE does not convert floating-point data to other formats for transfer to IQ. If you must use approximate numeric types, specify them as `double` or `real`, rather than as `float`.
- The maximum length in SAP ASE for a column with datatypes `char`, `unichar`, or `binary` depends on your installation's page size. The maximum size specified in the table is the largest possible column in a 16K page.
- SAP ASE typically requires two bytes per character to store Unicode characters. Because IQ does not include the Unicode datatype, SAP ASE transmits `unichar(N)` to IQ as `binary(N X 2)`. However, SAP ASE does not transform Unicode characters: SAP ASE pads Unicode strings with NULL(0x00) to transfer them to IQ.
- IQ does not have a native Unicode datatype. SAP ASE transfers Unicode strings to IQ as binary data, that is, two bytes long for each Unicode character. For example, `unichar(40)` in SAP ASE converts to `binary(80)` in IQ. IQ cannot display the Unicode data as strings after the transfer.

12.9.4 Store Transfer Information

Transfer information is stored in `spt_TableTransfer` or `monTableTransfer`.

- `spt_TableTransfer` – results from table transfers are stored in, and retrieved from, `spt_TableTransfer`
- `monTableTransfer` – contains historical transfer information for tables, for transfers currently in progress, and for those that are completed

spt_TableTransfer

The results of successful transfers retrieved from the table specified in the `transfer table` command are used as the defaults for subsequent transfers. For example, if you issue this command (including row and column separators):

```
transfer table mytable for csv
```

The next time you transfer table `mytable`, the `transfer` command, by default uses `for csv` and the same row and column separators.

Each database has its own version of `spt_TableTransfer`. The table stores history only for table transfers for tables in its same database that are marked for incremental transfer.

The `max transfer history` configuration parameter controls how many transfer history entries SAP ASE retains in the `spt_TableTransfer` table in each database. See, *Setting Configuration Parameters*, in the *System Administration Guide, Volume 1*.

Database owners use `sp_setup_table_transfer` to create the `spt_TableTransfer` table. `sp_setup_table_transfer` takes no parameters, and works in the current database.

`spt_TableTransfer` stores historical information about both successful and failed transfers. It does not store information for in-progress transfers.

`spt_TableTransfer` is a user table, rather than a system table. It is not created when you create SAP ASE, but SAP ASE automatically creates it in any database that has tables that are eligible for transfer, if you do not use `sp_setup_transfer_table` to manually create it (creating it manually may allow you to avoid unexpected errors that can occur when SAP ASE creates the table automatically).

`sp_help` reports incremental transfer as a table attribute.

The columns in `spt_TableTransfer` are:

Column	Datatype	Description
`end_code`	`unsigned smallint not null`	The transfer's ending status. 0 – success. error code – failure.
`id`	`int not null`	The object ID of the transferred table.
`ts_floor`	`bigint not null`	The beginning transaction timestamp.
`ts_ceiling`	`bigint not null`	The transaction timestamp after which rows are uncommitted, and therefore not transferred.
`time_begin`	`datetime not null`	• The date and time the transfer began, or • The time SAP ASE began to set up the command if the `transfer` fails before implementation. Otherwise, this is the time at which the command sent the first data to the output file.
`time_end`	`datetime not null`	• The date and time the transfer ended, or • If the `transfer` command fails, this is the time of failure. Otherwise, this is the time at which the command finishes sending data and closes the file
`row_count`	`bigint not null`	The number of rows transferred.
`byte_count`	`bigint not null`	The number of bytes written.
`sequence_id`	`int not null`	A number that tracks this transfer, unique to each transfer of a table.

Column	Datatype	Description
`col_order`	`tinyint not null`	A number representing the column order of the output: • 1 – id • 2 – offset • 3 – name • 4 – name_utf8
`output_to`	`tinyint not null`	A number representing the output format: • ○ 1 – ase ○ 2 – bcp ○ 3 – csv ○ 4 – iq
`tracking_id`	`int null`	An optional customer-supplied tracking ID. If you do not use with `tracking_id = nnn`, this column is null.
`pathname`	`varchar (512) null`	The output file name.
`row_sep`	`varchar (64) null`	The row separator string used for `for csv`.
`col_sep`	`varchar(64)` null	The column separator used for `for csv`.

monTableTransfer

`monTableTransfer` table provides:

- Historical transfer information for tables for which SAP ASE currently holds transfer information in memory. This is true for any table accessed since the most recent SAP ASE restart, unless you have not configured SAP ASE memory large enough to hold all current table information that:
- Information about transfers currently in progress and for completed transfers for tables that SAP ASE holds in memory. This includes information for tables that:
  - Are marked for incremental transfer
  - Have been involved in at least one transfer since you restarted SAP ASE.
  - Have descriptions you have not used for other tables. `monTableTransfer` does not search through every database looking for every table that was ever transferred; it searches only an active set of tables that have recent transfers.

See `monTableTransfer` in the *Reference Manual: Tables*.

12.9.5 Exceptions and Errors

Some restrictions apply when transferring data incrementally.

Error messages are generated if you attempt to:

- You cannot use the data transfer utility to transfer data using the bcp format over pipes or using pipes on the Windows platform.
- Transfer a table that does not exist
- Transfer an object that is not a table
- Transfer a table that you do not own and have not been granted permission to transfer, without having `sa_role` privileges
- Decrypt columns during transfer from a table containing encrypted columns, without having specific permission to decrypt those columns
- Transfer `for iq`, when the table contains text or image columns
- Transfer `for ase`, but specify a column order other than `offset`
- Transfer `for bcp`, but specify a column order other than `id`
- Issue an `alter table...set transfer table on` command that names a system catalog

Other reasons for transfer failure include:

- The requested file cannot close.
- You cannot transfer tables with off-row columns (`text`, `unitext`, `image`, and `Java` columns stored off-row).
- Failure to open the file. Make sure the directory exists and SAP ASE has `write` permission in the directory.
- SAP ASE cannot obtain sufficient memory for saved data for each table in a transfer. If this happens, increase the amount of memory available to SAP ASE.

12.9.6 Sample Incremental Transfer

Learn how to transfer data to an external file, change the data in the table, and then use the `transfer` command again to repopulate the table from this external file.

This example transfers data out of, and back into, the same table. However, in a typical user scenario, data would be transferred out of one table and then into a different one.

1. Create the `spt_TableTransfer` table, which stores transfer history:

```
sp_setup_table_transfer
```

2. Configure `max transfer history`. The default is 10, which means that SAP ASE retains 10 successful and 10 unsuccessful transfers for each table marked for incremental transfer. This example changes the value of `max transfer history` from 10 to 5:

```
sp_configure 'max transfer history', 5
```

```
Parameter Name               Default     Memory Used
    Config Value    Run Value              Unit
    Type                      Instance Name
-------------------------     -------     -----------
--------------   --------------------   -------
---------------          ---------------
max transfer history      10              0
5                5                       bytes
dynamic                  NULL
```

3. Create the `transfer_example` table, which has the `transfer` attribute enabled and uses datarow locking:

```
create table transfer_example (
f1 int,
f2 varchar(30),
f3 bigdatetime,
primary key (f1)
) lock datarows
   with transfer table on
```

4. Populate the `transfer_example` table with sample data:

```
set nocount on
declare @i int, @vc varchar(1024), @bdt bigdatetime
select @i = 1
while @i <= 10
begin
   select @vc = replicate(char(64 + @i), 3 * @i)
   select @bdt = current_bigdatetime()
   insert into transfer_example values ( @i, @vc, @bdt )
   select @i = @i + 1
end
set nocount off
```

The script produces this data:

```
select * from transfer_example
order by f1
```

```
f1       f2                              f3
--------- ------------------------------ ----------------------------
   1      AAA                             Jul 17 2009  4:40:14.465789PM
   2      BBBBBB                          Jul 17 2009  4:40:14.488003PM
   3      CCCCCCCCC                       Jul 17 2009  4:40:14.511749PM
   4      DDDDDDDDDDDD                    Jul 17 2009  4:40:14.536653PM
   5      EEEEEEEEEEEEEE                  Jul 17 2009  4:40:14.559480PM
   6      FFFFFFFFFFFFFFFFFF              Jul 17 2009  4:40:14.583400PM
   7      GGGGGGGGGGGGGGGGGGGG            Jul 17 2009  4:40:14.607196PM
   8      HHHHHHHHHHHHHHHHHHHHHHHH        Jul 17 2009 4:40:14.632152PM
   9      IIIIIIIIIIIIIIIIIIIIIIIIIII   Jul 17 2009 4:40:14.655184PM
  10      JJJJJJJJJJJJJJJJJJJJJJJJJJJJJJ  Jul 17 2009 4:40:14.678938PM
```

5. Transfer the `transfer_example` data to an external file using the `for ase` format:

```
transfer table transfer_example
to 'transfer_example-data.ase'
for ase
```

```
(10 rows affected)
```

The data transfer creates this history record in `spt_TableTransfer`:

```
select id, sequence_id, end_code, ts_floor, ts_ceiling, row_count
from spt_TableTransfer
where id = object_id('transfer_example')
```

```
id        sequence_id      end_code     ts_floor       ts_ceiling
row_count
```

```
---------   -----------       --------      -------------
   -------------------
592002109  1                0            0              5309
      10
```

6. Disable the `transfer` attribute from `transfer_example` to demonstrate that the receiving table does not need the `transfer` attribute enabled to receive incremental data (the database must have `select into` enabled before you can run `alter table`):

```
alter table transfer_example
set transfer table off
```

After the `alter table` command runs, `spt_TableTransfer` is empty:

```
select id, sequence_id, end_code, ts_floor, ts_ceiling, row_count
from spt_TableTransfer
where id = object_id('transfer_example')
```

```
id        sequence_id      end_code     ts_floor       ts_ceiling
row_count
---------   -----------       --------      -------------
   -------------------
(0 rows affected
```

7. Update `transfer_example` to set its character data to `no data` and to specify a date and time in its `bigdatetime` column so you can verify the table does not contain the original data.:

```
update transfer_example
set f2 = 'no data',
f3 = 'Jan 1, 1900 12:00:00.000001AM'
```

```
(10 rows affected)
```

After the `update`, `transfer_example` contains this data.

```
select * from transfer_example
order by f1
```

```
f1         f2                              f3
----------- ------------------------------- ---------------------------
1           no data                         Jan  1 1900 12:00:00.000001AM
2           no data                         Jan  1 1900 12:00:00.000001AM
3           no data                         Jan  1 1900 12:00:00.000001AM
4           no data                         Jan  1 1900 12:00:00.000001AM
5           no data                         Jan  1 1900 12:00:00.000001AM
6           no data                         Jan  1 1900 12:00:00.000001AM
7           no data                         Jan  1 1900 12:00:00.000001AM
8           no data                         Jan  1 1900 12:00:00.000001AM
9           no data                         Jan  1 1900 12:00:00.000001AM
10          no data                         Jan  1 1900 12:00:00.000001AM
(10 rows affected)
```

8. Transfer the example data from the external file into `transfer_example`. Even though `transfer_example` is no longer marked for incremental transfer, you can transfer data in to the table. Because it has a unique primary index, the incoming rows replace the existing data and do not create `duplicate key` errors:

```
transfer table transfer_example
```

```
from 'transfer_example-data.ase'
for ase
```

```
(10 rows affected)
```

9. Select all data from `transfer_example` to verify that the incoming data replaced the changed data. The `transfer` replaced the contents of `transfer_example.f2` and `transfer_example.f3` tables with the data originally created for them, which were stored in the `transfer_example-data.ase` output file.

```
select * from transfer_example
order by f1
```

```
f1      f2                              f3
------- ------------------------------- ----------------------------
   1    AAA                             Jul 17 2009 4:40:14.465789PM
   2    BBBBBB                          Jul 17 2009 4:40:14.488003PM
   3    CCCCCCCCC                       Jul 17 2009 4:40:14.511749PM
   4    DDDDDDDDDDDD                    Jul 17 2009 4:40:14.536653PM
   5    EEEEEEEEEEEEEE                  Jul 17 2009 4:40:14.559480PM
   6    FFFFFFFFFFFFFFFFFF              Jul 17 2009 4:40:14.583400PM
   7    GGGGGGGGGGGGGGGGGGGGG           Jul 17 2009 4:40:14.607196PM
   8    HHHHHHHHHHHHHHHHHHHHHHHH        Jul 17 2009 4:40:14.632152PM
   9    IIIIIIIIIIIIIIIIIIIIIIIIIII     Jul 17 2009 4:40:14.655184PM
  10    JJJJJJJJJJJJJJJJJJJJJJJJJJJJJJ  Jul 17 2009 4:40:14.678938PM
```

10. Reenable `transfer` for `transfer_example` so subsequent transfers use, by default, the previous parameters:

```
alter table transfer_example
set transfer table on
```

```
(10 rows affected)
```

12.9.6.1 Replacing Data with New Rows

When using incremental transfer, if you change the key values for some rows, the next `transfer` into the table considers the changed key data rows to be new data, but replaces the data for rows in which keys have not changed.

1. `transfer_example` uses the `f1` column as the primary-key column. SAP ASE uses this column to determine whether an incoming row contains new data, or whether it replaces an existing row.
   For example, if you replace rows with keys 3, 5, and 7 by adding 10 to their respective values:

```
update transfer_example
set f1 = f1 + 10
where f1 in (3,5,7)
```

```
(3 rows affected)
```

   `transfer_example` now includes rows with keys 13, 15, and 17, which `transfer` considers to be new rows. When you transfer the same data into `transfer_example`, `transfer` inserts rows with keys 3, 5, and 7, and retains rows with keys 13, 15, and 17.

```
transfer table transfer_example
```

```
from 'transfer_example-data.ase'
for ase
```

```
(10 rows affected)
```

2. Verify that the data for row 3 is the same as 13, row 5 is the same as row 15, and row 7 is the same as row 17 for `f2` and `f3`:

```
select * from transfer_example
order by f1
```

```
f1     f2                             f3
------- ------------------------------ ----------------------------
1       AAA                             Jul 17 2009  4:40:14.465789PM
2       BBBBBB                          Jul 17 2009  4:40:14.488003PM
3       CCCCCCCCC                       Jul 17 2009  4:40:14.511749PM
4       DDDDDDDDDDDD                    Jul 17 2009  4:40:14.536653PM
5       EEEEEEEEEEEEEEE                 Jul 17 2009  4:40:14.559480PM
6       FFFFFFFFFFFFFFFFFF              Jul 17 2009  4:40:14.583400PM
7       GGGGGGGGGGGGGGGGGGGGG           Jul 17 2009  4:40:14.607196PM
8       HHHHHHHHHHHHHHHHHHHHHHHHH       Jul 17 2009  4:40:14.632152PM
9       IIIIIIIIIIIIIIIIIIIIIIIIIII     Jul 17 2009  4:40:14.655184PM
10      JJJJJJJJJJJJJJJJJJJJJJJJJJJJJJ  Jul 17 2009  4:40:14.678938PM
13      CCCCCCCCC                       Jul 17 2009  4:40:14.511749PM
15      EEEEEEEEEEEEEEE                 Jul 17 2009  4:40:14.559480PM
17      GGGGGGGGGGGGGGGGGGGGG           Jul 17 2009  4:40:14.607196PM
(13 rows affected)
```

3. Transfer `transfer_example` out again: all 13 rows are transferred. SAP ASE views the rows you transferred in with keys 3, 5, and 7 as new since they replaced existing rows (this example uses a tracking ID value of 101):

```
transfer table transfer_example
to 'transfer_example-data-01.ase'
for ase
with tracking_id = 101
```

```
(13 rows affected)
```

4. Modify rows to show that incremental `transfer` transfers only rows that have been modified since the previous transfer (this update affects 3 rows).

```
update transfer_example
set f3 = current_bigdatetime()
where f1 > 10
```

```
(3 rows affected)
```

5. Transfer the table out again to verify that only the 3 changed rows are transferred. You need not specify `for ase`: SAP ASE uses this parameter, which was set in the previous transfer, as the default.

```
transfer table transfer_example
to 'transfer_example-data-02.ase'
with tracking_id = 102
```

```
(3 rows affected)
```

6. View the transfer information using the `tracking_id` from step 3:

```
select id, sequence_id, end_code, ts_floor, ts_ceiling, row_count
```

```
from spt_TableTransfer
where id = object_id('transfer_example')
and tracking_id = 101
```

```
id          sequence_id   end_code    ts_floor             ts_ceiling
row_count
-----------   -----------    --------     --------------------
   --------------------
592002109    3             0           5309                 5716
13
```

12.10 Delete Data

`delete` works for both single-row and multiple-row operations.

The `where` clause specifies which rows are to be removed. When no `where` clause is given in the `delete` statement, all rows in the table are removed. See the *Reference Manual: Commands*.

12.10.1 Use the from Clause with delete

The `from` clause in the second position of a `delete` statement allows you to select data from one or more tables and delete corresponding data from the first-named table.

The rows you select in the `from` clause specify the conditions for the `delete` command.

Suppose that a complex corporate deal results in the acquisition of all the Oakland authors and their books by another publisher. You must immediately remove all these books from the `titles` table, but you do not know their titles or identification numbers. The only information you have is the author's names and addresses.

You can delete the rows in `titles` by finding the author identification numbers for the rows that have Oakland as the town in the `authors` table and using these numbers to find the title identification numbers of the books in the `titleauthor` table. In other words, a three-way join is required to find the rows to delete in the `titles` table.

The three tables are all included in the `from` clause of the `delete` statement. However, only the rows in the `titles` table that fulfill the conditions of the `where` clause are deleted. To remove relevant rows in tables other than `titles`, use separate `delete` statements.

This is the correct statement:

```
delete titles
from authors, titles, titleauthor
where titles.title_id = titleauthor.title_id
and authors.au_id = titleauthor.au_id
and city = "Oakland"
```

The `deltitle` trigger in the `pubs2` database prevents you from actually performing this deletion, because it does not allow you to delete any titles that have sales recorded in the `sales` table.

> **Note**
>
> The optional `from` clause immediately after the delete keyword is included for compatibility with other versions of SQL.

12.10.2 Delete from IDENTITY Columns

You can use the `syb_identity` keyword in a `delete` statement on tables containing an IDENTITY column.

For example, to remove the row for which `row_id` equals 1, use:

```
delete sales_monthly
where syb_identity = 1
```

After you delete IDENTITY column rows, you may want to eliminate gaps in the table's IDENTITY column numbering sequence.

Related Information

Renumbering the Table IDENTITY Columns with bcp [page 366]

12.11 Delete All Rows from a Table

Use `truncate table` to delete all rows in a table. `truncate table` is almost always faster than a `delete` statement with no conditions, because the `delete` logs each change, while `truncate table` logs only the deallocation of entire data pages.

`truncate table` immediately frees all the space occupied by the table's data and indexes. The freed space can then be used by any object. The distribution pages for all indexes are also deallocated. Run `update statistics` after adding new rows to the table.

As with `delete`, a table emptied with `truncate table` remains in the database, along with its indexes and other associated objects, unless you enter a `drop table` command.

You cannot use `truncate table` if another table has rows that reference it through a referential integrity constraint. Delete the rows from the foreign table, or truncate the foreign table and then truncate the primary table.

Related Information

General Rules for Creating Referential Integrity Constraints [page 112]

12.11.1 truncate table Syntax

The `trucate table` command removes all rows from a table.

The syntax for `truncate table` is:

```
truncate table [ [ <database>.]<owner>.]<table_name>
    [ partition <partition_name ]>
```

For example, to remove all the data in `sales`, type:

```
truncate table sales
```

Permission to use `truncate table`, like `drop table`, defaults to the table owner and cannot be transferred.

A `truncate table` command is not caught by a `delete` trigger.

Related Information

Partition Tables and Indexes [page 138]
Triggers: Enforce Referential Integrity [page 583]

13 Views: Limit Access to Data

You can use views to focus, simplify, and customize each user's perception of the tables in a particular database. Views also provide a security mechanism by allowing users access only to the data they require.

A view is a named `select` statement that is stored in a database as an object. A view allows you to display a subset of rows or columns in one or more tables. Use a view by invoking its name in Transact-SQL statements.

A view is an alternative way of looking at the data in one or more tables. For example, suppose you are working on a project that is specific to the state of Utah. You can create a view that lists only the authors who live in Utah:

```
create view authors_ut
as select * from authors
where state = "UT"
```

To display the `authors_ut` view, enter:

```
select * from authors_ut
```

When the authors who live in Utah are added to or removed from the `authors` table, the `authors_ut` view reflects the updated `authors` table.

A view is derived from one or more real tables for which the data is physically stored in the database. The tables from which a view is derived are called its base tables or underlying tables. A view can also be derived from another view.

The definition of a view, in terms of the base tables from which it is derived, is stored in the database. No separate copies of data are associated with this stored definition. The data that you view is stored in the underlying tables.

A view looks exactly like any other database table. You can display it and operate on it almost exactly as you can any other table. There are no restrictions on querying through views and fewer than usual on modifying them.

When you modify the data in a view, you are actually changing the data in the underlying base tables. Conversely, changes to data in the underlying base tables are automatically reflected in the views that are derived from them.

13.1 Advantages of Views

You can use views to focus, simplify, and customize each user's perception of the database; views also provide an easy-to-use security measure.

Views can also be helpful when changes have been made to the structure of a database, but users prefer to work with the structure of the database they are accustomed to.

You can use views to:

- Focus on the data that interests each user, and on the tasks for which that user is responsible. Data that is not of interest to a user can be omitted from the view.
- Define frequently used joins, projections, and selections as views so that users need not specify all the conditions and qualifications each time an operation is performed on that data.
- Display different data for different users, even when they are using the same data at the same time. This is particularly usefull when users of many different interests and skill levels share the same database.

13.2 Security

Through a view, users can query and modify only the data they can see. The rest of the database is invisable and inaccessible.

Use the `grant` and `revoke` commands to restrict each user's access to the database to specified database objects—including views. If a view and all the tables and other views from which it is derived are owned by the same user, that user can grant permission to others to use the view while denying permission to use its underlying tables and views. This is a simple but effective security mechanism. See, *Managing User Permissions*, in the *Security Administration Guide*.

By defining different views and selectively granting permissions on them, users can be restricted to different subsets of data. For example, you can restrict access to:

- A subset of the rows of a base table, that is, a value-dependent subset. For example, you might define a view that contains only the rows for business and psychology books, to keep information about other types of books hidden from some users.
- A subset of the columns of a base table, that is, a value-independent subset. For example, you might define a view that contains all the rows of the `titles` table, except the `royalty` and `advance` columns.
- A row-and-column subset of a base table.
- The rows that qualify for a join of more than one base table. For example, you might define a view that joins the `titles`, `authors`, and `titleauthor` to display the names of the authors and the books they have written. However, this view hides personal data about authors and financial information about the books.
- A statistical summary of data in a base table. For example, through the view `category_price` a user can access only the average price of each type of book.
- A subset of another view or a combination of views and base tables. For example, through the view `hiprice_computer`, a user can access the title and price of computer books that meet the qualifications in the view definition of `hiprice`.

To create a view, a user must be granted `create view` permission by the database owner, and must have appropriate permissions on any tables or views referenced in the view definition.

If a view references objects in different databases, users of the view must be valid users or guests in each of the databases.

If you own an object on which other users have created views, you must be aware of who can see what data through what views. For example: the database owner has granted Harold `create view` permission, and Maude has granted Harold permission to `select` from a table she owns. Given these permissions, Harold can create a view that selects all columns and rows from the table owned by Maude. If Maude revokes permission for Harold to `select` from her table, he can still look at her data through the view he has created.

13.3 Logical Data Independence

Views can shield users from changes in the structure of the real tables if such changes become necessary.

For example, if you restructure the database by using `select into` to split the `titles` table into these two new base tables and then dropping the `titles` table:

```
titletext (title_id, title, type, notes)
```

```
titlenumbers (title_id, pub_id, price, advance, royalty, total_sales, pub_date)
```

You can regenerate the original `titles` table by joining on the `title_id` columns of the two new tables. You can create a view that is a join of the two new tables. You can even name it `titles`.

Any query or stored procedure that refers to the base table `titles` now refers to the view `titles`. `select` operations work exactly as before. Users who retrieve only from the new view need not even know that the restructuring has occurred.

Unfortunately, views provide only partial logical independence. Some data modification statements on the new `titles` are not allowed because of certain restrictions.

13.4 Create Views

You can build views on other views and procedures that reference views. You can define primary, foreign, and common keys on views. However, you cannot associate rules, defaults, or triggers with views or build indexes on them. You cannot create temporary views, or views on temporary tables.

View names must be unique for each user among the already existing tables and views. If you have `set quoted_identifier on`, you can use a delimited identifier for the view.

This example is a view derived from the `titles` table. Suppose you are interested only in books priced higher than $15 and for which an advance of more than $5000 was paid. This straightforward `select` statement finds the rows that qualify:

```
select *
from titles
where price > $15
  and advance > $5000
```

Now, suppose you have a lot of retrieval and update operations to perform on this data. You can combine the conditions shown in the previous query with any command that you issue or you can create a view that displays only the records of interest:

```
create view hiprice
as select *
from titles
where price > $15
  and advance > $5000
```

When SAP ASE receives this command, it stores the `select` statement, which is the definition of the view `hiprice`, in the system table `syscomments`. Entries are also made in `sysobjects` and in `syscolumns` for each column included in the view.

Now, when you display or operate on `hiprice`, SAP ASE combines your statement with the stored definition of `hiprice`. For example, you can change all the prices in `hiprice` just as you can change any other table:

```
update hiprice
set price = price * 2
```

SAP ASE finds the view definition in the system tables and converts the `update` command into the statement:

```
update titles
set price = price * 2
where price > $15
  and advance > $5000
```

In other words, SAP ASE knows from the view definition that the data to be updated is in `titles`. It also increases the prices only in the rows that meet the conditions on the `price` and `advance` columns given in the view definition and those in the update statement.

Having issued the update to `hiprice,` you can see its effect either in the view or in the `titles` table. Conversely, if you had created the view and then issued the second `update` statement, which operates directly on the base table, the changed prices would also be visible through the view.

Updating a view's underlying table in such a way that different rows qualify for the view affects the view. For example, suppose you increase the price of the book *You Can Combat Computer Stress* to $25.95. Since this book now meets the qualifying conditions in the view definition statement, it is considered part of the view.

However, if you alter the structure of a view's underlying table by adding columns, the new columns do not appear in a view that is defined with a `select *` clause unless you drop and redefine the view. This is because the asterisk in the original view definition considers only the original columns.

Related Information

Naming Convention Identifiers [page 29]

13.4.1 create view Syntax

You need not specify any column names in the `create` clause of a view definition statement. SAP ASE gives the columns of the view the same names and datatypes as the columns referred to in the select list of the `select` statement.

The `select` list can be designated by the asterisk (`*`), as in the example, or it can be a full or partial list of the column names in the base tables.

See the *Reference Manual: Commands*.

To build views without duplicate rows, use the `distinct` keyword of the `select` statement to ensure that each row in the view is unique. However, you cannot update `distinct` views.

Here is a view definition statement that makes the name of a column in the view different from its name in the underlying table:

```
create view pub_view1 (Publisher, City, State)
as select pub_name, city, state
from publishers
```

Here is an alternate method of creating the same view but renaming the columns in the `select` statement:

```
create view pub_view2
as select Publisher = pub_name,
City = city, State = state
from publishers
```

The examples of view definition statements in the next section illustrate the rest of the rules for including column names in the `create` clause.

> **Note**
>
> You cannot use local variables in view definitions.

13.4.2 select Statement Usage with create view

The `select` statement in the `create view` statement defines the view. You must have permission to `select` from any objects referenced in the `select` statement of a view you are creating.

You can create a view using more than one table and other views by using a `select` statement of any complexity.

On the `select` statements in a view definition, you cannot:

- Include `order by` or `compute` clauses.
- Include the `into` keyword.
- Reference a temporary table.

After you create a view, the source text describing the view is stored in the `text` column of the `syscomments` system table.

> **Note**
>
> Do not remove this information from `syscomments`. Instead, encrypt the text in `syscomments` with `sp_hidetext`. See the *Reference Manual: Procedures*.

Related Information

Compiled Objects [page 22]

13.4.2.1 View Definition with Projection

You can create a view with all the rows of the `titles` table, but with only a subset of its columns.

```
create view titles_view
as select title, type, price, pubdate
from titles
```

No column names are included in the `create view` clause. The view `titles_view` inherits the column names given in the select list.

13.4.2.2 View Definition with a Computed Column

You can use a view definition statement that creates a view with a computed column generated from the columns `price`, `royalty`, and `total_sales`.

```
create view accounts (title, advance, amt_due)
as select titles.title_id, advance,
(price * royalty /100) * total_sales
from titles, roysched
where price > $15
and advance > $5000
and titles.title_id = roysched.title_id
and total_sales between lorange and hirange
```

There is no name that can be inherited by the column computed by multiplying together `price`, `royalty`, and `total_sales`, so you must include the list of columns in the create clause. The computed column is named `amt_due`. It must be listed in the same position in the `create` clause as the expression from which it is computed is listed in the `select` clause.

13.4.2.3 View Definition with an Aggregate or Built-In Function

You can use a definition that includes an aggregate or built-in function must include column names in the `create` clause.

For example:

```
create view categories1 (category, average_price)
as select type, avg(price)
from titles
group by type
```

If you create a view for security reasons, be careful when using aggregate functions and the `group by` clause. The Transact-SQL extension that does not restrict the columns you can include in the `select` with `group by` may also cause the view to return more information than required. For example:

```
create view categories2 (category, average_price)
as select type, avg(price)
```

```
from titles
where type = "business"
```

You may have wanted the view to restrict its results to “business” categories, but the results have information about other categories.

Related Information

Organize Query Results into Groups: the group by Clause [page 290]

13.4.2.4 View Definition with a Join

You can create a view that is derived from more than one base table.

Here is an example of a view derived from both the `authors` and the `publishers` tables. The view contains the names and cities of the authors that live in the same city as a publisher, along with each publisher’s name and city:

```
create view cities (authorname, acity, publishername, pcity)
as select au_lname, authors.city, pub_name, publishers.city
from authors, publishers
where authors.city = publishers.city
```

13.4.2.5 Views Used with Outer Joins

If you define a view with an outer join, and then query the view with a qualification on a column from the inner table of the outer join, the query behaves as though the qualification were part of the `where` clause of the view, not part of the `on` clause of the outer join in the view.

Thus, the qualification operates only on rows after the outer join is complete. For example, the qualification operates on NULL extended rows if the outer join condition is met, and eliminates rows accordingly.

The following rules determine what types of updates you can make to columns through join views:

- `delete` statements are not allowed on join views.
- `insert` statements are not allowed on join views created `with check option`.
- `update` statements are allowed on join views `with check option`. The update fails if any of the affected columns appears in the `where` clause, in an expression that includes columns from more than one table.
- If you insert or update a row through a join view, all affected columns must belong to the same base table.

13.4.2.6 Views Derived From Other Views

You can define a view in terms of another view.

For example:

```
create view hiprice_computer
as select title, price
from hiprice
where type = "popular_comp"
```

13.4.2.7 Distinct Views

You can ensure that the rows contained in a view are unique.

For example:

```
create view author_codes
as select distinct au_id
from titleauthor
```

A row is a duplicate if all of its column values match the same column values contained in another row. Two null values are considered to be identical.

SAP ASE applies the `distinct` requirement to a view definition when it accesses the view for the first time, before it performs any projecting or selecting. Views look and act like any database table. If you select a projection of the distinct view (that is, you select only some of the view's columns, but all of its rows), you can get results that appear to be duplicates. However, each row in the view itself is still unique. For example, suppose that you create a `distinct` view, `myview`, with three columns, `a`, `b`, and `c`, that contains these values:

a	b	c
1	1	2
1	2	3
1	1	0

When you enter this query:

```
select a, b from myview
```

the results look like this:

```
a    b
---  ---
1    1
1    2
1    1

(3 rows affected)
```

The first and third rows appear to be duplicates. However, the underlying view's rows are still unique.

13.4.2.8 Views That Include IDENTITY Columns

You can define a view that includes an IDENTITY column by listing the column name, or the `syb_identity` keyword, in the view's `select` statement.

For example:

```
create view sales_view
as select syb_identity, stor_id
from sales_daily
```

However, you cannot add a new IDENTITY column to a view by using the `<identity_column_name>` = `identity(<precision>)` syntax.

You can use the `syb_identity` keyword to select the IDENTITY column from the view, unless the view:

- Selects the IDENTITY column more than once
- Computes a new column from the IDENTITY column
- Includes an aggregate function
- Joins columns from multiple tables
- Includes the IDENTITY column as part of an expression

If any of these conditions is true, SAP ASE does not recognize the column as an IDENTITY column with respect to the view. When you execute `sp_help` on the view, the column displays an "Identity" value of 0.

In the following example, the `row_id` column is not recognized as an IDENTITY column with respect to the `store_discounts` view because `store_discounts` joins columns from two tables:

```
create view store_discounts
as
select stor_name, discount
from stores, new_discounts
where stores.stor_id = new_discounts.stor_id
```

When you define the view, the underlying column retains the IDENTITY property. When you update a row through the view, you cannot specify a new value for the IDENTITY column. When you insert a row through the view, SAP ASE generates a new, sequential value for the IDENTITY column. Only the table owner, database owner, or system administrator can explicitly insert a value into the IDENTITY column after setting `identity_insert on` for the column's base table.

13.4.3 Validate a View's Selection Criteria

When you create a view using the `with check option` clause, each `insert` and `update` through the view is validated against the view's selection criteria. All rows inserted or updated through the view must remain visible through the view, or the statement fails.

Normally, SAP ASE does not check `insert` and `update` statements on views to determine whether the affected rows are within the scope of the view. A statement can insert a row into the underlying base table, but not into the view, or change an existing row so that it no longer meets the view's selection criteria.

Here is an example of a view, `stores_ca`, created using `with check option`. This view includes information about stores located in California, but excludes information about stores located in any other state. The view is created by selecting all rows from the `stores` table for which `state` has a value of "CA":

```
create view stores_ca
as select * from stores
where state = "CA"
with check option
```

When you try to insert a row through `stores_ca`, SAP ASE verifies that the new row falls within the scope of the view. The following `insert` statement fails because the new row would have a `state` value of "NY" rather than "CA":

```
insert stores_ca
values ("7100", "Castle Books", "351 West 24 St.", "New York", "NY", "USA",
"10011", "Net 30")
```

When you try to update a row through `stores_cal`, SAP ASE verifies that the update will not cause the row to disappear from the view. The following `update` statement fails because it attempts to change the value of `state` from "CA" to "MA." If this update was allowed, the row would no longer be visible through the view.

```
update stores_ca
set state = "MA"
where stor_id = "7066"
```

13.4.3.1 Views Derived from Other Views

When a view is created using `with check option`, all views derived from the base view must satisfy its check option. Each row inserted through the derived view must be visible through the base view. Each row updated through the derived view must remain visible through the base view.

Consider the view `stores_cal30`, which is derived from `stores_cal`. The new view includes information about stores in California with payment terms of "Net 30":

```
create view stores_cal30
as select * from stores_ca
where payterms = "Net 30"
```

Because `stores_cal` was created using `with check option`, all rows inserted or updated through `stores_cal30` must be visible through `stores_cal`. Any row with a `state` value other than "CA" is rejected.

`stores_cal30` does not have a `with check option` clause of its own. This means that you can insert or update a row with a `payterms` value other than "Net 30" through `stores_cal30`. The following `update` statement would be successful, even though the row would no longer be visible through `stores_cal30`:

```
update stores_cal30
set payterms = "Net 60"
where stor_id = "7067"
```

13.5 Retrieve Data Through Views

When you retrieve data through a view, SAP ASE verifies that all the database objects referenced anywhere in the statement exist and that they are valid in the context of the statement. If the checks are successful, SAP ASE combines the statement with the stored definition of the view and translates it into a query on the view's underlying tables.

This process is called view resolution.

Consider the following view definition statement and a query against it:

```
create view hiprice
as select *
from titles
where price > $15
and advance > $5000
select title, type
from hiprice
where type = "popular_comp"
```

Internally, SAP ASE combines the query of `hiprice` with its definition, converting the query to:

```
select title, type
from titles
where price > $15
and advance > $5000
and type = "popular_comp"
```

In general, you can query any view in any way just as if it were a real table. You can use joins, `group by` clauses, subqueries, and other query techniques on views, in any combination. However, if the view is defined with an outer join or aggregate function, you may get unexpected results when you query the view.

> **Note**
>
> You can use `select` on `text` and `image` columns in views. However, you cannot use `readtext` and `writetext` in views.

Related Information

Views Derived From Other Views [page 400]

13.5.1 View Resolution

When you define a view, SAP ASE verifies that all the tables or views listed in the `from` clause exist. Similar checks are performed when you query through the view.

Between the time a view is defined and the time it is used in a statement, things can change. For example, one or more of the tables or views listed in the `from` clause of the view definition may have been dropped. Or one or more of the columns listed in the `select` clause of the view definition may have been renamed.

To fully resolve a view, SAP ASE verifies that:

- All the tables, views, and columns from which the view was derived still exist.
- The datatype of each column on which a view column depends has not been changed to an incompatible type.
- If the statement is an `update`, `insert`, or `delete`, it does not violate the restrictions on modifying views.

If any of these checks fails, SAP ASE issues an error message.

Related Information

Modify Data Through Views [page 406]

13.5.2 Redefine Views

You can redefine a view without redefining other views that depend on it, unless the redefinition makes it impossible for SAP ASE to translate the dependent view.

For example, the `authors` table and three possible views are shown below. Each succeeding view is defined using the view that preceded it: `view2` is created from `view1`, and `view3` is created from `view2`. In this way, `view2` depends on `view1` and `view3` depends on both the preceding views.

Each view name is followed by the `select` statement used to create it.

`view1`:

```
create view view1
as select au_lname, phone
from authors
where postalcode like "94%"
```

`view2`:

```
create view view2
as select au_lname, phone
from view1
where au_lname like "[M-Z]%"
```

`view3`:

```
create view view3
```

```
as select au_lname, phone
from view2
where au_lname = "MacFeather"
```

The `authors` table on which these views are based consists of these columns: `au_id`, `au_lname`, `au_fname`, `phone`, `address`, `city`, `state`, and `postalcode`.

You can drop `view2` and replace it with another view, also named `view2`, that contains slightly different selection criteria, such as:

```
create view view2
as select au_lname, phone
from view3
where au_lname like "[M-P]"
```

`view3`, which depends on `view2`, is still valid and does not need to be redefined. When you use a query that references either `view2` or `view3`, view resolution takes place as usual.

If you redefine `view2` so that `view3` cannot be derived from it, `view3` becomes invalid. For example, if another new version of `view2` contains a single column, `au_lname`, rather than the two columns that `view3` expects, `view3` can no longer be used because it cannot derive the `phone` column from the object on which it depends.

However, `view3` still exists and you can use it again by dropping `view2` and re-creating `view2` with both the `au_lname` and the `phone` columns.

In summary, you can change the definition of an intermediate view without affecting dependent views as long as the `select` list of the dependent views remains valid. If this rule is violated, a query that references the invalid view produces an error message.

13.5.3 Rename Views

You can rename a view using `sp_rename`.

```
sp_rename <objname> , <newname>
```

For example, to rename `titleview` to `bookview`, enter:

```
sp_rename titleview, bookview
```

Follow these conventions when renaming views:

- Make sure the new name follows the rules used for identifiers.
- You can change the name of only views that you own. The database owner can change the name of any user's view.
- Make sure the view is in the current database.

Related Information

Naming Convention Identifiers [page 29]

13.5.4 Alter or Drop Underlying Objects

You can change the name of a view's underlying objects. For example, if a view references a table that is named `new_sales`, and you rename that table to `old_sales`, the view works on the renamed table.

However, if you have dropped a table referenced by a view, and someone tries to use the view, SAP ASE produces an error message. If a new table or view is created to replace the one that was dropped, the view again becomes usable.

If you define a view with a `select *` clause, and then alter the structure of its underlying tables by adding columns, the new columns do not appear. This is because the asterisk shorthand is interpreted and expanded when the view is first created. To see the new columns, drop the view and re-create it.

13.6 Modify Data Through Views

Although SAP ASE places no restrictions on retrieving data through views, and although Transact-SQL places fewer restrictions on modifying data through views than other versions of SQL, certain rules still apply to various data modification operations.

These rules are:

- `update`, `insert`, or `delete` operations that refer to a computed column or a built-in function in a view are not allowed.
- `update`, `insert`, or `delete` operations that refer to a view that includes aggregates or row aggregates are not allowed.
- `insert, delete,` and `update` operations that refer to a `distinct` view are not allowed.
- `insert` statements are not allowed unless all NOT NULL columns in the underlying tables or views are included in the view through which you are inserting new rows. SAP ASE has no way to supply values for NOT NULL columns in the underlying objects.
- If a view has a `with check option` clause, all rows inserted or updated through the view (or through any derived views) must satisfy the view's selection criteria.
- `delete` statements are not allowed on multitable views.
- `insert` statements are not allowed on multitable views created with the `with check option` clause.
- `update` statements are allowed on multitable views where `with check option` is used. The update fails if any of the affected columns appears in the `where` clause, in an expression that includes columns from more than one table.
- `insert` and `update` statements are not allowed on multitable `distinct` views.
- `update` statements cannot specify a value for an IDENTITY column. The table owner, database owner, or a system administrator can `insert` an explicit value into an IDENTITY column after setting `identity_insert on` for the column's base table.
- If you insert or update a row through a multitable view, all affected columns must belong to the same base table.
- `writetext` is not allowed on the `text` and `image` columns in a view.

When you attempt an `update`, `insert`, or `delete` for a view, SAP ASE checks to make sure that none of the above restrictions is violated and that no data integrity rules are violated.

13.6.1 Restrictions on Updating Views

Restrictions on updating views apply to these areas: computed columns in a view definition, `group by` or `compute` in a view definition, null values in underlying objects, view created using `with check option`, multitable views, and views with IDENTITY columns.

Computed Columns in a View Definition

This restriction applies to columns of views that are derived from computed columns or built-in functions. For example, the amt_due column in the view accounts is a computed column.

For example, the `amt_due` column in the view `accounts` is a computed column.

```
create view accounts (title_id, advance, amt_due)
as select titles.title_id, advance,
(price * royalty/100) * total_sales
from titles, roysched
where price > $15
  and advance > $5000
and titles.title_id = roysched.title_id
and total_sales between lorange and hirange
```

The rows visible through `accounts` are:

```
select * from accounts
```

```
title_id     advance     amt_due
--------     --------    ---------
PC1035       7,000.00    32,240.16
PC8888       8,000.00     8,190.00
PS1372       7,000.00       809.63
TC3218       7,000.00       785.63

(4 rows affected)
```

`updates` and `inserts` to the `amt_due` column are not allowed because there is no way to deduce the underlying values for price, royalty, or year-to-date sales from any value you might enter in the `amt_due` column. `delete` operations do not make sense because there is no underlying value to delete.

group by or compute in a View Definition

This restriction applies to all columns in views that contain aggregate values—that is, views that have a definition that includes a `group by` or `compute` clause. Here is a view defined with a `group by` clause and the rows seen through it:

Here is a view defined with a `group by` clause and the rows seen through it:

```
create view categories (category, average_price)
as select type, avg(price)
from titles
```

```
group by type
select * from categories
```

```
category         average_price
-------------    -------------
UNDECIDED                 NULL
business                 13.73
mod_cook                 11.49
popular_comp             21.48
psychology               13.50
trad_cook                15.96

(6 rows affected)
```

You cannot `insert` rows into the view `categories`, because the group to which an inserted row would belong cannot be determined. Updates on the `average_price` column are not allowed, because there is no way to determine how the underlying prices should be changed.

NULL Values in Underlying Objects

This restriction applies to `insert` statements when some NOT NULL columns are contained in the tables or views from which the view is derived.

For example, suppose null values are not allowed in a column of a table that underlies a view. Normally, when you `insert` new rows through a view, any columns in underlying tables that are not included in the view are given null values. If null values are not allowed in one or more of these columns, no inserts can be allowed through the view.

For example, in this view:

```
create view business_titles
as select title_id, price, total_sales
from titles
where type = "business"
```

Null values are not allowed in the `title` column of the underlying table `titles`, so no `insert` statements can be allowed through `business_view`. Although the `title` column does not even exist in the view, its prohibition of null values makes any inserts into the view illegal.

Similarly, if the `title_id` column has a unique index, updates or inserts that would duplicate any values in the underlying table are rejected, even if the entry does not duplicate any value in the view.

Views Created Using with check option

This restriction determines what types of modifications you can make through views with check options. If a view has a `with check option` clause, each row inserted or updated through the view must be visible within the view. This is true whether you insert or update the view directly or indirectly, through another derived view.

Multitable Views

This restriction that determines what types of modifications you can make through views that join columns from multiple tables. SAP ASE prohibits `delete` statements on multitable views, but allows `update` and `insert` statements that would not be allowed in other systems.

You can `insert` or update a multitable view if:

- The view has no `with check option` clause.
- All columns being inserted or updated belong to the same base table.

For example, consider the following view, which includes columns from both `titles` and `publishers` and has no `with check option` clause:

```
create view multitable_view
as select title, type, titles.pub_id, state
from titles, publishers
where titles.pub_id = publishers.pub_id
```

A single `insert` or update statement can specify values *either* for the columns from `titles` *or* for the column from `publishers`:

```
update multitable_view
set type = "user_friendly"
where type = "popular_comp"
```

However, this statement fails because it affects columns from both `titles` and `publishers`:

```
update multitable_view
set type = "cooking_trad",
state = "WA"
where type = "trad_cook"
```

Views with IDENTITY Columns

This restriction determines what types of modifications you can make to views that include IDENTITY columns. By definition, IDENTITY columns cannot be updated. Updates through a view cannot specify an IDENTITY column value.

Inserts to IDENTITY columns are restricted to:

- The table owner
- The database owner or the system administrator, if the table owner has granted them permission
- The database owner or the system administrator, if they are impersonating the table owner by using the `setuser` command.

To enable such inserts through a view, use `set identity_insert on` for the column's base table. You cannot use `set identity_insert on` for the view through which you are inserting.

13.7 Drop Views

To delete a view from the database, use `drop view`.

You can drop more than one view at a time, for example:

```
drop view [<owner>.]<view_name> [, [<owner>.]<view_name>]...
```

Only its owner (or the database owner) can drop a view.

When you issue `drop view`, information about the view is deleted from `sysprocedures`, `sysobjects`, `syscolumns`, `syscomments`, `sysprotects`, and `sysdepends`. All privileges on that view are also deleted.

If a view depends on a table or on another view that has been dropped, SAP ASE returns an error message if anyone tries to use the view. If a new table or view is created to replace the one that has been dropped, and if it has the same name as the dropped table or view, the view again becomes usable, as long as the columns referenced in the view definition exist.

13.8 Use Views as Security Mechanisms

Data in an underlying table that is not included in the view is hidden from users who are authorized to access the view but not the underlying table.

Permission to access the subset of data in a view must be explicitly granted or revoked, regardless of the permissions in force on the view's underlying tables.

For example, you may not want some users to access the columns that have to do with money and sales in the `titles` table. You can create a view of the `titles` table that omits those columns, and then give all users permission on the view, and give only the Sales Department permission on the table. For example:

```
revoke all on titles to public
grant all on bookview to public
grant all on titles to sales
```

See, *Managing User Permissions*, in the *Security Administration Guide*.

13.9 Get Information About Views

System procedures, catalog stored procedures, and SAP ASE built-in functions provide information from the system tables about views.

Use sp_help and sp_helptext to Display View Information

The `sp_help` command reports on a view. To display the text of the `create view` statement, use `sp_helptext`.

Use `sp_help` to report on a view:

```
sp_help hiprice
----------------
```

In the evaluated configuration, the system security officer must reset the `allow select on syscomments.text column` configuration parameter. (See evaluated configuration in the Glossary for more information.) When this happens, you must be the creator of the view or a system administrator to view the text of a view through `sp_helptext`.

To display the text of the `create view` statement, execute `sp_helptext`:

```
sp_helptext hiprice
# Lines of Text
-----------------
3 (1 row affected)
text



------------------------------------------------------------------------------
------------------------------------------------------------------------------
------------------------------------------------------------------------------
------------ --SAP ASE has expanded all '*' elements in the following statement
create view hiprice as select titles.title_id, titles.title, titles.type,
titles.pub_id, titles.price,
titles.advance, titles.total_sales, titles.notes, titles.pubdate,
titles.contract from titles where price > $15 and advance > $5000(3 rows
affected)
(return status = 0)
```

If the source text of a view was encrypted using `sp_hidetext`, SAP ASE displays a message advising you that the text is hidden. See the *Reference Manual: Procedures*.

Use sp_depends to List Dependent Objects

`sp_depends` lists all the objects that the view or table references in the current database, and all the objects that reference that view or table.

```
sp_depends titles
```

```
Things inside the current database that reference the object.
object               type
-------------        ---------------------------
dbo.history_proc     stored procedure
dbo.title_proc       stored procedure
dbo.titleid_proc     stored procedure
dbo.deltitle         trigger
dbo.totalsales_trig  trigger
dbo.accounts         view
dbo.bookview         view
dbo.categories       view
dbo.hiprice          view
dbo.multitable_view  view
dbo.titleview        view

(return status = 0)
```

List All Views in a Database

`sp_tables` lists all views in a database.

The syntax is:

```
sp_tables @table_type = "'VIEW'"
```

Find an Object Name and ID

The system functions `object_id` and `object_name` identify the ID and name of a view.

For example:

```
select object_id("titleview")
```

```
----------
 480004741
```

Object names and IDs are stored in the `sysobjects` table.

14 Defining Defaults and Rules for Data

A default is a value that SAP ASE inserts into a column if a user does not explicitly enter one. In database management, a rule specifies what you are or are not allowed to enter in a particular column, or in any column that uses a given user-defined datatype.

You can use defaults and rules to help maintain the integrity of data across the database.

You can define a value for a table column or user-defined datatype that is automatically inserted if a user does not explicitly enter a value.

For example, you can create a default that has the value "???" or the value "fill in later." You can also define rules for that table column or datatype to restrict the types of values users can enter for it.

In a relational database management system, every data element must contain some value, even if that value is null. Some columns do not accept the null value. For those columns, some other value must be entered, either a value explicitly entered by the user or a default entered by SAP ASE.

Rules enforce the integrity of data in ways not covered by a column's datatype. A rule can be connected to a specific column, to several specific columns or to a specified, user-defined datatype.

Every time a user enters a value, SAP ASE checks it against the most recent rule that has been bound to the specified column. Data entered prior to the creation and binding of a rule is not checked.

You can create sharable inline default objects for default clauses and automatically use the same default object for multiple tables and columns.

As an alternative to using defaults and rules, you can use the `default` clause and the `check` integrity constraint of the `create table` statement to accomplish some of the same tasks. However, these items are specific to each table and cannot be bound to columns of other tables or to user-defined datatypes.

14.1 Create Defaults

Define a default using `create default`, then bind the default to the appropriate table column or user-defined datatype using `sp_bindefault`.

You can test the bound default by inserting data.

You can drop defaults using `drop default` and remove their association using `sp_unbinddefault`.

When you create and bind defaults:

- Make sure the column is large enough for the default. For example, a `char (2)` column will not hold a 17-byte string like "Nobody knows yet."
- Be careful when you put a default on a user-defined datatype and a different default on an individual column of that type. If you bind the datatype default first and then the column default, the column default replaces the user-defined datatype default for the named column only. The user-defined datatype default is bound to all the other columns having that datatype.

However, once you bind another default to a column that has a default because of its type, that column ceases to be influenced by defaults bound to its datatype.

- Watch for conflicts between defaults and rules. Be sure the default value is allowed by the rule; otherwise, the default may be eliminated by the rule.
  For example, if a rule allows entries between 1 and 100, and the default is set to 0, the rule rejects the default entry. Either change the default or change the rule.
  The syntax is:

  ```
  create default [<owner>.]<default_name>
       as <constant_expression>
  ```

  Default names must follow the rules for identifiers.
  Within a database, default names must be unique for each user. For example, you cannot create two defaults called `phonedflt`. However, as "guest," you can create a `phonedflt` even if `dbo.phonedflt` already exists because the owner name makes each one distinct.
  Another example: suppose you want to create a default value of "Oakland" that can be used with the `city` column of `friends_etc` and possibly with other columns or user datatypes. To create the default, enter:

  ```
  create default citydflt
  as "Oakland"
  ```

  As you continue to follow this example, you can use any city name that works for the people you are going to enter in your personal table.
  Enclose character and date constants in quotes; money, integer, and floating point constants do not require them. Binary data must be preceded by "0x", and money data should be preceded by a dollar sign ($), or whatever monetary sign is the logical default currency for the area where you are working. The default value must be compatible with the datatype of the column, for example, you cannot use "none" as a default for a numeric column, but 0 is appropriate.
  Usually, you enter default values when you create a table. However, during a session in which you want to enter many rows having the same values in one or more columns, you may want to, before you begin, create a default tailored to that session.

  > **Note**
  >
  > You cannot issue `create table` with a declarative default and then insert data into the table in the same batch or procedure. Either separate the create and insert statements into two different batches or procedures, or use `execute` to perform the actions separately.

14.1.1 Bind Defaults

Use `sp_bindefault` to bind a default to a column or user-defined datatype.

For example, suppose you create the following default:

```
create default advancedflt as "UNKNOWN"
```

Now, bind the default to the appropriate column or user-defined datatype.

```
sp_bindefault advancedflt, "titles.advance"
```

The default takes effect only if the user does not make an entry in the `advance` column of the `titles` table. Not making an entry is different from entering a null value. A default can connect to a particular column, to a number of columns, or to all columns in the database that have a given user-defined datatype.

> **Note**
>
> To use the default, you must issue an `insert` or `update` command with a column list that does not include the column that has the default.

These restrictions apply:

- A default applies only to new rows. It does not retroactively change existing rows. Defaults take effect only when no entry is made. If you supply any value for the column, including NULL, the default has no effect.
- You cannot bind a default to a system datatype.
- You cannot bind a default to a `timestamp` column, because SAP ASE automatically generates values for `timestamp` columns.
- You cannot bind defaults to system tables.
- Although you can bind a default to an IDENTITY column or to a user-defined datatype with the IDENTITY property, SAP ASE ignores such defaults. When you insert a row into a table without specifying a value for the IDENTITY column, SAP ASE assigns a value that is 1 greater than the last value assigned.
- If a default already exists on a column, you must remove it before you can bind a new default. Use `sp_unbindefault` to remove defaults created with `sp_bindefault`. Use `alter table` to remove defaults created with `create table`.

To bind `citydflt` to the `city` column in `friends_etc`, type:

```
sp_bindefault citydflt, "friends_etc.city"
```

The table and column name must be enclosed in quotes, because of the embedded punctuation (the period).

If you create a special datatype for all city columns in every table in your database, and bind `citydflt` to that datatype, “Oakland” appears only where city names are appropriate. For example, if the user datatype is called `citytype`, here is how to bind `citydflt` to it:

```
sp_bindefault citydflt, citytype
```

To prevent existing columns or a specific user datatype from inheriting the new default, use the `futureonly` parameter when binding a default to a user datatype. However, do not use `futureonly` when binding a default to a column. Here is how you create and bind the new default “Berkeley” to the datatype `citytype` for use by new table columns only:

```
create default newcitydflt as "Berkeley"
sp_bindefault newcitydflt, citytype, futureonly
```

“Oakland” continues to appear as the default for any existing table columns using `citytype`.

If most of the people in your table live in the same postal code area, you can create a default to save data entry time. Here is one, along with its binding, that is appropriate for a section of Oakland:

```
create default zipdflt as "94609"
sp_bindefault zipdflt, "friends_etc.postalcode"
```

Here is the complete syntax for `sp_bindefault`:

```
sp_bindefault <defname, objname> [, futureonly]
```

`<defname>` is the name of the default created with `create default`. `<objname>` is the name of the table and column, or of the user-defined datatype, to which the default is to be bound. If the parameter is not of the form `<table>.<column>`, it is assumed to be a user-defined datatype.

All columns of a specified user-defined datatype become associated with the specified default unless you use the optional `futureonly` parameter, which prevents existing columns of that user datatype from inheriting the default.

> **Note**
>
> Defaults cannot be bound to columns and used during the same batch. `sp_bindefault` cannot be in the same batch as `insert` statements that invoke the default.

After you create a default, the source text describing the default is stored in the `text` column of the `syscomments` system table. Do not remove this information; doing so may cause problems for future versions of SAP ASE. Instead, use use `sp_hidetext` to encrypt the text in `syscomments`. See the *Reference Manual: Procedures*.

Related Information

Compiled Objects [page 22]

14.1.2 Unbind Defaults

Unbinding a default means disconnecting it from a particular column or user-defined datatype. An unbound default remains in the database and is available for future use. Use `sp_unbindefault` to remove the binding between a default and a column or datatype.

Here is how to unbind the current default from the `city` column of the `friends_etc` table:

```
execute sp_unbindefault "friends_etc.city"
```

To unbind a default from the user-defined datatype `citytype`, use:

```
sp_unbindefault citytype
```

The complete syntax of `sp_unbindefault` is:

```
sp_unbindefault <objname> [, futureonly]
```

If the `<objname>` parameter you give is not of the form `<table>.<column>`, SAP ASE assumes it is a user-defined datatype. When you unbind a default from a user-defined datatype, the default is unbound from all

columns of that type unless you give the optional `futureonly` parameter, which prevents existing columns of that datatype from losing their binding with the default.

14.1.3 How Defaults Affect NULL Values

If you specify NOT NULL when you create a column and do not create a default for it, SAP ASE produces an error message whenever a user inserts a row without making an entry in that column.

When you drop a default for a NULL column, SAP ASE inserts NULL in that position each time you add rows without entering any value for that column. When you drop a default for a NOT NULL column, you get an error message when rows are added, without a value entered for that column.

This table illustrates the relationship between the existence of a default and the definition of a column as NULL or NOT NULL.

Column Definition	User Entry	Result
Null and default defined	No value NULL value	Default used NULL used
Null defined, no default defined	No value NULL value	NULL used NULL used
Not null, default defined	No value NULL value	Default used Error
Not null, no default defined	No value NULL value	Error Error

14.2 Drop Defaults

To remove a default from the database, use the `drop default` command.

Unbind the default from all columns and user datatypes before you drop it. If you try to drop a default that is still bound, SAP ASE displays an error message and the `drop default` command fails.

Here is how to remove `citydflt`. First, you unbind it:

```
sp_unbindefault citydft
```

Then you can drop `citydft`:

```
drop default citydflt
```

The complete syntax of `drop default` is:

```
drop default [<owner>.]<default_name>
  [, [<owner>.]<default_name>] ...
```

A default can be dropped only by its owner. See the *Reference Manual: Procedures* and the *Reference Manual: Commands*.

Related Information

Unbind Defaults [page 416]

14.3 Create Rules

Create the rule using `create rule`, then bind the rule to a column or user-defined datatype using `sp_bindrule`.

You can test the bound rule by inserting data. Many errors in creating and binding rules can be caught only by testing with an `insert` or `update` command.

You can unbind a rule from the column or datatype either by using `sp_unbindrule` or by binding a new rule to the column or datatype.

The syntax is:

```
create rule [<owner>.]<rule_name>
     as <condition_expression>
```

Rule names must follow the rules for identifiers. You can create a rule only in the current database.

Within a database, rule names must be unique for each user. For example, a user cannot create two rules called `socsecrule`. However, two different users can create a rule named `socsecrule`, because the owner names make each one distinct.

Here is how the rule permitting five different `pub_id` numbers and one dummy value (99 followed by any two digits) was created:

```
create rule pub_idrule
as @pub_id in ("1389", "0736", "0877", "1622", "1756")
or @pub_id like "99[0-9][0-9]"
```

The `as` clause contains the name of the rule's argument, prefixed with "@", and the definition of the rule itself. The argument refers to the column value that is affected by the `update` or `insert` statement.

The argument is `@pub_id`, a convenient name, since this rule is to be bound to the `pub_id` column. You can use any name for the argument, but the first character must be "@." Using the name of the column or datatype to which the rule will be bound may help you remember what it is for.

The rule definition can contain any expression that is valid in a `where` clause, and can include arithmetic operators, comparison operators, `like`, `in`, `between`, and so on. However, the rule definition cannot directly

reference any column or other database object. It can include built-in functions that do not reference database objects.

The following example creates a rule that forces the values you enter to comply with a particular "picture." In this case, each value entered in the column must begin with "415" and be followed by 7 more characters:

```
create rule phonerule
as @phone like "415_ _ _ _ _ _ _"
```

To make sure that the ages you enter for your friends are between 1 and 120, but never 17, try this:

```
create rule agerule
as @age between 1 and 120 and @age ! = 17
```

14.3.1 Bind Rules

After you have created a rule, use `sp_bindrule` to link the rule to a column or user-defined datatype.

Here is the complete syntax for `sp_bindrule`:

```
sp_bindrule <rulename>, <objname> [, futureonly]
```

The `<rulename>` is the name of the rule created with `create rule`. The `<objname >`is the name of the table and column, or of the user-defined datatype to which the rule is to be bound. If the parameter is not of the form `<table.column>`, it is assumed to be a user datatype.

Use the optional `futureonly` parameter only when binding a rule to a user-defined datatype. All columns of a specified user-defined datatype become associated with the specified rule unless you specify `futureonly`, which prevents existing columns of that user datatype from inheriting the rule. If the rule associated with a given user-defined datatype has previously been changed, SAP ASE maintains the changed rule for existing columns of that user-defined datatype.

After you define a rule, the source text describing the rule is stored in the `text` column of the `syscomments` system table. Do not remove this information; doing so may cause problems for future versions of SAP ASE. Instead, encrypt the text in `syscomments` by using `sp_hidetext`. See the *Reference Manual: Procedures*.

These restrictions apply:

- You cannot bind a rule to a `text`, `unitext`, `image`, or `timestamp` datatype column.
- You cannot use rules on system tables.

Related Information

Compiled Objects [page 22]

14.3.1.1 Rules Bound to Columns

Bind a rule to a column using `sp_bindrule` with the rule name and the quoted table name and column name.

For example, to bind `pub_idrule` to `publishers.pub_id`, use:

```
sp_bindrule pub_idrule, "publishers.pub_id"
```

This rule ensures that all postal codes entered have 946 as the first 3 digits:

```
create rule postalcoderule946
as @postalcode like "946[0-9][0-9]"
```

Bind it to the `postalcode` column in `friends_etc` like this:

```
sp_bindrule postalcoderule946, "friends_etc.postalcode"
```

Rules cannot be bound to columns and used during the same batch. `sp_bindrule` cannot be in the same batch as `insert` statements that invoke the rule.

14.3.1.2 Rules Bound to User-Defined Datatypes

You cannot bind a rule to a system datatype, but you can bind one to a user-defined datatype. To bind `phonerule` to a user-defined datatype called `p#`, enter:

```
sp_bindrule phonerule, "p#"
```

14.3.1.3 Precedence of Rules

Rules that are bound to columns always take precedence over rules that are bound to user datatypes.

Binding a rule to a column replaces a rule bound to the user datatype of that column, but binding a rule to a datatype does not replace a rule bound to a column of that user datatype.

A rule bound to a user-defined datatype is activated only when you attempt to insert a value into, or update, a database column of the user-defined datatype. Because rules do not test variables, do not assign a value to a user-defined datatype variable that would be rejected by a rule bound to a column of the same datatype.

This table indicates the precedence when binding rules to columns and user datatypes where rules already exist:

New Rule Bound To	Old Rule Bound To	
	User datatype	*Column*
User datatype	Replaces old rule	No change

New Rule Bound To	Old Rule Bound To	
Column	Replaces old rule	Replaces old rule

When you are entering data that requires special temporary constraints on some columns, you can create a new rule to help check the data. For example, suppose that you are adding data to the `debt` column of the `friends_etc` table. You know that all the debts you want to record today are between $5 and $200. To avoid accidentally typing an amount outside those limits, create a rule like this one:

```
create rule debtrule
as @debt = $0.00 or @debt between $5.00 and $200.00
```

The `@debt` rule definition allows for an entry of $0.00 to maintain the default previously defined for this column.

Bind `debtrule` to the `debt` column like this:

```
sp_bindrule debtrule, "friends_etc.debt"
```

14.3.2 Rules and NULL Values

You cannot define a column to allow nulls, and then override this definition with a rule that prohibits null values.

For example, if a column definition specifies NULL and the rule specifies the following, an implicit or explicit NULL does not violate the rule:

```
@val in (1,2,3)
```

The column definition overrides the rule, even a rule that specifies:

```
@val is not null
```

14.3.3 Unbind Rules

Unbinding a rule disconnects it from a particular column or user-defined datatype. An unbound rule's definition remains in the database and is available for future use.

There are two ways to unbind a rule:

- Use `sp_unbindrule` to remove the binding between a rule and a column or user-defined datatype.
- Use `sp_bindrule` to bind a new rule to that column or datatype. The old one is automatically unbound.

Here is how to disassociate `debtrule` (or any other currently bound rule) from `friends_etc.debt`:

```
sp_unbindrule "friends_etc.debt"
```

The rule is still in the database, but it has no connection to `friends_etc.debt`.

To unbind a rule from the user-defined datatype `p#`,

```
sp_unbindrule "p#"
```

The complete syntax of `sp_unbindrule` is:

```
sp_unbindrule <objname> [, futureonly]
```

If the `<objname>` parameter you use is not of the form "`<table.column>`," SAP ASE assumes it is a user-defined datatype. When you unbind a rule from a user-defined datatype, the rule is unbound from all columns of that type unless:

- You use the optional `futureonly` parameter, which prevents existing columns of that datatype from losing their binding with the rule, or
- The rule on a column of that user-defined datatype has been changed so that its current value is different from the rule being unbound.

14.4 Drop Rules

To remove a rule from the database entirely, use the `drop rule` command.

Unbind the rule from all columns and user datatypes before you drop it. If you try to drop a rule that is still bound, SAP ASE displays an error message, and `drop rule` fails. However, you need not unbind and then drop a rule to bind a new one. Simply bind a new one in its place.

To remove `phonerule` after unbinding it:

```
drop rule phonerule
```

The complete syntax for `drop rule` is:

```
drop rule [<owner>.]<rule_name>
     [, [<owner>.]<rule_name>] ...
```

After you drop a rule, new data entered into the columns that previously were governed by it goes in without these constraints. Existing data is not affected in any way.

A rule can be dropped only by its owner.

14.5 Retrieve Information About Defaults and Rules

Use `sp_help` with a table name to show the rules and defaults that are bound to columns.

This example displays information about the `authors` table in the `pubs2` database, including the rules and defaults:

```
sp_help authors
```

`sp_help` also reports on a rule bound to a user-defined datatype. To check whether a rule is bound to the user-defined datatype `p#`, use:

```
sp_help "p#"
```

`sp_helptext` reports the definition (the `create` statement) of a rule or default.

If the source text of a default or rule was encrypted using `sp_hidetext`, SAP ASE displays a message advising you that the text is hidden. See the *Reference Manual: Procedures*.

If the system security officer has reset the `allow select on syscomments.text column` parameter with `sp_configure` (as required to run SAP ASE in the evaluated configuration), you must be the creator of the default or rule or a system administrator to view the text of a default or rule through `sp_helptext`. See, *Introduction to Security*, in the *Security Administration Guide*.

14.6 Share Inline Defaults

When you create a new inline default, SAP ASE looks for an existing shareable inline default that have the same value in the database belonging to the same user. If one exists, SAP ASE binds this object to the column instead of creating a new default.

However, if SAP ASE does not find an existing shareable inline default, it creates a new default.

SAP ASE shares inline defaults between tables only within the same database.

Enable shared inline defaults by setting `enable functionality group` to a value greater than 0. See, *Setting Configuration Parameters*, in the *System Administration Guide, Volume 1*.

14.6.1 Create an Inline Shared Default

SAP ASE automatically creates and uses sharable inline defaults when the default clause is used in `create table` or `alter table` commands.

For example, if you create this table:

```
create table my_titles
(title_id char(6),
title    varchar(80),
```

```
moddate  datetime default '12/12/2012')
```

Then create a second table with the same default:

```
create table my_authors2
(auth_id char(6),
title    varchar(80),
moddate  datetime default '12/12/2012')
```

`sysobjects` reports a single default shared between the two:

```
select id, name from sysobjects where type = 'D'
```

```
 id          name
 ----------- ---------------------------
 1791386948  my_titles_moddat_1791386948
```

Use `sp_helpconstraint` to view the definition for a shareable inline default object:

```
sp_helpconstraint my_titles
```

```
name                              defintion              created
--------------------------------  -------------------    ---------------
my_titles_moddate_1791386948      DEFAULT '12/12/2012'   Dec 6 2010 10:55AM
sp_helpconstraint my_authors2
name                              defintion             created
--------------------------------  -------------------   ---------------
my_titles_moddate_1791386948      DEFAULT '12/12/2012'   Dec 6 2010 10:55AM
```

`my_titles` and `my_authors2` show the same internal default name, `my_titles_moddate_1791386948`, which indicates there is a single default object shared between these columns. `sp_help` also shows the defaults associated with columns.

14.6.2 Unbind a Shared Inline Default

You cannot explicitly unbind a sharable inline default from a column. SAP ASE automatically unbinds or drops it from the column during `drop table` or `alter table` commands.

When you drop or alter a table, SAP ASE checks whether any inline defaults are shared with other columns. If they are, SAP ASE unbinds the columns from the inline default object without dropping the shared defaults.

If an inline default object is no longer used by any column, SAP ASE drops it.

14.6.3 Limitations for Shared Inline Defaults

Certain limitations apply to shared inline defaults.

- You cannot use shared inline defaults in global or user `tempdb`.
- Inline defaults that use a variable cannot be shared.
- You cannot use shared inline defaults between users.

- Inline defaults that use expressions cannot be shared.
- `<constant_value>` must be a constant literal.
- Inline defaults that are defined in `syscomments` as requiring more than one row (that is, more than 255 bytes) cannot be shared.
- Shareable inline defaults that exist on an SAP ASE server you are downgrading are treated as if they were created with the `create default` command from the previous release. These defaults are fully functional with the `sp_binddefault`, `sp_unbindefault` and `drop default` commands using the internal default names.

15 Precomputed Result Sets

A precomputed result set is a view for which the result is computed, stored, and available for future use. Once configured for precomputed result sets, SAP ASE precomputes queries and attempts to use the precomputed result during subsequent iterations. Precomputed result sets are also called materialized views.

Conceptually, a precomputed result set is both a view (because it includes query definition stored in the system tables) and a table (because it includes persistent data). You can perform many of the same operations that you perform on tables on precomputed result sets, including creating indexes and running update statistics.

Once SAP ASE is configured to use precomputed result sets, the optimizer attempts to automatically rewrite each query using a precomputed result set. However, the final plan the optimizer selects is primarily cost based.

When the optimizer rewrites a query using a precomputed result set, it decides which precomputed result set is the best candidate. If the optimizer chooses to replace all, or part, of a query with a precomputed result set, it also adds any necessary compensation to the rewritten query (that is, any predicates needed to ensure the rewritten query is equivalent to the original user query). For example, if the user query includes a join of:

```
c1=c2 and c2=c3 and c3=c4
```

but the precomputed result set includes a join for:

```
c1=c2 and c3=c4
```

the rewritten query using the precomputed result set must have a compensation predicate similar to c1=c3 to form an equivalent query.

Like an index, a precomputed result set has a maintenance cost for concurrent `insert`, `update`, and `delete` statements. Generally, precomputed result-set maintenance overhead consists of more than maintaining the indexes when the definition involves multiple table joins. Consequently, precomputed result sets are unsuitable for OLTP with heavy concurrent `insert`, `update`, and `delete` statements and simple index-based `select` statements.

> **Note**
>
> SAP ASE allows you to run `updates statistics` on precomputed result sets.

15.1 Benefits of Precomputed Result Sets

Whether your site benefits from precomputed result sets depends on how they are designed.

Although you may want to precompute as many queries as possible (particularly more joins) and make them available for multiple queries, precomputed result sets take extra disk space and have a higher maintenance cost. You can create extra indexes to help query performance, but these also incur an extra maintenance cost.

Precomputed result sets are best for frequently executed, expensive queries, such as those involving intensive aggregation and join operations. When you submit a query, the optimizer attempts to rewrite the query to use existing precomputed result sets instead of the base tables.

Generally, capture your application's workload and design your precomputed result sets based on this workload. A good place to start is to create a combined join graph for all queries—along with their frequency of use—to indicate good candidates for using the same precomputed result set for multiple queries.

Test your precomputed result sets before putting them into production. If the queries are read-only or read-most, measure their performance gain against the extra disk space they use and the amount of time it takes them to populate the data; if it is a mixture of read-only or read-most, measure the impact of the precomputed result sets against the throughput.

15.2 Configuring SAP ASE for Precomputed Result Sets

Before you create or alter precomputed result sets, verify that a number of session `set` parameters are set correctly

Context

Including:

- `set ansinull` – on
- `set arithabort` – on
- `set arithignore` – off
- `set string_rtruncation` – on

Procedure

1. Use `create precomputed result set` to create precomputed result sets.
2. To use precomputed result sets for your queries, issue the `set materialized_view_optimization` command for the session.

15.3 Creating Precomputed Result Sets

To use precomputed result sets for your queries, issue the `set materialized_view_optimization` command for the session.

Procedure

The syntax is:

```
 create {precomputed result set | materialized view}
    prs_name [(alternative_column_name
    [ [constraint constraint_name]
        unique (column_name,...)]
    [{immediate | manual} refresh]
    [{populate | nopopulate}]
    [enable | disable]
    [{enable | disable} use in optimization]
    [lock { datarows | datapages | allpages}]
    [on segment_name]
    [partition_clause]
as query_expression
```

You may specify the following in precomputed result sets:

- Partitions
- Segments
- Indexes (functional indexes not allowed)
- Unique keys (you must include the unique key constraint when you create precomputed result sets for immediate refresh)

If you drop the base table, the precomputed result set is changed to disabled.See *Reference Manual: Commands*.

15.4 Identifying Precomputed Result Sets

`sp_help` includes information about precomputed result sets in the `Object_type` and `object_status` columns.

Procedure

For example:

```
 sp_help mv1
 Name  Owner  Object_type  Object_status Create_date
------ ----- ------------- ------------- ------------
```

```
mv1 dbo precomputed result set  immediate, enabled, enabled for QRW
Apr 10 2012  8:57AM
...
```

`sysobjects` indicates an object is a precomputed result set with a value of `RS` in the `type` column, and `sp_showoptstats` and `sp_depends` display additional information about precomputed result sets.

15.5 Refreshing Precomputed Result Sets

Precomputed result sets do not necessarily remain synchronized with the base tables from which they are constructed, and must also be refreshed, either automatically or manually.

Configure the refresh policy when you create the precomputed result set, or later with the `alter precomputed result set` command:

- Immediate refresh – the precomputed result set is updated during the same transaction that updates the base tables. This is the default option. However, creating a precomputed result set for `immediate refresh` requires the user to own all the tables in the definition query. You can grant and revoke access permissions for precomputed result sets created with the `immediate refresh` parameter for the `select`, `update statistics`, and `delete statistics` commands.
- Manual refresh – the precomputed result set is updated with an explicit `refresh` command. Because manual refreshes are not maintained, SAP ASE considers the data they contain to be stale (even immediately after you issue `refresh`), and selects these precomputed result sets for query rewrites only if the query is acceptable with stale data. The `refresh` command is executed under isolation level 1 or above.

The syntax to manually refresh a precomputed result set is:

```
refresh {precomputed result set | materialized view}
    [owner_name.]prs_name
```

If the schema of any of the base tables from which the precomputed result set is derived has changed, or if it was dropped and re-created (that is, the object ID has changed), the `refresh` command fails and returns an error indicating the precomputed result set must be dropped and re-created.

Only the owner of the precomputed result set can use the `refresh` command. If a user has permission to update the base table, he or she can also maintain the precomputed result set.

In most situations, the optimizer should use precomputed result sets with `immediate refresh` instead of `manual refresh` for query rewriting (unless you set `materialized_view_optimization` to `stale`).

Manually refreshing the precomputed result set is best when you control when `insert`, `update`, and `delete` statements occur. After they occur, perform a planned manual refresh of the precomputed result sets, then use the precomputed result sets to help your read-only applications. However, be aware of the time and extra disk space required to perform a manual refresh and plan accordingly.

> **Note**
>
> After creating a precomputed result set, its owner may not have `select` permission on the base tables. If this occurs, manually refreshing the precomputed result set maintenance may fail, and it is not updated with the new changes from the base table. You cannot execute the refresh command as part of a batch.

This example illustrates how to refresh a precomputed result set

1. Create table `t1`:

```
create table t1 (
c1 int,
c2 int,
c3 char(5))
```

   And populate it with this data:

```
c1          c2          c3
----------- ----------- -----
          1           3  Aagg
          2           8  Xyz
```

2. Create table t2:

```
create table t2
(a1 int,
a2 int,
a3 char(5))
```

   And populate it with this data:

```
a1          a2          a3
----------- ----------- -----
          1           5  Ghr
          2           1  Gser
          3           6  agfh
```

3. Create the `prs_1` precomputed result set:

```
create precomputed result set prs_1
unique (t1.c1, t2.a2)
as select t1.c1, t2.a2 from t1, t2 where t1.c1=t2.a1
```

   `prs_1` is created and populated with these initial rows:

```
c1          a2
----------- -----------
          1           5
          2           1
```

4. If you insert the values 3, 7, and “fhi” into t1, `prs_1` is immediately updated with the values 3 and 6:

```
c1          a2
----------- -----------
          1           5
          2           1
          3           6
```

5. If you delete rows from `t2` where a1 = 2, `prs_1` is immediately updated with this change:

```
c1          a2
----------- -----------
          1           5
          3           6
```

If SAP ASE rolls back the transaction updating the base table, it also rolls back the immediate update on the base table's precomputed result set as part of the same transaction.

15.6 Altering Precomputed Result Sets

Use the `alter` command to change the precomputed result set's policies or properties.

Context

The syntax is:

```
alter {precomputed result set | materialized view}
<prs_name>
{immediate | manual} refresh
| enable | disable
| {enable | disable} use in optimization
```

Procedure

This example alters the `author_prs` precomputed result set from `manual` to `immediate`:

```
alter precomputed result set author_prs
immediate refresh
```

Next Steps

`alter` automatically refreshes the precomputed result set when you change from a `manual` to an `immediate` refresh, or from `disable` to `enable`. Altering a precomputed result set for `disable use in optimization` prevents the precomputed result set from participating in future query rewriting. However, any plans already cached using the precomputed result set are not recompiled.

Similar to other DDL commands, you cannot issue `alter precomputed result set` as part of a multistatement transaction (unless you set the `ddl in tran option` to `true` for the database). You must be the owner of the precomputed result set to issue alter precomputed result set

If the base table or view on which the precomputed result set is based is dropped or altered, the precomputed result set is automatically altered to disable. SAP ASE sets the precomputed result sets to `disable` when you run `bcp` in or `select into existing` against any base table from which the precomputed result set is generated.

When you alter a precomputed result set to `disable` (with the `alter precomputed result set` command or with the `alter table` command on the base table), any plans already cached that use the precomputed result set are recompiled when the plan is next executed.

15.7 Dropping or Truncating Precomputed Result Sets

Dropping a precomputed result set deletes its data, removes any system table entries, and deletes the precomputed result set.

Context

The syntax is:

```
drop {precomputed result set | materialized view}
    prs_name
```

You must be the owner of the precomputed result set to issue drop precomputed result set. See *Reference Manual: Commands*.

Procedure

This example drops `authors_prs`:

```
 drop precomputed result set authors_prs
```

Next Steps

Use the `truncate` command to truncate data in a precomputed result set. `truncate` retains the definition of the precomputed result set in the system table, ensuring that the precomputed result set can be later repopulated using the `refresh` command.

Truncating a precomputed result set moves it to a disabled state. SAP ASE moves the precomputed result set back to the enabled state when you issue `refresh prs`.

The syntax is:

```
truncate {precomputed result set | materialized view}
    prs_name
```

This example truncates the author_prs:

```
truncate precomputed result set authors_prs
```

SAP ASE implements the refresh command first as a `truncate` command and then recomputes the precomputed result set. In the unlikely event that the `truncate` command succeeds but the recompute fails, the precomputed result set is left disabled and you may reissue the `refresh` command.

15.8 Configuring Staleness

Precomputed result sets rely on updates from their base tables to ensure data is current.

Context

When a precomputed result set is configured for immediate updates, any base table updates also update the precomputed result set. This update occurs as an incremental maintenance using changes to the base tables. However, if a precomputed result set is configured for `manual` updates, data may become stale because updates only occur when you run the `refresh` command (during which SAP ASE recomputes the precomputed result set instead of performing an incremental maintenance).

Unless you specify otherwise, SAP ASE does not use a stale precomputed result set to rewrite queries.

Procedure

Use the set materialized_view_optimization to specify at the session level whether SAP ASE can use stale precomputed result sets when rewriting queries during optimization:

```
set materialized_view_optimization {disable | fresh | stale}
```

Next Steps

For SAP ASE to use stale precomputed result sets to rewrite queries:

- The user must be the owner of the stale precomputed result set, and
- `set materialized_view_optimization` must be set to `stale`.

See *Reference Manual: Commands*.

15.9 Querying Precomputed Result Sets

SAP ASE allows you to select information from precomputed result sets, but you cannot insert, update, or delete information from the precomputed result set. Instead, you must insert, update, or delete information from the base tables, and then refresh the precomputed result set.

15.10 Rewriting Queries

Query rewrite mechanisms generate alternative plans based on available precomputed results.

Context

The alternative plans compete with other plans in the optimizer, and SAP ASE selects the one with the lowest estimated cost. However, the query rewrite mechanism works only with select queries; it does not consider `insert`, `update`, `delete`, and `select into` queries for rewrite.

SAP ASE may rewrite an entire query to create an equivalent precomputed result set, or it may rewrite part of a query, depending on the query properties and the available precomputed result sets. The precomputed result set must completely cover the logical data set for queries that SAP ASE rewrites.

Procedure

For example, if you have a query similar to this, which is complicated and involves multitable joins and many predicates, groupings, and aggregations:

```
 select t1.col1,t2.col1,t3.col1,
     sum(t1.col3),sum(t2.col3), sum(t3.col3)
from t1, t2, t3
where t1.col1 = t2.col1
     and t2.col1 = t3.col1
     and t1.col2 < 60
     and t1.col1 > 5
     and t1.col2 + t2.col2 < 40
group by t1.col1, t2.col1, t3.col1
```

And create this precomputed result set:

```
create precomputed result set newprs
as
select t1.col1 as p11, t1.col2 as p12, t2.col1 as p21,
     t2.col2 as p22, t3.col1 as p31, t3.col2 as p32,
     sum(t1.col3) as agg_s13,sum(t2.col3) as agg_s23,
     sum(t3.col3) as agg_s33
from t1, t2, t3
```

```
where t1.col1 = t2.col1
and t1.col2 < 60
and t1.col2 + t2.col2 < 40
group by t1.col1, t2.col1, t3.col1, t1.col2,
     t2.col2, t3.col2
```

The query rewrite mechanism may alter the original query to something similar to this, which is much simpler and may be cheaper to execute:

```
select p11,p21,p31,
     sum(agg_s13),sum(agg_s23),sum(agg_s33)
from newprs
where p21 = p31
     and p11 > 5
group by p11, p21, p31
```

15.11 Replicating Precomputed Result Sets

SAP ASE supports replication of some precomputed result set commands.

Although precomputed result set DDLs can be replicated, precomputed result sets cannot be marked for replication. That is, unlike a regular table, SAP ASE does not replicate any maintenance changes occurring on data stored in a precomputed result set, regardless of whether the changes are initiated from a precomputed result set DDL (which itself could be replicated), or are part of the base table `update` transaction (initiated from an `immediate refresh` policy).

Precomputed result set commands that can be replicated include:

- `create precomputed result set`
- `alter precomputed result set`
- `drop precomputed result set`
- `truncate precomputed result set`
- `refresh precomputed result set`

Precomputed result set DDL replication is supported only between two SAP ASE servers that include the precomputed result set functionality

15.12 Restrictions for Precomputed Result Sets

Precomputed result sets include restrictions.

- References to another precomputed result set. However, precomputed result sets can reference a view if you create the precomputed result set with a manual `refresh` policy.
- Expressions in the `select` list. However, you can include an expression as part of the `group by` list.
- Encrypted columns, or result sets that encrypt their own column data.
- References to tables or functions in another database.
- References to virtual computed columns in a base table. However, precomputed result sets can refer to materialized computed columns in a base table.

- `compute`, `compute by`, `group by all`, or `order by` clauses.
- Nondeterministic functions (for example, `getdate`).
- XML.
- Subqueries.
- Outer and semijoins.
- Calls to user-defined functions.
- Derived tables.
- Reference to system, temporary, or fake tables.
- Union clauses.
- Any constraint other than unique key constraints.
- Columns that are defined with identity, null, or not null clauses.
- Defaults or rules.
- Cursors.
- Statistical aggregate functions.
- `text`, `image`, or `unitext` columns.

In addition to the previous restrictions, precomputed result sets using the immediate refresh policy cannot include:

- `top`, `min`, `max`, and `avg` commands
- `distinct` clauses
- Self joins
- Functions
- References to proxy tables
- References to a view
- `having` clauses that reference the result of an aggregate
- `sum` functions that reference a nullable expression

16 Batches and Control-of-Flow Language

Transact-SQL allows you to group a series of statements as a batch, either interactively or from an operating system file. You can also use Transact-SQL control-of-flow language to connect the statements, using programming constructs.

A variable is an entity that is assigned a value. This value can change during the batch or stored procedure in which the variable is used. SAP ASE has two kinds of variables: local and global. Local variables are user-defined, whereas global variables are predefined, and are supplied by the system during command execution.

SAP ASE can process multiple statements submitted as a batch, either interactively or from a file. A batch or batch file is a set of Transact-SQL statements that are submitted together and executed as a group, one after the other. A batch is terminated by an end-of-batch signal. With the `isql` utility, this is the word "go" on a line by itself. For details on `isql`, see the *Utility Guide*.

This batch contains two Transact-SQL statements:

```
select count(*) from titles
select count(*) from authors
go
```

A single Transact-SQL statement can constitute a batch, but it is more common to think of a batch as containing multiple statements. Frequently, a batch of statements is written to an operating system file before being submitted to `isql`.

Transact-SQL provides special keywords called control-of-flow language that allow users to control the flow of execution of statements. You can use control-of-flow language in single statements, in batches, in stored procedures, and in triggers.

Without control-of-flow language, separate Transact-SQL statements are performed sequentially, as they occur. Correlated subqueries, are a partial exception. Control-of-flow language permits statements to connect and to relate to each other using programming-like constructs.

Control-of-flow language, such as `if...else` for conditional performance of commands and `while` for repetitive execution, lets you refine and control the operation of Transact-SQL statements. The Transact-SQL control-of-flow language transforms standard SQL into a very high-level programming language.

16.1 Rules Associated with Batches

There are rules to govern which Transact-SQL statements can be combined into a single batch.

- Before referencing objects in a database, issue a `use` statement for that database. For example:

  ```
  use master
  go
  select count(*)
  from sysdatabases
  go
  ```

- You cannot combine the following database commands with other statements in a batch:
  - `create procedure`
  - `create rule`
  - `create default`
  - `create trigger`
- You can combine the following database commands with other Transact-SQL statements in a batch:
  - `create database` (except you cannot create a database and create or access objects in the new database in a single batch)
  - `create table`
  - `create index`
  - `create view`
- You cannot bind rules and defaults to columns and use them in the same batch. `sp_bindrule` and `sp_bindefault` cannot be in the same batch as `insert` statements that invoke the rule or default.
- You cannot `drop` an object and then reference or re-create it in the same batch.
- If a table already exists, you cannot re-create it in a batch, even if you include a test in the batch for the table's existence.
  SAP ASE compiles a batch before executing it. During compilation, SAP ASE makes no permission checks on objects, such as tables and views, that are referenced by the batch. Permission checks occur when SAP ASE executes the batch. An exception to this is when SAP ASE accesses a database other than the current one. In this case, SAP ASE displays an error message at compilation time without executing any statements in the batch.
  Assume that your batch contains these statements:

  ```
  select * from taba
  select * from tabb
  select * from tabc
  select * from tabd
  ```

  If you have the necessary permissions for all statements except the third one (`select * from tabc`), SAP ASE returns an error message for that statement and returns results for all the others.

16.1.1 Examples of Using Batches

When submitting a batch, you can use the same format as the `isql` utility, which has a clear end-of-batch signal—the word "go" on a line by itself.

Here is a batch that contains two `select` statements in a single batch:

```
select count(*) from titles
select count(*) from authors
go
```

```
-------------
          18

(1 row affected)
-------------
          23

(1 row affected)
```

You can create a table and reference it in the same batch. This batch creates a table, inserts a row into it, and then selects everything from it:

```
create table test
  (column1 char(10), column2 int)
insert test
  values ("hello", 598)
select * from test
go
```

```
(1 row affected)
column1  column2
-------  -------
hello        598

(1 row affected)
```

You can combine a `use` statement with other statements, as long as the objects you reference in subsequent statements are in the database in which you started. This batch selects from a table in the `master` database and then opens the `pubs2` database. The batch begins by making the `master` database current; afterwards, `pubs2` is the current database.

```
use master
go
select count(*) from sysdatabases
use pubs2
go
```

```
-------------
           6

(1 row affected)
```

You can combine a `drop` statement with other statements as long as you do not reference or re-create the dropped object in the same batch. This example combines a `drop` statement with a `select` statement:

```
drop table test
select count(*) from titles
go
```

```
------------
         18

(1 row affected)
```

If there is a syntax error anywhere in the batch, none of the statements are executed. For example, here is a batch with a typing error in the last statement, and the results:

```
select count(*) from titles
select count(*) from authors
slect count(*) from publishers
go
```

```
Msg 156, Level 15, State 1:
Line 3:
Incorrect syntax near the keyword 'count'.
```

Batches that violate a batch rule also generate error messages. Here are some examples of illegal batches:

```
create table test
    (column1 char(10), column2 int)
insert test
    values ("hello", 598)
select * from test
create procedure testproc as
    select column1 from test
go
```

```
Msg 111, Level 15, State 7:
Line 6:
CREATE PROCEDURE must be the first command in a
query batch.
```

```
create default phonedflt as "UNKNOWN"
sp_bindefault phonedflt, "authors.phone"
go
```

```
Msg 102, Level 15, State 1:
Procedure 'phonedflt', Line 2:
Incorrect syntax near 'sp_bindefault'.
```

The next batch works if you are already in the database you specify in the `use` statement. If you try it from another database such as `master`, however, you see an error message.

```
use pubs2
select * from titles
go
```

```
Msg 208, Level 16, State 1:
Server 'hq', Line 2:
titles not found. Specify owner.objectname or use sp_help to check whether the
object exists (sp_help may produce lots of output)
```

```
drop table test
create table test
(column1 char(10), column2 int)
go
```

```
Msg 2714, Level 16, State 1:
Server 'hq', Line 2:
There is already an object named 'test' in the
database.
```

16.1.2 Batches Submitted as Files

You can submit one or more batches of Transact-SQL statements to `isql` from an operating system file. A file can include more than one batch, that is, more than one collection of statements, each terminated by the word "go."

For example, an operating system file might contain the following three batches:

```
use pubs2
go
select count(*) from titles
select count(*) from authors
go
create table hello
   (column1 char(10), column2 int)
insert hello
   values ("hello", 598)
select * from hello
go
```

Here are the results of submitting this file to the `isql` utility:

```
-------------
           18

(1 row affected)
-------------
           23

(1 row affected)
column1      column2
---------    ---------
hello              598

(1 row affected)
```

See `isql` in the *Utility Guide* for environment-specific information about running batches stored in files.

16.2 Control-of-Flow Language Usage

Use control-of-flow language with interactive statements, in batches, and in stored procedures.

This table lists the control-of-flow and related keywords and their functions.

Keyword	Function
`if`	Defines conditional execution.
`…else`	Defines alternate execution when the `if` condition is false.
`case`	Defines conditional expressions using `when…then` statements instead of `if…else`.
`begin`	Beginning of a statement block.

Keyword	Function
…end	End of a statement block.
while	Repeat performance of statements while condition is true.
break	Exit from the end of the next outermost while loop.
…continue	Restart while loop.
declare	Declare local variables.
goto label	Go to label:, a position in a statement block.
return	Exit unconditionally.
waitfor	Set delay for command execution.
print	Print a user-defined message or local variable on user's screen.
raiserror	Print a user-defined message or local variable on user's screen and set a system flag in the global variable @@<error>.
/* <comment> */ or -- <comment>	Insert a comment anywhere in a Transact-SQL statement.

16.2.1 if...else

The keyword if, with or without its companion else, introduces a condition that determines whether the next statement is executed. The Transact-SQL statement executes if the condition is satisfied, that is, if it returns true.

The else keyword introduces an alternate Transact-SQL statement that executes when the if condition returns false.

The syntax for if and else is:

```
if
          <boolean_expression>
          <statement>
     [else
           [if <boolean_expression>]
           <statement> ]
```

A Boolean expression returns true or false. It can include a column name, a constant, any combination of column names and constants connected by arithmetic or bitwise operators, or a subquery, as long as the subquery returns a single value. If the Boolean expression contains a select statement, the select statement must be enclosed in parentheses, and it must return a single value.

Here is an example of using `if` alone:

```
if exists (select postalcode from authors
         where postalcode = "94705")
print "Berkeley author"
```

If one or more of the postal codes in the authors table has the value “94705,” the message “Berkeley author” prints. The `select` statement in this example returns a single value, either TRUE or FALSE, because it is used with the keyword `exists`. The `exists` keyword functions here just as it does in subqueries.

This example uses both `if` and `else`, and tests for the presence of user-created objects that have ID numbers greater than 50. If user objects exist, the `else` clause selects their names, types, and ID numbers.

```
if (select max(id) from sysobjects) < 50
   print "There are no user-created objects in this database."
else
   select name, type, id from sysobjects
   where id > 50 and type = "U"
```

```
(0 rows affected)

 name          type   id
------------   ----   ---------
 authors       U       16003088
 publishers    U       48003202
 roysched      U       80003316
 sales         U      112003430
 salesdetail   U      144003544
 titleauthor   U      176003658
 titles        U      208003772
 stores        U      240003886
 discounts     U      272004000
 au_pix        U      304004114
 blurbs        U      336004228
 friends_etc   U      704005539
 test          U      912006280
 hello         U     1056006793

(14 rows affected)
```

`if...else` constructs are frequently used in stored procedures where they test for the existence of some parameter.

`if` tests can nest within other `if` tests, either within another `if` or following an `else`. The expression in the `if` test can return only one value. Also, for each `if...else` construct, there can be one select statement for the `if` and one for the `else`. To include more than one select statement, use the `begin...end` keywords. The maximum number of `if` tests you can nest varies, depending on the complexity of the `select` statements (or other language constructs) you include with each `if...else` construct.

Related Information

Subqueries: Queries Within Other Queries [page 257]

16.2.2 case Expression

`case` expression simplifies many conditional Transact-SQL constructs. Instead of using a series of `if` statements, `case` expression allows you to use a series of conditions that return the appropriate values when the conditions are met. `case` expression is ANSI-SQL-compliant.

With `case` expression, you can:

- Simplify queries and write more efficient code
- Convert data between the formats used in the database (such as `int`) and the format used in an application (such as `char`)
- Return the first non-null value in a list of columns
- Write queries that avoid division by 0
- Compare two values and return the first value if the values do not match, or a null value if the values do match

`case` expression includes the keywords `case`, `when, then`, `coalesce`, and `nullif`. `coalesce` and `nullif` are an abbreviated form of `case` expression. See the *Reference Manual: Commands*.

16.2.2.1 case Expression for Alternative Representation

Use `case` expression to represent data in a manner that is more meaningful, for example, to display a phrase rather than a binary (1, 0) setting.

For example, the `pubs2` database stores a 1 or a 0 in the `contract` column of the `titles` table to indicate the status of the book's contract. However, in your application code or for user interaction, you may prefer to use the words "Contract" or "No Contract" to indicate the status of the book. To select the type from the `titles` table using the alternative representation:

```
select title, "Contract Status" =
    case
        when contract = 1 then "Contract"
        when contract = 0 then "No Contract"
    end
from titles
```

```
title                                   Contract Status
-----                                   ---------------
The Busy Executive's Database Guide     Contract
Cooking with Computers: Surreptitio     Contract
You Can Combat Computer Stress!         Contract
. . .
The Psychology of Computer Cooking      No Contract
. . .
Fifty Years in Buckingham Palace        Contract
Sushi, Anyone?                          Contract

(18 rows affected)
```

16.2.2.2 case and Division by Zero

`case` expression allows you to write queries that avoid division by zero (called exception avoidance).

This example divides the `total_sales` column for each book by the `advance` column. The query results in division by zero when the query attempts to divide the `total_sales` (2032) of `title_id` MC2222 by the `advance` (0.00):

```
select title_id, total_sales, advance, total_sales/advance from titles
```

```
title_id   total_sales          advance
-------    -----------          ---------     ------
BU1032     4095                  5,000.00     0.82
BU1111     3876                  5,000.00     0.78
BU2075     18722                10,125.00     1.85
BU7832     4095                  5,000.00     0.82

Divide by zero occurred.
```

You can use a `case` expression to avoid this by not allowing the zero to figure in the equation. In this example, when the query comes across the zero, it returns a predefined value, rather than performing the division:

```
select title_id, total_sales, advance, "Cost Per Book" =
     case
           when advance != 0
           then convert(char, total_sales/advance)
           else "No Books Sold"
     end
from titles
```

```
title_id       total_sales     advance          Cost Per Book
--------       -----------     ----------       -------------
BU1032                4095       5,000.00                0.82
BU1111                3876       5,000.00                0.78
BU2075               18722      10,125.00                1.85
BU7832                4095       5,000.00                0.82
MC2222                2032           0.00       No Books Sold
MC3021               22246      15,000.00                1.48
MC3026                NULL           NULL       No Books Sold
. . .
TC3218                 375       7,000.00                0.05
TC4203               15096       4,000.00                3.77
TC7777                4095       8,000.00                0.51

(18 rows affected)
```

The division by zero for `title_id` MC2222 no longer prevents the query from running. Also, the null values for MC3021 do not prevent the query from running.

`case` does not avoid division by 0 errors if the divisor evaluates to a constant expression, because SAP ASE evaluates constant expressions before executing the `case` logic. This sometimes causes division by zero. As a work around:

- Use `nullif()`. For example:

  ```
  (x/nullif(@foo.0)0
  ```

- Include column values so that they cancel each other, forcing SAP ASE to evaluate the expression for each row. For example:

```
(x/(@foo + (col1 - col1))
```

16.2.2.3 rand Functions in case Expressions

Expressions that reference the `rand` function, the `getdate` function, and so on, produce different values each time they are evaluated. This can yield unexpected results when you use these expressions in certain `case` expressions.

For example, the SQL standard specifies that `case` expressions with this form:

```
case expression
    when value1 then result1
    when value2 then result2
    when value3 then result3
...
end
```

are equivalent to this form of `case` expression:

```
case expression
    when expression=value1 then result1
    when expression=value2 then result2
    when expression=value3 then result3
...
end
```

This definition explicitly requires that the expression be evaluated repeatedly in each `when` clause that is examined. This definition of `case` expressions affects `case` expressions that reference functions such as the `rand` function. For example, this `case` expression:

```
select
CASE convert(int, (RAND() * 3))
        when 0 then "A"
        when 1 then "B"
        when 2 then "C"
        when 3 then "D"
    else "E"
end
```

is defined to be equivalent to this one, according to the SQL standard:

```
 select
CASE
        when convert(int, (RAND() * 3)) = 0 then "A"
        when convert(int, (RAND() * 3)) = 1 then "B"
        when convert(int, (RAND() * 3)) = 2 then "C"
        when convert(int, (RAND() * 3)) = 3 then "D"
    else "E"
end
```

In this form, a new `rand` value is generated for each `when` clause, and the `case` expression frequently produces the result “E”.

16.2.2.4 case Expression Results

`case` expression uses a series of alternative result expressions.

The series of alternative result expressions in the example below are: `<R1>`, `<R2,>` ..., `<Rn>`, which are specified by the `then` and `else` clauses.

The rules for determining the datatype of a `case` expression are based on the same rules that determine the datatype of a column in a `union` operation. For example:

```
case
          when <search_condition1> then <R1>
          when <search_condition2> then <R2>
          ...
          else <Rn
>     end
```

The datatypes of the result expressions `<R1>`, `<R2>`, ..., `<Rn>` are used to determine the overall datatype of `case`. The same rules that determine the datatype of a column of a `union` that specifies `<n>` tables, and has the expressions `<R1>`, `<R2, >`..., `<Rn>` as the `<i>`th column, also determine the datatype of a `case` expression. The datatype of `case` is determined in the same manner as by the following query:

```
select...R1...from ...
union
select...R2...from...
union...
...
select...Rn...from...
```

Not all datatypes are compatible, and if you specify two datatypes that are incompatible (for example, `<char>` and `<int>`), your Transact-SQL query fails. See the *Reference Manual: Building Blocks*.

16.2.2.5 case Expressions and set ansinull

Case expressions may produce different results for null values if you enable `set ansinull` and include a `when NULL` clause in certain queries.

This occures with query structures similar to:

```
select CVT =
   case <case_expression>
      when NULL  then 'YES'
      else 'NO'
   end
from A
```

If you set `ansinull` to `off` (the default) these `case` expressions match the NULL values and the predicate evaluates to `true` for NULL values (similar to how the query processor evaluates a `where` clause that includes a NULL value). If you set `ansinull` to `on` the `case` expression cannot compare the NULL values, and evaluates the predicate to `false`. For example:

```
select CVT =
   case advance
```

```
          when  NULL  then 'YES'
          else  'NO'
   end,
advance from titles
```

```
--- ------------------------
NO                 5,000.00
NO                15,000.00
YES                    NULL
NO                 7,000.00
NO                 8,000.00
YES                    NULL
NO                  7,000.00
```

However, if you enable `set ansinull` and run the same query, `case` expressions returns a `no` value when it encounters a NULL value:

```
set ansinull on
select CVT =
   case advance
          when  NULL  then 'YES'
          else  'NO'
   end,
advance from titles
```

```
 CVT advance
--- ------------------------
NO                  5,000.00
NO                 15,000.00
NO                     NULL
NO                  7,000.00
NO                  8,000.00
NO                     NULL
NO                   7,000.00
```

See the *Reference Manual: Commands*.

16.2.2.6 case Expression Requires at Least one Non-Null Result

At least one result from the `case` expression must return a value other than null.

This query:

```
select price,
    case
        when title_id like "%" then NULL
        when pub_id like "%" then NULL
    end
from titles
```

returns this error message:

```
All result expressions in a CASE expression must not be NULL
```

16.2.2.7 Determining the Result Set

Use `case` expression to test for conditions that determine the result set.

The syntax is:

```
case
          when <search_condition1> then <result1>
          when <search_condition2> then <result2>
          . . .
          when <search_conditionn> then <resultn>
          else <resultx>
     end
```

where `<search_condition>` is a logical expression, and `<result>` is an expression.

If `<search_condition1>` is true, the value of `case` is `<result1>`; if `<search_condition1>` is not true, `<search_condition2 >`is checked. If `<search_condition2>` is true, the value of `case` is `<result2>`, and so on. If none of the search conditions are true, the value of `case` is `<resultx>`. The `else` clause is optional. If it is not used, the default is `else NULL`. `end` indicates the end of the `case` expression.

The total sales of each book for each store are kept in the `salesdetail` table. To show a series of ranges for the book sales, you can track how each book sold at each store:

- Books that sold less than 1000 (low-selling books)
- Books that sold between 1000 and 3000 (medium-selling books)
- Books that sold more than 3000 (high-selling books)

Write the following query:

```
select stor_id, title_id, qty, "Book Sales Catagory" =
     case
           when qty < 1000
              then "Low Sales Book"
           when qty >= 1000 and qty <= 3000
              then "Medium Sales Book"
           when qty > 3000
           then "High Sales Book"
     end
from salesdetail
group by title_id
```

```
stor_id     title_id      qty      Book Sales Catagory
-------     --------      ----     -------------------
5023        BU1032        200      Low Sales Book
5023        BU1032       1000      Low Sales Book
7131        BU1032        200      Low Sales Book
. . .
7896        TC7777         75      Low Sales Book
7131        TC7777         80      Low Sales Book
5023        TC7777       1000      Low Sales Book
7066        TC7777        350      Low Sales Book
5023        TC7777       1500      Medium Sales Book
5023        TC7777       1090      Medium Sales Book

(116 rows affected)
```

The following example selects the `titles` from the `titleauthor` table according to the author's royalty percentage (`<royaltyer>`) and then assigns each title with a value of high, medium, or low royalty:

```
select title, royaltyper, "Royalty Category" =
   case
      when (select avg(royaltyper) from titleauthor tta
        where t.title_id = tta.title_id) > 60 then "High Royalty"
      when (select avg(royaltyper) from titleauthor tta
        where t.title_id = tta.title_id) between 41 and 59
      then "Medium Royalty"
      else "Low Royalty"
   end
from titles t, titleauthor ta
where ta.title_id = t.title_id
order by title
```

```
title                                  royaltyper   royalty Category
-------                                ----------   ----------------
But Is It User Friendly?               100          High Royalty
Computer Phobic and Non-Phobic Ind     25           Medium Royalty
Computer Phobic and Non-Phobic Ind     75           Medium Royalty
Cooking with Computers: Surreptiti     40           Medium Royalty
Cooking with Computers: Surreptiti     60           Medium Royalty
Emotional Security: A New Algorith     100          High Royalty
. . .
Sushi, Anyone?                         40           Low Royalty
The Busy Executive's Database Guide    40           Medium Royalty
The Busy Executive's Database Guide    60           Medium Royalty
The Gourmet Microwave                  75           Medium Royalty
You Can Combat Computer Stress!        100          High Royalty

(25 rows affected)
```

16.2.2.8 case and Value Comparisons

`case` can be used for value comparisons. It allows only an equality check between two values; no other comparisons are allowed.

The syntax is:

```
case <valueT>
          when <value1> then <result1>
          when <value2> then <result2>
          . . .
          when <valuen> then <resultn>
          else <result>x
     end
```

where `<value >`and`< result >`are expressions.

If `<valueT >`equals `<value1>`, the value of the `case` is `<result1>`. If `<valueT >`does not equal value1, `<valueT>` is compared to `<value2>`. If `<valueT>` equals `<value2>`, then the value of the `case` is `<result2, >`and so on.`< >`If `<valueT>` does not equal the value of `<value1>` through `<valuen>`, the value of the `case` is `<resultx>`.

At least one result must be non-null. All the result expressions must be compatible. Also, all `values` must be compatible.

The syntax described above is equivalent to:

```
case
          when <valueT> = <value>1 then <result1>
          when <valueT> = <value>2 then <result2>
          . . .
          when <valueT> = <value>n then <resultn>
          else <resultx>
     end
```

This is the same format used for `case` and search conditions.

The following example selects the `title` and `pub_id` from the `titles` table and specifies the publisher for each book based on the `pub_id`:

```
select title, pub_id, "Publisher" =
   case pub_id
     when "0736" then "New Age Books"
     when "0877" then "Binnet & Hardley"
     when "1389" then "Algodata Infosystems"
     else "Other Publisher"
   end
from titles
order by pub_id
```

```
title                          pub_id       Publisher
-----                          ------       -------------
Life Without Fear                0736       New Age Books
Is Anger the Enemy?              0736       New Age Books
You Can Combat Computer          0736       New Age Books
. . .
Straight Talk About Computers    1389       Algodata Infosystems
The Busy Executive's Database    1389       Algodata Infosystems
Cooking with Computers: Surre    1389       Algodata Infosystems

(18 rows affected)
```

This is equivalent to the following query, which uses a `case` and search condition syntax:

```
select title, pub_id, "Publisher" =
   case
     when pub_id = "0736" then "New Age Books"
     when pub_id = "0877" then "Binnet & Hardley"
     when pub_id = "1389" then "Algodata Infosystems"
     else "Other Publisher"
   end
from titles
order by pub_id
```

Related Information

Determining the Result Set [page 449]

16.2.2.9 coalesce

coalesce examines a series of values (<value1>, <value2>, ..., <valuen>) and returns the first non-null value.

The syntax of coalesce is:

```
coalesce(<value1>, <value2>, ..., <valuen>)
```

where <value1>, <value2>, ..., <valuen> are expressions. If <value1> is non-null, the value of coalesce is <value1>; if <value1> is null, <value>2 is examined, and so on. The examination continues until a non-null value is found. The first non-null value becomes the value of coalesce.

When you use coalesce, SAP ASE translates it internally to:

```
case
    when <value1> is not NULL then <value1>
    when <value2> is not NULL then <value>2
    . . .
    when <valuen-1> is not NULL then <valuen-1>
    else <valuen>
end
```

<valuen-1> refers to the next to last value, before the final value, <valuen>.

The example below uses coalesce to determine whether a store orders a low quantity (more than 100 but less than 1000) or a high quantity of books (more than 1000):

```
select stor_id, discount, "Quantity" =
    coalesce(lowqty, highqty)
from discounts
```

```
stor_id discount    Quantity
------- --------   ---------   ----
NULL               10.500000   NULL
NULL                6.700000    100
NULL               10.000000   1001
8042                5.000000   NULL

(4 rows affected)
```

16.2.2.10 nullif

Use nullif to find any missing, unknown, or inapplicable information that is stored in an encoded form.

For example, values that are unknown are sometimes historically stored as -1. Using nullif, you can replace the -1 values with null and get the null behavior defined by Transact-SQL. The syntax is:

```
nullif(<value1>, <value2>)
```

If <value1> equals <value2>, nullif returns NULL. If <value1> does not equal <value2>, nullif returns <value1>. <value1> and <value2> are expressions, and their datatypes must be comparable.

When you use `nullif`, SAP ASE translates it internally to:

```
case
          when <value1> = <value2> then NULL
          else <value1>
     end
```

For example, the `titles` table uses the value "UNDECIDED" to represent books whose type category is not yet determined. The following query performs a search on the `titles` table for book types; any book whose type is "UNDECIDED" is returned as type NULL (the following output is reformatted for display purposes):

```
select title, "type"=
   nullif(type, "UNDECIDED")
from titles
```

```
title                                   type
-----                                   --------
The Busy Executive's Database Guide     business
Cooking with Computers: Surreptiti      business
You Can Combat Computer Stress!         business
. . .
The Psychology of Computer Cooking      NULL
Fifty Years in Buckingham Palace K      trad_cook
Sushi, Anyone?                          trad_cook

(18 rows affected)
```

The Psychology of Computing is stored in the table as "UNDECIDED," but the query returns it as type NULL.

16.2.3 begin...end

The `begin` and `end` keywords enclose a series of statements so they are treated as a unit by control-of-flow constructs like `if...else`. A series of statements enclosed by `begin` and `end` is called a statement block.

The syntax of `begin...end` is:

```
begin
          <statement block>
end
```

For example:

```
if (select avg(price) from titles) < $15
begin
   update titles
   set price = price * 2

   select title, price
   from titles
   where price > $28
end
```

Without `begin` and `end`, the `if` condition applies only to the first Transact-SQL statement. The second statement executes independently of the first.

`begin...end` blocks can nest within other `begin...end` blocks.

16.2.4 while and break...continue

`while` sets a condition for the repeated execution of a statement or statement block. The statements are executed repeatedly as long as the specified condition is true.

The syntax is:

```
while <boolean_expression>
     <statement>
```

In this example, the `select` and `update` statements are repeated, as long as the average price remains less than $30:

```
while (select avg(price) from titles) < $30
begin
   select title_id, price
   from titles
   where price > $20
   update titles
   set price = price * 2
end
```

```
(0 rows affected)

title_id    price
------      -------
PC1035      22.95
PS1372      21.59
TC3218      20.95

(3 rows affected)
(18 rows affected)
(0 rows affected)
title_id    price
------      -------
BU1032      39.98
BU1111      23.90
BU7832      39.98
MC2222      39.98
PC1035      45.90
PC8888      40.00
PS1372      43.18
PS2091      21.90
PS3333      39.98
TC3218      41.90
TC4203      23.90
TC7777      29.98

(12 rows affected)
(18 rows affected)
(0 rows affected)
```

`break` and `continue` control the operation of the statements inside a `while` loop. `break` causes an exit from the `while` loop. Any statements that appear after the `end` keyword that marks the end of the loop are executed. `continue` causes the `while` loop to restart, skipping any statements after `continue` but inside the loop. `break` and `continue` are often activated by an `if` test.

The syntax for `break...continue` is:

```
while <boolean> <expression>
     begin
```

```
        <statement>
        [<statement>]...
        break
        [<statement>]...
        continue
        [<statement>]...
    end
```

Here is an example using `while`, `break`, `continue`, and `if` that reverses the inflation caused by the previous examples. As long as the average price remains more than $20, all prices are reduced by half. The maximum price is then selected. If it is less than $40, the `while` loop exits; otherwise, it attempts to loop again. `continue` allows `print` to execute only when the average is more than $20. After the `while` loop ends, a message and a list of the highest priced books print.

```
while (select avg(price) from titles) > $20
begin
    update titles
        set price = price / 2
    if (select max(price) from titles) < $40
        break
    else
        if (select avg(price) from titles) < $20
            continue
    print "Average price still over $20"
end

select title_id, price from titles
    where price > $20

print "Not Too Expensive"
```

```
(18 rows affected)
(0 rows affected)
(0 rows affected)
Average price still over $20
(0 rows affected)
(18 rows affected)
(0 rows affected)

title_id    price
--------    -------
PC1035        22.95
PS1372        21.59
TC3218        20.95

(3 rows affected)
Not Too Expensive
```

If two or more `while` loops are nested, `break` exits to the next outermost loop. First, all the statements after the `end` of the inner loop execute. Then, the outer loop restarts.

16.2.5 declare and Local Variables

Local variables are declared, named, and typed with the `declare` keyword and are assigned an initial value with a `select` statement; this must all happen within the same batch or procedure.

Related Information

Local Variables [page 463]

16.2.6 goto

The `goto` keyword causes unconditional branching to a user-defined label. `goto` and labels can be used in stored procedures and batches.

A label's name must follow the rules for identifiers and must be followed by a colon when it is first given. It is not followed by a colon when it is used with `goto`.

The syntax for `goto` is:

```
<label>:
     goto <label>
```

This example uses `goto` and a label, a `while` loop, and a local variable as a counter:

```
declare @count smallint
select @count = 1
restart:
print "yes"
select @count = @count + 1
while @count <=4
   goto restart
```

`goto` is usually dependent on a `while` or `if` test or some other condition, to avoid an endless loop between `goto` and the label.

16.2.7 return

The `return` keyword exits from a batch or procedure unconditionally. It can be used at any point in a batch or a procedure. When used in stored procedures, `return` can accept an optional argument to return a status to the caller. Statements after `return` are not executed.

The syntax is:

```
return [<int_expression>]
```

This stored procedure uses `return` as well as `if...else` and `begin...end`:

```
create procedure findrules @nm varchar(30) = null as
if @nm is null
begin
  print "You must give a user name"
  return
end
else
begin
   select sysobjects.name, sysobjects.id, sysobjects.uid
   from sysobjects, master..syslogins
   where master..syslogins.name = @nm
   and sysobjects.uid = master..syslogins.suid
   and sysobjects.type = "R"
end
```

If no user name is given as a parameter when `findrules` is called, the `return` keyword causes the procedure to exit after a message has been sent to the user's screen. If a user name is given, the names of the rules owned by the user are retrieved from the appropriate system tables.

`return` is similar to the `break` keyword used inside `while` loops.

Related Information

Stored Procedures [page 498]

16.2.8 print

The `print` keyword, used in the previous example, displays a user-defined message or the contents of a local variable on the user's screen.

The local variable must be declared within the same batch or procedure in which it is used. The message can be up to 255 bytes long.

The syntax is:

```
print {<format_string> | @<local_variable> |
          @@<global>_<variable>} [,<arg_list>]
```

For example:

```
if exists (select postalcode from authors
   where postalcode = "94705")
print "Berkeley author"
```

Here is how to use `print` to display the contents of a local variable:

```
declare @msg char(50)
select @msg = "What's up, doc?"
print @msg
```

print recognizes placeholders in the character string to be printed out. Format strings can contain up to 20 unique placeholders in any order. These placeholders are replaced with the formatted contents of any arguments that follow <format_string> when the text of the message is sent to the client.

To allow reordering of the arguments when format strings are translated to a language with a different grammatical structure, placeholders are numbered. A placeholder for an argument appears in this format: <%nn!>. The components are a percent sign, followed by an integer from 1 to 20, followed by an exclamation point. The integer represents placeholder position in the string in the original language. "%1!" is the first argument in the original version, "%2!" is the second argument, and so on. Indicating the position of the argument makes translations correctly, even when the order in which the arguments appear in the target language is different from their order in the source language.

For example, assume the following is an English message:

```
%1! is not allowed in %2!.
```

The German version of this message is:

```
%1! ist in %2! nicht zulässig.
```

The Japanese version of the message is:

```
%2! の中で %1! は許されません。
```

The characters "%1!" are in different places in the phrase. In this example, "%1!" in English, German, and Japanese represents the same argument, and "%2!" also represents a single argument in all three languages.

You cannot skip placeholder numbers when using placeholders in a format string, although you do not need to use placeholders in numerical order. For example, you cannot have placeholders 1 and 3 in a format string without having placeholder 2 in the same string.

The optional <arg_list> can be a series of either variables or constants. An argument can be any datatype except text or image; it is converted to the char datatype before it is included in the final message. If no argument list is provided, the format string must be the message to be printed, without any placeholders.

The maximum output string length of <format_string> plus all arguments after substitution is 512 bytes.

16.2.9 raiserror

raiserror displays a user-defined error or local variable message on the user's screen and sets a system flag to record the fact that an error has occurred.

As with print, the local variable must be declared within the same batch or procedure in which it is used. The message can be up to 255 characters long.

The syntax for raiserror is:

```
raiserror <error_number>
     [{<format_string> | @<local_variable>}] [, <arg_list>]
     [<extended_value>  = <extended_value>  [{,
     <extended_value>  = <extended_value>}...]]
```

The <error_number> is placed in the global variable <error@@>, which stores the error number most recently generated by SAP ASE. Error numbers for user-defined error messages must be greater than 17,000. If the <error_number> is between 17,000 and 19,999, and <format_string> is missing or empty (“ ”), SAP ASE retrieves error message text from the sysmessages table in the master database. These error messages are used chiefly by system procedures.

The length of the <format_string> alone is limited to 255 bytes; the maximum output length of <format_string> plus all arguments is 512 bytes. Local variables used for raiserror messages must be char or varchar. The <format_string >or variable is optional; if you do not include one, SAP ASE uses the message corresponding to the <error_number> from sysusermessages in the default language. As with print, you can substitute variables or constants defined by <arg_list> in the <format_string>.

You can define extended error data for use by an Open Client application (when you include <extended_values> with raiserror). For more information about extended error data, see your Open Client documentation or the *Reference Manual: Commands*.

Use raiserror instead of print when you want to store an error number in <error@@>. This example uses raiserror in the procedure named findrules:

```
raiserror 99999 "You must give a user name"
```

The severity level of all user-defined error messages is 16, which indicates that the user has made a nonfatal mistake.

16.2.10 Create Messages for print and raiserror

You can call messages from sysusermessages using either print or raiserror with sp_getmessage. Use sp_addmessage to create a set of messages.

This example uses sp_addmessage, sp_getmessage, and print to install a message in sysusermessages in both English and German, retrieve it for use in a user-defined stored procedure, and print it:

```
/*
** Install messages
** First, the English (langid = NULL)
*/
set language us_english
go
sp_addmessage 25001,
  "There is already a remote user named '%1!' for remote server '%2!'."
go
/* Then German*/
sp_addmessage 25001,
        "Remotebenutzername '%1!' existiert bereits auf dem Remoteserver '
%2!'.","german"
go
create procedure test_proc @remotename varchar(30),
            @remoteserver varchar(30)
as
        declare @msg varchar(255)
        declare @arg1 varchar(40)
        /*
        ** check to make sure that there is not
        ** a @remotename for the @remoteserver.
```

```
        */
        if exists (select *
            from master.dbo.sysremotelogins l,
                master.dbo.sysservers s
            where l.remoteserverid = s.srvid
                and s.srvname = @remoteserver
                and l.remoteusername = @remotename)
        begin
            exec sp_getmessage 25001, @msg output
            select @arg1=isnull(@remotename, "null")
            print @msg, @arg1, @remoteserver
            return (1)
        end
return(0)
go
```

To drop a user-defined message, use `sp_dropmessage`. To change a message, drop it with `sp_dropmessage` and add it again with `sp_addmessage`.

Related Information

Create Error Messages for Constraints [page 107]

16.2.11 waitfor

The `waitfor` keyword specifies a specific time of day, a time interval, or an event at which the execution of a statement block, stored procedure, or transaction is to occur.

The syntax is:

```
waitfor {delay "<time>" | time "<time>" | errorexit |  processexit | mirrorexit}
```

where `delay '<time>'` instructs SAP ASE to wait until the specified period of time has passed. `time '<time>'` instructs SAP ASE to wait until the specified time, given in the valid format for `datetime` data.

However, you cannot specify dates—the date portion of the `datetime` value is not allowed. The time you specify with `waitfor time` or `waitfor delay` can include hours, minutes, and seconds—up to a maximum of 24 hours. Use the format “hh:mm:ss”.

For example, this command instructs SAP ASE to wait until 4:23 p.m.:

```
waitfor time "16:23"
```

This command instructs SAP ASE to wait 1 hour and 30 minutes:

```
waitfor delay "01:30"
```

`errorexit` instructs SAP ASE to wait until a process terminates abnormally. `processexit` waits until a process terminates for any reason. `mirrorexit` waits until a read or write to a mirrored device fails.

You can use `waitfor errorexit` with a procedure that kills the abnormally terminated process to free system resources that would otherwise be taken up by an infected process. To find out which process is infected, use `sp_who` to check the `sysprocesses` table.

The following example instructs SAP ASE to wait until 2:20 p.m. Then it updates the `chess` table with the next move and executes a stored procedure called `sendmessage`, which inserts a message into one of Judy's tables notifying her that a new move now exists in the `chess` table.

```
begin
waitfor time "14:20"
insert chess(next_move)
values("Q-KR5")
execute sendmessage "Judy"
end
```

To send the message to Judy after 10 seconds instead of waiting until 2:20, substitute this `waitfor` statement in the preceding example:

```
waitfor delay "0:00:10"
```

After you give the `waitfor` command, you cannot use your connection to SAP ASE until the time or event that you specified occurs.

Related Information

Enter Times [page 214]

16.2.12 Comments

Use the comment notation to attach comments to statements, batches, and stored procedures. Comments are not executed and have no maximum length.

You can insert a comment on a line by itself or at the end of a command line. Two comment styles are available: the "slash-asterisk" style:

```
/* text of comment */
```

and the "double-hyphen" style:

```
-- text of comment
```

16.2.12.1 Slash-Asterisk Style Comments

The /* style comment is a Transact-SQL extension. Multiple-line comments are acceptable, as long as each comment starts with "/*" and ends with "*/". Everything between "/*" and "*/" is treated as part of the comment. The /* form permits nesting.

A stylistic convention often used for multiple-line comments is to begin the first line with "/*" and subsequent lines with "**". End such a comment with "*/" as usual:

```
select * from titles
/* A comment here might explain the rules
** associated with using an asterisk as
** shorthand in the select list.*/
where price > $5
```

This procedure includes several comments:

```
/* this procedure finds rules by user name*/
create procedure findmyrule @nm varchar(30) = null
as
if @nm is null
begin
  print "You must give a user name"
  return
  print "I have returned"
/* this statement follows return,
** so won't be executed */
end
else      /* print the rule names and IDs, and
          the user ID */
  select sysobjects.name, sysobjects.id,
     sysobjects.uid
  from sysobjects, master..syslogins
  where  master..syslogins.name = @nm
  and sysobjects.uid = master..syslogins.suid
  and sysobjects.type = "R"
```

16.2.12.2 Double-Hyphen Style Comments

This comment style begins with two consecutive hyphens followed by a space (--) and terminates with a new-line character. Therefore, you cannot use multiple-line comments.

SAP ASE does not interpret two consecutive hyphens within a string literal or within a /*-style comment as signaling the beginning of a comment.

To represent an expression that contains two consecutive minus signs (binary followed by unary), put a space or an opening parenthesis between the two hyphens.

Following are examples:

```
-- this procedure finds rules by user name
create procedure findmyrule @nm varchar(30) = null
as
if @nm is null
begin
  print "You must give a user name"
  return
```

```
  print "I have returned"
-- each line of a multiple-line comment
-- must be marked separately.
end
else        -- print the rule names and IDs, and
        -- the user ID
  select sysobjects.name, sysobjects.id,
     sysobjects.uid
  from sysobjects, master..syslogins
  where  master..syslogins.name = @nm
  and sysobjects.uid = master..syslogins.suid
  and sysobjects.type = "R"
```

16.3 Local Variables

Local variables are often used as counters for `while` loops or `if...else` blocks in a batch or stored procedure.

When they are used in stored procedures, they are declared for automatic, noninteractive use by the procedure when it executes. You can use variables nearly anywhere the Transact-SQL syntax indicates that an expression can be used, such as `<char_expr>`, `<integer_expression>`, `<numeric_expr>`, or `<float_expr>`.

To `declare` a local variable's name and datatype, use:

```
declare @<variable_name> <datatype>
   [, @<variable_name> <datatype>]...
```

The variable name must be preceded by the @ sign and conform to the rules for identifiers. Specify either a user-defined datatype or a system-supplied datatype other than `text`, `image`, or `sysname`.

In terms of memory and performance, this is more efficient:

```
declare @a int, @b char(20), @c float
```

than this:

```
declare @a int
declare @b char(20)
declare @c float
```

16.3.1 Local Variables and select Statements

When you declare a variable, it has the value NULL. Use a `select` statement to assign values to local variables.

As with `declare` statements, it is more efficient to use:

```
select @a = 1, @b = 2, @c = 3
```

than:

```
select @a = 1
select @b = 2
select @c = 3
```

See the *Reference Manual: Commands*.

Do not use a single `select` statement to assign a value to one variable and then to another whose value is based on the first. Doing so can yield unpredictable results. For example, the following queries both try to find the value of`< @c2>`. The first query yields NULL, while the second query yields the correct answer, 0.033333:

```
/* this is wrong*/
declare @c1 float, @c2 float
select @c1 = 1000/1000, @c2 = @c1/30
select @c1, @c2

/* do it this way */
declare @c1 float, @c2 float
select @c1 = 1000/1000
select @c2 = @c1/30
select @c1 , @c2
```

You cannot use a `select` statement that assigns values to variables to also return data to the user. The first `select` statement in the following example assigns the maximum price to the local variable `<@veryhigh>`; the second `select` statement is needed to display the value:

```
declare @veryhigh money
select @veryhigh = max(price)
   from titles
select @veryhigh
```

If the `select` statement that assigns values to a variable returns more than one value, the last value that is returned is assigned to the variable. The following query assigns the variable the last value returned by "select advance from titles."

```
declare @m money
select @m = advance from titles
select @m
```

```
(18 rows affected)
------------------------
                8,000.00

(1 row affected)
```

The assignment statement indicates how many rows were affected (returned) by the `select` statement.

If a `select` statement that assigns values to a variable fails to return any values, the variable is left unchanged by the statement.

Local variables can be used as arguments to `print` or `raiserror`.

Related Information

Use the set Clause with Update [page 372]

16.3.2 Local Variables and update Statements

You can assign variables directly in an `update` statement. You do not need to use a `select` statement to assign a value to a variable. When you declare a variable, it has the value NULL.

See the *Reference Manual: Commands*.

16.3.3 Local Variables and Subqueries

A subquery that assigns a value to the local variable can return only one value.

Here are some examples:

```
declare @veryhigh money
select @veryhigh = max(price)
   from titles
if @veryhigh > $20
   print "Ouch!"
declare @one varchar(18), @two varchar(18)
select @one = "this is one", @two = "this is two"
if @one = "this is one"
   print "you got one"
if @two = "this is two"
   print "you got two"
else print "nope"
declare @tcount int, @pcount int
select @tcount = (select count(*) from titles),
   @pcount = (select count(*) from publishers)
select @tcount, @pcount
```

16.3.4 Local Variables and while Loops and if...else Blocks

You can use local variables in a counter in a `while` loop, for performing matching in a `where` clause and in an `if` statement, and for setting and resetting values in `select` statements.

```
/* Determine if a given au_id has a row in au_pix*/
/* Turn off result counting */
set nocount on
/* declare the variables */
declare @c int,
        @min_id varchar(30)
/*First, count the rows*/
select @c = count(*) from authors
/* Initialize @min_id to "" */
select @min_id = ""
/* while loop executes once for each authors row */
while @c > 0
begin
        /*Find the smallest au_id*/
        select @min_id = min(au_id)
            from authors
            where au_id > @min_id
        /*Is there a match in au_pix?*/
        if exists (select au_id
            from au_pix
```

```
            where au_id = @min_id)
            begin
            print "A Match! %1!", @min_id
        end
      select @c = @c -1  /*decrement the counter */
end
```

16.3.5 Variables and Null Values

Local variables are assigned the value NULL when they are declared, and may be assigned the null value by a `select` statement.

The special meaning of NULL requires that the comparison between null-value variables and other null values follow special rules.

This table shows the results of comparisons between null-value columns and null-value expressions using different comparison operators. An expression can be a variable, a literal, or a combination of variables, literals, and arithmetic operators.

Type of Comparison	Using the = Operator	Using the <, >, <=, !=, !<, !>, or <> Operator
Comparing `<column_value>` to `<column_value>`	FALSE	FALSE
Comparing `<column_value>` to `<expression>`	TRUE	FALSE
Comparing `<expression>` to `<column_value>`	TRUE	FALSE
Comparing `<expression>` to `<expression>`	TRUE	FALSE

For example, this test:

```
declare @v int, @i int
if @v = @i select "null = null, true"
if @v > @i select "null > null, true"
```

shows that only the first comparison returns true:

```
-----------------
null = null, true

(1 row affected)
```

This example returns all the rows from the `titles` table where the `advance` has the value NULL:

```
declare @m money
select title_id, advance
from titles
where advance = @m
```

```
title_id advance
-------- ----------------
MC3026              NULL
PC9999              NULL
```

```
(2 rows affected)
```

Using a `not in` clause when you compare a null value with a null value produces different results, depending on whether the variable or column contains the null values. This table shows the results compares between `not null`-value columns and `null`-value expressions using different comparison operators

Type of Comparison	Using the = Operator	Using the <, >, <=, !=, !<, !>, or <> Operator
Comparing `<column_value>` to `<column_value>`		
Comparing `<column_value>` to `<expression>`		
Comparing `<expression>` to `<column_value>`		
Comparing `<expression>` to `<expression>`		

For example, if you create this table:

```
create table #test(i int not null, a char(5) null)
insert into #test(i) values (1)
```

This example returns true because the `<@a>` variable contains a null value

```
declare @a char(5)
select @a = a from #test where i = 1
if @a not in ('NTTRD', 'NTOFF')
print 'true'
else
print 'false'
true
```

However, this example returns false because the column a contains a null value:

```
if (select a from #test where i = 1) not in ('NTTRD', 'NTOFF')
print 'true'
else
print 'false'
false
```

16.4 Global Variables

Global variables are system-supplied, predefined variables, and are a Transact-SQL extension. Global variables are distinguished from local variables by the two @ signs preceding their names—for example, `<@@error>`. The two @ signs are considered part of the identifier used to define the global variable.

Users cannot create global variables and cannot update the value of global variables directly in a `select` statement. If a user declares a local variable that has the same name as a global variable, that variable is treated as a local variable.

See, *Global Variables*, in the *Reference Manual: Building Blocks* for a complete list of the global variables.

16.4.1 Transactions and Global Variables

Some global variables provide information to use in transactions.

16.4.1.1 Check for Errors with @@error

The `<@@error>` global variable is commonly used to check the error status of the most recently executed batch in the current user session. `<@@error>` contains 0 if the last transaction succeeded; otherwise, `<@@error>` contains the last error number generated by the system.

A statement such as `if @@error != 0` followed by `return` causes an exit on error.

Every Transact-SQL statement, including `print` statements and `if` tests, resets `<@@error>`, so the status check must immediately follow the batch for which success is in question.

The `<@@sqlstatus>` global variable has no effect on `<@@error>` output.

Related Information

Check the Status From the Last Fetch [page 469]

16.4.1.2 Check IDENTITY Values with @@identity

`<@@identity>` contains the last value inserted into an IDENTITY column in the current user session.

`<@@identity>` is set each time an `insert`, `select into`, or `bcp` attempts to insert a row into a table. The value of `<@@identity>` is not affected by the failure of an `insert`, `select into`, or `bcp` statement or the rollback of the transaction that contained it. `<@@identity>` retains the last value inserted into an IDENTITY column, even if the statement that inserted it fails to commit.

If a statement inserts multiple rows, `<@@identity>` reflects the IDENTITY value for the last row inserted. If the affected table does not contain an IDENTITY column, `<@@identity>` is set to 0.

16.4.1.3 Check the Transaction Nesting Level with @@trancount

`<@@trancount>` contains the nesting level of transactions in the current user session.

Each `begin transaction` in a batch increments the transaction count. When you query `<@@trancount>` in chained transaction mode, its value is never 0 because the query automatically initiates a transaction.

16.4.1.4 Check the Transaction State with @@transtate

<@@transtate> contains the current state of a transaction after a statement executes in the current user session. Unlike <@@error>, <@@transtate> is not cleared for each batch.

<@@transtate> may contain the values in this table:

Value	Meaning
0	Transaction in progress: an explicit or implicit transaction is in effect; the previous statement executed successfully.
1	Transaction succeeded: the transaction completed and committed its changes.
2	Statement aborted: the previous statement was aborted; no effect on the transaction.
3	Transaction aborted: the transaction aborted and rolled back any changes.

<@@transtate> changes only due to execution errors. Syntax and compile errors do not affect the value of <@@transtate>.

Related Information

Transaction States [page 639]

16.4.1.5 Check the Nesting Level with @@nestlevel

<@@nestlevel> contains the nesting level of current execution with the user session, initially 0.

Each time a stored procedure or trigger calls another stored procedure or trigger, the nesting level is incremented. The nesting level is also incremented by one when a cached statement is created. If the maximum of 16 is exceeded, the transaction aborts.

16.4.1.6 Check the Status From the Last Fetch

<@@sqlstatus> contains status information resulting from the last `fetch` statement for the current user session.

<@@sqlstatus> may contain the following values:

Value	Meaning
0	The `fetch` statement completed successfully.

Value	Meaning
1	The `fetch` statement resulted in an error.
2	There is no more data in the result set. This warning occurs if the current cursor position is on the last row in the result set and the client submits a `fetch` command for that cursor.

`<@@sqlstatus>` has no effect on `<@@error>` output. For example, the following batch sets `<@@sqlstatus>`to 1 by causing the `fetch <@@error>` statement to result in an error. However, `<@@error>` reflects the number of the error message, not the `<@@sqlstatus>` output:

```
declare csr1 cursor
for select * from sysmessages
for read only

open csr1

begin
    declare @xyz varchar(255)
    fetch csr1 into @xyz
    select error = @@error
    select sqlstatus = @@sqlstatus
end
```

```
Msg 553, Level 16, State 1:
Line 3:
The number of parameters/variables in the FETCH INTO clause does not match the
number of columns in cursor 'csr1' result set.
```

At this point, the `<@@error>` global variable is set to 553, the number of the last generated error. `<@@sqlstatus>` is set to 1.

`<@@fetch_status>` returns the status of the most recent `fetch`:

Value	Meaning
0	`fetch` operation successful.
-1	`fetch` operation unsuccessful.
-2	Reserved for future use.

17 Transact-SQL Functions

SAP ASE functions are Transact-SQL routines that return information from the database or the system tables. SAP ASE provides a set of built-in functions. In addition, you can use the `create function` command to create Transact-SQL and SQLJ functions.

See the *Reference Manual:Building Blocks* for a complete list and description of the built-in functions. See the *Reference Manual:Commands* for information about the `create function` commands.

Typically, you can use functions in a `select` list, a `where` clause, and anywhere an expression is allowed.

The general syntax is:

```
select <function_name>[(<arguments>)]
```

For example, to find the identification number of the user who logs in as "emilya," enter:

```
select user_id("emilya")
```

The server returns:

```
------------------
                1209
```

17.1 Built-In Functions

There are various types of built-in functions including but not limited to system functions, string, text, and image functions, aggregate functions, and mathematical and date functions. The rules governing how you use these function types can differ.

See the *Reference Manual:Building Blocks* and, for XML functions, *XML Services*.

17.1.1 System Functions

System functions return information about the database. Many provide a shorthand way to query the system tables.

When the argument to a system function is optional, the current database, host computer, server user, or database user is assumed. With the exception of `user`, system functions are always used with parentheses, even if the argument is NULL.

For example, to find the name of the current user, omit the argument, but include the parentheses. For example:

```
select user_name()
```

```
-------------------
dbo
```

17.1.2 String Functions

String functions perform various operations on character strings, expressions, and sometimes on binary data.

When you use constants with a string function, enclose them in single or double quotation marks. You can concatenate binary and character expressions, and nest them.

Most string functions can be used only on `char`, `nchar`, `unichar`, `varchar`, `univarchar`, and `nvarchar` datatypes and on datatypes that implicitly convert to `char, unichar, varchar, or univarchar`. A few string functions can also be used on `binary` and `varbinary` data. `patindex` can be used on `text`, `unitext`, `char`, `nchar`, `unichar`, `varchar`, `nvarchar`, and `univarchar` columns.

Each function also accepts arguments that can be implicitly converted to the specified type. For example, functions that accept approximate numeric expressions also accept integer expressions. SAP ASE automatically converts the argument to the desired type.

Related Information

Concatenating Expressions [page 472]
Nest String Functions [page 474]

17.1.2.1 Concatenating Expressions

You can concatenate binary or character expressions; that is, you can combine two or more character or binary strings, character or binary data, or a combination of them with the + string concatenation operator. The maximum length of a concatenated string is 16384 bytes.

You can concatenate `binary` and `varbinary` columns and `char`, `unichar`, `nchar`, `varchar`, `univarchar`, and `nvarchar` columns. If you concatenate `unichar` and `univarchar` with `char`, `nchar`, `nvarchar`, and `varchar`, the result is `unichar` or `univarchar`. You cannot concatenate `text`, `unitext`, or `image` columns.

The concatenation syntax is:

```
select (<expression> + <expression> [+ <expression>]...)
```

For example, to combine two character strings:

```
select ("abc" + "def")
```

```
-------
abcdef
```

```
(1 row affected)
```

To combine California authors' first and last names under the column heading `Moniker` in last name-first name order, with a comma and space after the last name, enter:

```
select Moniker = (au_lname + ", " + au_fname)
from authors
where state = "CA"
```

```
Moniker
-----------------------------------------------
White, Johnson
Green, Marjorie
Carson, Cheryl
O'Leary, Michael
Straight, Dick
Bennet, Abraham
Dull, Ann
...
```

When a string function accepts two character expressions but only one expression is `unichar`, the other expression is "promoted" and internally converted to `unichar`. This follows existing rules for mixed-mode expressions. However, this conversion may cause truncation, since `unichar` data sometimes takes twice the space.

To concatenate noncharacter or nonbinary columns, use the `convert` function. For example:

```
select "The due date is " + convert(varchar(30),
    pubdate)
from titles
where title_id = "BU1032"
```

```
---------------------------------------
The due date is Jun 12 2006 12:00AM
```

SAP ASE evaluates the empty string (" " or ' ') as a single space. This statement:

```
select "abc" + "" + "def"
```

produces:

```
abc def
```

17.1.2.1.1 Concatenation Operators and LOB Locators

The + and || Transact-SQL operators accept LOB locators as expressions for a concatenation operation. The result of a concatenation operation involving one or more locators is a LOB locator with the same datatype as that referenced by the input locator.

For example, assume that `<@v>` and `<@w>` are text locator variables. These are valid concatenation operations:

- `select @v + @w`

- `select @v || "abdcef"`
- `select "xyz" + @w`

17.1.2.2 Nest String Functions

You can nest string functions. For example, use the `substring` function to display the last name and the first initial of each author, with a comma after the last name and a period after the first name.

Enter:

```
select (au_lname + "," + " " + substring(au_fname, 1, 1) + ".")
from authors
where city = "Oakland"
```

```
--------------------------------------------
Green, M.
Straight, D.
Stringer, D.
MacFeather, S.
Karsen, L.
```

To display the `pub_id` and the first two characters of each `title_id` for books priced more than $20, enter:

```
select substring(pub_id + title_id, 1, 6)
from titles
where price > $20
```

```
--------------
1389PC
0877PS
0877TC
```

17.1.2.3 Limits on String Functions

Results of string functions are limited to 16KB. This limit is independent of the server's page size. In Transact-SQL string functions and string variables, literals can be as large as 16K even on a 2KB page size.

If `set string_rtruncation` is on, a user receives an error if an insert or update truncates a character string. However, SAP ASE does not report an error if a displayed string is truncated. For example:

```
select replicate("a", 16383) + replicate("B", 4000)
```

This shows that the total length would be 20383, but the result string is restricted to 16K.

17.1.3 Text and Image Functions

Text functions perform operations on `text`, `image`, and `unitext` data. Use the set `textsize` option to limit the amount of `text`, `image`, and `unitext` data that is retrieved by a select statement.

> **Note**
>
> You can also use the `<@@textcolid>`, `<@@textdbid>`, `<@@textobjid>`, `<@@textptr>`, and `<@@textsize>` global variables to manipulate `text`, `image`, and `unitext` data.

For example, use the `textptr` function to locate the `text` column, `copy`, associated with `title_id` BU7832 in table `blurbs`. The text pointer, a 16-byte binary string, is put into a local variable, `<@val>`, and supplied as a parameter to the `readtext` command. `readtext` returns 5 bytes starting at the second byte, with an offset of 1. `readtext` provides a way to retrieve `text`, `unitext`, and `image` values if you want to retrieve only a portion of a column's data.

```
declare @val binary(16)
select @val = textptr(copy) from blurbs
where au_id = "486-29-1786"
readtext blurbs.copy @val 1 5
```

`textptr` returns a 16-byte `varbinary` string. SAP suggests that you put this string into a local variable, as in the preceding example, and use it by reference.

An alternative to using `textptr` in the preceding `declare` example is the `<@@textptr>` global variable:

```
readtext texttest.blurb @@textptr 1 5
```

> **Note**
>
> You can also use the string functions `patindex` and `datalength` on `text`, `image`, and `unitext` columns.

17.1.3.1 readtext on unitext Columns Usage

You can use the `readtext` command to retrieve `text`, `unitext`, and `image` values for only a selected portion of a column's data.

`readtext` requires the name of the table and column, the text pointer, a starting offset within the column, and the number of characters or bytes to retrieve. You cannot use `readtext` on `text`, `unitext`, or `image` columns in views.

For detailed information about using `readtext` with `unitext` columns, see the *Reference Manual: Commands*.

17.1.4 Aggregate Functions

Aggregate functions generate summary values that appear as new columns in the query results. They can be used in a `select` list or the `having` clause of a `select` statement or subquery.

When an aggregate function is applied to a `char` datatype value, it implicitly converts the value to `varchar`, stripping all trailing blanks. In a similar manner, a `unichar` datatype value is implicitly converted to `univarchar`.

Limitations

- Aggregate functions cannot be used in a `where` clause.
- Because each aggregate in a query requires its own worktable, an aggregate query cannot exceed the maximum number (46) of worktables allowed in a query.
- If you include an aggregate function in the select clause of a cursor, that cursor cannot be updated
- You cannot use aggregate functions on virtual tables such as `sysprocesses` and `syslocks`.

17.1.4.1 Aggregate Functions Used with the group by Clause

Aggregate functions are often used with the `group by` clause, which divides the tables into groups.

Aggregate functions produce a single value for each group. Without `group by`, an aggregate function in the `select` list produces a single value as a result, whether it is operating on all rows in a table or a subset of rows defined by the `where` clause.

17.1.4.2 Aggregate Functions and Null Values

Aggregate functions calculate the summary values of the non-null values in a particular column.

If the `ansinull` option is set `off` (the default), there is no warning when an aggregate function encounters a null. If `ansinull` is set on, a query returns the following SQLSTATE warning when an aggregate function encounters a null:

```
Warning- null value eliminated in set function
```

17.1.4.3 Vector and Scalar Aggregates

Aggregate functions can be applied to all the rows in a table, producing a single value, which is called a scalar aggregate.

They can also be applied to all the rows that have the same value in a specified column or expression (using the `group by` and, optionally, the `having` clause), in which case, they produce a value for each group, a vector aggregate. The results of the aggregate functions are shown as new columns.

You can nest a vector aggregate inside a scalar aggregate. For example:

```
select type, avg(price), avg(avg(price))
from titles
group by type
```

```
type
------------  ------------  ------------
UNDECIDED             NULL        15.23
business             13.73        15.23
mod_cook             11.49        15.23
popular_comp         21.48        15.23
psychology           13.50        15.23
trad_cook            15.96        15.23
(6 rows affected)
```

The `group by` clause applies to the vector aggregate—in this case, `avg(price)`. The scalar aggregate, `avg(avg(price))`, is the average of the average prices by type in the `titles` table.

In standard SQL, when a `select` list includes an aggregate, all the `select` list columns must either have aggregate functions applied to them or be in the `group by` list. Transact-SQL has no such restrictions.

Example 1 shows a select statement with the standard restrictions. Example 2 shows the same statement with another item (`title_id`) added to the `select` list. `order by` is also added to illustrate the difference in displays. These "extra" columns can also be referenced in a `having` clause.

Example 1

```
select type, avg(price), avg(advance)
from titles
group by type
```

```
type
------------  ------------  ------------
UNDECIDED             NULL        NULL
business             13.73     6,281.25
mod_cook             11.49     7,500.00
popular_comp         21.48     7,500.00
psychology           13.50     4,255.00
trad_cook            15.96     6,333.33

(6 rows affected)
```

Example 2

You can use either a column name or any other expression (except a column heading or alias) after `group by`.

Null values in the `group by` column are placed into a single group.

```
select type, title_id, avg(price), avg(advance)
from titles
group by type
order by type
```

```
type         title_id
-----------  --------    ---------- ---------
UNDECIDED       MC3026      NULL       NULL
business        BU1032      13.73   6,281.25
business        BU1111      13.73   6,281.25
business        BU2075      13.73   6,281.25
business        BU7832      13.73   6,281.25
mod_cook        MC2222      11.49   7,500.00
mod_cook        MC3021      11.49   7,500.00
popular_comp    PC1035      21.48   7,500.00
popular_comp    PC8888      21.48   7,500.00
popular_comp    PC9999      21.48   7,500.00
psychology      PS1372      13.50   4,255.00
psychology      PS2091      13.50   4,255.00
psychology      PS2106      13.50   4,255.00
psychology      PS3333      13.50   4,255.00
psychology      PS7777      13.50   4,255.00
trad_cook       TC3218      15.96   6,333.33
trad_cook       TC4203      15.96   6,333.33
trad_cook       TC7777      15.96   6,333.33
```

Example 3

The compute clause in a select statement uses row aggregates to produce summary values. The row aggregates make it possible to retrieve detail and summary rows with one command. Exampl3 3 illustrates this feature:

```
select type, title_id, price, advance
from titles
where type = "psychology"
order by type
compute sum(price), sum(advance) by type
```

```
type        title_id    price       advance
----------- -------     ----------  ---------
psychology  PS1372          21.59   7,000.00
psychology  PS2091          10.95   2,275.00
psychology  PS2106          7.00    6,000.00
psychology  PS3333          19.99   2,000.00
psychology  PS7777          7.99    4,000.00
                            sum     sum
                            ------- ----------
                            67.52   21,275.00
```

17.1.4.4 Aggregate Functions as Row Aggregates

Row aggregate functions generate summary values that appear as additional rows in query results.

To use the aggregate functions as row aggregates, use:

```
<Start of select statement>
```

```
compute <row_aggregate>(<column_name>)
        [, <row_aggregate>(<column_name>)]...
    [by <column_name> [, <column_name>]...]
```

where:

- `<column_name>` – is the name of a column, which must be enclosed in parentheses. Only exact numeric, approximate numeric, and money columns can be used with `sum` and `avg`.
  One compute clause can apply the same function to several columns. When using more than one function, use more than one compute clause.
- `by` – indicates that row aggregate values are to be calculated for subgroups. Whenever the value of the `by` item changes, row aggregate values are generated. If you use `by`, you must use `order by`.
  Listing more than one item after `by` breaks a group into subgroups, and applies a function at each level of grouping.

The row aggregates let you retrieve detail and summary rows with one command. The aggregate functions, however, ordinarily produce a single value for all the selected rows in the table or for each group, and these summary values are shown as new columns.

These examples illustrate the differences:

```
select type, sum(price), sum(advance)
from titles
where type like "%cook"
group by type
```

```
type
----------  ----------  ----------------
mod_cook         22.98         15,000.00
trad_cook        47.89         19,000.00

(2 rows affected)
```

```
select type, price, advance
from titles
where type like "%cook"
order by type
compute sum(price), sum(advance) by type
```

```
type        price       advance
----------  ----------  ----------------
mod_cook          2.99         15,000.00
mod_cook         19.99              0.00
            sum         sum
            ----------  ----------------
                 22.98         15,000.00
type         price      advance
----------  ----------  ----------------
trad_cook        11.95          4,000.00
trad_cook        14.99          8,000.00
```

```
trad_cook        20.95          7,000.00
            sum         sum
            ----------  ----------------
                 47.89         19,000.00
(7 rows affected)
type        price       advance
----------  ----------  ----------------
mod_cook          2.99         15,000.00
mod_cook         19.99              0.00
Compute Result:
---------------------- -----------------
                 22.98         15,000.00
type         price         advance
----------  ----------  ----------------
trad_cook        11.95          4,000.00
trad_cook        14.99          8,000.00
trad_cook        20.95          7,000.00
Compute Result:
---------------------- -----------------
                 47.89         19,000.00
(7 rows affected)
```

The columns in the compute clause must appear in the select list.

The order of columns in the `select` list overrides the order of the aggregates in the compute clause. For example:

```
create table t1 (a int, b int, c int null)
insert t1 values(1,5,8)
insert t1 values(2,6,9)
```

```
(1 row affected)
```

```
compute sum(c),  max(b), min(a)
select a, b, c from t1
```

```
a           b           c
----------- ----------- -----------
         1           5           8
         2           6           9
Compute Result:
----------- ----------- -----------
         1           6          17
```

If the `ansinull` option is set off (the default), there is no warning when a row aggregate encounters a null. If `ansinull` is set on, a query returns the following SQLSTATE warning when a row aggregate encounters a null:

```
Warning - null value eliminated in set function
```

You cannot use select `into` in the same statement as a compute clause because there is no way to store the compute clause output in the resulting table.

17.1.5 Statistical Aggregate Functions

Statistical aggregate functions enable you to perform statistical analysis on numeric data.

These functions are true aggregate functions in that they can compute values for a group of rows as determined by the query's `group by` clause. As with other basic aggregate functions, such as `max` or `min`,

computation ignores null values in the input. Also, regardless of the domain of the expression being analyzed, all variance and standard deviation computation uses the Institute of Electrical and Electronic Engineers (IEEE) double-precision, floating-point standard.

If the input to any variance or standard deviation function is the empty set, then each function returns a null value. If the input to any variance or standard deviation function is a single value, then each function returns 0.

Statistical aggregate functions are similar to the `avg` aggregate function in that:

- The syntax is:

```
<statistical_agg_function_name> ([all | distinct] <expression>)
```

- Only expressions with numerical datatypes are valid.
- Null values do not participate in the calculation.
- The result is NULL only if no data participates in the calculation.
- The `distinct` or `all` clauses can precede the expression (the default is `all`).
- You can use statistical aggregates as vector aggregates (with `group by`), scalar aggregates (without `group by`), or in the `compute` clause.

Unlike the `avg` aggregate function, however, the results are:

- Always of `float` datatype (that is, a double-precision floating-point), whereas for the `avg` aggregate, the datatype of the result is the same as that of the expression (with exceptions).
- 0.0 for a single data point.

Related Information

Aggregate Functions [page 476]

17.1.5.1 Formulas for Computing Standard Deviations

The `stddev_samp`, `stddev_pop` `var_samp`, and `var_pop` functions are similar but have different purposes.

See the `stddev_samp`, `stddev_pop` `var_samp`, and `var_pop` reference pages in the *Reference Manual: Blocks* to see the formulas that SAP ASE uses to define variances and standard deviations.

The functions are similar, but are used for different purposes:

- `var_samp` and `stddev_samp` – are used when you want evaluate a sample—that is, a subset—of a population as being representative of the entire population.
- `var_pop` and `stddev_pop` – are used when you have all of the data available for a population, or when `<n>` is so large that the difference between `<n>` and `<n-1>` is insignificant.

17.1.6 Mathematical Functions

Mathematical functions return values that are commonly needed for operations on mathematical data.

Each function also accepts arguments that can be implicitly converted to the specified type. For example, functions that accept approximate numeric types also accept integer types. SAP ASE converts the argument to the desired type.

SAP ASE provides error traps to handle domain or range errors. Use the `arithabort` and `arithignore` functions to specify how domain errors are handled.

This table displays examples using the `floor`, `ceiling`, and `round` mathematical functions.

Statement	Result
• `select floor(123)` • `select floor(123.45)` • `select floor(1.2345E2)` • `select floor(-123.45)` • `select floor(-1.2345E2)` • `select floor($123.45)`	`123 123.000000 123.000000 -124.000000 -124.000000 123.00`
• `select ceiling(123.45)` • `select ceiling(-123.45)` • `select ceiling(1.2345E2)` • `select ceiling(-1.2345E2)` • `select ceiling($123.45)`	`124.000000 -123.000000 124.000000 -123.000000 124.00`
• `select round(123.4545, 2)` • `select round(123.45, -2)` • `select round(1.2345E2, 2)` • `select round(1.2345E2, -2)`	`123.4500 100.00 123.450000 100.000000`

Related Information

Conversion Error Handling [page 495]

17.1.7 Date Functions

The date functions perform arithmetic operations and display information about `datetime`, `bigtime`, `bigdatetime`, `smalldatetime`, `date`, and `time` values. You can use them in the `select` list or the `where` clause of your query.

Use `datetime` datatypes for values later than January 1, 1753; use `date` for dates from January 1, 0001 to January 1, 9999. Enclose the date values in double or single quotes. SAP ASE recognizes many different date formats. See the *Reference Manual: Building Blocks* for more information about datatypes.

This is the default display format:

```
Apr 15 2010 10:23PM
```

Each date is divided into parts with abbreviations recognized by SAP ASE. This table lists each date part, its abbreviation (if there is one), and possible integer values for that part.

Date Part	Abbreviation	Values
`year`	`yy`	1753 – 9999 (`datetime`) 1900 – 2079 (`smalldatetime`) 0001 – 9999 (`date`)
`quarter`	`qq`	1– 4
`month`	`mm`	1– 12
`week`	`wk`	1– 54
`day`	`dd`	1– 31
`dayofyear`	`dy`	1– 366
`weekday`	`dw`	1– 7 (Sunday – Saturday)
`hour`	`hh`	0 – 23
`minute`	`mi`	0 – 59
`second`	`ss`	0 – 59
`millisecond`	`ms`	0 – 999
`microsecond`	`us`	0 – 999999

For example, use the `datediff` function to calculate the amount of time in date parts between the first and second of the two dates you specify. The result is a signed integer value equal to `<date2>` - `<date1 >`in date parts.

This query finds the number of days between `pubdate` and November 30, 2010:

```
select pubdate, newdate = datediff(day, pubdate,
    "Nov 30 2010")
from titles
```

```
pubdate                          newdate
------------------------ ---------------
Jun 12 2006 12:00AM                1632
Jun  9 2005 12:00AM                2000
Jun 30 2005 12:00AM                1979
Jun 22 2004 12:00AM                2352
Jun  9 2006 12:00AM                1635
Jun 15 2004 12:00AM                2356
...
```

Use the `dateadd` function to add an interval (specified as a integer) to a date you specify. For example, if the publication dates of all the books in the `titles` table slipped three days, you could get the new publication dates with this statement:

```
select dateadd(day, 3, pubdate)
from titles
```

```
---------------------
Jun 15 2006 12:00AM
Jun 12 2005 12:00AM
Jul  3 2005 12:00AM
Jun 25 2004 12:00AM
Jun 12 2006 12:00AM
Jun 21 2004 12:00AM
...
```

17.1.8 Datatype Conversion Functions

Datatype conversions change an expression from one datatype to another, and reformat the display format for `date` and `time` information.

SAP ASE performs certain datatype conversions, called implicit conversions. For example, if you compare a `char` expression and a `datetime` expression, or a `smallint` expression and an `int` expression, or `char` expressions of different lengths, SAP ASE automatically converts one datatype to another.

For other conversions, called explicit conversions, you must use a datatype conversion function to make the datatype conversion. For example, before concatenating numeric expressions, you must convert them to character expressions. If you attempt to explicitly convert a `date` to a `datetime` and the value is outside the datetime range such as "Jan 1, 1000" the conversion is not allowed and an error message appears.

Some datatype conversions are not allowed, either implicitly or explicitly. For example, you cannot convert `smallint` or `binary` data to `datetime`.

- E – explicit datatype conversion required.
- I – conversion can be explicit or implicit.
- U – datatype conversion is not allowed.

Table 6: Explicit, Implicit, and Unsupported Datatype Conversions

From	binary	varbinary	bit	[n]char	[n]varchar	datetime	smalldatetime	bigdatetime	bigtime	tinyint	smallint	unsigned smallint	int	unsigned int
binary	–	I	I	I	I	U	U	I	I	I	I	I	I	I
varbinary	I	–	I	I	I	U	U	I	I	I	I	I	I	I
bit	I	I	–	I	I	U	U	U	U	I	I	I	I	I
[n]char	I	I	E	–	I	I	I	I	I	E	E	E	E	E
[n]varchar	I	I	E	I	–	I	I	I	I	E	E	E	E	E
datetime	I	I	U	I	I	–	I	I	I	U	U	U	U	U
smalldatetime	I	I	U	I	I	I	–	I	I	U	U	U	U	U
bigdatetime	I	I	U	I	I	I	I	-	I	U	U	U	U	U
bigtime	I	I	U	I	I	I	I	I	-	U	U	U	U	U
tinyint	I	I	I	E	E	U	U	U	U	–	I	I	I	I
smallint	I	I	I	E	E	U	U	U	U	I	–	I	I	I
unsigned smallint	I	I	I	E	E	U	U	U	U	I	I	–	I	I
int	I	I	I	E	E	U	U	U	U	I	I	I	–	I
unsigned int	I	I	I	E	E	U	U	U	U	I	I	I	I	–
bigint	I	I	I	E	E	U	U	U	U	I	I	I	I	I
unsigned bigint	I	I	I	E	E	U	U		U	I	I	I	I	I
decimal	I	I	I	E	E	U	U	U	U	I	I	I	I	I
numeric	I	I	I	E	E	U	U	U	U	I	I	I	I	I
float	I	I	I	E	E	U	U	U	U	I	I	I	I	I
real	I	I	I	E	E	U	U	U	U	I	I	I	I	I
money	I	I	I	I	I	U	U	U	U	I	I	I	I	I
smallmoney	I	I	I	I	I	U	U	U	U	I	I	I	I	I

From	binary	varbinary	bit	[n]char	[n]varchar	datetime	smalldatetime	bigdatetime	bigtime	tinyint	smallint	unsigned smallint	int	unsigned int
text	U	U	U	E	E	U	U	U	U	U	U	U	U	U
unitext	E	E	E	E	E	U	U	U	U	U	U	U	U	U
image	E	E	U	U	U	U	U	U	U	U	U	U	U	U
unichar	I	I	E	I	I	I	I	I	I	E	E	E	E	E
univarchar	I	I	E	I	I	I	I	I	I	E	E	E	E	E
date	I	I	U	I	I	I	U	I	U	U	U	U	U	U
time	I	I	U	I	I	I	U	I	I	U	U	U	U	U

Table 7: Explicit, Implicit, and Unsupported Datatype Conversions

From	bigint	unsigned bigint	decimal	numeric	float	real	money	smallmoney	text	unitext	image	unichar	univarchar	date	time
binary	I	I	I	I	I	I	I	I	U	I	I	I	I	I	I
varbinary	I	I	I	I	I	I	I	I	U	I	I	I	I	I	I
bit	I	I	I	I	I	I	I	I	U	U	U	E	E	U	U
[n]char	E	E	E	E	E	E	E	E	I	I	I	I	I	I	I
[n]varchar	E	E	E	E	E	E	E	E	I	I	I	I	I	I	I
datetime	U	U	U	U	U	U	U	U	U	U	U	I	I	I	I
smalldatetime	U	U	U	U	U	U	U	U	U	U	U	I	I	I	I
bigdatetime	U	U	U	U	U	U	U	U	U	U	U	I	I	I	I
bigtime	U	U	U	U	U	U	U	U	U	U	U	I	I	U	I
tinyint	I	I	I	I	I	I	I	I	U	U	U	E	E	U	U
smallint	I	I	I	I	I	I	I	I	U	U	U	U	E	U	U

From	bigint	unsigned bigint	decimal	numeric	float	real	money	smallmoney	text	unitext	image	unichar	univarchar	date	time
unsigned smallint	I	I	I	I	I	I	I	I	U	U	U	E	E	U	U
int	I	I	I	I	I	I	I	I	U	U	U	E	E	U	U
unsigned int	I	I	I	I	I	I	I	I	U	U	U	E	E	U	U
bigint	–	I	I	I	I	I	I	I	U	U	U	E	E	U	U
unsigned bigint	I	–	I	I	I	I	I	I	U	U	U	E	E	U	U
decimal	I	I	–	I	I	I	I	I	U	U	U	E	E	U	U
numeric	I	I	I	–	I	I	I	I	U	U	U	E	E	U	U
float	I	I	I	I	–	I	I	I	U	U	U	E	E	U	U
real	I	I	I	I	I	–	I	I	U	U	U	E	E	U	U
money	I	I	I	I	I	I	–	I	U	U	U	E	E	U	U
smallmoney	I	I	I	I	I	I	I	–	U	U	U	E	E	U	U
text	U	U	U	U	U	U	U	U	–	I	U	E	E	U	U
unitext	U	U	U	U	U	U	U	U	I	–	I	U	U	U	U
image	U	U	U	U	U	U	U	U	U	I	–	E	E	U	U
unichar	E	E	E	E	E	E	E	E	I	I	I	–	I	I	I
univarchar	E	E	E	E	E	E	E	E	I	I	I	I	–	I	I
date	U	U	U	U	U	U	U	U	U	U	U	I	I	–	I
time	U	U	U	U	U	U	U	U	U	U	U	I	I	I	–

Related Information

convert Function Usage for Explicit Conversions [page 488]

17.1.8.1 convert Function Usage for Explicit Conversions

The general conversion function, `convert`, converts between a variety of datatypes and, for date and time information, specifies a new display format.

Its syntax is:

```
convert(<datatype> [(<length>) | (<precision>[, <scale>])] [null | not null],
    <expression> [, <style> ] )
```

This example uses `convert` in the `select` list:

```
select title, convert(char(5), total_sales)
from titles
where type = "trad_cook"
```

```
title
------------------------------------     -----
Onions, Leeks, and Garlic: Cooking
    Secrets of the Mediterranean           375
Fifty Years in Buckingham Palace
    Kitchens                             15096
Sushi, Anyone?                            4095

(3 rows affected)
```

Certain datatypes expect either a length or a precision and scale. If you do not specify a length, SAP ASE uses the default length of 30 for character and binary data. If you do not specify a precision or scale, SAP ASE uses the defaults of 18 and 0, respectively.

17.1.8.2 Datatype Conversion Guidelines and Constraints

There are general guidelines and limitations to consider for datatype conversions.

17.1.8.2.1 Convert Character Data to a Noncharacter Type

You can convert character data to a noncharacter type—such as a money, date/time, exact numeric, or approximate numeric type—if it consists entirely of characters that are valid for the new type.

Leading blanks are ignored. However, if a `char` expression that consists of a blank or blanks is converted to a `datetime` expression, SAP ASE converts the blanks into the default `datetime` value of "Jan 1, 1900."

Syntax errors occur when the data includes unacceptable characters. These are examples of characters that cause syntax errors:

- Commas or decimal points in integer data
- Commas in monetary data
- Letters in exact or approximate numeric data or bit stream data
- Misspelled month names in `date` and `time` data

Implicit conversions between `unichar/univarchar` and `datetime/smalldatetime` are supported.

17.1.8.2.2 Convert from One Character Type to Another

When converting from a multibyte character set to a single-byte character set, characters with no single-byte equivalent are converted to question marks.

You can explicitly convert `text` and `unitext` columns to `char, nchar, varchar, unichar, univarchar,` or `nvarchar`. You are limited to the maximum length of the `character` datatypes, which is determined by the maximum column size for your server's logical page size. If you do not specify the length, the converted value has a default length of 30 bytes.

17.1.8.2.3 Convert Numbers to a Character Type

You can convert exact and approximate numeric data to a character type.

If the new type is too short to accommodate the entire string, an insufficient space error is generated. For example, the following conversion tries to store a 5-character string in a 1-character type:

```
select convert(char(1), 12.34)
```

```
Insufficient result space for explicit conversion
of NUMERIC value '12.34' to a CHAR field.
```

When converting `float` data to a character type, the new type should be at least 25 characters long.

> **Note**
>
> The `str` function may be preferable to `convert` or `cast` when making conversions, because it provides more control over conversions and avoids errors.

17.1.8.2.4 Convert to or from unitext

You can implicitly convert to `unitext` from other character and binary datatypes. You can explicitly convert from `unitext` to other datatypes, and vice versa.

However, the conversion result is limited to the maximum length of the destination datatype. When a `unitext` value cannot fit the destination buffer on a Unicode character boundary, data is truncated. If you have set `enable surrogate processing`, the `unitext` value is never truncated in the middle of a surrogate pair of values, which means that fewer bytes may be returned after the datatype conversion. For example, if a `unitext` column `ut` in table `tb` stores the string "U+0041U+0042U+00c2" (U+0041 represents the Unicode character "A"), this query returns the value "AB" if the server's character set is UTF-8, because U+00C2 is converted to 2-byte UTF-8 0xc382:

```
select convert(char(3), ut) from tb
```

Currently, the `alter table modify` command does not support `text`, `image`, or `unitext` columns as the modified column. To migrate from a `text` to a `unitext` column, you must first use `bcp out` to copy the existing data out, create a table with `unitext` columns, and then use `bcp in` to place data into the new table. This migration path works only when you invoke `bcp` with the `-Jutf8` option.

17.1.8.2.5 Rounding During Conversion To and From Money Types

The `money` and `smallmoney` types store four digits to the right of the decimal point, but rounds up to the nearest hundredth (.01) for display purposes. When data is converted to a money type, it is rounded up to four places.

Data converted from a money type follows the same rounding behavior if possible. If the new type is an exact numeric with less than three decimal places, the data is rounded to the scale of the new type. For example, when $4.50 is converted to an integer, it yields 5:

```
select convert(int, $4.50)
```

```
 -----------
           5
```

Data converted to `money` or `smallmoney` is assumed to be in full currency units, such as dollars, rather than in fractional units, such as cents. For example, the integer value of 5 is converted to the money equivalent of 5 dollars, not 5 cents, in the us_english language.

17.1.8.2.6 Convert Date and Time Information

Data that is recognizable as a date can be converted to `datetime`, `smalldatetime`, `date`, or `time`. Incorrect month names lead to syntax errors. Dates that fall outside the acceptable range for the datatype lead to arithmetic overflow errors.

When `datetime` values are converted to `smalldatetime`, they are rounded to the nearest minute.

Related Information

Change the Date Format [page 493]

17.1.8.2.7 Convert Between Numeric Types

You can convert data from one numeric type to another. Errors can occur if the new type is an exact numeric with precision or scale that is insufficient to hold the data.

For example, if you provide a float or numeric value as an argument to a built-in function that expects an integer, the value of the float or numeric is truncated. However, SAP ASE does not implicitly convert numerics that have a fractional part but return a scale error message. For example, SAP ASE returns error 241 for numerics that have a fractional part and error 257 if other datatypes are passed.

Use the `arithabort` and `arithignore` options to determine how SAP ASE handles errors resulting from numeric conversions.

17.1.8.2.8 Convert Between Binary and Integer Types

`binary` and `varbinary` types store hexadecimal-like data consisting of a "0x" prefix followed by a string of digits and letters.

These strings are interpreted differently by different platforms. For example, the string "0x0000100" represents 65536 on machines that consider byte 0 most significant (little-endian) and 256 on machines that consider byte 0 least significant (big-endian).

Binary types can be converted to integer types either explicitly, using the `convert` function, or implicitly. If the data is too short for the new type, it is stripped of its "0x" prefix and zero-padded. If it is too long, it is truncated.

Both `convert` and the implicit datatype conversions evaluate binary data differently on different platforms. Because of this, results may vary from one platform to another. Use the `hextoint` function for platform-independent conversion of hexadecimal strings to integers, and the `inttohex` function for platform-independent conversion of integers to hexadecimal values. Use the `hextobigint` function for platform-independent conversion of hexadecimal strings to 64-bit integers, and the `biginttohex` function for platform-independent conversion of 64-bit integers to hexadecimal values.

17.1.8.2.9 Convert Between Binary and Numeric or Decimal Types

When you convert a `binary` or `varbinary` type to `numeric` or `decimal`, you must specify the "00" or "01" values after the "0x" digit; otherwise, the conversion fails.

In `binary` and `varbinary` data strings, the first two digits after "0x" represent the `binary` type: "00" represents a positive number and "01" represents a negative number.

For example, to convert the following `binary` data to `numeric` use:

```
select convert(numeric
(38, 18),0x00000000000000006b14bd1e6eea0000000000000000000000000000000000)
```

```
----------
```

```
123.456000
```

To convert the same `numeric` data back to `binary` use:

```
select convert(binary,convert(numeric(38, 18), 123.456))
```

```
----------------------------------------------------------------
0x000000000000000006b14bd1e6eea00000000000000000000000000000000
```

17.1.8.2.10 Convert Image Columns to Binary Types

You can use the `convert` function to convert an `image` column to `binary` or `varbinary`.

You are limited to the maximum length of the `binary` datatypes, which is determined by the maximum column size for your server's logical page size. If you do not specify the length, the converted value has a default length of 30 characters.

17.1.8.2.11 Convert Other Types to bit

Exact and approximate numeric types can be implicitly converted to the `bit` type. Character types require an explicit `convert` function.

The expression being converted must consist only of digits, a decimal point, a currency symbol, and a plus or minus sign. The presence of other characters generates syntax errors.

The `bit` equivalent of 0 is 0. The `bit` equivalent of any other number is 1.

17.1.8.2.12 Convert Hexadecimal Data

For conversion results that are reliable across platforms, use the `hextoint` and `inttohex` functions.

Similar functions, `hextobigint` and `biginttohex`, are available to convert to and from 64-bit integers.

`hextoint` accepts literals or variables consisting of digits and the uppercase and lowercase letters A – F, with or without a "0x" prefix. These are all valid uses of `hextoint`:

```
select hextoint("0x00000100FFFFF")
select hextoint("0x00000100")
select hextoint("100")
```

`hextoint` strips data of the "0x" prefix. If the data exceeds 8 digits, `hextoint` truncates it. If the data is fewer than 8 digits, `hextoint` right-justifies and pads it with zeros. Then `hextoint` returns the platform-independent integer equivalent. The above expressions all return the same value, 256, regardless of the platform that executes the `hextoint` function.

The `inttohex` function accepts integer data and returns an 8-character hexadecimal string without a "0x" prefix. `inttohex` always returns the same results, regardless of platform.

17.1.8.2.13 Convert bigtime and bigdatetime Data

Implicit and explicit conversions are allowed where a decreased precision results in the loss of data.

Implicit conversion between types without matching primary fields may cause either data truncation, the insertion of a default value, or an error message to be raised. For example, when a `bigdatetime` value is converted to a `date` value, the time portion is truncated leaving only the date portion. If a `bigtime` value is converted to a `bigdatetime` value, a default date portion of Jan 1, 0001 is added to the new `bigdatetime` value. If a `date` value is converted to a `bigdatetime` value, a default time portion of 00:00:00.000000 is added to the `bigdatetime` value.

17.1.8.2.14 Convert NULL Value

You can use the `convert` function to change NULL to NOT NULL and NOT NULL to NULL.

17.1.8.3 Change the Date Format

The `<style>` parameter of `convert` provides a variety of date display formats for converting `datetime` or `smalldatetime` data to `char` or `varchar`.

The number argument you supply as the `<style>` parameter determines how the data appears. The year can use either two or four digits. Add 100 to a `<style>` value to get a 4-digit year, including the century (`<yyyy>`).

The table below shows the possible values for `<style, >`and the variety of date formats you can use. When you use `<style>` with `smalldatetime`, the styles that include seconds or milliseconds show zeros in those positions.

In the table, “mon” indicates a month spelled out, “mm” the month number or minutes. “HH ”indicates a 24-hour clock value, “hh” a 12-hour clock value. The last row, 23, includes a literal “T” to separate the date and time portions of the format.

Without Century (yy)	With Century (yyyy)	Standard	Output
-	0 or 100	Default	`<mon dd yyyy hh:mm>` AM (or PM)
1	101	USA	`<mm/dd/yy >`
2	2	SQL standard	`<yy.mm.dd >`
3	103	English/French	`<dd/mm/yy >`
4	104	German	`<dd.mm.yy >`
5	105		`<dd-mm-yy >`
6	106		`<dd mon yy >`

Without Century (yy)	With Century (yyyy)	Standard	Output
7	107		<mon dd, yy >
8	108		<HH:mm:ss >
-	9 or 109	Default + milliseconds	<mon dd yyyy hh:mm:sss> AM (or PM)
10	110	USA	<mm-dd-yy >
11	111	Japan	<yy/mm/dd >
12	112	ISO	<yymmdd >
13	113		<yy/dd/mm>
14	114		<mm/yy/dd>
15	115		<dd/yy/mm>
-	16 or 116		<mon dd yyyy HH:mm:ss>
17	117		<hh:mmAM>
18	118		<HH:mm>
19			<hh:mm:ss:zzzAM>
20			<HH:mm:ss:zzz>
21			<yy/mm/dd> <HH:mm:ss>
22			<yy/mm/dd hh:mm >AM (or PM)
	23		<yyyy-mm-ddTHH:mm:ss>

The default values, style 0 or 100, and 9 or 109, always return the century (<yyyy>).

This example converts the current date to style 3, <dd/mm/yy>:

```
select convert(char(12), getdate(), 3)
```

When converting date data to a character type, use style numbers 1 – 7 (101 – 107) or 10 – 12 (110 – 112) to specify the display format. The default value is 100 (<mon dd yyyy hh:mi>AM (or PM)). If date data is converted to a style that contains a time portion, that time portion reflects the default value of zero.When converting time data to a character type, use style number 8 or 9 (108 or 109) to specify the display format. The default is 100 (<mon dd yyyy hh:mi>AM (or PM)). If time data is converted to a style that contains a date portion, the default date of Jan 1, 1900 appears.

> **Note**
>
> `convert` with NULL in the `<style>` argument returns the same result as `convert` with no `<style>` argument. For example:
>
> ```
> select convert(datetime, "01/01/01")
> ------------
> ```
>
> ```
> Jan 1 2001 12:00AM
> ```
>
> ```
> select convert(datetime, "01/01/01", NULL)
> ------------
> ```
>
> ```
> Jan 1 2001 12:00AM
> ```

17.1.8.4 Conversion Error Handling

Conversion results can produce errors for divide-by-zero, arithmetic overflow, scale, and domain.

Arithmetic Overflow and Divide-by-Zero Errors

Divide-by-zero errors occur when SAP ASE tries to divide a numeric value by zero. Arithmetic overflow errors occur when the new type has too few decimal places to accommodate the results.

This happens during:

- Explicit or implicit conversions to exact types with a lower precision or scale
- Explicit or implicit conversions of data that falls outside the acceptable range for a money or date/time type
- Conversions of hexadecimal strings requiring more than 4 bytes of storage using `hextoint`

Both arithmetic overflow and divide-by-zero errors are considered serious, whether they occur during an implicit or explicit conversion. Use the `arithabort arith_overflow` option to specify how SAP ASE handles these errors. The default setting, `arithabort arith_overflow on`, rolls back the entire transaction in which the error occurs. If the error occurs in a batch that does not contain a transaction, `arithabort arith_overflow on` does not roll back earlier commands in the batch, and SAP ASE does not execute statements that follow the error-generating statement in the batch. If you set `arithabort arith_overflow off`, SAP ASE aborts the statement that causes the error, but continues to process other statements in the transaction or batch.You can use the `<@@error>` global variable to check statement results.

Use the `arithignore arith_overflow` option to determine whether SAP ASE displays a message after these errors. The default setting, `off`, displays a warning message when a divide-by-zero error or a loss of precision occurs. Setting `arithignore arith_overflow on` suppresses warning messages after these errors. You can omit optional `arith_overflow` keyword without any effect.

Scale Errors

When an explicit conversion results in a loss of scale, the results are truncated without warning.

For example, when you explicitly convert a `float`, `numeric`, or `decimal` type to an `integer`, SAP ASE assumes you want the result to be an integer and truncates all numbers to the right of the decimal point.

During implicit conversions to `numeric` or `decimal` types, loss of scale generates a scale error. Use the `arithabort numeric_truncation` option to determine how serious such an error is considered. The default setting, `arithabort numeric_truncation on`, aborts the statement that causes the error, but continues to process other statements in the transaction or batch. If you set `arithabort numeric_truncation off`, SAP ASE truncates the query results and continues processing.

> **Note**
>
> For entry-level ANSI SQL compliance, set:
>
> - `arithabort arith_overflow off`
> - `arithabort numeric_truncation on`
> - `arithignore off`

Domain Errors

The `convert` function generates a domain error when the function's argument falls outside the range over which the function is defined. This rarely happens.

17.1.9 Security Functions

The security functions return information about security services and user-defined roles.

See the *Security Administration Guide* for information about managing user permissions.

17.1.10 XML Functions

The XML functions let you manage XML in the SAP ASE database.

The XML functions are described in the *XML Services* book.

17.2 User-Created Functions

Use the `create function` command to create and save your own scalar Transact-SQL functions.

You can include:

- `declare` statements to define data variables and cursors that are local to the function
- Assigned values to objects local to the function (for example, assigning values to scalar and variables local to a table with `select` or `set` commands)
- Cursor operations that reference local cursors that are declared, opened, closed, and deallocated in the function
- Control-of-flow statements
- `set` options (valid only in the scope of the function)

You cannot include :

- `select` or `fetch` statements that return data to the client
- `insert`, `update`, or `delete` statements
- Utility commands, such as `dbcc`, `dump` and `load`
- `print` statements
- Statements that references `rand`, `rand2`, `getdate`, or `newid`

You can include `select` or `fetch` statements that assign values only to local variables.

See the *Reference Manual: Commands*.

> **Note**
>
> You can also create Transact-SQL functions that return a value specified by a Java method. Use the `create function` (SQLJ) command to add a Transact-SQL wrapper to a Java method. See *Java in the SAP ASE Database* and the *Reference Manual: Commands*.

18 Stored Procedures

A stored procedure is a named collection of SQL statements or control-of-flow language. You can create stored procedures for commonly used functions, and to improve performance. SAP ASE also provides system procedures for performing administrative tasks that update the system tables.

Stored procedures can:

- Take parameters
- Call other procedures
- Return a status value to a calling procedure or batch to indicate success or failure and the reason for failure
- Return values of parameters to a calling procedure or batch
- Be executed on remote servers

The ability to write stored procedures greatly enhances the power, efficiency, and flexibility of SQL. Compiled procedures dramatically improve the performance of SQL statements and batches. In addition, stored procedures on other servers can be executed if both your server and the remote server are set up to allow remote logins. You can write triggers on your local server that execute procedures on a remote server whenever certain events, such as deletions, updates, or inserts, take place locally.

Stored procedures differ from ordinary SQL statements and from batches of SQL statements in that they are precompiled. The first time you run a procedure, the SAP ASE query processor analyzes it and prepares an execution plan that is, after successful execution, stored in a system table. Subsequently, the procedure is executed according to the stored plan. Since most of the query processing work has already been performed, stored procedures execute almost instantly.

SAP ASE supplies a variety of stored procedures as convenient tools for the user. The procedures stored in the `sybsystemprocs` database whose names begin with "sp_" are known as system procedures, because they insert, update, delete, and report on data in the system tables.

The *Reference Manual: Procedures* includes a complete list of all SAP-provided system procedures.

You can also create and use extended stored procedures to call procedural language functions from SAP ASE.

Related Information

Extended Stored Procedures Usage [page 535]

18.1 Examples

Examples are provided to illustrate how to create and execute a stored procedure.

To create a simple stored procedure, without special features such as parameters, the syntax is:

```
create procedure <procedure_name>
    as <SQL_statements>
```

Stored procedures are database objects, and their names must follow the rules for identifiers.

Any number and kind of SQL statements can be included except for `create` statements.

A procedure can be as simple as a single statement that lists the names of all the users in a database:

```
create procedure namelist
as select name from sysusers
```

To execute a stored procedure, use the keyword `execute` and the name of the stored procedure, or just use the procedure's name, as long as it is submitted to SAP ASE by itself or is the first statement in a batch. You can execute `namelist` in any of these ways:

```
namelist
execute namelist
exec namelist
```

To execute a stored procedure on a remote SAP ASE, include the server name. The syntax for a remote procedure call is:

```
execute <server_name>.[<database_name>].[<owner>].<procedure_name>
```

The database name is optional only if the stored procedure is located in your default database. The owner name is optional only if the database owner ("dbo") owns the procedure or if you own it. You must have permission to execute the procedure.

A procedure can include more than one statement.

```
create procedure showall as
select count(*) from sysusers
select count(*) from sysobjects
select count(*) from syscolumns
```

When the procedure is executed, the results of each command appear in the order in which the statement appears in the procedure.

```
showall
------------
```

```
           5

(1 row affected)

------------
          88

(1 row affected)
```

```
------------
         349

(1 row affected, return status = 0)
```

When a `create procedure` command is successfully executed, the procedure's name is stored in `sysobjects`, and its source text is stored in `syscomments`.

After you create a stored procedure, the source text describing the procedure is stored in the `text` column of the `syscomments` system table. Do not remove this information from `syscomments`; doing so can cause problems for future upgrades of SAP ASE. Use `sp_hidetext` to encrypt the text in `syscomments`. See the *Reference Manual: Procedures*.

Use `sp_helptext` to display the source text of a procedure:

```
sp_helptext showall
```

```
# Lines of Text
---------------
              1

(1 row affected)
text
----------------------------------------
create procedure showall as
select count(*) from sysusers
select count(*) from sysobjects
select count(*) from syscolumns

(1 row affected, return status = 0)
```

When you create procedures with deferred name resolution (which lets you create a stored procedure that references objects that do not yet exist), the text in `syscomments` is stored without performing the `select *` expansion. After the procedure's first successful execution, SAP ASE performs the `select *` expansion and the text for the procedure is updated with the expanded text. Since the `select *` expansion is executed before updating the text, the final text contains the expanded `select *`, as this example shows:

```
create table t (a int, b int)
---------set deferred_name_resolution on
---------------create proc p as select * from t
------------------sp_helptext p
------------- # Lines of Text
---------------
             1(1 row affected)
text
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
create proc p as select * from t
(1 row affected)
(return status = 0)exec p
------------
a           b
----------- -----------(0 rows affected)
(return status = 0)sp_helptext p
------------- # Lines of Text
---------------
              1(1 row affected)
text
```

```
------------------------------------------------------------------------------
------------------------------------------------------------------------------
------------------------ /* SAP ASE has expanded all '*' elements in the
following statement */
create proc p as select t.a, t.b from t
(1 row affected)
(return status = 0)
```

Related Information

Restrictions Associated with Stored Procedures [page 528]
Execute Procedures Remotely [page 515]
Compiled Objects [page 22]

18.2 Permissions

Stored procedures can serve as security mechanisms, since a user can be granted permission to execute a stored procedure, even if she or he does not have permissions on the tables or views referenced in it, or permission to execute specific commands.

You can protect the source text of a stored procedure against unauthorized access by restricting `select` permission on the `text` column of the `syscomments` table to the creator of the procedure and the system administrator. This restriction is required to run SAP ASE in the evaluated configuration. To enact this restriction, a system security officer must reset the `allow select on syscomments.text column` parameter using `sp_configure`. See, *Setting Configuration Parameters*, the *System Administration Guide: Volume 1*.

Another way to protect access to the source text of a stored procedure is to use `sp_hidetext` to hide the source text . See the *Reference Manual: Procedures*.

See, *Managing User Permissions*, in the *Security Administration Guide*.

18.3 Performance

As a database changes, you can optimize the original query plans used to access its tables by recompiling them. This saves you from having to find, drop, and re-create every stored procedure and trigger.

This example marks every stored procedure and trigger that accesses the table `titles` to be recompiled the next time it is executed.

```
sp_recompile titles
```

See the *Reference Manual: Procedures*.

18.4 Create and Execute Stored Procedures

You can create a procedure in the current database.

In versions of SAP ASE earlier than 15.5, all referenced objects where required to exist when a procedure was created. The deferred name resolution feature allows objects, except for user-created datatype objects, to be resolved when the stored procedure is initially executed.

Deferred name resolution uses the `deferred name resolution` configuration parameter, which operates at the server level, or `set deferred_name_resolution` parameter, which operates at the connection level.

The default behavior is to resolve the objects before execution. You must explicitly indicate deferred name resolution using the configuration option `deferred name resolution`, or the `set` parameter.

See the *System Administration Guide: Volume 1*, and the *Reference Manual: Commands*.

Permission to issue `create procedure` defaults to the database owner, who can transfer it to other users.

18.4.1 Deferred Name Resolution Usage

When deferred name resolution is active objects inside procedures are resolved at execution time, instead of at creation time.

You can use this option to create procedures that reference objects that did not exist when the procedure was created. For example, using `deferred_name_resolution` allows creating a procedure that references a table that does not yet exist. This example shows an attempt to create a procedure without `deferred_name_resolution`:

```
select * from non_existing_table
-----------
```

```
error message
Msg 208, Level 16, State 1:
Line 1:
non_existing_table not found. Specify owner.objectname or use sp_help to check
whether the object exists (sp_help may produce lots of output).
```

However, using this option allows you to create the procedure without raising an error because of the missing objects.

```
set deferred_resolution_on
-----------------
create proc p as select * from non_existing_table
-----------------
```

> **Note**
>
> `deferred_name_resolution` does not resolve user-defined datatypes at execution. They are resolved at creation time, so if the resolution fails, the procedure cannot be created.

Resolving objects at creation time means that object resolution errors are also raised at execution, not creation.

18.4.2 Parameters

A parameter is an argument to a stored procedure. You can optionally declare one or more parameters in a `create procedure` statement. The value of each parameter named in a `create procedure` statement must be supplied by the user when the procedure is executed.

Parameter names must be preceded by an @ sign and must conform to the rules for identifiers. Parameter names are local to the procedure that creates them; the same parameter names can be used in other procedures. Enclose any parameter value that includes punctuation (such as an object name qualified by a database name or owner name) in single or double quotes. Parameter names, including the @ sign, can be a maximum of 255 bytes long.

Parameters must be given a system datatype (except `text`, `unitext`, or `image`) or a user-defined datatype, and (if required for the datatype) a length or precision and scale in parentheses.

Here is a stored procedure for the `pubs2` database. Given an author's last and first names, the procedure displays the names of any books written by that person and the name of each book's publisher.

```
create proc au_info @lastname varchar(40),
  @firstname varchar(20) as
select au_lname, au_fname, title, pub_name
from authors, titles, publishers, titleauthor
where au_fname = @firstname
and au_lname = @lastname
and authors.au_id = titleauthor.au_id
and titles.title_id = titleauthor.title_id
and titles.pub_id = publishers.pub_id
```

Now, execute `au_info`:

```
au_info Ringer, Anne
```

```
au_lname au_fname title                 pub_name
-------- -------- --------------------- ----------
Ringer   Anne     The Gourmet Microwave Binnet & Hardley
Ringer   Anne     Is Anger the Enemy?   New Age Books
(2 rows affected, return status = 0)
```

The following stored procedure queries the system tables. Given a table name as the parameter, the procedure displays the table name, index name, and index ID:

```
create proc showind @table varchar(30) as
select table_name = sysobjects.name,
index_name = sysindexes.name, index_id = indid
from sysindexes, sysobjects
where sysobjects.name = @table
and sysobjects.id = sysindexes.id
```

The column headings, for example, `table_name`, were added to improve the readability of the results. Here are acceptable syntax forms for executing this stored procedure:

```
execute showind titles
exec showind titles
execute showind @table = titles
execute GATEWAY.pubs2.dbo.showind titles
showind titles
```

The last syntax form, without exec or execute, is acceptable as long as the statement is the only one or the first one in a batch.

Here are the results of executing showind in the pubs2 database when <titles> is given as the parameter:

```
table_name  index_name  index_id
----------  ----------  ----------
titles      titleidind      0
titles      titleind        2

(2 rows affected, return status = 0)
```

If you supply the parameters in the form "@<parameter> = <value>" you can supply them in any order. Otherwise, you must supply parameters in the order of their create procedure statement. If you supply one value in the form "@<parameter> = <value>", then supply all subsequent parameters this way.

This procedure displays the datatype of the qty column from the salesdetail table:

```
create procedure showtype @tabname varchar(18), @colname varchar(18) as
  select syscolumns.name, syscolumns.length,
  systypes.name
  from syscolumns, systypes, sysobjects
  where sysobjects.id = syscolumns.id
  and @tabname = sysobjects.name
  and @colname = syscolumns.name
  and syscolumns.type = systypes.type
```

When you execute this procedure, you can give the <@tabname> and <@colname> in a different order from the create procedure statement if you specify them by name:

```
exec showtype
@colname = qty , @tabname = salesdetail
```

You can use case expressions in any stored procedure where you use a value expression. The following example checks the sales for any book in the titles table:

```
create proc booksales @titleid tid
as
select title, total_sales,
case
when total_sales != null then "Books sold"
when total_sales = null then "Book sales not available"
end
from titles
where @titleid = title_id
```

For example:

```
booksales MC2222
```

```
title                                total_sales
------------------------             -----------
Silicon Valley Gastronomic Treats    2032 Books sold

(1 row affected)
```

Related Information

Naming Convention Identifiers [page 29]

18.4.3 Default Parameters

You can assign a default value for a parameter in the `create procedure` statement. This value, which can be any constant, is used as the argument to the procedure if the user does not supply one.

Here is a procedure that displays the names of all the authors who have written a book published by the publisher given as a parameter. If no publisher name is supplied, the procedure shows the authors published by Algodata Infosystems.

```
create proc pub_info
  @pubname varchar(40) = "Algodata Infosystems" as
select au_lname, au_fname, pub_name
from authors a, publishers p, titles t, titleauthor ta
where @pubname = p.pub_name
and a.au_id = ta.au_id
and t.title_id = ta.title_id
and t.pub_id = p.pub_id
```

If the default value is a character string that contains embedded blanks or punctuation, it must be enclosed in single or double quotes.

When you execute `pub_info`, you can give any publisher's name as the parameter value. If you do not supply any parameter, SAP ASE uses the default, Algodata Infosystems.

```
exec pub_info
```

```
au_lname        au_fname      pub_name
--------------  ------------  --------------------
Green           Marjorie      Algodata Infosystems
Bennet           Abraham       Algodata Infosystems
O'Leary         Michael       Algodata Infosystems
MacFeather      Stearns       Algodata Infosystems
Straight        Dick          Algodata Infosystems
Carson           Cheryl        Algodata   Infosystems
Dull            Ann           Algodata Infosystems
Hunter          Sheryl        Algodata Infosystems
Locksley        Chastity      Algodata Infosystems

(9 rows affected, return status = 0)
```

This procedure, `showind2`, assigns "titles" as the default value for the @`<table>` parameter:

```
create proc showind2
@table varchar(30) = titles as
select table_name = sysobjects.name,
    index_name = sysindexes.name, index_id = indid
from sysindexes, sysobjects
where sysobjects.name = @table
and sysobjects.id = sysindexes.id
```

The column headings, for example, `table_name`, clarify the result display. Here is what `showind2` shows for the `authors` table:

```
showind2 authors
```

```
table_name  index_name        index_id
----------- -------------   ---------
authors     auidind                  1
authors     aunmind                 2

(2 rows affected, return status = 0)
```

If the user does not supply a value, SAP ASE uses the default, `<titles>`.

```
showind2
```

```
table_name  index_name  index_id
----------- ----------- ---------
titles      titleidind         1
titles      titleind           2

(2 rows affected, return status =0)
```

If a parameter is expected but none is supplied, and a default value is not supplied in the `create procedure` statement, SAP ASE displays an error message listing the parameters expected by the procedure.

18.4.3.1 Default Parameters Usage

If you create a stored procedure that uses defaults for parameters, and a user issues the stored procedure, but misspells the parameter name, SAP ASE executes the stored procedure using the default value and does not issue an error message.

For example, if you create the following procedure:

```
create procedure test @x int = 1
as select @x
```

It returns the following:

```
exec test @x = 2
go
----------------
                 2
```

However, if you pass this stored procedure an incorrect parameter, it returns an incorrect result set, but does not issue a error message:

```
exec test @z = 4
go
-----------
          1
(1 row affected)
(return status = 0)
```

18.4.3.2 NULL as the Default Parameter

In the `create procedure` statement, you can declare null as the default value for individual parameters.

The syntax is:

```
create procedure <procedure_name>
     @<param datatype> [ = null ]
     [, @<param datatype> [ = null ]]...
```

If the user does not supply a parameter, SAP ASE executes the stored procedure without displaying an error message.

The procedure definition can specify an action be taken if the user does not give a parameter, by checking to see that the parameter's value is null. Here is an example:

```
create procedure showind3
@table varchar(30) = null as
if @table is null
    print "Please give a table name."
else
   select table_name = sysobjects.name,
     index_name = sysindexes.name,
     index_id = indid
   from sysindexes, sysobjects
   where sysobjects.name = @table
   and sysobjects.id = sysindexes.id
```

If the user does not give a parameter, SAP ASE prints the message from the procedure on the screen.

For other examples of setting the default to null, examine the source text of system procedures using `sp_helptext`.

18.4.3.3 Wildcard Characters in the Default Parameter

The default can include the wildcard characters (%, _, [] , and [^]) if the procedure uses the parameter with the `like` keyword.

For example, you can modify `showind` to display information about the system tables if the user does not supply a parameter, like this:

```
create procedure showind4
@table varchar(30) = "sys%" as
select table_name = sysobjects.name,
    index_name = sysindexes.name,
    index_id = indid
from sysindexes, sysobjects
where sysobjects.name like @table
and sysobjects.id = sysindexes.id
```

18.4.4 Using Multiple Parameters

Examples are provided for using multiple parameters in a procedure.

An example of a variant of `au_info` that uses defaults with wildcard characters for both parameters.

```
create proc au_info2
  @lastname varchar(30) = "D%",
  @firstname varchar(18) = "%" as
select au_lname, au_fname, title, pub_name
from authors, titles, publishers, titleauthor
where au_fname like @firstname
and au_lname like @lastname
and authors.au_id = titleauthor.au_id
and titles.title_id = titleauthor.title_id
and titles.pub_id = publishers.pub_id
```

If you execute `au_info2` with no parameters, all the authors with last names beginning with “D” are returned:

```
au_info2
```

```
au_lname au_fname title                          pub_name
-------- -------  -------------------------   -------------
Dull     Ann      Secrets of Silicon Valley   Algodata Infosystems
DeFrance Michel     The Gourmet Microwave        Binnet & Hardley

(2 rows affected)
```

If defaults are available for parameters, parameters can be omitted at execution, beginning with the last parameter. You cannot skip a parameter unless NULL is its supplied default.

> **Note**
>
> If you supply parameters in the form `<@parameter>` = `<value>`, you can supply parameters in any order. You can also omit a parameter for which a default has been supplied. If you supply one value in the form `<@parameter>` = `<value>`, then supply all subsequent parameters this way.

As an example of omitting the second parameter when defaults for two parameters have been defined, you can find the books and publishers for all authors with the last name “Ringer” like this:

```
au_info2 Ringer
```

```
au_lname   au_fname    title                      Pub_name
--------   --------    ---------------------      ------------
Ringer     Anne        The Gourmet Microwave      Binnet & Hardley
Ringer          Anne             Is Anger the Enemy?         New Age Books
Ringer          Albert            Is Anger the Enemy?        New Age Books
Ringer           Albert           Life Without Fear          New Age Books
```

If a user executes a stored procedure and specifies more parameters than the number of parameters expected by the procedure, SAP ASE ignores the extra parameters. For example, `sp_helplog` displays the following for the `pubs2` database:

```
sp_helplog
```

```
In database ‘pubs2’, the log starts on device ‘pubs2dat’.
```

If you erroneously add some meaningless parameters, the output of `sp_helplog` is the same:

```
sp_helplog one, two, three
```

```
In database 'pubs2', the log starts on device 'pubs2dat'.
```

Remember that SQL is a free-form language. There are no rules about the number of words you can put on a line or where you must break a line. If you issue a stored procedure followed by a command, SAP ASE attempts to execute the procedure and then the command. For example, if you issue:

```
sp_help checkpoint
```

SAP ASE returns the output from `sp_help` and runs the `checkpoint` command. Using delimited identifiers for procedure parameters can produce unintended results.

18.4.5 LOB Datatypes in Stored Procedures

You can declare a large object (LOB) `text`, `image`, or `unitext` datatype for a local variable, and pass that variable as an input parameter to a stored procedure.

This example uses an LOB datatype in a stored procedure:

1. Suppose we create `table_1`:

   ```
   create table t1 (a1 int, a2 text)
           insert into t1 values(1, "aaaa")
           insert into t1 values(2, "bbbb")
           insert into t1 values(3, "cccc")
   ```

2. Create a stored procedure using a LOB local variable as a parameter:

   ```
   create procedure my_procedure @loc text
           as select @loc
   ```

3. Declare the local variable and execute the stored procedure.

   ```
   declare @a text
   select @a = a2 from t1 where a1 = 3
   ```

   ```
   exec my_procedure @a
   ```

   ```
   ---------------------------
   cccc
   ```

Certain restrictions apply. An LOB datatype:

- Cannot be used as an output parameter of a stored procedure
- Cannot be used in a datatype conversion using the `convert()` function
- Is not supported for replication

18.4.6 Procedure Groups

You can optionally use a semicolon and integer number after the name of a procedure in the `create procedure`, and `execute` statements, to group procedures of the same name so that they can be dropped together with a single `drop procedure` statement.

Procedures used in the same application are often grouped this way. For example, you might create a series of procedures called `orders;1`, `orders;2`, and so on. To drop the entire group, use:

```
drop proc orders
```

Once you have grouped procedures by appending a semicolon and number to their names, you cannot drop them individually. For example, the following statement is not allowed:

```
drop proc orders;2
```

To run SAP ASE in the evaluated configuration, prohibit procedure grouping. This ensures that every stored procedure has a unique object identifier and can be dropped individually. To disallow procedure grouping, a system security officer must reset the `allow procedure grouping` configuration parameter. See, *Setting Configuration Parameters*, in the *System Administration Guide: Volume 1*.

18.4.7 with recompile in create procedure

In the `create procedure` statement, the optional clause `with recompile` comes immediately before the SQL statements. It instructs SAP ASE not to save a plan for this procedure. A new plan is created each time the procedure is executed.

In the absence of `with recompile`, SAP ASE stores the execution plan that it created. This execution plan is usually satisfactory. However, a change in the data or parameter values supplied for subsequent executions may cause SAP ASE to create an execution plan that is different from the one it created when the procedure was first executed. In such situations, SAP ASE needs a new execution plan. Use `with recompile` in a `create procedure` statement when you think you need a new plan.

If the operated data for various executions of a stored procedure is not uniform, then the stored procedure should be created using `with recompile`, so that SAP ASE recompiles the stored procedure for each execution rather than using the plan from a previous execution.

The problem of using query plans from a previous execution can be exacerbated from simultaneous executions of the stored procedure, when multiple copies of a stored procedure are included in the procedure cache. If the different executions of the stored procedure used very different data sets, the result is two or more copies of the stored procedure in the procedure cache, each using very different plans. Subsequent executions of the stored procedure will use the copy chosen on the basis of the most recently used (MRU) algorithm.

This problem can cause dramatic performance swings on different executions of the same stored procedure.

> **Note**
>
> When troubleshooting performance problems with stored procedures, use `with recompile` to make sure that each of the stored procedures used during the test are recompiled, so that no plan from a previous compilation is used during the test.

18.4.8 with recompile in execute

In the `execute` statement, the optional clause `with recompile` comes after any parameters. It instructs SAP ASE to compile a new plan, which is used for subsequent executions.

Use `with recompile` when you execute a procedure if your data has changed a great deal, or if the parameter you are supplying is atypical—that is, if you have reason to believe that the plan stored with the procedure might not be optimal for this execution of it.

Using `execute procedure <with recompile>` many times can adversely affect procedure cache performance. Since a new plan is generated every time you use `<with recompile>`, a useful performance plan may age out of the cache if there is insufficient space for new plans.

If you use `select *` in your `create procedure` statement, the procedure, even if you use the `with recompile` option to `execute`, does not pick up any new columns added to the table. You must drop the procedure and re-create it.

18.4.9 Nesting Procedures

Nesting occurs when one stored procedure or trigger calls another. The nesting level is incremented when the called procedure or trigger begins execution and is decremented when the called procedure or trigger completes execution.

The nesting level is also incremented by one when a cached statement is created. Exceeding the maximum of 16 levels of nesting causes the procedure to fail. The current nesting level is stored in the`< @@nestlevel >`global variable.

You can call another procedure by name or by a variable name in place of the actual procedure name. For example:

```
create procedure test1 @proc_name varchar(30)
    as exec @proc_name
```

18.4.10 Temporary Tables in Stored Procedures

You can create and use temporary tables in a stored procedure, but they exists only for the duration of the stored procedure that creates it. When the procedure completes, SAP ASE automatically drops temporary tables.

A single procedure can:

- Create a temporary table
- Insert, update, or delete data
- Run queries on the temporary table
- Call other procedures that reference the temporary table

Since the temporary table must exist to create procedures that reference it, here are the steps to follow:

1. Create the temporary table using a `create table` statement or a `select into` statement. For example:

   ```
   create table #tempstores
       (stor_id char(4), amount money)
   ```

   > **Note**
   >
   > Using `set deferred_name_resolution` makes this step unnecessary.

2. Create the procedures that accesses the temporary table.

   ```
   create procedure inv_amounts as
       select stor_id, "Total Due" = sum(amount)
       from #tempstores
       group by stor_id
   ```

3. Drop the temporary table:

   ```
   drop table #tempstores
   ```

   This step is unnecessary if you use `deferred_name_resolution`.

4. Create the procedure that creates the table and calls the procedures created in step 2:

   ```
   create procedure inv_proc as
   create table #tempstores
   (stor_id char(4), amount money)
   insert #tempstores
   select stor_id, sum(qty*(100-discount)/100*price)
   from salesdetail, titles
   where salesdetail.title_id = titles.title_id
   group by stor_id,
   salesdetail.title_id
   exec inv_amounts
   ```

   When you run the `inv_proc` procedure, it creates the `#tempstores` table, inserts a row into it, and executes `subprocedure inv_amounts`. The `#tempstores` table only exists until `inv_proc` exits. Trying to insert values into the `#tempstores` table or running the `inv_amounts` procedure after executing `inv_proc` will fail:

   ```
   execute inv_proc
   insert #tempstores
   select stor_id,
   sum(qty*(100-discount)/100*price)
   ```

```
from salesdetail, titles
where salesdetail.title_id =
titles.title_id
group by stor_id,
salesdetail.title_id
execute inv_proc
exec inv_amounts
```

Errors are raised because the `#tempstores` table no longer exists after `inv_proc` has completed.

You can also create temporary tables without the # prefix, using `create table tempdb..<tablename>...` from inside a stored procedure. These tables do not disappear when the procedure completes, so they can be referenced by independent procedures. Follow the above steps to create these tables.

Related Information

Deferred Name Resolution Usage [page 502]

18.4.11 Set Options in Stored Procedures

You can use almost all of the `set` command options inside stored procedures. The `set` option remains in effect during the execution of the procedure, and most options revert to the former setting at the close of the procedure.

Only the `dateformat`, `datefirst`, `language`, and `role` options do not revert to their former settings.

However, if you use a `set` option (such as `identity_insert`) that requires the user to be the object owner, a user who is not the object owner cannot execute the stored procedure.

18.4.11.1 Query Optimization Settings

You can export optimization settings, such as `set plan optgoal` and `set plan optcriteria`, using `set export_options on`. Optimization settings are not local to the stored procedure; they apply to the entire user session.

> **Note**
>
> By default, `set export_options` is enabled for login triggers.

18.4.11.2 Maximum Number of Arguments

The maximum number of arguments for stored procedures is 2048. However, you may notice a performance degradation if you execute procedures with large numbers of arguments, because the query processing engine must process all the arguments and copy argument values to and from memory.

SAP recommends that you first test any stored procedures you write that include large numbers of arguments before you implement them in a production environment.

18.4.11.3 Maximum Size for Expressions, Variables, and Arguments

The maximum size for expressions, variables, and arguments passed to stored procedures is 16384 bytes (16K), for any page size. This can be either character or binary data. You can insert variables and literals up to this maximum size into text columns without using the `writetext` command.

Some early versions of SAP ASE had a maximum size of 255 bytes for expressions, variables, and arguments for stored procedures.

Any scripts or stored procedures that you wrote for earlier versions of SAP ASE, restricted by the lower maximum, may now return larger string values because of the larger maximum page sizes.

Because of the larger value, SAP ASE may truncate the string, or the string may cause overflow if it was stored in another variable or inserted into a column or string.

If columns of existing tables are modified to increase the length of character columns, you must change any stored procedures that operate data on these columns to reflect the new length.

```
select datalength(replicate("x", 500)),
    datalength("abcdefgh....255 byte long string.." +
    "xxyyzz ... another 255 byte long string")
```

```
----------- -----------
255         255
```

18.4.12 Execution of Stored Procedures

You can execute stored procedures after a time delay, or remotely.

18.4.12.1 Execute Procedures After a Time Delay

The `waitfor` command delays execution of a stored procedure until a specified time or until a specified amount of time has passed.

For example, to execute the procedure `testproc` in half an hour:

```
begin
    waitfor delay "0:30:00"
    exec testproc
end
```

After issuing the `waitfor` command, you cannot use that connection to SAP ASE until the specified time or event occurs.

18.4.12.2 Execute Procedures Remotely

You can execute procedures on a remote server from your local server.

Once both servers are properly configured, you can execute any procedure on the remote server simply by using the server name as part of the identifier. For example, to execute a procedure named `remoteproc` on a server named GATEWAY:

```
exec gateway.remotedb.dbo.remoteproc
```

The following examples execute the procedure `namelist` in the `pubs2` database on the GATEWAY server:

```
execute gateway.pubs2..namelist
gateway.pubs2.dbo.namelist
exec gateway...namelist
```

The last example works only if `pubs2` is your default database.

See, *Managing Remote Servers*, in the *System Administration Guide: Volume 1*. You can pass one or more values as parameters to a remote procedure from the batch or procedure that contains the `execute` statement for the remote procedure. Results from the remote SAP ASE appear on your local terminal.

Use the return status from procedures to capture and transmit information messages about the execution status of your procedures.

> ⚠ Caution
>
> If Component Integration Services is not enabled, SAP ASE does not treat remote procedure calls (RPCs) as part of a transaction. Therefore, if you execute an RPC as part of a transaction, and then roll back the transaction, SAP ASE does not roll back any changes made by the RPC. When Component Integration

Services is enabled, use `set transactional rpc` and `set cis rpc handling` to use transactional RPCs. See the *Reference Manual: Commands*.

Related Information

Return Status [page 521]

18.4.12.3 Execute a Procedure with execute as owner or execute as caller

You can create a procedure using `execute as owner` or `execute as caller`, which checks runtime permissions, executes DDL, and resolves objects names.

If you create a procedure using `execute as caller`, SAP ASE performs these operations as the procedure caller. If you on create a procedure using `execute as owner`, these operations are performed on the behalf of the procedure owner. Creating procedures for execution as the procedure owner is useful for applications that require all actions in a procedure to be checked against the privileges of the procedure owner. The application end user requires no privilege in the database other than `execute` permission on the stored procedure. Additionally, any DDL executed by the procedure is conducted on behalf of the procedure owner, and any objects created in the procedure are owned by the procedure owner. This relieves the administrator of the requirement of having to grant DDL commands to the application user. Creating the procedures for execution as the session user or caller is necessary if permissions must be checked on behalf of the individual user. For example, use `execute as caller` if a table accessed by the procedure is subject to fine-grained access control through predicated privileges, such that one user is entitled to see one set of rows and another user another set of rows. If the `execute as` is omitted:

- Object names are resolved on behalf of the procedure owner.
- DDL commands and cross-database access are on behalf of the procedure caller.
- Permission checks for DML, `execute`, `truncate table` and `update stats` are made on behalf of the caller unless there exists an ownership chain between the referenced object and the procedure, in which case permission checks are bypassed.

If the `execute as owner` is specified, the procedure behavior conforms to the expected behavior following an implicit `set proxy` to the owner at the beginning of execution. This behavior includes:

- Object names are resolved on behalf of the procedure owner. If the procedure references a table or other object without qualifying the name with an owner name, SAP ASE looks up a table of that name belonging to the procedure owner. If no such table exists, SAP ASE looks for a table of that name owned by the database owner.
- DDL commands and cross-database access are on behalf of the procedure owner.
- All access control checks are based on the procedure owner's permission, his group, his system roles, his default user-defined roles, and those roles granted to the owner that are activated in the procedure body.
- Procedures called from an `execute as owner` procedure are executed as the owner of the calling procedure unless the nested procedure is defined as `execute as owner`.

- Dynamic SQL statements inside a procedure are executed with permissions of procedure owner regardless of the `'Dynamic Ownership Chain'` setting on `sp_procxmode`.
- Because temporary tables are owned by the session, temporary tables created outside the procedure by the caller are available inside the procedure to the procedure owner. This behavior reflects temporary table availability after a `set proxy` command is executed in a session.
- Audit records of statements executed within the procedure show the procedure owner's `suid`.

If the `execute as caller` is specified:

- Objects are resolved on behalf of caller. If the procedure references a table or other object without qualifying the name with an owner name, SAP ASE looks up a table of that name belonging to the user who called the procedure. If no such table exists, SAP ASE looks for a table of that name owned by the database owner.
- No implicit granting of permissions through ownership chains occurs.
- DDL commands and cross-database access are on behalf of the caller.
- Permissions are checked on behalf of caller, caller's group, active roles and system roles.
- Procedures called from an `execute as caller` procedure are executed on behalf of the caller of the parent procedure unless the nested procedure is defined as `execute as owner`.
- Dynamic SQL executes as caller regardless of the `'Dynamic Ownership Chain'` setting on `sp_procxmode`.
- Temporary tables created outside the procedure are available inside the procedure.
- Unqualified object references by the procedure are not entered into `sysdepends`.
- `select *` is not expanded in `syscomments` .
- Plans in the procedure cache for the same procedure are not shared across users.

In the following example, the procedure created by user Jane has no `execute as` clause. The procedure selects from `jane.employee` into an intermediate table named `emp_interim`.:

```
create procedure p_emp
```

```
    select * into emp_interim
    from jane.employee
```

```
grant execute on p_emp to bill
```

Bill executes the procedure:

```
exec jan.p_emp
```

- Bill is not required to have select permission on `jane.employee` because Jane owns `p_emp` and `employee`. By granting `execute` permission on `p_emp` to Bill, Jane has implicitly granted him `select` on `employee`.
- Bill must have been granted create table permission. The `emp_interim` table will be owned by Bill.

In following example, Jane creates a procedure with an identical body using the `execute as owner` clause and Bill execute the procedure:

```
create procedure p_emp
```

```
    with execute as owner as
```

```
    select * into emp_interim
    from jane.employee
```

```
grant execute on p_emp to bill
```

- Bill requires only `execute` permission to run the procedure successfully.
- `emp_interim` table is created on behalf of Jane, meaning Jane is the owner. If Jane does not have create table permission, the procedure will fail.

In following example, Jane creates the same procedure with the `execute as caller` clause:

```
create procedure p_emp
```

```
    with execute as caller as
```

```
    select * into emp_interim
    from jane.employee
```

```
grant execute on p_emp to bill
```

- Bill must have `select` permission on `jane.employee`.
- Bill must have create table permission. `emp_interim` is created on behalf of Bill, meaning Bill is the owner.

18.4.12.3.1 Example with execute as Omitted

Create a procedure with references to an object with an unqualified name.

This is an example where the procedure has no `execute as` clause.

```
create procedure insert_p
```

```
    insert t1 (c1) values (100)
```

```
grant execute on insert_p to bill
```

Bill executes the procedure:

```
exec jane.insert_p
```

- SAP ASE will look for a table named `t1` owned by Jane. If `jane.t1` does not exist, SAP ASE will look for `dbo.t1`.
- If SAP ASE resolves `t1` to `dbo.t1`, permission to insert into `t1` must be held by Bill.
- If `t1` resolves to `jane.t1`, Bill will have implicit insert permission because of the ownership chain between `jane.insert_p` and `jane.t1`.

In the following example, Jane creates the same procedure as above with `execute as owner`:

```
create procedure insert_p
```

```
    with execute as owner as
```

```
    insert t1 (c1) values (100)
```

```
grant execute on insert_p to bill
```

Bill executes the procedure:

```
exec jane.insert_p
```

- SAP ASE will look for a table named `t1` owned by Jane. If `jane.t1` does not exist SAP ASE will look for `dbo.t1`.
- If SAP ASE resolves `t1` to `dbo.t1`, permission to insert into `t1` must been granted to Jane.

In the following example, Jane creates the same procedure as above with `execute as caller`:

```
create procedure insert_p
```

```
    with execute as caller as
```

```
    insert t1 (c1) values (100)
```

```
grant execute on insert_p to bill
```

Bill executes the procedure:

```
exec jane.insert_p
```

- SAP ASE will look for a table named `t1` owned by Bill. If `bill.t1` does not exist SAP ASE will look for `dbo.t1`.
- If SAP ASE resolves `t1` to `dbo.t1`, Bill must have permission to insert into `t1`.

18.4.12.3.2 Example of Procedure with execute as

Create a procedure that invokes a nested procedure in another database with a fully qualified name.

In the following example, Jane creates a procedure that invokes a nested procedure in another database with a fully qualified name. The login associated with Jane resolves to user Jane in `otherdb`.

This example uses `execute as owner`:

```
create procedure p_master
```

```
    with execute as owner
```

```
    as exec otherdb.jim.p_child
```

```
grant execute on p_master to bill
```

Bill executes the procedure:

- SAP ASE checks that user Jane in `otherdb` has `execute` permission on `jim.p_child`.
- If `jim.p_child` has been created `execute as owner` then `p_child` will be executed on behalf of Jim.
- If `jim.p_child` has been created `execute as caller` then `p_child` will execute on behalf of Jane.

In the following example, Jane creates the same procedure as above using `execute as caller`. The login associated with user Bill in the current database resolves to user Bill in `otherdb`:

```
create procedure p_master
```

```
    with execute as caller
```

```
    as exec otherdb.jim.p_child
```

```
grant execute p_master to bill
```

Bill executes the procedure:

```
exec jim.p_master
```

- SAP ASE checks that Bill in `otherdb` has `execute` permission on `jim.p_child`.
- If `jim.p_child` has been created `execute as owner` then `p_child` will be executed on behalf of Jim.
- If `jim.p_child` has been created `execute as caller` then `p_child` will execute on behalf of Bill.

18.5 Deferred Compilation in Stored Procedures

SAP ASE optimizes stored procedures when they are first executed, as long as the values that are passed for variables are available.

With deferred compilation, SAP ASE has already executed statements that appear earlier in the stored procedure, such as statements that assign a value to a local variable or create a temporary table. This means the statement is optimized based on known values and temporary tables, rather than on magic numbers. Using real values allows the optimizer to select a better plan for executing the stored procedure for the given data set.

SAP ASE can reuse the same plan for subsequent executions of the stored procedure, as long as the data operated on is similar to the data used when the stored procedure was compiled.

Deferred compilation is used for stored procedures that reference local variables or temporary tables are not compiled until they are ready to be executed.

Since the plan is optimized specifically for the values and data set used in the first execution, it may not be a good plan for subsequent executions of the stored procedure with different values and data sets.

18.6 Information Returned From Stored Procedures

Stored procedures return the certain types of information, including return status and parameters, and the privileges that are assigned to the user who executed each procedure.

The following types of information are returned:

- Return status – indicates whether or not the stored procedure completed successfully.
- `proc role` function – checks whether the procedure was executed by a user with `sa_role`, `sso_role`, or `ss_oper` privileges.
- Return parameters – report the parameter values back to the caller, who can then use conditional statements to check the returned value.

Return status and return parameters allow you to modularize your stored procedures. A set of SQL statements that are used by several stored procedures can be created as a single procedure that returns its execution status or the values of its parameters to the calling procedure. For example, many SAP ASE system procedures include another procedure that verifies certain parameters as valid identifiers.

Remote procedure calls, which are stored procedures that run on a remote SAP ASE, also return both status and parameters. All the examples below can be executed remotely if the syntax of the execute statement includes the server, database, and owner names, as well as the procedure name.

18.6.1 Return Status

Stored procedures report a return status that indicates whether or not they completed successfully, and if they did not, the reasons for failure.

This value can be stored in a variable when a procedure is called, and used in future Transact-SQL statements. System-defined return status values for failure range from -1 through -99; you can define your own return status values outside this range.

Here is an example of a batch that uses the form of the `execute` statement that returns the status:

```
declare @status int
execute @status = byroyalty 50
select @status
```

The execution status of the `byroyalty` procedure is stored in the variable `<@status>`. “50” is the supplied parameter, based on the `royaltyper` column of the `titleauthor` table. This example prints the value with a `select` statement; later examples use this return value in conditional clauses.

18.6.1.1 Reserved Return Status Values

SAP ASE reserves 0, to indicate a successful return, and negative values from -1 through -99, to indicate the reasons for failure.

Value	Meaning
0	Procedure executed without error
-1	Missing object
-2	Datatype error
-3	Process was chosen as deadlock victim
-4	Permission error
-5	Syntax error
-6	Miscellaneous user error
-7	Resource error, such as out of space
-8	Nonfatal internal problem
-9	System limit was reached
-10	Fatal internal inconsistency
-11	Fatal internal inconsistency
-12	Table or index is corrupt
-13	Database is corrupt
-14	Hardware error

Values -15 through -99 are reserved for future use by SAP ASE.

If more than one error occurs during execution, the status with the highest absolute value is returned.

18.6.1.2 User-Generated Return Values

You can generate your own return values in stored procedures by adding a parameter to the `return` statement. You can use any integer outside the 0 through -99 range.

The following example returns 1 when a book has a valid contract and returns 2 in all other cases:

```
create proc checkcontract @titleid tid
as
if (select contract from titles where
        title_id = @titleid) = 1
```

```
   return 1
else
   return 2
```

For example:

```
checkcontract MC2222
```

```
(return status = 1)
```

The following stored procedure calls `checkcontract`, and uses conditional clauses to check the return status:

```
create proc get_au_stat @titleid tid
as
declare @retvalue int
execute @retvalue = checkcontract @titleid
if (@retvalue = 1)
   print "Contract is valid."
else
    print "There is not a valid contract."
```

Here are the results when you execute `get_au_stat` with the `title_id` of a book with a valid contract:

```
get_au_stat MC2222
```

```
Contract is valid
```

18.6.2 Check Roles in Procedures

If a stored procedure performs system administration or security-related tasks, you may want to ensure that only users who have been granted a specific role can execute it.

The `proc_role` function allows you to check roles when the procedure is executed; it returns 1 if the user possesses the specified role. The role names are `sa_role, sso_role,` and `oper_role`.

Here is an example using `proc_role` in the stored procedure `test_proc` to require the invoker to be a system administrator:

```
create proc test_proc
as
if (proc_role("sa_role") = 0)
begin
    print "You do not have the right role."
    return -1
end
else
    print "You have SA role."
    return 0
```

For example:

```
test_proc
```

```
You have SA role.
```

18.6.3 Return Parameters

Another way that stored procedures can return information to the caller is through return parameters. The caller can then use conditional statements to check the returned value.

When both a `create procedure` statement and an `execute` statement include the `output` option with a parameter name, the procedure returns a value to the caller. The caller can be a SQL batch or another stored procedure. The value returned can be used in additional statements in the batch or calling procedure. When return parameters are used in an `execute` statement that is part of a batch, the return values are printed with a heading before subsequent statements in the batch are executed.

This stored procedure performs multiplication on two integers (the third integer, @`<result>`, is defined as an `output` parameter):

```
create procedure mathtutor
@mult1 int, @mult2 int, @result int output
as
select @result = @mult1 * @mult2
```

To use `mathtutor` to figure a multiplication problem, you must declare the `<@result>` variable and include it in the `execute` statement. Adding the `output` keyword to the `execute` statement displays the value of the return parameters.

```
declare @result int
exec mathtutor 5, 6, @result output
```

```
(return status = 0)

Return parameters:

-----------
         30
```

If you wanted to guess at the answer and execute this procedure by providing three integers, you would not see the results of the multiplication. The `select` statement in the procedure assigns values, but does not print:

```
mathtutor 5, 6, 32
```

```
(return status = 0)
```

The value for the `output` parameter must be passed as a variable, not as a constant. This example declares the @<guess> variable to store the value to pass to `mathtutor` for use in @<result>. SAP ASE prints the return parameters:

```
declare @guess int
select @guess = 32
exec mathtutor 5, 6,
@result = @guess output
```

```
(1 row affected)
(return status = 0)

Return parameters:

@result
-----------
        30
```

The value of the return parameter is always reported, whether or not its value has changed. Note that:

- In the example above, the `output` parameter @<result> *must* be passed as "@<parameter> = @<variable>". If it were not the last parameter passed, subsequent parameters would have to be passed as "@<parameter> = <value>".
- @<result> does not have to be declared in the calling batch; it is the name of a parameter to be passed to `mathtutor`.
- Although the changed value of @<result> is returned to the caller in the variable assigned in the `execute` statement (in this case @<guess>), it appears under its own heading, @<result>.

To use the initial value of @<guess> in conditional clauses after the `execute` statement, store it in another variable name during the procedure call. The following example illustrates the last two bulleted items, above, by using @<store> to hold the value of the variable during the execution of the stored procedure, and by using the "new" returned value of @<guess> in conditional clauses:

```
declare @guess int
declare @store int
select @guess = 32
select @store = @guess
execute mathtutor 5, 6,
@result = @guess output
select Your_answer = @store,
Right_answer = @guess
if @guess = @store
    print "Bingo!"
else
    print "Wrong, wrong, wrong!"
```

```
(1 row affected)
(1 row affected)
(return status = 0)

@result
-----------
         30

 Your_answer Right_answer
 ----------- ------------
          32           30

Wrong, wrong, wrong!
```

This stored procedure checks to determine whether new book sales would cause an author's royalty percentage to change (the @<pc> parameter is defined as an output parameter):

```
create proc roy_check @title tid, @newsales int,
        @pc int output
as
declare @newtotal int
select @newtotal = (select titles.total_sales + @newsales
from titles where title_id = @title)
select @pc = royalty from  roysched
   where @newtotal  >= roysched.lorange and
          @newtotal < roysched.hirange
   and roysched.title_id = @title
```

The following SQL batch calls the roy_check after assigning a value to the <percent> variable. The return parameters are printed before the next statement in the batch is executed:

```
declare @percent int
select @percent = 10
execute roy_check "BU1032", 1050, @pc = @percent output
select Percent = @percent
go
```

```
(1 row affected)
(return status = 0)

@pc
-----------
         12
Percent
-----------
         12

(1 row affected)
```

The following stored procedure calls roy_check and uses the return value for <percent> in a conditional clause:

```
create proc newsales @title tid, @newsales int
as
declare @percent int
declare @stor_pc int
select @percent = (select royalty from roysched, titles
        where roysched.title_id = @title
        and total_sales >= roysched.lorange
        and total_sales < roysched.hirange
        and roysched.title_id = titles.title_id)
select @stor_pc = @percent
execute roy_check @title, @newsales, @pc = @percent
  output
if
  @stor_pc != @percent
begin
  print "Royalty is changed."
  select Percent = @percent
end
else
  print "Royalty is the same."
```

If you execute this stored procedure with the same parameters used in the earlier batch, you see:

```
execute newsales "BU1032", 1050
```

```
Royalty is changed
Percent
-----------
         12

(1 row affected, return status = 0)
```

In the two preceding examples that call `roy_check`, @<pc> is the parameter that is passed to `roy_check`, and <@percent> is the variable containing the output. When `newsales` executes `roy_check`, the value returned in <@percent> may change, depending on the other parameters that are passed. To compare the returned value of <percent> with the initial value of @<pc>, you must store the initial value in another variable. The preceding example saved the value in <stor_pc>.

18.6.3.1 Pass Values in Parameters

You cannot pass constants; there must be a variable name to "receive" the return value. The parameters can be of any SAP ASE datatype except `text`, `unitext`, or `image`.

The syntax is:

```
@<parameter> = @<variable>
```

> **Note**
>
> If the stored procedure requires several parameters, either pass the return value parameter last in the `execute` statement or pass all subsequent parameters in the form <@parameter = value>.

18.6.3.2 The Output Keyword

A stored procedure can return several values; each must be defined as an `output` variable in the stored procedure and in the calling statements. You can abbreviate the `output` keyword to `out`.

```
exec myproc @a = @myvara out, @b = @myvarb out
```

If you specify `output` while you are executing a procedure, and the parameter is not defined using `output` in the stored procedure, you see an error message. It is not an error to call a procedure that includes return value specifications without requesting the return values with `output`. However, you do not get the return values. The stored procedure writer has control over the information users can access, and users have control over their variables.

18.7 Restrictions Associated with Stored Procedures

Several restrictions apply when creating stored procedures.

- You cannot combine `create procedure` statements with other statements in the same batch.
- The `create procedure` definition itself can include any number and kind of SQL statements, except `use` and these `create` statements:
  - `create view`
  - `create default`
  - `create rule`
  - `create trigger`
  - `create procedure`
- You can create other database objects within a procedure. You can reference an object you created in the same procedure, as long as you create it before you reference it. The `create` statement for the object must come first in the actual order of the statements within the procedure.
- Within a stored procedure, you cannot create an object, drop it, and then create a new object with the same name.
- SAP ASE creates the objects defined in a stored procedure when the procedure is executed, not when it is compiled.
- If you execute a procedure that calls another procedure, the called procedure can access objects created by the first procedure.
- You can reference temporary tables within a procedure.
- If you create a temporary table with the # prefix inside a procedure, the temporary table exists only for purposes of the procedure—it disappears when you exit the procedure. Temporary tables created using `create table tempdb..<tablename>` do not disappear unless you explicitly drop them.
- The maximum number of parameters in a stored procedure is 255.
- The maximum number of local and global variables in a procedure is limited only by available memory.

18.7.1 Qualify Names Inside Procedures

Inside a stored procedure, object names used with `create table` and `dbcc` must be qualified with the object owner's name, if other users are to use the stored procedure.

Object names used with other statements, like `select` and `insert`, inside a stored procedure need not be qualified, because the names are resolved when the procedure is compiled.

For example, user "mary," who owns table `marytab`, should qualify the name of her table with her own name when it is used with `select` or `insert`, if she wants other users to execute the procedure in which the table is used. Object names are resolved when the procedure is compiled, and stored as a database ID or object ID pair. If this pair is not available at runtime, the object is resolved again, and if it is not qualified with the owner's name, the server looks for a table called `marytab` owned by the user "mary" and not a table called `marytab` owned by the user executing the stored procedure. If it finds no object ID "marytab," it looks for an object with the same name owned by the database owner.

Thus, if `marytab` is not qualified, and user "john" tries to execute the procedure, SAP ASE looks for a table called `marytab` owned by the owner of the procedure ("mary," in this case) or by the database owner if the

user table does not exist. For example, if the table `mary.marytab` is dropped, the procedure references `dbo.marytab`.

- If you cannot qualify an object name used with `create table` with the object owner's name, use "dbo," or "guest" to qualify the object name.
- If a user with `sa_role` privileges executes the stored procedure, the user should qualify the table name as `tempdb.dbo.mytab`.
- If a user without `sa_role` privileges executes the stored procedure, the user should qualify the table name as `tempdb.guest.mytab`. If an object name in a temporary database is already qualified with the default owner's name, a query such as the following may not return a correct object ID when users without `sa_role` privileges execute the stored procedure:

  ```
  select object_id ('tempdb..mytab')
  ```

  To obtain the correct object ID when you do not have `sa_role` privileges, use the `execute` command:

  ```
  exec("select object_id('tempdb..mytab')")
  ```

18.8 Rename Stored Procedures

Use `sp_rename` to rename stored procedures.

```
sp_rename <objname>, <newname>
```

For example, to rename `showall` to `countall`:

```
sp_rename showall, countall
```

The new name must follow the rules for identifiers. You can change the name only of stored procedures that you own. The database owner can change the name of any user's stored procedure. The stored procedure must be in the current database.

18.8.1 Rename Objects Referenced by Procedures

If you rename any of the objects a stored procedure references, you must drop and re-create the procedure.

Although a stored procedure that references a table or view with a changed name may seem to work fine for a while, it in fact works only until SAP ASE recompiles it. Recompiling occurs for many reasons and without notification to the user.

Use `sp_depends` to get a report of the objects referenced by a procedure.

18.9 Stored Procedures as Security Mechanisms

You can use stored procedures as security mechanisms to control access to information in tables, and to control the ability to perform data modification.

For example, you can deny other users permission to use the `select` command on a table that you own, and create a stored procedure that allows them to see only certain rows or certain columns. You can also use stored procedures to limit `update`, `delete`, or `insert` statements.

The person who owns the stored procedure must own the table or view used in the procedure. Not even a system administrator can create a stored procedure to perform operations on another user's tables, if the system administrator has not been granted permissions on those tables.

See, *Managing User Permissions*, in the *Security Administration Guide*.

18.10 Dropping Stored Procedures

Use `drop procedure` to remove stored procedures.

The syntax is:

```
drop proc[edure] [<owner>.]<procedure_name>
     [, [<owner>.]<procedure_name>] ...
```

If a stored procedure that was dropped is called by another stored procedure, SAP ASE displays an error message. However, if a new procedure of the same name is defined to replace the one that was dropped, other procedures that reference the original procedure can call it successfully.

Once procedures have been grouped, procedures within the group cannot be dropped individually.

18.11 System Procedures

System procedures are shortcuts for retrieving information from the system tables, and mechanisms for performing database administration and other tasks that involve updating system tables.

Usually, system tables can be updated only through stored procedures. A system administrator can allow direct updates of system tables by changing a configuration variable and issuing the `reconfigure with override` command. See, *Managing User Permissions*, in the *Security Administration Guide*.

System procedures are created by the `installmaster` script in the `sybsystemprocs` database during SAP ASE installation. The name of the system procedure usually indicates its purpose. For example `sp_addalias` adds an alias.

18.11.1 Execute System Procedures

You can run system procedures from any database. If a system procedure is executed from a database other than the `sybsystemprocs` database, any references to system tables are mapped to the database from which the procedure is executed.

For example, if the database owner of `pubs2` runs `sp_adduser` from `pubs2`, the new user is added to `pubs2..sysusers`. To run a system procedure in a specific database, either open that database with the `use` command and execute the procedure, or qualify the procedure name with the database name.

When the parameter for a system procedure is an object name, and the object name is qualified by a database name or owner name, the entire name must be enclosed in single or double quotes.

18.11.2 Permissions on System Procedures

Since system procedures are located in the `sybsystemprocs` database, their permissions are also set there.

Some system procedures can be run only by database owners. These procedures ensure that the user executing the procedure is the owner of the database on which they are executed. Other system procedures can be executed by any user who has been granted `execute` permission on them, but this permission must be granted in the `sybsystemprocs` database. This situation has two consequences:

- A user can have permission to execute a system procedure either in all databases or in none of them.
- The owner of a user database cannot directly control permissions on the system procedures within his or her own database.

18.11.3 Types of System Procedures

System procedures can be grouped by function, such as auditing, security administration, data definition, and so on.

See the *Reference Manual: Procedures* for detailed descriptions of all system procedures, listed alphabetically.

18.11.4 Other SAP ASE-Supplied Stored Procedures

SAP ASE provides catalog stored procedures, system extended stored procedures (system ESPs), and `dbcc` procedures.

- Catalog stored procedures – system procedures that retrieve information from the system tables in tabular form.
- Extended stored procedures (ESPs) – call procedural language functions from SAP ASE. The system extended stored procedures, created by `installmaster` at installation, are located in the `sybsystemprocs` database and are owned by the system administrator. They can be run from any database and their names begin with “xp_”.

- dbcc procedures – created by installdbccdb, are stored procedures for generating reports on information created by dbcc checkstorage. These procedures reside in the dbccdb database or in the alternate database, dbccalt.

18.12 Get Information About Stored Procedures

Several system procedures provide information from the system tables about stored procedures.

18.12.1 Get a Report with sp_help

You can get a report on a stored procedure using sp_help. For example, you can get information on the stored procedure byroyalty, which is part of the pubs2 database, using:

```
sp_help byroyalty
```

```
Name       Owner   Object_type        Create_date
--------   ------  ----------------
byroyalty  dbo     stored procedure   Jul 27 2005 4:30PM
(1 row affected)
Parameter_name Type    Length   Prec  Scale  Param_order Mode
-------------- ------  ------   ----  -----  -----------
@percentage    int          4   NULL   NULL            1

(return status = 0)
```

You can get help on a system procedure by executing sp_help when using the sybsystemprocs database.

18.12.2 View the Source Text of a Procedure with sp_helptext

To display the source text of the create procedure statement, execute sp_helptext.

```
sp_helptext byroyalty
```

```
# Lines of Text
---------------
              1

(1 row affected)

text
----------------------------------------------------
create procedure byroyalty @percentage int
as
select au_id from titleauthor
where titleauthor.royaltyper = @percentage

(1 row affected, return status = 0)
```

You can view the source text of a system procedure by executing `sp_helptext` when using the `sybsystemprocs` database.

If the source text of a stored procedure was encrypted using `sp_hidetext`, SAP ASE displays a message advising you that the text is hidden. See the *Reference Manual: Procedures*.

18.12.3 Identify Dependent Objects with sp_depends

`sp_depends` lists all the stored procedures that reference the object you specify or all the procedures that it is dependent upon.

For example, this command lists all the objects referenced by the user-created stored procedure `byroyalty`:

```
sp_depends byroyalty
```

```
Things the object references in the current database.
object           type       updated      selected
---------------- ---------- ---------    --------
dbo.titleauthor  user table  no          no

(return status = 0)
```

The following statement uses `sp_depends` to list all the objects that reference the table `titleauthor`:

```
sp_depends titleauthor
```

```
Things inside the current database that reference the object.

object          type
--------------  ------------------
dbo.byroyalty   stored procedure
dbo.titleview   view

(return status = 0)
Dependent objects that reference all columns in the table. Use sp_depends on
each column to get more information. Columns referenced in stored procedures
views, or triggers are not included in this report.
........
(1 row affected)
(return status = 0)
```

You must drop and re-create the procedure if any of its referenced objects have been renamed.

18.12.3.1 Use sp_depends with deferred_name_resolution

Since procedures created using deferred name resolution dependency information are created at execution, a message is raised when `sp_depends` executes a procedure created with deferred name resolution but not yet executed.

```
sp_depends p
----------
```

```
The dependencies of the stored procedure cannot be determined until the first
successful execution.
(return status = 0)
```

After the first successful execution, the dependency information is created and the execution of `sp_depends` returns the expected information.

For example:

```
 set deferred_name_resolution on
-----------------create procedure p as
select id from sysobjects
where id =1sp_depends p
------------
The dependencies of the stored procedure cannot be determined until the first
successful execution.(return status = 0)exec p
```

```
id -----------
          1
(1 row affected)
(return status = 0)
sp_depends p
----------------
The object references in the current database.
object                       type
updated
         selected---------------------------- ------------------------
--------------
         ---------------- dbo.sysobjects                          system table
           no
                      no(return status = 0)
```

18.12.4 Identify Permissions with sp_helprotect

`sp_helprotect` reports permissions on a stored procedure (or any other database object).

For example:

```
sp_helprotect byroyalty
```

```
grantor    grantee    type  action    object    column  grantable
---------  ---------  ----  --------- -------   -----   ---------
dbo        public     Grant Execute   byroyalty All     FALSE

(return status = 0)
```

19 Extended Stored Procedures Usage

Extended stored procedures (ESPs) provide a mechanism for calling external procedural language functions from within SAP ASE. Users invoke ESPs using the same syntax as stored procedures; the difference is that an ESP executes procedural language code rather than Transact-SQL statements.

Each ESP is associated with a corresponding function, which is executed when the ESP is invoked from SAP ASE.

An ESP allows SAP ASE to perform a task outside SAP ASE in response to an event occurring within SAP ASE. For example, you could create an ESP function to sell a security. This ESP is invoked in response to a trigger, fired when the price of the security reaches a certain value. Or you could create an ESP function that sends an e-mail notification or a network-wide broadcast in response to an event occurring within the relational database system.

For the purposes of ESPs, "a procedural language" is a programming language that is capable of calling a C language function and manipulating C-language datatypes.

After a function has been registered in a database as an ESP, it can be invoked just like a stored procedure from `isql`, from a trigger, from another stored procedure, or from a client application.

ESPs can:

- Take input parameters
- Return a status value indicating success or failure and the reason for the failure
- Return values of output parameters
- Return result sets

SAP ASE supplies some system ESPs. For example, one system ESP, `xp_cmdshell`, executes an operating system command from within SAP ASE. You can also write your own ESPs using a subset of the Open Server application programming interface (API).

19.1 XP Server

Extended stored procedures are implemented by an Open Server application called XP Server, which runs on the same machine as SAP ASE. SAP ASE and XP Server communicate through remote procedure calls (RPCs).

Running ESPs in a separate process prevents SAP ASE being affected by failures that may result from faulty ESP code. The advantage of using ESPs over RPCs is that the ESP runs in SAP ASE the same way a stored procedure runs; you do not need to have Open Server to run the ESP.

XP Server is automatically installed with SAP ASE. However, if you intend to develop XP Server libraries, you must purchase an Open Server license. Everything you need to use XP Server DLLs and run XP Server commands is included with your SAP ASE license.

XP Server must be running for SAP ASE to execute an ESP. SAP ASE starts XP Server the first time an ESP is invoked and shuts down XP Server when SAP ASE exits.

Normally, there is no reason for a user to start XP Server manually, since SAP ASE starts it when it receives the first ESP request of the session. However, if you are creating and debugging your own ESPs, you may find it necessary to manually start XP Server from the command line using the `xpserver` utility.

See the *Utility Guide* for the syntax of `xpserver`.

19.1.1 CIS RPC Mechanism

You can execute XP Server procedures using a CIS RPC mechanism, in addition to routing through the site handler.

Set the options needed to use this mechanism, `cis rpc handling` and `negotiated logins`, by entering:

```
//to set 'cis rpc handling'//
```

```
sp_configure 'cis rpc handling', 1
//or at the session level//
set 'cis rpc handling' on
```

```
//to set 'negotiated logins'//
sp_serveroption XPServername, 'negotiated logins', true
```

If either option is not set, the ESP is routed through the site handler, and no warning message appears.

19.1.2 sybesp_dll_version

SAP recommends that all libraries loaded into XP Server implement the function `sybesp_dll_version`. The function returns the Open Server API version used by the DLL.

Enter:

```
CS_INT sybesp_dll_version()
----------
CS_CURRENT_VERSION
```

`CS_CURRENT_VERSION` is a macro defined in the Open Server API. Using this DLL prevents you from having to hard-code a specific value. If `CS_CURRENT_VERSION` is not implemented, XP Server does not attempt version matching, and prints error message 11554 in the log file. (For information on error messages, see the *Troubleshooting and Error Messages Guide*.) However, XP Server continues loading the DLL.

If there is a mismatch in versions, XP Server prints error message 11555 in the log file, but continues loading the DLL.

19.2 Dynamic Link Library Support

The procedural functions that contain the ESP code are compiled and linked into dynamic link libraries (DLLs), which are loaded into XP Server memory in response to an ESP execution request.

The library remains loaded unless:

- XP Server exits
- The `sp_freedll` system procedure is invoked
- The `esp unload dll` configuration parameter is set using `sp_configure`

19.3 Open Server API

SAP ASE uses the Open Server API, which allows users to run the system ESPs provided with SAP ASE. Users can also use the Open Server API to implement their own ESPs.

This table lists the Open Server routines required for ESP development. For complete documentation of these routines, see the *Open Server Server-Library/C Reference Manual*.

Function	Purpose
`srv_bind`	Describes and binds a program variable to a parameter
`srv_descfmt`	Describes a parameter
`srv_numparams`	Returns the number of parameters in the ESP client request
`srv_senddone`	Sends results completion message
`srv_sendinfo`	Sends messages
`srv_sendstatus`	Sends status value
`srv_xferdata`	Sends and receives parameters or data
`srv_yield`	Suspends execution of the current thread and allows another thread to execute

After an ESP function has been written, compiled, and linked into a DLL, you can create an ESP for the function using the `as external name` clause of the `create procedure` command:

```
create procedure <procedure_name> [<parameter_list>]
     as external name <dll_name>
```

`<procedure_name>` is the name of the ESP, which must be the same as the name of its implementing function in the DLL. ESPs are database objects, and their names must follow the rules for identifiers.

`<dll_name>` is the name of the DLL in which the implementing function is stored. The naming conventions for DLL are platform-specific.

Table 8: Naming conventions for DLL extensions

Platform	DLL extension
HP 9000/800 HP-UX	`.sl`
Sun Solaris	`.so`
Windows	`.dll`

The following statement creates an ESP named `getmsgs`, which is in `msgs.dll`. The `getmsgs` ESP takes no parameters. This example is for a Windows SAP ASE:

```
create procedure getmsgs
as external name "msgs.dll"
```

The next statement creates an ESP named `getonemsg`, which is also in `msgs.dll`. The `getonemsg` ESP takes a message number as a single parameter.

```
create procedure getonemsg @msg int
as external name "msgs.dll"
```

When SAP ASE creates an ESP, it stores the procedure's name in the `sysobjects` system table, with an object type of "XP" and the name of the DLL containing the ESP's function in the `text` column of the `syscomments` system table.

Execute an ESP as if it were a user-defined stored procedure or system procedure. You can use the keyword `execute` and the name of the stored procedure, or just give the procedure's name, as long as it is submitted to SAP ASE by itself or is the first statement in a batch. For example, you can execute `getmsgs` in any of these ways:

```
getmsgs
execute getmsgs
exec getmsgs
```

You can execute `getonemsg` in any of these ways:

```
getonemsg 20
getonemsg @msg=20
execute getonemsg 20
execute getonemsg @msg=20
exec getonemsg 20
exec getonemsg @msg=20
```

19.4 ESPs and Permissions

You can grant and revoke permissions on an ESP as you would on a regular stored procedure.

In addition to normal SAP ASE security, you can use the `xp_cmdshell context` configuration parameter to restrict execution permission of `xp_cmdshell` to users who have system administration privileges. Use this configuration parameter to prevent ordinary users from using `xp_cmdshell` to execute operating system

commands that they would not have permission to execute directly from the command line. The behavior of the `xp_cmdshell` configuration parameter is platform-specific.

By default, a user must have the `sa_role` to execute `xp_cmdshell`. To grant permission to other users to use `xp_cmdshell`, use the `grant` command. You can revoke the permission with `revoke`. The `grant` or `revoke` permission is applicable whether `xp_cmdshell` context is set to 0 or 1.

19.5 ESPs and Performance

Since both SAP ASE and XP Server reside on the same machine, they can affect each other's performance when XP Server is executing a function that consumes significant resources.

Use `esp execution priority` to set the priority of the XP Server thread high, so the Open Server scheduler runs it before other threads on its run queue, or low, so the scheduler runs XP Server only when there are no other threads to run. The default value of `esp execution priority` is 8, but you can set it anywhere from 0 to 15.

All ESPs running on the same server must yield to one another, using the Open Server `srv_yield` routine to suspend their XP Server thread and allow another thread to execute.

See the discussion of multithread programming in the *Open Server Server-Library/C Reference Manual*.

You can minimize the amount of memory XP Server uses by unloading a DLL from XP Server memory after the ESP request that loaded it terminates. To do so, set `esp unload dll` so that the DLLs are automatically unloaded when ESP execution finishes. If `esp unload dll` is not set, you can use `sp_freedll` to explicitly free DLLs.

You cannot unload DLLs that support system ESPs.

19.6 Create Functions for ESPs

A function that implements an ESP must be written in a procedural programming language that is capable of calling a C-language function, manipulating C-language datatypes, and linking with the Open Server API. .

However, an ESP function should not call a C runtime signal routine on Windows. This can cause XP Server to fail, because Open Server does not support signal handling on Windows.

By using the Open Client API, an ESP function can send requests to SAP ASE, either to the one from which it was originally invoked or to another one.

19.6.1 Files for ESP Development

To use the Open Server libraries for development, you must first purchase an Open Server license. The header files needed for ESP development are in `$SAP/$SAP_OCS/include`.

To locate these files in your source files, include the following in the source code:

- `ospublic.h`
- `oserror.h`

The Open Server library is in `$SAP/$SAP_OCS/lib`.The source for the sample program is in `$SAP/$SAP_ASE/sample/esp`.

Related Information

ESP Function Example [page 541]

19.6.2 Open Server Data Structures

Certain data structures are available for writing ESP functions.

- SRV_PROC – all ESP functions are coded to accept a single parameter, which is a pointer to a SRV_PROC structure. The SRV_PROC structure passes information between the function and its calling process. ESP developers cannot directly manipulate this structure.
  The ESP function passes the SRV_PROC pointer to the Open Server routines that get parameter types and data and return output parameters, status codes, and result sets.
- CS_SERVERMSG – Open Server uses the CS_SERVERMSG structure to send error messages to a client using the `<srv_sendinfo>` routine. See the *Open Server-Library/C Reference Manual* for information about CS_SERVERMSG..
- CS_DATAFMT – Open Server uses the CS_DATAFMT structure to describe data values and program variables.

19.6.3 Open Server Return Codes

Open Server functions return a code of type CS_RETCODE.

The most common CS_RETCODE values for ESP functions are:

- CS_SUCCEED
- CS_FAIL

19.6.4 Outline of a Simple ESP Function

An ESP function should follow a certain structure with regard to its interaction with the Open Server API.

Context

The structure should be:

Procedure

1. Get the number of parameters.
2. Get the values of the input/output parameters and bind them to local variables.
3. Use the input parameter values to perform processing, and store the results in local variables.
4. Initialize any output parameters with appropriate values, bind them with local variables, and transfer them to the client.
5. Use `srv_sendinfo` to send the returned row to the client.
6. Use `srv_sendstatus` to send the status to the client.
7. Use `srv_senddone` to inform the client that processing is done.
8. If there is an error condition, use `srv_sendinfo` to send the error message to the client.

Results

See the *Open Server Server-Library/C Reference Manual* for documentation of the Open Server routines.

19.6.5 ESP Function Example

`xp_echo.c` includes an ESP that accepts a user-supplied input parameter and echoes it to the ESP client, which invokes the ESP. An example demonstrates the inclusion of the `xp_message` function, which sends messages and status, and the `xp_echo` function which processes the input parameter and performs the echoing.

You can use this example as a template for building your own ESP functions. The source is in `$SAP/$SAP_ASE/sample/esp`.

```
/*
** xp_echo.c
**
**      Description:
**              The following sample program is generic in
```

```
**              nature. It echoes an input string which is
**              passed as the first parameter to the xp_echo
**              ESP. This string is retrieved into a buffer
**              and then sent back (echoed) to the ESP client.
*/
#include <string.h>
#include <stdlib.h>
#include <malloc.h>
/* Required Open Server include files.*/
#include <ospublic.h>
#include <oserror.h>
/*
** Constant defining the length of the buffer that receives the
** input string. All of the  parameters related
** to ESP may not exceed 255 char long.
*/
#define ECHO_BUF_LEN    255
/*
** Function:
**      xp_message
**      Purpose: Sends information, status and completion of the
**      command to the server.
** Input:
**      SRV_PROC *
**      char *  a message string.
** Output:
**      void
*/
void xp_message
(
        SRV_PROC *srvproc,  /* Pointer to Open Server thread
                             control structure */
        char     *message_string /* Input message string */
)
{
        /*
        ** Declare a variable that will contain information
        ** about the message being sent to the SQL client.
        */
        CS_SERVERMSG    *errmsgp;
        /*
        ** A SRV_DONE_MORE instead of a SRV_DONE_FINAL must
        ** complete the result set of an Extended Stored
        ** Procedure.
        */
        srv_senddone(srvproc, SRV_DONE_MORE, 0, 0);
        free(errmsgp);
}
/*
** Function: xp_echo
** Purpose:
**      Given an input string, this string is echoed as an output
**      string to the corresponding SQL (ESP) client.
** Input:
**      SRV_PROC *
** Output
**      SUCCESS or FAILURE
*/
CS_RETCODE xp_echo
(
        SRV_PROC        *srvproc
)
{
        CS_INT          paramnum;  /* number of parameters */
        CS_CHAR         echo_str_buf[ECHO_BUF_LEN + 1];
                                /* buffer to hold input string */
        CS_RETCODE      result = CS_SUCCEED;
        CS_DATAFMT      paramfmt; /* input/output param format */
```

```
        CS_INT          len;      /* Length of input param */
        CS_SMALLINT     outlen;
        /*
        ** Get number of input parameters.*/
        */
        srv_numparams(srvproc, &paramnum);
        /*
        ** Only one parameter is expected.*/
        */
        if (paramnum != 1)
        {
                /*
                ** Send a usage error message.*/
                */
                xp_message(srvproc, "Invalid number of
                 parameters");
                result = CS_FAIL;
        }
        else
        {
                /*
                ** Perform initializations.
                */
                outlen = CS_GOODDATA;
                memset(&paramfmt, (CS_INT)0,
                  (CS_INT)sizeof(CS_DATAFMT));
                /*
                ** We are receiving data through an ESP as the
                ** first parameter. So describe this expected
                ** parameter.
                */
                if ((result == CS_SUCCEED) &&
                  srv_descfmt(srvproc, CS_GET
                  SRV_RPCDATA, 1, &paramfmt) != CS_SUCCEED)
                {
                        result = CS_FAIL;
                }
                /*
                ** Describe and bind the buffer to receive the
                ** parameter.
                */
                if ((result == CS_SUCCEED) &&
                  (srv_bind(srvproc, CS_GET, SRV_RPCDATA,
                   1, &paramfmt,(CS_BYTE *) echo_str_buf,
                   &len, &outlen) != CS_SUCCEED))
                {
                        result = CS_FAIL;
                }
                /* Receive the expected data.*/
                if ((result == CS_SUCCEED) &&
                  srv_xferdata(srvproc,CS_GET,SRV_RPCDATA)
                  != CS_SUCCEED)
                {
                        result = CS_FAIL;
                }                /*
                ** Now we have the input info and are ready to
                ** send the output info.
                */
                if (result == CS_SUCCEED)
                {
                        /*
                        ** Perform initialization.
                        */
                        if (len == 0)
                                outlen = CS_NULLDATA;
                        else
                                outlen = CS_GOODDATA;
                        memset(&paramfmt, (CS_INT)0,
```

```
                          (CS_INT)sizeof(CS_DATAFMT));
                        strcpy(paramfmt.name, "xp_echo");
                        paramfmt.namelen = CS_NULLTERM;
                        paramfmt.datatype = CS_CHAR_TYPE;
                        paramfmt.format = CS_FMT_NULLTERM;
                        paramfmt.maxlength = ECHO_BUF_LEN;
                        paramfmt.locale = (CS_LOCALE *) NULL;
                        paramfmt.status |= CS_CANBENULL;
                        /*
                        ** Describe the data being sent.
                        */
                        if ((result == CS_SUCCEED) &&
                                srv_descfmt(srvproc, CS_SET,
                                  SRV_ROWDATA, 1, &paramfmt)
                                  != CS_SUCCEED)
                        {
                                result = CS_FAIL;
                        }
                        /*
                        ** Describe and bind the buffer that
                        ** contains the data to be sent.
                        */
                        if ((result == CS_SUCCEED) &&
                                (srv_bind(srvproc, CS_SET,
                                SRV_ROWDATA, 1,
                                &paramfmt, (CS_BYTE *)
                                echo_str_buf, &len, &outlen)
                                != CS_SUCCEED))
                        {
                                result = CS_FAIL;
                        }
                        /*
                        ** Send the actual data.
                        */
                        if ((result == CS_SUCCEED) &&
                                srv_xferdata(srvproc, CS_SET,
                                SRV_ROWDATA)!= CS_SUCCEED)
                        {
                                result = CS_FAIL;
                        }
                }
                /*
                ** Indicate to the ESP client how the
                ** transaction was performed.
                */
                if (result == CS_FAIL)
                        srv_sendstatus(srvproc, 1);
                else
                        srv_sendstatus(srvproc, 0);
                /*
                ** Send a count of the number of rows sent to
                ** the client.
                */
                srv_senddone(srvproc,(SRV_DONE_COUNT |
                  SRV_DONE_MORE), 0, 1);
        }
        return result;
}
```

19.6.6 Building the DLL

To compile a function that uses the Open Server API, you can use any compiler that can produce the required DLL on your server platform.

For general information about compiling and linking, see the *Open Client/Server Supplement*.

Search Order for DLLs

Windows searches for DLLs in a ccertain order.

The order is:

1. The directory from which the application was invoked
2. The current directory
3. The system directory (SYSTEM32)
4. Directories listed in the PATH environment variable

UNIX searches for the library in the directories listed in the LD_LIBRARY_PATH environment variable (on Solaris), SHLIB_PATH (on HP), or LIBPATH in AIX, in the order in which they are listed.

If XP Server does not find the library for an ESP function in the search path, it attempts to load it from `$SAP/DLL` on Windows or `$SAP/lib` on other platforms.

Absolute path names for the DLL are not supported.

Sample Makefile (UNIX)

A sample makefile, `make.unix`, used to create the dynamically linked shared library for the `xp_echo` program on UNIX platforms.

It generates a file named `examples.so` on Solaris, and `examples.sl` on HP. The source is in `$SAP/$SAP_ASE/sample/esp`, so you can modify it for your own use.

To build the example library using this makefile, enter:

```
make -f make.unix
```

```
#
# This makefile creates a shared library.  It needs the open
# server header
# files usually installed in $SAP/include directory.
# This make file can be used for generating the template ESPs.
# It references the following macros:
#
# PROGRAM is the name of the shared library you may want to
create.PROGRAM        = example.so
BINARY        = $(PROGRAM)
EXAMPLEDLL    = $(PROGRAM)
# Include path where ospublic.h etc reside. You may have them in
# the standard places like /usr/lib etc.
```

```
INCLUDEPATH    = $(SAP)/include
# Place where the shared library will be generated.
DLLDIR        = .
RM        = /usr/bin/rm
ECHO        = echo
MODE        = normal
# Directory where the source code is kept.
SRCDIR        = .
# Where the objects will be generated.
OBJECTDIR    = .
OBJS        = xp_echo.o
CFLAGS        = -I$(INCLUDEPATH)
LDFLAGS        =  $(GLDFLAGS) -Bdynamic
DLLLDFLAGS    =  -dy -G
#===============================================================
$(EXAMPLEDLL) : $(OBJS)
    -@$(RM) -f $(DLLDIR)/$(EXAMPLEDLL)
    -@$(ECHO) "Loading $(EXAMPLEDLL)"
    -@$(ECHO) " "
    -@$(ECHO) "    MODE:           $(MODE)"
    -@$(ECHO) "    OBJS:           $(OBJS)"
    -@$(ECHO) "    DEBUGOBJS:      $(DEBUGOBJS)"
    -@$(ECHO) " "
    cd $(OBJECTDIR); \
    ld -o $(DLLDIR)/$(EXAMPLEDLL) $(DEBUGOBJS) $(DLLLDFLAGS) $(OBJS)
    -@$(ECHO) "$(EXAMPLEDLL) done"
    exit 0
#===============================================================
$(OBJS) : $(SRCDIR)/xp_echo.c
    cd $(SRCDIR); \
    $(CC) $(CFLAGS) -o $(OBJECTDIR)/$(OBJS) -c xp_echo.c
```

Sample Definitions File

A sample definitions file that lists every function to be used as an ESP function in the EXPORTS section.

The following file, `xp_echo.def`, must be in the same directory as `xp_echo.mak`.

```
LIBRARY   examples
CODE      PRELOAD MOVEABLE DISCARDABLE
DATA      PRELOAD SINGLE
EXPORTS
        xp_echo  .   1
```

19.7 Registering ESPs

Once you have created an ESP function and linked it into a DLL, register it as an ESP in a database, which lets users execute the function as an ESP.

To register an ESP, use either:

- The Transact-SQL `create procedure` command, or,
- `sp_addextendedproc`.

19.7.1 create procedure Usage

When creating a procedue, the name must be the same as its supporting function.

The syntax is:

```
create procedure [<owner>.]<procedure_name>
     [[(]@<parameter_name> <datatype> [= default] [output]
     [, @<parameter_name> <datatype> [= default]
     [output]]...[)]] [with recompile]
     as external name <dll_name>
```

`<procedure_name>` is the name of the ESP as it is known in the database.

SAP ASE ignores the `with recompile` clause if it is included in a `create procedure` command used to create an ESP.

The `<dll_name>` is the name of the library containing the ESP's supporting function. Specify it as a name with no extension (for example, `msgs`), or as a name with a platform-specific extension, such as `msgs.dll` on Windows or `msgs.so` on Solaris. In either case, the platform-specific extension is assumed to be part of the library's actual file name.

Since `create procedure` registers an ESP in a specific database, specify the database in which you are registering the ESP before invoking the command. From `isql`, specify the database with the `use <database>` command, if you are not already working in the target database.

The following statements register an ESP supported by the `xp_echo` routine, assuming that the function is compiled in a DLL named`< examples.dll>`. The ESP is registered in the `pubs2` database.

```
use pubs2
create procedure xp_echo @in varchar(255)
as external name "examples.dll"
```

See the *Reference Manual: Commands*.

Related Information

Stored Procedures [page 498]
ESP Function Example [page 541]

19.7.2 sp_addextendedproc Usage

You can use `sp_addextendedproc` as an alternative to `create procedure`.

The syntax is:

```
sp_addextendedproc <esp_name>, <dll_name>
```

`<esp_name>` is the name of the ESP. It must match the name of the function that supports the ESP.

<dll_name> is the name of the DLL containing the ESP's supporting function.

sp_addextendedproc must be executed in the master database by a user who has the sa_role. Therefore, sp_addextendedproc always registers the ESP in the master database, unlike create procedure, which registers the ESP in the current database. Unlike create procedure, sp_addextendedproc does not allow for parameter checking in SAP ASE or for default values for parameters.

The following statements register in the master database an ESP supported by the xp_echo routine, assuming that the function is compiled in a DLL named< examples.dll>:

```
use master
sp_addextendedproc "xp_echo", "examples.dll"
```

See the *Reference Manual: Procedures*.

Related Information

ESP Function Example [page 541]

19.8 Remove ESPs

To remove an ESP from the database, use either drop procedure or sp_dropextendedproc.

The syntax for drop procedure is the same as for stored procedures:

```
drop procedure [<owner>.]<procedure_name>
```

For example:

```
drop procedure xp_echo
```

The syntax for sp_dropextendedproc is:

```
sp_dropextendedproc <esp_name>
```

For example:

```
sp_dropextendedproc xp_echo
```

Both methods drop the ESP from the database by removing references to it in the sysobjects and syscomments system tables. They have no effect on the underlying DLL.

19.8.1 Renaming ESPs

Because an ESP name is bound to the name of its function, you cannot rename an ESP using `sp_rename`, as you can with a stored procedure.

Procedure

1. Use `drop procedure` or `sp_dropextendedproc` to remove the ESP.
2. Rename and recompile the supporting function.
3. `create procedure` or `sp_addextendedproc` to re-create the ESP, assigning to it the new name.

19.9 Execute ESPs

Execute an ESP using the same `execute` command that you use to execute a regular stored procedure.

You can also execute an ESP remotely.

Because the execution of any ESP involves a remote procedure call between SAP ASE and XP Server, you cannot combine parameters by name and parameters by value in the same `execute` command. All the parameters must be passed by name, or all must be passed by value. This is the only way in which the execution of extended stored procedures differs from that of regular stored procedures.

ESPs can return:

- A status value indicating success or failure, and the reason for failure
- Values of output parameters
- Result sets

An ESP function reports the return status value with the `srv_sendstatus` Open Server routines. The return status values from `srv_sendstatus` are application-specific. However, a status of zero indicates that the request completed normally.

When there is no parameter declaration list for an extended stored procedure, SAP ASE ignores all supplied parameters but issues no error message. If you supply more parameters when you execute the ESP than you declare in the declaration list, SAP ASE calls them anyway. To avoid confusion about what parameters are called, check that the parameters on the declaration list match the parameters supplied at run-time. Also check the number of parameters in the specified ESP at the Open Server function build.

An ESP function returns the values of output parameters and result sets using the `srv_descfmt`, `srv_bind`, and `srv_xferdata` Open Server routine.

See the *Open Server Server-Library/C Reference Manual* for more information about passing values from an ESP function. From the SAP ASE side, returned values from an ESP are handled the same as for a regular stored procedure.

Related Information

Create and Execute Stored Procedures [page 502]
Execute Procedures Remotely [page 515]
ESP Function Example [page 541]

19.10 System ESPs

In addition to `xp_cmdshell`, there are several system ESPs that support Windows features, such as the integration of SAP ASE with the Windows Event Log or mail system.

The names of all system ESPs begin with "xp_". They are created in the `sybsystemprocs` database during SAP ASE installation. Since system ESPs are located in the `sybsystemprocs` database, their permissions are set there. However, you can run system ESPs from any database. The system ESPs are:

- `xp_cmdshell`
- `xp_enumgroups`
- `xp_logevent`

See, *System Extended Stored Procedures*, in the *Reference Manual: Procedures* for information about the system ESPs, and the *Configuration Guide for Windows* discusses some specific features in more detail, such as Event Log integration.

19.11 Get Information About ESPs

Use `sp_helpextendedproc` to get information about ESPs that are registered in the current database.

With no parameters, `sp_helpextendedproc` displays all the ESPs in the database, with the names of the DLLs containing their associated functions. With an ESP name as a parameter, it provides the same information only for the specified ESP.

```
sp_helpextendedproc getmsgs
```

```
ESP Name    DLL
--------    ---------
getmsgs     msgs.dll
```

Since the system ESPs are in the `sybsystemprocs` database, you must be using the `sybsystemprocs` database to display their names and DLLs:

```
use sybsystemprocs
sp_helpextendedproc
```

```
ESP Name      DLL
--------      ---------
```

```
xp_freedll   sybyesp
xp_cmdshell  sybyesp
```

If the source text of an ESP was encrypted using `sp_hidetext`, SAP ASE displays a message advising you that the text is hidden. See the *Reference Manual: Procedures*.

19.12 ESP Exceptions and Messages

SAP ASE handles all messages and exceptions from XP Server. It logs standard ESP messages in the log file in addition to sending them to the client. User-defined messages from user-defined ESPs are not logged, but they are sent to the client.

ESP-related messages may be generated by XP Server, by a system procedure that creates or manipulates ESPs, or by a system ESP. See the *Troubleshooting and Error Messages Guide* for the list of ESP-related messages.

A function for a user-defined ESP can generate a message using the `srv_sendinfo` Open Server routine.

Related Information

ESP Function Example [page 541]

20 Cursors: Accessing Data

A cursor accesses the results of a SQL `select` statement one or more rows at a time. Cursors allow you to modify or delete individual rows or a group of rows.

Cursors are associated with a `select` statement.

They consists of:

- Cursor result set – the set (table) of qualifying rows that results from the execution of a query associated with the cursor.
- Cursor position – a pointer to a row within the cursor result set. If a cursor is not specified as read-only, you can explicitly modify or delete that row using `update` or `delete` statements.

You can use two keywords to specify sensitivity when declaring a cursor.

- `insensitive`
- `semi_sensitive`

If you declare a cursor `insensitive`, the cursor shows only the result set as it is when the cursor is opened; data changes in the underlying tables are invisable. If you declare a cursor `semi_sensitive` (the default value), some changes in the base tables made since opening the cursor may appear in the result set. Data changes may or may not be visible to a semisensitive cursor.

There are also two keywords to specify scrollability:

- `scroll`
- `no scroll`

> **Note**
>
> Scrollable cursor allows you to select one or several rows, and to scroll back and forth among them. Scrollable cursors are always read-only. Using a nonscrollable cursor, you cannot go back to a row you have already selected, and you cannot move more than one row at a time.SAP continues to support the default, forward-only cursor, but recommends that you use the more convenient and flexible scrollable cursor whenever you do not need to update a result set through a cursor.

If you use `scroll` to declare a cursor, you can fetch the result rows either sequentially or non-sequentially, and you can scan the result set repeatedly. If `no scroll` (the default value) appears in the cursor declaration, the cursor is nonscrollable; the result set appears in a forward-only direction, one row at a time.

If you specify neither attribute, the default value is `no scroll`.

Think of a cursor as a “handle” on the result set of a `select` statement. The cursor can be fetched either sequentially or nonsequentially, depending on the cursor’s scrollability.

A nonscrollable cursor can be fetched only in a forward direction; you cannot go back to a row that is already fetched. A scrollable cursor can be fetched in either direction, backward or forward.

A scrollable cursor allows you to set the position of the cursor anywhere in the cursor result set as long as the cursor is open, by specifying the option `first`, `last`, `absolute`, `next`, `prior`, or `relative` in a `fetch` statement.

To fetch the last row in a result set, enter:

```
fetch last [from] <cursor_name>
```

All scrollable cursors are read-only. Any cursor that can be updated is nonscrollable.

For detailed information on the global variables, commands, and functions that support cursors, see the *Reference Manual: Building Blocks* and the *Reference Manual: Commands*.

20.1 Types of Cursors

There are four types of cursors: client, execute, server, and language.

- Client cursors – declared through Open Client calls (or Embedded SQL). Open Client keeps track of the rows returned from and buffers them for the application. Updates and deletes to the result set of client cursors can be performed only through Open Client calls. Client cursors are the most frequently used type of cursors.
- Execute cursors – a subset of client cursors, for which the result set is defined by a stored procedure. The stored procedure can use parameters. Parameter values are sent through Open Client calls.
- Server cursors – declared in SQL. If server cursors are used in stored procedures, the client executing the stored procedure is unaware of them. Results returned to the client for a `fetch` are the same as the results from a normal `select`.
- Language cursors – declared in SQL without using Open Client. As with server cursors, the client is unaware of the cursors, and results are returned to the client in the same format as a normal `select`.

To simplify the discussion of cursors, the examples in this manual are only for language and server cursors. For examples of client or execute cursors, see your Open Client or Embedded SQL documentation.

20.2 Cursor Scope

A cursor's existence depends on its scope, which refers to the context in which the cursor is used: within a user session, within a stored procedure, or within a trigger.

Within a user session, the cursor exists only until a user ends the session. After the user logs off, SAP ASE deallocates the cursors created in that session. The cursor does not exist for additional sessions that other users start.

A cursor name must be unique within a given scope. SAP ASE detects name conflicts within a particular scope only during runtime. A stored procedure or trigger can define two cursors with the same name if only one is executed. For example, the following stored procedure works because only one `names_crsr` cursor is defined in its scope:

```
create procedure proc2 @flag int
as
if @flag > 0
    declare names_crsr cursor
    for select au_fname from authors
```

```
else
    declare names_crsr cursor
    for select au_lname from authors
return
```

20.3 Cursor Scans and the Cursor Result Set

The method SAP ASE uses to create the cursor result set depends on the cursor and on the query plan for the cursor `select` statement. If a worktable is not required, SAP ASE performs a `fetch` by positioning the cursor in the base table, using the table's index keys.

This executes similarly to a `select` statement, except that it returns the number of rows specified by the `fetch`. After the `fetch`, SAP ASE positions the cursor at the next valid index key, until you fetch again or close the cursor.

All scrollable cursors and `insensitive` nonscrollable cursors require worktables to hold cursor result sets. Some queries also require worktables to generate cursor result sets. To verify whether a particular cursor uses a worktable, check the output of a `set showplan, no exec on` statement.

When a worktable is used, the rows retrieved with a cursor `fetch` statement may not reflect the values in the actual base table rows. For example, a cursor declared with an `order by` clause usually requires the creation of a worktable to order the rows for the cursor result set. SAP ASE does not lock the rows in the base table that correspond to the rows in the worktable, which permits other clients to update these base table rows. The rows returned to the client from the cursor statement are different from the base table rows.

In general, the cursor result set for both default and `semi_sensitive` cursors is generated as the rows are returned through a `fetch` of that cursor. This means that a cursor `select` query is processed like a normal `select` query. This process, known as a cursor scan, provides a faster turnaround time and eliminates the need to read rows the application does not require.

SAP ASE requires cursor scans to use a unique index of a table, particularly for isolation-level 0 reads. If the table has an IDENTITY column and you must create a nonunique index on it, use the `identity in nonunique index` database option to include an IDENTITY column in the table's index keys so that all indexes created on the table are unique. This option makes logically nonunique indexes internally unique and allows them to process updatable cursors for isolation-level 0 reads.

You can still use cursors that reference tables without indexes, if none of those tables are updated by another process that causes the current row position to move. For example:

```
declare storinfo_crsr cursor
for select stor_id, stor_name, payterms
    from stores
    where state = "CA"
```

The table `stores`, specified with the above cursor, does not have any indexes. SAP ASE allows the declaration of cursors on tables without unique indexes, as long as you have not specified `for update` in the `declare cursor` statement. If an `update` does not change the position of the row, the cursor position does not change until the next `fetch`.

Related Information

Cursors and Locking [page 577]

20.4 Make Cursors Updatable

You can update or delete a row returned by a cursor if the cursor is updatable. If the cursor is read-only, you cannot update or delete it. By default, SAP ASE attempts to determine whether a cursor can be updated before designating it as read-only.

You can explicitly specify whether a cursor is read-only by using the `read only` or `update` keywords in the `declare` statement. Specifying a cursor as read-only ensures that SAP ASE correctly performs positioned updates. Make sure the table being updated has a unique index. If it does not, SAP ASE rejects the `declare cursor` statement.

All scrollable cursors and all `insensitive` cursors are read-only.

The following example defines an updatable result set for the `pubs_crsr` cursor:

```
declare pubs_crsr cursor
for select pub_name, city, state
from publishers
for update of city, state
```

The example includes all the rows from the `publishers` table, but it explicitly defines only the `city` and `state` columns as updatable.

Unless you plan to update or delete rows through a cursor, declare it as read-only. If you do not explicitly specify `read only` or `update`, the `semi_sensitive` nonscrollable cursor is implicitly updatable when the `select` statement does *not* contain any of the following constructs:

- `distinct` option
- `group by` clause
- Aggregate function
- `Subquery`
- `union` operator
- `at isolation read uncommitted` clause

You cannot specify the `for update` clause if a cursor's `select` statement contains one of these constructs. SAP ASE also defines a cursor as read-only if you declare certain types of cursors that include an `order by` clause as part of their `select` statement.

Related Information

select for update [page 233]
Types of Cursors [page 553]

20.4.1 Determine Which Columns Can Be Updated

Scrollable cursors and `insensitive` nonscrollable cursors are read-only. If you do not specify a `<column_name_list>` with the `for update` clause, all the specified columns in the query can be updated. SAP ASE attempts to use unique indexes for updatable cursors when scanning the base table.

For cursors, SAP ASE considers an index containing an IDENTITY column to be unique, even if it is not so declared.

SAP ASE allows you to update columns in the `<column_name_list>` that are not specified in the list of columns of the cursor's `select` statement, but that are part of the tables specified in the `select` statement. However, when you specify a `<column_name_list>` with `for update`, you can update only the columns in that list.

In the following example, SAP ASE uses the unique index on the `pub_id` column of `publishers` (even though `pub_id` is not included in the definition of `newpubs_crsr`):

```
declare newpubs_crsr cursor
for select pub_name, city, state
from publishers
for update
```

If you do not specify the `for update` clause, SAP ASE chooses any unique index, although it can also use other indexes or table scans if no unique index exists for the specified table columns. However, when you specify the `for update` clause, SAP ASE must use a unique index defined for one or more of the columns to scan the base table. If no unique index exists, SAP ASE returns an error message.

In most cases, include only columns to be updated in the `<column_name_list>` of the `for update` clause. If the cursor is declared with a `for update` clause, and the table has only one unique index, you cannot include its column in the `for update` `<column_name_list>`; SAP ASE uses it during the cursor scan. If the table has more than one unique index, you can include the index column in the `for update` `<column_name_list>`, so that SAP ASE can use another unique index, which may not be in the `<column_name_list>`, to perform the cursor scan. For example, the table used in the following `declare cursor` statement has one unique index, on the column `c3`, so that column should not be included in the `for update` list:

```
declare mycursor cursor
for select c1, c2, 3
from mytable
for update of c1, c2
```

However, if `mytable` has more than one unique index, for example, on columns `c3` and `c4`, you must specify one unique index in the `for update` clause as follows:

```
declare mycursor cursor
for select c1, c2, 3
from mytable
for update of c1, c2, c3
```

You cannot include both `c3` and `c4` in the `<column_name_list>`. In general, SAP ASE needs at least one unique index key, not on the list, to perform a cursor scan.

Allowing SAP ASE to use the unique index in the cursor scan in this manner helps prevent an update anomaly called the Halloween problem. The Halloween problem occurs when a client updates a column through a

cursor, and that column defines the order in which the rows are returned from the base tables (that is, a unique indexed column). For example, if SAP ASE accesses a base table using an index, and the index key is updated by the client, the updated index row can move within the index and be read again by the cursor. The row seems to appear twice in the result set: when the index key is updated by the client and when the updated index row moves farther down the result set.

Another way to avoid the Halloween problem is to create tables with the `unique auto_identity index` database option set to `on`. See, *Optimization for Cursors*, in the *Performance and Tuning Series: Query Processing and Abstract Plans*.

20.5 How SAP ASE Processes Cursors

When accessing data using cursors, SAP ASE divides the process into several operations.

- When you declare a cursor, SAP ASE creates a cursor structure. The server does not compile the cursor from the cursor declaration, however, until the cursor is open.
  The following cursor declaration of a default, nonscrollable cursor, `business_crsr`, finds the titles and identification numbers of all business books in the `titles` table.

  ```
  declare business_crsr cursor
  for select title, title_id
  from titles
  where type = "business"
  for update of price
  ```

  Using the `for update` clause when declaring a cursor ensures that SAP ASE correctly performs the positioned updates. In this example, it allows you to use the cursor to change the price.
  This example declares a scrollable cursor, `authors_scroll_crsr`, which finds authors from California in the `authors` table.

  ```
  declare authors_scroll_crsr scroll cursor
  for select au_fname, au_lname
  from authors
  where state = 'CA'
  ```

  Because scrollable cursors are read-only, you cannot use a `for update` clause in a cursor declaration.
- When you open a cursor that has been declared outside of a stored procedure, SAP ASE compiles the cursor and generates an optimized query plan. It then performs the preliminary operations for scanning the rows defined in the cursor and is ready to return a result row.
  When you declare a cursor within a stored procedure, SAP ASE compiles the cursor the first time the stored procedure is called. SAP ASE also generates an optimized query plan, and stores the plan for later use. When the stored procedure is called again, the cursor already exists in compiled form. When the cursor is opened, SAP ASE needs only to perform preliminary operations for executing a scan and returning a result set.

  > **Note**
  >
  > Since Transact-SQL statements are compiled during the open phase of a cursor, any error messages related to declaring the cursor appear during the cursor open phase.

- The `fetch` command executes the compiled cursor to return one or more rows meeting the conditions defined in the cursor. By default, a `fetch` returns only a single row.
  In nonscrollable cursors, the first `fetch` returns the first row that meets the cursor's search conditions, and stores the current position of the cursor. The second `fetch` uses the cursor position from the first `fetch`, returns the next row that meets the search conditions, and stores its current position. Each subsequent `fetch` uses the cursor position of the previous `fetch` to locate the next cursor row.
  In scrollable cursors, you can fetch any rows and set the current cursor position to any row in the result set, by specifying a `fetch` orientation in a `fetch` statement. The orientation options are `first`, `last`, `next`, `prior`, `absolute`, and `relative`. `fetch` for scrollable cursors executes in both forward and backward directions, and the result set can be scanned repeatedly.
  You can change the number of rows returned by a `fetch` by using `set cursor rows`.
  In the following example, the `fetch` command displays the title and identification number of the first row in the `titles` table containing a business book:

  ```
  fetch business_crsr
  ```

  ```
  title                                  title_id
  -----------------------------------    --------
  The Busy Executive's Database Guide    BU1032

  (1 row affected)
  ```

  Running `fetch business_crsr` a second time displays the title and identification number of the next business book in `titles`.
  In the following example, the first `fetch` command to a scrollable cursor displays the tenth row in the `authors` table, containing authors from California:

  ```
  fetch absolute 10 authors_scroll_crsr
  au_fname au_lname
  --------------------
  Akiko Yokomoto
  ```

  A second `fetch`, with the orientation option `prior`, returns the row before the tenth row:

  ```
  fetch prior authors_scroll_crsr
  au_fname au_lname
  ------------------
  Chastity Locksley
  ```

- SAP ASE updates or deletes the data in the cursor result set (and in the corresponding base tables that supplied the data) at the current cursor position. These operations are optional.
  The following `update` statement raises the price of business books by 5 percent; it affects only the book currently pointed to by the `business_crsr` cursor:

  ```
  update titles
  set price = price * .05 + price
  where current of business_crsr
  ```

  Updating a cursor row involves changing data in the row or deleting the row. You cannot use cursors to insert rows. All updates performed through a cursor affect the corresponding base tables included in the cursor result set.

- SAP ASE closes the cursor result set, removes any remaining temporary tables, and releases the server resources held for the cursor structure. However, it keeps the query plan for the cursor so that it can be opened again. Use the `close` command to close a cursor. For example:

  ```
  close business_crsr
  ```

  When you close a cursor and then reopen it, SAP ASE re-creates the cursor result, and positions the cursor before the first valid row. This allows you to process a cursor result set as many times as necessary. You can close the cursor at any time; you do not have to go through the entire result set.
- SAP ASE removes the query plan from memory and eliminates all trace of the cursor structure. To deallocate a cursor, use the `deallocate cursor` command. For example:

  ```
  deallocate cursor business_crsr
  ```

  In SAP ASE version 15.0 and later, The keyword `cursor` is optional for this command.
  You must declare the cursor again before using it.

Related Information

declare cursor [page 560]
Get Multiple Rows With Each Fetch [page 566]

20.6 Monitor Cursor Statements

SAP ASE uses the `monCachedStatement` table to monitor cursors.

The `StmtType` column in `monCachedStatement` indicates the types of queries in the statement cache. The values for `StmtType` are:

- 1 – batch statement
- 2 – cursor statement
- 3 – dynamic statement

> **Note**
>
> You must set the `enable functionality group` configuration parameter to 1 to monitor cursor statements.

This example shows the contents of `monCachedStatement` (including the SSQLID for `new_cursor`), and then uses the `show_cached_text` function to show the SQL text for `new_cursor`:

```
select InstanceID, SSQLID, Hashkey, UseCount, StmtType
from monCachedStatement
```

```
InstanceID   SSQLID       Hashkey       UseCount      StmtType ----------
  -----------  -----------   -----------  ----------
           0    329111220   1108036110             0           2
```

```
          0    345111277   1663781964             1           1
```

```
select show_cached_text(329111220)
--------------------------------------
select id from sysroles
```

See, *Monitoring Tables*, in the *Reference Manual: Tables* and the *Performance and Tuning Series: Monitoring Tables*.

20.7 declare cursor

The `declare cursor` statement must precede any `open` statement for that cursor. You cannot combine `declare cursor` with other statements in the same Transact-SQL batch, except when using a cursor in a stored procedure.

The `<select_statement>` is the query that defines the cursor result set. In general, `<select_statement>` can use nearly the full syntax and semantics of a Transact-SQL `select` statement, including the `holdlock` keyword. However, it cannot contain a `compute`, `for browse`, or `into` clause.

Examples of the declare cursor Command

The following `declare cursor` statement defines a result set for the `authors_crsr` cursor that contains all authors who do not reside in California:

```
declare authors_crsr cursor
for select au_id, au_lname, au_fname
from authors
where state != 'CA'
for update
```

The following example defines an `insensitive` scrollable result set, of the `stores_scrollcrsr`, containing bookstores in California, for:

```
declare storinfo_crsr insensitive scroll cursor
for select stor_id, stor_name, payterms
from stores
where state = "CA"
```

To declare an `insensitive`, nonscrollable cursor called “C1,” enter:

```
declare C1 insensitive cursor for
select fname from emp_tab
```

To declare an `insensitive`, scrollable cursor called “C3,”enter:

```
declare C3 insensitive scroll cursor for
select fname from emp_tab
```

To fetch the first row, enter:

```
fetch first from <cursor_name>
```

You can also fetch the columns of the first row from the result set. To place them in the variables you specify in `<fetch_target_list>`, enter:

```
fetch first from <cursor_name> into <fetch_target_list>
```

You can fetch the 20th row in the result set directly, regardless of the cursor's current position:

```
fetch absolute 20 from <cursor_name> into <fetch_target_list>
```

20.7.1 cursor_scrollability

You can use either `scroll` or `no scroll` to specify `cursor_scrollability`. If the cursor is scrollable, you can scroll through the cursor result set by fetching any, or many rows back and forth; you can also repeatedly scan the result set.

All scrollable cursors are read-only, and cannot be used with `for update` in a cursor declaration.

20.7.2 Cursor Sensitivity

You can use either `insensitive` or `semi_sensitive` to explicitly specify cursor sensitivity.

An `insensitive` cursor is a snapshot of the result set, taken when the cursor is opened. An internal worktable is created and fully populated with the cursor result set when you open the cursor.

Any locks on the base tables are released, and only the worktable is accessed when you execute `fetch`. Any data changes in the base table on which the cursor is declared do not affect the cursor result set. The cursor is read-only, and cannot be used with `for update`.

In a `semi_sensitive` cursor, some data changes in the base tables may appear in the cursor. The query plan chosen and whether the data rows have been fetched at least once may affect the visibility of the base table data change.

`semi_sensitive` scrollable cursors are like `insensitive` cursors, in that they use a worktable to hold the result set for scrolling purposes. In `semi_sensitive` mode, the cursor's worktable materializes as the rows are fetched, rather than when you open the cursor. The membership of the result set is fixed only after all the rows have been fetched once, and copied to the scrolling worktable.

If you do not specify cursor sensitivity, the default value is `semi_sensitive`.

Even if you declare a cursor `semi_sensitive`, the visibility of data changes in the base table of the cursor depends on the query plan chosen by the optimizer.

Any `sort` command forces the cursor to become `insensitive`, even if you have declared it `semi_sensitive`, because it requires the rows in a table to be ordered before `sort` can be executed. A worktable, however, can be populated before any rows can be fetched.

For example, if a select statement contains an order by clause, and there is no index on the order by column, the worktable is fully populated when the cursor is opened, whether or not you declare the cursor to be semi_sensitive. The cursor becomes insensitive.

Generally, rows that have not yet been fetched can display data changes, while rows that have already been fetched do not.

The main benefit of using a semi_sensitive scrollable cursor instead of an insensitive scrollable cursor is that the first row of the result set is returned promptly to the user, since the table lock is applied row by row. If you fetch a row and update it, it becomes part of the worktable through fetch, and the update is executed on the base table. There is no need to wait for the result set worktable to be fully populated.

20.7.3 read_only Option

The read_only option specifies that the cursor result set cannot be updated. In contrast, the for update option specifies that the cursor result set is updatable. You can specify of <column_name_list> after for update with the list of columns from the <select_statement> that is defined as updatable.

20.8 Open Cursors

After you declare a cursor, you must open it to fetch, update, or delete rows. Opening a cursor lets SAP ASE begin to create the cursor result set by evaluating the select statement that defines the cursor and makes it available for processing.

```
open <cursor_name>
```

You cannot open a cursor that is already open or that has not been defined with the declare cursor statement. You can reopen a closed cursor to reset the cursor position to the beginning of the cursor result set.

Depending on the cursor's type and the query plan, a worktable may be created and populated when you open the cursor.

20.9 Fetch Data Rows Using Cursors

A fetch completes the cursor result set and returns one or more rows to the client. Depending on the type of query defined in the cursor, SAP ASE creates the cursor result set either by scanning the tables directly or by scanning a worktable generated by the query type and cursor type.

The fetch command positions a nonscrollable cursor before the first row of the cursor result set. If the table has a valid index, SAP ASE positions the cursor at the first index key.

20.9.1 fetch Syntax

first, next, prior, last, absolute, and relative specify the fetch direction of the scrollable cursor. If no keyword is specified, the default value is next.

See the *Reference Manual: Commands*.

If you use fetch absolute or fetch relative, specify <fetch_offset>. The value can be a literal of an integer, an exact, signed numeric with a scale of 0, or a Transact-SQL local variable with an integer or numeric datatype with a scale of 0. When the cursor is positioned beyond the last row or before the first row, no data is returned and no error is raised.

When you use fetch absolute and <fetch_offset> is greater than or equal to 0, the offset is calculated from the position before the first row of the result set. If fetch absolute is less than 0, the offset is calculated from the position after the last row of the result set.

If you use fetch relative when <fetch_offset> n is greater than 0, the cursor is placed <n> rows after the current position; if <fetch_offset n>0>, the cursor is placed <abs(n)> rows before the current position.

For example, with the scrollable cursor <stores_scrollcrsr>, you can fetch any row you want. This fetch positions the cursor on the third row in the result set:

```
fetch absolute 3 stores_scrollcrsr
stor_id stor_name
--------------------------------------
7896 Fricative Bookshop
```

A subsequent fetch prior operation positions the cursor on the second row of the result set:

```
fetch prior stores_scrollcrsr
stor_id stor_name
--------------------------------------
7067 News & Brews
```

A subsequent fetch relative -1 positions the cursor on the first row of the result set:

```
fetch relative -1 stores_scrollcrsr
stor_id stor_name
-------------------------
7066     Barnum's
```

After generating the cursor result set, in a fetch statement for a nonscrollable cursor, SAP ASE moves the cursor position one row in the result set. It retrieves the data from the result set and stores the current position, allowing additional fetches until SAP ASE reaches the end of the result set.

The next example illustrates a nonscrollable cursor. After declaring and opening the authors_crsr cursor, you can fetch the first row of its result set as follows:

```
fetch authors_crsr
```

```
au_id        au_lname            au_fname
-----------  -------------------- ---------------
341-22-1782  Smith               Meander

(1 row affected)
```

Each subsequent `fetch` retrieves the next row from the cursor result set. For example:

```
fetch authors_crsr
```

```
au_id       au_lname            au_fname
----------- ------------------- ---------------
527-72-3246 Greene               Morningstar

(1 row affected)
```

After you `fetch` all the rows, the cursor points to the last row of the result set. If you `fetch` again, SAP ASE returns a warning through the `<@@sqlstatus>` or `<@@fetch_status>` global variables, indicating that there is no more data. The cursor position remains unchanged.

If you are using nonscrollable cursors, you cannot fetch a row that has already been fetched. Close and reopen the cursor to generate the cursor result set again, and start fetching again from the beginning.

Related Information

Check Cursor Status [page 564]

20.9.2 into Clause Usage

The `into` clause specifies that SAP ASE returns column data into the specified variables.

The `<fetch_target_list>` must consist of previously declared Transact-SQL parameters or local variables.

For example, after declaring the `<@name>`, `<@city>`, and `<@state>` variables, you can fetch rows from the `pubs_crsr` cursor as follows:

```
fetch pubs_crsr into @name, @city, @state
```

You can also fetch only the columns of the first row from the result set. To place the fetch columns in a list, enter:

```
fetch first from <cursor_name> into <fetch_target_list>
```

20.9.3 Check Cursor Status

SAP ASE returns a status value after each fetch. You can access the value through the global variables `<@@sqlstatus>`, `<@@fetch_status>`, or `<@@cursor_rows>`. `<@@fetch_status>` and `<@@cursor_rows>` are supported only in SAP ASE version 15.0 and later.

This table lists `<@@sqlstatus>` values and their meanings:

Value	Meaning
0	Successful completion of the `fetch` statement.
1	The `fetch` statement resulted in an error.
2	There is no more data in the result set. This warning can occur if the current cursor position is on the last row in the result set and the client submits a `fetch` statement for that cursor.

This table lists `<@@fetch_status>` values and meanings:

Value	Meaning
0	`fetch` operation successful.
-1	`fetch` operation unsuccessful.
-2	Value reserved for future use.

The following example determines the `<@@sqlstatus>` for the currently open `authors_crsr` cursor:

```
select @@sqlstatus
```

```
---------
        0

(1 row affected)
```

The following example determines the `<@@fetch_status>` for the currently open `authors_crsr` cursor:

```
select @@fetch_status
```

```
---------
        0

(1 row affected)
```

Only a `fetch` statement can set `<@@sqlstatus>` and `<@@fetch_status>`. Other statements have no effect on `<@@sqlstatus>`.

`<@@cursor_rows>` indicates the number of rows in the cursor result set that were last opened and fetched.

Value	Meaning
-1	Indicates one of the following: • The cursor is dynamic; since a dynamic cursor reflects all changes, the number of rows that qualify for the cursor is constantly changing. You can never definitively state that all qualified rows are retrieved. • The cursor is semisensitive and scrollable, but the scrolling worktable is not yet populated. The number of rows that qualify for the result set is unknown.

Value	Meaning
0	No cursors have been opened, no rows are qualified from the last opened cursor, or the last opened cursor is closed or deallocated.
n	The last opened or fetched cursor result set has been fully populated; the value returned (n) is the total number of rows in the cursor result set.

20.9.4 Get Multiple Rows With Each Fetch

You can use the `set cursor rows` command to change the number of rows that are returned by `fetch`. However, this option does not affect a `fetch` containing an `into` clause.

The syntax for `set cursor rows` is:

```
set cursor rows <number> for <cursor_name>
```

`<number>` specifies the number of rows for the cursor. The `<number>` can be a numeric literal with no decimal point, or a local variable of type `integer`. The default setting is 1 for each cursor you declare. You can `set` the `cursor rows` option for any cursor, whether it is open or closed.

For example, you can change the number of rows fetched for the `authors_crsr` cursor:

```
set cursor rows 3 for authors_crsr
```

After you set the number of cursor rows, each `fetch` of `authors_crsr` returns three rows:

```
fetch authors_crsr
```

```
au_id       au_lname            au_fname
----------- ------------------- ---------------
648-92-1872 Blotchet-Halls      Reginald
712-45-1867 del Castillo        Innes
722-51-5424 DeFrance            Michel

(3 rows affected)
```

The cursor is positioned on the last row fetched (the author Michel DeFrance in the example).

Fetching several rows at a time works especially well for client applications. If you fetch more than one row, Open Client or Embedded SQL buffers the rows sent to the client application. The client still sees row-by-row access, but each `fetch` results in fewer calls to SAP ASE, which improves performance.

20.9.5 Check the Number of Rows Fetched

Use the <@@rowcount> global variable to monitor the number of rows of the cursor result set returned to the client up to the last fetch. This variable displays the total number of rows seen by the cursor at any one time.

In a nonscrollable cursor, once all the rows are read from a cursor result set, <@@rowcount> represents the total number of rows in that result set. The total number of rows represents the maximum value of <@@cursor_rows> in the last fetched cursor.

The following example determines the <@@rowcount> for the currently open authors_crsr cursor:

```
select @@rowcount
----------
```

```
        5

(1 row affected)
```

In a scrollable cursor, there is no maximum value for <@@rowcount>.The value continues to increment with each fetch operation, regardless of the direction of the fetch.

The following example shows the <@@rowcount> value for authors_scrollcrsr, a scrollable, insensitive cursor. Assume there are five rows in the result set. After the cursor is open, the initial value of <@@rowcount> is 0: all rows of the result set are fetched from the base table and saved to the worktable. All the rows in the following fetch example are accessed from the worktable.

```
fetch last authors_scrollcrsr   @@rowcount = 1
```

```
fetch first authors_scrollcrsr  @@rowcount = 2
```

```
fetch next authors_scrollcrsr   @@rowcount = 3
```

```
fetch relative 2 authors_scrollcrsr @@rowcount = 4
```

```
fetch absolute 3 authors_scrollcrs  @@rowcount = 5
```

```
fetch absolute -2 authors_scrollcrsr  @@rowcount = 6
```

```
fetch first authors_scrollcrsr  @@rowcount = 7
```

```
fetch absolute 0 authors_scrollcrsr  @@rowcount =7 (nodatareturned)
```

```
fetch absolute 2 authors_scrollcrsr  @@rowcount = 8
```

20.10 Update and Delete Rows Using Cursors

If the cursor is updatable, use the `update` or `delete` statement to update or delete rows.

SAP ASE determines whether the cursor is updatable by checking the `select` statement that defines the cursor. You can also explicitly define a cursor as updatable, using the `for update` clause of the `declare cursor` statement.

Related Information

Make Cursors Updatable [page 555]

20.10.1 Update Cursor Result Set Rows

You can use the `where current of` clause of the `update` statement to update the row at the current cursor position. Any update to the cursor result set also affects the base table row from which the cursor row is derived.

The syntax for `update...where current of` is:

```
update [[<database>.]<owner>.] {<table_name> | <view_name>}
     set [[[<database>.]<owner>.] {<table_name>. | <view_name>.}]
          <column_name1>  =
               {<expression1> | NULL | (<select_statement>)}
          [, <column_name2>  =
               {<expression2> | NULL | (<select_statement>)}]...
     where current of <cursor_name>
```

The `set` clause specifies the cursor's result set column name and assigns the new value. When more than one column name and value pair is listed, separate them with commas.

The `<table_name>` or `<view_name>` must be the table or view specified in the first `from` clause of the `select` statement that defines the cursor. If that `from` clause references more than one table or view (using a join), you can specify only the table or view actually being updated.

For example, you can update the row that the `pubs_crsr` cursor currently points to:

```
update publishers
set city = "Pasadena",
    state = "CA"
where current of pubs_crsr
```

After the update, the cursor position remains unchanged. You can continue to update the row at that cursor position, as long as another SQL statement does not move the position of the cursor.

SAP ASE allows you to update columns that are not specified in the list of columns of the cursor's `<select_statement>`, but are part of the tables specified in that statement. However, when you specify a `<column_name_list>` with `for update`, you can update only the columns in that list.

20.10.2 Delete Cursor Result Set Rows

Using the `where current of` clause of the `delete` statement, you can delete the row at the current cursor position.

When you delete a row from the cursor's result set, the row is deleted from the underlying database table. You can delete only one row at a time using the cursor.

The syntax for `delete...where current of` is:

```
delete [from]
     [[<database>.]<owner>.] {<table_name >| <view_name>}
     where current of <cursor_name>
```

The `<table_name>` or `<view_name>` must be the table or view specified in the first `from` clause of the `select` statement that defines the cursor.

For example, delete the row that the `authors_crsr` cursor currently points to by entering:

```
delete from authors
where current of authors_crsr
```

The `from` keyword is optional.

> **Note**
>
> You cannot delete a row from a cursor defined by a `select` statement containing a join, even if the cursor is updatable.

After you delete a row from a cursor, SAP ASE positions the cursor before the row following the deleted row in the cursor result set. You must still use `fetch` to access that row. If the deleted row is the last row in the cursor result set, SAP ASE positions the cursor after the last row of the result set.

For example, after deleting the current row in the example (the author Michel DeFrance), you can fetch the next three authors in the cursor result set (assuming that `cursor rows` is still set to 3):

```
fetch authors_crsr
```

```
au_id        au_lname            au_fname
-----------  ------------------- ---------------
807-91-6654  Panteley            Sylvia
899-46-2035  Ringer              Anne
998-72-3567  Ringer              Albert

(3 rows affected)
```

You can delete a row from the base table without referring to a cursor. The cursor result set changes as changes are made to the base table.

20.11 Close and Deallocate Cursors

When you are finished with the result set of a cursor, you can `close` it.

The syntax is:

```
close <cursor_name>
```

Closing a cursor does not change its definition. If you reopen a cursor, SAP ASE creates a new cursor result set using the same query as before. For example:

```
close authors_crsr
open authors_crsr
```

You can then fetch from `authors_crsr`, starting from the beginning of its cursor result set. Any conditions associated with that cursor (such as the number of rows fetched, defined by `set cursor rows`) remain in effect.

To discard a cursor, deallocate it using:

```
deallocate cursor <cursor_name>
```

> **Note**
>
> In SAP ASE 15.0 and later, the word `cursor` is optional.

Deallocating a cursor frees any resources associated with it, including the cursor name. You cannot reuse a cursor name until you deallocate it. If you deallocate an open cursor, SAP ASE automatically closes it. Terminating a client connection to a server also closes and deallocates any open cursors.

20.12 Cursor Examples

Examples of scrollable and forward-only cursors are provided.

Forward-Only (Default) Cursors

```
select author = au_fname + " " + au_lname, au_id
from authors
```

The results of the query are:

```
author                      au_id
-------------------------   -----------
Johnson White               172-32-1176
Marjorie Green              213-46-8915
Cheryl Carson               238-95-7766
```

```
Michael O'Leary             267-41-2394
Dick Straight               274-80-9391
Meander Smith               341-22-1782
Abraham Bennet              409-56-7008
Ann Dull                    427-17-2319
Burt Gringlesby             472-27-2349
Chastity Locksley           486-29-1786
Morningstar Greene          527-72-3246
Reginald Blotchet Halls     648-92-1872
Akiko Yokomoto              672-71-3249
Innes del Castillo          712-45-1867
Michel DeFrance             722-51-5454
Dirk Stringer               724-08-9931
Stearns MacFeather          724-80-9391
Livia Karsen                756-30-7391
Sylvia Panteley             807-91-6654
Sheryl Hunter               846-92-7186
Heather McBadden            893-72-1158
Anne Ringer                 899-46-2035
Albert Ringer               998-72-3567

(23 rows affected)
```

To use a cursor with the query above:

1. Declare the cursor.
   This `declare cursor` statement defines a cursor using the `select` statement shown above:

   ```
   declare newauthors_crsr cursor for
   select author = au_fname + " " + au_lname, au_id
   from authors
   for update
   ```

2. Open the cursor:

   ```
   open newauthors_crsr
   ```

3. Fetch rows using the cursor:

   ```
   fetch newauthors_crsr
   ```

   ```
   author                      au_id
   -------------------------   -----------
   Johnson White                 172-32-1176

   (1 row affected)
   ```

   You can fetch more than one row at a time by specifying the number of rows with the `set cursor rows` command:

   ```
   set cursor rows 5 for newauthors_crsr
   go
   fetch newauthors_crsr
   ```

   ```
   author                      au_id
   -------------------------   -----------
   Marjorie Green              213-46-8915
   Cheryl Carson               238-95-7766
   Michael O'Leary             267-41-2394
   Dick Straight               274-80-9391
   Meander Smith               341-22-1782

   (5 rows affected)
   ```

Each subsequent `fetch` returns the next five rows:

```
fetch newauthors_crsr
```

```
author                       au_id
-------------------------    -----------
Abraham Bennet               409-56-7008
Ann Dull                     427-17-2319
Burt Gringlesby              472-27-2349
Chastity Locksley            486-29-1786
Morningstar Greene           527-72-3246

(5 rows affected)
```

The cursor is now positioned at author Morningstar Greene, the last row of the current `fetch`.

4. To change the first name of Greene, enter:

```
update authors
set au_fname = "Voilet"
where current of newauthors_crsr
```

The cursor remains at Ms. Greene's record until the next `fetch`.

5. When you are finished with the cursor, close it:

```
close newauthors_crsr
```

If you `open` the cursor again, SAP ASE re-runs the query and places the cursor before the first row in the result set. The cursor is still set to return five rows with each `fetch`.

6. To remove the cursor, use:

```
deallocate cursor newauthors_crsr
```

You cannot reuse a cursor name until you deallocate it.

Insensitive Scrollable Cursors

When you declare and open an insensitive cursor, a worktable is created and fully populated with the cursor result set. Locks on the base table are released, and only the worktable is used for fetching.

To declare cursor CI as an insensitive cursor, enter:

```
declare CI insensitive scroll cursor for
select emp_id, fname, lname
from emp_tb
where emp_id > 2002000
open CI
```

To change the name "Sam" to "Joe," enter:

```
.....
update emp_tab set fname = "Joe"
where fname = "Sam"
```

Now four “Sam” rows in the base table `emp_tab` disappear, replaced by four “Joe” rows.

```
fetch absolute 2 CI
```

The cursor reads the second row from the cursor result set, and returns Row 2, “2002020, Sam, Clarac.”

This next command inserts one more qualified row (that is, a row that meets the query condition in `declare cursor`) into table `emp_tab`, but the row membership is fixed in a cursor, so the added row is invisible to cursor CI. Enter:

```
insert into emp_tab values (2002101, "Sophie", "Chen", .., ..., ...)
```

The following `fetch` command scrolls the cursor to the end of the worktable, and reads the last row in the result set, returning the row value “2002100, Sam, West.” Again, because the cursor is `insensitive`, the new row inserted in `emp_tab` is invisible in cursor CI’s result set.

```
fetch last CI
```

Semisensitive Scrollable Cursors

Semisensitive scrollable cursors are like insensitive cursors in that they use a worktable to hold the result set for scrolling purposes.

But in `semi_sensitive` mode, the cursor’s worktable materializes as the rows are fetched, rather than when you open the cursor. The membership of the result set is fixed only after all the rows have been fetched once.

To declare cursor CSI semisensitive and scrollable, enter:

```
declare CSI semi_sensitive scroll cursor for
select emp_id, fname, lname
from emp_tab
where emp_id > 2002000
open CSI
```

Because the cursor is semisensitive, none of the rows are copied to the worktable when you open the cursor. To fetch the first record, enter:

```
fetch first CSI
```

The cursor reads the first row from `emp_tab` and returns 2002010, Mari, Cazalis. This row is copied to the worktable. Fetch the next row by entering:

```
fetch next CSI
```

The cursor reads the second row from `emp_tab` and returns 2002020, Sam, Clarac. This row is copied to the worktable. To replace the name “Sam” with the name “Joe,” enter:

```
......
update emp_tab set fname = "Joe"
where fname = "Sam"
```

The four “Sam” rows in the base table `emp_tab` disappear, and four “Joe” rows appear instead. To fetch only the second row, enter:

```
fetch absolute 2 CSI
```

The cursor reads the second row from the result set and returns employee ID 2002020, but the value of the returned row is “Sam,” not “Joe.” Because the cursor is semisensitive, this row was copied into the worktable before the row was updated, and the data change made by the `update` statement is invisible to the cursor, since the row returned comes from the result set scrolling worktable.

To fetch the fourth row, enter:

```
fetch absolute 4 CSI
```

The cursor reads the fourth row from the result set. Since Row 4, (2002040, Sam, Burke) was fetched after “Sam” was updated to “Joe,” the returned employee ID 2002040 is Joe, Burke. The third and fourth rows are now copied to the worktable.

To add a new row, enter:

```
insert into emp_tab values (2002101, "Sophie", "Chen", .., ..., ...)
```

One more qualified row is added in the result set. This row is visible in the following `fetch` statement, because the cursor is semisensitive and because we have not yet fetched the last row. Fetch the updated version by entering:

```
fetch last CSI
```

The `fetch` statement reads 2002101, Sophie, Chen in the result set.

After using fetch with the `last` option, you have copied all the qualified rows of the cursor CSI to the worktable. Locking on the base table, `emp_tab`, is released, and the result set of cursor CSI is fixed. Any further data changes in `emp_tab` do not affect the result set of CSI.

> **Note**
>
> Locking schema and transaction isolation level also affect cursor visibility. The above example is based on the default isolation level, level 1.

Table for Scrollable Cursors

```
select emp_id, fname, lname
from emp_tab
where emp_id > 2002000
```

The base table, `emp_tab`, is a datarows-locking table with a clustered index on the `<emp_id>` field. “Row position” is an imaginary column, in which the values represent the position of each row in the result set. The result set in this table is used in the examples in the following sections, which illustrate both insensitive and semisensitive cursors.

Row Position	emp_id	fname	lname
1	2002010	Mari	Cazalis
2	2002020	Sam	Clarac
3	2002030	Bill	Darby
4	2002040	Sam	Burke
5	2002050	Mary	Armand
6	2002060	Mickey	Phelan
7	2002070	Sam	Fife
8	2002080	Wanda	Wolfe
9	2002090	Nina	Howe
10	2002100	Sam	West

20.13 Cursors in Stored Procedures

Cursors are particularly useful in stored procedures. They allow you to use only one query to accomplish a task that would otherwise require several queries. However, all cursor operations must execute within a single procedure.

A stored procedure cannot open, fetch, or close a cursor that was not declared in the procedure. Cursors are undefined outside the scope of the stored procedure.

For example, the stored procedure `au_sales` checks the `sales` table to see if any books by a particular author have sold well. It uses a cursor to examine each row, and then prints the information. If you did not use a cursor, you would need several `select` statements to accomplish the same task. Outside stored procedures, you cannot include other statements with `declare cursor` in the same batch.

```
create procedure au_sales (@author_id id)
as

/* declare local variables used for fetch */
declare @title_id tid
declare @title varchar(80)
declare @ytd_sales int
declare @msg varchar(120)

/* declare the cursor to get each book written
    by given author */
declare author_sales cursor for
select ta.title_id, t.title, t.total_sales
from titleauthor ta, titles t
where ta.title_id = t.title_id
and ta.au_id = @author_id
```

```
open author_sales
fetch author_sales
        into @title_id, @title, @ytd_sales
if (@@sqlstatus = 2)
begin
    print "We do not sell books by this author."
    close author_sales
    return
end

/* if cursor result set is not empty, then process
    each row of information */
while (@@sqlstatus = 0)
begin
    if (@ytd_sales = NULL)
    begin
        select @msg = @title +
           " -- Had no sales this year."
        print @msg
    end
    else if (@ytd_sales < 500)
    begin
        select @msg = @title +
            " -- Had poor sales this year."
        print @msg
    end
    else if (@ytd_sales < 1000)
    begin
        select @msg = @title +
            " -- Had mediocre sales this year."
        print @msg
    end
    else
    begin
        select @msg = @title +
            " -- Had good sales this year."
        print @msg
    end

    fetch author_sales into @title_id, @title,
    @ytd_sales
end
```

For example:

```
au_sales "172-32-1176"
```

```
Prolonged Data Deprivation: Four Case Studies -- Had good sales this year.

(return status = 0)
```

Related Information

Cursor Scope [page 553]
Stored Procedures [page 498]

20.14 Cursors and Locking

Cursor-locking methods are similar to other locking methods for SAP ASE. In general, statements that read data (such as `select` or `readtext`) use shared locks on each data page to avoid reading changed data from an uncommitted transaction. Update statements use exclusive locks on each page they change.

If you run `select for update` within a cursor context, the cursor `open` and `fetch` statements must be within the context of a transaction.

To reduce deadlocks and improve concurrency, SAP ASE often precedes an exclusive lock with an update lock, which indicates that the client intends to change data on the page.

For updatable cursors, SAP ASE uses update locks by default when scanning tables or views referenced with the `for update` clause of `declare cursor`. If the `for update` clause is included, but the list is empty, all tables and views referenced in the `from` clause of the `<select_statement>` receive update locks by default. If the `for update` clause is not included, the referenced tables and views receive shared locks. You can use shared locks instead of update locks by adding the `shared` keyword to the `from` clause after each table name for which you prefer a shared lock.

In insensitive cursors, the base table lock is released after the worktable is fully populated. In semisensitive scrollable cursors, the base table lock is released after the last row of the result set has been fetched once.

> **Note**
>
> SAP ASE releases an update lock when the cursor position moves off the data page. Since an application buffers rows for client cursors, the corresponding server cursor may be positioned on a different data row and page than the client cursor. In this case, a second client could update the row that represents the current cursor position of the first client, even if the first client used the `for update` option.

For more information on `select for update`, see the *Reference Manual:Commands*.

Any exclusive locks acquired by a cursor in a transaction are held until the end of that transaction. This also applies to shared or update locks when you use the `holdlock` keyword or the `set isolation level 3` option. However, if you do not set the `close on endtran` option, the cursor remains open past the end of the transaction, and its current page lock remains in effect. It can also continue to acquire locks as it fetches additional rows.

See the *Performance and Tuning Series: Locking and Concurrency Control*.

Related Information

select for update [page 233]

20.14.1 Cursor-Locking Options

Specifying the `holdlock` or `shared` options (of the `select` statement) when you define an updatable cursor have certain effects.

- If you omit both options, you can read data only on the currently fetched pages. Other users cannot update your currently fetched pages, through a cursor or otherwise. Other users can declare a cursor on the same tables you use for your cursor, but they cannot get an update lock on your currently fetched pages.
- If you specify the `shared` option, you can read data on the currently fetched pages only. Other users cannot update your currently fetched pages, through a cursor or otherwise.
- If you specify the `holdlock` option, you can read data on all pages fetched (in a current transaction) or only the pages currently fetched (if not in a transaction). Other users cannot update your currently fetched pages or pages fetched in your current transaction, through a cursor or otherwise. Other users can declare a cursor on the same tables you use for your cursor, but they cannot get an update lock on your currently fetched pages or the pages fetched in your current transaction.
- If you specify both options, you can read data on all pages fetched (in a current transaction) or only the pages currently fetched (if not in a transaction). Other users cannot update your currently fetched pages, through a cursor or otherwise.

20.15 Transaction Support for Updatable Cursors

When `select for update` is set, SAP ASE supports `fetch` operations on open cursors after the transaction has been committed.

When you open a cursor, SAP ASE uses different locking mechanisms based on the transaction mode:

- Chained mode – SAP ASE implicitly starts a transaction and uses exclusive locks for fetched rows. If you `commit` a transaction after a `fetch`, a subsequent `fetch` command starts a new transaction. SAP ASE continues to use exclusive locks for fetched rows in the new transaction.
- Unchained mode – SAP ASE uses exclusive locks only if you execute an explicit `begin tran` statement before opening the cursor. Otherwise, SAP ASE acquires update row locks on fetched rows and displays a warning that exclusive locks are not acquired for subsequently fetched rows.

When you execute a `commit` between two `fetch` commands, or between closing and reopening a cursor, SAP ASE releases all exclusive locks. For subsequent `fetch` commands, SAP ASE acquires locks based on the transaction mode:

- Chained mode – SAP ASE acquires exclusive row locks on fetched rows. SAP ASE may also acquire update row locks in certain non-optimized conditions.
- Unchained mode – SAP ASE acquires update row locks on fetched rows. If the `fetch` commands are preceded by `begin tran`, SAP ASE acquires exclusive row locks.

If update row locks are acquired, SAP ASE releases them only when:

- The cursor is closed – at isolation levels 2 and 3.
- The cursor moves to the next row – at isolation level 1.

Related Information

Use select for update in Cursors and DML [page 234]

20.16 Get Information About Cursors

Use `sp_cursorinfo` to find information about a cursor's name, its current status, and its result columns.

This example displays information about `authors_crsr`:

```
sp_cursorinfo 0, authors_crsr
```

```
Cursor name 'authors_crsr' is declared at nesting level '0'.
The cursor is declared as NON-SCROLLABLE cursor.
The cursor id is 851969.
The cursor has been successfully opened 1 times.
The cursor was compiled at isolation level 1.
The cursor is currently scanning at a nonzero isolation level.
The cursor is positioned on a row.
There have been 4 rows read, 0 rows updated and 0 rows deleted through this
cursor.
The cursor will remain open when a transaction is committed or rolled back.
The number of rows returned for each FETCH is 1.
The cursor is updatable.
This cursor is using 3432 bytes of memory.
There are 3 columns returned by this cursor.
The result columns are:
Name = 'au_id', Table = 'authors', Type = VARCHAR, Length = 11 (updatable)
Name = 'au_lname', Table = 'authors', Type = VARCHAR, Length = 40 (updatable)
Name = 'au_fname', Table = 'authors', Type = VARCHAR, Length = 20 (updatable)
Showplan output for the cursor:
QUERY PLAN FOR STATEMENT 1 (at line 1).
Optimized using Serial Mode
       STEP 1
           The type of query is DECLARE CURSOR.
           1 operator(s) under root
         |ROOT:EMIT Operator (VA = 1)
         |
         |  |SCAN Operator (VA = 0)
         |  |  FROM TABLE
         |  |  authors
         |  |  Using Clustered Index.
         |  |  Index : auidind
         |  |  Forward Scan.
         |  |  Positioning at start of table.
         |  |  Using I/O Size 2 Kbytes for data pages.
         |  |  With LRU Buffer Replacement Strategy for data pages.
```

This example displays information about scrollable cursors:

```
sp_cursorinfo 0, authors_scrollcrsr
Cursor name 'authors_scrollcrsr' is declared at nesting level '0'.
The cursor is declared as SEMI_SENSITIVE SCROLLABLE cursor.
The cursor id is 786434.
The cursor has been successfully opened 1 times.
The cursor was compiled at isolation level 1.
The cursor is currently scanning at a nonzero isolation level.
The cursor is positioned on a row.
```

```
There have been 1 rows read, 0 rows updated and 0 rows deleted through this
cursor.
The cursor will remain open when a transaction is committed or rolled back.
The number of rows returned for each FETCH is 1.
The cursor is read only.
This cursor is using 19892 bytes of memory.
There are 2 columns returned by this cursor.
The result columns are:
Name = 'au_fname', Table = 'authors', Type =VARCHAR, Length = 20 (not updatable)
Name = 'au_lname', Table = 'authors', Type = VARCHAR, Length = 40 (not updatable)
```

You can also check the status of a cursor using the `<@@sqlstatus>`, `<@@fetch_status>`, `<@@cursor_rows>`, and `<@@rowcount>` global variables.

See the *Reference Manual: Procedures*.

Related Information

Check Cursor Status [page 564]
Check the Number of Rows Fetched [page 567]

20.17 Browse Mode Versus Cursors

Browse mode lets you search through a table and update its values one row at a time. It is used in front-end applications that use DB-Library and a host programming language.

Browse mode provides compatibility with Open Server applications and older Open Client libraries. However, because cursors provide the same functionality in a more portable and flexible manner, SAP discourages browse mode use in more recent Client-Library applications (versions 10.0.x and later). Additionally, browse mode is SAP-specific, and therefore not suited to heterogeneous environments.

Normally, use cursors to update data when you want to change table values row by row. Client-Library applications can use Client-Library cursors to implement some browse-mode features, such as updating a table while fetching rows from it. However, cursors may cause locking contention in the tables being selected.

For more information on browse mode, see the `dbqual` function in the Open Client/Server documentation.

Browse-Mode Restrictions

You cannot use the `for browse` clause in statements that use the `union` operator, or in cursor declarations.

You cannot use the keyword `holdlock` in a `select` statement that includes the `for browse` option.

The keyword `distinct` in the select statement is ignored in browse mode.

Browse a Table

To browse a table in a front-end application, append the `for browse` keywords to the end of the `select` statement.

For example:

```
<Start of> select <statement in an Open Client application>
. .
for browse
<Completion of the Open Client application routine>
```

You can browse a table in a front-end application if its rows have been timestamped.

Add a timestamp Column to a New Table

When you create a new table for browsing, include a column named `timestamp` in the table definition. This column is automatically assigned the `timestamp` datatype.

For example:

```
create table newtable(col1 int, timestamp,
    col3 char(7))
```

When you insert or update a row, SAP ASE identifies the event by automatically assigning a unique `varbinary` value to the `timestamp` column.

Add a timestamp Column to an Existing Table

To prepare an existing table for browsing, use `alter table` to add a column named `timestamp`.

For example:

```
alter table oldtable add timestamp
```

A `timestamp` column with a null value is added to each existing row. To generate a timestamp, update each row without specifying new column values.

For example:

```
update oldtable
set col1 = col1
```

Comparing timestamp Values

Use the `tsequal` system function to compare timestamps when you are using browse mode in a front-end application.

For example, the following statement updates a row in `publishers` that has been browsed. It compares the `timestamp` column in the browsed version of the row with the hexadecimal timestamp in the stored version. If the two timestamps are not equal, you receive an error message, and the row is not updated.

```
update publishers
set city = "Springfield"
where pub_id = "0736"
and tsequal(timestamp,0x0001000000002ea8)
```

Do not use the `tsequal` function in the `where` clause as a search argument. When you use `tsequal`, the rest of the `where` clause should uniquely match a single row. Use the `tsequal` function only in `insert` and `update` statements. If you use a `timestamp` column as a search clause, compare it like a regular `varbinary` column, that is,`< timestamp1 = timestamp2>`.

21 Triggers: Enforce Referential Integrity

You can use triggers to perform a number of automatic actions, such as cascading changes through related tables, enforcing column restrictions, comparing the results of data modifications, and maintaining the referential integrity of data across a database.

Triggers are automatic no matter what caused the data modification—a clerk's data entry or an application action. A trigger is specific to one or more data modification operations (`update`, `insert`, and `delete`), and is executed once for each SQL statement.

For example, to prevent users from removing any publishing companies from the `publishers` table, you could use:

```
create trigger del_pub
on publishers
for delete
as
begin
    rollback transaction
    print "You cannot delete any publishers!"
end
```

The next time someone tries to remove a row from the `publishers` table, the `del_pub` trigger cancels the deletion, rolls back the transaction, and prints a message.

A trigger "fires" only after the data modification statement has completed and SAP ASE has checked for any datatype, rule, or integrity constraint violation. The trigger and the statement that fires it are treated as a single transaction that can be rolled back from within the trigger. If SAP ASE detects a severe error, the entire transaction is rolled back.

Use triggers to:

- Cascade changes through related tables in the database. For example, a delete trigger on the `title_id` column of the `titles` table can delete matching rows in other tables, using the `title_id` column as a unique key to locating rows in `titleauthor` and `roysched`.
- Disallow, or roll back, changes that would violate referential integrity, canceling the attempted data modification transaction. Such a trigger might go into effect when you try to insert a foreign key that does not match its primary key. For example, you could create an insert trigger on `titleauthor` that rolled back an insert if the new `titleauthor.title_id` value did not have a matching value in `titles.title_id`.
- Enforce restrictions that are much more complex than those that are defined with rules. Unlike rules, triggers can reference columns or database objects. For example, a trigger can roll back updates that attempt to increase a book's price by more than 1 percent of the advance.
- Perform simple "what if" analyses. For example, a trigger can compare the state of a table before and after a data modification and take action based on that comparison.

21.1 Use Triggers Versus Integrity Constraints

As an alternative to using triggers, you can use the referential integrity constraint of the `create table` statement to enforce referential integrity across tables in the database.

However, referential integrity constraints cannot:

- Cascade changes through related tables in the database
- Enforce complex restrictions by referencing other columns or database objects
- Perform “what if” analyses

Also, referential integrity constraints do not roll back the current transaction as a result of enforcing data integrity. With triggers, you can either roll back or continue the transaction, depending on how you handle referential integrity.

If your application requires one of the above tasks, use a trigger. Otherwise, use a referential integrity constraint to enforce data integrity. SAP ASE checks referential integrity constraints before it checks triggers so that a data modification statement that violates the constraint does not also fire the trigger.

Related Information

Transactions: Maintain Data Consistency and Recovery [page 632]
Databases and Tables [page 50]

21.2 Create Triggers

A trigger is a database object. When you create a trigger, you specify the table and the data modification commands that should fire, or activate, the trigger. Then, you specify any actions the trigger is to take.

For example, this trigger prints a message every time anyone tries to insert, delete, or update data in the `titles` table:

```
create trigger t1
on titles
for insert, update, delete
as
print "Now modify the titleauthor table the same way."
```

> **Note**
>
> Unless you specify the existence of multiple triggers, each new trigger for the same operation—`insert`, `update` or `delete`—on a table or column overwrites the previous one without warning, and old triggers are dropped automatically.

21.2.1 create trigger Syntax

The `create` clause creates and names the trigger. A trigger's name must conform to the rules for identifiers.

The `on` clause gives the name of the table that activates the trigger. This table is sometimes called the trigger table.

A trigger is created in the current database, although it can reference objects in other databases. The owner name that qualifies the trigger name must be the same as the one in the table. Only a table owner can create a trigger on a table. If the table owner is given with the table name in the `create trigger` clause or the `on` clause, it must also be specified in the other clause.

The `for` clause specifies which data modification commands on the trigger table activate the trigger. In the earlier example, an `insert`, `update`, or `delete` to `titles` makes the message print.

The SQL statements specify trigger conditions and trigger actions. Trigger conditions specify additional criteria that determine whether `insert`, `delete`, or `update` causes the trigger actions to be carried out. You can group multiple trigger actions in an `if` clause with `begin` and `end`.

An `if update` clause tests for an insert or update to a specified column. For updates, the `if update` clause evaluates to true when the column name is included in the `set` clause of an `update` statement, even if the update does not change the value of the column. Do not use the `if update` clause with `delete`. You can specify more than one column, and you can use more than one `if update` clause in a `create trigger` statement. Since you specify the table name in the `on` clause, do not use the table name in front of the column name with `if update`. See the *Reference Manual: Commands*.

These statements are not allowed in triggers:

- All `create` commands, including `create database`, `create table`, `create index`, `create procedure`, `create default`, `create rule`, `create trigger`, and `create view`
- All `drop` commands
- `alter table` and `alter database`
- `truncate table`
- `grant` and `revoke`
- `update statistics`
- `reconfigure`
- `load database` and `load transaction`
- `disk init`, `disk mirror`, `disk refit`, `disk reinit`, `disk remirror`, `disk unmirror`
- `select into`

21.3 Use Triggers to Maintain Referential Integrity

Triggers maintain referential integrity, which assures that vital data in your database—such as the unique identifier for a given piece of data—remains accurate and can be used consistently as other data in the database changes. Referential integrity is coordinated through the use of primary and foreign keys.

The primary key is a column or combination of columns with values that uniquely identify a row. The value cannot be null and must have a unique index. A table with a primary key is eligible for joins with foreign keys in

other tables. Think of the primary-key table as the master table in a master-detail relationship. There can be many such master-detail groups in a database.

You can use `sp_primarykey` to mark a primary key for use with `sp_helpjoins` to add it to the `syskeys` table.

For example, the `title_id` column is the primary key of `titles`. It uniquely identifies the books in `titles` and joins with `title_id` in `titleauthor`, `salesdetail`, and `roysched`. The `titles` table is the master table in relation to `titleauthor`, `salesdetail`, and `roysched`.

The foreign key is a column, or combination of columns, match the primary key. A foreign key does not have to be unique. It is often in a many-to-one relationship to a primary key. Foreign-key values should be copies of the primary-key values. That means no value in the foreign key should exist unless the same value exists in the primary key. A foreign key may be null; if any part of a composite foreign key is null, the entire foreign key must be null. Tables with foreign keys are often called detail tables or dependent tables to the master table.

You can use `sp_foreignkey` to mark foreign keys in your database. This flags them for use with `sp_helpjoins` and other procedures that reference the `syskeys` table. The `title_id` columns in `titleauthor`, `salesdetail`, and `roysched` are foreign keys; the tables are detail tables. In most cases, you can enforce referential integrity between tables using referential constraints (constraints that ensure the data inserted into a particular column has matching values in another table), because the maximum number of references allowed for a single table is 200. If a table exceeds that limit, or has special referential integrity needs, use referential integrity triggers.

Referential integrity triggers keep the values of foreign keys in sync with those in primary keys. When a data modification affects a key column, triggers compare the new column values to related keys by using temporary worktables called trigger test tables. When you write your triggers, base your comparisons on the data that is temporarily stored in the trigger test tables.

21.3.1 Test Data Modifications Against the Trigger Test Tables

SAP ASE uses two special tables in trigger statements: the `deleted` table and the `inserted` table. These are temporary tables used in trigger tests. When you write triggers, you can use these tables to test the effects of a data modification and to set conditions for trigger actions.

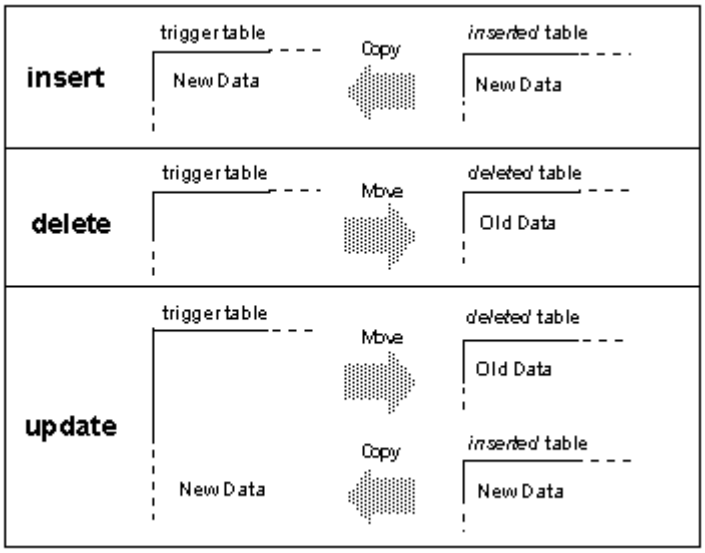
You cannot directly alter the data in the trigger test tables, but you can use the tables in `select` statements to detect the effects of an `insert`, `update`, or `delete`.

- The `deleted` table stores copies of the affected rows during `delete` and `update` statements. During the execution of a `delete` or `update` statement, rows are removed from the trigger table and transferred to the `deleted` table. The `deleted` and trigger tables ordinarily have no rows in common.
- The `inserted` table stores copies of the affected rows during `insert` and `update` statements. During an `insert` or an `update`, new rows are added to the `inserted` and trigger tables at the same time. The rows in `inserted` are copies of the new rows in the trigger table. The following trigger fragment uses the `inserted` table to test for changes to the `titles` table `title_id` column:

```
if (select count(*)
    from titles, inserted
    where titles.title_id = inserted.title_id) !=
    @@rowcount
```

> **Note**
>
> Both inserted and deleted tables appear as views in the transaction log, but they are fake tables in `syslogs`.

An `update` is, effectively, a delete followed by an insert; the old rows are copied to the `deleted` table first; then the new rows are copied to the trigger table and to the `inserted` table. The following illustration shows the condition of the trigger test tables during an `insert`, a `delete`, and an `update`:



When setting trigger conditions, use the trigger test tables that are appropriate for the data modification. It is not an error to reference `deleted` while testing an `insert` or `inserted` while testing a `delete`; however, those trigger test tables do not contain any rows.

> **Note**
>
> A given trigger fires only once per query. If trigger actions depend on the number of rows affected by a data modification, use tests, such as an examination of `<@@rowcount>` for multirow data modifications, and take appropriate actions.

The following trigger examples accommodate multirow data modifications where necessary. The `<@@rowcount>` variable, which stores the "number of rows affected" by the most recent data modification operation, tests for a multirow `insert`, `delete`, or `update`. If any other `select` statement precedes the test on `<@@rowcount>` within the trigger, use local variables to store the value for later examination. All Transact-SQL statements that do not return values reset `<@@rowcount>` to 0.

21.3.2 Insert Trigger Example

When you insert a new foreign-key row, make sure the foreign key matches a primary key. The trigger should check for joins between the inserted rows (using the `inserted` table) and the rows in the primary-key table, and then roll back any inserts of foreign keys that do not match a key in the primary-key table.

The following trigger compares the `title_id` values from the `inserted` table with those from the `titles` table. It assumes that you are making an entry for the foreign key and that you are not inserting a null value. If the join fails, the transaction is rolled back.

```
create trigger forinsertrig1
on salesdetail
for insert
as
if (select count(*)
    from titles, inserted
    where titles.title_id = inserted.title_id) !=
    @@rowcount
/* Cancel the insert and print a message.*/
  begin
    rollback transaction
    print "No, the title_id does not exist in
    titles."
  end
/* Otherwise, allow it. */
else
  print "Added! All title_id's exist in titles."
```

`<@@rowcount>` refers to the number of rows added to the `salesdetail` table. This is also the number of rows added to the `inserted` table. The trigger joins `titles` and `inserted` to determine whether all the `title_id`s added to `salesdetail` exist in the `titles` table. If the number of joined rows, which is determined by the `select count(*)` query, differs from `<@@rowcount>`, then one or more of the inserts is incorrect, and the transaction is canceled.

This trigger prints one message if the insert is rolled back and a different one if it is accepted. To test for the first condition, try this `insert` statement:

```
insert salesdetail
values ("7066", "234517", "TC9999", 70, 45)
```

To test for the second condition, enter:

```
insert salesdetail
values ("7896", "234518", "TC3218", 75, 80)
```

21.3.3 Delete Trigger Examples

When you delete a primary-key row, also delete corresponding foreign-key rows in dependent tables. This preserves referential integrity by ensuring that detail rows are removed when their master row is deleted.

If you do not delete the corresponding rows in the dependent tables, you may end up with a database with detail rows that cannot be retrieved or identified. To properly delete the dependent foreign-key rows, use a trigger that performs a cascading `delete`.

Cascading Delete Example

When a `delete` statement on `titles` is executed, one or more rows are removed from the `titles` table and are added to `deleted`.

A trigger can check the dependent tables—`titleauthor`, `salesdetail`, and `roysched`—to see if they have any rows with a `title_id` that matches the `title_id`s removed from `titles` and is now stored in the `deleted` table. If the trigger finds any such rows, it removes them.

```
create trigger delcascadetrig
on titles
for delete
as
delete titleauthor
from titleauthor, deleted
where titleauthor.title_id = deleted.title_id
/* Remove titleauthor rows that match deleted
** (titles) rows.*/
delete salesdetail
from salesdetail, deleted
where salesdetail.title_id = deleted.title_id
/* Remove salesdetail rows that match deleted
** (titles) rows.*/
delete roysched
from roysched, deleted
where roysched.title_id = deleted.title_id
/* Remove roysched rows that match deleted
** (titles) rows.*/
```

Restricted Delete Examples

In practice, you may want to keep some of the detail rows, either for historical purposes (to check how many sales were made on discontinued titles while they were active) or because transactions on the detail rows are not yet complete.

A well-written trigger should take these factors into consideration.

Preventing Primary Key Deletions:

The `deltitle` trigger supplied with `pubs2` prevents the deletion of a primary key if there are any detail rows for that key in the `salesdetail` table. This trigger preserves the ability to retrieve rows from `salesdetail`:

```
create trigger deltitle
on titles
for delete
as
if (select count(*)
    from deleted, salesdetail
    where salesdetail.title_id =
    deleted.title_id) > 0
  begin
    rollback transaction
    print "You cannot delete a title with sales."
  end
```

In this trigger, the row or rows deleted from `titles` are tested by being joined with the `salesdetail` table. If a join is found, the transaction is canceled.

Similarly, the following restricted delete prevents deletes if the primary table, `titles`, has dependent children in `titleauthor`. Instead of counting the rows from `deleted` and `titleauthor`, it checks to see if `title_id` was deleted. This method is more efficient for performance reasons because it checks for the existence of a particular row rather than going through the entire table and counting all the rows.

Recording Errors That Occur:

The next example uses `raiserror` for error message 35003. `raiserror` sets a system flag to record that the error occurred. Before trying this example, add error message 35003 to the `sysusermessages` system table:

```
sp_addmessage 35003, "restrict_dtrig - delete failed: row exists in titleauthor
for this title_id."
```

The trigger is:

```
create trigger restrict_dtrig
on titles
for delete as
if exists (select * from titleauthor, deleted where
          titleauthor.title_id = deleted.title_id)
   begin
           rollback transaction
           raiserror 35003
           return
   end
```

To test this trigger, try this `delete` statement:

```
delete titles
where title_id = "PS2091"
```

21.3.4 Update Trigger Examples

A primary key is the unique identifier for its row and for foreign-key rows in other tables. Generally, you should not allow updates to primary keys. A change or an update to a foreign key by itself is probably an error. A foreign key is a copy of the primary key. Never design the two to be independent.

This example cascades an update from the primary table `titles` to the dependent tables `titleauthor` and `roysched`.

```
create trigger cascade_utrig
on titles
for update as
if update(title_id)
begin
     update titleauthor
           set title_id = inserted.title_id
           from titleauthor, deleted, inserted
           where deleted.title_id = titleauthor.title_id
     update roysched
           set title_id = inserted.title_id
           from roysched, deleted, inserted
           where deleted.title_id = roysched.title_id
     update salesdetail
           set title_id = inserted.title_id
           from salesdetail, deleted, inserted
           where deleted.title_id = salesdetail.title_id
```

```
end
```

To test this trigger, suppose that the book *Secrets of Silicon Valley* was reclassified to a psychology book from popular_comp. The following query updates the `title_id` PC8888 to PS8888 in `titleauthor`, `roysched`, and `titles`.

```
update titles
set title_id = "PS8888"
where title_id = "PC8888"
```

Restricted Update Triggers

An attempt to update a primary key should be taken very seriously. In this case, protect referential integrity by rolling back the update unless specified conditions are met.

SAP suggests that you prohibit any editing changes to a primary key, for example, by revoking all permissions on that column. However, to prohibit updates only under certain circumstances, use a trigger.

Restricted Update Trigger Using Date Functions:

The following trigger prevents updates to `titles.title_id` on the weekend. The `if update` clause in `stopupdatetrig` allows you to focus on a particular column, `titles.title_id`. Modifications to the data in that column cause the trigger fire. Changes to the data in other columns do not. When this trigger detects an update that violates the trigger conditions, it cancels the update and prints a message. To test this, substitute a different day of the week for “Saturday” or “Sunday.”

```
create trigger stopupdatetrig
on titles
for update
as
/* If an attempt is made to change titles.title_id
** on Saturday or Sunday, cancel the update. */
if update (title_id)
    and datename(dw, getdate())
    in ("Saturday", "Sunday")
  begin
    rollback transaction
    print "We do not allow changes to "
    print "primary keys on the weekend."
  end
```

Restricted Update Triggers With Multiple Actions:

You can specify multiple trigger actions on more than one column using `if update`. The following example modifies `stopupdatetrig` to include additional trigger actions for updates to `titles.price` or `titles.advance`. The example prevents updates to the primary key on weekends, and prevents updates to the price or advance of a title, unless the total revenue amount for that title surpasses its advance amount. You can use the same trigger name because the modified trigger replaces the old trigger when you create it again.

```
create trigger stopupdatetrig
on titles
for update
as
if update (title_id)
  and datename(dw, getdate())
  in ("Saturday", "Sunday")
```

```
  begin
    rollback transaction
    print "We do not allow changes to"
    print "primary keys on the weekend!"
  end
if update (price) or update (advance)
  if exists (select * from inserted
    where (inserted.price * inserted.total_sales)
    < inserted.advance)
    begin
      rollback transaction
      print "We do not allow changes to price or"
      print "advance for a title until its total"
      print "revenue exceeds its latest advance."
    end
```

The next example, created on `titles`, prevents `update` if any of the following conditions is true:

- The user tries to change a value in the primary key `title_id` in `titles`
- The dependent key `pub_id` is not found in `publishers`
- The target column does not exist or is null

Before you run this example, make sure the following error messages exist in `sysusermessages`:

```
sp_addmessage 35004, "titles_utrg - Update Failed: update of primary keys %1! is
not allowed."
sp_addmessage 35005, "titles_utrg - Update Failed: %1! not found in authors."
```

The trigger is as follows:

```
create trigger title_utrg
on titles
for update as
begin
     declare @num_updated int,
             @col1_var varchar(20),
             @col2_var varchar(20)
/* Determine how many rows were updated. */
select @num_updated = @@rowcount
    if @num_updated = 0
    return
/* Ensure that title_id in titles is not changed. */
if update(title_id)
    begin
       rollback transaction
     select @col1_var = title_id from inserted
     raiserror 35004 , @col1_var
     return
   end
 /* Make sure dependencies to the publishers table are accounted for. */
 if update(pub_id)
   begin
     if (select count(*) from inserted, publishers
         where inserted.pub_id = publishers.pub_id
         and inserted.pub_id is not null) != @num_updated
     begin
         rollback transaction
         select @col1_var = pub_id from inserted
         raiserror 35005, @col1_var
         return
     end
   end
/* If the column is null, raise error 24004 and rollback the
** trigger. If the column is not null, update the roysched table
** restricting the update. */
```

```
   if update(price)
     begin
         if exists (select count(*) from inserted
         where price = null)
     begin
         rollback trigger with
         raiserror 24004 "Update failed : Price cannot be null. "
     end
     else
     begin
         update roysched
         set lorange = 0,
         hirange = price * 1000
         from inserted
         where roysched.title_id =  inserted.title_id
     end
    end
end
```

To test for the first error message, 35004 (failure to update the primary keys), enter:

```
update titles
set title_id = "BU7777"
where title_id = "BU2075"
```

To test for the second error message, 35005 (update failed, object not found):

```
update titles
set pub_id = "7777"
where pub_id = "0877"
```

To test for the third error, which generates message 24004 (update failed, object is null):

```
update titles
set price = 10.00
where title_id = "PC8888"
```

This query fails because the `price` column in `titles` is null. If it were not null, it would have updated the price for title PC8888 and performed the necessary recalculations for the `roysched` table. Error 24004 is not in `sysusermessages` but it is valid in this case. It demonstrates the "rollback trigger with raiserror" section of the code.

Update a Foreign Key

To allow updates of a foreign key, protect integrity by creating a trigger that checks updates against the `master` table and rolls them back if they do not match the primary key.

In the following example, the trigger tests for two possible sources of failure: either the `title_id` is not in the `salesdetail` table or it is not in the `titles` table.

This example uses nested `if...else` statements. The first `if` statement is true when the value in the `where` clause of the `update` statement does not match a value in `salesdetail`, that is, the `inserted` table will not contain any rows, and the `select` returns a null value. If this test is passed, the next `if` statement ascertains whether the new row or rows in the `inserted` table join with any `title_id` in the `titles` table. If any row

does not join, the transaction is rolled back, and an error message prints. If the join succeeds, a different message prints.

```
create trigger forupdatetrig
on salesdetail
for update
as
declare @row int
/* Save value of rowcount. */
select @row = @@rowcount
if update (title_id)
  begin
    if (select distinct inserted.title_id
        from inserted) is null
      begin
        rollback transaction
        print "No, the old title_id must be in"
        print "salesdetail."
      end
    else
      if (select count(*)
          from titles, inserted
          where titles.title_id =
          inserted.title_id) != @row
        begin
          rollback transaction
          print "No, the new title_id is not in"
          print "titles."
        end
      else
        print "salesdetail table updated"
  end
```

21.4 Multirow Considerations

Multirow considerations are particularly important when the function of a trigger is to recalculate summary values, or provide ongoing tallies.

Triggers used to maintain summary values should contain `group by` clauses or subqueries that perform implicit grouping. This creates summary values when more than one row is being inserted, updated, or deleted. Since a `group by` clause imposes extra overhead, the following examples are written to test whether `<@@rowcount>` = 1, meaning that only one row in the trigger table was affected. If `<@@rowcount>` = 1, the trigger actions take effect without a `group by` clause.

21.4.1 Insert Trigger Example Using Multiple Rows

Use a insert trigger to update a column everytime a row is added.

This insert trigger example updates the `total_sales` column in the `titles` table every time a new `salesdetail` row is added.

The trigger fires whenever you record a sale by adding a row to the `salesdetail` table. It updates the `total_sales` column in the `titles` table so that `total_sales` is equal to its previous value plus the value added to `salesdetail.qty`. This keeps the totals up to date for inserts into `salesdetail.qty`.

```
create trigger intrig
on salesdetail
for insert as
    /* check value of @@rowcount */
if @@rowcount = 1
    update titles
      set total_sales = total_sales + qty
      from inserted
      where titles.title_id = inserted.title_id
else
    /* when @@rowcount is greater than 1,
       use a group by clause */
    update titles
      set total_sales =
        total_sales + (select sum(qty)
      from inserted
      group by inserted.title_id
      having titles.title_id = inserted.title_id)
```

21.4.2 Delete Trigger Example Using Multiple Rows

Use a delete trigger to update a column when rows are deleted.

This delete trigger example updates the `total_sales` column in the `titles` table every time one or more `salesdetail` rows are deleted.

```
create trigger deltrig
on salesdetail
for delete
as
    /* check value of @@rowcount */
if @@rowcount = 1
    update titles
      set total_sales = total_sales - qty
      from deleted
      where titles.title_id = deleted.title_id
else
    /* when rowcount is greater than 1,
       use a group by clause */
    update titles
      set total_sales =
        total_sales - (select sum(qty)
      from deleted
      group by deleted.title_id
      having titles.title_id = deleted.title_id)
```

This trigger goes into effect whenever a row is deleted from the `salesdetail` table. It updates the `total_sales` column in the `titles` table so that `total_sales` is equal to its previous value minus the value subtracted from `salesdetail.qty`.

21.4.3 Update Trigger Example Using Multiple Rows

Use a trigger to update a column everytime a field in a row is updated.

This update trigger example updates the `total_sales` column in the `titles` table every time the `qty` field in a `salesdetail` row is updated (an update is an insert followed by a delete). This trigger references both the `inserted` and the `deleted` trigger test tables.

```
create trigger updtrig
on salesdetail
for update
as
if update (qty)
begin
    /* check value of @@rowcount */
    if @@rowcount = 1
        update titles
          set total_sales = total_sales +
            inserted.qty - deleted.qty
          from inserted, deleted
          where titles.title_id = inserted.title_id
          and inserted.title_id = deleted.title_id
    else
    /* when rowcount is greater than 1,
       use a group by clause */
    begin
        update titles
          set total_sales = total_sales +
            (select sum(qty)
                from inserted
                group by inserted.title_id
                having titles.title_id =
                inserted.title_id)
        update titles
          set total_sales = total_sales -
            (select sum(qty)
                from deleted
                group by deleted.title_id
                having titles.title_id =
                deleted.title_id)
    end
end
```

21.4.4 Conditional Insert Trigger Example Using Multiple Rows

You do not have to roll back all data modifications simply because some of them are unacceptable. Using a correlated subquery in a trigger can force the trigger to examine the modified rows one by one.

The trigger can then take different actions on different rows.

The following trigger example assumes the existence of a table called `junesales`. Here is its `create` statement:

```
create table junesales
(stor_id    char(4)     not null,
ord_num     varchar(20) not null,
title_id    tid         not null,
```

```
qty         smallint    not null,
discount    float       not null)
```

Insert four rows in the `junesales` table, to test the conditional trigger. Two of the `junesales` rows have `title_ids` that do not match any of those already in the `titles` table.

```
insert junesales values ("7066", "BA27619", "PS1372", 75, 40)
insert junesales values ("7066", "BA27619", "BU7832", 100, 40)
insert junesales values ("7067", "NB-1.242", "PSxxxx", 50, 40)
insert junesales values ("7131", "PSyyyy", "PSyyyy", 50, 40)
```

When you insert data from `junesales` into `salesdetail`, the statement looks like this:

```
insert salesdetail
select * from junesales
```

The trigger `conditionalinsert` analyzes the insert row by row and deletes the rows that do not have a `title_id` in `titles`:

```
create trigger conditionalinsert
on salesdetail
for insert as
if
(select count(*) from titles, inserted
where titles.title_id = inserted.title_id)
    != @@rowcount
begin
  delete salesdetail from salesdetail, inserted
    where salesdetail.title_id = inserted.title_id
    and inserted.title_id not in
    (select title_id from titles)
  print "Only records with matching title_ids
    added."
end
```

The trigger deletes the unwanted rows. This ability to delete rows that have just been inserted relies on the order in which processing occurs when triggers are fired. First, the rows are inserted into the table and the `inserted` table; then, the trigger fires.

Related Information

Correlated Subqueries [page 281]

21.5 Roll Back Triggers

You can roll back triggers using either the `rollback trigger` statement or the `rollback transaction` statement (if the trigger is fired as part of a transaction). However, `rollback trigger` rolls back only the effect of the trigger and the statement that caused the trigger to fire; `rollback transaction` rolls back the entire transaction.

For example:

```
begin tran
insert into publishers (pub_id) values ("9999")
insert into publishers (pub_id) values ("9998")
commit tran
```

If the second `insert` statement causes a trigger on `publishers` to issue a `rollback trigger`, only that `insert` is affected; the first `insert` is not rolled back. If that trigger issues a `rollback transaction` instead, both `insert` statements are rolled back as part of the transaction.

The syntax for `rollback trigger` is:

```
rollback trigger
     [with raiserror_statement]
```

`<raiserror_statement>` prints a user-defined error message and sets a system flag to record that an error condition has occurred. This provides the ability to raise an error to the client when the `rollback trigger` is executed, so that the transaction state in the error reflects the rollback. For example:

```
rollback trigger with raiserror 25002
    "title_id does not exist in titles table."
```

The following example of an insert trigger performs a similar task to the trigger `forinsertrig1`. However, this trigger uses a `rollback trigger` instead of a `rollback transaction` to raise an error when it rolls back the insertion but not the transaction.

```
create trigger forinsertrig2
on salesdetail
for insert
as
if (select count(*) from titles, inserted
    where titles.title_id = inserted.title_id) !=
    @@rowcount
   rollback trigger with raiserror 25003
     "Trigger rollback: salesdetail row not added
     because a title_id does not exist in titles."
```

When the `rollback trigger` is executed, SAP ASE aborts the currently executing command and halts execution of the rest of the trigger. If the trigger that issues the `rollback trigger` is nested within other triggers, SAP ASE rolls back all the work done in these triggers up to and including the update that caused the first trigger to fire.

When triggers that include `rollback transaction` statements fired from a batch, they abort the entire batch. In the following example, if the `insert` statement fires a trigger that includes a `rollback transaction` (such as `forinsertrig1`), the `delete` statement is not executed, since the batch is aborted:

```
insert salesdetail values ("7777", "JR123",
    "PS9999", 75, 40)
delete salesdetail where stor_id = "7067"
```

If triggers that include `rollback transaction` statements are fired from within a user-defined transaction, the `rollback transaction` rolls back the entire batch. In the following example, if the `insert` statement fires a trigger that includes a `rollback transaction`, the `update` statement is also rolled back:

```
begin tran
```

```
update stores set payterms = "Net 30"
    where stor_id = "8042"
insert salesdetail values ("7777", "JR123",
    "PS9999", 75, 40)
commit tran
```

SAP ASE ignores a `rollback trigger` executed outside of a trigger and does not issue a `raiserror` associated with the statement. However, a `rollback trigger` executed outside a trigger but inside a transaction generates an error that causes SAP ASE to roll back the transaction and abort the current statement batch.

Related Information

Transactions: Maintain Data Consistency and Recovery [page 632]
Batches and Control-of-Flow Language [page 437]
Insert Trigger Example [page 588]
Transactions: Maintain Data Consistency and Recovery [page 632]

21.6 Global Login Triggers

Use `sp_logintrigger` to set a global login trigger that is executed at each user login. To take user-specific actions, use `alter login` or `create login` to set a user-specific login trigger.

See the *Reference Manual: Commnands*.

21.7 Nesting Triggers

Triggers can nest to a depth of 16 levels. The current nesting level is stored in the`< @@nestlevel>` global variable.

Nesting is enabled at installation. A system administrator can use the `allow nested triggers` configuration parameter to turn trigger nesting on and off.

If nested triggers are enabled, a trigger that changes a table on which there is another trigger fires the second trigger, which can in turn fire a third trigger, and so forth. If any trigger in the chain sets off an infinite loop, the nesting level is exceeded and the trigger aborts. You can use nested triggers to perform useful housekeeping functions such as storing a backup copy of rows affected by a previous trigger.

For example, you can create a trigger on `titleauthor` that saves a backup copy of `titleauthor` rows that was deleted by the `delcascadetrig` trigger. With the `delcascadetrig` trigger in effect, deleting the `title_id` "PS2091" from `titles` also deletes any corresponding rows from `titleauthor`. To save the data, you can create a `delete` trigger on `titleauthor` that saves the deleted data in another table, `del_save`:

```
create trigger savedel
```

```
on titleauthor
for delete
as
insert del_save
select * from deleted
```

SAP suggests that you use nested triggers in an order-dependent sequence. Use separate triggers to cascade data modifications.

> **Note**
>
> When you put triggers into a transaction, a failure at any level of a set of nested triggers cancels the transaction and rolls back all data modifications. Use `print` or `raiserror` statements in your triggers to determine where failures occur.

A `rollback transaction` in a trigger at any nesting level rolls back the effects of each trigger and cancels the entire transaction. A `rollback trigger` affects only the nested triggers and the data modification statement that caused the initial trigger to fire.

21.7.1 Trigger Self-Recursion

By default, a trigger does not recursively call itself. That is, an update trigger does not call itself in response to a second update to the same table within the trigger. If an update trigger on one column of a table results in an update to another column, the update trigger fires only once.

However, you can turn on the `self_recursion` option of the `set` command to allow triggers to call themselves recursively. The `allow nested triggers` configuration variable must also be enabled for self-recursion to occur.

The `self_recursion` setting remains in effect only for the duration of a current client session. If the option is set as part of a trigger, its effect is limited by the scope of the trigger that sets it. If the trigger that sets `self_recursion on` returns or causes another trigger to fire, this option reverts to `off`. Once a trigger turns on the `self_recursion` option, it can repeatedly loop, if its own actions cause it to fire again, but it cannot exceed the limit of 16 nesting levels.

For example, assume that the following `new_budget` table exists in `pubs2`:

```
select * from new_budget
```

```
unit            parent_unit     budget
--------------- --------------- -------
one_department  one_division    10
one_division    company_wide    100
company_wide    NULL            1000

(3 rows affected)
```

You can create a trigger that recursively updates `new_budget` whenever its `budget` column is changed, as follows:

```
create trigger budget_change
on new_budget
for update as
```

```
if exists (select * from inserted
            where parent_unit is not null)
begin
    set self_recursion on
    update new_budget
    set new_budget.budget = new_budget.budget +
        inserted.budget - deleted.budget
    from inserted, deleted, new_budget
    where new_budget.unit = inserted.parent_unit
        and new_budget.unit = deleted.parent_unit
end
```

If you update `new_budget.budget` by increasing the budget of unit `one_department` by 3, SAP ASE behaves as follows (assuming that nested triggers are enabled):

1. Increasing `one_department` from 10 to 13 fires the `budget_change` trigger.
2. The trigger updates the budget of the parent of `one_department` (in this case, `one_division`) from 100 to 103, which fires the trigger again.
3. The trigger updates the parent of `one_division` (in this case `company_wide`) from 1000 to 1003, which causes the trigger to fire for the third time.
4. The trigger attempts to update the parent of `company_wide`, but since none exists (the value is "NULL"), the last `update` never occurs and the trigger is not fired, ending the self-recursion. You can query `new_budget` to see the final results, as follows:

   ```
   select * from new_budget
   ```

   ```
   unit            parent_unit     budget
   --------------- --------------- -------
   one_department  one_division    13
   one_division    company_wide    103
   company_wide    NULL            1003

   (3 rows affected)
   ```

A trigger can also be recursively executed in other ways. A trigger calls a stored procedure that performs actions that cause the trigger to fire again (it is reactivated only if nested triggers are enabled). Unless conditions within the trigger limit the number of recursions, the nesting level can overflow.

For example, if an update trigger calls a stored procedure that performs an update, the trigger and stored procedure execute only once if `nested triggers` is set to `off`. If `nested triggers` is set to `on`, and the number of updates exceeds 16 by some condition in the trigger or procedure, the loop continues until the trigger or procedure exceeds the 16-level maximum nesting value.

21.8 Rules Associated with Triggers

In addition to anticipating the effects of a multirow data modification, trigger rollbacks, and trigger nesting, there are other factors to consider when you write triggers, such as permissions, restrictions, and performance.

In additional to the rules associated with triggers, consider these situations when you create triggers:

- Suppose you have an `insert` or `update` trigger that calls a stored procedure, which in turn updates the base table. If the `nested triggers` configuration parameter is set to true, the trigger enters an infinite loop. Before executing an `insert` or `update` trigger, set `sp_configure "nested triggers"` to false.
- When you execute `drop table`, any triggers that are dependent on that table are also dropped. To preserve any such triggers, use `sp_rename` to change their names before dropping the table.
- Use `sp_depends` to see a report on the tables and views referred to in a trigger.
- Use `sp_rename` to rename a trigger.
- A trigger fires only once per query. If the query is repeated in a loop, the trigger fires as many times as the query is repeated.

21.8.1 Triggers and Permissions

A trigger is defined on a table. Only the table owner has `create trigger` and `drop trigger` permissions for that table; these permissions cannot be transferred to others.

SAP ASE accepts a trigger definition that attempts actions for which you do not have permission. The existence of such a trigger aborts any attempt to modify the trigger table because incorrect permissions cause the trigger to fire and fail. The transaction is canceled. You must rectify the permissions or drop the trigger.

For example, Jose owns `salesdetail` and creates a trigger on it. The trigger is designed to update `titles.total_sales` when `salesdetail.qty` is updated. However, Mary is the owner of `titles`, and has not granted Jose permission on `titles`. When Jose tries to update `salesdetail`, SAP ASE detects not only the trigger, but also Jose's lack of permissions on `titles`, and rolls back the update transaction. Jose must either get update permission on `titles.total_sales` from Mary or drop the trigger on `salesdetail`.

21.8.2 Trigger Restrictions

SAP ASE imposes certain limitations on triggers.

- A table can have a maximum of three triggers: one update trigger, one insert trigger, and one delete trigger.
- Each trigger can apply to only one table. However, a single trigger can incorporate all three user actions: `update`, `insert`, and `delete`.
- You cannot create a trigger on a view or on a session-specific temporary table, though triggers can reference views or temporary tables.
- The `writetext` statement does not activate insert or update triggers.

- Although a `truncate table` statement is, similar to a `delete` without a `where` clause, because it removes all rows, it cannot fire a trigger, because individual row deletions are not logged.
- You cannot create a trigger or build an index or a view on a temporary object (`@object`).
- You cannot create triggers on system tables. If you try to create a trigger on a system table, SAP ASE returns an error message and cancels the trigger.
- You cannot use triggers that select from a `text` column or an `image` column in a table that has the same trigger inserts or deletes.
- If Component Integration Services is enabled, triggers have limited usefulness on proxy tables because you cannot examine the rows being inserted, updated, or deleted (via the `inserted` and `deleted` tables). You can create, then invoke, a trigger on a proxy table. However, deleted or inserted data is not written to the transaction log for proxy tables because the `insert` is passed to the remote server. Therefore, the inserted and deleted tables, which are actually views to the transaction log, contain no data for proxy tables.

21.8.3 Implicit and Explicit Null Values

The `if update(<column_name>)` clause is true for an `insert` statement whenever the column is assigned a value in the select list or in the `values` clause. An explicit null or a default assigns a value to a column, and thus activates the trigger. An implicit null does not.

For example, suppose you create this table:

```
create table junk
(a int null,
 b int not null)
```

then write this trigger:

```
create trigger junktrig
on junk
for insert
as
if update(a) and update(b)
        print "FIRING"
    /*"if update" is true for both columns.
      The trigger is activated.*/
insert junk (a, b) values (1, 2)
    /*"if update" is true for both columns.
      The trigger is activated.*/
insert junk values (1, 2)
    /*Explicit NULL:
      "if update" is true for both columns.
      The trigger is activated.*/
insert junk values (NULL, 2)
    /* If default exists on column a,
      "if update" is true for either column.
      The trigger is activated.*/
insert junk (b) values (2)
    /* If no default exists on column a,
      "if update" is not true for column a.
      The trigger is not activated.*/
insert junk (b) values (2)
```

The same results are produced using only this clause:

```
if update(a)
```

To create a trigger that disallows the insertion of implicit nulls, you can use:

```
if update(a) or update(b)
```

SQL statements in the trigger can then test to see if `<a>` or `<b >`is null.

21.8.4 Triggers and Performance

In terms of performance, trigger overhead is usually very low. The time involved in running a trigger is spent mostly in referencing other tables, which may be either in memory or on the database device.

The `deleted` and `inserted` trigger test tables are always in active memory. The location of other tables referenced by the trigger determines the amount of time the operation takes.

21.8.5 set Commands in Triggers

You can use the `set` command inside a trigger. The `set` option you invoke remains in effect during execution of the trigger. Then the trigger reverts to its former setting.

21.8.6 Renaming and triggers

If you change the name of an object referenced by a trigger, you must drop, then re-create the trigger so that its source text reflects the new name of the object being referenced.

Use `sp_depends` to get a report of the objects referenced by a trigger. The safest course of action is to not rename any tables or views that are referenced by a trigger.

21.9 Disable Triggers

To disable triggers during bulk `insert`, `update`, or `delete` operations, use the `disable trigger` option of the `alter table` command.

You can use this option either to disable all triggers associated with the table, or to specify a particular trigger to disable. However, any triggers you disable are fired after the copy is complete. The `insert`, `update`, and `delete` commands normally fire any trigger they encounter, which increases the time needed to perform the operation.

bcp, to maintain maximum speed for loading data, does not fire rules and triggers.To find any rows that violate rules and triggers, copy the data into the table and run queries or stored procedures that test the rule or trigger conditions.

alter table... disable trigger uses this syntax:

```
alter table [<database_name>.[<owner_name>].]<table_name>
     {enable | disable } trigger [<trigger_name>]
```

where <table_name> is the name of the table for which you are disabling triggers, and <trigger_name> is the name of the trigger you are disabling. For example, to disable the del_pub trigger in the pubs2 database:

```
alter table pubs2
disable del_pubs
```

If you do not specify a trigger, alter table disables all the triggers defined in the table.

Use alter table... enable trigger to reenable the triggers after the load database is complete. To reenable the del_pub trigger, issue:

```
alter table pubs2
enable del_pubs
```

> **Note**
>
> You can use the disable trigger feature only if you are the table owner or database administrator. If a trigger includes any insert, update, or delete statements, these statements will not run while the trigger is disabled, which may affect the referential integrity of a table.

21.10 Drop Triggers

You can remove a trigger by dropping it or by dropping the trigger table with which it is associated.

The drop trigger syntax is:

```
drop trigger [<owner>.]<trigger_name>
     [, [owner.]<trigger_name>]...
```

When you drop a table, SAP ASE drops any triggers associated with it. drop trigger permission defaults to the trigger table owner cannot be transferred.

21.11 Multiple Triggers

You can create up to 50 different triggers on a table for each command (insert, update, and delete), as well as specify the order in which the triggers are fired after statement execution by using the order parameter in the create trigger command. You can create multiple triggers without an order clause.

The syntax is:

```
create [or replace] trigger [<owner>.]<trigger_name>
    on [<owner>.]<table_name>
    for {insert | update | delete}
    [order <integer>]
    as <sql_statements>
```

`order <integer>` specifies a partial or full ordering of trigger firing:

- Full ordering occurs when you create all the triggers using the `order` clause.
- Partial ordering occurs if you do not specify the `order` clause on some of the triggers. Triggers without the `order` clause implicitly take order number 0 and do not have a defined order, except that they fire after those triggers created using `order`.

> **Note**
>
> You can only use the `order <integer>` clause with `for {insert | update | delete}`; you cannot use it with `instead of {insert | update | delete}` triggers.

If you use a duplicate number for `order`, SAP ASE reports an error. `order` numbers need not be consecutive; in fact, nonconsecutive numbers might be preferable, as they allow you to insert new triggers into the middle of an order.

Use `sp_helptrigger` to list:

- All triggers created on the table specified by `<tablename>`
- The `insert`, `update`, or `delete` command that provided the triggering action
- The trigger's order number

21.11.1 Changing the Order of When a Trigger Is Fired

To change the order in which a trigger is fired, use the `or replace` option in the `create trigger` command, using the same trigger name, action, and trigger body as the original trigger, but specifying a new `order` number.

21.11.2 Order of Triggers in Merge Statements

The `merge` statement fires triggers in a specific order.

The `merge` statement fires triggers on the target table in this order:

1. `insert`
2. `update`
3. `delete`

This means that even if the order number for an `update` statement is lower than the order number for an `insert`, the trigger for the `insert` statement fires first.

Multiple triggers for the same operation, however, are ordered. That is, all the triggers for `insert` are fired in order number first, followed by all the triggers, in order, for `update`, and so on.

Example

In this example, the dbo creates these triggers on the `GlobalSales` table:

- trigger1 for delete with order 1
- trigger2 for delete with order 4
- trigger3 for insert with order 1
- trigger4 for insert with order 5

A merge statement merges data into `GlobalSales`. If the merge includes both insert and delete operations on `GlobalSales`, SAP ASE fires the triggers in this order after execution:

- trigger3
- trigger4
- trigger1
- trigger2

21.11.3 Transactional Behavior with Multiple Triggers

A rollback transaction executed in a trigger of the `insert`, `update`, or `delete` statement that fired the trigger, along with any work performed by the trigger to be rolled back.

For multiple triggers, a rollback transaction from one trigger also rolls back the work of other triggers already fired and withholds firing any remaining triggers for the current insert, update or delete command.

21.11.4 Disabling and Reenabling Triggers

You can disable and reenable multiple triggers using the `alter table` command.

Use the `alter table` command to disable or reenable multiple triggers individually:

To disable or reenable the trigger, use:

```
alter table <table_name> {disable | enable} trigger <trigger_name>
```

21.12 Get Information About Triggers

As database objects, triggers are listed in `sysobjects` by name. The `type` column of `sysobjects` identifies triggers with the abbreviation “TR”.

This query finds the triggers that exist in a database:

```
select *
from sysobjects
where type = "TR"
```

In versions of SAP ASE earlier than 16.0, `sysobjects` saved the following:

- `deltrig` column for `delete` triggers
- `instrig` column for `insert` triggers
- `updtrig` column for `update` triggers

In SAP ASE 16.0, the first trigger created on a table for `delete`, `insert`, and `update` operations, where the trigger is created without the `order` clause (or `order 0`) is associated with the table in one of the above columns in `sysobjects`.

The second and subsequent triggers created for a given action are associated with the table through a row in `syscontraints`. In addition, any trigger with `order 1` or greater always uses `sysconstraints` for the table/trigger association.

The `sysobjects` row for the dependent table uses bits in the `sysstat2` field to indicate that a trigger is disabled. In versions earlier than SAP ASE 16.0, there could be no more than one insert, delete, or upgrade trigger, which used these three bits:

- `disable_instrig` 0x001000000
- `disable_deltrig` 0x002000000
- `disable_updtrig` 0x004000000

These bits exist in SAP ASE 16.0 and are used when a table's trigger ID is stored in `sysobjects`, as described above.

The `<@@trigger_name>` global variable returns the name of the trigger that is currently executing.

You can place the following in the body of a trigger or in the body of a stored procedure that is called (at any level of nesting) from a trigger:

```
select @@<trigger_name>
```

If nested triggers are fired, `@@<trigger_name>` holds the name of the most recently fired trigger.

The source text for each trigger is stored in `syscomments`. Execution plans for triggers are stored in `sysprocedures`. The system procedures described in the following sections provide information from the system tables about triggers.

21.12.1 sp_help

Get a report on a trigger using `sp_help`.

For example, you can get information on `deltitle` using:

```
sp_help deltitle
```

```
Name        Owner   Type
----------- ------- -----------
deltitle    dbo     trigger
Data_located_on_segment  When_created
-----------------------  -----------------
not applicable          Jul 10 1997 3:56PM
(return status = 0)
```

You can also use `sp_help` to report the status of a disabled or enabled trigger:

```
1> sp_help trig_insert
2> go
Name Owner
Type
------------------------------------
trig_insert dbo
trigger
(1 row affected)
data_located_on_segment When_created
------------------------------------
not applicable Aug 30 1998 11:40PM
Trigger enabled
(return status = 0)
```

21.12.2 sp_helptext

To display the source text of a trigger, execute `sp_helptext`.

```
sp_helptext deltitle
```

```
# Lines of Text
---------------
              1
text
---------------------------------------------
create trigger deltitle
on titles
for delete
as
if (select count(*) from deleted, salesdetail
where salesdetail.title_id = deleted.title_id) >0
begin
      rollback transaction
      print "You can’t delete a title with sales."
end
```

If the source text of a trigger was encrypted using `sp_hidetext`, SAP ASE displays a message advising you that the text is hidden. See the *Reference Manual: Procedures*.

If the system security officer has reset the `allow select on syscomments.text column` parameter with `sp_configure` (as required to run SAP ASE in the evaluated configuration), you must be the creator of the trigger or a system administrator to view the source text of a trigger through `sp_helptext`. See, *Introduction to Security*, in the *Security Administration Guide*.

21.12.3 sp_depends

`sp_depends` lists the triggers that reference an object, or all the tables or views that the trigger affects.

This example shows how to use `sp_depends` to get a list of all the objects referenced by the trigger `deltitle`:

```
sp_depends deltitle
```

```
Things the object references in the current database.
object              type       updated  selected
----------------    ----------  -------  --------
dbo.salesdetail     user table  no       no
dbo.titles          user table  no       no
(return status = 0)
```

This statement lists all the objects that reference the `salesdetail` table:

```
sp_depends salesdetail
```

```
Things inside the current database that reference the object.
object                        type
----------------------------  ----------------
dbo.deltitle                  trigger
dbo.history_proc              stored procedure
dbo.insert_salesdetail_proc   stored procedure
dbo.totalsales_trig           trigger
(return status = 0)
```

21.13 instead of Triggers

`instead of` triggers are special stored procedures that override the default action of a triggering statement (`insert`, `update`, and `delete`), and perform user-defined actions.

The `instead of` trigger, like the `for` trigger, executes each time a data modification statement executes on a specific view. A `for` trigger fires after an `insert/update/delete` statement on a table, and is sometimes called an `after` trigger. A single `instead of` trigger can apply to one specific triggering action:

```
instead of update
```

It can also apply to multiple actions, in which the same trigger executes all the actions listed:

```
instead of insert,update,delete
```

Like `for` triggers, `instead of` triggers use the logical inserted and deleted tables to store modified records while the trigger is active. Each column in these tables maps directly to a column in the base view referenced in the trigger. For example, if a view named V1 contains columns named C1, C2, C3, and C4, the `inserted` and `deleted` tables contain the values for all four columns, even if the trigger modifies only columns C1 and C3. SAP ASE automatically creates and manages the `inserted` and `deleted` tables as memory-resident objects.

`instead of` triggers allow views to support updates, and allow implementation of code logic that requires rejecting parts of a batch, while allowing other parts to succeed.

An `instead of` trigger is fired only once per data modification statement. A complex query containing a `while` loop may repeat an `update` or `insert` statement many times, firing the `instead of` trigger each time.

21.13.1 Inserted and Deleted Logical Tables

Deleted and inserted tables are logical (conceptual) tables. An inserted table is a pseudo-table containing rows with the inserted values of an `insert` statement, and a deleted table is a pseudo-table containing rows with the updated values (after image) of an `update` statement.

The schema of the inserted and deleted tables is identical to that of the view for which the `instead of` trigger is defined; that is, the view on which a user action is attempted. The difference between these tables and the view is that all the columns of inserted and deleted tables are nullable, even when the corresponding column of the view is not. For example, if the view has a column of datatype `char`, and an `insert` statement provides a `char` value to be inserted, the inserted table has a datatype `varchar` for the corresponding column, and the input value is converted to `varchar`. However, trailing blanks are not truncated from the value when the value is added to the inserted table.

When you specify a value of a datatype different from that of the column into which you are inserting it, the value is internally converted to the column datatype. If the conversion succeeds, the converted value is inserted into the table, but if the conversion fails, the statement is aborted. In this example, if a view selects an integer column from a table:

```
CREATE VIEW v1 AS SELECT intcol FROM t1
```

The following `insert` statement causes the `instead of` trigger on v1 to execute, because the value, 1.0, can be successfully converted to the integer value, 1:

```
INSERT INTO v1 VALUES (1.0)
```

However, this next statement causes an exception to be raised, and is aborted before the `instead of` trigger can execute:

```
INSERT INTO v1 VALUES (1.1)
```

The deleted and inserted tables can be examined by the trigger, to determine whether or how trigger action should be carried out, but the tables themselves cannot be altered by trigger action.

The deleted table is used with `delete` and `update`; the inserted table, with `insert` and `update`.

> **Note**
>
> `instead of` triggers create the inserted and deleted tables as in-memory tables or worktables. `for` triggers generate the rows of inserted and deleted tables on the fly, by reading the transactional log, `syslogs`.

LOB Datatypes

Values for columns in inserted or deleted tables that are large object (LOB) datatypes are stored in memory, and can lead to significant memory usage if there are many rows in these tables, and the rows contain large LOB values.

21.13.2 Triggers and Transactions

Both `rollback trigger` and `rollback transaction` are used for `instead of` triggers. `rollback trigger` rolls back the work done in any and all nested `instead of` triggers fired by the triggering statement. `rollback transaction` rolls back the work done in the entire transaction, up to the most recent `savepoint`.

21.13.3 Nesting

Like `for` triggers, you can nest `instead of` triggers to 16 levels. The current nesting level is stored in `<@@nestlevel>`.

A system administrator can use the configuration parameter `allow nested triggers` to turn trigger nesting on and off; by default nesting if on. If nested triggers are enabled, a trigger that changes a table containing another trigger executes the second trigger, which in turn can execute another trigger, and so forth, producing an infinite loop. In this case, processing ends when the nesting level is exceeded, and the trigger aborts. A rollback transaction in a trigger at any nesting level rolls back the effects of each trigger, and cancels the entire transaction. A rollback trigger affects only the nested triggers and the data modification statement that caused the initial trigger to execute.

You can interleave nesting `instead of` and `for` triggers. For example, an `update` statement on a view with an `instead of update` trigger causes the trigger to execute. If the trigger contains a SQL statement updating a table with a `for` trigger defined on it, that trigger fires. The `for` trigger may contain a SQL statement that updates another view with an `instead of` trigger that then executes, and so forth.

21.13.4 Recursion

`instead of` and `for` triggers have different recursive behaviors.

`for` triggers support recursion, while `instead of` triggers do not. If an `instead of` trigger references the same view on which the trigger was fired, the trigger is not called recursively. Rather, the triggering statement applies directly to the view; in other words, the statement is resolved as modifications against the base tables underlying the view. In this case, the view definition must meet all restrictions for an updatable view. If the view is not updatable, an error is raised.

For example, if a trigger is defined as an `instead of update` trigger for a view, the `update` statement executed against the same view within the `instead of` trigger does not cause the trigger to execute again. The update exercised by the trigger is processed against the view, as though the view did not have an `instead of` trigger. The columns changed by the update must be resolved to a single base table.

21.13.5 instead of insert Triggers

You can define `instead of insert` triggers on a view to replace the standard action of the `insert` statement. Usually, this trigger is defined on a view to insert data into one or more base tables.

Columns in the view `select` list can be nullable or not nullable. If a `view` column does not allow nulls, a SQL statement that inserts rows into the view must provide a value for the column. In addition to valid non-null values, an explicit value of null is also accepted for a non-nullable column of the view. `view` columns allow nulls if the expression defining the `view` column includes such items as:

- References to any base table column that allows nulls
- Arithmetic operators
- References to functions
- CASE with a nullable subexpression
- NULLIF

`sp_help` reports which `view` columns allow nulls.

An `insert` statement that references a view with an `instead of insert` trigger must supply values for every `view` column that does not allow nulls. This includes `view` columns that reference columns in the base table for which input values cannot be specified, such as:

- Computed columns in the base table
- Identity columns in the base table for which `identity insert` is `OFF`

If the `instead of insert` trigger contains an `insert` statement against the base table using data in the inserted table, the `insert` statement must ignore the values for these types of columns by not including them in the `select` list of the statement. The `insert` statement against the view can generate dummy values for these columns, but the `insert` statement in the `instead of insert` trigger ignores those values and SAP ASE supplies the correct values.

An `insert` statement must specify a value for a `view` column that maps to an identity or computed column in a base table. However, it can supply a placeholder value.

The `insert` statement in the `instead of` trigger that inserts the values into the base table is written to ignore the supplied value.

For example, these statements create a table, view, and trigger that illustrate the process:

```
CREATE TABLE BaseTable
(PrimaryKey         int IDENTITY
Color               varchar (10) NOT NULL,
Material            varchar (10) NOT NULL,
TranTime            timestamp
)
---------------------
--Create a view that contains all columns from the base table.
CREATE VIEW  InsteadView
AS SELECT PrimaryKey, Color, Material, TranTime
FROM BaseTable
---------------------
Create an INSTEAD OF INSERT trigger on the view.
CREATE TRIGGER InsteadTrigger on InsteadView
INSTEAD OF INSERT
AS
BEGIN
    --Build an INSERT statement ignoring
```

```
    --inserted.PrimaryKey and
    --inserted.TranTime.
INSERT INTO BaseTable
    SELECT Color, Material
    FROM inserted
END
```

An `insert` statement that refers directly to `BaseTable` cannot supply a value for the `PrimaryKey` and `TranTime` columns. For example:

```
--A correct INSERT statement that skips the PrimaryKey
--and TranTime columns.
INSERT INTO BaseTable (Color, Material)
            VALUES ('Red', 'Cloth')
--View the results of the INSERT statement.
SELECT PrimaryKey, Color, Material, TranTime
FROM BaseTable
--An incorrect statement that tries to supply a value
--for the PrimaryKey and TranTime columns.
INSERT INTO BaseTable
    VALUES (2, 'Green', 'Wood', 0x0102)
INSERT statements that refer to InsteadView, however,
    must supply a value for PrimaryKey:
--A correct INSERT statement supplying a dummy value for
--the PrimaryKey column. A value for TranTime is not
--required because it is a nullable column.
INSERT INTO InsteadView (PrimaryKey, Color, Material)
        VALUES (999, 'Blue', 'Plastic')
--View the results of the INSERT statement.
SELECT PrimaryKey, Color, Material, TranTime
FROM InsteadView
```

The `inserted` table passed to `InsteadTrigger` is built with a non-nullable `PrimaryKey` column; therefore, the `insert` statement referencing the view must supply a value for this column. The value 999 is passed in to `InsteadTrigger`, but the `insert` statement in `InsteadTrigger` does not select `inserted.PrimaryKey`, and therefore, the value is ignored. The row actually inserted into `BaseTable` has 2 in `PrimaryKey` and an SAP ASE-generated `timestamp` value in `TranTime`.

If a `not null` column with a `default` definition is referenced in a view with an `instead of insert` trigger, any `insert` statement that references the view must supply a value for the column. This value is required to build the `inserted` table passed to the trigger. A convention is required for a value that signals the trigger that

the default value should be used. A possible convention is to supply an explicit null value for the `not null` column in the `insert` statement. The `instead of insert` trigger can ignore the explicit null when inserting into the table upon which the view is defined, causing the default value to be inserted into the table. For example:

```
--Create a base table with a not null column that has
--a default
CREATE TABLE td1 (coll int DEFAULT 9 NOT NULL, col2 int)
--Create a view that contains all of the columns of
--the base table.
CREATE VIEW vtd1 as select * from td1
--create an instead of trigger on the view
CREATE TRIGGER vtd1insert on vtd1 INSTEAD OF INSERT AS
BEGIN
    --Build an INSERT statement that inserts all rows
    --from the inserted table that have a NOT NULL value
    --for col1.
    INSERT INTO td1 (col1,col2) SELECT * FROM inserted
    WHERE col != null
    --Build an INSERT statement that inserts just the
    --value of col2 from inserted for those rows that
    --have NULL as the value for col1 in inserted. In
    --this case, the default value of col1 will be
    --inserted.
    INSERT INTO td1 (col2) SELECT col2 FROM inserted
    WHERE col1 = null
END
```

The `deleted` table in an `instead of insert` trigger is always empty.

21.13.6 instead of update Trigger

Usually, the `instead of update` trigger is defined on a view, to modify data in one or more base tables.

In `update` statements that reference views with `instead of update` triggers, any subset of the columns in the view can appear in the `set` clause of the `update` statement, whether the columns in the subset are non-nullable columns or not.

Even view columns that can not be updated (those referencing columns in the base table for which input values cannot be specified) can appear in the `set` clause of the update statement. Columns you cannot update include:

- Computed columns in the base table
- Identity columns in the base table, for which `identity insert` is set to `off`
- Base table columns of the `timestamp` datatype

Usually, when an `update` statement that references a table attempts to set the value of a computed, `identity`, or `timestamp` column, an error is generated, because SAP ASE must generate the values for these columns. However, if the `update` statement references a view with an `instead of update` trigger, the logic defined in the trigger can bypass these columns and avoid the error. To do so, the `instead of update` trigger cannot update the values for the corresponding columns in the base table. To do this, exclude the columns in the `set` clause of the `update` statement from the definition of the trigger.

This solution works because an `instead of update` trigger does not need to process data from the inserted columns that are not updatable. The inserted table contains the values of columns not specified in the `set`

clause as they existed before the `update` statement was issued. The trigger can use an `if update` (column) clause to test whether a specific column has been updated.

`instead of update` triggers should use values supplied for computed, `identity`, or `timestamp` columns only in `where` clause search conditions. The logic that an `instead of update` trigger on a view uses to process updated values for computed, `identity`, `timestamp`, or default columns is the same as the logic applied to inserted values for these column types.

The inserted and deleted tables each contain a row for every row that qualifies to be updated, in an `update` statement on a view with an `instead of update` trigger. The rows of the inserted table contain the values of the columns after the update operation, and the rows of the deleted table contain the values of the columns before the update operation.

21.13.7 instead of delete Trigger

`instead of delete` triggers can replace the standard action of a `delete` statement. Usually, an `instead of delete` trigger is defined on a view to modify data in base tables.

`delete` statements do not specify modifications to existing data values, only the rows to be deleted. A `delete` table is a pseudotable containing rows with the deleted values of a `delete` statement, or the preupdated values (before image) of an `update` statement.The `deleted` table sent to an `instead of delete` trigger contains an image of the rows as they existed before the `delete` statement was issued. The format of the `deleted` table is based on the format of the `select` list defined for the view, with the exception that all `not null` columns be converted to nullable columns.

The `inserted` table passed to a `delete` trigger is always empty.

21.13.8 Searched and Positioned update and delete

You can search or position `update` and `delete` statements.

A searched `delete` contains an optional predicate expression in the `where` clause, that qualifies the rows to be deleted. A searched `update` contains an optional predicate expression in the `where` clause, that qualifies the rows to be updated.An example of a `searched delete` statement is:

```
DELETE myview WHERE myview.col1 > 5
```

This statement is executed by examining all the rows of `myview`, and applying the predicate (`myview.col1 > 5`) specified in the `where` clause to determine which rows should be deleted.

Joins are not allowed in searched `update` and `delete` statements. To use the rows of another table to find the qualifying rows of the view, use a subquery. For example, this statement is not allowed:

```
DELETE myview FROM myview, mytab
    where myview.col1 = mytab.col1
```

But the equivalent statement, using a subquery, is allowed:

```
DELETE myview WHERE col1 in (SELECT col1 FROM mytab)
```

Positioned `update` and `delete` statements are executed only on the result set of a cursor, and affect only a single row. For example:

```
DECLARE mycursor CURSOR FOR SELECT * FROM myview
OPEN mycursor
FETCH mycursor
```

```
DELETE myview WHERE CURRENT OF mycursor
```

The positioned `delete` statement deletes only the row of `myview` on which `mycursor` is currently positioned.

If an `instead of` trigger exists on a view, it always executes for a qualifying searched `delete` or `update` statement; that is, a statement without joins. For an `instead of` trigger to execute on a positioned `delete` or `update` statement, the following two conditions must be met:

- The `instead of` trigger exists when the cursor is declared; that is, when the command `declare cursor` is executed.
- The `select` statement that defines the cursor can access only the view; for example, the `select` statement contains no joins, but it can access any subset of the view columns.

The `instead of` trigger also executes when positioned `delete` or `update` statements are executed against scrollable cursors. However, `instead of` triggers do not fire in one case, when using a **client cursor** and the command `set cursor rows`.

Client Cursors

A client cursor is declared and fetched in an application using the Open Client library functions for cursor processing. The Open Client library functions can retrieve multiple rows from SAP ASE in a single `fetch` command, and buffer these rows, returning one row at a time to the application on subsequent `fetch` commands, without having to retrieve any more rows from SAP ASE until the buffered rows are all read. By default, SAP ASE returns a single row to the Open Client library functions for each `fetch` command it receives. However, the command `set cursor rows` can change the number of rows SAP ASE returns.

Positioned `update` and `delete` statements for client cursors, for which `set cursor rows` is not used to increase the number of rows returned per `fetch`, cause an `instead of` trigger to execute. However, if `set cursor rows` increases the number of rows returned per `fetch` command, an `instead of` trigger executes only if the cursor is not marked read-only during the internal processing of `declare cursor`. For example:

```
--Create a view that is read-only (without an instead
--of trigger) because it uses DISTINCT.
CREATE VIEW myview AS
SELECT DISTINCT (col1) FROM tab1
--Create an INSTEAD OF DELETE trigger on the view.
CREATE TRIGGER mydeltrig ON myview
INSTEAD OF DELETE
AS
BEGIN
    DELETE tab1 WHERE col1 in (SELECT col1 FROM deleted)
END
```

```
Declare a cursor to read the rows of the view
DECLARE cursor1 CURSOR FOR SELECT * FROM myview
OPEN cursor1
FETCH cursor1
--The following positioned DELETE statement will
--cause the INSTEAD OF TRIGGER, mydeltrig, to fire.
DELETE myview WHERE CURRENT OF cursor1
--Change the number of rows returned by ASE for
--each FETCH.
SET CURSOR ROWS 10 FOR cursor1
FETCH cursor1
--The following positioned DELETE will generate an
--exception with error 7732: "The UPDATE/DELETE WHERE
--CURRENT OF failed for cursor 'cursor1' because
--the cursor is read-only."
DELETE myview WHERE CURRENT OF cursor1
```

Using `set cursor rows` creates a disconnect between the position of the cursor in SAP ASE and in the application: SAP ASE is positioned on the last row of the 10 rows returned, but the application can be positioned on any one of the 10 rows, since the Open Client library functions buffer the rows and scroll through them without sending information to SAP ASE. Because SAP ASE cannot determine the position of cursor1 when the `positioned delete` statement is sent by the application, the Open Client library functions also send the values of a subset of the columns of the row where cursor1 is positioned in the application. These values are used to convert the `positioned delete` into a `searched delete` statement. This means that if the value of `col1` is 5 in the current row of `cursor1`, a clause such as `'where col1 = 5'` is used by SAP ASE to find the row.

When a `positioned delete` is converted to a `searched delete`, the cursor must be updatable, just as for cursors on tables and on views without `instead of` triggers. In the example above, the `select` statement that defines cursor1 is replaced by the `select` statement that defines `myview`:

```
DECLARE cursor1 CURSOR FOR SELECT * FROM myview
```

becomes:

```
DECLARE cursor1 CURSOR FOR SELECT DISTINCT (col1)
FROM tab1
```

Because of the `distinct` option in the `select` list, cursor1 is not updatable; in other words, it is read-only. This leads to the 7732 error when the positioned `delete` is processed.

If the cursor that results from replacing the view by its defining `select` statement is updatable, the `instead of` trigger fires, whether `set cursor rows` is used or not. Using `set cursor rows` does not prevent an `instead of` trigger from firing in other types of cursors SAP ASE supports.

21.13.9 Get Information About `instead of` Triggers

Information about `instead of` triggers is stored as it is for `for` triggers.

- The definition query tree for a trigger is stored in `sysprocedures`.
- Each trigger has an identification number (object ID), stored in a new row in `sysobjects`. The object ID of the view to which the trigger applies is stored in the `deltrig`, `instrig`, and `updtrig` columns of the `sysobjects` row for the trigger. The object ID of the trigger is stored in the `deltrig`, `instrig`, or `putrid` columns in the `sysobjects` row of the view to which the trigger applies.

- Use `sp_helptext` to display the text of a trigger stored in `syscomments`. If the system security officer has reset the column parameter `allow select on syscomments.text` with `sp_configure`, you must be the creator of the trigger or a system administrator to view the text of the trigger through `sp_helptext`.
- Use `sp_help` to obtain a report on a trigger. `sp_help` reports `instead of` trigger as the `<object_type>` of an `instead of` trigger.
- Use `sp_depends` to report on the views referenced by an `instead of` trigger. `sp_depends <view_name>` reports the trigger name and its type as `instead of` trigger.

22 In-Row Off-Row LOB

SAP ASE supports the storage of in-row LOB columns for `text`, `image`, and `unitext` datatypes when they are small, and subject to available space in the page.

When a LOB expands in size or its space is used for other in-row columns (such as those used for `varchar` and `varbinary` datatypes), SAP ASE seamlessly migrates the in-row LOB data to off-row storage, automatically replacing the data with an in-row text pointer.

You can use:

- `create table` to specify in-row storage of LOB columns
- `alter table` to perform modifications of how LOB columns are stored
- `create` or `alter database` commands to manage database-wide in-row lengths for LOB columns

When you map Component Integration Services (CIS) proxy tables to remote SAP ASE tables that contain in-row LOB columns, you must define the LOB columns in the proxy tables as off-row LOBs. All data transmissions occur as off-row LOB column data.

In versions earlier than 15.7, SAP ASE always stored large object (LOB) columns (such as `text`, `image`, unitext, and XML) off-row, using a text pointer (`txtptr`) to identify the start page ID value, and storing the pointer in the data row. This includs serialized Java classes, which are stored in-row only if they are shorter than a fixed maximum length.

22.1 In-Row LOB Columns Compression

If you define the LOB column for compression, the LOB compression extends to the data portion of the in-row LOBs when the LOB data is moved off-row, and the log data size exceeds the logical page size.

If you set up a table for row- or page-level compression, any in-row LOB columns (or their metadata) are not compressed.

As with off-row LOB columns, the data for different in-row LOB columns may be compressed by different LOB compression levels when the LOB data is moved off-row, however, SAP ASE does not compress the text pointer field for off-row LOB columns.

If you set the compression level for all LOB columns in a table using the `with lob_compression` clause, make sure you use the "`not compressed`" column-level clause to define individual in-row or other LOB columns you do not want to compress.

22.2 Migrate Off-Row LOB Data to In-Row Storage

Some data definition language (DDL) and utility operations that work on an entire table copy all table data by rewriting the data row to match that of the target schema.

There are several ways to convert LOBs to use in-row storage:

- `update set <column> = <column>`
- `alter table` schema changes such as `add not null`, `modify` datatype or nullability, or `drop column` – row contents and layout are reorganized to accommodate the schema change. The entire row is rebuilt. `alter table ... partition by` changes the partitioning schema by distributing the data rows to different partitions. Rows are reformatted as part of the data copy. This operation, however, does not change the table schema.
- `reorg rebuild` – causes rows to be rebuilt as part of the data movement.
- `bcp` bulk copy utility – supports tables that use in-row LOB columns.

You can migrate existing data using any of these methods, reducing space usage for the text pages and moving to in-row LOB storage.

Related Information

In-Row LOB Columns and Bulk Copy [page 621]

22.2.1 In-Row LOB Columns and Bulk Copy

The SAP ASE `bcp` utility supports tables that use in-row LOB columns, storing LOB columns in-row as long as the off-row data fits your defined in-row size, and the resulting row meets the page-size limitation.

In addition, `bcp in` handles character-set conversions for the `text` datatype when stored as in-row LOB in the same way the utility handles `char` and `varchar` datatype conversions.

This means that when server-side character-set conversion is active, SAP ASE rejects in-row LOB data if the required space after conversion differs from the original length. When this occurs, you see:

```
BCP insert operation is disabled when data size is
changing between client and server character sets.
Please use BCP's -Y option to invoke client-side
conversion.
```

For this reason, SAP recommends you use the `bcp -Y` option to force SAP ASE to perform character-set conversions on the client instead of on the server. Ensuring that the character data for both the client and server have the same length.

22.2.2 Methods for Migrating Existing Data

Since the results are all very similar, the migration method you choose depends on your situation and preference.

Each example creates a copy of the original table, `mymsgs`, using `select into`. The in-row length of the `description` column of the copied table is then altered. The method being illustrated is used to migrate the off-row LOB to in-row storage. Space usage is compared between the original table and the modified copy to show that LOB storage decreases substantially.

Set Up the mymsgs Example Table

This example creates the `mymsgs` table from the `pubs2` database, specifying `text` rather than `varchar` for the `description` column in preparation for migrating the column's contents from off-row to in-row storage.

```
1> use pubs2
2> go
1> exec sp_drop_object mymsgs, 'table'
2> go
1> create table mymsgs (
         error                int               not null
       , severity             smallint          not null
       , dlevel               smallint          not null
       , description          text
       , langid               smallint          null
       , sqlstate             varchar (5)       null
) lock datarows
2> go
1> insert mymsgs select * from master..sysmessages
2> go
```

```
(9564 rows affected)
```

```
1> exec sp_spaceusage display, 'table', mymsgs
2> go
```

```
All the page counts in the result set are in the unit 'KB'.
 OwnerName TableName IndId  NumRows UsedPages RsvdPages ExtentUtil ExpRsvdPages
PctBloatUsedPages  PctBloatRsvdPages
 --------- --------- ----- ------- --------- --------- ---------- ------------
---------------------- ------------------
 dbo       mymsgs       0    9564.0    372.0     384.0      96.87        320.0
           16.25             20.00
 dbo       mymsgs     255    NULL    19132.0   19140.0      99.95      19136.0
           00                 0.02
```

```
1> use pubs2
1>
2> dump tran pubs2 with no_log
1>
2> /* Drop the spaceusage stats table before each run */
3> exec sp_drop_object spaceusage_object, 'table'
```

```
Dropping table spaceusage_object
```

```
(return status = 0)
```

```
1>
2> exec sp_spaceusage archive, 'table', mymsgs
```

```
Data was successfully archived into table 'pubs2.dbo.spaceusage_object'.
(return status = 0)
```

Migrate Using Update Statement

This example uses `update set <column> = <column>` to migrate off-row LOBs to in-row storage.

The `select into` command first creates a copy of `mymsgs`, including its off-row data, into `mymsgs_test_upd`, moving the off-row data in-row in the process. You can then move the `update` command to relocate off-row LOBs to in-row storage:

```
1> exec sp_drop_object mymsgs_test_upd, 'table'
Dropping table mymsgs_test_upd
(return status = 0)
1>
2> select * into mymsgs_test_upd from mymsgs
(9564 rows affected)
1>
2> exec sp_spaceusage display, 'table', mymsgs_test_upd
All the page counts in the result set are in the unit 'KB'.
 OwnerName TableName      IndId NumRows UsedPages RsvdPages ExtentUtil ExpRsvdPag
es
    PctBloatUsedPages PctBloatRsvdPages
 --------- --------------- ----- ------- --------- --------- ----------
------------
    ----------------- -----------------
dbo       mymsgs_test_upd    0  9564.0     318.0     320.0     99.37         272.
0
                22.31             17.65
dbo       mymsgs_test_upd  255    NULL   19132.0   19136.0     99.97       19136.
0
                 0.00              0.00
(1 row affected)
(return status = 0)
```

The space usage of the `mymsgs_test_upd` is nearly the same as that of the `mymsgs` table. The off-row LOB consumes about 19KB of storage.

```
)
1> alter table mymsgs_test_upd modify description in row (300)
1> sp_spaceusage
2> go
2> update mymsgs_test_upd set description = description
(9564 rows affected)
1>
2> exec sp_spaceusage display, 'table', mymsgs_test_upd
All the page counts in the result set are in the unit 'KB'.
 OwnerName TableName      IndId NumRows UsedPages RsvdPages ExtentUtil ExpRsvdPag
es
    PctBloatUsedPages PctBloatRsvdPages
 --------- --------------- ----- ------- --------- --------- ----------
------------
    ----------------- -----------------
```

```
dbo       mymsgs_test_upd    0  9564.0    1246.0    1258.0
     99.04         272.0
               379.23            362.50
dbo       mymsgs_test_upd  255    NULL       6.0      32.0     18.75           16
.0
                 0.00            100.00
(1 row affected)
(return status = 0)
1>
2> exec sp_spaceusage archive, 'table', mymsgs_test_upd
Data was successfully archived into table 'pubs2.dbo.spaceusage_object'.
(return status = 0)
```

The size of `RsvdPages` for the data layer, `indid=0`, has changed; what used to be 320KB, is now 1258KB, while the reserved pages for the LOB column, `indid=255`, has decreased from 19136KB to around 32KB, showing that the off-row storage changed to in-row.

> **Note**
>
> If you have a very large table (for example, over a million rows), executing an `update` statement may take a very long time. If you use a `where` clause to select fewer rows at a time, make sure you use a key index to identify all the rows in the table to ensure that you do not miss any rows during the conversion.

Use reorg rebuild

This example uses `reorg rebuild` to rebuild rows as part of the data movement, rebuilding `mymsgs_test_reorg` so that it is capable of storing in-row LOBs.

```
1> exec sp_drop_object mymsgs_test_reorg, 'table'
Dropping table mymsgs_test_reorg
(return status = 0)
1>
2> select * into mymsgs_test_reorg from mymsgs
(9564 rows affected)
1> alter table mymsgs_test_reorg modify description in row (300)
1>
2> REORG REBUILD mymsgs_test_reorg
Beginning REORG REBUILD of table 'mymsgs_test_reorg'.
(9564 rows affected)
REORG REBUILD of table 'mymsgs_test_reorg' completed.
1>
2> exec sp_spaceusage display, 'table', mymsgs_test_reorg
All the page counts in the result set are in the unit 'KB'.
 OwnerName TableName      IndId NumRows UsedPages RsvdPages ExtentUtil ExpRsvdPag
es
    PctBloatUsedPages PctBloatRsvdPages
 --------- --------------- ----- ------- --------- --------- ----------
------------
    ----------------- -----------------
dbo       mymsgs_test_reorg   0  9564.0    1230.0    1242.0     99.03         272
.0
               373.08            356.62
dbo       mymsgs_test_reorg 255    NULL       6.0      32.0     18.75           16
.0
                 0.00              0.00
(1 row affected)
(return status = 0)
1>
2> exec sp_spaceusage archive, 'table', mymsgs_test_reorg
```

```
Data was successfully archived into table 'pubs2.dbo.spaceusage_object'.
(return status = 0)
```

Migrate Using alter table with Data Copy

This example uses `alter table`, dropping a column to add it back as a new column so that the row content, while essentially unchanged, modifies the `description` column.

The example shows how you can move LOB columns to in-row storage as a side-effect of an `alter table` schema change operation that might require a data copy (such as `drop` column, `add not null` column, and so on):

```
1> exec sp_drop_object mymsgs_test_alttab, 'table'
Dropping table mymsgs_test_alttab
(return status = 0)
1>
2> select * into mymsgs_test_alttab from mymsgs
(9564 rows affected)
1> alter table mymsgs_test_alttab modify description in row (300)
1>
2> alter table mymsgs_test_alttab
3> DROP dlevel
4> ADD newdlevel int default 0 not null
(9564 rows affected)
1>
2> exec sp_spaceusage display, 'table', mymsgs_test_alttab
Warning: Some output column values in the result set may be incorrect. Running
'update statistics' may help correct them.
All the page counts in the result set are in the unit 'KB'.
 OwnerName TableName      IndId NumRows UsedPages RsvdPages ExtentUtil ExpRsvdPag
es
    PctBloatUsedPages PctBloatRsvdPages
 --------- --------------- ----- ------- --------- --------- ----------
------------
    ----------------- -----------------
dbo       mymsgs_test_alttab   0 9564.0    1252.0    1258.0     99.52        1728
.0
               -27.46            -27.20
dbo       mymsgs_test_alttab 255   NULL       6.0      32.0     18.75          16
.0
                 0.00            100.00
(1 row affected)
(return status = 0)
1>
2> exec sp_spaceusage archive, 'table', mymsgs_test_alttab
Warning: Some output column values in the result set may be incorrect. Running
'update
statistics' may help correct them.
Data was successfully archived into table 'pubs2.dbo.spaceusage_object'.
(return status = 0)
```

This is the summary report, showing the space usage information for the tables used in the examples, and the significant decrease in space used. While the size of `RsvdPages` was 19140KB in the original `mymsgs` table for the LOB columns (`indid=255`), this space has decreased by over 95 percent in all three sample tables:

```
1> exec sp_spaceusage report, 'table', 'mymsgs%', 'OwnerName, TableName, IndId,
NumRows, RsvdPages, UsedPages, ExtentUtil'
All the page counts in the result set are in the unit 'KB'.
 OwnerName TableName          IndId NumRows RsvdPages UsedPages ExtentUtil
 --------- ------------------ ----- ------- --------- --------- ----------
```

```
 dbo       mymsgs                 0  9564.0     318.0     318.0     100.00
 dbo       mymsgs               255    NULL   19140.0   19132.0      99.95
 dbo       mymsgs_test_alttab     0  9564.0    1258.0    1252.0      99.52
 dbo       mymsgs_test_alttab   255    NULL      32.0       6.0      18.75
 dbo       mymsgs_test_reorg      0  9564.0    1242.0    1230.0      99.03
 dbo       mymsgs_test_reorg    255    NULL      32.0       6.0      18.75
 dbo       mymsgs_test_upd        0  9564.0    1258.0    1246.0      99.04
 dbo       mymsgs_test_upd      255    NULL      32.0       6.0      18.75
(1 row affected)
(return status = 0)
```

22.2.2.1 Set Up the mymsgs Example Table

An example that creates the `mymsgs` table from the `pubs2` database, specifying `text` rather than `varchar` for the `description` column in preparation for migrating the column's contents from off-row to in-row storage.

```
1> use pubs2
2> go
1> exec sp_drop_object mymsgs, 'table'
2> go
1> create table mymsgs (
          error                int               not null
        , severity             smallint          not null
        , dlevel               smallint          not null
        , description          text
        , langid               smallint          null
        , sqlstate             varchar (5)       null
) lock datarows
2> go
1> insert mymsgs select * from master..sysmessages
2> go
```

```
(9564 rows affected)
```

```
1> exec sp_spaceusage display, 'table', mymsgs
2> go
```

```
All the page counts in the result set are in the unit 'KB'.
 OwnerName TableName IndId  NumRows UsedPages RsvdPages ExtentUtil ExpRsvdPages
PctBloatUsedPages  PctBloatRsvdPages
 --------- --------- ----- ------- --------- --------- ---------- ------------
---------------------- ------------------
 dbo       mymsgs       0    9564.0    372.0     384.0      96.87        320.0
           16.25             20.00
 dbo       mymsgs     255    NULL    19132.0   19140.0      99.95      19136.0
           00                 0.02
```

```
1> use pubs2
1>
2> dump tran pubs2 with no_log
1>
2> /* Drop the spaceusage stats table before each run */
3> exec sp_drop_object spaceusage_object, 'table'
```

```
Dropping table spaceusage_object
```

```
(return status = 0)
```

```
1>
2> exec sp_spaceusage archive, 'table', mymsgs
```

```
Data was successfully archived into table 'pubs2.dbo.spaceusage_object'.
(return status = 0)
```

22.2.2.2 Migrate Using Update Statement

An example using `update set <column> = <column>` to migrate off-row LOBs to in-row storage.

The `select into` command first creates a copy of `mymsgs`, including its off-row data, into `mymsgs_test_upd`, moving the off-row data in-row in the process. You can then move the `update` command to relocate off-row LOBs to in-row storage:

```
1> exec sp_drop_object mymsgs_test_upd, 'table'
Dropping table mymsgs_test_upd
(return status = 0)
1>
2> select * into mymsgs_test_upd from mymsgs
(9564 rows affected)
1>
2> exec sp_spaceusage display, 'table', mymsgs_test_upd
All the page counts in the result set are in the unit 'KB'.
 OwnerName TableName      IndId NumRows UsedPages RsvdPages ExtentUtil ExpRsvdPag
es
    PctBloatUsedPages PctBloatRsvdPages
 --------- --------------- ----- ------- --------- --------- ----------
------------
    ----------------- -----------------
dbo       mymsgs_test_upd    0  9564.0     318.0     320.0     99.37         272.
0
                22.31             17.65
dbo       mymsgs_test_upd  255    NULL   19132.0   19136.0     99.97       19136.
0
                 0.00              0.00
(1 row affected)
(return status = 0)
```

The space usage of the `mymsgs_test_upd` is nearly the same as that of the `mymsgs` table. The off-row LOB consumes about 19KB of storage.

```
)
1> alter table mymsgs_test_upd modify description in row (300)
1> sp_spaceusage
2> go
2> update mymsgs_test_upd set description = description
(9564 rows affected)
1>
2> exec sp_spaceusage display, 'table', mymsgs_test_upd
All the page counts in the result set are in the unit 'KB'.
 OwnerName TableName      IndId NumRows UsedPages RsvdPages ExtentUtil ExpRsvdPag
es
    PctBloatUsedPages PctBloatRsvdPages
 --------- --------------- ----- ------- --------- --------- ----------
------------
    ----------------- -----------------
```

```
dbo       mymsgs_test_upd    0  9564.0    1246.0    1258.0
     99.04         272.0
               379.23            362.50
dbo       mymsgs_test_upd  255    NULL       6.0      32.0     18.75           16
.0
                 0.00            100.00
(1 row affected)
(return status = 0)
1>
2> exec sp_spaceusage archive, 'table', mymsgs_test_upd
Data was successfully archived into table 'pubs2.dbo.spaceusage_object'.
(return status = 0)
```

The size of `RsvdPages` for the data layer, `indid=0`, has changed; what used to be 320KB, is now 1258KB, while the reserved pages for the LOB column, `indid=255`, has decreased from 19136KB to around 32KB, showing that the off-row storage changed to in-row.

> **Note**
>
> If you have a very large table (for example, over a million rows), executing an `update` statement may take a very long time. If you use a `where` clause to select fewer rows at a time, make sure you use a key index to identify all the rows in the table to ensure that you do not miss any rows during the conversion.

22.2.2.3 Use reorg rebuild

An example using `reorg rebuild` to rebuild rows as part of the data movement, rebuilding `mymsgs_test_reorg` so that it is capable of storing in-row LOBs.

```
1> exec sp_drop_object mymsgs_test_reorg, 'table'
Dropping table mymsgs_test_reorg
(return status = 0)
1>
2> select * into mymsgs_test_reorg from mymsgs
(9564 rows affected)
1> alter table mymsgs_test_reorg modify description in row (300)
1>
2> REORG REBUILD mymsgs_test_reorg
Beginning REORG REBUILD of table 'mymsgs_test_reorg'.
(9564 rows affected)
REORG REBUILD of table 'mymsgs_test_reorg' completed.
1>
2> exec sp_spaceusage display, 'table', mymsgs_test_reorg
All the page counts in the result set are in the unit 'KB'.
 OwnerName TableName      IndId NumRows UsedPages RsvdPages ExtentUtil ExpRsvdPag
es
    PctBloatUsedPages PctBloatRsvdPages
 --------- --------------- ----- ------- --------- --------- ----------
------------
    ----------------- -----------------
dbo       mymsgs_test_reorg   0  9564.0    1230.0    1242.0     99.03         272
.0
               373.08            356.62
dbo       mymsgs_test_reorg 255    NULL       6.0      32.0     18.75           16
.0
                 0.00              0.00
(1 row affected)
(return status = 0)
1>
2> exec sp_spaceusage archive, 'table', mymsgs_test_reorg
```

```
Data was successfully archived into table 'pubs2.dbo.spaceusage_object'.
(return status = 0)
```

22.2.2.4 Migrate Using alter table with Data Copy

An example using `alter table`, dropping a column to add it back as a new column so that the row content, while essentially unchanged, modifies the `description` column.

The example shows how you can move LOB columns to in-row storage as a side-effect of an `alter table` schema change operation that might require a data copy (such as `drop` column, `add not null` column, and so on):

```
1> exec sp_drop_object mymsgs_test_alttab, 'table'
Dropping table mymsgs_test_alttab
(return status = 0)
1>
2> select * into mymsgs_test_alttab from mymsgs
(9564 rows affected)
1> alter table mymsgs_test_alttab modify description in row (300)
1>
2> alter table mymsgs_test_alttab
3> DROP dlevel
4> ADD newdlevel int default 0 not null
(9564 rows affected)
1>
2> exec sp_spaceusage display, 'table', mymsgs_test_alttab
Warning: Some output column values in the result set may be incorrect. Running
'update statistics' may help correct them.
All the page counts in the result set are in the unit 'KB'.
 OwnerName TableName      IndId NumRows UsedPages RsvdPages ExtentUtil ExpRsvdPag
es
    PctBloatUsedPages PctBloatRsvdPages
 --------- --------------- ----- ------- --------- --------- ----------
------------
    ----------------- -----------------
dbo       mymsgs_test_alttab   0 9564.0    1252.0    1258.0     99.52        1728
.0
               -27.46            -27.20
dbo       mymsgs_test_alttab 255   NULL       6.0      32.0     18.75          16
.0
                 0.00            100.00
(1 row affected)
(return status = 0)
1>
2> exec sp_spaceusage archive, 'table', mymsgs_test_alttab
Warning: Some output column values in the result set may be incorrect. Running
'update
statistics' may help correct them.
Data was successfully archived into table 'pubs2.dbo.spaceusage_object'.
(return status = 0)
```

This is the summary report, showing the space usage information for the tables used in the examples, and the significant decrease in space used. While the size of `RsvdPages` was 19140KB in the original `mymsgs` table for the LOB columns (`indid=255`), this space has decreased by over 95 percent in all three sample tables:

```
1> exec sp_spaceusage report, 'table', 'mymsgs%', 'OwnerName, TableName, IndId,
NumRows, RsvdPages, UsedPages, ExtentUtil'
All the page counts in the result set are in the unit 'KB'.
 OwnerName TableName          IndId NumRows RsvdPages UsedPages ExtentUtil
 --------- ------------------ ----- ------- --------- --------- ----------
```

```
 dbo       mymsgs                 0  9564.0     318.0     318.0     100.00
 dbo       mymsgs               255    NULL   19140.0   19132.0      99.95
 dbo       mymsgs_test_alttab     0  9564.0    1258.0    1252.0      99.52
 dbo       mymsgs_test_alttab   255    NULL      32.0       6.0      18.75
 dbo       mymsgs_test_reorg      0  9564.0    1242.0    1230.0      99.03
 dbo       mymsgs_test_reorg    255    NULL      32.0       6.0      18.75
 dbo       mymsgs_test_upd        0  9564.0    1258.0    1246.0      99.04
 dbo       mymsgs_test_upd      255    NULL      32.0       6.0      18.75
(1 row affected)
(return status = 0)
```

22.2.3 Guidelines for Selecting the In-Row LOB Length

The choice of the in-row LOB length affects the storage space used for the data pages, LOB pages, and the number of rows that can fit on a data page.

- Specifying an in-row LOB length greater than the logical page size is ineffective, as only LOB values that are smaller than a page size are considered for in-row storage. Conversely, specifying a very small in-row LOB value may move very few LOB columns in-row, and not yield the potential savings in LOB storage space.
- A typical in-row LOB length is somewhere between the range of the minimum data length of a LOB column and the logical page size. A large in-row length value may fill an entire data page with just one row, so a practical useful value lies close to the average data length of the off-row LOB column where the length is less than a page size.
- The choice of the in-row LOB length can also potentially affect scan performance for queries that return large numbers of rows, and that do not reference LOB columns. If very few rows fit on the data page due to large in-row LOB values, then the number of data pages scanned might be very big, thereby slowing query responses.
  Examine the data lengths in your tables to estimate the in-row LOB length so that more than just one or two rows fit on the page. Balance performance impacts against the reduced LOB storage.

22.2.4 Identifying In-Row LOB Length Selection

Identify in-row LOB length selection to estimate the different amounts of LOB storage savings.

Procedure

1. Identify the minimum, maximum and average data row size for the table with LOBs being stored off-row:

```
1> select i.minlen, t.datarowsize, i.maxlen
2> from sysindexes i, systabstats t
3> where i.id = object_id('DYNPSOURCE')
4>   and i.indid in (0, 1)
5>   and i.id = t.id
6> go
```

```
 minlen datarowsize                 maxlen
```

```
------  --------------------------- ------
 9                       105.000000    201
```

2. Compute the mininum, average, and maximum data lengths for the off-row column of interest:

```
1> select datalength(FIELDINFO) as fieldinfo_len into
       #dynpsource_FIELDINFO
2> from DYNPSOURCE
3> where datalength(FIELDINFO) < @@maxpagesize
4> go
```

```
(65771 rows affected)
```

```
1> select minlen = min(fieldinfo_len), avglen = avg(fieldinfo_len),
       maxlen = max(fieldinfo_len)
2> from #dynpsource_FIELDINFO
3> go
```

```
 minlen     avglen      maxlen
----------- ----------- -----------
        536        7608       16080
```

Out of a total of about 190,000 rows in the `DYPNSOURCE` table, approximately 65000 of them had the off-row LOB column `DYPNSOURCE` such that its data length was well within the logical page size of 16K.

By choosing an in-row LOB length that lies between the minlen or avglen in the above output, different numbers of off-row LOBs can be brought in-row, thereby providing different amounts of LOB storage savings.

22.3 Downgrading Tables Containing In-Row LOB Columns

You cannot downgrade an SAP ASE server that has any tables that are defined with an in-row LOB column, regardless of whether the table actually contains any data in that column.

If you must downgrade such an SAP ASE server, copy all data out (`bcp out`), then copy it back in (`bcp in`) on the affected tables.

23 Transactions: Maintain Data Consistency and Recovery

A transaction treats a set of Transact-SQL statements as a unit. Either all statements in the group are executed or no statements are executed.

SAP ASE automatically manages all data modification commands, including single-step change requests, as transactions. By default, each `insert`, `update`, and `delete` statement is considered a single transaction.

However, consider the following scenario: Lee must make a series of data retrievals and modifications to the `authors`, `titles`, and `titleauthors` tables. As she is doing so, Lil begins to update the `titles` table. Lil's updates may cause inconsistent results with the work that Lee is doing. To prevent this from happening, Lee can group her statements into a single transaction, which locks Lil out of the portions of the tables that Lee is working on. This allows Lee to complete her work based on accurate data. After she completes her table updates, Lil's updates can take place.

Use these commands to create transactions:

- `begin transaction` – marks the beginning of the transaction block. The syntax is:

  ```
  begin {transaction | tran} [<transaction_name>]
  ```

  `<transaction_name>` is the name assigned to the transaction, which must conform to the rules for identifiers. Use transaction names only on the outermost pair of nested `begin/commit` or `begin/rollback` statements.
- `save transaction` – marks a savepoint within a transaction:

  ```
  save {transaction | tran} <savepoint_name>
  ```

  `<savepoint_name>` is the name assigned to the savepoint, which must conform to the rules for identifiers.
- `commit` – commits the entire transaction:

  ```
  commit [transaction | tran | work]
  [<transaction_name>]
  ```
- `rollback` – rolls a transaction back to a savepoint or to the beginning of a transaction:

  ```
  rollback [transaction | tran | work]
  [<transaction_name >|< savepoint_name>]
  ```

For example, Lee wants to change the royalty split for two authors of *The Gourmet Microwave*. Since the database would be inconsistent between the two updates, they must be grouped into a transaction, as shown in the following example:

```
begin transaction royalty_change

update titleauthor
set royaltyper = 65
from titleauthor, titles
where royaltyper = 75
and titleauthor.title_id = titles.title_id
```

```
and title = "The Gourmet Microwave"

update titleauthor
set royaltyper = 35
from titleauthor, titles
where royaltyper = 25
and titleauthor.title_id = titles.title_id
and title = "The Gourmet Microwave"

save transaction percentchanged

/* After updating the royaltyper entries for
** the two authors, insert the savepoint
** percentchanged, then determine how a 10%
** increase in the book's price would affect
** the authors' royalty earnings. */

update titles
set price = price * 1.1
where title = "The Gourmet Microwave"

select (price * total_sales) * royaltyper
from titles, titleauthor
where title = "The Gourmet Microwave"
and titles.title_id = titleauthor.title_id

/* The transaction is rolled back to the savepoint
** with the rollback transaction command. */

rollback transaction percentchanged

commit transaction
```

Transactions allow SAP ASE to guarantee:

- Consistency – simultaneous queries and change requests cannot collide with each other, and users never see or operate on data that is partially through a change.
- Recovery – in case of system failure, database recovery is complete and automatic.

To support SQL-standards-compliant transactions, SAP ASE allows you to select the mode and isolation level for your transactions. Applications that require transactions to be compliant with SQL standards should set those options at the beginning of every session.

23.1 Transactions and Consistency

In a multiuser environment, SAP ASE must prevent simultaneous queries and data modification requests from interfering with each other. If the data being processed by a query can be changed by another user's update, the results of the query may be ambiguous.

SAP ASE automatically sets the appropriate level of locking for each transaction. You can make shared locks more restrictive on a query-by-query basis by including the `holdlock` keyword in a `select` statement.

23.2 Transactions and Recovery

A transaction is both a unit of work and a unit of recovery. Because SAP ASE handles single-step change requests as transactions, the database can be recovered completely in case of failure.

The SAP ASE recovery time is measured in minutes and seconds. You can specify the maximum acceptable recovery time.

> **Note**
>
> Grouping large numbers of Transact-SQL commands into one long-running transaction may affect recovery time. If SAP ASE fails before the transaction commits, recovery takes longer, because SAP ASE must undo the transaction.

If you are using a remote database with Component Integration Services, there are a few differences in the way transactions are handled. See the *Component Integration Services User's Guide*.

If you have purchased and installed SAP ASE DTM features, transactions that update data in multiple servers can also benefit from transactional consistency. See *Using SAP ASE Distributed Transaction Features*.

Related Information

Backup and Recovery of Transactions [page 659]

23.3 Transaction Usage

The `begin transaction` and `commit transaction` commands tell SAP ASE to process any number of individual commands as a single unit. `rollback transaction` undoes the transaction, either back to its beginning, or back to a savepoint. Use `save transaction` to define a savepoint inside a transaction.

In addition to grouping SQL statements to behave as a single unit, transactions improve performance, since system overhead is incurred once per transaction, rather than once for each individual command.

Any user can define a transaction. No permission is required for any of the transaction commands.

23.3.1 Allow Data Definition Commands in Transactions

You can use certain data definition language commands, such as `create table`, `grant`, and `alter table`, in transactions by setting the `ddl in tran` database option to true.

If `ddl in tran` is true in the `model` database, you can issue the commands inside transactions in all databases created after `ddl in tran` was set to true in `model`. To check the current settings of `ddl in tran`, use `sp_helpdb`.

> **Caution**
>
> Use data definition commands with caution. The only scenario in which using data definition language commands inside transactions is justified is in `create schema`. Data definition language commands hold locks on system tables such as `sysobjects`. If you use data definition language commands inside transactions, keep the transactions short.
>
> Avoid using data definition language commands on `tempdb` within transactions; doing so can slow performance to a halt. Always leave `ddl in tran` set to false in `tempdb`.

To set `ddl in tran` to true, enter:

```
sp_dboption <database_name>,"ddl in tran", true
```

Then execute the `checkpoint` command in that database.

The first parameter specifies the name of the database in which to set the option. You must be using the `master` database to execute `sp_dboption`. Any user can execute `sp_dboption` with no parameters to display the current option settings. To set options, however, you must be either a system administrator or the database owner.

These commands are allowed inside a transaction only if the `ddl in tran` option to `sp_dboption` is set to true:

- `create default`
- `create index`
- `create procedure`
- `create rule`
- `create schema`
- `create table`
- `create trigger`
- `create view`
- `drop default`
- `drop index`
- `drop procedure`
- `drop rule`
- `drop table`
- `drop trigger`
- `drop view`
- `grant`
- `revoke`

You cannot use system procedures that change the `master` database or create temporary tables inside transactions.

Do not use these commands inside a transaction:

- `alter database`
- `alter table...partition`
- `alter table...unpartition`

- create database
- disk init
- dump database
- dump transaction
- drop database
- load transaction
- load database
- reconfigure
- select into
- update statistics
- truncate table

23.3.2 System Procedures That Are Not Allowed in Transactions

You cannot use certain system procedures within transactions.

- sp_helpdb, sp_helpdevice, sp_helpindex, sp_helpjoins, sp_helpserver, sp_lookup, and sp_spaceused (because they create temporary tables)
- sp_configure
- System procedures that change the master database

23.3.3 Begin and Commit Transactions

The begin transaction and commit transaction commands can enclose any number of SQL statements and stored procedures.

The syntax for both statements is:

```
begin {transaction | tran} [<transaction_name>]
```

```
commit {transaction | tran | work} [<transaction_name>]
```

where <transaction_name> is the name assigned to the transaction, which must conform to the rules for identifiers.

The keywords transaction, tran, and work (in commit transaction) are synonymous; you can use them interchangeably. However, transaction and tran are Transact-SQL extensions; only work is compliant with SQL-standards.

For example:

```
begin tran
    <statement>
    <procedure>
    <statement>
```

```
commit tran
```

`commit transaction` does not affect SAP ASE if the transaction is not currently active.

An example showing how you might specify a transaction:

```
begin transaction royalty_change
/* A user sets out to change the royalty split */
/* for the two authors of The Gourmet Microwave. */
/* Since the database would be inconsistent */
/* between the two updates, they must be grouped */
/* into a transaction. */
update titleauthor
set royaltyper = 65
from titleauthor, titles
where royaltyper = 75
and titleauthor.title_id = titles.title_id
and title = "The Gourmet Microwave"
update titleauthor
set royaltyper = 35
from titleauthor, titles
where royaltyper = 25
and titleauthor.title_id = titles.title_id
and title = "The Gourmet Microwave"
save transaction percent_changed
/* After updating the royaltyper entries for */
/* the two authors, the user inserts the */
/* savepoint "percent_changed," and then checks */
/* to see how a 10 percent increase in the */
/* price would affect the authors' royalty */
/* earnings. */
update titles
set price = price * 1.1
where title = "The Gourmet Microwave"
select (price * royalty * total_sales) * royaltyper
from titles, titleauthor, roysched
where title = "The Gourmet Microwave"
and titles.title_id = titleauthor.title_id
and titles.title_id = roysched.title_id
rollback transaction percent_changed
/* The transaction rolls back to the savepoint */
/* with the rollback transaction command. */
/* Without a savepoint, it would roll back to */
/* the begin transaction. */
commit transaction
```

23.3.4 Roll Back and Save Transactions

If you must cancel a transaction before it commits—either because of some failure or because of a change by the user—you must undo all of its completed statements or procedures.

You can cancel or roll back a transaction with the `rollback transaction` command any time before `commit transaction` has been given. Using savepoints, you can cancel either an entire transaction or part of it. However, you cannot cancel a transaction after it has been committed.

The syntax of `rollback transaction` is:

```
rollback {transaction | tran | work}
     [<transaction_name> | <savepoint_name>]
```

A savepoint is a marker that a user puts inside a transaction to indicate a point to which it can be rolled back. You can commit only certain portions of a batch by rolling back the undesired portion to a savepoint before committing the entire batch.

Insert a savepoint by placing a `save transaction` command in the transaction:

```
save {transaction | tran} <savepoint_name>
```

The savepoint name must conform to the rules for identifiers.

If no `<savepoint_name>` or `<transaction_name>` is given with `rollback transaction`, the transaction is rolled back to the first `begin transaction` in a batch.

Here is how you can use the `save transaction` and `rollback transaction` commands:

`begin tran`	
`<statements>`	Group A
`save tran mytran`	
`<statements>`	Group B
`rollback tran mytran`	Rolls back group B
`<statements>`	Group C
`commit tran`	Commits groups A and C

Until you issue a `commit transaction`, SAP ASE considers all subsequent statements to be part of the transaction, unless it encounters another `begin transaction` statement. At that point, SAP ASE considers all subsequent statements to be part of the new, nested transaction.

`rollback transaction` or `save transaction` does not affect SAP ASE and does not return an error message if the transaction is not currently active.

You can also use `save transaction` to create transactions in stored procedures or triggers in such a way that they can be rolled back without affecting batches or other procedures. For example:

```
create proc myproc as
begin tran
save tran mytran
<statements>
if ...
    begin
        rollback tran mytran
    /*
    ** Rolls back to savepoint.
    */
        commit tran
    /*
    ** This commit needed; rollback to a savepoint
    ** does not cancel a transaction.
    */
   end
else
commit tran
    /*
```

```
    ** Matches begin tran; either commits
    ** transaction (if not nested) or
    ** decrements nesting level.
    */
```

Unless you are rolling back to a savepoint, use transaction names only on the outermost pair of `begin/commit` or `begin/rollback` statements.

> **Caution**
>
> Transaction names are ignored, or can cause errors, when used in nested transaction statements. If you are using transactions in stored procedures or triggers that could be called from within other transactions, do not use transaction names.

Related Information

Nested Transactions [page 641]

23.3.5 Transaction States

The global variable `<@@transtate>` keeps track of the current state of a transaction. SAP ASE determines what state to return by keeping track of any transaction changes after a statement executes.

@@transtate Value	Meaning
0	Transaction in progress. A transaction is in effect; the previous statement executed successfully.
1	Transaction succeeded. The transaction completed and committed its changes.
2	Statement aborted. The previous statement was aborted; no effect on the transaction.
3	Transaction aborted. The transaction aborted and rolled back any changes.

SAP ASE does not clear `<@@transtate>` after every statement. In a transaction, you can use `<@@transtate>` after a statement (such as an `insert`) to determine whether it was successful or aborted, and to determine its effect on the transaction. The following example checks `<@@transtate>` during a transaction (after a successful `insert`) and after the transaction commits:

```
begin transaction
```

```
insert into publishers (pub_id) values ("9999")
```

```
(1 row affected)
```

```
select @@transtate
```

```
----------
         0

(1 row affected)
```

```
commit transaction
select @@transtate
```

```
----------
         1

(1 row affected)
```

The next example checks `<@@transtate>` after an unsuccessful `insert` (due to a rule violation) and after the transaction rolls back:

```
begin transaction
insert into publishers (pub_id) values ("7777")
```

```
Msg 552, Level 16, State 1:
A column insert or update conflicts with a rule bound to the column. The command
is aborted. The conflict occured in database 'pubs2', table 'publishers',
rule 'pub_idrule', column 'pub_id'.
```

```
select @@transtate
```

```
----------
        2

(1 row affected)
```

```
rollback transaction
select @@transtate
```

```
----------
        3

(1 row affected)
```

SAP ASE changes `<@@transtate>` only in response to an action taken by a transaction. Syntax and compile errors do not affect the value of `<@@transtate>`.

23.3.6 Nested Transactions

You can nest transactions within other transactions. When you nest `begin transaction` and `commit transaction` statements, the outermost pair actually begin and commit the transaction. The inner pairs only track the nesting level.

SAP ASE does not commit the transaction until the `commit transaction` that matches the outermost `begin transaction` is issued. Normally, this transaction "nesting" occurs as stored procedures or triggers that contain `begin/commit` pairs call each other.

The `<@@trancount>` global variable keeps track of the current nesting level for transactions. An initial implicit or explicit `begin transaction` sets `<@@trancount>` to 1. Each subsequent `begin transaction` increments `<@@trancount>`, and a `commit transaction` decrements it. Firing a trigger also increments `<@@trancount>`, and the transaction begins with the statement that causes the trigger to fire. Nested transactions are not committed unless `<@@trancount>` equals 0.

For example, the following nested groups of statements are not committed by SAP ASE until the final `commit transaction`:

```
begin tran
    select @@trancount
    /* @@trancount = 1 */
    begin tran
        select @@trancount
        /* @@trancount = 2 */
        begin tran
            select @@trancount
            /* @@trancount = 3 */
        commit tran
    commit tran
commit tran
select @@trancount
/* @@ trancount = 0 */
```

When you nest a `rollback transaction` statement without including a transaction or savepoint name, it rolls back to the outermost `begin transaction` statement and cancels the transaction.

23.3.7 Example of a Transaction

An example showing how you might specify a transaction.

```
begin transaction royalty_change
/* A user sets out to change the royalty split */
/* for the two authors of The Gourmet Microwave. */
/* Since the database would be inconsistent */
/* between the two updates, they must be grouped */
/* into a transaction. */
update titleauthor
set royaltyper = 65
from titleauthor, titles
where royaltyper = 75
and titleauthor.title_id = titles.title_id
and title = "The Gourmet Microwave"
update titleauthor
set royaltyper = 35
```

```
from titleauthor, titles
where royaltyper = 25
and titleauthor.title_id = titles.title_id
and title = "The Gourmet Microwave"
save transaction percent_changed
/* After updating the royaltyper entries for */
/* the two authors, the user inserts the */
/* savepoint "percent_changed," and then checks */
/* to see how a 10 percent increase in the */
/* price would affect the authors' royalty */
/* earnings. */
update titles
set price = price * 1.1
where title = "The Gourmet Microwave"
select (price * royalty * total_sales) * royaltyper
from titles, titleauthor, roysched
where title = "The Gourmet Microwave"
and titles.title_id = titleauthor.title_id
and titles.title_id = roysched.title_id
rollback transaction percent_changed
/* The transaction rolls back to the savepoint */
/* with the rollback transaction command. */
/* Without a savepoint, it would roll back to */
/* the begin transaction. */
commit transaction
```

23.4 Transaction Mode and Isolation Level

SAP ASE provides certain options that support transactions that must be compliant with SQL standards.

- The transaction mode lets you set whether transactions begin with or without an implicit `begin transaction` statement.
- The isolation level refers to the degree to which data can be accessed by other users during a transaction.

Set these options at the beginning of every session that requires SQL-standards-compliant transactions.

23.4.1 Choose a Transaction Mode

SAP ASE supports chained and unchained transaction modes. You can set either mode using the `chained` option of the `set` command.

The transaction modes supported are:

- chained mode implicitly begins a transaction before any data-retrieval or modification statement: `delete`, `insert`, `open`, `fetch`, `select`, and `update`. You must still explicitly end the transaction with `commit transaction` or `rollback transaction`.
- The default mode, called unchained mode or Transact-SQL mode, requires explicit `begin transaction` statements paired with `commit transaction` or `rollback transaction` statements to complete the transaction.

Do not mix these transaction modes in your applications. The behavior of stored procedures and triggers can vary, depending on the mode, and you may require special action to run a procedure in one mode that was created in the other.

The SQL standards require every SQL data-retrieval and data-modification statement to occur inside a transaction, using chained mode. A transaction automatically starts with the first data-retrieval or data-modification statement after the start of a session or after the previous transaction commits or aborts. This is the chained transaction mode.

You can set this mode for your current session by turning on the `chained` option of the `set` statement:

However, you cannot execute the `set chained` command within a transaction. To return to the unchained transaction mode, set the `chained` option to off.

The following group of statements produce different results, depending on which mode you use:

```
insert into publishers
    values ("9906", null, null, null)
begin transaction
delete from publishers where pub_id = "9906"
rollback transaction
```

In unchained transaction mode, `rollback` affects only the `delete` statement, so `publishers` still contains the inserted row. In chained mode, the `insert` statement implicitly begins a transaction, and the rollback affects all statements up to the beginning of that transaction, including the `insert`.

All application programs and ad hoc user queries should address the correct transaction mode. The transaction mode you use depends on whether or not a particular query or application requires compliance to the SQL standards. Applications that use chained transactions (for example, the Embedded SQL precompiler) should set chained mode at the beginning of each session.

23.4.1.1 Transaction Modes and Nested Transactions

Although chained mode implicitly begins transactions with data-retrieval or modification statements, you can nest transactions only by explicitly using `begin transaction` statements.

Once the first transaction implicitly begins, further data-retrieval or modification statements no longer begin transactions until after the first transaction commits or aborts. For example, in the following query, the first `commit transaction` commits all changes in chained mode; the second commit is unnecessary:

```
insert into publishers
    values ("9907", null, null, null)
    insert into publishers
        values ("9908", null, null, null)
    commit transaction
commit transaction
```

> **Note**
>
> In chained mode, a data-retrieval or modification statement begins a transaction whether or not it executes successfully. Even a `select` that does not access a table begins a transaction.

23.4.1.2 Find the Status of the Current Transaction Mode

The global variable `<@@tranchained>` to determine SAP ASE's current transaction mode. `select <@@tranchained>` returns 0 for unchained mode or 1 for chained mode.

23.4.2 Choose an Isolation Level

The ANSI SQL standard defines four levels of isolation for transactions. Each isolation level specifies the kinds of actions that are not permitted while concurrent transactions are executing. Higher levels include the restrictions imposed by the lower levels.

- Level 0 – ensures that data written by one transaction represents the actual data. Level 0 prevents other transactions from changing data that has already been modified (through an `insert`, `delete`, `update`, and so on) by an uncommitted transaction. The other transactions are blocked from modifying that data until the transaction commits. However, other transactions can still read the uncommitted data, which results in dirty reads. 8.4
- Level 1 – prevents dirty reads. Such reads occur when one transaction modifies a row, and a second transaction reads that row before the first transaction commits the change. If the first transaction rolls back the change, the information read by the second transaction becomes invalid. Level 1 is the default isolation level supported by SAP ASE.
- Level 2 – prevents nonrepeatable reads, which occur when one transaction reads a row and a second transaction modifies that row. If the second transaction commits its change, subsequent reads by the first transaction yield different results than the original read.
  SAP ASE supports level 2 for data-only-locked tables. It is not supported for allpages-locked tables.
- Level 3 – ensures that data read by one transaction is valid until the end of that transaction, preventing phantom rows. SAP ASE supports this level through the `holdlock` keyword of the `select` statement, which applies a read-lock on the specified data. Phantom rows occur when one transaction reads a set of rows that satisfy a search condition, and then a second transaction modifies the data (through an `insert`, `delete`, `update`, and so on). If the first transaction repeats the read with the same search conditions, it obtains a different set of rows.

You can set the isolation level for your session by using the `transaction isolation level` option of the `set` command. You can enforce the isolation level for only a single query as opposed to using the `at isolation` clause of the `select` statement. For example:

```
set transaction isolation level 0
```

23.4.2.1 Default Isolation Levels for SAP ASE and ANSI SQL

By default, the SAP ASE transaction isolation level is 1. The ANSI SQL standard requires level 3 to be the default isolation for all transactions. This prevents dirty reads, nonrepeatable reads, and phantom rows.

To enforce this default level of isolation, Transact-SQL provides the `transaction isolation level 3` option of the `set` statement. This option instructs SAP ASE to apply a `holdlock` to all `select` operations in a transaction. For example:

```
set transaction isolation level 3
```

Applications that use `transaction isolation level 3` should set that isolation level at the beginning of each session. However, setting `transaction isolation level 3` causes SAP ASE to hold any read locks for the duration of the transaction. If you also use the chained transaction mode, that isolation level remains in effect for any data-retrieval or modification statement that implicitly begins a transaction. In both cases, this can lead to concurrency problems for some applications, since more locks may be held for longer periods of time.

To return your session to the SAP ASE default isolation level:

```
set transaction isolation level 1
```

23.4.2.2 Dirty Reads

Applications that are not impacted by dirty reads may have better concurrency and reduced deadlocks when accessing the same data if you set `transaction isolation level 0` at the beginning of each session.

An example is an application that finds the momentary average balance for all savings accounts stored in a table. Since it requires only a snapshot of the current average balance, which probably changes frequently in an active table, the application should query the table using isolation level 0. Other applications that require data consistency, such as deposits and withdrawals to specific accounts in the table, should avoid using isolation level 0.

Scans at isolation level 0 do not acquire any read locks, so they do not block other transactions from writing to the same data, and vice versa. However, even if you set your isolation level to 0, utilities (like `dbcc`) and data modification statements (like `update`) still acquire read locks for their scans, because they must maintain the database integrity by ensuring that the correct data has been read before modifying it.

Because scans at isolation level 0 do not acquire any read locks, the result set of a level 0 scan may change while the scan is in progress. If the scan position is lost due to changes in the underlying table, a unique index is required to restart the scan. In the absence of a unique index, the scan may be aborted.

By default, a unique index is required for a level 0 scan on a table that does not reside in a read-only database. You can override this requirement by forcing SAP ASE to choose a nonunique index or a table scan, as follows:

```
select * from <table_name> (index <table_name>)
```

Activity on the underlying table may abort the scan before completion.

23.4.2.3 Repeatable Reads

A transaction performing repeatable reads locks all rows or pages read during the transaction. After one query in the transaction has read rows, no other transaction can update or delete the rows until the repeatable-reads transaction completes.

Repeatable-reads transactions do not provide phantom protection by performing range locking, as serializable transactions do. Other transactions can insert values that can be read by the repeatable-reads transaction and can update rows so that they match the search criteria of the repeatable-reads transaction.

A transaction performing repeatable reads locks all rows or pages read during the transaction. After one query in the transaction has read rows, no other transaction can update or delete the rows until the repeatable reads transaction completes. However, repeatable-reads transactions do not provide phantom protection by performing range locking, as serializable transactions do. Other transactions can insert values that can be read by the repeatable-reads transaction and can update rows so that they match the search criteria of the repeatable-reads transaction.

> **Note**
>
> Transaction isolation level 2 is supported only in data-only-locked tables. If you use transaction isolation level 2 (repeatable reads) on allpages-locked tables, isolation level 3 (serializable reads) is also enforced.

To enforce repeatable reads at a session level, use:

```
set transaction isolation level 2
```

or:

```
set transaction isolation level repeatable read
```

To enforce transaction isolation level 2 from a query, use:

```
select title_id, price, advance
from titles
at isolation 2
```

or:

```
select title_id, price, advance
from titles
at isolation repeatable read
```

Transaction isolation level 2 is supported only at the transaction level. You cannot use the `at isolation` clause in a `select` or `readtext` statement to set the isolation level of a query to 2.

Related Information

Change the Isolation Level for a Query [page 647]

23.4.2.4 Find the Status of the Current Isolation Level

The global variable <@@isolation> contains the current isolation level of your Transact-SQL session. Querying <@@isolation> returns the value of the active level (0, 1, or 3).

For example:

```
select @@isolation
```

```
--------
       1
```

23.4.2.5 Change the Isolation Level for a Query

Change the isolation level for a query by using the `at isolation` clause with the `select` or `readtext` statements.

The `at isolation` clause supports isolation levels 0, 1, and 3. It does not support isolation level 2. The `read uncommitted`, `read committed`, and `serializable` options support these isolation levels:

at isolation **Option**	**Isolation Level**
`read uncommited`	0
`read committed`	1
`serializable`	3

For example, the following two statements query the same table at isolation levels 0 and 3, respectively:

```
select *
from titles
at isolation read uncommitted
select *
from titles
at isolation serializable
```

The `at isolation` clause is valid only for single `select` and `readtext` queries or in the `declare cursor` statement. SAP ASE returns a syntax error if you use `at isolation`:

- With a query using the `into` clause
- Within a subquery
- With a query in the `create view` statement
- With a query in the `insert` statement
- With a query using the `for browse` clause

If there is a `union` operator in the query, you must specify the `at isolation` clause after the last `select`.

The SQL-92 standard defines `read uncommitted`, `read committed`, and `serializable` as options for `at isolation` and `set transaction isolation level`. A Transact-SQL extension also allows you to

specify 0, 1, or 3, but not 2, for `at isolation`. To simplify the discussion of isolation levels, the `at isolation` examples in this manual do not use this extension.

You can also enforce isolation level 3 using the `holdlock` keyword of the `select` statement. However, you cannot specify `noholdlock` or `shared` in a query that also specifies `at isolation read uncommitted`. (If you specify `holdlock` and isolation level 0 in a query, SAP ASE issues a warning and ignores the `at isolation` clause.) When you use different ways to set an isolation level, the `holdlock` keyword takes precedence over the `at isolation` clause (except for isolation level 0), and `at isolation` takes precedence over the session level defined by `set transaction isolation level`.

See the *Performance and Tuning Series: Locking and Concurrency Control*.

23.4.2.6 Isolation Level Precedences

Precedence rules apply to different methods of defining isolation levels.

1. The `holdlock`, `noholdlock`, and `shared` keywords take precedence over the `at isolation` clause and `set transaction isolation level` option, except in the case of isolation level 0. For example:

   ```
   /* This query executes at isolation level 3 */
   select *
       from titles holdlock
       at isolation read committed
   create view authors_nolock
       as select * from authors noholdlock
   set transaction isolation level 3
   /* This query executes at isolation level 1 */
   select * from authors_nolock
   ```

2. The `at isolation` clause takes precedence over the `set transaction isolation level` option. For example:

   ```
   set transaction isolation level 2
   /* executes at isolation level 0 */
   select * from publishers
       at isolation read uncommitted
   ```

   You cannot use the `read uncommitted` option of `at isolation` in the same query as the `holdlock`, `noholdlock`, and `shared` keywords.

3. The `transaction isolation level 0` option of the `set` command takes precedence over the `holdlock`, `noholdlock`, and `shared` keywords. For example:

   ```
   set transaction isolation level 0
   /* executes at isolation level 0 */
   select *
       from titles holdlock
   ```

   SAP ASE issues a warning before executing the above query.

23.4.2.7 Cursors and Isolation Levels

SAP ASE provides three isolation levels for cursors.

- Level 0 – SAP ASE uses no locks on base table pages that contain a row representing a current cursor position. Cursors acquire no read locks for their scans, so they do not block other applications from accessing the same data. However, cursors operating at this isolation level are not updatable, and they require a unique index on the base table to ensure the accuracy of their scans.
- Level 1 – SAP ASE uses a shared or update lock on base table pages that contain a row representing a current cursor position. The page remains locked until the current cursor position moves off the page (as a result of `fetch` statements), or the cursor is closed. If an index is used to search the base table rows, it also applies shared or update locks to the corresponding index pages. This is the default locking behavior for SAP ASE.
- Level 3 – SAP ASE uses a shared or update lock on any base table pages that have been read in a transaction on behalf of the cursor. In addition, the locks are held until the transaction ends, as opposed to being released when the data page is no longer needed. The `holdlock` keyword applies this locking level to the base tables, as specified by the query on the tables or views.

Isolation level 2 is not supported for cursors.

Besides using `holdlock` for isolation level 3, you can use `set transaction isolation level` to specify any of the four isolation levels for your session. When you use `set transaction isolation level,` any cursor you open uses the specified isolation level, unless the transaction isolation level is set at 2. In this case, the cursor uses isolation level 3. You can also use the `select` statement's `at isolation` clause to specify isolation level 0, 1, or 3 for a specific cursor. For example:

```
declare commit_crsr cursor
for select *
from titles
at isolation read committed
```

This statement makes the cursor operate at isolation level 1, regardless of the isolation level of the transaction or session. If you declare a cursor at isolation level 0 (`read uncommitted`), SAP ASE also defines the cursor as read-only. You cannot specify the `for update` clause along with `at isolation read uncommitted` in a `declare cursor` statement.

SAP ASE determines a cursor's isolation level when you open the cursor (not when you declare it), based on the following:

- If the cursor was declared with the `at isolation` clause, that isolation level overrides the transaction isolation level in which it is opened.
- If the cursor was not declared with `at isolation`, the cursor uses the isolation level in which it is opened. If you close the cursor and reopen it later, the cursor acquires the current isolation level of the transaction.

SAP ASE compiles the cursor's query when you declare it. This compilation process is different for isolation level 0 as compared to isolation levels 1 or 3. If you declare a language or client cursor in a transaction with isolation level 1 or 3, opening it in a transaction at isolation level 0 causes an error.

For example:

```
set transaction isolation level 1
declare publishers_crsr cursor
    for select *
    from publishers
open publishers_crsr     /* no error */
```

```
fetch publishers_crsr
close publishers_crsr
set transaction isolation level 0
open publishers_crsr     /* error */
```

23.4.2.8 Stored Procedures and Isolation Levels

System procedures always operate at isolation level 1, regardless of the isolation level of the transaction or session.

User stored procedures operate at the isolation level of the transaction that executes it. If the isolation level changes within a stored procedure, the new isolation level remains in effect only during the execution of the stored procedure.

23.4.2.9 Triggers and Isolation Levels

Since triggers are fired by data modification statements (like `insert`), all triggers execute at either the transaction's isolation level or isolation level 1, whichever is higher.

So, if a trigger fires in a transaction at level 0, SAP ASE sets the trigger's isolation level to 1 before executing its first statement.

23.4.3 Compliance with SQL Standards

To get transactions that comply with SQL standards, you must `set` the `chained` and `transaction isolation level 3` options at the beginning of every application that changes the mode and isolation level for subsequent transactions.

TIf your application uses cursors, you must also set the `close on endtran` option.

Related Information

Use Cursors in Transactions [page 657]

23.4.4 Use the Lock Table Command to Improve Performance

The `lock table` command allows you to explicitly request a table lock for the duration of a transaction. This is useful when an immediate table lock may reduce the overhead of acquiring a large number of row or page locks and save locking time.

Examples of such cases are:

- A table will be scanned more than once in the same transaction, and each scan may need to acquire many page or row locks.
- A scan will exceed a table's lock-promotion threshold and will therefore attempt to escalate to a table lock. If a table lock is not explicitly requested, a scan acquires page or row locks until it reaches the table's lock promotion threshold (see *Reference Manual: Procedures*), at which point it tries to acquire a table lock.

The syntax of `lock table` is:

```
lock table <table_name> in {share | exclusive} mode
          [wait [<no_of_seconds>] | nowait]
```

The `wait/nowait` option allows you to specify how long the command waits to acquire a table lock if it is blocked

These considerations apply to the use of `lock table`:

- You can issue `lock table` only within a transaction.
- You cannot use `lock table` on system tables.
- You can first use `lock table` to lock a table in `share` mode, then use it to upgrade the lock to `exclusive` mode.
- You can use separate `lock table` commands to lock multiple tables within the same transaction.
- Once a table lock is obtained, there is no difference between a table locked with `lock table` and a table locked through lock promotion without the `lock table` command.

Related Information

wait/nowait Option of the Lock Table Command [page 660]

23.5 Transactions in Stored Procedures and Triggers

You can use transactions in stored procedures and triggers just as with statement batches. If a transaction in a batch or stored procedure invokes another stored procedure or trigger containing a transaction, the second transaction is nested into the first one.

The first explicit or implicit (using chained mode) `begin transaction` starts the transaction in the batch, stored procedure, or trigger. Each subsequent `begin transaction` increments the nesting level. Each subsequent `commit transaction` decrements the nesting level until it reaches 0. SAP ASE then commits

the entire transaction. A `rollback transaction` aborts the entire transaction up to the first `begin transaction` regardless of the nesting level or the number of stored procedures and triggers it spans.

In stored procedures and triggers, the number of `begin transaction` statements must match the number of `commit transaction` statements. This also applies to stored procedures that use chained mode. The first statement that implicitly begins a transaction must also have a matching `commit transaction`.

`rollback transaction` statements in stored procedures do not affect subsequent statements in the procedure or batch that originally called the procedure. SAP ASE executes subsequent statements in the stored procedure or batch. However, `rollback transaction` statements in triggers abort the batch so that subsequent statements are not executed.

> **Note**
>
> `rollback` statements in triggers: 1) roll back the transaction, 2) complete subsequent statements in the trigger, and 3) abort the batch so that subsequent statements in the batch are not executed.

For example, the following batch calls the stored procedure `myproc`, which includes a `rollback transaction` statement:

```
begin tran
update titles set ...
insert into titles ...
execute myproc
delete titles where ...
```

The `update` and `insert` statements are rolled back and the transaction is aborted. SAP ASE continues the batch and executes the `delete` statement. However, if there is an `insert` trigger on a table that includes a `rollback transaction`, the entire batch is aborted and the `delete` is not executed. For example:

```
begin tran
update authors set ...
insert into authors ...
delete authors where ...
```

Different transaction modes or isolation levels for stored procedures have certain requirements. Triggers are not affected by the current transaction mode, since they are called as part of a data modification statement.

Related Information

Transaction Modes and Stored Procedures [page 655]

23.5.1 Errors and Transaction Rollbacks

Data integrity errors can affect the state of implicit or explicit transactions.

They affect it in the following ways:

- Errors with severity levels of 19 or greater:
  Since these errors terminate the user connection to the server, any errors of level 19 or greater that occur while a user transaction is in progress abort the transaction and roll back all statements to the outermost `begin transaction`. SAP ASE always rolls back any uncommitted transactions at the end of a session.
- Errors in data modification commands that affect data integrity:
  - Arithmetic overflow and divide-by-zero errors (effects on transactions can be changed with the `set arithabort arith_overflow` command)
  - Permissions violations
  - Rules violations
  - Duplicate key violations

Context	Effects of *rollback*
Transaction only	All data modifications since the start of the transaction are rolled back. If a transaction spans multiple batches, `rollback` affects all of those batches. Any commands issued after the rollback are executed.
Stored procedure only	None.
Stored procedure in a transaction	All data modifications since the start of the transaction are rolled back. If a transaction spans multiple batches, `rollback` affects all those batches. Any commands issued after the rollback are executed. Stored procedure produces error message 266: `Transaction count after EXECUTE indicates that a COMMIT or ROLLBACK TRAN is missing`.
Trigger only	Trigger completes, but trigger effects are rolled back. Any remaining commands in the batch are not executed. Processing resumes at the next batch.
Trigger in a transaction	Trigger completes, but trigger effects are rolled back. All data modifications since the start of the transaction are rolled back. If a transaction spans multiple batches, `rollback` affects all those batches. Any remaining commands in the batch are not executed. Processing resumes at the next batch.
Nested trigger	Inner trigger completes, but all trigger effects are rolled back. Any remaining commands in the batch are not executed. Processing resumes at the next batch.
Nested trigger in a transaction	Inner trigger completes, but all trigger effects are rolled back. All data modifications since the start of the transaction are rolled back. If a transaction spans multiple batches, `rollback` affects all those batches. Any remaining commands in the batch are not executed. Processing resumes at the next batch.

In stored procedures and triggers, the number of `begin transaction` statements must match the number of `commit` statements. A procedure or trigger that contains unpaired `begin/commit` statements produces a warning message when it is executed. This also applies to stored procedures that use chained mode: the first statement that implicitly begins a transaction must have a matching `commit`.

With duplicate key errors and rules violations, the trigger completes (unless there is also a `return` statement), and statements such as `print`, `raiserror`, or remote procedure calls are performed. Then, the trigger and the rest of the transaction are rolled back, and the rest of the batch is aborted. Remote procedure calls executed from inside a normal SQL transaction (not using the DB-Library two-phase commit) are not rolled back by a `rollback` statement.

This table summarizes how a rollback caused by a duplicate key error or a rules violation affects SAP ASE processing in several different contexts.

Context	Effects of Data Modification Errors During Transactions
Transaction only	Current command is aborted. Previous commands are not rolled back, and subsequent commands are executed.
Transaction within a stored procedure	Same as above.
Stored procedure in a transaction	Same as above.
Trigger only	Trigger completes, but trigger effects are rolled back. Any remaining commands in the batch are not executed. Processing resumes at the next batch.
Trigger in a transaction	Trigger completes, but trigger effects are rolled back. All data modifications since the start of the transaction are rolled back. If a transaction spans multiple batches, the rollback affects all of those batches. Any remaining commands in the batch are not executed. Processing resumes at the next batch.
Nested trigger	Inner trigger completes, but all trigger effects are rolled back. Any remaining commands in the batch are not executed. Processing resumes at the next batch.
Nested trigger in a transaction	Inner trigger completes, but all trigger effects are rolled back. All data modifications since the start of the transaction are rolled back. If a transaction spans multiple batches, the rollback affects all of those batches. Any remaining commands in the batch are not executed. Processing resumes at the next batch.
Trigger with rollback followed by an error in the transaction	Trigger effects are rolled back. All data modifications since the start of the transaction are rolled back. If a transaction spans multiple batches, the rollback affects all of those batches. Trigger continues and gets duplicate key or rules error. Normally, the trigger rolls back effects and continues, but in this case, trigger effects are not rolled back. After the trigger completes, any remaining commands in the batch are not executed. Processing resumes at the next batch.

23.5.2 Transaction Modes and Stored Procedures

Stored procedures written to use the unchained transaction mode may be incompatible with other transactions using chained mode, and vice versa.

For example, here is a valid stored procedure using chained transaction mode:

```
create proc myproc
as
insert into publishers
    values ("9996", null, null, null)
commit work
```

A program using unchained transaction mode fails if it calls this procedure because the `commit` does not have a corresponding `begin`. You may encounter other problems:

- Applications that start a transaction using chained mode may create impossibly long transactions or may hold data locks for the entire length of their session, degrading SAP ASE performance.
- Applications may nest transactions at unexpected times. This can produce different results, depending on the transaction mode.

As a rule, applications using one transaction mode should call stored procedures written to use that mode. The exceptions to that rule are SAP system procedures (except for `sp_procxmode`) that can be invoked by sessions using any transaction mode. If no transaction is active when you execute a system procedure, SAP ASE turns off chained mode for the duration of the procedure. Before returning, it resets the mode its original setting.

SAP ASE tags all procedures with the transaction mode (chained or unchained) of the session in which they are created. This helps avoid problems associated with transactions that use one mode to invoke transactions that use the other mode. A stored procedure tagged as chained is not executable in sessions using unchained transaction mode, and vice versa.

Triggers are executable in any transaction mode. Since they are always called as part of a data modification statement, either they are part of a chained transaction (if the session uses chained mode) or they maintain their current transaction mode.

> ⚠ Caution
>
> When using transaction modes, be aware of the effects each setting can have on your applications.

23.5.2.1 Run System Procedures in Chained Mode

SAP ASE allows some system procedures to run in sessions that use chained transaction mode.

- These system procedures can run in sessions using chained transaction mode if there are no open transactions:
  - `sp_configure`
  - `sp_engine`
  - `sp_rename`
- These system procedures can run in sessions using chained transactions after you use `sp_procxmode` to change the transaction mode to `anymode`:

- sp_addengine
- sp_dropengine
- sp_showplan
- sp_sjobcontrol
- sp_sjobcmd
- sp_sjobcreate

  See the *Reference Manual: Procedures*.
- sp_sjobdrop can run in sessions using chained transaction mode, but fails if you execute it during an open transaction.

When you execute these stored procedures, SAP ASE implicitly commits the changes performed by these stored procedures when there are no open transactions, so you need not issue a commit or rollback.

If an open transaction exists when you issue:

- sp_rename, sp_configure, sp_engine, sp_addengine, or sp_dropengine – the procedures fail with error 17260 because they cannot run within a transaction.
- sp_sjobcontrol, sp_sjobcmd, sp_sjobcreate, sp_sjobdrop, or sp_showplan – SAP ASE leaves the transaction open after the procedure executes. You must explicitly issue commit or rollback for the entire transaction.
  If these procedures receive an error when they execute, they roll back only the operations performed inside the procedure. Operations performed before the procedures executed, even if those operations have been performed in the same transaction.

Use set chained {on | off} to set the chained mode for the session. See the *Reference Manual: Commands*.

23.5.2.2 Set Transaction Modes for Stored Procedures

Use sp_procxmode to display or change the transaction mode of stored procedures.

For example, to change the transaction mode for the stored procedure byroyalty to chained, enter:

```
sp_procxmode byroyalty, "chained"
```

sp_procxmode “anymode” lets stored procedures run under either chained or unchained transaction mode. For example:

```
sp_procxmode byroyalty, "anymode"
```

Use sp_procxmode without any parameter values to display the transaction modes for all stored procedures in the current database:

```
sp_procxmode
```

```
procedure name                transaction mode
-------------------------     --------------------
byroyalty                     Any Mode
discount_proc                 Unchained
history_proc                  Unchained
insert_sales_proc             Unchained
```

```
insert_salesdetail_proc      Unchained
storeid_proc                 Unchained
storename_proc               Unchained
title_proc                   Unchained
titleid_proc                  Unchained
```

You can use `sp_procxmode` only in unchained transaction mode.

To change a procedure's transaction mode, you must be a system administrator, the database owner, or the owner of the procedure.

23.6 Use Cursors in Transactions

By default, SAP ASE does not change a cursor's state (open or closed) when a transaction ends through a `commit` or `rollback`. However, SQL standards associate an open cursor with its active transaction; committing or rolling back that transaction automatically closes any open cursors associated with it.

To enforce this SQL-standards-compliant behavior, SAP ASE provides the `close on endtran` option of the `set` command. In addition, if you set chained mode to `on`, SAP ASE starts a transaction when you open a cursor and closes that cursor when the outermost transaction is committed or rolled back.

For example, by default, this sequence of statements produces an error:

```
open test_crsr
commit tran
open test_crsr
```

If you set either the `close on endtran` or `chained` options to `on`, the cursor's state changes from open to closed after the outermost transaction is committed. This allows the cursor to be reopened.

> **Note**
>
> Since client application buffer rows are returned through cursors, and allow users to scroll within those buffers, those client applications should not scroll backward after a transaction aborts. The rows in a client cache may become invalid because of a transaction rollback (unknown to the client) that is enforced by the `close on endtran` option or the chained mode.

Any exclusive locks acquired by a cursor in a transaction are held until the end of that transaction. This also applies to shared locks when you use the `holdlock` keyword, the `at isolation serializable` clause, or the `set isolation level 3` option.

The following rules define the behavior of updates through a cursor with regard to transactions:

- An update occuring within an explicit transaction is considered part of the transaction. If the transaction commits, any updates included with the transaction also commit. If the transaction aborts, any updates included with the transaction are rolled back. Updates through the same cursor that occurred outside the aborted transaction are not affected.
- If updates through a cursor occur within an explicit (and client-specified) transaction, SAP ASE does not commit them when the cursor is closed. It commits or rolls back pending updates only when the transaction associated with that cursor ends.
- A transaction commit or abort has no effect on SQL cursor statements that do not manipulate result rows, such as `declare cursor`, `open cursor`, `close cursor`, `set cursor rows`, and `deallocate`

cursor. For example, if the client opens a cursor within a transaction, and the transaction aborts, the cursor remains open after the abort (unless close on endtran is set or chained mode is used).

However, if you do not set the close on endtran option, the cursor remains open past the end of the transaction, and its current page lock remains in effect. It may also continue to acquire locks as it fetches additional rows.

23.7 Issues to Consider When Using Transactions

There are issues to consider when using transactions in your applications.

- A rollback statement, without a transaction or savepoint name, always rolls back statements to the outermost begin transaction (explicit or implicit) statement and cancels the transaction. If there is no current transaction when you issue rollback, the statement has no effect.
  In triggers or stored procedures, rollback statements, without transaction or savepoint names, roll back all statements to the outermost begin transaction (explicit or implicit).
- rollback does not produce any messages to the user. If warnings are needed, use raiserror or print statements.
- Grouping a large number of Transact-SQL commands into one long-running transaction may affect recovery time. If SAP ASE fails during a long transaction, recovery time increases, since SAP ASE must first undo the entire transaction.
- You can have as many databases in a user transaction as there are in your SAP ASE installation. For example, if your SAP ASE has 25 databases, you can include 25 databases in your user transactions.
- You can independently execute a remote procedure call (RPC) from any transaction in which it is included. In a standard transaction (one that does not use Open Client DB-Library/C two-phase commit or SAP ASE distributed transaction management features), commands executed via an RPC by a remote server are not rolled back with rollback and do not depend on commit to be executed.
- Transactions cannot span more than one connection between a client application and a server. For example, a DB-Library/C application cannot group SQL statements in a transaction across multiple open DBPROCESS connections.
- SAP ASE performs two scans of the log: the first scan looks for data page deallocation and unreserved pages, the second scan looks for log page deallocation. These scans are an internal optimization, transparent to users, and are performed automatically; you cannot switch the scans on or off.
  With post-commit optimization, SAP ASE remembers the “next” log page (in the backward direction) containing these log records. During the post-commit phase, SAP ASE moves to the “next” page requiring post-commit work after processing records from a page. In a concurrent environment, where many users log their transactions to syslogs at the same time, post-commit optimization can improve the performance of post commit operation by avoiding reads or scans of unnecessary log pages.
  The optimization does not show up in any diagnostics.

23.8 Backup and Recovery of Transactions

Every change to a database, whether it is the result of a single `update` statement or a grouped set of SQL statements, is recorded in the system table `syslogs`. This table is called the transaction log and the information it holds is vital for performing recovery procedures.

Some commands that change the database are not logged, such as `truncate table`, bulk-copy into a table that has no indexes, `select into`, `writetext`, and `dump transaction with no_log`.

The transaction log records `update`, `insert`, or `delete` statements on a moment-to-moment basis. When a transaction begins, a `begin transaction` event is recorded in the log. As each data modification statement is received, it is recorded in the log.

The change is always recorded in the log before any change is made in the database itself. This type of log, called a write-ahead log, ensures that the database can be recovered completely in case of a failure.

Failures can be due to hardware or media problems, system software problems, application software problems, program-directed cancellations of transactions, or a user decision to cancel the transaction.

In case of any of these failures, the transaction log can be played back against a copy of the database restored from a backup made with the `dump` commands.

To recover from a failure, transactions that were in progress but not yet committed at the time of the failure must be undone, because a partial transaction is not an accurate change. Completed transactions must be redone if there is no guarantee that they have been written to the database device.

If there are active, long-running transactions that are not committed when SAP ASE fails, undoing the changes may require as much time as the transactions have been running. Such cases include transactions that do not contain a `commit transaction` or `rollback transaction` to match a `begin transaction`. This prevents SAP ASE from writing any changes and increases recovery time.

The SAP ASE dynamic dump allows the database and transaction log to be backed up while use of the database continues. Make frequent backups of your database transaction log. The more often you back up your data, the smaller the amount of work lost if a system failure occurs.

The owner of each database or a user with the `ss_oper` role is responsible for backing up the database and its transaction log with the `dump` commands, though permission to execute them can be transferred to other users. Permission to use the `load` commands, however, defaults to the database owner and cannot be transferred.

Once the appropriate `load` commands are issued, SAP ASE handles all aspects of the recovery process. SAP ASE also controls the checkpoint interval, which is the point at which all data pages that have been changed are guaranteed to have been written to the database device. Users can force a checkpoint, if necessary, with the `checkpoint` command.

For more information about backup and recovery, see the *Reference Manual: Commands* and, *Developing a Backup and Recovery Plan*, in the *System Administration Guide: Volume 2*.

24 Locking Commands and Options

SAP ASE allows you to specify a lock wait period or use readpast locking to silently skip all incompatible locks, without blocking, terminating, or generating a message.

Set a Time Limit on Waiting for Locks

- You can use the `wait` or `nowait` option of the `lock table` command to specify a time limit on waiting to obtain a table lock.
- During a session, you can use the `set lock` command to specify a lock wait period for all subsequent commands issued during the session.
- The `sp_configure` parameter `lock wait period`, used with the session-level setting `set lock wait nnn`, is applicable only to user-defined tables. These settings have no influence on system tables.
- Within a stored procedure, you can use the `set lock` command to specify a lock wait period for all subsequent commands issued within the stored procedure.
- You can use the `lock wait period` option of `sp_configure` to set a server-wide lock wait period.

24.1 wait/nowait Option of the Lock Table Command

Within a transaction, the `lock table` command allows you to request a table lock on a table without waiting for the command to acquire enough row-level or page-level locks to escalate to a table lock.

The `lock table` command contains a `wait/nowait` option that allows you to specify the length of time the command waits until operations in other transactions relinquish any locks they have on the target table.

The syntax for `lock table` is:

```
lock table <table_name> in {share | exclusive} mode
     [wait [<no_of_seconds>] | nowait]
```

The following command, inside a transaction, sets a wait period of 2 seconds for acquiring a table lock on the `titles` table:

```
lock table titles in share mode wait 2
```

If the wait time expires before a table lock is acquired, the transaction proceeds, and row or page locking is used exactly as it would have been without `lock table`, and the following informational message (error number 12207) is generated:

```
Could not acquire a lock within the specified wait period. COMMAND level wait...
```

For a code example of handling this error message during a transaction, see the *Reference Manual: Commands*.

> **Note**
>
> If you use `lock table...wait` without specifying `<no_of_seconds>`, the command waits indefinitely for a lock.

You can set time limits on waiting for a lock at the session level and the system level, as described in the following sections. The wait period set with the `lock table` command overrides both of these

The `nowait` option is equivalent to the `wait` option with a 0-second wait: `lock table` either obtains a table lock immediately or generates the informational message given above. If the lock is not acquired, the transaction proceeds as it would have without the `lock table` command.

You can use the `set lock` command at either the session level or within a stored procedure to control the length of time a task waits to acquire locks.

A system administrator can use the `sp_configure` option, `lock wait period`, to set a server-wide time limit on acquiring locks.

24.2 Session-Level Lock-Wait Limit

You can use `set lock wait` to control the length of time that a command in a session or in a stored procedure waits to acquire locks.

The syntax is:

```
set lock {wait <no_of_seconds >| nowait}
```

`<no_of_seconds>` is an integer. Thus, the following example sets a session-level time limit of 5 seconds on waiting for locks:

```
set lock wait 5
```

With one exception, if the `set lock wait` period expires before a command acquires a lock, the command fails, the transaction containing it is rolled back, and the following error message is generated:

```
Msg 12205, Level 17, State 2:
Server 'sagan', Line 1:
Could not acquire a lock within the specified wait period. SESSION level wait
period=300 seconds, spid=12, lock type=shared page, dbid=9, objid=2080010441,
pageno=92300, rowno=0. Aborting the transaction.
```

The exception to this occurs when `lock table` in a transaction sets a longer wait period than `set lock wait.` In this case, the transaction uses the `lock table` wait period before timing out, as described in the preceding section.

The `set lock nowait` option is equivalent to the `set lock wait` option with a 0-second wait. If a command other than `lock table` cannot obtain a requested lock immediately, the command fails, its transaction is rolled back, and the preceding error message is generated.

If both a server-wide lock-wait limit and a session-level lock-wait limit are set, the session-level limit takes precedence. If no session-level wait period is set, the server-level wait period is used.

24.3 Server-Wide Lock-Wait Limit

A system administrator can configure a server-wide lock-wait limit using the `lock wait period` configuration parameter.

If the lock-wait period expires before a command acquires a lock, unless there is an overriding `set lock wait` or `lock table` wait period, the command fails, the transaction containing it is rolled back, and the following error message is generated:

```
Msg 12205, Level 17, State 2:
Server 'wiz', Line 1:
Could not acquire a lock within the specified wait period. SERVER level wait
period=300 seconds, spid=12, lock type=shared page, dbid=9, objid=2080010441,
pageno=92300, rowno=0. Aborting the transaction.
```

A time limit entered through `set lock wait` or `lock table wait` overrides a server-level lock-wait period. Thus, for example, if the server-level wait period is 5 seconds and the session-level wait period is 10 seconds, an `update` command waits 10 seconds to acquire a lock before failing and aborting its transaction.

The default server-level lock-wait period is effectively "wait forever." To restore the default after setting a time-limited wait, use `sp_configure` to set the value of `lock wait period`:

```
sp_configure "lock wait period", 0, "default"
```

24.4 Information on the Number of Lock-Wait Timeouts

`sp_sysmon` reports on the number of times tasks waiting for locks did not acquire the lock within the specified period.

24.5 Readpast Locking for Queue Processing

Readpast locking instructs a command to silently skip all incompatible locks it encounters, without blocking, terminating, or generating a message. It is primarily used when the rows of a table constitute a queue.

In such a case, a number of tasks may access the table to process the queued rows, which could, for example, represent queued customers or customer orders. A given task is not concerned with processing a specific member of the queue, but with processing any available members of the queue that meet its selection criteria.

Readpast locking is an option that is available for the `select` and `readtext` commands and the data modification commands `update`, `delete`, and `writetext`.

These examples illustrating readpast locking:

To skip all rows that have exclusive locks on them:

```
select * from titles readpast
```

To update only rows that are not locked by another session:

```
update titles
    set price = price * 1.1
    from titles readpast
```

To use readpast locking on the `titles` table, but not on the `authors` or `titleauthor` table:

```
select *
    from titles readpast, authors, titleauthor
    where titles.title_id = titleauthor.title_id
    and authors.au_id = titleauthor.au_id
```

To delete only rows that are not locked in the `stores` table, but to allow the scan to block on the `authors` table:

```
delete stores from stores readpast, authors
where stores.city = authors.city
```

24.5.1 Incompatible Locks During readpast Queries

For `select` and `readtext` commands, incompatible locks are exclusive locks. Therefore, `select` and `readtext` commands can access any rows or pages on which shared or update locks are held.

For `delete`, `update`, and `writetext` commands, any type of page or row lock is incompatible, so that:

- All rows with shared, update, or exclusive row locks are skipped in datarows-locked tables, and
- All pages with shared, update, or exclusive locks are skipped in datapages-locked tables.

All commands specifying `readpast` are blocked if there is an exclusive table lock, except `select` commands executed at transaction isolation level 0.

24.5.2 Allpages-Locked Tables and readpast Queries

The `readpast` option is ignored when it is specified for an allpages-locked table.

The command operates at the isolation level specified for the command or session:

- If the isolation level is 0, dirty reads are performed, the command returns values from locked rows, and does not block.
- If the isolation level is 1 or 3, the command blocks when pages with incompatible locks must be read.

24.5.3 Effects of Isolation Levels Select Queries with readpast

Readpast locking will have different effects in the `select` command based on the transaction isolation level.

Readpast locking is designed to be used at transaction isolation level 1 or 2.

Session-Level Transaction Isolation Levels and readpast

This table that shows the effects of `readpast` on a table in a `select` command on data-only-locked tables:

Session Isolation Level	Effects
`0`, `read uncommitted` (dirty reads)	`readpast` is ignored, and rows containing uncommitted transactions are returned to the user. A warning message prints.
`1`, `read committed`	Rows or pages with incompatible locks are skipped; no locks are held on the rows or pages read.
`2`, `repeatable read`	Rows or pages with incompatible locks skipped; shared locks are held on all rows or pages that are read until the end of the statement or transaction.
`3`, `serializable`	`readpast` is ignored, and the command executes at level 3. The command blocks on any rows or pages with incompatible locks.

Query-Level Isolation Levels and readpast

If `select` commands that specify `readpast` also include any of the following clauses, the commands fail and display error messages.

The commands fail when:

- The `at isolation` clause, specifying `0` or `read uncommitted`
- The `at isolation` clause, specifying `3` or `serializable`
- The `holdlock` keyword on the same table

If a `select` query that specifies `readpast` also specifies `at isolation 2` or `at isolation repeatable read`, shared locks are held on the `readpast` table or tables until the statement or transaction completes.

`readtext` commands that include `readpast` and that specify `at isolation read uncommitted` automatically run at isolation level 0 after issuing a warning message.

24.5.4 Data Modification Commands with readpast and Isolation Levels

If the transaction isolation level for a session is 0, the `delete`, `update`, and `writetext` commands that use `readpast` do not issue warning messages.

- For datapages-locked tables, these commands modify all rows on all pages that are not locked with incompatible locks.
- For datarows-locked tables, the commands affect all rows that are not locked with incompatible locks.

If the transaction isolation level for a session is 3 (serializable reads), the `delete`, `update`, and `writetext` commands that use `readpast` automatically block when they encounter a row or page with an incompatible lock.

At transaction isolation level 2 (serializable reads), the `delete`, `update`, and `writetext` commands:

- Modify all rows on all pages that are not locked with incompatible locks.
- For datarows-locked tables, the commands affect all rows that are not locked with incompatible locks.

24.5.5 text, unitext, and image columns and readpast

If a `select` command with the `readpast` option encounters a text column that has an incompatible lock on it, readpast locking retrieves the row, but returns the text column with a value of null. No distinction is made, in this case, between a text column containing a null value and a null value returned because the column is locked.

If an `update` command with the `readpast` option applies to two or more text columns, and the first text column checked has an incompatible lock on it, readpast locking skips the row. If the column does not have an incompatible lock, the command acquires a lock and modifies the column. If any subsequent text column in the row has an incompatible lock on it, the command blocks until it can obtain a lock and modify the column.

A `delete` command with the `readpast` option skips the row if any of the text columns in the row have an incompatible lock.

25 The pubs2 Database

The sample database pubs2, contains the tables publishers, authors, titles, titleauthor, salesdetail, sales, stores, roysched, discounts, blurbs, and au_pix.

The pubs2 database also lists primary and foreign keys, rules, defaults, views, triggers, and stored procedures used to create these tables.

For information about installing pubs2, see the installation guide for your platform.

To change the sample database using create or data modification statements, you may need to get additional permissions from a system administrator. If you do change the sample database, SAP suggests that you return it to its original state for the sake of future users. Ask a system administrator if you need help restoring the sample databases.

25.1 Tables in the pubs2 Database

In each of the tables in the pubs2 database, the column header specifies the column name, its datatype (including any user-defined datatypes), and its null or not null status. The column header also specifies any defaults, rules, triggers, and indexes that affect the column.

25.1.1 publishers Table

The publishers table contains the publisher name and ID, and the city and state in which each publisher is located.

publishers is defined as:

```
create table publishers
(pub_id char(4) not null,
pub_name varchar(40) not null,
city varchar(20) null,
state char(2) null)
```

Its primary key is pub_id:

```
sp_primarykey publishers, pub_id
```

Its pub_idrule rule is defined as:

```
create rule pub_idrule
as @pub_id in
("1389", "0736", "0877", "1622", "1756")
or @pub_id like "99[0-9][0-9]"
```

25.1.2 authors Table

The `authors` table contains the name, telephone number, author ID, and other information about authors.

`authors` is defined as:

```
create table authors
(au_id id not null,
au_lname varchar(40) not null,
au_fname varchar(20) not null,
phone char(12) not null,
address varchar(40) null,
city varchar(20) null,
state char(2) null,
country varchar(12) null,
postalcode char(10) null)
```

Its primary key is `au_id`:

```
sp_primarykey authors, au_id
```

Its nonclustered index for the `au_lname` and `au_fname` columns is defined as:

```
create nonclustered index aunmind
on authors (au_lname, au_fname)
```

The `phone` column uses this default:

```
create default phonedflt as "UNKNOWN"
sp_bindefault phonedft, "authors.phone"
```

The following view uses `authors`:

```
create view titleview
as
select title, au_ord, au_lname,
price, total_sales, pub_id
from authors, titles, titleauthor
where authors.au_id = titleauthor.au_id
and titles.title_id = titleauthor.title_id
```

25.1.3 titles Table

The `titles` table contains the title id, title, type, publisher ID, price, and other information about titles.

`titles` is defined as:

```
create table titles
(title_id tid not null,
title varchar(80) not null,
type char(12) not null,
pub_id char(4) null,
price money null,
advance money null,
total_sales int null,
notes varchar(200) null,
```

```
pubdate datetime not null,
contract bit not null)
```

Its primary key is `title_id`:

```
sp_primarykey titles, title_id
```

Its `pub_id` column is a foreign key to the `publishers` table:

```
sp_foreignkey titles, publishers, pub_id
```

Its nonclustered index for the `title` column is defined as:

```
create nonclustered index titleind
on titles (title)
```

Its `title_idrule` is defined as:

```
create rule title_idrule
as
@title_id like "BU[0-9][0-9][0-9][0-9]" or
@title_id like "[MT]C[0-9][0-9][0-9][0-9]" or
@title_id like "P[SC][0-9][0-9][0-9][0-9]" or
@title_id like "[A-Z][A-Z]xxxx" or
@title_id like "[A-Z][A-Z]yyyy"
```

The `type` column uses this default:

```
create default typedflt as "UNDECIDED"
sp_bindefault typedflt, "titles.type"
```

The `pubdate` column has this default:

```
create default datedflt as getdate()
sp_bindefault datedflt, "titles.pubdate"
```

`titles` uses this trigger:

```
create trigger deltitle
on titles
for delete
as
if (select count(*) from deleted, salesdetail
where salesdetail.title_id = deleted.title_id) >0
begin
   rollback transaction
   print "You can’t delete a title with sales."
end
```

The following view uses `titles`:

```
create view titleview
as
select title, au_ord, au_lname,
price, total_sales, pub_id
from authors, titles, titleauthor
where authors.au_id = titleauthor.au_id
and titles.title_id = titleauthor.title_id
```

25.1.4 titleauthor Table

the `titleauthor` table shows the author ID, title ID, and royalty of titles by percentage.

`titleauthor` is defined as:

```
create table titleauthor
(au_id id not null,
title_id tid not null,
au_ord tinyint null,
royaltyper int null)
```

Its primary keys are `au_id` and `title_id`:

```
sp_primarykey titleauthor, au_id, title_id
```

Its `title_id` and `au_id` columns are foreign keys to `titles` and `authors`:

```
sp_foreignkey titleauthor, titles, title_id
sp_foreignkey titleauthor, authors, au_id
```

Its nonclustered index for the `au_id` column is defined as:

```
create nonclustered index auidind
on titleauthor(au_id)
```

Its nonclustered index for the `title_id` column is defined as:

```
create nonclustered index titleidind
on titleauthor(title_id)
```

The following view uses `titleauthor`:

```
create view titleview
as
select title, au_ord, au_lname,
price, total_sales, pub_id
from authors, titles, titleauthor
where authors.au_id = titleauthor.au_id
and titles.title_id = titleauthor.title_id
```

The following procedure uses `titleauthor`:

```
create procedure byroyalty @percentage int
as
select au_id from titleauthor
where titleauthor.royaltyper = @percentage
```

25.1.5 salesdetail Table

The `salesdetail` table shows the store ID, order ID, title number, quantity of sales, and discounts of sales.

`salesdetail` is defined as:

```
create table salesdetail
(stor_id char(4) not null,
ord_num numeric(6,0),
title_id tid not null,
qty smallint not null,
discount float not null)
```

Its primary keys are `stor_id` and `ord_num`:

```
sp_primarykey salesdetail, stor_id, ord_num
```

Its `title_id`, `stor_id`, and `ord_num` columns are foreign keys to `titles` and `sales`:

```
sp_foreignkey salesdetail, titles, title_id
sp_foreignkey salesdetail, sales, stor_id, ord_num
```

Its nonclustered index for the `title_id` column is defined as:

```
create nonclustered index titleidind
on salesdetail (title_id)
```

Its nonclustered index for the `stor_id` column is defined as:

```
create nonclustered index salesdetailind
on salesdetail (stor_id)
```

Its `title_idrule` rule is defined as:

```
create rule title_idrule
as
@title_id like "BU[0-9][0-9][0-9][0-9]" or
@title_id like "[MT]C[0-9][0-9][0-9][0-9]" or
@title_id like "P[SC][0-9][0-9][0-9][0-9]" or
@title_id like "[A-Z][A-Z]xxxx" or
@title_id like "[A-Z][A-Z]yyyy"
```

`salesdetail` uses this trigger:

```
create trigger totalsales_trig on salesdetail
    for insert, update, delete
as
/* Save processing:  return if there are no rows affected */
if @@rowcount = 0
    begin
       return
end
/* add all the new values */
/* use isnull:  a null value in the titles table means
**              "no sales yet" not "sales unknown"
*/
update titles
    set total_sales = isnull(total_sales, 0) + (select sum(qty)
    from inserted
```

```
    where titles.title_id = inserted.title_id)
    where title_id in (select title_id from inserted)
/* remove all values being deleted or updated */
update titles
    set total_sales = isnull(total_sales, 0) - (select sum(qty)
    from deleted
    where titles.title_id = deleted.title_id)
    where title_id in (select title_id from deleted)
```

25.1.6 sales Table

The `sales` table contains store IDs, order numbers, and dates of sales.

`sales` is defined as:

```
create table sales
(stor_id char(4) not null,
ord_num varchar(20) not null,
date datetime not null)
```

Its primary keys are `stor_id` and `ord_num`:

```
sp_primarykey sales, stor_id, ord_num
```

Its `stor_id` column is a foreign key to `stores`:

```
sp_foreignkey sales, stores, stor_id
```

25.1.7 stores Table

The `stores` table contains the names, addresses, ID numbers, and payment terms for stores.

`stores` is defined as:

```
create table stores
(stor_id char(4) not null,
stor_name varchar(40) not null,
stor_address varchar(40) null,
city varchar(20) null,
state char(2) null,
country varchar(12) null,
postalcode char(10) null,
payterms varchar(12) null)
```

Its primary key is `stor_id`:

```
sp_primarykey stores, stor_id
```

25.1.8 roysched Table

The `roysched` table contains royalties, defined as a percentage of price.

`roysched` is defined as:

```
create table roysched
title_id tid not null,
lorange int null,
hirange int null,
royalty int null)
```

Its primary key is `title_id`:

```
sp_primarykey roysched, title_id
```

Its `title_id` column is a foreign key to `titles`:

```
sp_foreignkey roysched, titles, title_id
```

Its nonclustered index for the `title_id` column is defined as:

```
create nonclustered index titleidind
on roysched (title_id)
```

25.1.9 discounts Table

The `discounts` table contains discounts in stores.

`discounts` is defined as:

```
create table discounts
(discounttype varchar(40) not null,
stor_id char(4) null,
lowqty smallint null,
highqty smallint null,
discount float not null)
```

Its primary keys are `discounttype` and `stor_id`:

```
sp_primarykey discounts, discounttype, stor_id
```

Its `stor_id` is a foreign key to `stores`:

```
sp_foreignkey discounts, stores, stor_id
```

25.1.10 blurbs Table

The `blurbs` table contains sample blurbs for books.

`blurbs` is defined as:

```
create table blurbs
(au_id  id not null,
copy text null)
```

Its primary key is `au_id`:

```
sp_primarykey blurbs, au_id
```

Its `au_id` column is a foreign key to `authors`:

```
sp_foreignkey blurbs, authors, au_id
```

25.1.11 au_pix Table

The `author_pix` table contains photographs of authors in the `pubs2` database.

`au_pix` is defined as:

```
create table au_pix
(au_id char(11) not null,
pic image null,
format_type char(11) null,
bytesize int null,
pixwidth_hor char(14) null,
pixwidth_vert char(14) null)
```

Its primary key is `au_id`:

```
sp_primarykey au_pix, au_id
```

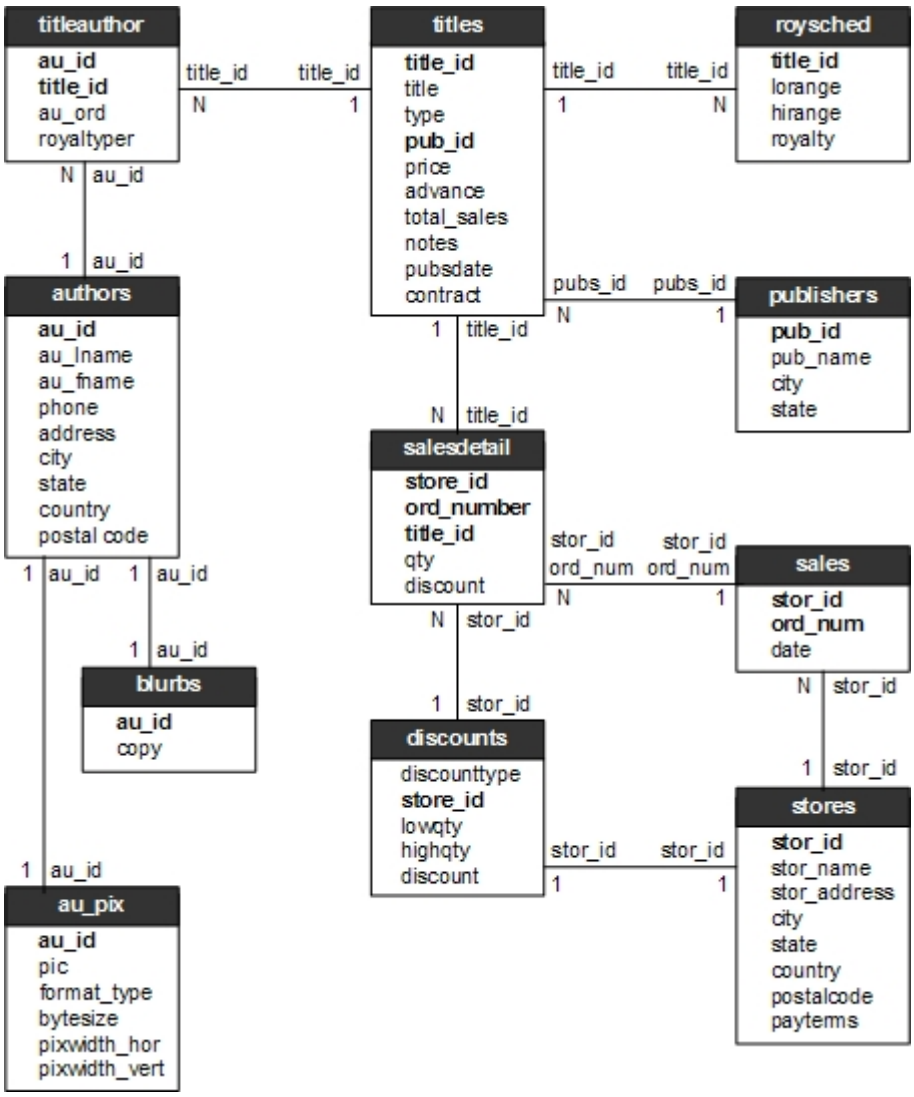
Its `au_id` column is a foreign key to authors:

```
sp_foreignkey au_pix, authors, au_id
```

The `pic` column contains binary data. Since the `image` data (six pictures, two each in PICT, TIF, and Sunraster file formats) is quite large, run the `installpix2` script only to use or test the `image` datatype. The `image` data is supplied to show how SAP stores `image` data. SAP does not supply any tools for displaying `image` data: you must use the appropriate screen graphics tools to display the images once you have extracted them from the database.

25.2 Diagram of the pubs2 Database

The diagram of the pubs2 database shows the database relationships among the tables.



26 The pubs3 Database

The sample database pubs3 contains the tables publishers, authors, titles, titleauthor, salesdetail, sales, stores, store_employees, roysched, discounts, and blurbs.

It lists the primary primary and foreign keys, rules, defaults, views, triggers, and stored procedures used to create each table.

For information about installing pubs3, see the installation guide for your platform.

To change the sample database using create or data modification statements, you may need to get additional permissions from a system administrator. If you do change the sample database, SAP suggests that you return it to its original state for the sake of future users. Ask a system administrator if you need help restoring the sample databases.

26.1 Tables in the pubs3 Database

In each of the tables in the pubs3 database, the column header specifies the column name, its datatype (including any user-defined datatypes), its null or not null status, and how it uses referential integrity.

The column header also specifies any defaults, rules, triggers, and indexes that affect the column.

26.1.1 publishers Table

The publishers table contains the publisher ID, and name, city, and state.

publishers is defined as:

```
create table publishers
(pub_id char(4) not null,
pub_name varchar(40) not null,
city varchar(20) null,
state char(2) null,
unique nonclustered (pub_id))
```

Its pub_idrule rule is defined as:

```
create rule pub_idrule
as @pub_id in
("1389", "0736", "0877", "1622", "1756")
or @pub_id like "99[0-9][0-9]"
```

26.1.2 authors Table

The `authors` table contains the names, phone numbers, and other information about authors.

`authors` is defined as:

```
create table authors
(au_id id not null,
au_lname varchar(40) not null,
au_fname varchar(20) not null,
phone char(12) not null,
address varchar(40) null,
city varchar(20) null,
state char(2) null,
country varchar(12) null,
postalcode char(10) null,
unique nonclustered (au_id))
```

Its nonclustered index for the `au_lname` and `au_fname` columns is defined as:

```
create nonclustered index aunmind
on authors (au_lname, au_fname)
```

The `phone` column uses this default:

```
create default phonedflt as "UNKNOWN"
sp_bindefault phonedft, "authors.phone"
```

The following view uses `authors`:

```
create view titleview
as
select title, au_ord, au_lname,
price, num_sold, pub_id
from authors, titles, titleauthor
where authors.au_id = titleauthor.au_id
and titles.title_id = titleauthor.title_id
```

26.1.3 titles Table

The `titles` table contains the name, title ID, type, and other information about titles.

`titles` is defined as:

```
create table titles
(title_id tid not null,
title varchar(80) not null,
type char(12) not null,
pub_id char(4) null
    references publishers(pub_id),
price money null,
advance numeric(12,2) null,
num_sold int null,
notes varchar(200) null,
pubdate datetime not null,
contract bit not null,
```

```
unique nonclustered (title_id))
```

Its nonclustered index for the `title` column is defined as:

```
create nonclustered index titleind
on titles (title)
```

Its `title_idrule` is defined as:

```
create rule title_idrule
as
@title_id like "BU[0-9][0-9][0-9][0-9]" or
@title_id like "[MT]C[0-9][0-9][0-9][0-9]" or
@title_id like "P[SC][0-9][0-9][0-9][0-9]" or
@title_id like "[A-Z][A-Z]xxxx" or
@title_id like "[A-Z][A-Z]yyyy"
```

The `type` column uses this default:

```
create default typedflt as "UNDECIDED"
sp_bindefault typedflt, "titles.type"
```

The `pubdate` column uses this default:

```
create default datedflt as getdate()
sp_bindefault datedflt, "titles.pubdate"
```

`titles` uses this trigger:

```
create trigger deltitle
on titles
for delete
as
if (select count(*) from deleted, salesdetail
where salesdetail.title_id = deleted.title_id) >0
begin
   rollback transaction
   print "You can’t delete a title with sales."
end
```

This view uses `titles`:

```
create view titleview
as
select title, au_ord, au_lname,
price, num_sold, pub_id
from authors, titles, titleauthor
where authors.au_id = titleauthor.au_id
and titles.title_id = titleauthor.title_id
```

26.1.4 titleauthor Table

The `titleauthor` table contains the title and author IDs, royalty percentages, and other information about titles and authors.

`titleauthor` is defined as:

```
create table titleauthor
(au_id id not null
    references authors(au_id),
title_id tid not null
    references titles(title_id),
au_ord tinyint null,
royaltyper int null)
```

Its nonclustered index for the `au_id` column is defined as:

```
create nonclustered index auidind
on titleauthor(au_id)
```

Its nonclustered index for the `title_id` is:

```
create nonclustered index titleidind
on titleauthor(title_id)
```

This view uses `titleauthor`:

```
create view titleview
as
select title, au_ord, au_lname,
price, num_sold, pub_id
from authors, titles, titleauthor
where authors.au_id = titleauthor.au_id
and titles.title_id = titleauthor.title_id
```

This procedure uses `titleauthor`:

```
create procedure byroyalty @percentage int
as
select au_id from titleauthor
where titleauthor.royaltyper = @percentage
```

26.1.5 salesdetail Table

The `salesdetail` table contains the store ID, order number, and other details of sales.

`salesdetail` is defined as:

```
create table salesdetail
(stor_id char(4) not null
    references sales(stor_id),
ord_num numeric(6,0)
    references sales(ord_num),
title_id tid not null
    references titles(title_id),
```

```
qty smallint not null,
discount float not null)
```

Its nonclustered index for the `title_id` column is defined as:

```
create nonclustered index titleidind
on salesdetail (title_id)
```

Its nonclustered index for the `stor_id` column is defined as:

```
create nonclustered index salesdetailind
on salesdetail (stor_id)
```

Its `title_idrule` rule is defined as:

```
create rule title_idrule
as
@title_id like "BU[0-9][0-9][0-9][0-9]" or
@title_id like "[MT]C[0-9][0-9][0-9][0-9]" or
@title_id like "P[SC][0-9][0-9][0-9][0-9]" or
@title_id like "[A-Z][A-Z]xxxx" or
@title_id like "[A-Z][A-Z]yyyy"
```

`salesdetail` uses this trigger:

```
create trigger totalsales_trig on salesdetail
    for insert, update, delete
as
/* Save processing:  return if there are no rows affected */
if @@rowcount = 0
    begin
       return
end
/* add all the new values */
/* use isnull:  a null value in the titles table means
**              "no sales yet" not "sales unknown"
*/
update titles
    set num_sold = isnull(num_sold, 0) + (select sum(qty)
    from inserted
    where titles.title_id = inserted.title_id)
    where title_id in (select title_id from inserted)
/* remove all values being deleted or updated */
update titles
    set num_sold = isnull(num_sold, 0) - (select sum(qty)
    from deleted
    where titles.title_id = deleted.title_id)
    where title_id in (select title_id from deleted)
```

26.1.6 sales Table

The `sales` table contains the store ID, order numbers, and dates of sales.

`sales` is defined as:

```
create table sales
(stor_id char(4) not null
    references stores(stor_id),
```

```
ord_num numeric(6,0) identity,
date datetime not null,
unique nonclustered (ord_num))
```

26.1.7 stores Table

The `stores` table contains the store ID, store name, and other information about stores.

`stores` is defined as:

```
create table stores
(stor_id char(4) not null,
stor_name varchar(40) not null,
stor_address varchar(40) null,
city varchar(20) null,
state char(2) null,
country varchar(12) null,
postalcode char(10) null,
payterms varchar(12) null,
unique nonclustered (stor_id))
```

26.1.8 store_employees Table

The `store_employees` table contains the store, employer, and manager IDs, and other information about store employees.

`store_employees` is defined as:

```
create table store_employees
(stor_id char(4) null
    references stores(stor_id),
emp_id id not null,
mgr_id id null
    references store_employees(emp_id),
emp_lname varchar(40) not null,
emp_fname varchar(20) not null,
phone char(12) null,
address varchar(40) null,
city varchar(20) null,
state char(2) null,
country varchar(12) null,
postalcode varchar(10) null,
unique nonclustered (emp_id))
```

26.1.9 roysched Table

The `roysched` table contains title ID, royalty percentage, and other information about title royalties.

`roysched` is defined as:

```
create table roysched
```

```
title_id tid not null
    references titles(title_id),
lorange int null,
hirange int null,
royalty int null)
```

Its nonclustered index for the `title_id` column is defined as:

```
create nonclustered index titleidind
on roysched (title_id)
```

26.1.10 discounts Table

The `discount` table contains the discount type, store ID, quantity, and percentage discount.

`discounts` is defined as:

```
create table discounts
(discounttype varchar(40) not null,
stor_id char(4) null
    references stores(stor_id),
lowqty smallint null,
highqty smallint null,
discount float not null)
```

26.1.11 blurbs Table

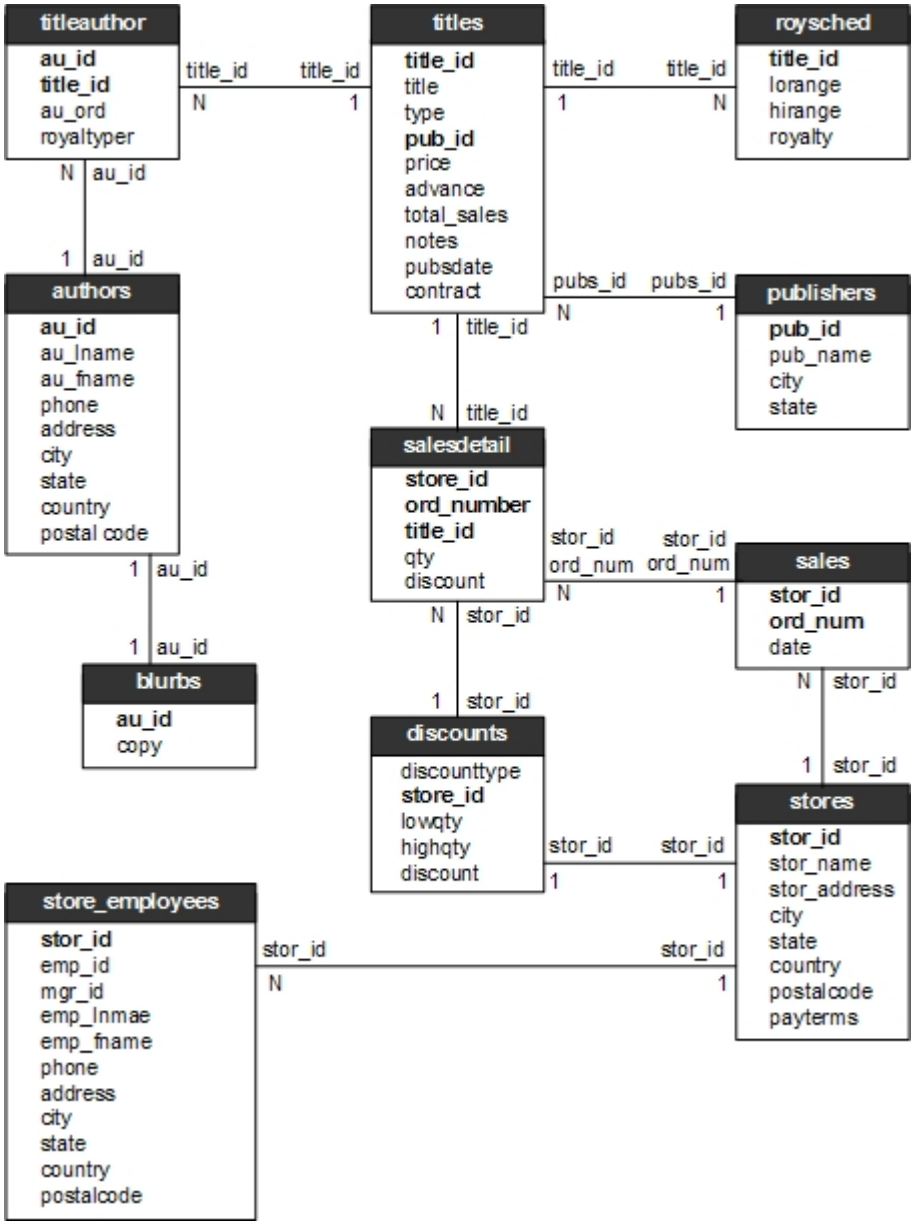
The `blurbs` table contains the author ID and blurb for books in the `pubs3` database.

`blurbs` is defined as:

```
create table blurbs
(au_id  id not null
    references authors(au_id),
copy text null)
```

26.2 Diagram of the pubs3 Database

The diagram of the pubs3 database shows the database relationships among the tables.


titleauthor
au_id
title_id
au_ord
royaltyper
title_id
title_id
N
1
titles
title_id
title
type
pub_id
price
advance
total_sales
notes
pubsdate
contract
title_id
title_id
1
N
roysched
title_id
lorange
hirange
royalty
N
au_id
1
au_id
authors
au_id
au_lname
au_fname
phone
address
city
state
country
postal code
pubs_id
pubs_id
N
1
publishers
pub_id
pub_name
city
state
1
title_id
N
title_id
salesdetail
store_id
ord_number
title_id
qty
discount
stor_id
stor_id
ord_num
ord_num
N
1
sales
stor_id
ord_num
date
1
au_id
1
au_id
blurbs
au_id
copy
N
stor_id
1
stor_id
discounts
discounttype
store_id
lowqty
highqty
discount
N
stor_id
1
stor_id
stores
stor_id
stor_name
stor_address
city
state
country
postalcode
payterms
stor_id
stor_id
1
1
store_employees
stor_id
emp_id
mgr_id
emp_lnmae
emp_fname
phone
address
city
state
country
postalcode
stor_id
stor_id
N
1

Important Disclaimers and Legal Information

Coding Samples

Any software coding and/or code lines / strings ("Code") included in this documentation are only examples and are not intended to be used in a productive system environment. The Code is only intended to better explain and visualize the syntax and phrasing rules of certain coding. SAP does not warrant the correctness and completeness of the Code given herein, and SAP shall not be liable for errors or damages caused by the usage of the Code, unless damages were caused by SAP intentionally or by SAP's gross negligence.

Accessibility

The information contained in the SAP documentation represents SAP's current view of accessibility criteria as of the date of publication; it is in no way intended to be a binding guideline on how to ensure accessibility of software products. SAP in particular disclaims any liability in relation to this document. This disclaimer, however, does not apply in cases of willful misconduct or gross negligence of SAP. Furthermore, this document does not result in any direct or indirect contractual obligations of SAP.

Gender-Neutral Language

As far as possible, SAP documentation is gender neutral. Depending on the context, the reader is addressed directly with "you", or a gender-neutral noun (such as "sales person" or "working days") is used. If when referring to members of both sexes, however, the third-person singular cannot be avoided or a gender-neutral noun does not exist, SAP reserves the right to use the masculine form of the noun and pronoun. This is to ensure that the documentation remains comprehensible.

Internet Hyperlinks

The SAP documentation may contain hyperlinks to the Internet. These hyperlinks are intended to serve as a hint about where to find related information. SAP does not warrant the availability and correctness of this related information or the ability of this information to serve a particular purpose. SAP shall not be liable for any damages caused by the use of related information unless damages have been caused by SAP's gross negligence or willful misconduct. All links are categorized for transparency (see: http://help.sap.com/disclaimer).

go.sap.com/registration/contact.html

© 2016 SAP SE or an SAP affiliate company. All rights reserved.
No part of this publication may be reproduced or transmitted in any form or for any purpose without the express permission of SAP SE or an SAP affiliate company. The information contained herein may be changed without prior notice.
Some software products marketed by SAP SE and its distributors contain proprietary software components of other software vendors. National product specifications may vary.
These materials are provided by SAP SE or an SAP affiliate company for informational purposes only, without representation or warranty of any kind, and SAP or its affiliated companies shall not be liable for errors or omissions with respect to the materials. The only warranties for SAP or SAP affiliate company products and services are those that are set forth in the express warranty statements accompanying such products and services, if any. Nothing herein should be construed as constituting an additional warranty.
SAP and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP SE (or an SAP affiliate company) in Germany and other countries. All other product and service names mentioned are the trademarks of their respective companies.
Please see http://www.sap.com/corporate-en/legal/copyright/index.epx for additional trademark information and notices.