# Contents

## CHAPTER 1  SQL Building Blocks

<table>
<thead>
<tr>
<th>Topics</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL in Adaptive Server</td>
<td>1</td>
</tr>
<tr>
<td>Queries, data modification, and commands</td>
<td>2</td>
</tr>
<tr>
<td>Tables, columns, and rows</td>
<td>3</td>
</tr>
<tr>
<td>Relational operations</td>
<td>3</td>
</tr>
<tr>
<td>Compiled objects</td>
<td>4</td>
</tr>
<tr>
<td>Compliance to ANSI standards</td>
<td>5</td>
</tr>
<tr>
<td>Federal Information Processing Standards (FIPS) flagger</td>
<td>6</td>
</tr>
<tr>
<td>Chained transactions and isolation levels</td>
<td>7</td>
</tr>
<tr>
<td>Identifiers</td>
<td>7</td>
</tr>
<tr>
<td>SQL standard-style comments</td>
<td>7</td>
</tr>
<tr>
<td>Right truncation of character strings</td>
<td>8</td>
</tr>
<tr>
<td>Permissions required for update and delete statements</td>
<td>8</td>
</tr>
<tr>
<td>Arithmetic errors</td>
<td>8</td>
</tr>
<tr>
<td>Synonymous keywords</td>
<td>9</td>
</tr>
<tr>
<td>Treatment of nulls</td>
<td>9</td>
</tr>
<tr>
<td>Naming conventions</td>
<td>10</td>
</tr>
<tr>
<td>SQL data characters</td>
<td>10</td>
</tr>
<tr>
<td>SQL language characters</td>
<td>10</td>
</tr>
<tr>
<td>Identifiers</td>
<td>11</td>
</tr>
<tr>
<td>Expressions in Adaptive Server</td>
<td>18</td>
</tr>
<tr>
<td>Arithmetic and character expressions</td>
<td>18</td>
</tr>
<tr>
<td>Relational and logical expressions</td>
<td>24</td>
</tr>
<tr>
<td>Transact-SQL extensions</td>
<td>25</td>
</tr>
<tr>
<td>compute clause</td>
<td>26</td>
</tr>
<tr>
<td>Control-of-flow language</td>
<td>26</td>
</tr>
<tr>
<td>Stored procedures</td>
<td>26</td>
</tr>
<tr>
<td>Extended stored procedures</td>
<td>27</td>
</tr>
<tr>
<td>Triggers</td>
<td>27</td>
</tr>
<tr>
<td>Defaults and rules</td>
<td>28</td>
</tr>
<tr>
<td>Error handling and set options</td>
<td>28</td>
</tr>
<tr>
<td>Additional Adaptive Server extensions to SQL</td>
<td>29</td>
</tr>
<tr>
<td>Adaptive Server login accounts</td>
<td>30</td>
</tr>
<tr>
<td>isql utility</td>
<td>31</td>
</tr>
<tr>
<td>Default databases</td>
<td>32</td>
</tr>
</tbody>
</table>
Contents

Null values and group by .......................................................... 89
where clause and group by ...................................................... 91
group by and all ........................................................................ 92
Aggregates without group by ................................................... 93
Selecting groups of data: the having clause .......................... 94
How the having, group by, and where clauses interact ........ 96
Using having without group by ............................................... 99
Sorting query results: the order by clause .............................. 100
order by and group by ............................................................ 102
order by and group by used with select distinct .................. 103
Summarizing groups of data: the compute clause ............... 104
Row aggregates and compute ................................................. 107
Specifying more than one column after compute ............... 108
Using more than one compute clause .................................. 109
Applying an aggregate to more than one column ............... 110
Using different aggregates in the same compute clause ...... 110
Generating totals: compute without by ................................. 111
Combining queries: the union operator ................................. 112
Guidelines for union queries .................................................. 114
Using union with other Transact-SQL commands .............. 116

CHAPTER 4 Joins: Retrieving Data from Several Tables ................. 119
How joins work .................................................................. 120
Join syntax .......................................................................... 120
Joins and the relational model ............................................. 121
How joins are structured ...................................................... 121
The from clause ................................................................. 123
The where clause ............................................................... 124
How joins are processed ...................................................... 126
Equijoins and natural joins .................................................. 127
Joins with additional conditions ......................................... 128
Joins not based on equality ................................................ 129
Self-joins and correlation names .......................................... 130
The not-equal join ............................................................... 131
Not-equal joins and subqueries .......................................... 133
Joining more than two tables ............................................. 134
Outer joins .......................................................................... 135
Inner and outer tables ......................................................... 136
Outer join restrictions ........................................................ 136
Views used with outer joins ................................................ 137
ANSI inner and outer joins ................................................ 137
ANSI outer joins ............................................................... 138
Transact-SQL outer joins .................................................... 154
Relocated joins ................................................................. 159
# Contents

Using relocated joins ............................................................. 159  
Configuring relocated joins .................................................... 160  
How null values affect joins .......................................................... 160  
Determining which table columns to join ...................................... 161  

## CHAPTER 5  Subqueries: Using Queries Within Other Queries ................. 163  
How subqueries work ................................................................... 163  
Subquery restrictions ............................................................. 164  
Example of using a subquery ................................................ 165  
Qualifying column names ...................................................... 166  
Subqueries with correlation names ............................................. 167  
Multiple levels of nesting ....................................................... 168  
Using an asterisk in nested select statements ..................................... 168  
Subqueries in update, delete, and insert statements .................... 173  
Subqueries in conditional statements ....................................... 174  
Subqueries instead of expressions .............................................. 174  
Types of subqueries..................................................................... 175  
Expression subqueries .......................................................... 176  
Quantified predicate subqueries ............................................ 179  
Subqueries used with in ......................................................... 185  
Subqueries used with not in ..................................................... 187  
Subqueries using not in with NULL ........................................ 187  
Subqueries used with exists....................................................... 188  
Subqueries used with not exists................................................... 190  
Finding intersection and difference with exists.......................... 191  
Subqueries using SQL derived tables ......................................... 192  
Using correlated subqueries ..................................................... 193  
Correlated subqueries with correlation names ......................... 194  
Correlated subqueries with comparison operators .................... 195  
Correlated subqueries in a having clause .................................. 196  

## CHAPTER 6  Using and Creating Datatypes .................................................... 197  
How Transact-SQL datatypes work ............................................. 197  
Using system-supplied datatypes ............................................. 198  
Exact numeric types: integers ................................................. 200  
Exact numeric types: decimal numbers ...................................... 201  
Approximate numeric datatypes ............................................ 202  
Money datatypes .......................................................... 202  
Date and time datatypes ......................................................... 203  
Character datatypes .............................................................. 204  
Binary datatypes .......................................................... 209  
The bit datatype .......................................................... 210  
The timestamp datatype .......................................................... 211  

Adaptive Server Enterprise
Contents

The sysname and longsysname datatype............................. 211
Using LOB Locators in Transact-SQL Statements.................. 211
Creating a LOB locator...................................................... 212
Converting the locator value to the LOB value ................... 214
Locator scope..................................................................... 215
Converting between datatypes.......................................... 215
Mixed-mode arithmetic and datatype hierarchy .................... 216
Working with money datatypes ......................................... 218
Determining precision and scale ........................................ 219
Creating user-defined datatypes ........................................ 219
Specifying length, precision, and scale .............................. 220
Specifying null type .......................................................... 220
Associating rules and defaults with user-defined datatypes.. 221
Creating user-defined datatype with IDENTITY property ...... 221
Creating IDENTITY columns from user-defined datatypes ... 222
Dropping a user-defined datatype ....................................... 222
Getting information about datatypes ................................ 222

CHAPTER 7 Adding, Changing, Transferring, and Deleting Data.......... 225
Referential integrity ......................................................... 226
Transactions.................................................................... 227
Using the sample databases............................................. 227
Datatype entry rules ........................................................ 228
c, nchar, unichar, univarchar, varchar, nvarchar, unitext, and
text............................................................................. 228
date and time.................................................................... 229
binary, varbinary, and image .......................................... 234
money and smallmoney ................................................... 234
float, real, and double precision ...................................... 235
decimal and numeric ....................................................... 235
Integer types and their unsigned counterparts .................... 236
timestamp................................................................. 237
Adding new data .............................................................. 237
Adding new rows with values ........................................... 238
Inserting data into specific columns ................................ 238
Adding new rows with select ............................................ 247
Creating nonmaterialized, non-null columns ................. 250
Adding nonmaterialized columns .................................... 250
Tables that already have nonmaterialized columns .......... 251
Nonmaterialized column storage ...................................... 251
Altering nonmaterialized columns ................................... 252
Limitations...................................................................... 252
Changing existing data..................................................... 253
Using the set clause with update................................. 254
Using the where clause with update ......................................... 255
Using the from clause with update ........................................... 255
Performing updates with joins ............................................... 256
Updating IDENTITY columns ................................................. 256
Changing text, unitext, and image data .................................... 257
Truncating trailing zeros .......................................................... 258
Transferring data incrementally .............................................. 262
Marking tables for incremental transfer .................................. 263
Transferring tables from a destination file ................................ 263
Converting Adaptive Server datatypes to IQ ......................... 264
Storing transfer information ................................................... 266
Exceptions and errors ............................................................ 269
Sample session for incremental data transfer ....................... 270
Deleting data ............................................................................ 276
Using the from clause with delete .......................................... 276
Deleting from IDENTITY columns ......................................... 277
Deleting all rows from a table ................................................. 277
truncate table syntax ............................................................. 278

CHAPTER 8

Creating Databases and Tables ................................................. 279
Databases and tables .............................................................. 279
Enforcing data integrity in databases ..................................... 280
Permissions within databases .................................................. 281
Using and creating databases ............................................... 282
Choosing a database: use ......................................................... 283
Creating a user database with create database ....................... 283
quiesce database command .................................................... 286
Altering the sizes of databases ................................................. 287
Dropping databases .............................................................. 288
Creating tables ........................................................................ 288
Maximum number of columns per table ......................... 288
Example ................................................................................. 289
Choosing table names ............................................................ 290
Creating tables in different databases .................................... 290
create table syntax ................................................................. 290
Using IDENTITY columns ....................................................... 291
Allowing null values in a column .......................................... 295
Using temporary tables .......................................................... 297
Managing identity gaps in tables ............................................ 301
Parameters for controlling identity gaps .................................. 302
Comparison of identity burning set factor and identity_gap .. 303
Setting the table-specific identity gap .................................... 304
Changing the table-specific identity gap ................................ 304
Displaying table-specific identity gap information ............. 305
## Content

<table>
<thead>
<tr>
<th>SQL-derived table syntax</th>
<th>367</th>
</tr>
</thead>
<tbody>
<tr>
<td>Derived column lists</td>
<td>368</td>
</tr>
<tr>
<td>Correlated SQL-derived tables</td>
<td>369</td>
</tr>
<tr>
<td>Using SQL-derived tables</td>
<td>369</td>
</tr>
<tr>
<td>Nesting</td>
<td>370</td>
</tr>
<tr>
<td>Subqueries using SQL-derived tables</td>
<td>370</td>
</tr>
<tr>
<td>Unions</td>
<td>370</td>
</tr>
<tr>
<td>Unions in subqueries</td>
<td>371</td>
</tr>
<tr>
<td>Renaming columns with SQL-derived tables</td>
<td>371</td>
</tr>
<tr>
<td>Constant expressions</td>
<td>371</td>
</tr>
<tr>
<td>Aggregate functions</td>
<td>373</td>
</tr>
<tr>
<td>Joins with SQL-derived tables</td>
<td>373</td>
</tr>
<tr>
<td>Unions in subqueries</td>
<td>374</td>
</tr>
<tr>
<td>Renaming columns with SQL-derived tables</td>
<td>374</td>
</tr>
<tr>
<td>Creating a table from a SQL-derived table</td>
<td>375</td>
</tr>
<tr>
<td>Correlated attributes</td>
<td>375</td>
</tr>
</tbody>
</table>

### Chapter 10: Partitioning Tables and Indexes

<table>
<thead>
<tr>
<th>Upgrading from Adaptive Server 12.5.x and earlier</th>
<th>377</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data partitions</td>
<td>378</td>
</tr>
<tr>
<td>Index partitions</td>
<td>379</td>
</tr>
<tr>
<td>Partition IDs</td>
<td>379</td>
</tr>
<tr>
<td>Locks and partitions</td>
<td>380</td>
</tr>
<tr>
<td>Partitioning types</td>
<td>380</td>
</tr>
<tr>
<td>Range partitioning</td>
<td>381</td>
</tr>
<tr>
<td>Hash partitioning</td>
<td>381</td>
</tr>
<tr>
<td>List partitioning</td>
<td>382</td>
</tr>
<tr>
<td>Round-robin partitioning</td>
<td>382</td>
</tr>
<tr>
<td>Composite partitioning keys</td>
<td>383</td>
</tr>
<tr>
<td>Partition pruning</td>
<td>384</td>
</tr>
<tr>
<td>Indexes and partitions</td>
<td>385</td>
</tr>
<tr>
<td>Global indexes</td>
<td>385</td>
</tr>
<tr>
<td>Local indexes</td>
<td>390</td>
</tr>
<tr>
<td>Guaranteeing a unique index</td>
<td>392</td>
</tr>
<tr>
<td>Creating and managing partitions</td>
<td>393</td>
</tr>
<tr>
<td>Partitioning tasks</td>
<td>393</td>
</tr>
<tr>
<td>Creating data partitions</td>
<td>395</td>
</tr>
<tr>
<td>Creating partitioned indexes</td>
<td>398</td>
</tr>
<tr>
<td>Creating a partitioned table from an existing table</td>
<td>400</td>
</tr>
<tr>
<td>Altering data partitions</td>
<td>401</td>
</tr>
<tr>
<td>Changing an unpartitioned table to a partitioned table</td>
<td>401</td>
</tr>
<tr>
<td>Adding partitions to a partitioned table</td>
<td>402</td>
</tr>
<tr>
<td>Changing the partitioning type</td>
<td>402</td>
</tr>
<tr>
<td>Changing the partitioning key</td>
<td>402</td>
</tr>
<tr>
<td>Unpartitioning round-robin–partitioned tables</td>
<td>403</td>
</tr>
<tr>
<td>Using the partition parameter</td>
<td>403</td>
</tr>
</tbody>
</table>
Contents

Altering partition-key columns ................................................. 404
Configuring partitions ............................................................... 405
Updating, deleting, and inserting in partitioned tables .................. 405
Updating values in partition-key columns ..................................... 406
Displaying information about partitions ....................................... 407
Using functions ....................................................................... 407
Truncating a partition ............................................................... 408
Using partitions to load table data ............................................... 408
Updating partition statistics ....................................................... 409

CHAPTER 11  
Virtually hashed tables ............................................................... 411
Structure of a virtually hashed table ............................................ 412
Creating a virtually hashed table ................................................ 413
Limitations for virtually hashed tables ....................................... 416
Commands that support virtually hashed tables .......................... 417
Query processor support ........................................................... 417
Monitor counter support ............................................................ 418
System procedure support .......................................................... 418

CHAPTER 12  
Views: Limiting Access to Data .................................................. 419
How views work ....................................................................... 419
Advantages of views ............................................................... 420
Examples ................................................................................ 422
Creating views .......................................................................... 424
create view syntax ................................................................. 424
Using the select statement with create view .............................. 425
Validating a view’s selection criteria ............................................ 430
Retrieving data through views .................................................... 431
View resolution ........................................................................ 432
Redefining views ..................................................................... 433
Renaming views ....................................................................... 434
Modifying data through views .................................................... 435
Restrictions on updating views .................................................... 436
Dropping views ....................................................................... 440
Using views as security mechanisms .......................................... 440
Getting information about views ............................................... 441
Using sp_help and sp_helptext to display view information ....... 441
Using sp_depends to list dependent objects ................................ 442
Listing all views in a database .................................................... 442
Finding an object name and ID .................................................... 442
Contents

CHAPTER 13 Creating Indexes on Tables ........................................................ 443
  How indexes work ........................................................................ 443
  Comparing the two ways to create indexes........................... 445
  Guidelines for using indexes ................................................. 445
  Creating indexes .......................................................................... 446
    create index syntax ............................................................... 447
  Indexing more than one column: composite indexes .......... 447
  Indexing with function-based indexes ................................... 448
  Using the unique option......................................................... 449
  Including IDENTITY columns in nonunique indexes .......... 449
  Ascending and descending index-column values .......... 450
    Using fillfactor, max_rows_per_page, and reservepagegap . 450
  Indexes on computed columns ........................................ 452
    Function-based indexes ............................................................... 452
  Using clustered or nonclustered indexes .................. 453
    Creating clustered indexes on segments ...................... 454
  Specifying index options .............................................................. 455
    Using the ignore_dup_key option.................................... 455
    Using the ignore_dup_row and allow_dup_row options . 456
    Using the sorted_data option ........................................ 457
    Using the on segment_name option .................................... 457
  Dropping indexes ......................................................................... 457
  Determining what indexes exist on a table........................... 458
  Updating statistics about indexes......................................... 460

CHAPTER 14 Defining Defaults and Rules for Data ........................................ 463
  How defaults and rules work .................................................. 463
  Creating defaults .......................................................................... 464
    create default syntax ............................................................. 465
    Binding defaults.................................................................... 466
    Unbinding defaults.............................................................. 468
  How defaults affect NULL values ........................................ 469
  Dropping defaults ......................................................................... 469
  Creating rules............................................................................... 470
    create rule syntax ................................................................. 470
    Binding rules......................................................................... 471
    Rules and NULL values.......................................................... 473
    Unbinding rules ....................................................................... 474
  Dropping rules .............................................................................. 475
  Getting information about defaults and rules .................. 475
  Sharing inline defaults............................................................ 476
    Creating an inline shared default................................. 476
    Unbinding a shared inline default................................. 477
    Limitations ............................................................................. 478
CHAPTER 15 Using Batches and Control-of-Flow Language ..................... 479
  Introduction ................................................................................. 479
  Rules associated with batches ................................................. 480
  Examples of using batches ...................................................... 481
  Batches submitted as files ....................................................... 484
  Using control-of-flow language ................................................. 485
    if...else .................................................................................. 486
    case expression ...................................................................... 488
    begin...end.......................................................................... 499
    while and break...continue .................................................. 500
    declare and local variables .................................................. 502
    goto .................................................................................... 503
    return .................................................................................. 503
    print ..................................................................................... 504
    raiserror .............................................................................. 506
    Creating messages for print and raiserror ......................... 506
    waitfor ................................................................................ 508
  Comments ................................................................................ 509
  Local variables .......................................................................... 511
    Declaring local variables ..................................................... 511
    Local variables and select statements .................................. 511
    Local variables and update statements ............................... 513
    Local variables and subqueries .......................................... 513
    Local variables and while loops and if...else blocks ............ 513
    Variables and null values .................................................... 514
  Global variables ....................................................................... 516
    Transactions and global variables ...................................... 516

CHAPTER 16 Using Transact-SQL Functions in Queries ..................... 519
  Setting up a query ..................................................................... 519
  Built-in functions ..................................................................... 520
    System functions .................................................................. 520
    String functions .................................................................. 520
    Text and image functions .................................................... 523
    Aggregate functions ............................................................ 524
    Statistical aggregate functions .......................................... 530
    Mathematical functions ....................................................... 531
    Date functions ..................................................................... 532
    Datatype conversion functions ............................................ 534
    Security functions .................................................................. 546
    XML functions ...................................................................... 546
    User-created functions ......................................................... 546
<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CHAPTER 17</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Using Stored Procedures</strong></td>
<td>549</td>
</tr>
<tr>
<td>How stored procedures work</td>
<td>549</td>
</tr>
<tr>
<td>Examples</td>
<td>550</td>
</tr>
<tr>
<td>Permissions</td>
<td>553</td>
</tr>
<tr>
<td>Performance</td>
<td>554</td>
</tr>
<tr>
<td>Creating and executing stored procedures</td>
<td>554</td>
</tr>
<tr>
<td>Using deferred name resolution</td>
<td>555</td>
</tr>
<tr>
<td>Parameters</td>
<td>556</td>
</tr>
<tr>
<td>Default parameters</td>
<td>558</td>
</tr>
<tr>
<td>Using more than one parameter</td>
<td>561</td>
</tr>
<tr>
<td>Using large object text, unitext, and image datatypes in stored</td>
<td></td>
</tr>
<tr>
<td>procedures</td>
<td>563</td>
</tr>
<tr>
<td>Procedure groups</td>
<td>564</td>
</tr>
<tr>
<td>Using with recompile in create procedure</td>
<td>565</td>
</tr>
<tr>
<td>Using with recompile in execute</td>
<td>566</td>
</tr>
<tr>
<td>Nesting procedures within procedures</td>
<td>566</td>
</tr>
<tr>
<td>Using temporary tables in stored procedures</td>
<td>567</td>
</tr>
<tr>
<td>Setting options in stored procedures</td>
<td>568</td>
</tr>
<tr>
<td>Executing stored procedures</td>
<td>569</td>
</tr>
<tr>
<td>Deferred compilation in stored procedures</td>
<td>571</td>
</tr>
<tr>
<td>Returning information from stored procedures</td>
<td>572</td>
</tr>
<tr>
<td>Return status</td>
<td>572</td>
</tr>
<tr>
<td>Checking roles in procedures</td>
<td>574</td>
</tr>
<tr>
<td>Return parameters</td>
<td>575</td>
</tr>
<tr>
<td>Restrictions associated with stored procedures</td>
<td>580</td>
</tr>
<tr>
<td>Qualifying names inside procedures</td>
<td>581</td>
</tr>
<tr>
<td>Renaming stored procedures</td>
<td>582</td>
</tr>
<tr>
<td>Renaming objects referenced by procedures</td>
<td>582</td>
</tr>
<tr>
<td>Using stored procedures as security mechanisms</td>
<td>583</td>
</tr>
<tr>
<td>Dropping stored procedures</td>
<td>583</td>
</tr>
<tr>
<td>System procedures</td>
<td>583</td>
</tr>
<tr>
<td>Executing system procedures</td>
<td>584</td>
</tr>
<tr>
<td>Permissions on system procedures</td>
<td>584</td>
</tr>
<tr>
<td>Types of system procedures</td>
<td>585</td>
</tr>
<tr>
<td>Other Sybase-supplied procedures</td>
<td>585</td>
</tr>
<tr>
<td>Getting information about stored procedures</td>
<td>585</td>
</tr>
<tr>
<td>Getting a report with sp_help</td>
<td>586</td>
</tr>
<tr>
<td>Viewing the source text of a procedure with sp_helptext</td>
<td>586</td>
</tr>
<tr>
<td>Identifying dependent objects with sp_depes</td>
<td>587</td>
</tr>
<tr>
<td>Identifying permissions with sp_helprotect</td>
<td>589</td>
</tr>
<tr>
<td><strong>CHAPTER 18</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Using Extended Stored Procedures</strong></td>
<td>591</td>
</tr>
<tr>
<td>Overview</td>
<td>591</td>
</tr>
<tr>
<td>XP Server</td>
<td>592</td>
</tr>
</tbody>
</table>
## Contents

Dynamic link library support .................................................. 593  
Open Server API ..................................................................... 594  
ESPs and permissions .......................................................... 596  
ESPs and performance .......................................................... 596  
Creating functions for ESPs ....................................................... 597  
Files for ESP development ..................................................... 597  
Open Server data structures .................................................... 597  
Open Server return codes ...................................................... 598  
Outline of a simple ESP function ............................................. 598  
ESP function example ............................................................ 599  
Building the DLL .................................................................... 603  
Registering ESPs ......................................................................... 606  
Using create procedure ......................................................... 606  
Using sp_addextendedproc ................................................... 607  
Removing ESPs ........................................................................... 608  
Renaming ESPs .................................................................... 608  
Executing ESPs ........................................................................... 608  
System ESPs ............................................................................. 609  
Getting information about ESPs ............................................. 610  
ESP exceptions and messages .................................................. 611  

### CHAPTER 19  Cursors: Accessing Data ........................................................... 613  
Selecting rows with cursors .................................................. 614  
Sensitivity and scrollability .................................................. 614  
Types of cursors ........................................................................... 615  
Cursor scope ................................................................................ 616  
Cursor scans and the cursor result set .................................... 617  
Making cursors updatable .................................................... 618  
Determining which columns can be updated ....................... 619  
How Adaptive Server processes cursors ................................. 621  
Monitoring cursor statements .............................................. 624  
Using declare cursor ............................................................. 625  
\textit{\texttt{declare}} cursor examples ............................................. 626  
Opening cursors ........................................................................... 627  
Fetching data rows using cursors ........................................ 628  
\textit{\texttt{fetch}} syntax ................................................................. 628  
Checking cursor status ........................................................... 630  
Getting multiple rows with each fetch .................................. 632  
Checking the number of rows fetched .................................. 633  
Updating and deleting rows using cursors .................................. 634  
Updating cursor result set rows .................................................. 634  
Deleting cursor result set rows .................................................. 635  
Closing and deallocating cursors ............................................. 636  
Scrollable and forward-only cursor examples .......................... 637  

---

Transact-SQL Users Guide  

xv
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward-only (default) cursors</td>
<td>637</td>
</tr>
<tr>
<td>Example table for scrollable cursors</td>
<td>639</td>
</tr>
<tr>
<td>Inensitive scrollable cursors</td>
<td>640</td>
</tr>
<tr>
<td>Semisensitive scrollable cursors</td>
<td>641</td>
</tr>
<tr>
<td>Using cursors in stored procedures</td>
<td>643</td>
</tr>
<tr>
<td>Cursors and locking</td>
<td>645</td>
</tr>
<tr>
<td>Cursor-locking options</td>
<td>646</td>
</tr>
<tr>
<td>Enhanced transaction support for updatable cursors</td>
<td>647</td>
</tr>
<tr>
<td>Getting information about cursors</td>
<td>648</td>
</tr>
<tr>
<td>Using browse mode instead of cursors</td>
<td>650</td>
</tr>
<tr>
<td>Browsing a table</td>
<td>650</td>
</tr>
<tr>
<td>Browse-mode restrictions</td>
<td>651</td>
</tr>
<tr>
<td>Timestamping a new table for browsing</td>
<td>651</td>
</tr>
<tr>
<td>Timestamping an existing table</td>
<td>651</td>
</tr>
<tr>
<td>Comparing timestamp values</td>
<td>652</td>
</tr>
</tbody>
</table>

**CHAPTER 20**

**Triggers: Enforcing Referential Integrity** .................................... 653

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>How triggers work</td>
<td>653</td>
</tr>
<tr>
<td>Using triggers versus integrity constraints</td>
<td>654</td>
</tr>
<tr>
<td>Creating triggers</td>
<td>655</td>
</tr>
<tr>
<td>create trigger syntax</td>
<td>656</td>
</tr>
<tr>
<td>Using triggers to maintain referential integrity</td>
<td>657</td>
</tr>
<tr>
<td>Testing data modifications against the trigger test tables</td>
<td>658</td>
</tr>
<tr>
<td>Insert trigger example</td>
<td>660</td>
</tr>
<tr>
<td>Delete trigger examples</td>
<td>661</td>
</tr>
<tr>
<td>Update trigger examples</td>
<td>663</td>
</tr>
<tr>
<td>Multirow considerations</td>
<td>668</td>
</tr>
<tr>
<td>Insert trigger example using multiple rows</td>
<td>668</td>
</tr>
<tr>
<td>Delete trigger example using multiple rows</td>
<td>669</td>
</tr>
<tr>
<td>Update trigger example using multiple rows</td>
<td>670</td>
</tr>
<tr>
<td>Conditional insert trigger example using multiple rows</td>
<td>671</td>
</tr>
<tr>
<td>Rolling back triggers</td>
<td>672</td>
</tr>
<tr>
<td>Global login triggers</td>
<td>674</td>
</tr>
<tr>
<td>Nesting triggers</td>
<td>674</td>
</tr>
<tr>
<td>Trigger self-recursion</td>
<td>675</td>
</tr>
<tr>
<td>Rules associated with triggers</td>
<td>677</td>
</tr>
<tr>
<td>Triggers and permissions</td>
<td>677</td>
</tr>
<tr>
<td>Trigger restrictions</td>
<td>678</td>
</tr>
<tr>
<td>Implicit and explicit null values</td>
<td>678</td>
</tr>
<tr>
<td>Triggers and performance</td>
<td>680</td>
</tr>
<tr>
<td>set commands in triggers</td>
<td>680</td>
</tr>
<tr>
<td>Renaming and triggers</td>
<td>680</td>
</tr>
<tr>
<td>Trigger tips</td>
<td>680</td>
</tr>
<tr>
<td>Disabling triggers</td>
<td>681</td>
</tr>
</tbody>
</table>
### Contents

Dropping triggers ................................................................. 682  
Getting information about triggers ........................................ 682  
  sp_help ........................................................................ 683  
  sp_helptext .................................................................... 683  
  sp_depends .................................................................... 684

**CHAPTER 21**  
**In-Row Off-Row LOB** ..................................................... 685  
Overview ............................................................................. 685  
Compressing in-row LOB columns ......................................... 686  
Migrating off-row LOB data to use in-row storage ..................... 686  
  In-row LOB columns and bulk copy ..................................... 687  
Examples of methods for migrating existing data ...................... 687  
Guidelines for selecting the in-row LOB length ......................... 692  
Downgrading with tables containing in-row LOB columns .......... 694

**CHAPTER 22**  
**Using instead of Triggers** .................................................. 695  
Inserted and deleted logical tables ........................................... 696  
Triggers and transactions ...................................................... 697  
Nesting and recursion ........................................................... 697  
  instead of insert triggers .................................................... 698  
    Example ........................................................................ 699  
  instead of update trigger .................................................. 701  
  instead of delete trigger .................................................... 703  
  searched and positioned update and delete ............................ 703  
Getting information about triggers ......................................... 706

**CHAPTER 23**  
**Transactions: Maintaining Data Consistency and Recovery**.... 709  
How transactions work .......................................................... 709  
  Transactions and consistency ............................................. 711  
  Transactions and recovery .................................................. 712  
Using transactions ............................................................... 712  
  Allowing data definition commands in transactions ............... 713  
  System procedures that are not allowed in transactions ........ 715  
  Beginning and committing transactions ............................... 715  
  Rolling back and saving transactions .................................. 715  
  Checking the state of transactions ...................................... 717  
  Nested transactions .......................................................... 719  
  Example of a transaction .................................................. 720  
Selecting the transaction mode and isolation level ...................... 721  
  Choosing a transaction mode ............................................ 721  
  Choosing an isolation level .............................................. 723  
Compliance with SQL standards ............................................. 731
Contents

Using the lock table command to improve performance ...... 731
Using transactions in stored procedures and triggers......... 732
Errors and transaction rollbacks........................................ 733
Transaction modes and stored procedures ...................... 736
Using cursors in transactions ............................................. 739
Issues to consider when using transactions....................... 740
Backup and recovery of transactions .............................. 741

CHAPTER 24 Locking Commands and Options .......................... 743
Setting a time limit on waiting for locks ........................... 743
wait/nowait option of the lock table command................... 743
Setting a session-level lock-wait limit ............................... 745
Setting a server-wide lock-wait limit ............................... 745
Information on the number of lock-wait timeouts ............... 746
Readpast locking for queue processing .............................. 746
Incompatible locks during readpast queries ....................... 746
Allpages-locked tables and readpast queries .................... 747
Effects of isolation levels select queries with readpast ........ 747
Data modification commands with readpast and isolation levels. 748
text, unitext, and image columns and readpast ................. 749
Readpast-locking examples .......................................... 749

APPENDIX A The pubs2 Database............................................. 751
Tables in the pubs2 database .................................................. 751
publishers table ............................................................... 751
authors table .................................................................... 752
titles table ....................................................................... 753
titleauthor table ............................................................. 754
salesdetail table ............................................................. 755
sales table ...................................................................... 757
stores table ..................................................................... 757
roysched table ............................................................... 758
discounts table ............................................................... 758
blurbs table .................................................................... 758
au_pix table .................................................................... 759
Diagram of the pubs2 database .......................................... 760

APPENDIX B The pubs3 Database............................................. 763
Tables in the pubs3 database .................................................. 763
publishers table ............................................................... 763
authors table .................................................................... 764
titles table ................................................................. 765
titleauthor table ........................................................ 766
salesdetail table ...................................................... 767
sales table ............................................................... 768
stores table ............................................................. 769
store_employees table ........................................... 769
roysched table ........................................................ 770
discounts table ........................................................ 770
blurbs table ............................................................ 770
Diagram of the pubs3 database .................................. 771

Index .................................................................................. 773
CHAPTER 1

SQL Building Blocks

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL in Adaptive Server</td>
<td>1</td>
</tr>
<tr>
<td>Compliance to ANSI standards</td>
<td>5</td>
</tr>
<tr>
<td>Naming conventions</td>
<td>10</td>
</tr>
<tr>
<td>Expressions in Adaptive Server</td>
<td>18</td>
</tr>
<tr>
<td>Transact-SQL extensions</td>
<td>25</td>
</tr>
<tr>
<td>Adaptive Server login accounts</td>
<td>30</td>
</tr>
<tr>
<td>isql utility</td>
<td>31</td>
</tr>
<tr>
<td>Displaying SQL text</td>
<td>33</td>
</tr>
</tbody>
</table>

SQL in Adaptive Server

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 Sybase® 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.
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.

**Note** If Java is enabled on your Adaptive 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. See *Java in Adaptive Server Enterprise*.

Adaptive Server includes the pubs2 and pubs3 sample databases, which are used for most examples in the Adaptive Server documentation. See Appendix A, “The pubs2 Database,” and Appendix B, “The pubs3 Database.”

**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:

```sql
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:

```sql
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:

```sql
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.
Tables, columns, and rows

In relational database management systems, users access and modify data 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.

Relational operations

The basic query operations in a relational system are 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:

```sql
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:

```sql
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.

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

### Compiled objects

Adaptive Server uses **compiled objects** to contain vital information about each database and to help you access and manipulate data. A compiled object is any object that requires entries in the `sysprocedures` table, including:

- 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, Adaptive Server:

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.

**Saving or restoring source text**

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

**Verifying and encrypting source text**

Adaptive Server allows you to verify the existence of source text, and encrypt the text if you choose. Use these commands when you are working with source text:

- `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_helpext` – 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.

**Compliance to ANSI standards**

Certain behaviors defined by the SQL standards are incompatible with Adaptive Server applications. Transact-SQL provides set options that allow you to toggle these behaviors.
Compliance to ANSI standards

By default, compliant behavior is enabled for all Embedded SQL™ precompiler applications. Other applications needing to match SQL standard behavior can use the options in Table 1-1 for entry-level ANSI SQL compliance. See the Reference Manual: Commands.

Table 1-1: Set command flags for entry-level ANSI SQL compliance

<table>
<thead>
<tr>
<th>Option</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>ansi_permissions</td>
<td>on</td>
</tr>
<tr>
<td>ansi_null</td>
<td>on</td>
</tr>
<tr>
<td>arithabort</td>
<td>off</td>
</tr>
<tr>
<td>arithabort numeric_truncation</td>
<td>on</td>
</tr>
<tr>
<td>arithignore</td>
<td>off</td>
</tr>
<tr>
<td>chained</td>
<td>on</td>
</tr>
<tr>
<td>close on endtran</td>
<td>on</td>
</tr>
<tr>
<td>fipsflagger</td>
<td>on</td>
</tr>
<tr>
<td>quoted_identifier</td>
<td>on</td>
</tr>
<tr>
<td>string_rtruncation</td>
<td>on</td>
</tr>
<tr>
<td>transaction isolation level</td>
<td>3</td>
</tr>
</tbody>
</table>

The following sections describe the differences between standard behavior and default Transact-SQL behavior.

Federal Information Processing Standards (FIPS) flagger

For customers writing applications that must conform to the ANSI SQL standard, Adaptive Server 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.

See Chapter 12, “Introduction to Security,” in the System Administration Guide: Volume 1 for information about FIPS.
Chained transactions and isolation levels

Adaptive Server 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. See Chapter 23, “Transactions: Maintaining Data Consistency and Recovery.”

Identifiers

To be compliant with entry-level ANSI SQL, identifiers cannot:

- Begin with a pound sign (#)
- Have more than 18 characters
- Contain lowercase letters

Adaptive Server supports delimited identifiers for table, view, and column names. Delimited identifiers are object names enclosed in double quotation marks. Using them allows you to avoid certain restrictions on object names.

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.

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.
Compliance to ANSI standards

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.

Permissions required for update and delete statements
The ansi_permissions set option determines permissions required for delete and update statements. When enabled, Adaptive Server 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.

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 Adaptive Server does not execute statements in the batch that follow the error-generating statement.

  If you set arithabort arith_overflow off, Adaptive Server 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, Adaptive Server truncates the query results and continues processing. For compliance with the ANSI SQL standard, enter set arithabort numeric_truncation on.
• arithignore arith_overflow determines whether Adaptive Server 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.

Note See the jConnect Programmers Reference Guide for information about handling warnings in JDBC code. See Programming Information > Handling error messages > Handling numeric overflows that are returned as warnings from ASE.

Synonymous keywords

Table 1-2 lists keywords that have been added for SQL standard compatibility that are synonymous with existing Transact-SQL keywords.

Table 1-2: ANSI-compatible keyword synonyms

<table>
<thead>
<tr>
<th>Current syntax</th>
<th>Additional syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>commit tran, commit transaction,</td>
<td>commit work, rollback work</td>
</tr>
<tr>
<td>rollback tran, rollback transaction</td>
<td></td>
</tr>
<tr>
<td>any</td>
<td>some</td>
</tr>
<tr>
<td>grant all</td>
<td>grant all privileges</td>
</tr>
<tr>
<td>revoke all</td>
<td>revoke all privileges</td>
</tr>
<tr>
<td>max (expression)</td>
<td>max ([all</td>
</tr>
<tr>
<td>min (expression)</td>
<td>min ([all</td>
</tr>
<tr>
<td>user_name function</td>
<td>user keyword</td>
</tr>
</tbody>
</table>

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 Adaptive Server evaluates null values in other kinds of SQL statements such as create table.
Naming conventions

The characters recognized by Adaptive Server 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.

The sections that follow describe the sets of characters that can be used for each part of a statement. The section on identifiers also describes naming conventions for database objects.

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 Adaptive Server 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 the following ASCII characters:
Table 1-3: ASCII characters used in SQL

<table>
<thead>
<tr>
<th>character</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>;</td>
<td>semicolon</td>
</tr>
<tr>
<td>(</td>
<td>open parenthesis</td>
</tr>
<tr>
<td>)</td>
<td>close parenthesis</td>
</tr>
<tr>
<td>,</td>
<td>comma</td>
</tr>
<tr>
<td>:</td>
<td>colon</td>
</tr>
<tr>
<td>%</td>
<td>percent sign</td>
</tr>
<tr>
<td>-</td>
<td>minus sign</td>
</tr>
<tr>
<td>?</td>
<td>question mark</td>
</tr>
<tr>
<td>'</td>
<td>single quote</td>
</tr>
<tr>
<td>&quot;</td>
<td>double quote</td>
</tr>
<tr>
<td>+</td>
<td>plus sign</td>
</tr>
<tr>
<td>_</td>
<td>underscore</td>
</tr>
<tr>
<td>*</td>
<td>asterisk</td>
</tr>
<tr>
<td>/</td>
<td>slash</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than operator</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than operator</td>
</tr>
<tr>
<td>=</td>
<td>equals operator</td>
</tr>
<tr>
<td>&amp;</td>
<td>ampersand</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>circumflex</td>
</tr>
<tr>
<td>[</td>
<td>left bracket</td>
</tr>
<tr>
<td>]</td>
<td>right bracket</td>
</tr>
<tr>
<td>@</td>
<td>at sign</td>
</tr>
<tr>
<td>~</td>
<td>tilde</td>
</tr>
<tr>
<td>!</td>
<td>exclamation point</td>
</tr>
<tr>
<td>$</td>
<td>dollar sign</td>
</tr>
<tr>
<td>#</td>
<td>number sign</td>
</tr>
<tr>
<td>.</td>
<td>period</td>
</tr>
</tbody>
</table>

Identifiers

Conventions for naming database objects apply throughout Adaptive Server 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. Table 1-4 specifies the byte limit for the different identifier types.

**Table 1-4: Byte limits for identifiers**

<table>
<thead>
<tr>
<th>255-byte-limit identifiers</th>
<th>30-byte-limit identifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>table name</td>
<td>cursor name</td>
</tr>
<tr>
<td>column name</td>
<td>server name</td>
</tr>
<tr>
<td>index name</td>
<td>host name</td>
</tr>
<tr>
<td>view name</td>
<td>login name</td>
</tr>
<tr>
<td>user-defined datatype</td>
<td>password</td>
</tr>
<tr>
<td>trigger name</td>
<td>host process identification</td>
</tr>
<tr>
<td>default name</td>
<td>application name</td>
</tr>
<tr>
<td>rule name</td>
<td>initial language name</td>
</tr>
<tr>
<td>constraint name</td>
<td>character set name</td>
</tr>
<tr>
<td>stored procedure name</td>
<td>user name</td>
</tr>
<tr>
<td>variable name</td>
<td>group name</td>
</tr>
<tr>
<td>JAR name</td>
<td>database name</td>
</tr>
<tr>
<td>Lightweight processes (LWPs) or dynamic statement name</td>
<td>cache name</td>
</tr>
<tr>
<td>function name</td>
<td>logical device name</td>
</tr>
<tr>
<td>time range name</td>
<td>segment name</td>
</tr>
<tr>
<td>function name</td>
<td>session name</td>
</tr>
<tr>
<td>application context name</td>
<td>execution class name</td>
</tr>
<tr>
<td></td>
<td>engine name</td>
</tr>
<tr>
<td></td>
<td>quiesce tag name</td>
</tr>
</tbody>
</table>

You must declare the first character of an identifier as an alphabetic character in the character set definition in use on Adaptive Server. 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, Adaptive Server 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, Adaptive Server 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 Adaptive Server automatically updates.

The case sensitivity of Adaptive Server 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 Adaptive Server:

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


### 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, Sybase 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.

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. (This restriction applies to Adaptive Server 11.5 and later versions.)

Before you create or reference a delimited identifier, execute:

```
set quoted_identifier on
```

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

```sql
create table "lone"(col1 char(3))
select * from "lone"
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 Adaptive Server to treat them as identifiers. For example, to insert a character string into `col1` of `lone table`, use:

```sql
insert "lone"(col1) values ('abc')
```

rather than:

```sql
insert "lone"(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:

```sql
insert "lone"(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:

```sql
set quoted identifier on
go
create table "1one" (cl 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

```sql
select object_id('lone')
```

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

```sql
create table "embedded""quote" (cl int)
```

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

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

Bracketed delimited identifiers

Sybase also supports bracketed identifiers, for which 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.

```sql
create table [bracketed identifier](cl int)
```

Support for brackets with delimited identifiers increases platform compatibility.

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 in Adaptive Server.
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, Adaptive Server 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:

\[ \text{pubs2.sharon.authors.city} \]

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

\[ \text{pubs2.dbo.titleview} \]
\[ \text{dbo.postalcoderule} \]

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:

\[ \text{database.owner."table_name"."column_name"} \]

rather than:

\[ \text{database.owner."table_name.column_name"} \]

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:

\[ [[\text{database.}]\text{owner.}]\text{object_name} \]

or:

\[ \text{[owner.]object_name} \]

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, Adaptive Server first looks at all the objects you own, and then at the objects owned by the database owner. As long as Adaptive Server has 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.
```

**Remote servers**

You can execute stored procedures on a remote Adaptive 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, Adaptive Server looks 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:
Expressions in Adaptive Server

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.

Arithmetic and character expressions

Adaptive Server includes several arithmetic and character expressions.

Operator precedence

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

1. unary (single argument) - + ~
2. * / %
When all operators in an expression are of the same level, the order of execution is left to right. Change the order of execution with parentheses—the most deeply nested expression is executed first.

### Arithmetic operators

Adaptive Server uses these arithmetic operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
</tr>
<tr>
<td>−</td>
<td>Subtraction</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td>%</td>
<td>Modulo (Transact-SQL extension)</td>
</tr>
</tbody>
</table>

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) Adaptive Server follows specific rules for determining the type of the result. See Chapter 6, “Using and Creating Datatypes.”

### 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.
Table 1-6 and Table 1-7 summarize the results for operands of 0 and 1. If either operand is NULL, the bitwise operator returns NULL:

**Table 1-6: Truth tables for bitwise operations**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Binary form</th>
<th>Result</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp; (and)</td>
<td>10101010</td>
<td>10001010</td>
<td>Result column equals 1 if both A and B are 1. Otherwise, result column equals 0.</td>
</tr>
<tr>
<td></td>
<td>01001011</td>
<td>00010101</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(or)</td>
<td>11101011</td>
<td>235</td>
</tr>
<tr>
<td></td>
<td>11101011</td>
<td>11101011</td>
<td></td>
</tr>
<tr>
<td>^ (exclusive or)</td>
<td>11100001</td>
<td>225</td>
<td>Result column equals 1 if either A or B, but not both, is 1.</td>
</tr>
<tr>
<td>~ (not)</td>
<td>1</td>
<td>FALSE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

**Table 1-7: Examples of bitwise operations**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Binary form</th>
<th>Result</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A &amp; B)</td>
<td>10101010</td>
<td>10</td>
<td>Result column equals 1 if both A and B are 1. Otherwise, result column equals 0.</td>
</tr>
<tr>
<td></td>
<td>01001011</td>
<td>00010101</td>
<td></td>
</tr>
<tr>
<td>(A</td>
<td>B)</td>
<td>10101010</td>
<td>235</td>
</tr>
<tr>
<td></td>
<td>01001011</td>
<td>11101011</td>
<td></td>
</tr>
<tr>
<td>(A ^ B)</td>
<td>10101010</td>
<td>225</td>
<td>Result column equals 1 if either A or B, but not both, is 1.</td>
</tr>
<tr>
<td></td>
<td>01001011</td>
<td>11100001</td>
<td></td>
</tr>
</tbody>
</table>
The string concatenation operator

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

```sql
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.”

```sql
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:

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

The comparison operators

Adaptive Server uses these comparison operators:
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 Adaptive 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"
```

### Nonstandard operators

The following operators are Transact-SQL extensions:

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

Adaptive Server 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.

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.

Quotation marks

There are two ways to specify literal quotes within a char or varchar entry. 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?"
Expressions in Adaptive Server

'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 Adaptive Server treats the data as an identifier. For more information about quoted identifiers, see “Delimited identifiers” on page 14.

### 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, all, and in**

*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. See Chapter 5, “Subqueries: Using Queries Within Other Queries.”

*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 and or

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.

Table 1-9 shows the results of logical operations, including those that involve null values:

<table>
<thead>
<tr>
<th>and</th>
<th>TRUE</th>
<th>FALSE</th>
<th>NULL</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>UNKNOWN</td>
</tr>
<tr>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td>NULL</td>
<td>UNKNOWN</td>
<td>FALSE</td>
<td>UNKNOWN</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>or</th>
<th>TRUE</th>
<th>FALSE</th>
<th>NULL</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>UNKNOWN</td>
</tr>
<tr>
<td>NULL</td>
<td>TRUE</td>
<td>UNKNOWN</td>
<td>UNKNOWN</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>not</th>
<th>TRUE</th>
<th>FALSE</th>
<th>NULL</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>NULL</td>
</tr>
<tr>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>UNKNOWN</td>
<td>TRUE</td>
<td>UNKNOWN</td>
</tr>
</tbody>
</table>

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.

### 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.

Most of the Transact-SQL enhancements (known as extensions) are summarized here. The Transact-SQL extensions for each command are in the *Reference Manual: Commands*. 

---

Transact-SQL Users Guide  25
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. The compute clause is discussed in Chapter 3, “Using Aggregates, Grouping, and Sorting.”

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, and 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.

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.

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.

**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 Adaptive Server 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 Adaptive-Server-supplied ESPs, called **system extended stored procedures**. One of these, *xp_cmdshell*, allows you to execute an operating system command from within Adaptive Server. Chapter 18, “Using Extended Stored Procedures,” describes ESPs. See also Chapter 3, “System Extended Stored Procedures,” Reference Manual: Procedures.

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

**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.

See Chapter 20, “Triggers: Enforcing Referential Integrity.”

**Defaults and rules**

Transact-SQL provides keywords for maintaining entity integrity to ensure a value is supplied for every column that requires one, and domain integrity to ensure each value in a column belongs to the set of legal values for that column. Defaults and rules define 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. Rules and defaults are discussed in Chapter 14, “Defining Defaults and Rules for Data.”

**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 such as `@@error`. 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.

`set options` customize the display of results, show processing statistics, and provide other diagnostic aids for debugging your Transact-SQL programs. All `set options` except `showplan` and `char_convert` take effect immediately.

See the *Reference Manual: Commands*. 
Additional Adaptive Server extensions to SQL

Transact-SQL includes these extensions to SQL:

- The following extensions to SQL search conditions: modulo operator (%), negative comparison operators (>, <, and =), bitwise operators (–, ^, |, and &), join operators (+= and =*), wildcard characters ([ ] and -), and the not operator (¬). See Chapter 2, “Queries: Selecting Data from a Table.”

- Fewer restrictions on the group by clause and the order by clause. See Chapter 3, “Using Aggregates, Grouping, and Sorting.”

- Subqueries, which can be used almost anywhere an expression is allowed. See Chapter 5, “Subqueries: Using Queries Within Other Queries.”

- Temporary tables and other temporary database objects, which exist only for the duration of the current work session. See Chapter 8, “Creating Databases and Tables.”


- The ability to insert data from a table into that same table. See Chapter 7, “Adding, Changing, Transferring, and Deleting Data.”

- The ability to extract data from one table and put it into another with the update command. See Chapter 7, “Adding, Changing, Transferring, and Deleting Data.”

- The ability to remove data based on data in other tables using the join in a delete statement. See Chapter 7, “Adding, Changing, Transferring, and Deleting Data.”

- A fast way to delete all rows in a specified table and reclaim the space they took up with the truncate table command. See Chapter 7, “Adding, Changing, Transferring, and Deleting Data.”

- Identity columns, which provide system-generated values that uniquely identify each row within a table. See Chapter 7, “Adding, Changing, Transferring, and Deleting Data.”

- 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. See Chapter 12, “Views: Limiting Access to Data.”

- Dozens of built-in functions. See Chapter 16, “Using Transact-SQL Functions in Queries.”
Adaptive Server login accounts

Each Adaptive Server user must have a login account that is established by a system security officer. Login accounts have:

- A login name, unique on that server.
- (Optional) A default database. If a default is defined, the user starts each Adaptive Server session in the defined database without having to issue the use command. If no default is defined, each session starts in the master database.
- (Optional) A default language. This specifies the language in which prompts and messages display. If a language is not defined, Adaptive Server’s default language, which is set at installation, is used.

Adaptive Server login accounts

- Options to the create index command for fine-tuning aspects of performance determined by indexes, and controlling the treatment of duplicate keys and rows. See Chapter 13, “Creating Indexes on Tables.”
- Control over what happens when a user attempts to enter duplicate keys in a unique index, or duplicate rows in a table. See Chapter 13, “Creating Indexes on Tables.”
- Bitwise operators for use with integer and bit type columns. See “Bitwise operators” on page 19 and Chapter 6, “Using and Creating Datatypes.”
- Support for text and image datatypes. See Chapter 6, “Using and Creating Datatypes.”
- The ability to gain access to both Sybase and non-Sybase 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.
• (Optional) A full name. This is your full name, which can be useful for documentation and identification purposes.

Use sp_displaylogin to view information about your own Adaptive Server 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 Chapter 14, “Managing Adaptive Server Logins, Database Users, and Client Connections,” in the System Administration Guide Volume 1

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 Chapter 13, “Getting Started With Security Administration in Adaptive Server,” in the System Administration Guide Volume 1.

You can execute stored procedures on a remote Adaptive Server using remote procedure calls if you have been granted access to the remote server and an appropriate database on that server. See Chapter 15, “Managing Remote Servers,” in the System Administration Guide Volume 1.

**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 Adaptive Server. 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:

```
quid
```

or:

```
exit
```

See the *Utility Guide*.

To connect to a non-Sybase 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*.

### Default databases

When your Adaptive Server account was created, you may have been assigned a default database to which you are 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 Adaptive Server login can be a guest. To change your default database, use `sp_modifylogin`, which is described in *Reference Manual: Procedures*.

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 Adaptive 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`. 

---

**isql utility**
Network-based security services with `isql`

You can specify the `-V` option of `isql` to use network-based security services such as unified login. With unified login, you can be authenticated with a security mechanism offered by a third-party provider and then log in to Adaptive Server without specifying a login name or a password.


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, Adaptive Server prints all SQL text 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.
CHAPTER 2

Queries: Selecting Data from a Table

The `select` command retrieves data stored in the rows and columns of database tables using a procedure called a **query**. A query has three main parts: the `select` clause, the `from` clause, and the `where` clause.

This chapter focuses on basic single-table `select` statements. Many sections contain sample statements that you can use to practice writing queries. If you want to integrate other Transact-SQL functionality, such as joins, subqueries, and aggregates, you can find more complex query examples later in this book.

### Queries

A SQL query requests data from the database. This process, also known as **data retrieval**, is expressed using the `select` statement. You can use it for **selections**, which retrieve a subset of the rows in one or more tables, and you can use it 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
```
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:

```sql
select au_fname, au_lname
from authors
where city = "Oakland"
```

Results of this query appear in columnar format:

<table>
<thead>
<tr>
<th>au_fname</th>
<th>au_lname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Marjorie</td>
<td>Green</td>
</tr>
<tr>
<td>Dick</td>
<td>Straight</td>
</tr>
<tr>
<td>Dirk</td>
<td>Stringer</td>
</tr>
<tr>
<td>Stearns</td>
<td>MacFeather</td>
</tr>
<tr>
<td>Livia</td>
<td>Karsen</td>
</tr>
</tbody>
</table>

(5 rows affected)

**select syntax**

The `select` syntax can be simpler or more complex than shown in the previous example. 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 documented 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. See the *Reference Manual: Commands*.

Use the clauses in a `select` statement in the order shown above. For example, 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
```

Since the examples in this chapter involve single-table queries, column names in syntax models and examples are usually not qualified with the names of the tables, owners, or databases to which they belong. These elements are omitted for readability; it is never wrong to include qualifiers. The remaining sections in this chapter analyze the syntax of the `select` statement in more detail.

This chapter describes only some of the clauses and keywords included in the syntax of the `select` command. The following clauses are discussed in other chapters:

- `group by`, `having`, `order by`, and `compute` are described in Chapter 3, “Using Aggregates, Grouping, and Sorting.”
- `into` is described in Chapter 8, “Creating Databases and Tables.”
- `at isolation` is described in Chapter 23, “Transactions: Maintaining Data Consistency and Recovery.”

The `holdlock`, `noholdlock`, and `shared` keywords (which deal with locking in Adaptive Server) and the `index` clause are described in Chapter 4, “Using Locking Commands,” in the Performance and Tuning Series: Locking and Concurrency Control. For information about the `for read only` and `for update` clauses, see the `declare cursor` command in the 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. See also “Using browse mode instead of cursors” on page 650.

### Checking for identifiers in a `select` statement

When the source text of a stored procedure or trigger is stored in the `syscomments` system table, 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>
```
Choosing columns using the select clause

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.

Choosing 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).

Choosing all columns using select *

The asterisk (*) selects all the column names in all the tables specified by the from clause. Use it to save typing time and errors when you want to see all the columns in a table. * retrieves the data 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:

<table>
<thead>
<tr>
<th>pub_id</th>
<th>pub_name</th>
<th>city</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>------</td>
<td>--------</td>
<td>----</td>
<td>-----</td>
</tr>
</tbody>
</table>
CHAPTER 2  Queries: Selecting Data from a Table

If you listed all the column names in the table in order after the select keyword, you would get exactly the same results:

```sql
select pub_id, pub_name, city, state
from publishers
```
You can also use `*` more than once in a query:

```sql
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:

```sql
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.

Choosing specific columns

Use this syntax to select only specific columns from a table, separating column names with commas:

```sql
select column_name[, column_name]...
from table_name
```
For example:

```sql
select au_lname, au_fname
from authors
```
Choosing columns using the select clause

Rearranging the column order

The order in which you list column names in the `select` clause determines the order in which the columns appear. 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
```

```
<table>
<thead>
<tr>
<th>pub_id</th>
<th>pub_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0736</td>
<td>New Age Books</td>
</tr>
<tr>
<td>0877</td>
<td>Binnet &amp; Hardley</td>
</tr>
<tr>
<td>1389</td>
<td>Algodata Infosystems</td>
</tr>
</tbody>
</table>
```

(3 rows affected)

```
select pub_name, pub_id
from publishers
```

```
<table>
<thead>
<tr>
<th>pub_name</th>
<th>pub_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>New Age Books</td>
<td>0736</td>
</tr>
<tr>
<td>Binnet &amp; Hardley</td>
<td>0877</td>
</tr>
<tr>
<td>Algodata Infosystems</td>
<td>1389</td>
</tr>
</tbody>
</table>
```

(3 rows affected)

Renaming columns in query results

In query results, the default heading for each column is the name given to it when it was created. You can rename a column heading for display purposes by using one of the following 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
```
chapter 2  queries: selecting data from a table

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:

<table>
<thead>
<tr>
<th>Publisher</th>
<th>pub_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>New Age Books</td>
<td>0736</td>
</tr>
<tr>
<td>Binnet &amp; Hardley</td>
<td>0877</td>
</tr>
<tr>
<td>Algodata Infosystems</td>
<td>1389</td>
</tr>
</tbody>
</table>

(3 rows affected)

Using expressions

The select statement can also include one or more expressions, which allow you to manipulate the data 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.

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.

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
```

<table>
<thead>
<tr>
<th>Publisher’s Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>New Age Books</td>
</tr>
<tr>
<td>Binnet &amp; Hardley</td>
</tr>
<tr>
<td>Algodata Infosystems</td>
</tr>
</tbody>
</table>
Choosing columns using the select clause

(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:

```sql
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.

Character strings in query results

The select statements in the examples thus far produce results showing data in the database. You can also 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:

```sql
select "The publisher’s name is", Publisher = pub_name
from publishers
```

<table>
<thead>
<tr>
<th>Publisher</th>
<th>Publisher's name is</th>
</tr>
</thead>
<tbody>
<tr>
<td>New Age Books</td>
<td>The publisher’s name is</td>
</tr>
<tr>
<td>Binnet &amp; Hardley</td>
<td>The publisher’s name is</td>
</tr>
<tr>
<td>Algodata Infosystems</td>
<td>The publisher’s name is</td>
</tr>
</tbody>
</table>

(3 rows affected)

Computed values in the select list

You can perform certain arithmetic operations on date/time columns using the date functions. See Chapter 16, “Using Transact-SQL Functions in Queries,” for information. 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:

```sql
select title_id, total_sales, total_sales * 2
```
Here are the results:

<table>
<thead>
<tr>
<th>title_id</th>
<th>total_sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>BU1032</td>
<td>4095</td>
</tr>
<tr>
<td>BU1111</td>
<td>3876</td>
</tr>
<tr>
<td>BU2075</td>
<td>18722</td>
</tr>
<tr>
<td>BU7832</td>
<td>4095</td>
</tr>
<tr>
<td>MC2222</td>
<td>2032</td>
</tr>
<tr>
<td>MC3021</td>
<td>22246</td>
</tr>
<tr>
<td>MC3026</td>
<td>NULL</td>
</tr>
<tr>
<td>PC1035</td>
<td>8780</td>
</tr>
<tr>
<td>PC8888</td>
<td>4095</td>
</tr>
<tr>
<td>PC9999</td>
<td>NULL</td>
</tr>
<tr>
<td>PS1372</td>
<td>375</td>
</tr>
<tr>
<td>PS2091</td>
<td>2045</td>
</tr>
<tr>
<td>PS2106</td>
<td>111</td>
</tr>
<tr>
<td>PS3333</td>
<td>4072</td>
</tr>
<tr>
<td>PS7777</td>
<td>3336</td>
</tr>
<tr>
<td>TC3218</td>
<td>375</td>
</tr>
<tr>
<td>TC4203</td>
<td>15096</td>
</tr>
<tr>
<td>TC7777</td>
<td>4095</td>
</tr>
</tbody>
</table>

(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:

```sql
select title_id, total_sales,
       proj_sales = total_sales * 2
from titles
```

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:

```sql
select title_id, total_sales * 2
```
Choosing columns using the select clause

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)

**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
```
Choosing columns using the `select` clause

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:

```sql
select title_id, (total_sales * price - advance) / 2
from titles
```

<table>
<thead>
<tr>
<th>title_id</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BU1032</td>
<td>38,429.53</td>
</tr>
<tr>
<td>BU1111</td>
<td>20,659.10</td>
</tr>
<tr>
<td>BU2075</td>
<td>22,926.89</td>
</tr>
<tr>
<td>BU7832</td>
<td>38,429.53</td>
</tr>
<tr>
<td>MC2222</td>
<td>20,309.84</td>
</tr>
<tr>
<td>MC3021</td>
<td>25,757.77</td>
</tr>
<tr>
<td>MC3026</td>
<td>NULL</td>
</tr>
<tr>
<td>PC1035</td>
<td>97,250.50</td>
</tr>
<tr>
<td>PC8888</td>
<td>36,950.00</td>
</tr>
<tr>
<td>PC9999</td>
<td>NULL</td>
</tr>
<tr>
<td>PS1372</td>
<td>548.13</td>
</tr>
<tr>
<td>PS2091</td>
<td>10,058.88</td>
</tr>
<tr>
<td>PS2106</td>
<td>-2,611.50</td>
</tr>
<tr>
<td>PS3333</td>
<td>39,699.64</td>
</tr>
<tr>
<td>PS7777</td>
<td>11,327.32</td>
</tr>
<tr>
<td>TC3218</td>
<td>428.13</td>
</tr>
<tr>
<td>TC4203</td>
<td>88,198.60</td>
</tr>
<tr>
<td>TC7777</td>
<td>26,692.03</td>
</tr>
</tbody>
</table>

(18 rows affected)

Selecting `text`, `untext`, and `image` values

`text`, `untext`, and `image` values can be quite large. When a select list includes `text`, `untext`, `image` values, 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 Adaptive Server; the default value for `isql` is 32K. To change the value, use the `set` command:

```sql
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 Adaptive Server default value, use:

```
set textsize 0
```

If the actual length of returned data is less than `textsize`, the entire data string appears. See Chapter 6, “Using and Creating Datatypes.”

### Using `readtext`

The `readtext` command provides a way to retrieve `text`, `unitext`, and `image` values to 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, Adaptive Server 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.

See “Text and image functions” on page 523 for an advanced discussion of the `readtext` command.
Choosing columns using the select clause

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:
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

You can also include aggregate functions, which are discussed in Chapter 3,
“Using Aggregates, Grouping, and Sorting.”

48

Adaptive Server Enterprise


Using **select for update**

Adaptive Server versions 15.7 and later support **select for update** to exclusively lock rows in datarows-locked tables for subsequent updates within the same transaction, and for updatable cursors. This prevents other concurrent tasks from updating these rows and from blocking the subsequent update. **select for update** is supported at isolation levels 1, 2, and 3.

In versions earlier than 15.7, a **select** statement with a **for update** clause can be issued only within a **declare cursor** statement.

In Adaptive Server versions 15.7 and later, **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. Otherwise, Adaptive Server reverts to pre-15.7 functionality.

**Syntax**

```sql
select <col-list> from ... where ...
   [for update[ of col-list ]]
```

**Note** To obtain version 15.7 functionality, and exclusive locks, you must set the **select for update** configuration parameter to 1, and include the **for update** clause. If the configuration parameter is not set, Adaptive Server reverts to pre-15.7 functionality.

---

Using **select for update** in cursors and DML

**select for update** functionality is based on the value of the configuration parameter **select for update**:

- 0 – Adaptive Server version pre-15.7 functionality applies. **select for update** is available only through cursors.
- 1 – Adaptive Server version 15.7 functionality applies. You can use **select for update** at the language level, outside of a cursor context.

In versions earlier than 15.7:

- **select for update** is supported only for cursors.
- Update locks are acquired for qualifying rows and block writers but not other readers.
Using select for update

- You cannot use an order by clause with an updatable cursor because the clause automatically makes a cursor read-only.

In version 15.7 and later:
- 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.

For details on select for update syntax and usage, and for cursor scope and usage, see the Reference Manual: Commands.

Concurrency issues

If a session has an open transaction 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.

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Isolation Level</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>select qualified rows</td>
<td>Not blocked</td>
</tr>
<tr>
<td>select unqualified rows</td>
<td>Not blocked</td>
</tr>
<tr>
<td>update qualified rows</td>
<td>Blocked¹</td>
</tr>
<tr>
<td>update unqualified rows</td>
<td>Not blocked²</td>
</tr>
<tr>
<td>select for update qualified rows</td>
<td>N/A³</td>
</tr>
</tbody>
</table>

¹ Adaptive Server 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, Adaptive Server prevents “phantom rows” by exclusively locking more rows than just the qualifying rows. In this case, Adaptive Server blocks the second session on these additional unqualified rows.
³ Although the second session issues these DMLs at isolation level 0, Adaptive Server executes them at isolation level 2.
⁴ Adaptive Server does not support select for update at isolation level 0.
Eliminating duplicate query results with `distinct`

The optional `distinct` keyword eliminates duplicate rows from the default results of a `select` statement.

For compatibility with other implementations of SQL, Adaptive Server syntax allows the use of `all` to explicitly ask for all rows. The default for `select` statements is `all`. If you do not specify `distinct`, you will get, by default, all rows, including duplicates.

For example, here is the result of searching for all the author identification codes in the `titleauthor` table without `distinct`:

```sql
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
```
Eliminating duplicate query results with distinct

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. See “order by and group by used with select distinct” on page 103.

**Specifying tables with the from clause**

The from clause is required in every select statement involving 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 50 tables and 46 worktables (such as those created by aggregate functions). The 50-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

See the 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 Adaptive Server adds an internal numeric suffix of 17 bytes to ensure that the name is unique. See Chapter 8, “Creating Databases and Tables.”

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.
Selecting rows using the where clause

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.

Selecting 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 \ast 2 > total_sales \ast price
  ```

- Ranges (\(between\) and \(not\ between\))
  
  ```
  where total_sales between 4095 and 12000
  ```

- Lists (\(in\) and \(not\ in\))
  
  ```
  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, or)
  
  ```
  where advance < 5000 or total_sales between 2000 and 2500
  ```

The **where** keyword can also introduce:

- Join conditions (see Chapter 4, “Joins: Retrieving Data from Several Tables”)

54 Adaptive Server Enterprise
• Subqueries (see Chapter 5, “Subqueries: Using Queries Within Other Queries”)

**Note** The only where condition that you can use on text columns is \_\_\_like (or not like).

Adaptive Server does not necessarily evaluate and execute predicates in left-to-right order. Instead, Adaptive Server can evaluate and execute predicates in any order. For example, for this query:

```sql
where x != 0 and y = 10 or z = 100
```

Adaptive Server may not evaluate and execute `x != 0` first.

For more information on search conditions, see the *Reference Manual: Commands*.

**Comparison operators in where clauses**

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. Place apostrophes or quotation marks around all char, nchar, unichar, unitext, varchar, nvarchar, univarchar, text, and date/time data:

```sql
select *
from titleauthor
where royaltyper < 50
```

```sql
select authors.au_lname, authors.au_fname
from authors
where au_lname > "McBadden"
```

```sql
select au_id, phone
from authors
where phone != "415 658-9932"
```

```sql
select title_id, newprice = price * 1.15
from pubs2..titles
where advance > 5000
```
Selecting rows using the where clause

For more information on entering date and time data, see Chapter 7, “Adding, Changing, Transferring, and Deleting Data.”

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 (!>).

```sql
select title_id, type, advance
from titles
where (type = "business" or type = "psychology")
and not advance >5500
```

```sql
select title_id, type, advance
from titles
where (type = "business" or type = "psychology")
and advance !>5500
```

Both return the same result set:

<table>
<thead>
<tr>
<th>title_id</th>
<th>type</th>
<th>advance</th>
</tr>
</thead>
<tbody>
<tr>
<td>BU1032</td>
<td>business</td>
<td>5,000.00</td>
</tr>
<tr>
<td>BU1111</td>
<td>business</td>
<td>5,000.00</td>
</tr>
<tr>
<td>BU7832</td>
<td>business</td>
<td>5,000.00</td>
</tr>
<tr>
<td>PS2091</td>
<td>psychology</td>
<td>2,275.00</td>
</tr>
<tr>
<td>PS3333</td>
<td>psychology</td>
<td>2,000.00</td>
</tr>
<tr>
<td>PS7777</td>
<td>psychology</td>
<td>4,000.00</td>
</tr>
</tbody>
</table>

(6 rows affected)

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:

```sql
select title_id, total_sales
from titles
where total_sales between 4095 and 12000
```

<table>
<thead>
<tr>
<th>title_id</th>
<th>total_sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>BU1032</td>
<td>4095</td>
</tr>
<tr>
<td>BU7832</td>
<td>4095</td>
</tr>
<tr>
<td>PC1035</td>
<td>8780</td>
</tr>
</tbody>
</table>
You can specify an exclusive range using the greater than (>) and less than (<) operators:

```sql
select title_id, total_sales
from titles
where total_sales > 4095 and total_sales < 12000
```

<table>
<thead>
<tr>
<th>title_id</th>
<th>total_sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC1035</td>
<td>8780</td>
</tr>
</tbody>
</table>

(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:

```sql
select title_id, total_sales
from titles
where total_sales not between 4095 and 12000
```

<table>
<thead>
<tr>
<th>title_id</th>
<th>total_sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>BU1111</td>
<td>3876</td>
</tr>
<tr>
<td>BU2075</td>
<td>18722</td>
</tr>
<tr>
<td>MC2222</td>
<td>2032</td>
</tr>
<tr>
<td>MC3021</td>
<td>22246</td>
</tr>
<tr>
<td>PS1372</td>
<td>375</td>
</tr>
<tr>
<td>PS2091</td>
<td>2045</td>
</tr>
<tr>
<td>PS2106</td>
<td>111</td>
</tr>
<tr>
<td>PS3333</td>
<td>4072</td>
</tr>
<tr>
<td>PS7777</td>
<td>3336</td>
</tr>
<tr>
<td>TC3218</td>
<td>375</td>
</tr>
<tr>
<td>TC4203</td>
<td>15096</td>
</tr>
</tbody>
</table>

(11 rows affected)
Lists (in and not in)

The in keyword allows you to select values that match any one of a list of values. 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, untext, 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
Panteeley 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. See Chapter 5, “Subqueries: Using Queries Within Other Queries.”
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 \textit{in}, but without listing the two tables in the same \textit{from} clause, you can extract the information you need.

The following query:

- Searches the titleauthor table for all au_ids 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_ids that match the results from the titleauthor query. The results show that several authors fall into the less than 50 percent category.

\begin{verbatim}
select au_lname, au_fname 
from authors 
where au_id in 
  (select au_id 
   from titleauthor 
   where royaltyper <50)
\end{verbatim}

\begin{tabular}{ll}
au_lname & au_fname \\
------------------ & ------------ \\
Green & Marjorie \\
O’Leary & Michael \\
Gringlesby & Burt \\
Yokomoto & Akiko \\
MacFeather & Stearns \\
Ringer & Anne \\
\end{tabular}

(6 rows affected)

\begin{verbatim}
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)
\end{verbatim}

\begin{tabular}{ll}
au_lname & au_fname \\
------------------ & ------------ \\
White & Johnson \\
Carson & Cheryl \\
Straight & Dick \\
\end{tabular}
Matching patterns

You can include wildcard characters in the where clause to search for unknown characters or to group data according to common features. This section describes pattern matching using SQL and Transact-SQL. For more information on pattern matching, see the Reference Manual: Commands.

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, unichar, unitext, univarchar binary, varbinary, and date/time data.

The syntax for like is:

```sql
(where | having) [not] column_name [not] like "match_string"
```

match_string can include the symbols in Table 2-2:
Table 2-2: Special symbols for matching character strings

<table>
<thead>
<tr>
<th>Symbols</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td>Matches any string of zero or more characters.</td>
</tr>
<tr>
<td>_</td>
<td>Matches a single character.</td>
</tr>
<tr>
<td>[specifier]</td>
<td>Brackets enclose ranges or sets, such as [a – f] or [abcdef]. specifier can take two forms:</td>
</tr>
<tr>
<td></td>
<td>• rangespec1 – rangespec2:</td>
</tr>
<tr>
<td></td>
<td>• rangespec1 indicates the start of a range of characters.</td>
</tr>
<tr>
<td></td>
<td>• – is a special character, indicating a range.</td>
</tr>
<tr>
<td></td>
<td>• rangespec2 indicates the end of a range of characters.</td>
</tr>
<tr>
<td></td>
<td>• 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.</td>
</tr>
<tr>
<td>^[specifier]</td>
<td>A caret (^) preceding a specifier indicates noninclusion.[^a – f] means “not in the range a – f”;[^a2bR] means “not a, 2, b, or R.”</td>
</tr>
</tbody>
</table>

You can match the column data to constants, variables, or other columns that contain the wildcard characters shown in Table 2-2. 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 “[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:

```sql
SELECT phone
```
Matching patterns

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”:

```sql
select * from blurbs
where copy like "%computer%"
```

Adaptive Server 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.

```sql
select phone
from authors
where phone = "415%"
```

When you use like with datetime values, Adaptive Server 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 Adaptive Server converts the entry into “Jan 1 1900 9:20AM”:

```sql
where arrival_time = "9:20"
```

However, this query finds the 9:20 value:

```sql
where arrival_time like "%9:20%"
```

You can also use the date and time datatypes for like transactions.

Using not like

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:

```sql
select phone
from authors
where phone not like "415%"
```

```sql
select phone
from authors
```
where not phone like "415%"

**Getting 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”:

```sql
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.

**Using wildcard characters as literal characters**

You can search for wildcard characters by escaping them 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.
Matching patterns

Table 2-3: Using square brackets to search for wildcard characters

<table>
<thead>
<tr>
<th>like clause</th>
<th>Searches for</th>
</tr>
</thead>
<tbody>
<tr>
<td>like &quot;5%&quot;</td>
<td>5 followed by any string of 0 or more characters</td>
</tr>
<tr>
<td>like &quot;5(%)&quot;</td>
<td>5%</td>
</tr>
<tr>
<td>like &quot;_.n&quot;</td>
<td>an, in, on, and so forth</td>
</tr>
<tr>
<td>like &quot;_[n]&quot;</td>
<td>_n</td>
</tr>
<tr>
<td>like &quot;[a-cdf]&quot;</td>
<td>a, b, c, d, or f</td>
</tr>
<tr>
<td>like &quot;[-acdf]&quot;</td>
<td>-, a, c, d, or f</td>
</tr>
<tr>
<td>like &quot;[[[]]]&quot;</td>
<td>[</td>
</tr>
</tbody>
</table>

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.

Table 2-4: Using the escape clause

<table>
<thead>
<tr>
<th>like clause</th>
<th>Searches for</th>
</tr>
</thead>
<tbody>
<tr>
<td>like &quot;5@%&quot; escape &quot;@&quot;</td>
<td>5%</td>
</tr>
<tr>
<td>like &quot;_.n&quot; escape &quot;***&quot;</td>
<td>_n</td>
</tr>
<tr>
<td>like &quot;%80@%%&quot; escape &quot;@&quot;</td>
<td>String containing 80%</td>
</tr>
<tr>
<td>like &quot;<em>_sql</em>%&quot; escape &quot;***&quot;</td>
<td>String containing _sql*</td>
</tr>
<tr>
<td>like &quot;%#####_##%&quot; escape &quot;#&quot;</td>
<td>String containing ##_%</td>
</tr>
</tbody>
</table>

An escape character is valid only within its like clause and has no effect on other like clauses in the same statement.

The only characters that are valid following an escape character are the wildcard characters (\_, \%, [ ], and [^]), and the escape character itself. The escape character affects only the character following it. If a pattern contains two literal occurrences of a character that happens to be an escape character, the string must contain four consecutive escape characters (see the last example in Table 2-4). Otherwise, Adaptive Server raises a SQLSTATE error condition and returns an error message.

Specifying more than one escape character raises a SQLSTATE error condition, and Adaptive Server returns an error message:

```
like "%XX_\%" escape "XX"
like "%XX\%_XX" escape "XX"
```
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.

Using trailing blanks and %

Adaptive Server 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.

Using 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:

```sql
create table special_discounts
(id_type char(3), discount int)
insert into special_discounts
values("BU%", 10)
...
```

The table should contain the following data:

<table>
<thead>
<tr>
<th>id_type</th>
<th>discount</th>
</tr>
</thead>
<tbody>
<tr>
<td>BU%</td>
<td>10</td>
</tr>
<tr>
<td>PS%</td>
<td>12</td>
</tr>
<tr>
<td>MC%</td>
<td>15</td>
</tr>
</tbody>
</table>

The following query uses wildcard characters in id_type in the where clause:
Matching patterns

```sql
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:

<table>
<thead>
<tr>
<th>title_id</th>
<th>discount</th>
<th>price</th>
<th>price - (price*discount/100)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BU1032</td>
<td>10</td>
<td>19.99</td>
<td>17.99</td>
</tr>
<tr>
<td>BU1111</td>
<td>10</td>
<td>11.95</td>
<td>10.76</td>
</tr>
<tr>
<td>BU2075</td>
<td>10</td>
<td>2.99</td>
<td>2.69</td>
</tr>
<tr>
<td>BU7832</td>
<td>10</td>
<td>19.99</td>
<td>17.99</td>
</tr>
<tr>
<td>PS1372</td>
<td>12</td>
<td>21.59</td>
<td>19.00</td>
</tr>
<tr>
<td>PS2091</td>
<td>12</td>
<td>10.95</td>
<td>9.64</td>
</tr>
<tr>
<td>PS2106</td>
<td>12</td>
<td>7.00</td>
<td>6.16</td>
</tr>
<tr>
<td>PS3333</td>
<td>12</td>
<td>19.99</td>
<td>17.59</td>
</tr>
<tr>
<td>PS7777</td>
<td>12</td>
<td>7.99</td>
<td>7.03</td>
</tr>
<tr>
<td>MC2222</td>
<td>15</td>
<td>19.99</td>
<td>16.99</td>
</tr>
<tr>
<td>MC3021</td>
<td>15</td>
<td>2.99</td>
<td>2.54</td>
</tr>
<tr>
<td>MC3026</td>
<td>15</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

(12 rows affected)

This type of example permits sophisticated pattern matching without having to construct a series of or clauses.

**“Unknown” values: NULL**

A NULL 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, Adaptive Server 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"
```

<table>
<thead>
<tr>
<th>title_id</th>
<th>type</th>
<th>advance</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC2222</td>
<td>mod_cook</td>
<td>0.00</td>
</tr>
<tr>
<td>MC3021</td>
<td>mod_cook</td>
<td>15,000.00</td>
</tr>
<tr>
<td>MC3026</td>
<td>UNDECIDED</td>
<td>NULL</td>
</tr>
<tr>
<td>PS1372</td>
<td>psychology</td>
<td>7,000.00</td>
</tr>
<tr>
<td>TC3218</td>
<td>trad_cook</td>
<td>7,000.00</td>
</tr>
<tr>
<td>TC4203</td>
<td>trad_cook</td>
<td>4,000.00</td>
</tr>
<tr>
<td>TC7777</td>
<td>trad_cook</td>
<td>8,000.00</td>
</tr>
</tbody>
</table>

(7 rows affected)

Testing a column for null values

Use is null in where, if, and while clauses (discussed in Chapter 15, “Using Batches and Control-of-Flow Language”) 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
```

Adaptive Server 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:
Matching patterns

- `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):

```sql
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: <, <=, !<, >, >=, !>.

Adaptive Server can determine that a column value is NULL. Thus, this is considered true:

\[ \text{column1} = \text{NULL} \]

However, the following comparisons can never be determined, since NULL means “having an unknown value:”

\[ \text{where column1} > \text{null} \]

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:

\[ \text{where column\_name is [not] null} \]

For example:

\[ \text{where advance} < 5000 \text{ or advance is null} \]

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.

\[ \text{select title\_id, advance}
\text{from titles}
\text{where advance} < 5000 \]

<table>
<thead>
<tr>
<th>title_id</th>
<th>advance</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC2222</td>
<td>0.00</td>
</tr>
<tr>
<td>PS2091</td>
<td>2,275.00</td>
</tr>
<tr>
<td>PS3333</td>
<td>2,000.00</td>
</tr>
<tr>
<td>PS7777</td>
<td>4,000.00</td>
</tr>
<tr>
<td>TC4203</td>
<td>4,000.00</td>
</tr>
</tbody>
</table>

(5 rows affected)
Here is a query for books with an advance of less than 5000 or a null value in the advance column:

```sql
select title_id, advance
from titles
where advance < 5000
  or advance is null
```

<table>
<thead>
<tr>
<th>title_id</th>
<th>advance</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC2222</td>
<td>0.00</td>
</tr>
<tr>
<td>MC3026</td>
<td>NULL</td>
</tr>
<tr>
<td>PC9999</td>
<td>NULL</td>
</tr>
<tr>
<td>PS2091</td>
<td>2,275.00</td>
</tr>
<tr>
<td>PS3333</td>
<td>2,000.00</td>
</tr>
<tr>
<td>PS7777</td>
<td>4,000.00</td>
</tr>
<tr>
<td>TC4203</td>
<td>4,000.00</td>
</tr>
</tbody>
</table>

(7 rows affected)

See Chapter 8, “Creating Databases and Tables,” for information on NULL in the create table statement and for information on the relationship between NULL and defaults. See Chapter 7, “Adding, Changing, Transferring, and Deleting Data,” for information on inserting null values into a table.

**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.

**Substituting 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:

```sql
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.

```sql
select isnull(notes, "unknown")
from titles
```
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
```

Concatenating strings and NULL

If you concatenate a string and NULL, the expression evaluates to the string. For example:

```
select "abc" + NULL + "def"
```

```
abcdef
```

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.

Connecting 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
```
Matching patterns

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.

```sql
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.

```sql
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:

```sql
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:

```sql
select * from authors
where (city = "Oakland" or city = "Berkeley") and state = "CA"
```

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. See “Bitwise operators” on page 19.

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.

```sql
select title_id, type, advance
```
from titles
where type = "business" or type = "psychology"
   and advance > 5500

<table>
<thead>
<tr>
<th>title_id</th>
<th>type</th>
<th>advance</th>
</tr>
</thead>
<tbody>
<tr>
<td>BU1032</td>
<td>business</td>
<td>5,000.00</td>
</tr>
<tr>
<td>BU1111</td>
<td>business</td>
<td>5,000.00</td>
</tr>
<tr>
<td>BU2075</td>
<td>business</td>
<td>10,125.00</td>
</tr>
<tr>
<td>BU7832</td>
<td>business</td>
<td>5,000.00</td>
</tr>
<tr>
<td>PS1372</td>
<td>psychology</td>
<td>7,000.00</td>
</tr>
<tr>
<td>PS2106</td>
<td>psychology</td>
<td>6,000.00</td>
</tr>
</tbody>
</table>

(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:

```sql
select title_id, type, advance
from titles
where (type = "business" or type = "psychology")
   and advance > 5500
```

<table>
<thead>
<tr>
<th>title_id</th>
<th>type</th>
<th>advance</th>
</tr>
</thead>
<tbody>
<tr>
<td>BU2075</td>
<td>business</td>
<td>10,125.00</td>
</tr>
<tr>
<td>PS1372</td>
<td>psychology</td>
<td>7,000.00</td>
</tr>
<tr>
<td>PS2106</td>
<td>psychology</td>
<td>6,000.00</td>
</tr>
</tbody>
</table>

(3 rows affected)

Using multiple select items in a nested exists query

The use of multiple columns in this example is the same as selecting a single c1 or c2 column in the nested exists query.

```sql
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:

```sql
1> select * from t1
   where exists (select t2.*, c1 from t2)
2> go
```

Msg 102, Level 15, State 1:
Line 1:
Incorrect syntax near ','.

---

**Using 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:

```sql
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 '%$attr$')
```

Adaptive Server ignores the alias (the allowed column heading) in this example.
CHAPTER 3

Using Aggregates, Grouping, and Sorting

This chapter addresses the sum, avg, count, count(*), count_big, count_big(*), max, and min aggregate functions that enable you to summarize the data retrieved in a query. This chapter also discusses how to organize data into categories and subgroups using the group by, having, and order by clauses. The compute clause and the union operator, two Transact-SQL extensions, are also discussed.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using aggregate functions</td>
<td>75</td>
</tr>
<tr>
<td>Organizing query results into groups: the group by clause</td>
<td>82</td>
</tr>
<tr>
<td>Selecting groups of data: the having clause</td>
<td>94</td>
</tr>
<tr>
<td>Sorting query results: the order by clause</td>
<td>100</td>
</tr>
<tr>
<td>Summarizing groups of data: the compute clause</td>
<td>104</td>
</tr>
<tr>
<td>Combining queries: the union operator</td>
<td>112</td>
</tr>
</tbody>
</table>

If your Adaptive 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.

Using aggregate functions

The aggregate functions are: sum, avg, count, min, max, count_big, count(*), and count_big(*). You can use aggregate functions to calculate and summarize data. For example, to find out the number of books sold in the titles table of the pubs2 database, enter:

```sql
select sum(total_sales)
from titles
```

-------------

97746
Using aggregate functions

There is no column heading for the aggregate column in the example. An aggregate function takes 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, Adaptive Server generates a single value.

Here is the syntax of the aggregate function:

\[
\text{aggregate\_function}\ (\ [\text{all} | \text{distinct}] \ \text{expression})
\]

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.

<table>
<thead>
<tr>
<th>Aggregate function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>`sum([all</td>
<td>distinct] expression)`</td>
</tr>
<tr>
<td>`avg([all</td>
<td>distinct] expression)`</td>
</tr>
<tr>
<td>`count([all</td>
<td>distinct] expression)`</td>
</tr>
<tr>
<td>`count_big ([all</td>
<td>distinct] expression)`</td>
</tr>
<tr>
<td><code>count(*)</code></td>
<td>Number of selected rows as an integer</td>
</tr>
<tr>
<td><code>count_big(*)</code></td>
<td>Number of selected rows as a bigint</td>
</tr>
<tr>
<td><code>max(expression)</code></td>
<td>Highest value in the expression</td>
</tr>
<tr>
<td><code>min(expression)</code></td>
<td>Lowest value in the expression</td>
</tr>
</tbody>
</table>
You can use the aggregate functions in a select list, as shown in the previous example, or in the having clause. For information about the having clause, see “Selecting groups of data: the having clause” on page 94.

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 (see “Organizing query results into groups: the group by clause” on page 82), 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"
```

<table>
<thead>
<tr>
<th>advance and sales</th>
<th>6,281.25</th>
<th>30788</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1 row affected)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Aggregate functions and datatypes**

You can use the aggregate functions with any type of column, with these exceptions:

- 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:
Using aggregate functions

```sql
select min(au_lname)
from authors
------------------------
Bennet

(1 row affected)
```

However, you cannot average the contents of a text column:

```sql
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.
```

**count vs. count(*)**

While 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:

```sql
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:

```sql
select count(*), avg(price) from titles
where advance > 1000
---------- --------
15 14.42

(1 row affected)
```
Aggregate functions with *distinct*

You can use the optional keyword *distinct* only with `sum`, `avg`, `count_big`, and `count`. When you use *distinct*, Adaptive Server 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:

```sql
select count(distinct city)
from authors
```

```plaintext
-------------
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:

```sql
select avg(price)
from titles
where type = "business"
```

```plaintext
-------------
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:

```sql
select avg(distinct price)
from titles
where type = "business"
```

```plaintext
-------------
11.64
```

(1 row affected)
Null values and the aggregate functions

Adaptive Server ignores any null values in the column on which the 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, Adaptive Server 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:

```sql
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:

```sql
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)
```
Using statistical aggregates

Aggregate functions summarize data over a group of rows from the database. 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 summarize data over a group of rows from the database.

Adaptive Server 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. See the Reference Manual: Building Blocks

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.

Standard deviation and variance

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.

*See Reference Manual: Building Blocks.*

---

**Organizing 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:

```sql
select type, advance
from titles
group by type
```

<table>
<thead>
<tr>
<th>type</th>
<th>advance</th>
</tr>
</thead>
<tbody>
<tr>
<td>popular comp</td>
<td>7,000.00</td>
</tr>
<tr>
<td>popular comp</td>
<td>8,000.00</td>
</tr>
<tr>
<td>popular comp</td>
<td>NULL</td>
</tr>
<tr>
<td>business</td>
<td>5,000.00</td>
</tr>
<tr>
<td>business</td>
<td>5,000.00</td>
</tr>
<tr>
<td>business</td>
<td>10,125.00</td>
</tr>
<tr>
<td>mod_cook</td>
<td>0.00</td>
</tr>
<tr>
<td>mod_cook</td>
<td>15,000.00</td>
</tr>
<tr>
<td>trad_cook</td>
<td>7,000.00</td>
</tr>
</tbody>
</table>
CHAPTER 3 Using Aggregates, Grouping, and Sorting

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:

```sql
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 “Using aggregate functions” on page 75).

See the Reference Manual: Commands.

**group by and SQL standards**

The SQL standards for group by are more restrictive than the Sybase 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.

### Nesting 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
```

<table>
<thead>
<tr>
<th>pub_id</th>
<th>type</th>
<th>avg(price)</th>
<th>sum(total_sales)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0736</td>
<td>business</td>
<td>2.99</td>
<td>18,722</td>
</tr>
<tr>
<td>0736</td>
<td>psychology</td>
<td>11.48</td>
<td>9,564</td>
</tr>
<tr>
<td>0877</td>
<td>UNDECIDED</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>0877</td>
<td>mod_cook</td>
<td>11.49</td>
<td>24,278</td>
</tr>
<tr>
<td>0877</td>
<td>psychology</td>
<td>21.59</td>
<td>375</td>
</tr>
<tr>
<td>0877</td>
<td>trad_cook</td>
<td>15.96</td>
<td>19,566</td>
</tr>
<tr>
<td>1389</td>
<td>business</td>
<td>17.31</td>
<td>12,066</td>
</tr>
<tr>
<td>1389</td>
<td>popular_comp</td>
<td>21.48</td>
<td>12,875</td>
</tr>
</tbody>
</table>

(8 rows affected)

You can nest groups within groups. The maximum number of group by columns (or expressions) is not explicitly limited.

### Referencing 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.

Sybase 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:

```sql
select type, title_id, avg(price), avg(advance)
from titles
group by type
type title_id
```

<table>
<thead>
<tr>
<th>title_id</th>
<th>type</th>
<th>avg(price)</th>
<th>avg(advance)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BU1032</td>
<td>business</td>
<td>13.73</td>
<td>6,281.25</td>
</tr>
<tr>
<td>BU1111</td>
<td>business</td>
<td>13.73</td>
<td>6,281.25</td>
</tr>
<tr>
<td>BU2075</td>
<td>business</td>
<td>13.73</td>
<td>6,281.25</td>
</tr>
<tr>
<td>BU7832</td>
<td>business</td>
<td>13.73</td>
<td>6,281.25</td>
</tr>
<tr>
<td>MC2222</td>
<td>mod_cook</td>
<td>11.49</td>
<td>7,500.00</td>
</tr>
<tr>
<td>MC3021</td>
<td>mod_cook</td>
<td>11.49</td>
<td>7,500.00</td>
</tr>
<tr>
<td>MC3026</td>
<td>UNDECIDED</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>PC1035</td>
<td>popular_comp</td>
<td>21.48</td>
<td>7,500.00</td>
</tr>
<tr>
<td>PC8888</td>
<td>popular_comp</td>
<td>21.48</td>
<td>7,500.00</td>
</tr>
<tr>
<td>PC9999</td>
<td>popular_comp</td>
<td>21.48</td>
<td>7,500.00</td>
</tr>
<tr>
<td>PS1372</td>
<td>psychology</td>
<td>13.50</td>
<td>4,255.00</td>
</tr>
<tr>
<td>PS2091</td>
<td>psychology</td>
<td>13.50</td>
<td>4,255.00</td>
</tr>
<tr>
<td>PS2106</td>
<td>psychology</td>
<td>13.50</td>
<td>4,255.00</td>
</tr>
<tr>
<td>PS3333</td>
<td>psychology</td>
<td>13.50</td>
<td>4,255.00</td>
</tr>
<tr>
<td>PS7777</td>
<td>psychology</td>
<td>13.50</td>
<td>4,255.00</td>
</tr>
<tr>
<td>TC3218</td>
<td>trad_cook</td>
<td>15.96</td>
<td>6,333.33</td>
</tr>
<tr>
<td>TC4203</td>
<td>trad_cook</td>
<td>15.96</td>
<td>6,333.33</td>
</tr>
<tr>
<td>TC7777</td>
<td>trad_cook</td>
<td>15.96</td>
<td>6,333.33</td>
</tr>
</tbody>
</table>

(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:
Organizing query results into groups: the group by clause

```sql
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
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:

```sql
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
```
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`, Sybase suggests that you use the `fipsflagger` option to identify queries that contain Transact-SQL extensions. See “group by and SQL standards” on page 83 and the Reference Manual: Commands.

### Expressions and `group by`

Another Transact-SQL extension allows you to group by an expression that does not include aggregate functions. For example:

```sql
SELECT avg(total_sales), total_sales * price
FROM titles
GROUP BY total_sales * price
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2045</td>
<td>22,392.75</td>
</tr>
<tr>
<td>2032</td>
<td>40,619.68</td>
</tr>
<tr>
<td>4072</td>
<td>81,399.28</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>4095</td>
<td>61,384.05</td>
</tr>
<tr>
<td>18722</td>
<td>55,978.78</td>
</tr>
<tr>
<td>375</td>
<td>7,856.25</td>
</tr>
</tbody>
</table>
Organizing query results into groups: the group by clause

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:

```sql
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.”

```sql
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)
```
Using group by in nested aggregates

Another Transact-SQL extension allows you 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:

```sql
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:

```sql
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.

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, the null values form a single group. This example uses group by and the advance column, which contains some null values:

```sql
select advance, avg(price * 2)
from titles
group by advance
```

```
advance
--------------
```

Null values and group by
Organizing query results into groups: the group by clause

<table>
<thead>
<tr>
<th>price</th>
<th>count(price)</th>
<th>count(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>2.99</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>7.00</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7.99</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10.95</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11.95</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>14.99</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>19.99</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>20.00</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>20.95</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>21.59</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>22.95</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

(12 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:

```sql
select price, count(price), count(*)
from titles
group by price
```

<table>
<thead>
<tr>
<th>price</th>
<th>count(price)</th>
<th>count(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>2.99</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>7.00</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7.99</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10.95</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11.95</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>14.99</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>19.99</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>20.00</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>20.95</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>21.59</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>22.95</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

(12 rows affected)
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:

```sql
select type, avg(price)
from titles
where advance > 5000
group by type
```

<table>
<thead>
<tr>
<th>type</th>
<th>--------</th>
</tr>
</thead>
<tbody>
<tr>
<td>business</td>
<td>2.99</td>
</tr>
<tr>
<td>mod_cook</td>
<td>2.99</td>
</tr>
<tr>
<td>popular_comp</td>
<td>21.48</td>
</tr>
<tr>
<td>psychology</td>
<td>14.30</td>
</tr>
<tr>
<td>trad_cook</td>
<td>17.97</td>
</tr>
</tbody>
</table>

(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 Adaptive Server handles extra columns in the select list and the where clause may seem contradictory. For example:

```sql
select type, advance, avg(price)
from titles
where advance > 5000
group by type
```

<table>
<thead>
<tr>
<th>type</th>
<th>advance</th>
<th>--------</th>
</tr>
</thead>
<tbody>
<tr>
<td>business</td>
<td>5,000.00</td>
<td>2.99</td>
</tr>
<tr>
<td>business</td>
<td>5,000.00</td>
<td>2.99</td>
</tr>
<tr>
<td>business</td>
<td>10,125.00</td>
<td>2.99</td>
</tr>
<tr>
<td>business</td>
<td>5,000.00</td>
<td>2.99</td>
</tr>
<tr>
<td>mod_cook</td>
<td>0.00</td>
<td>2.99</td>
</tr>
<tr>
<td>mod_cook</td>
<td>15,000.00</td>
<td>2.99</td>
</tr>
<tr>
<td>popular_comp</td>
<td>7,000.00</td>
<td>21.48</td>
</tr>
<tr>
<td>popular_comp</td>
<td>8,000.00</td>
<td>21.48</td>
</tr>
<tr>
<td>popular_comp</td>
<td>NULL</td>
<td>21.48</td>
</tr>
<tr>
<td>psychology</td>
<td>7,000.00</td>
<td>14.30</td>
</tr>
<tr>
<td>psychology</td>
<td>2,275.00</td>
<td>14.30</td>
</tr>
<tr>
<td>psychology</td>
<td>6,000.00</td>
<td>14.30</td>
</tr>
<tr>
<td>psychology</td>
<td>2,000.00</td>
<td>14.30</td>
</tr>
<tr>
<td>psychology</td>
<td>4,000.00</td>
<td>14.30</td>
</tr>
<tr>
<td>trad_cook</td>
<td>7,000.00</td>
<td>17.97</td>
</tr>
</tbody>
</table>
Organizing query results into groups: the group by clause

When you look at the results for the advance (extended) column, it may seem as though the query is ignoring the where clause. Adaptive Server 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. See “Selecting groups of data: the having clause” on page 94.

**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:

```sql
select type, avg(advance)
from titles
where advance > 1000 and advance < 10000
group by type

<table>
<thead>
<tr>
<th>type</th>
<th>advance</th>
</tr>
</thead>
<tbody>
<tr>
<td>business</td>
<td>5,000.00</td>
</tr>
<tr>
<td>popular_comp</td>
<td>7,500.00</td>
</tr>
<tr>
<td>psychology</td>
<td>4,255.00</td>
</tr>
<tr>
<td>trad_cook</td>
<td>6,333.33</td>
</tr>
</tbody>
</table>

(4 rows affected)

select type, avg(advance)
from titles
where advance > 1000 and advance < 10000
group by all type

<table>
<thead>
<tr>
<th>type</th>
<th>advance</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNDECIDED</td>
<td>NULL</td>
</tr>
</tbody>
</table>
```

(17 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. Adaptive Server returns a NULL result for all groups that lack qualifying rows.

**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. The Transact-SQL extension that 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:

<table>
<thead>
<tr>
<th>pub_id</th>
<th>pub_name</th>
<th>city</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>0736</td>
<td>New Age Books</td>
<td>Boston</td>
<td>MA</td>
</tr>
<tr>
<td>0877</td>
<td>Binnet &amp; Hardley</td>
<td>Washington</td>
<td>DC</td>
</tr>
<tr>
<td>1389</td>
<td>Algodata Infosystems</td>
<td>Berkeley</td>
<td>CA</td>
</tr>
</tbody>
</table>

It contains three rows. The following query produces a three-row scalar aggregate based on each row of the table:

```sql
select pub_id, count(pub_id)
from publishers
```

<table>
<thead>
<tr>
<th>pub_id</th>
<th>count (pub_id)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0736</td>
<td>3</td>
</tr>
<tr>
<td>0877</td>
<td>3</td>
</tr>
<tr>
<td>1389</td>
<td>3</td>
</tr>
</tbody>
</table>

(3 rows affected)
Because Adaptive Server 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:

```sql
select pub_id, count(pub_id)
from publishers
where pub_id < "1000"
```

<table>
<thead>
<tr>
<th>pub_id</th>
<th>count(pub_id)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0736</td>
<td>2</td>
</tr>
<tr>
<td>0877</td>
<td>2</td>
</tr>
<tr>
<td>1389</td>
<td>2</td>
</tr>
</tbody>
</table>

(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.

### Selecting 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:

```sql
select title_id
from titles
where title_id like "PS%"
having avg(price) > $20
```

But this example is not:

```sql
select title_id
from titles
where avg(price) > $20
```

---

94 Adaptive Server Enterprise
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:

```sql
select type
from titles
group by type
having count(*) > 1
```

<table>
<thead>
<tr>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>business</td>
</tr>
<tr>
<td>mod_cook</td>
</tr>
<tr>
<td>popular_comp</td>
</tr>
<tr>
<td>psychology</td>
</tr>
<tr>
<td>trad_cook</td>
</tr>
</tbody>
</table>

(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”:

```sql
select type
from titles
group by type
having type like "p%"
```

<table>
<thead>
<tr>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>popular_comp</td>
</tr>
<tr>
<td>psychology</td>
</tr>
</tbody>
</table>

(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)

How the having, group by, and where clauses interact

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
CHAPTER 3  Using Aggregates, Grouping, and Sorting

(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.

```sql
select type, avg(price)
from titles
group by type
having sum(total_sales) > 10000
```

<table>
<thead>
<tr>
<th>type</th>
<th>avg(price)</th>
</tr>
</thead>
<tbody>
<tr>
<td>business</td>
<td>13.73</td>
</tr>
<tr>
<td>mod_cook</td>
<td>11.49</td>
</tr>
<tr>
<td>popular_comp</td>
<td>21.48</td>
</tr>
<tr>
<td>trad_cook</td>
<td>15.96</td>
</tr>
</tbody>
</table>

(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 described earlier in this chapter. For example:

```sql
select type, avg(price)
from titles
group by type
```
Selecting groups of data: the having clause

having total_sales > 4000

<table>
<thead>
<tr>
<th>type</th>
<th>advance</th>
</tr>
</thead>
<tbody>
<tr>
<td>business</td>
<td>13.73</td>
</tr>
<tr>
<td>business</td>
<td>13.73</td>
</tr>
<tr>
<td>business</td>
<td>13.73</td>
</tr>
<tr>
<td>mod_cook</td>
<td>11.49</td>
</tr>
<tr>
<td>popular_comp</td>
<td>21.48</td>
</tr>
<tr>
<td>popular_comp</td>
<td>21.48</td>
</tr>
<tr>
<td>psychology</td>
<td>13.50</td>
</tr>
<tr>
<td>trad_cook</td>
<td>15.96</td>
</tr>
<tr>
<td>trad_cook</td>
<td>15.96</td>
</tr>
</tbody>
</table>

(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 Adaptive Server 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
```

<table>
<thead>
<tr>
<th>type</th>
<th>advance</th>
</tr>
</thead>
<tbody>
<tr>
<td>business</td>
<td>10,125.00</td>
</tr>
<tr>
<td>mod_cook</td>
<td>15,000.00</td>
</tr>
<tr>
<td>popular_comp</td>
<td>7,000.00</td>
</tr>
<tr>
<td>popular_comp</td>
<td>8,000.00</td>
</tr>
<tr>
<td>psychology</td>
<td>7,000.00</td>
</tr>
<tr>
<td>psychology</td>
<td>6,000.00</td>
</tr>
<tr>
<td>trad_cook</td>
<td>7,000.00</td>
</tr>
<tr>
<td>trad_cook</td>
<td>8,000.00</td>
</tr>
</tbody>
</table>

(8 rows affected)
CHAPTER 3  Using Aggregates, Grouping, and Sorting

Using 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:

```sql
select title_id, advance, price
from titles
where advance < 4000
having price > avg(price)
```

<table>
<thead>
<tr>
<th>title_id</th>
<th>advance</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>BU1032</td>
<td>5,000.00</td>
<td>19.99</td>
</tr>
<tr>
<td>BU7832</td>
<td>5,000.00</td>
<td>19.99</td>
</tr>
<tr>
<td>MC2222</td>
<td>0.00</td>
<td>19.99</td>
</tr>
<tr>
<td>PC1035</td>
<td>7,000.00</td>
<td>22.95</td>
</tr>
<tr>
<td>PC8888</td>
<td>8,000.00</td>
<td>20.00</td>
</tr>
<tr>
<td>PS1372</td>
<td>7,000.00</td>
<td>21.59</td>
</tr>
<tr>
<td>PS3333</td>
<td>2,000.00</td>
<td>19.99</td>
</tr>
<tr>
<td>TC3218</td>
<td>7,000.00</td>
<td>20.95</td>
</tr>
</tbody>
</table>

(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.

```sql
select pub_id, count(pub_id)
from publishers
having pub_id < "1000"
```

<table>
<thead>
<tr>
<th>pub_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>0736</td>
</tr>
<tr>
<td>0877</td>
</tr>
</tbody>
</table>
Sorting query results: the order by clause

The order by clause allows you to sort query results by up to as many as 31 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:

```sql
select pub_id, type, title_id
from titles
order by pub_id
```

<table>
<thead>
<tr>
<th>pub_id</th>
<th>type</th>
<th>title_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>0736</td>
<td>business</td>
<td>BU2075</td>
</tr>
<tr>
<td>0736</td>
<td>psychology</td>
<td>PS2091</td>
</tr>
<tr>
<td>0736</td>
<td>psychology</td>
<td>PS2106</td>
</tr>
<tr>
<td>0736</td>
<td>psychology</td>
<td>PS3333</td>
</tr>
<tr>
<td>0736</td>
<td>psychology</td>
<td>PS7777</td>
</tr>
<tr>
<td>0877</td>
<td>UNDECIDED</td>
<td>MC3026</td>
</tr>
<tr>
<td>0877</td>
<td>mod_cook</td>
<td>MC2222</td>
</tr>
<tr>
<td>0877</td>
<td>mod_cook</td>
<td>MC3021</td>
</tr>
<tr>
<td>0877</td>
<td>psychology</td>
<td>PS1372</td>
</tr>
<tr>
<td>0877</td>
<td>trad_cook</td>
<td>TC3218</td>
</tr>
<tr>
<td>0877</td>
<td>trad_cook</td>
<td>TC4203</td>
</tr>
<tr>
<td>0877</td>
<td>trad_cook</td>
<td>TC7777</td>
</tr>
<tr>
<td>1389</td>
<td>business</td>
<td>BU1032</td>
</tr>
<tr>
<td>1389</td>
<td>business</td>
<td>BU1111</td>
</tr>
<tr>
<td>1389</td>
<td>business</td>
<td>BU7832</td>
</tr>
<tr>
<td>1389</td>
<td>popular_comp</td>
<td>PC1035</td>
</tr>
<tr>
<td>1389</td>
<td>popular_comp</td>
<td>PC8888</td>
</tr>
<tr>
<td>1389</td>
<td>popular_comp</td>
<td>PC9999</td>
</tr>
</tbody>
</table>

(18 rows affected)

Multiple columns  If you name more than one column in the order by clause, Adaptive Server 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). Adaptive Server sorts null values first within any group.

```sql
select stor_id, payterms, country
from stores
```

(2 rows affected)
CHAPTER 3 Using Aggregates, Grouping, and Sorting

order by stor_id desc, payterms

<table>
<thead>
<tr>
<th>stor_id</th>
<th>payterms</th>
<th>country</th>
</tr>
</thead>
<tbody>
<tr>
<td>8042</td>
<td>Net 30</td>
<td>USA</td>
</tr>
<tr>
<td>7896</td>
<td>Net 60</td>
<td>USA</td>
</tr>
<tr>
<td>7131</td>
<td>Net 60</td>
<td>USA</td>
</tr>
<tr>
<td>7067</td>
<td>Net 30</td>
<td>USA</td>
</tr>
<tr>
<td>7066</td>
<td>Net 30</td>
<td>USA</td>
</tr>
<tr>
<td>6380</td>
<td>Net 60</td>
<td>USA</td>
</tr>
<tr>
<td>5023</td>
<td>Net 60</td>
<td>USA</td>
</tr>
</tbody>
</table>

(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.

```sql
select pub_id, type, title_id
from titles
order by 1 desc, 2, 3
```

```sql
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:

```sql
select id, min(id) from tab
union
select id, max(id) from tab
ORDER BY 2
```
Sorting query results: the order by clause

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 Adaptive Server. The basic choices are binary, dictionary order, and case-insensitive. `sp_helpsort` displays the sort order for your server. See Chapter 9, “Configuring Character Sets, Sort Orders, and Languages” in the *System Administration Guide: Volume 1*.

**Limitations**  Adaptive Server does not allow subqueries or variables in the order by list.

You cannot use order by on text, unitext, or image columns.

**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
----------
```
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:

```sql
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, Adaptive Server 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:

```sql
select distinct pub_id, type
from titles
order by type
```

```
pub_id type
----- ------------
0877 UNDECIDED
0736 business
1389 business
```
Summarizing 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 (see “Row aggregates and compute” on page 107) for the same group.

The general syntax for compute is:

```sql
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.

```sql
select type, sum(price), sum(advance)
from titles
group by type
```

<table>
<thead>
<tr>
<th>type</th>
<th>sum(price)</th>
<th>sum(advance)</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNDECIDED</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>business</td>
<td>54.92</td>
<td>25,125.00</td>
</tr>
<tr>
<td>mod_cook</td>
<td>22.98</td>
<td>15,000.00</td>
</tr>
<tr>
<td>popular_comp</td>
<td>42.95</td>
<td>15,000.00</td>
</tr>
<tr>
<td>psychology</td>
<td>67.52</td>
<td>21,275.00</td>
</tr>
<tr>
<td>trad_cook</td>
<td>47.89</td>
<td>19,000.00</td>
</tr>
</tbody>
</table>

(6 rows affected)

```sql
select type, price, advance
from titles
order by type
compute sum(price), sum(advance) by type
```

```sql
Compute Result:
<table>
<thead>
<tr>
<th>type</th>
<th>price</th>
<th>advance</th>
<th>price</th>
<th>advance</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNDECIDED</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>business</td>
<td>2.99</td>
<td>10,125.00</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>business</td>
<td>11.95</td>
<td>5,000.00</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>business</td>
<td>19.99</td>
<td>5,000.00</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>business</td>
<td>19.99</td>
<td>5,000.00</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>
```

(6 rows affected)
Summarizing groups of data: the compute clause

<table>
<thead>
<tr>
<th>type</th>
<th>price</th>
<th>advance</th>
</tr>
</thead>
<tbody>
<tr>
<td>mod_cook</td>
<td>2.99</td>
<td>15,000.00</td>
</tr>
<tr>
<td>mod_cook</td>
<td>19.99</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Compute Result:

<table>
<thead>
<tr>
<th>type</th>
<th>price</th>
<th>advance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>price</th>
<th>advance</th>
</tr>
</thead>
<tbody>
<tr>
<td>popular_comp</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>popular_comp</td>
<td>20.00</td>
<td>8,000.00</td>
</tr>
<tr>
<td>popular_comp</td>
<td>22.95</td>
<td>7,000.00</td>
</tr>
</tbody>
</table>

Compute Result:

<table>
<thead>
<tr>
<th>type</th>
<th>price</th>
<th>advance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>price</th>
<th>advance</th>
</tr>
</thead>
<tbody>
<tr>
<td>psychology</td>
<td>7.00</td>
<td>6,000.00</td>
</tr>
<tr>
<td>psychology</td>
<td>7.99</td>
<td>4,000.00</td>
</tr>
<tr>
<td>psychology</td>
<td>10.95</td>
<td>2,275.00</td>
</tr>
<tr>
<td>psychology</td>
<td>19.99</td>
<td>2,000.00</td>
</tr>
<tr>
<td>psychology</td>
<td>21.59</td>
<td>7,000.00</td>
</tr>
</tbody>
</table>

Compute Result:

<table>
<thead>
<tr>
<th>type</th>
<th>price</th>
<th>advance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>price</th>
<th>advance</th>
</tr>
</thead>
<tbody>
<tr>
<td>trad_cook</td>
<td>11.95</td>
<td>4,000.00</td>
</tr>
<tr>
<td>trad_cook</td>
<td>14.99</td>
<td>8,000.00</td>
</tr>
<tr>
<td>trad_cook</td>
<td>20.95</td>
<td>7,000.00</td>
</tr>
</tbody>
</table>

Compute Result:

<table>
<thead>
<tr>
<th>type</th>
<th>price</th>
<th>advance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(24 rows affected)

Each summary value is treated as a row.
Row aggregates and `compute`

The row aggregates used with `compute` are listed in Table 3-2:

<table>
<thead>
<tr>
<th>Row aggregates</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sum</code></td>
<td>Total of the values in the expression</td>
</tr>
<tr>
<td><code>avg</code></td>
<td>Average of the values in the expression</td>
</tr>
<tr>
<td><code>max</code></td>
<td>Highest value in the expression</td>
</tr>
<tr>
<td><code>min</code></td>
<td>Lowest value in the expression</td>
</tr>
<tr>
<td><code>count</code></td>
<td>Number of selected rows as an integer</td>
</tr>
<tr>
<td><code>count_big</code></td>
<td>Number of selected rows as a bigint</td>
</tr>
</tbody>
</table>

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.

Rules for `compute` clauses

- Adaptive Server 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` (see Chapter 8, “Creating Databases and Tables”) 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:

```sql
order by a, b, c
```

The `compute` clause can be any or all of these:

```sql
compute row_aggregate (column_name) by a, b, c
compute row_aggregate (column_name) by a, b
```
Summarizing groups of data: the compute clause

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. The compute keyword without by is discussed under “Generating totals: compute without by” on page 111.

Specifying more than one column after compute

Listing more than one column after the by keyword affects the query by breaking a group into subgroups and applying 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:

```sql
select type, pub_id, price
from titles
where type = "psychology"
order by type, pub_id, price
compute sum(price) by type, pub_id
```

<table>
<thead>
<tr>
<th>type</th>
<th>pub_id</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>psychology</td>
<td>0736</td>
<td>7.00</td>
</tr>
<tr>
<td>psychology</td>
<td>0736</td>
<td>7.99</td>
</tr>
<tr>
<td>psychology</td>
<td>0736</td>
<td>10.95</td>
</tr>
<tr>
<td>psychology</td>
<td>0736</td>
<td>19.99</td>
</tr>
</tbody>
</table>

Compute Result:
---------------
45.93

<table>
<thead>
<tr>
<th>type</th>
<th>pub_id</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>psychology</td>
<td>0877</td>
<td>21.59</td>
</tr>
</tbody>
</table>

Compute Result:
--------------
Using 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:

```sql
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
```

<table>
<thead>
<tr>
<th>type</th>
<th>pub_id</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>psychology</td>
<td>0736</td>
<td>7.00</td>
</tr>
<tr>
<td>psychology</td>
<td>0736</td>
<td>7.99</td>
</tr>
<tr>
<td>psychology</td>
<td>0736</td>
<td>10.95</td>
</tr>
<tr>
<td>psychology</td>
<td>0736</td>
<td>19.99</td>
</tr>
</tbody>
</table>

Compute Result:
---------------

<table>
<thead>
<tr>
<th>type</th>
<th>pub_id</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>psychology</td>
<td>0877</td>
<td>21.59</td>
</tr>
</tbody>
</table>

Compute Result:
---------------

<table>
<thead>
<tr>
<th>type</th>
<th>pub_id</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>psychology</td>
<td></td>
<td>21.59</td>
</tr>
</tbody>
</table>

(8 rows affected)
Applying 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
```

<table>
<thead>
<tr>
<th>type</th>
<th>price</th>
<th>advance</th>
</tr>
</thead>
<tbody>
<tr>
<td>mod_cook</td>
<td>2.99</td>
<td>15,000.00</td>
</tr>
<tr>
<td>mod_cook</td>
<td>19.99</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Compute Result:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>22.98</td>
<td>15,000.00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>price</th>
<th>advance</th>
</tr>
</thead>
<tbody>
<tr>
<td>trad_cook</td>
<td>11.95</td>
<td>4,000.00</td>
</tr>
<tr>
<td>trad_cook</td>
<td>14.99</td>
<td>8,000.00</td>
</tr>
<tr>
<td>trad_cook</td>
<td>20.95</td>
<td>7,000.00</td>
</tr>
</tbody>
</table>

Compute Result:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>47.89</td>
<td>19,000.00</td>
</tr>
</tbody>
</table>

(7 rows affected)

Remember, the columns to which the aggregates apply must also be in the select list.

Using different aggregates in the same 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
```

<table>
<thead>
<tr>
<th>type</th>
<th>pub_id</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>trad_cook</td>
<td>19</td>
<td></td>
</tr>
</tbody>
</table>

(7 rows affected)
CHAPTER 3  Using Aggregates, Grouping, and Sorting

mod_cook  0877  2.99
mod_cook  0877  19.99

Compute Result:
--------------- ----
             22.98 0877

<table>
<thead>
<tr>
<th>type</th>
<th>pub_id</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>trad_cook</td>
<td>0877</td>
<td>11.95</td>
</tr>
<tr>
<td>trad_cook</td>
<td>0877</td>
<td>14.99</td>
</tr>
<tr>
<td>trad_cook</td>
<td>0877</td>
<td>20.95</td>
</tr>
</tbody>
</table>

Compute Result:
--------------- ----
             47.89 0877

(7 rows affected)

Generating 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:

```sql
select type, price, advance
from titles
where price > 20
compute sum(price), sum(advance)
```

<table>
<thead>
<tr>
<th>type</th>
<th>price</th>
<th>advance</th>
</tr>
</thead>
<tbody>
<tr>
<td>popular_comp</td>
<td>22.95</td>
<td>7,000.00</td>
</tr>
<tr>
<td>psychology</td>
<td>21.59</td>
<td>7,000.00</td>
</tr>
<tr>
<td>trad_cook</td>
<td>20.95</td>
<td>7,000.00</td>
</tr>
</tbody>
</table>

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)
```

<table>
<thead>
<tr>
<th>type</th>
<th>price</th>
<th>advance</th>
</tr>
</thead>
<tbody>
<tr>
<td>mod_cook</td>
<td>2.99</td>
<td>15,000.00</td>
</tr>
<tr>
<td>mod_cook</td>
<td>19.99</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Compute Result:

<table>
<thead>
<tr>
<th></th>
<th>22.98</th>
<th>15,000.00</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>price</th>
<th>advance</th>
</tr>
</thead>
<tbody>
<tr>
<td>trad_cook</td>
<td>11.95</td>
<td>4,000.00</td>
</tr>
<tr>
<td>trad_cook</td>
<td>14.99</td>
<td>8,000.00</td>
</tr>
<tr>
<td>trad_cook</td>
<td>20.95</td>
<td>7,000.00</td>
</tr>
</tbody>
</table>

Compute Result:

<table>
<thead>
<tr>
<th></th>
<th>47.89</th>
<th>19,000.00</th>
</tr>
</thead>
</table>

Compute Result:

<table>
<thead>
<tr>
<th></th>
<th>70.87</th>
<th>34,000.00</th>
</tr>
</thead>
</table>

(8 rows affected)

**Combining 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.

Figure 3-1 shows two 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.

**Figure 3-1: Union combining queries**

<table>
<thead>
<tr>
<th>Table T1</th>
<th>Table T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>char(4)</td>
<td>char(4)</td>
</tr>
<tr>
<td>abc</td>
<td>ghi</td>
</tr>
<tr>
<td>def</td>
<td>jkl</td>
</tr>
<tr>
<td>ghi</td>
<td>mno</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

The following query creates a union between the two tables:

```sql
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
```
Combining queries: the union operator

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, Adaptive Server 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.

Guidelines for union queries

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 Chapter 1, “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:

**Figure 3-2: Union comparing columns**

<table>
<thead>
<tr>
<th>Table T3</th>
<th>Table T4</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>int</td>
<td>char(4)</td>
</tr>
<tr>
<td>abc</td>
<td>abc</td>
</tr>
<tr>
<td>def</td>
<td>def</td>
</tr>
<tr>
<td>ghi</td>
<td>ghi</td>
</tr>
</tbody>
</table>

Table 3-2 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
```
Combining queries: the union operator

select b, a from T4

The query produces:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>abc</td>
</tr>
<tr>
<td>2</td>
<td>def</td>
</tr>
<tr>
<td>3</td>
<td>ghi</td>
</tr>
</tbody>
</table>

(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, Adaptive Server converts them to the datatype with the most precision.

- Adaptive Server 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

Using union with other Transact-SQL commands

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:

```sql
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
```

- Starting with Adaptive Server version 12.5, you can use the union operator within a create view statement. If you are using an earlier version of Adaptive Server, however, you cannot 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.
Combining queries: the union operator
Joins: Retrieving 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. See Chapter 5, “Subqueries: Using Queries Within Other Queries.”

When Component Integration Services is enabled, you can perform joins across remote servers. See the *Component Integration Services Users Guide.*
How joins work

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:

\[
\text{select au_fname, au_lname, pub_name} \\
\text{from authors, publishers} \\
\text{where authors.city = publishers.city} \\
\]

<table>
<thead>
<tr>
<th>au_fname</th>
<th>au_lname</th>
<th>pub_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cheryl</td>
<td>Carson</td>
<td>Algodata Infosystems</td>
</tr>
<tr>
<td>Abraham</td>
<td>Bennet</td>
<td>Algodata Infosystems</td>
</tr>
</tbody>
</table>

(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:

\[
\text{where authors.city = publishers.city} \\
\]

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:

\[
\text{start of select, update, insert, delete, or subquery} \\
\text{from \{table_list | view_list\}} \\
\text{where [not]} \\
\text{[table_name, | view_name.]column_name join_operator} \\
\text{[table_name, | view_name.]column_name} \\
\text{[and | or] [not]} \\
\text{[table_name, | view_name.]column_name join_operator} \\
\text{[table_name, | view_name.]column_name]}... \\
\text{End of select, update, insert, delete, or subquery} \\
\]
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.

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:

```sql
select au_fname, au_lname, pub_name
from authors, publishers
```
How joins are structured

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:

```sql
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, Adaptive Server 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:

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

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`:

```sql
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.

### The from clause

Use the from clause to specify which tables and views to join. This is the clause that indicates to Adaptive Server 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 50 tables and 46 worktables (such as those created by aggregate functions). The 50-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:

```sql
begin tran
update titles
set price = price * 2
from titles, publishers
where titles.pub_id = publishers.pub_id
and publishers.state = "CA"
rollback tran
```

See “Joining more than two tables” on page 134 for information on joins involving more than two tables or views.

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:

```sql
select au_lname, au_fname
from pubs2.blue.authors, pubs2.blue.publishers
where authors.city = publishers.city
```

You can join views in exactly the same way as tables and use views wherever tables are used. Chapter 12, “Views: Limiting Access to Data” discusses views; this chapter uses only tables in its examples.
How joins are structured

The *where* clause

Use the *where* clause to determine which rows are included in the results. *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:

```sql
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.” For details of *where* clause syntax, see Chapter 2, “Queries: Selecting Data from a Table.”

**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. “How joins are processed” on page 126 explains this result.

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. See “How joins are processed” on page 126 for an example.

Join operators

Joins that match columns on the basis of equality are called **equijoins**. A more precise definition of an **equijoin** is given under “Equijoins and natural joins” on page 127, along with examples of joins not based on equality.

Equijoins use the following comparison operators:
Joins that use the relational operators are collectively called **theta joins**. Another set of join operators is used for **outer joins**, also discussed in detail later in this chapter. The outer join operators are Transact-SQL extensions, as shown in Table 4-2:

**Table 4-2: Outer join operators**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>*=</td>
<td>Include in the results all the rows from the first table, not just the ones where the joined columns match.</td>
</tr>
<tr>
<td>=&gt;*</td>
<td>Include in the results all the rows from the second table, not just the ones where the joined columns match.</td>
</tr>
</tbody>
</table>

### Datatypes in join columns

The columns being joined must have the same or compatible datatypes. Use the `convert` function when comparing columns whose datatypes 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, Adaptive Server automatically converts them. For example, Adaptive Server converts among any of the numeric type columns—`bigint`, `int`, `smallint`, `tinyint`, `unsigned bigint`, `unsigned int`, `unsigned smallint`, `decimal`, or `float`, and among any of the character type and date columns—`char`, `varchar`, `unichar`, `univarchar`, `nchar`, `nvarchar`, `datetime`, `date`, and `time`. For details on datatype conversion, see Chapter 16, “Using Transact-SQL Functions in Queries,” and Chapter 1, “System and User-Defined Datatypes,” in the *Reference Manual: Building Blocks*. 

---

**Table 4-1: Join operators**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Equal to</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal to</td>
</tr>
<tr>
<td>&lt;*</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>&lt;=*</td>
<td>Greater than or equal to</td>
</tr>
</tbody>
</table>
How joins are processed

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)
```

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. This section describes the processing of joins in conceptual terms. The actual procedure Adaptive Server uses is more sophisticated.

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, Adaptive Server 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 Adaptive Server 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:
Equijoins and natural joins

Joins 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:

```sql
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:

```sql
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:

```sql
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.

```sql
select type, au_lname, au_fname, title
from authors, titles, titleauthor
where authors.au_id = titleauthor.au_id
```
Joins with additional conditions

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

select title, pub_name, advance
from titles, publishers
where titles.pub_id = publishers.pub_id
and advance > $7500

<table>
<thead>
<tr>
<th>title</th>
<th>pub_name</th>
<th>advance</th>
</tr>
</thead>
<tbody>
<tr>
<td>You Can Combat Computer Stress!</td>
<td>New Age Books</td>
<td>10,125.00</td>
</tr>
<tr>
<td>The Gourmet Microwave</td>
<td>Binnet &amp; Hardley</td>
<td>15,000.00</td>
</tr>
<tr>
<td>Secrets of Silicon Valley</td>
<td>Algodata Infosystems</td>
<td>8,000.00</td>
</tr>
<tr>
<td>Sushi, Anyone?</td>
<td>Binnet &amp; Hardley</td>
<td>8,000.00</td>
</tr>
</tbody>
</table>

(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.
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.

```sql
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'
```

<table>
<thead>
<tr>
<th>pub_name</th>
<th>state</th>
<th>au_lname</th>
<th>au_fname</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>New Age Books</td>
<td>MA</td>
<td>Greene</td>
<td>Morningstar</td>
<td>TN</td>
</tr>
<tr>
<td>New Age Books</td>
<td>MA</td>
<td>Blotchet-Halls</td>
<td>Reginald</td>
<td>OR</td>
</tr>
<tr>
<td>New Age Books</td>
<td>MA</td>
<td>del Castillo</td>
<td>Innes</td>
<td>MI</td>
</tr>
<tr>
<td>New Age Books</td>
<td>MA</td>
<td>Panteley</td>
<td>Sylvia</td>
<td>MD</td>
</tr>
<tr>
<td>New Age Books</td>
<td>MA</td>
<td>Ringer</td>
<td>Anne</td>
<td>UT</td>
</tr>
<tr>
<td>New Age Books</td>
<td>MA</td>
<td>Ringer</td>
<td>Albert</td>
<td>UT</td>
</tr>
</tbody>
</table>

(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.

```sql
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
```

<table>
<thead>
<tr>
<th>title_id</th>
<th>total_sales</th>
<th>royalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>BU1032</td>
<td>4095</td>
<td>10</td>
</tr>
<tr>
<td>BU1111</td>
<td>3876</td>
<td>10</td>
</tr>
<tr>
<td>BU2075</td>
<td>1872</td>
<td>24</td>
</tr>
<tr>
<td>BU7832</td>
<td>4095</td>
<td>10</td>
</tr>
<tr>
<td>MC2222</td>
<td>2032</td>
<td>12</td>
</tr>
<tr>
<td>MC3021</td>
<td>2246</td>
<td>24</td>
</tr>
<tr>
<td>PC1035</td>
<td>8780</td>
<td>16</td>
</tr>
<tr>
<td>PC8888</td>
<td>4095</td>
<td>10</td>
</tr>
<tr>
<td>PS1372</td>
<td>375</td>
<td>10</td>
</tr>
</tbody>
</table>
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
```

<table>
<thead>
<tr>
<th>au_fname</th>
<th>au_lname</th>
<th>au_fname</th>
<th>au_lname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Marjorie</td>
<td>Green</td>
<td>Marjorie</td>
<td>Green</td>
</tr>
<tr>
<td>Dick</td>
<td>Straight</td>
<td>Dick</td>
<td>Straight</td>
</tr>
<tr>
<td>Dick</td>
<td>Straight</td>
<td>Dirk</td>
<td>Stringer</td>
</tr>
<tr>
<td>Dirk</td>
<td>Stringer</td>
<td>Dirk</td>
<td>Straight</td>
</tr>
<tr>
<td>Dirk</td>
<td>Stringer</td>
<td>Livia</td>
<td>Karsen</td>
</tr>
<tr>
<td>Stearns</td>
<td>MacFeather</td>
<td>Stearns</td>
<td>MacFeather</td>
</tr>
<tr>
<td>Livia</td>
<td>Karsen</td>
<td>Dick</td>
<td>Straight</td>
</tr>
<tr>
<td>Livia</td>
<td>Karsen</td>
<td>Dirk</td>
<td>Stringer</td>
</tr>
<tr>
<td>Livia</td>
<td>Karsen</td>
<td>Livia</td>
<td>Karsen</td>
</tr>
</tbody>
</table>
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:

```sql
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
```

<table>
<thead>
<tr>
<th>au_f_name</th>
<th>au_l_name</th>
<th>au_f_name</th>
<th>au_l_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dick</td>
<td>Straight</td>
<td>Dirk</td>
<td>Stringer</td>
</tr>
<tr>
<td>Dick</td>
<td>Straight</td>
<td>Livia</td>
<td>Karsen</td>
</tr>
<tr>
<td>Dirk</td>
<td>Stringer</td>
<td>Livia</td>
<td>Karsen</td>
</tr>
</tbody>
</table>

(3 rows affected)

It is now clear that Dick Straight, Dirk Stringer, and Livia Karsen all have the same postal code.

**The not-equal join**

The not-equal join is particularly useful in 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:

```sql
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
```

<table>
<thead>
<tr>
<th>type</th>
<th>price</th>
</tr>
</thead>
</table>

(3 rows affected)
The not-equal join

<table>
<thead>
<tr>
<th>Business</th>
<th>Psychology</th>
<th>Trad_Cook</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.99</td>
<td>7.00</td>
<td>11.95</td>
</tr>
<tr>
<td>11.95</td>
<td>7.99</td>
<td>14.99</td>
</tr>
<tr>
<td>10.95</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(7 rows affected)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

```sql
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
```

<table>
<thead>
<tr>
<th>AU_ID</th>
<th>Title_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>213-46-8915</td>
<td>BU1032</td>
</tr>
<tr>
<td>409-56-7008</td>
<td>BU1032</td>
</tr>
<tr>
<td>267-41-2394</td>
<td>BU1111</td>
</tr>
<tr>
<td>724-80-9391</td>
<td>BU1111</td>
</tr>
<tr>
<td>722-51-5454</td>
<td>MC3021</td>
</tr>
<tr>
<td>899-46-2035</td>
<td>MC3021</td>
</tr>
<tr>
<td>427-17-2319</td>
<td>PC8888</td>
</tr>
<tr>
<td>846-92-7186</td>
<td>PC8888</td>
</tr>
<tr>
<td>724-80-9391</td>
<td>PS1372</td>
</tr>
<tr>
<td>756-30-7391</td>
<td>PS1372</td>
</tr>
<tr>
<td>899-46-2035</td>
<td>PS2091</td>
</tr>
<tr>
<td>998-72-3567</td>
<td>PS2091</td>
</tr>
<tr>
<td>267-41-2394</td>
<td>TC7777</td>
</tr>
<tr>
<td>472-27-2349</td>
<td>TC7777</td>
</tr>
<tr>
<td>672-71-3249</td>
<td>TC7777</td>
</tr>
</tbody>
</table>

(15 rows affected)

For each book in titles, the following example finds all other books of the same type that have a different price:

```sql
select t1.type, t1.title_id, t1.price, t2.title_id, t2.price
```
CHAPTER 4    Joins: Retrieving Data from Several Tables

from titles t1, titles t2
where t1.type = t2.type
and t1.price != t2.price

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:
### Joining more than two tables

The titleauthor table of pubs2 offers a good example of a situation in which joining more than two tables is helpful. To find the titles of all the books of a particular type and the names of their authors, use:

```sql
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"
```

<table>
<thead>
<tr>
<th>au_lname</th>
<th>au_fname</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Panteley</td>
<td>Sylvia</td>
<td>Onions, Leeks, and Garlic: Cooking</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Secrets of the Mediterranean</td>
</tr>
<tr>
<td>Blotchet-Halls</td>
<td>Reginald</td>
<td>Fifty Years in Buckingham Palace</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Kitchens</td>
</tr>
<tr>
<td>O’Leary</td>
<td>Michael</td>
<td>Sushi, Anyone?</td>
</tr>
<tr>
<td>Gringlesby</td>
<td>Burt</td>
<td>Sushi, Anyone?</td>
</tr>
<tr>
<td>Yokomoto</td>
<td>Akiko</td>
<td>Sushi, Anyone?</td>
</tr>
</tbody>
</table>

(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:

```sql
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

<table>
<thead>
<tr>
<th>title_id</th>
<th>total_sales</th>
<th>lorange</th>
<th>hirange</th>
<th>royalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>BU1032</td>
<td>4095</td>
<td>0</td>
<td>5000</td>
<td>10</td>
</tr>
<tr>
<td>BU1111</td>
<td>3876</td>
<td>0</td>
<td>4000</td>
<td>10</td>
</tr>
<tr>
<td>BU2075</td>
<td>18722</td>
<td>14001</td>
<td>50000</td>
<td>24</td>
</tr>
<tr>
<td>BU7832</td>
<td>4095</td>
<td>0</td>
<td>5000</td>
<td>10</td>
</tr>
<tr>
<td>MC2222</td>
<td>2032</td>
<td>2001</td>
<td>4000</td>
<td>12</td>
</tr>
<tr>
<td>MC3021</td>
<td>2224</td>
<td>12001</td>
<td>50000</td>
<td>24</td>
</tr>
<tr>
<td>PC1035</td>
<td>8780</td>
<td>4001</td>
<td>10000</td>
<td>16</td>
</tr>
<tr>
<td>PC8888</td>
<td>4095</td>
<td>0</td>
<td>5000</td>
<td>10</td>
</tr>
<tr>
<td>PS1372</td>
<td>375</td>
<td>0</td>
<td>10000</td>
<td>10</td>
</tr>
<tr>
<td>PS2091</td>
<td>2045</td>
<td>1001</td>
<td>5000</td>
<td>12</td>
</tr>
<tr>
<td>PS2106</td>
<td>111</td>
<td>0</td>
<td>2000</td>
<td>10</td>
</tr>
<tr>
<td>PS3333</td>
<td>4072</td>
<td>0</td>
<td>5000</td>
<td>10</td>
</tr>
<tr>
<td>PS7777</td>
<td>3336</td>
<td>0</td>
<td>5000</td>
<td>10</td>
</tr>
<tr>
<td>TC3218</td>
<td>375</td>
<td>0</td>
<td>2000</td>
<td>10</td>
</tr>
<tr>
<td>TC4203</td>
<td>15096</td>
<td>8001</td>
<td>16000</td>
<td>14</td>
</tr>
<tr>
<td>TC7777</td>
<td>4095</td>
<td>0</td>
<td>5000</td>
<td>10</td>
</tr>
</tbody>
</table>

(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.

### Outer joins

Joins that include all rows, regardless of whether there is a matching row, are called **outer joins**. Adaptive Server supports both left and right outer joins. For example, the following query joins the titles and the titleauthor tables on their title_id column:

```sql
select *
from titles, titleauthor
where titles.title_id *= titleauthor.title_id
```
Sybase 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 Sybase as part of the Transact-SQL language. See “Transact-SQL outer joins” on page 154.

ANSI outer joins use the keywords `left join` and `right join` to indicate a left and right join, respectively. Sybase implemented the ANSI outer join syntax to fully comply with the ANSI standard. See “ANSI inner and outer joins” on page 137. This is the previous example rewritten as an ANSI outer join:

```sql
select *
from titles left join titleauthor
on titles.title_id = titleauthor.title_id
```

### Inner and outer tables

The terms **outer table** and **inner table** describe the placement of the 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:

```sql
T1 left join T2
T2 right join T1
```

Or, using Transact-SQL syntax:

```
T1 *= T2
T2 =* T1
```

### 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:

```sql
select distinct sales.stor_id, stor_name, title
from sales, stores, titles, salesdetail
```
CHAPTER 4  Joins: Retrieving Data from Several Tables

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.

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.

ANSI inner and outer joins

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
```

<table>
<thead>
<tr>
<th>title_id</th>
<th>title</th>
<th>ord_num</th>
<th>qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>BU1032</td>
<td>The Busy Executive</td>
<td>AX-532-FED-452-2Z7</td>
<td>200</td>
</tr>
<tr>
<td>BU1032</td>
<td>The Busy Executive</td>
<td>NP-123-ADS-642-9G3</td>
<td>1000</td>
</tr>
<tr>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
<td>.</td>
</tr>
<tr>
<td>TC7777</td>
<td>Sushi, Anyone?</td>
<td>ZD-123-DFG-752-9G8</td>
<td>1500</td>
</tr>
<tr>
<td>TC7777</td>
<td>Sushi, Anyone?</td>
<td>XS-135-DER-432-8J2</td>
<td>1090</td>
</tr>
</tbody>
</table>

(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. See “ANSI inner joins” on page 140. 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. See “ANSI outer joins” on page 143.

**Note** You can also use ANSI syntax to join views.

Sybase ANSI join syntax does not support the using clause.

**Correlation name and column referencing rules for ANSI joins**

The following are the correlation name and column reference rules specifically for ANSI joins. See “Self-joins and correlation names” on page 130.
• 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 \( \text{pub}_id \) column is specified:

\[
\text{select title, } t.\text{pub}_id, \text{pub_name} \\
\text{from titles } t \text{ left join publishers } p \\
\text{on } t.\text{pub}_id = p.\text{pub}_id
\]

However, the following example incorrectly uses the table name instead of the correlation name for the \( \text{titles} \) table (\( t.\text{pub}_id \)) in the \( \text{on} \) clause of the query, and produces the subsequent error message:

\[
\text{select title, } t.\text{pub}_id, \text{pub_name} \\
\text{from titles } t \text{ left join publishers } p \\
\text{on titles.\text{pub}_id = p.\text{pub}_id} \\
\text{Msg 107, Level 15, State 1:} \\
\text{Server 'server_name', Line 1:} \\
\text{The column prefix 't' does not match with a table name or alias name used in the query. Either the table is not specified in the FROM clause or it has a correlation name which must be used instead.}
\]

• The restriction specified in the \( \text{on} \) 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 \( \text{on} \) 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:

\[
\text{select *} \\
\text{from titles left join titleauthor} \\
\text{on titles.\text{title}_id=roysched.\text{title}_id} \quad /*\text{join #1}*/ \\
\text{left join roysched} \\
\text{on titleauthor.\text{title}_id=roysched.\text{title}_id} \quad /*\text{join #2}*/ \\
\text{where titles.\text{title}_id } \neq \text{ "PS7777"}
\]
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:

```sql
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:

```sql
select title, price, titleauthor.au_id, titleauthor.title_id, pub_name, publishers.city
from roysched, titles
left join titleauthor
left join authors
left join publishers
on roysched.title_id=titleauthor.title_id
on roysched.title_id=authors.au_id
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.

**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. See “ANSI outer joins” on page 143.

Adaptive Server 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:

```sql
select au_id, titles.title_id, title, price
from titleauthor, titles
where titleauthor.title_id = titles.title_id
and price > $15
```

See “How joins are structured” on page 121.

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

<table>
<thead>
<tr>
<th>au_id</th>
<th>title_id</th>
<th>title</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>213-46-8915</td>
<td>BU1032</td>
<td>The Busy Executive’s Datab</td>
<td>19.99</td>
</tr>
<tr>
<td>409-56-7008</td>
<td>BU1032</td>
<td>The Busy Executive’s Datab</td>
<td>19.99</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>172-32-1176</td>
<td>PS3333</td>
<td>Prolonged Data Deprivation</td>
<td>19.99</td>
</tr>
<tr>
<td>807-91-6654</td>
<td>TC3218</td>
<td>Onions, Leeks, and Garlic:</td>
<td>20.95</td>
</tr>
</tbody>
</table>

(11 rows affected)

The two methods of writing joins, ANSI or Transact-SQL, are equivalent. For
easy, 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"
```
Outer joins

The join table of an inner join

An ANSI join specifies which tables or views to join in the query. The table references 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:

```sql
select title, price, advance, royaltyper
from titles inner join titleauthor
on titles.title_id = titleauthor.title_id
```

<table>
<thead>
<tr>
<th>title</th>
<th>price</th>
<th>advance</th>
<th>royaltyper</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Busy...</td>
<td>19.99</td>
<td>5,000.00</td>
<td>40</td>
</tr>
<tr>
<td>The Busy...</td>
<td>19.99</td>
<td>5,000.00</td>
<td>60</td>
</tr>
<tr>
<td>Sushi, A...</td>
<td>14.99</td>
<td>8,000.00</td>
<td>30</td>
</tr>
<tr>
<td>Sushi, A...</td>
<td>14.99</td>
<td>8,000.00</td>
<td>40</td>
</tr>
</tbody>
</table>

(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 50 user tables (or 14 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`

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, your performance may be better if these 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:

```sql
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`. See “Self-joins and correlation names” on page 130.
The `on` clause often compares the ANSI joins tables, as in the third and fourth line of this query:

```sql
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 further 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:

```sql
select stor_name, stor_address, ord_num, qty
from salesdetail inner join stores
on salesdetail.stor_id = stores.stor_id
where qty > 3000
```

and:

```sql
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.

**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.
Sybase recommends that your applications use ANSI outer joins because they unambiguously specify whether the on or where clause contains the predicate.

This section discusses only the ANSI outer joins; for information about Transact-SQL outer joins, see “Transact-SQL outer joins” on page 154.

**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:

```sql
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:

```sql
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:

```sql
select au_fname, au_lname, pub_name
from publishers right join authors
on authors.city = publishers.city
```

**Should the predicate be 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:

```sql
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
```

<table>
<thead>
<tr>
<th>title</th>
<th>title_id</th>
<th>price</th>
<th>au_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>But Is It User F...</td>
<td>PC1035</td>
<td>22.95</td>
<td>238-95-7766</td>
</tr>
<tr>
<td>Computer Phobic ...</td>
<td>PS1372</td>
<td>21.59</td>
<td>724-80-9391</td>
</tr>
<tr>
<td>Computer Phobic ...</td>
<td>PS1372</td>
<td>21.59</td>
<td>756-30-7391</td>
</tr>
<tr>
<td>Onions, Leeks, a...</td>
<td>TC3218</td>
<td>20.95</td>
<td>807-91-6654</td>
</tr>
</tbody>
</table>

(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:

```sql
select title, titles.title_id, price, au_id
```
Outer joins

from titles left join titleauthor
on titles.title_id = titleauthor.title_id
and titles.price > $20.00

<table>
<thead>
<tr>
<th>title</th>
<th>title_id</th>
<th>price</th>
<th>au_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Busy Executive’s</td>
<td>BU1032</td>
<td>19.99</td>
<td>NULL</td>
</tr>
<tr>
<td>Cooking with Compute</td>
<td>BU1111</td>
<td>11.95</td>
<td>NULL</td>
</tr>
<tr>
<td>You Can Combat Compu</td>
<td>BU2075</td>
<td>2.99</td>
<td>NULL</td>
</tr>
<tr>
<td>Straight Talk About</td>
<td>BU7832</td>
<td>19.99</td>
<td>NULL</td>
</tr>
<tr>
<td>Silicon Valley Gastro</td>
<td>MC2222</td>
<td>19.99</td>
<td>NULL</td>
</tr>
<tr>
<td>The Gourmet Microwave</td>
<td>MC3021</td>
<td>2.99</td>
<td>NULL</td>
</tr>
<tr>
<td>The Psychology of Com</td>
<td>MC3026</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>But Is It User Friend</td>
<td>PC1035</td>
<td>22.95</td>
<td>238-95-7766</td>
</tr>
<tr>
<td>Secrets of Silicon Va</td>
<td>PC8888</td>
<td>20.00</td>
<td>NULL</td>
</tr>
<tr>
<td>Net Etiquette</td>
<td>PC9999</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>Computer Phobic and</td>
<td>PS1372</td>
<td>21.59</td>
<td>724-80-9391</td>
</tr>
<tr>
<td>Computer Phobic and</td>
<td>PS1372</td>
<td>21.59</td>
<td>756-30-7391</td>
</tr>
<tr>
<td>Is Anger the Enemy?</td>
<td>PS2091</td>
<td>10.95</td>
<td>NULL</td>
</tr>
<tr>
<td>Life Without Fear</td>
<td>PS2106</td>
<td>7.00</td>
<td>NULL</td>
</tr>
<tr>
<td>Prolonged Data Depri</td>
<td>PS3333</td>
<td>19.99</td>
<td>NULL</td>
</tr>
<tr>
<td>Emotional Security:</td>
<td>PS7777</td>
<td>7.99</td>
<td>NULL</td>
</tr>
<tr>
<td>Onions, Leeks, and Ga</td>
<td>TC3218</td>
<td>20.95</td>
<td>807-91-6654</td>
</tr>
<tr>
<td>Fifty Years in Buckin</td>
<td>TC4203</td>
<td>11.95</td>
<td>NULL</td>
</tr>
<tr>
<td>Sushi, Anyone?</td>
<td>TC7777</td>
<td>14.99</td>
<td>NULL</td>
</tr>
</tbody>
</table>

(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.

The following query includes a restriction on an inner table in the where clause:

```sql
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

<table>
<thead>
<tr>
<th>title</th>
<th>title_id</th>
<th>price</th>
<th>au_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>But Is It U...</td>
<td>PC1035</td>
<td>22.95</td>
<td>238-95-7766</td>
</tr>
<tr>
<td>Computer Ph...</td>
<td>PS1372</td>
<td>21.59</td>
<td>724-80-9391</td>
</tr>
<tr>
<td>Computer Ph...</td>
<td>PS1372</td>
<td>21.59</td>
<td>756-30-7391</td>
</tr>
<tr>
<td>Onions, Lee...</td>
<td>TC3218</td>
<td>20.95</td>
<td>807-91-6654</td>
</tr>
</tbody>
</table>

(4 rows affected)

Because the restriction of the \textit{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 \textit{where} clause is not true for all null-supplied values and removes them. A join that places its restriction in the \textit{where} clause is effectively an inner join.

However, if you move the restriction to the \textit{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:

```sql
select title, titles.title_id, price, au_id
from titleauthor left join titles
on titles.title_id = titleauthor.title_id
and price > $20.00
```

<table>
<thead>
<tr>
<th>title</th>
<th>title_id</th>
<th>price</th>
<th>au_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>172-32-1176</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>213-46-8915</td>
</tr>
<tr>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
</tr>
<tr>
<td>Onions,</td>
<td>TC3218</td>
<td>20.95</td>
<td>807-91-6654</td>
</tr>
<tr>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>998-72-3567</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>998-72-3567</td>
</tr>
</tbody>
</table>

(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 \textit{on} clause; this preserves the rows of the outer table. If you include a restriction for an inner table in the \textit{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.

The restriction in the where clause of the following query includes both the inner and outer tables:

```sql
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
```

<table>
<thead>
<tr>
<th>title</th>
<th>title_id</th>
<th>price</th>
<th>qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Silicon Valley Ga</td>
<td>MC2222</td>
<td>19.99</td>
<td>40,619.68</td>
</tr>
<tr>
<td>But Is It User Fr</td>
<td>PC1035</td>
<td>22.95</td>
<td>45,900.00</td>
</tr>
<tr>
<td>But Is It User Fr</td>
<td>PC1035</td>
<td>22.95</td>
<td>45,900.00</td>
</tr>
<tr>
<td>Secrets of Silico</td>
<td>PC8888</td>
<td>20.00</td>
<td>40,000.00</td>
</tr>
<tr>
<td>Prolonged Data De</td>
<td>PS3333</td>
<td>19.99</td>
<td>53,713.13</td>
</tr>
<tr>
<td>Fifty Years in Bu</td>
<td>TC4203</td>
<td>11.95</td>
<td>32,265.00</td>
</tr>
<tr>
<td>Fifty Years in Bu</td>
<td>TC4203</td>
<td>11.95</td>
<td>41,825.00</td>
</tr>
</tbody>
</table>

(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:

```sql
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
```

<table>
<thead>
<tr>
<th>title</th>
<th>title_id</th>
<th>price</th>
<th>qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>75</td>
</tr>
</tbody>
</table>
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 \textit{where} clause. However, to include all the rows of the outer table, regardless of whether they satisfy the restriction, place the restriction in the \textit{on} clause.

**Nested ANSI outer joins**

Nested outer joins use the result set of one outer join as the table reference for another. For example:

```sql
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:

```sql
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"
```

<table>
<thead>
<tr>
<th>stor_name</th>
<th>qty</th>
<th>date</th>
<th>ord_num</th>
</tr>
</thead>
<tbody>
<tr>
<td>News &amp; Brews</td>
<td>200</td>
<td>Jun 13</td>
<td>12:00AM</td>
</tr>
<tr>
<td>News &amp; Brews</td>
<td>250</td>
<td>Jun 13</td>
<td>12:00AM</td>
</tr>
<tr>
<td>News &amp; Brews</td>
<td>345</td>
<td>Jun 13</td>
<td>12:00AM</td>
</tr>
<tr>
<td>Thoreau Read</td>
<td>1005</td>
<td>Mar 21</td>
<td>12:00AM</td>
</tr>
<tr>
<td>Thoreau Read</td>
<td>2500</td>
<td>Mar 21</td>
<td>12:00AM</td>
</tr>
<tr>
<td>Thoreau Read</td>
<td>4000</td>
<td>Mar 21</td>
<td>12:00AM</td>
</tr>
</tbody>
</table>

In this example, the second join (between the `stores` and `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:

```sql
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

<table>
<thead>
<tr>
<th>title</th>
<th>price</th>
<th>au_fname</th>
<th>au_lname</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Busy Exe...</td>
<td>19.99</td>
<td>Marjorie</td>
<td>Green</td>
</tr>
<tr>
<td>The Busy Exe...</td>
<td>19.99</td>
<td>Abrahame</td>
<td>Bennet</td>
</tr>
<tr>
<td>Sushi, Anyon...</td>
<td>14.99</td>
<td>Burt</td>
<td>Gringlesby</td>
</tr>
<tr>
<td>Sushi, Anyon...</td>
<td>14.99</td>
<td>Akiko</td>
<td>Yokomoto</td>
</tr>
</tbody>
</table>

(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):

```sql
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"
```

<table>
<thead>
<tr>
<th>title</th>
<th>price</th>
<th>au_fname</th>
<th>au_lname</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Busy Executive’s</td>
<td>19.99</td>
<td>Marjorie</td>
<td>Green</td>
</tr>
<tr>
<td>The Busy Executive’s</td>
<td>19.99</td>
<td>Abrahame</td>
<td>Bennet</td>
</tr>
<tr>
<td>Sushi, Anyone?</td>
<td>14.99</td>
<td>Burt</td>
<td>Gringlesby</td>
</tr>
<tr>
<td>Sushi, Anyone?</td>
<td>14.99</td>
<td>Akiko</td>
<td>Yokomoto</td>
</tr>
</tbody>
</table>

(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.
**Outer joins**

**Converting outer joins with join-order-dependency**

Almost all Transact-SQL outer joins written for earlier versions of Adaptive Server that are run on version 12.0 and later produce the same result set. However, there is a category of outer join queries for which the result sets depend on the join order chosen during optimization. Depending on where in the query the predicate is evaluated, these queries may produce different result sets when they are issued using the later versions of Adaptive Server. The result sets they return are determined by the ANSI rules for assigning predicates to joins.

Predicates cannot be evaluated until all the tables they reference are processed. That is, in the following query the predicate “and titles.price > 20” cannot be evaluated until the titles table is processed:

```sql
select title, price, au_ord
from titles, titleauthor
where titles.title_id *= titleauthor.title_id
and titles.price > 20
```

Predicates in versions of Adaptive Server earlier than 12.0 were evaluated according to these semantics:

- If the predicate was evaluated on the inner table of an outer join, the predicate had **on** clause semantics.
- If the predicate was evaluated with a table that is outer to all outer joins, or is join-order independent, the predicate had **where** clause semantics.

**Note** Before you run Adaptive Server in a production environment, start it with trace flag 4413 and run any queries that you think may be join-order-dependent in versions earlier than 12.0. You see a message similar to the following when you run a join-order-dependent query:

```
Warning: The results of the statement on line %d are join-order independent. Results may differ on pre-12.0 releases, where the query is potentially join-order dependent.
```

Resolve dependencies your applications have on result sets of join-order queries produced versions earlier than 12.0.

Generally, you will not have any problem from join-order-dependent queries because predicates typically reference only:

- An outer table, which is evaluated using **where** clause semantics
- An inner table, which is evaluated using **on** clause semantics
The inner table and tables upon which the inner table is dependent

These do not produce join-order-dependent outer joins. Transact-SQL queries that have any of the following characteristics, however, may produce a different result set after they are translated to an ANSI outer join:

- Predicates that contain `or` statements and reference an inner table of an outer join and another table that this inner table is not dependent on
- Inner table attributes that are referenced in the same predicate as a table which is not in the inner table’s join-order-dependency
- An inner table referenced in a subquery as a correlated reference

The following examples demonstrate the issues of translating Transact-SQL queries with join-order-dependency to ANSI outer join queries.

This query is join-order independent because the outer join references both the `titleauthor` and the `titles` tables, and the `authors` table can be joined with these tables according to three join orders:

- `authors, titleauthors, titles` (as part of the `on` clause)
- `titleauthors, authors, titles` (as part of the `on` clause)
- `titleauthors, titles, authors` (as part of the `where` clause)

```sql
select title, price, authors.au_id, au_lname
from titles, titleauthor, authors
where titles.title_id = titleauthor.title_id
and titleauthor.au_id = authors.au_id
and (titles.price is null or authors.postalcode = '94001')
```

And produces this message:

```
Warning: The results of the statement on line 1 are join-order independent. Results may differ on pre-12.0 releases, where the query is potentially join-order dependent. Use trace flag 4413 to suppress this warning message.
```

This is the ANSI equivalent query:

```sql
select title, price, authors.au_id, au_lname
from titles right join
(titleauthor inner join authors
on titleauthor.au_id = authors.au_id)
on titles.title_id = titleauthor.title_id
where (titles.price is null or authors.postalcode = '94001')
```

The query is join-order-dependent for the same reason as the previous example:

```sql
select title, au_fname, au_lname, titleauthor.au_id, price
```
Outer joins

```
from titles, titleauthor, authors
where authors.au_id *= titleauthor.au_id
and titleauthor.au_ord*titles.price > 40
```

This is the ANSI equivalent query:
```
select title, au_fname, au_lname, titleauthor.au_id, price
from titles,(authors left join titleauthor
on titleauthor.au_id = authors.au_id)
where titleauthor.au_ord*titles.price > 40
```

Transact-SQL outer joins

Transact-SQL includes syntax for both left and right outer joins. The left outer join, `*=`, selects all rows from the first table 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:

```sql
select au_fname, au_lname, pub_name
from authors, publishers
where authors.city *= publishers.city
```

<table>
<thead>
<tr>
<th>au_fname</th>
<th>au_lname</th>
<th>pub_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Johnson</td>
<td>White</td>
<td>NULL</td>
</tr>
<tr>
<td>Marjorie</td>
<td>Green</td>
<td>NULL</td>
</tr>
<tr>
<td>Cheryl</td>
<td>Carson</td>
<td>Algodata Infosystems</td>
</tr>
<tr>
<td>Michael</td>
<td>O’Leary</td>
<td>NULL</td>
</tr>
<tr>
<td>Dick</td>
<td>Straight</td>
<td>NULL</td>
</tr>
<tr>
<td>Meander</td>
<td>Smith</td>
<td>NULL</td>
</tr>
<tr>
<td>Abraham</td>
<td>Bennet</td>
<td>Algodata Infosystems</td>
</tr>
<tr>
<td>Ann</td>
<td>Dull</td>
<td>NULL</td>
</tr>
<tr>
<td>Burt</td>
<td>Gringlesby</td>
<td>NULL</td>
</tr>
<tr>
<td>Chastity</td>
<td>Locksley</td>
<td>NULL</td>
</tr>
<tr>
<td>Morningstar</td>
<td>Greene</td>
<td>NULL</td>
</tr>
<tr>
<td>Reginald</td>
<td>Blotche-Halls</td>
<td>NULL</td>
</tr>
<tr>
<td>Akiko</td>
<td>Yokomoto</td>
<td>NULL</td>
</tr>
<tr>
<td>Innes</td>
<td>del Castillo</td>
<td>NULL</td>
</tr>
<tr>
<td>Michel</td>
<td>DeFrance</td>
<td>NULL</td>
</tr>
<tr>
<td>Dirk</td>
<td>Stringer</td>
<td>NULL</td>
</tr>
<tr>
<td>Stearns</td>
<td>MacFeather</td>
<td>NULL</td>
</tr>
<tr>
<td>Livia</td>
<td>Karsen</td>
<td>NULL</td>
</tr>
<tr>
<td>Sylvia</td>
<td>Panteley</td>
<td>NULL</td>
</tr>
<tr>
<td>Sheryl</td>
<td>Hunter</td>
<td>NULL</td>
</tr>
<tr>
<td>Heather</td>
<td>McBadden</td>
<td>NULL</td>
</tr>
<tr>
<td>Anne</td>
<td>Ringer</td>
<td>NULL</td>
</tr>
<tr>
<td>Albert</td>
<td>Ringer</td>
<td>NULL</td>
</tr>
</tbody>
</table>

(23 rows affected)

The comparison operator `*=` distinguishes the outer join from an ordinary join. This left outer join tells Adaptive Server 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
```  

<table>
<thead>
<tr>
<th>au_fname</th>
<th>au_lname</th>
<th>pub_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>New Age Books</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>Binnet &amp; Hardley</td>
</tr>
<tr>
<td>Cheryl</td>
<td>Carson</td>
<td>Algodata Infosystems</td>
</tr>
<tr>
<td>Abraham</td>
<td>Bennet</td>
<td>Algodata Infosystems</td>
</tr>
</tbody>
</table>

(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)
**Outer joins**

**Using 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 Sybase 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 Sybase 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:

```sql
select publishers.pub_id, titles.price
from publishers, titles
where publishers.pub_id *= titles.pub_id
and titles.price > 20.00
```

```
<table>
<thead>
<tr>
<th>pub_id</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>0736</td>
<td>NULL</td>
</tr>
<tr>
<td>0877</td>
<td>20.95</td>
</tr>
<tr>
<td>0877</td>
<td>21.59</td>
</tr>
<tr>
<td>1389</td>
<td>22.95</td>
</tr>
</tbody>
</table>
```

(4 rows affected)

Similarly, to create an outer join and aggregate column for which the result contains a null-supplied row, enter:

```sql
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
```

```
<table>
<thead>
<tr>
<th>pub_id</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>0736</td>
<td>NULL</td>
</tr>
<tr>
<td>0877</td>
<td>21.59</td>
</tr>
<tr>
<td>1389</td>
<td>22.95</td>
</tr>
</tbody>
</table>
```

(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:

```sql
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)

### 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. This avoids a significant amount of network traffic.

### Using relocated joins

A join between local table `ls1` and remote table `rl1` results in this query being sent to the remote server:

```sql
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.*
Configuring relocated joins

You must specifically enable relocated joins for each remote server involved. The remote server must be able to connect back into the local server using Component Integration Services (CIS).

To configure relocated joins:

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 `sys.servers` 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`

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. Figure 4-1, 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.
Figure 4-1: Null values in outer join

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:

```sql
select *
from t1, t2
where a *= c
```

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>one</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>three</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>4</td>
<td>join4</td>
<td>4</td>
<td>four</td>
</tr>
</tbody>
</table>

(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.

Determining which table columns to join

`sp_helpjoins` lists the columns in two tables or views that are likely join candidates:

```sql
sp_helpjoins table1, table2
```

For example, to find the likely join columns between `titleauthor` and `titles`, use:

```sql
sp_helpjoins titleauthor, titles
```

```
first_pair
```

```
title_id         title_id
-------------     -------------
```
Determining which table columns to join

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.

Subqueries: Using 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. See Chapter 4, “Joins: Retrieving Data from Several Tables.”

How subqueries work

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:

```sql
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.

See the Reference Manual: Commands.

Subquery restrictions

A subquery is subject to these 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; Adaptive Server 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. For more information on using SQL derived tables, see Chapter 9, “SQL-Derived Tables.”

- 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 50.

• 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 Chapter 4, “Expressions, Identifiers, and Wildcard “Characters,” in the Reference Manual: Building Blocks.

Example of using a subquery

To find the books that have the same price as Straight Talk About Computers, first find the price of Straight Talk:

```sql
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:

```sql
select title, price
from titles
where price = $19.99
```

<table>
<thead>
<tr>
<th>title</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Busy Executive’s Database Guide</td>
<td>19.99</td>
</tr>
<tr>
<td>Straight Talk About Computers</td>
<td>19.99</td>
</tr>
<tr>
<td>Silicon Valley Gastronomic Treats</td>
<td>19.99</td>
</tr>
<tr>
<td>Prolonged Data Deprivation: Four Case Studies</td>
<td>19.99</td>
</tr>
</tbody>
</table>
How subqueries work

You can use a subquery to receive the same results in only one step:

```sql
select title, price
from titles
where price =
  (select price
   from titles
   where title = "Straight Talk About Computers")
```

<table>
<thead>
<tr>
<th>title</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Busy Executive’s Database Guide</td>
<td>19.99</td>
</tr>
<tr>
<td>Straight Talk About Computers</td>
<td>19.99</td>
</tr>
<tr>
<td>Silicon Valley Gastronomic Treats</td>
<td>19.99</td>
</tr>
<tr>
<td>Prolonged Data Deprivation: Four Case Studies</td>
<td>19.99</td>
</tr>
</tbody>
</table>

Qualifying column names

Column names in a statement are implicitly qualified by the table 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:

```sql
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:

```sql
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.
Subqueries with correlation names

Table correlation names are required in self-joins because the table being joined to itself appears in two different roles (see Chapter 4, “Joins: Retrieving Data from Several Tables”). 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:

```sql
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:

```sql
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:

```sql
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.
How subqueries work

Multiple levels of nesting

A subquery can include one or more subqueries. You can nest up to 50 subqueries in a statement.

For example, to find the names of authors who have participated in writing at least one popular computing book, enter:

```sql
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")
  )
```

<table>
<thead>
<tr>
<th>au_lname</th>
<th>au_fname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carson</td>
<td>Cheryl</td>
</tr>
<tr>
<td>Dull</td>
<td>Ann</td>
</tr>
<tr>
<td>Locksley</td>
<td>Chastity</td>
</tr>
<tr>
<td>Hunter</td>
<td>Sheryl</td>
</tr>
</tbody>
</table>

(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:

```sql
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"
```

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:

- 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 * from t1 where c1 in (select t2.c1 from t1, t2)
2> go
```

In versions of Adaptive Server earlier than 15.7, you could use an asterisk only in a nested `exists` subquery.

### Using 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, such as:

```
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
```
How subqueries work

Using 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

Examples

Example 1  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

Example 2  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.

Usage

Adaptive Server 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. Adaptive Server
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)
How subqueries work

CREATE PROC P1
AS/* Adaptive Server 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.

Adaptive Server 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.
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.

```sql
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:

```sql
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:

```sql
delete salesdetail
where title_id in
  (select title_id
    from titles
    where type = "business")
```

An equivalent *delete* statement using a join is:

```sql
delete salesdetail
from salesdetail, titles
where salesdetail.title_id = titles.title_id
  and type = "business"
```
How subqueries work

Subqueries in conditional statements

You can use subqueries in conditional statements. Rewrite the preceding subquery that removes all records of sales of business books, as shown in the next example, to check for the records before deleting them:

```sql
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
```

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. For example, a subquery can compare with a column from the inner table of an outer join.

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:

```sql
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 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:

```sql
select title, price,
    (select max(price) from titles)
from titles
where total_sales > 5000
```

<table>
<thead>
<tr>
<th>title</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>You Can Combat Computer Stress!</td>
<td>2.99</td>
</tr>
<tr>
<td>The Gourmet Microwave</td>
<td>2.99</td>
</tr>
<tr>
<td>But Is It User Friendly?</td>
<td>22.95</td>
</tr>
<tr>
<td>Fifty Years in Buckingham Palace Kitchens</td>
<td>11.95</td>
</tr>
</tbody>
</table>

(4 rows affected)

**Types of subqueries**

There are two basic types of subqueries:

- **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).
Types of subqueries

- 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 Adaptive Server actually processes statements with subqueries. Noncorrelated subqueries can alternatively be stated as joins and are processed as joins by Adaptive Server.

- 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. Correlated subqueries are discussed in detail at the end of this chapter.

Expression subqueries

Expression subqueries include:

- Subqueries in a select list (introduced with in)
- Subqueries 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:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>(=)</td>
<td>Equal to</td>
</tr>
<tr>
<td>(&gt;)</td>
<td>Greater than</td>
</tr>
<tr>
<td>(&lt;)</td>
<td>Less than</td>
</tr>
<tr>
<td>(&gt;=)</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>(&lt;=)</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>(!=)</td>
<td>Not equal to</td>
</tr>
<tr>
<td>(&lt;&gt;)</td>
<td>Not equal to</td>
</tr>
<tr>
<td>(!&gt;)</td>
<td>Not greater than</td>
</tr>
<tr>
<td>(!&lt;)</td>
<td>Not less than</td>
</tr>
</tbody>
</table>

Adaptive Server Enterprise
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, Adaptive Server 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
```

**Using 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)
```

```
---------------------------------------------
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?
```
Types of subqueries

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?

Using **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:

```sql
select title
from titles
where price >
  (select min(price)
   from titles
   group by type
   having type = "trad_cook")
```

Using **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:

```sql
select pub_name from publishers
where pub_id =
  (select distinct pub_id
   from titles
   where pub_id = publishers.pub_id)
```
Quantified predicate subqueries

Quantified predicate subqueries, which return a list of zero and 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]

Though Adaptive Server allows the keyword distinct in quantified predicate subqueries, it always processes the subquery as if distinct were not included.

Subqueries with *any and all*

The keywords all and any modify a comparison operator that introduces 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.
Types of subqueries

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:

\[
\{\text{where} \mid \text{having} \}\ [\text{not}] \quad \text{expression comparison\_operator} \{\text{any} \mid \text{all}\} \ (\text{subquery})
\]

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:

```sql
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")
```

```
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.
> *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:

```sql
select title from titles where price > all
(select price from titles
 where type = "mod_cook")
```

<table>
<thead>
<tr>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>---------------------------------------------------</td>
</tr>
<tr>
<td>But Is It User Friendly?</td>
</tr>
<tr>
<td>Secrets of Silicon Valley</td>
</tr>
<tr>
<td>Computer Phobic and Non-Phobic Individuals: Behavior Variations</td>
</tr>
<tr>
<td>Onions, Leeks, and Garlic: Cooking Secrets of the Mediterranean</td>
</tr>
</tbody>
</table>

(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:

```sql
select title from titles where price > all
(select price from titles
 where type = "popular_comp")
```

<table>
<thead>
<tr>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>---------------------------------------------------</td>
</tr>
<tr>
<td>(0 rows affected)</td>
</tr>
</tbody>
</table>

No rows are returned because the subquery finds that one of the books, *Net Etiquette*, has a null price.

= *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.
Types of subqueries

For example, the following query identifies the authors who live in the same city at the postal code:

```sql
select au_fname, au_lname, city
from authors
where city = all
  (select city
   from authors
   where postalcode like "946%")
```

> _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.

```sql
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.

\[ = \text{any} \]

\text{any} means equal to some value

The \( = \text{any} \) operator is an existence check; it is equivalent to \( \text{in} \). For example, to find authors that live in the same city as any publisher, you can use either \( = \text{any} \) or \( \text{in} \):

\[
\text{select au\_lname, au\_fname}
\text{from authors}
\text{where city = any}
\text{(select city}
\text{from publishers)}
\]

\[
\text{select au\_lname, au\_fname}
\text{from authors}
\text{where city in}
\text{(select city}
\text{from publishers)}
\]

<table>
<thead>
<tr>
<th>au_lname</th>
<th>au_fname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carson</td>
<td>Cheryl</td>
</tr>
<tr>
<td>Bennet</td>
<td>Abraham</td>
</tr>
</tbody>
</table>

However, the \( \neq \text{any} \) operator is different from \( \text{not in} \). The \( \neq \text{any} \) operator means “\( \neq a \text{ or } \neq b \text{ or } \neq c \)”; \( \text{not in} \) means “\( \neq a \text{ and } \neq b \text{ and } \neq c \)”.

For example, to find the authors who live in a city where no publisher is located:

\[
\text{select au\_lname, au\_fname}
\text{from authors}
\text{where city \( \neq \text{any} \)}
\text{(select city}
\text{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 \texttt{not} in the same query:

\begin{verbatim}
select au_lname, au_fname 
from authors 
where city not in 
    (select city 
     from publishers)
\end{verbatim}

\begin{verbatim}
au_lname  au_fname
----------  ----------
White      Johnson
Green      Marjorie
O'Leary    Michael
Straight   Dick
Smith      Meander
Dull       Ann
Gringlesby Burt
Locksley   Chastity
Greene     Morningstar
Blotchets-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
\end{verbatim}

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 \texttt{!=all}, which is equivalent to \texttt{not in}:

\begin{verbatim}
select au_lname, au_fname 
from authors 
where city != all 
    (select city 
     from publishers)
\end{verbatim}
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:

```sql
select pub_name
from publishers
where pub_id in
  (select pub_id
   from titles
   where type = "business")
```

```
- 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:

```sql
select pub_name
from publishers
where pub_id in ("1389", "0736")
```

Another way to formulate this query using a subquery is:

```sql
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:

```sql
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"
```

<table>
<thead>
<tr>
<th>pub_name</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algodata Infosystems</td>
<td>The Busy Executive’s Database Guide</td>
</tr>
<tr>
<td>Algodata Infosystems</td>
<td>Cooking with Computers: Surreptitious Balance Sheets</td>
</tr>
<tr>
<td>New Age Books</td>
<td>You Can Combat Computer Stress!</td>
</tr>
<tr>
<td>Algodata Infosystems</td>
<td>Straight Talk About Computers</td>
</tr>
</tbody>
</table>

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)
```

<table>
<thead>
<tr>
<th>au_lname</th>
<th>au_fname</th>
</tr>
</thead>
<tbody>
<tr>
<td>MacFeather Stearns</td>
<td></td>
</tr>
</tbody>
</table>

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
```
CHAPTER 5    Subqueries: Using Queries Within Other Queries

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.

Subqueries used with *not in*

Subqueries that are introduced with the keyword phrase *not in* also return a list of 0 and higher values. *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, the inverse of the example in “Subqueries used with in” on page 185:

```
select pub_name from publishers
where pub_id not in
   (select pub_id
    from titles
    where type = "business")
```

```
----------------------------------------
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. The difficulties interpreting the meaning of joins that are not based on equality are discussed in detail in Chapter 4, “Joins: Retrieving Data from Several Tables.”

Subqueries using *not in* with NULL

A subquery using *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:

```sql
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.

**Subqueries used with exists**

Use the exists keyword with a subquery to test for the existence of some result from the subquery:

```
{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:

```sql
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 (see “Using correlated subqueries” on page 193).

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:

```sql
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)
```
Here are two queries that find titles of books published by any publisher located in a city that begins with the letter “B”:

```sql
select title
from titles
where exists
  (select *
   from publishers
   where pub_id = titles.pub_id
   and city like "B%")
```

```sql
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

**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:

```sql
select pub_name
from publishers
where NOT exists
  (select *
   from titles
   where pub_id = publishers.pub_id
   and title like "business books")
```
CHAPTER 5  Subqueries: Using Queries Within Other Queries

from publishers
where not exists
  (select *
    from titles
    where pub_id = publishers.pub_id
    and type = "business")

------------------------
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

Finding 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)
```

<table>
<thead>
<tr>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gary</td>
</tr>
<tr>
<td>Covelo</td>
</tr>
<tr>
<td>Oakland</td>
</tr>
<tr>
<td>Lawrence</td>
</tr>
<tr>
<td>San Jose</td>
</tr>
<tr>
<td>Ann Arbor</td>
</tr>
<tr>
<td>Corvallis</td>
</tr>
<tr>
<td>Nashville</td>
</tr>
<tr>
<td>Palo Alto</td>
</tr>
<tr>
<td>Rockville</td>
</tr>
<tr>
<td>Vacaville</td>
</tr>
<tr>
<td>Menlo Park</td>
</tr>
<tr>
<td>Walnut Creek</td>
</tr>
<tr>
<td>San Francisco</td>
</tr>
<tr>
<td>Salt Lake City</td>
</tr>
</tbody>
</table>

**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
title where titles.pub_id = publishers.pub_id)
```

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. See Chapter 9, “SQL-Derived Tables.”
Using correlated subqueries

Evaluate many of the previous queries by executing the subquery once and substituting the resulting values into the where clause of the outer query; these are noncorrelated 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)
```

<table>
<thead>
<tr>
<th>au_lname</th>
<th>au_fname</th>
</tr>
</thead>
<tbody>
<tr>
<td>White</td>
<td>Johnson</td>
</tr>
<tr>
<td>Green</td>
<td>Marjorie</td>
</tr>
<tr>
<td>Carson</td>
<td>Cheryl</td>
</tr>
<tr>
<td>Straight</td>
<td>Dick</td>
</tr>
<tr>
<td>Locksley</td>
<td>Chastity</td>
</tr>
<tr>
<td>Blotchet-Hall</td>
<td>Reginald</td>
</tr>
<tr>
<td>del Castillo</td>
<td>Innes</td>
</tr>
<tr>
<td>Panteley</td>
<td>Sylvia</td>
</tr>
<tr>
<td>Ringer</td>
<td>Albert</td>
</tr>
</tbody>
</table>

(9 rows affected)

Unlike most of the previous examples, 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 Adaptive Server 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"
```
Using correlated subqueries

The result is 100, so the outer query evaluates to:

```sql
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:

```sql
select t2.b1, (select t2.b2 from t1 where t2.b1 *= t1.a1) from t2
```

**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:

```sql
select distinct t1.type
from titles t1
where t1.type in
  (select t2.type
   from titles t2
   where t1.pub_id != t2.pub_id)
```

<table>
<thead>
<tr>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>business</td>
</tr>
<tr>
<td>psychology</td>
</tr>
</tbody>
</table>

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:

```sql
select distinct t1.type
from titles t1, titles t2
where t1.type = t2.type
and t1.pub_id != t2.pub_id
```
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:

```sql
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)
```

<table>
<thead>
<tr>
<th>ord_num</th>
<th>title_id</th>
<th>qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>91-A-7</td>
<td>PS3333</td>
<td>90</td>
</tr>
<tr>
<td>91-A-7</td>
<td>PS2106</td>
<td>30</td>
</tr>
<tr>
<td>55-V-7</td>
<td>PS2106</td>
<td>31</td>
</tr>
<tr>
<td>AX-532-FED-452-2Z7</td>
<td>PS7777</td>
<td>125</td>
</tr>
<tr>
<td>BA71224</td>
<td>PS7777</td>
<td>200</td>
</tr>
<tr>
<td>NB-3.142</td>
<td>PS2091</td>
<td>200</td>
</tr>
<tr>
<td>NB-3.142</td>
<td>PS7777</td>
<td>250</td>
</tr>
<tr>
<td>NB-3.142</td>
<td>PS3333</td>
<td>345</td>
</tr>
<tr>
<td>ZD-123-DFG-752-9G8</td>
<td>PS3333</td>
<td>750</td>
</tr>
<tr>
<td>91-A-7</td>
<td>PS7777</td>
<td>180</td>
</tr>
<tr>
<td>356921</td>
<td>PS3333</td>
<td>200</td>
</tr>
</tbody>
</table>

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:

```sql
select t1.type, t1.title
from titles t1
where t1.price >
    (select avg(t2.price)
        from titles t2
        where t1.type = t2.type)
```

<table>
<thead>
<tr>
<th>type</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>business</td>
<td>The Busy Executive’s Database Guide</td>
</tr>
</tbody>
</table>
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:

```sql
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)

---
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.
CHAPTER 6

Using and Creating Datatypes

A **datatype** defines the kind of information each column in a table holds, and how that information is stored. You can use Adaptive Server system datatypes when you are defining columns, or you can create user-defined datatypes.

How Transact-SQL datatypes work

In Transact-SQL, datatypes specify the type of information, size, and storage format of table columns, stored procedure parameters, and local variables. 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.

Adaptive Server 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:

\[
\text{declare @switch bit}
\]

Subsequent chapters describe in more detail how to declare columns, local variables, and parameters using the datatypes described in this chapter. Use `sp_help` to determine which datatypes have been defined for columns of existing tables.

### Using system-supplied datatypes

Table 6-1 lists the system-supplied datatypes provided for various types of information, the synonyms recognized by Adaptive Server, and the range and storage size for each. The system datatypes are printed in lowercase characters, although Adaptive Server allows you to enter them in either uppercase or lowercase. Most Adaptive Server-supplied datatypes are not reserved words and can be used to name other objects.

<table>
<thead>
<tr>
<th>Datatypes by category</th>
<th>Synonyms</th>
<th>Range</th>
<th>Bytes of storage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Exact numeric: integers</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bigint</td>
<td></td>
<td>Whole numbers between (2^{63}) and (2^{-63})</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- 1 (from (-9,223,372,036,854,775,808) to (+9,223,372,036,854,775,807), inclusive)</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>integer</td>
<td>(2^{15} - 1,099,511,627,776) (from (-2,147,483,647) to (+2,147,483,648))</td>
<td>4</td>
</tr>
<tr>
<td>smallint</td>
<td></td>
<td>(2^{15} - 1,32,767) (from (-32,767) to (-32,768))</td>
<td>2</td>
</tr>
<tr>
<td>tinyint</td>
<td></td>
<td>0 to 255 (negative numbers are not permitted)</td>
<td>1</td>
</tr>
<tr>
<td>unsigned bigint</td>
<td></td>
<td>Whole numbers between 0 and 18,446,744,073,709,551,615</td>
<td>8</td>
</tr>
<tr>
<td>unsigned int</td>
<td></td>
<td>Whole numbers between 0 and 4,294,967,295</td>
<td>4</td>
</tr>
<tr>
<td>unsigned smallint</td>
<td></td>
<td>Whole numbers between 0 and 65535</td>
<td>2</td>
</tr>
</tbody>
</table>

**Exact numeric: decimals**
### Datatypes by Category

<table>
<thead>
<tr>
<th>Datatypes by category</th>
<th>Synonyms</th>
<th>Range</th>
<th>Bytes of storage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Numeric</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>numeric (precision, scale)</td>
<td></td>
<td>$10^{-38}$ to $10^{38}$</td>
<td>2 to 17</td>
</tr>
<tr>
<td>decimal (precision, scale)</td>
<td>dec</td>
<td>$10^{-38}$ to $10^{38}$</td>
<td>2 to 17</td>
</tr>
<tr>
<td><strong>Approximate numeric</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>float (precision)</td>
<td>Machine-dependent</td>
<td></td>
<td>4 for default precision &lt; 16, 8 for default precision &gt;= 16</td>
</tr>
<tr>
<td>double precision</td>
<td>Machine-dependent</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>real</td>
<td>Machine-dependent</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td><strong>Money</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>smallmoney</td>
<td></td>
<td>214,748.3647 to -214,748.3648</td>
<td>4</td>
</tr>
<tr>
<td>money</td>
<td></td>
<td>922,337,203,685.477.5807 to -922,337,203,685.57808</td>
<td>8</td>
</tr>
<tr>
<td><strong>Date/Time</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>smalldatetime</td>
<td></td>
<td>January 1, 1900 to June 6, 2079</td>
<td>4</td>
</tr>
<tr>
<td>datetime</td>
<td></td>
<td>January 1, 1753 to December 31, 9999</td>
<td>8</td>
</tr>
<tr>
<td>date</td>
<td></td>
<td>January 1, 0001 to December 31, 9999</td>
<td>4</td>
</tr>
<tr>
<td>time</td>
<td></td>
<td>12:00:00 a.m to 11:59:59:999 p.m.</td>
<td>4</td>
</tr>
<tr>
<td>bigdatetime</td>
<td></td>
<td>January 1, 0001 to December 31, 9999 and 12:00:00.000000 a.m. to 11:59:59.999999 p.m.</td>
<td>8</td>
</tr>
<tr>
<td>bigtime</td>
<td></td>
<td>12:00:00.000000 a.m to 11:59:59.999999 p.m.</td>
<td>8</td>
</tr>
<tr>
<td><strong>Character</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>char(n)</td>
<td>character</td>
<td>page size</td>
<td>n</td>
</tr>
<tr>
<td>varchar(n)</td>
<td>character varying, char varying</td>
<td>page size</td>
<td>Actual entry length</td>
</tr>
<tr>
<td>unichar</td>
<td>Unicode character</td>
<td>page size</td>
<td>n * @@unicharsize ( @@unicharsize equals 2)</td>
</tr>
<tr>
<td>univarchar</td>
<td>Unicode character varying, char varying</td>
<td>page size</td>
<td>actual number of characters * @@unicharsize</td>
</tr>
<tr>
<td>nchar(n)</td>
<td>national character, national char</td>
<td>page size</td>
<td>n * @@ncharsize</td>
</tr>
<tr>
<td>nvarchar(n)</td>
<td>nchar varying, national char varying, national character varying</td>
<td>page size</td>
<td>@@ncharsize * number of characters</td>
</tr>
</tbody>
</table>
Using system-supplied datatypes

<table>
<thead>
<tr>
<th>Datatypes by category</th>
<th>Synonyms</th>
<th>Range</th>
<th>Bytes of storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td></td>
<td>$2^{31} - 1 , (2,147,483,647)$ bytes or fewer</td>
<td>0 when uninitialized; multiple of 2K after initialization</td>
</tr>
<tr>
<td>unitext</td>
<td></td>
<td>1,073,741,823 Unicode characters or fewer</td>
<td>0 when uninitialized; multiple of 2K after initialization</td>
</tr>
<tr>
<td><strong>Binary</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary(n)</td>
<td></td>
<td>pagesize</td>
<td>n</td>
</tr>
<tr>
<td>varbinary(n)</td>
<td></td>
<td>pagesize</td>
<td>actual entry length</td>
</tr>
<tr>
<td>image</td>
<td></td>
<td>$2^{31} - 1 , (2,147,483,647)$ bytes or fewer</td>
<td>0 when uninitialized; multiple of 2K after initialization</td>
</tr>
<tr>
<td><strong>Bit</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bit</td>
<td></td>
<td>0 or 1</td>
<td>1 (one byte holds up to 8 bit columns)</td>
</tr>
</tbody>
</table>

**Exact numeric types: integers**

Adaptive Server provides datatypes bigint, int, smallint, tinyint, unsigned bigint, unsigned int, and unsigned smallint to store integers (whole numbers). These types are exact numeric types; 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. Table 6-2 describes the range of the signed and unsigned versions of these datatypes.
CHAPTER 6 Using and Creating Datatypes

Table 6-2: Ranges for signed and unsigned datatypes

<table>
<thead>
<tr>
<th>Datatype</th>
<th>Range of signed datatypes</th>
<th>Datatype</th>
<th>Range of unsigned datatypes</th>
</tr>
</thead>
<tbody>
<tr>
<td>bigint</td>
<td>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)</td>
<td>unsigned bigint</td>
<td>Whole numbers between 0 and 18,446,744,073,709,551,615</td>
</tr>
<tr>
<td>int</td>
<td>Whole numbers between -2^31 and 2^31 - 1 (-2,147,483,648 and 2,147,483,647), inclusive</td>
<td>unsigned int</td>
<td>Whole numbers between 0 and 4,294,967,295</td>
</tr>
<tr>
<td>smallint</td>
<td>Whole numbers between -2^15 and 2^15 - 1 (-32,768 and 32,767), inclusive</td>
<td>unsigned smallint</td>
<td>Whole numbers between 0 and 65535</td>
</tr>
</tbody>
</table>

Exact numeric types: decimal numbers

Use the exact numeric types, numeric and decimal, for numbers that include decimal points. Data stored in numeric and decimal 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 ])]
```

Adaptive Server 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.

**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 Chapter 1, “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.

**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 Adaptive Server 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.
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. Adaptive Server recognizes a wide variety of data entry formats, which are described in Chapter 7, “Adding, Changing, Transferring, and Deleting Data.” However, Adaptive Server rejects values such as 0 or 00/00/00, which are not recognized as dates.
Using system-supplied datatypes

The default display format for dates is “Apr 15 1987 10:23 p.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 Adaptive Server 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 accommodate this increased precision.

- hh:mi:ss.zzzzzz AM or PM
- hh:mi:ss.zzzzzz
- mon dd yyyy
- hh:mi:ss/yyyy
- yyyy-mm-dd
- hh:mi:ss.zzzzzz

Character datatypes

Use the character datatypes to store strings consisting 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.
### Table 6-3: Character datatypes

<table>
<thead>
<tr>
<th>Datatype</th>
<th>Stores</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>char(n)</code></td>
<td>Fixed-length data, such as social security numbers or postal codes</td>
</tr>
<tr>
<td><code>varchar(n)</code></td>
<td>Data, such as names, that is likely to vary greatly in length</td>
</tr>
<tr>
<td><code>unichar</code></td>
<td>Fixed-length Unicode data, comparable to <code>char</code></td>
</tr>
<tr>
<td><code>univarchar</code></td>
<td>Unicode data that is likely to vary greatly in length, comparable to <code>varchar</code></td>
</tr>
<tr>
<td><code>nvarchar(n)</code></td>
<td>Fixed-length data in multibyte character sets</td>
</tr>
<tr>
<td><code>nvarchar(n)</code></td>
<td>Variable-length data in multibyte character sets</td>
</tr>
<tr>
<td><code>text</code></td>
<td>Up to 2,147,483,647 bytes of printable characters on linked lists of data pages</td>
</tr>
<tr>
<td><code>unitext</code></td>
<td>Up to 1,073,741,823 Unicode characters on linked lists of data pages</td>
</tr>
</tbody>
</table>

Adaptive Server 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, \( (\text{unichar} = n \times \text{@unicharsize}) \); for `nvarchar`, \( n \) times the average national character length \( (\text{@ncharsize}) \). When you create a `char`, `unichar`, or `nvarchar` column that allows nulls, Adaptive Server converts it to a `varchar`, `univarchar`, or `nvarchar` column and uses the storage rules for those datatypes. This is not true of `char` and `nvarchar` 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.

**unichar datatype**

The `unichar` and `univarchar` datatypes support the UTF-16 encoding of Unicode in the Adaptive Server. 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:

```sql
create table unithbl (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, Adaptive Server 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.

### 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.

Table 6-4 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.

Table 6-4: Relational expressions

| 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. |

Join operators

Join operators appear in the same manner as comparison operators. In fact, you can use any comparison operator can be used 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.

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.
Using system-supplied datatypes

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.

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, Adaptive Server 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.

For more information about the text datatype, see “Changing text, unitext, and image data” on page 257 and the Reference Manual: Building Blocks.

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 Adaptive Server 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, Adaptive Server 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, Adaptive Server 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.

Binary datatypes

The 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,” Adaptive Server assumes that the value is an ASCII value and converts it.

Note  Adaptive Server manipulates the binary types in a platform-specific manner. For true hexadecimal data, use the hextoint and inttohex functions. See Chapter 16, “Using Transact-SQL Functions in Queries.”

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, Adaptive Server 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, Adaptive Server 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.
Using system-supplied datatypes

**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, Adaptive Server 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.

See “Changing text, unitext, and image data” on page 257.

**The 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.
The *timestamp* datatype

The timestamp user-defined datatype is necessary for columns in tables that are to be 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.

The *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. See “Creating user-defined datatypes” on page 219.

Using 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 Adaptive Server, and reduces the amount of memory otherwise needed by the client to process the LOB.
Using LOB Locators in Transact-SQL Statements

Adaptive Server 15.7 lets client applications send and receive locators as host variables and parameter markers.

When you create an LOB locator, Adaptive Server 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. Adaptive Server 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.

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.

Creating a LOB locator

You can create a LOB locator explicitly or implicitly.
Explicitly creating 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 Adaptive Server memory.

**Note** When explicitly creating a locator, Adaptive Server 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.

Implicitly creating 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. Each 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
```
Using LOB Locators in Transact-SQL Statements

```sql
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:

```sql
set send_locator on
select textcol from my_table where id = 5
```

Because `send_locator` is on, Adaptive Server creates a locator for each row value of `textcol`, and sends the resulting locators to the client. If `send_locator` is off (the default), Adaptive Server sends the actual text value.

### Converting 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 dereference a locator, use the `return_lob` function. For example, to return the LOB value for `@w`, enter:

```sql
declare @w text_locator
select return_lob(text, @w)
```

You can also implicitly dereference the locator. For example, to insert the actual LOB value for `@w` into the `textcol` column of `my_table`, enter:

```sql
insert my_table(textcol) values (@w)
```

**Note** The `return_lob` command overrides the `set send_locator on` command. `return_lob` always returns the LOB.

### Parameter markers

You can explicitly dereference a locator when using parameter markers in Transact-SQL statements. For example:

```sql
insert my_table (textcol) values (return_lob(text, ?))
```

Use the `locator_literal` function to identify the locator:

```sql
insert my_table (imagecol) values (locator_literal(image_locator, binary_locator_value))
```
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 Adaptive Server memory before the transaction commits, and marks the locator as invalid.

Converting between datatypes

Adaptive Server automatically handles many conversions from one datatype to another. These are called implicit conversions. Use the `convert`, `int2hex`, 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, Adaptive Server 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 Adaptive Server 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.

See Chapter 16, “Using Transact-SQL Functions in Queries” for details on the convert, inttohex, and hextoint functions.

**Mixed-mode arithmetic and datatype hierarchy**

When you perform arithmetic on values with different datatypes, Adaptive Server 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 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

<table>
<thead>
<tr>
<th>name</th>
<th>hierarchy</th>
</tr>
</thead>
<tbody>
<tr>
<td>floatn</td>
<td>1</td>
</tr>
<tr>
<td>float</td>
<td>2</td>
</tr>
<tr>
<td>datetimn</td>
<td>3</td>
</tr>
<tr>
<td>datetime</td>
<td>4</td>
</tr>
<tr>
<td>real</td>
<td>5</td>
</tr>
<tr>
<td>numericn</td>
<td>6</td>
</tr>
<tr>
<td>numeric</td>
<td>7</td>
</tr>
<tr>
<td>decimaln</td>
<td>8</td>
</tr>
<tr>
<td>decimal</td>
<td>9</td>
</tr>
<tr>
<td>moneyn</td>
<td>10</td>
</tr>
<tr>
<td>money</td>
<td>11</td>
</tr>
<tr>
<td>smallmoney</td>
<td>12</td>
</tr>
<tr>
<td>smalldatet</td>
<td>13</td>
</tr>
<tr>
<td>intn</td>
<td>14</td>
</tr>
<tr>
<td>uintn</td>
<td>15</td>
</tr>
<tr>
<td>bigint</td>
<td>16</td>
</tr>
<tr>
<td>ubigint</td>
<td>17</td>
</tr>
<tr>
<td>int</td>
<td>18</td>
</tr>
<tr>
<td>uint</td>
<td>19</td>
</tr>
<tr>
<td>smallint</td>
<td>20</td>
</tr>
<tr>
<td>usmallint</td>
<td>21</td>
</tr>
<tr>
<td>tinyint</td>
<td>22</td>
</tr>
<tr>
<td>bit</td>
<td>23</td>
</tr>
<tr>
<td>univarchar</td>
<td>24</td>
</tr>
<tr>
<td>unichar</td>
<td>25</td>
</tr>
<tr>
<td>unitext</td>
<td>26</td>
</tr>
<tr>
<td>sysname</td>
<td>27</td>
</tr>
<tr>
<td>varchar</td>
<td>27</td>
</tr>
<tr>
<td>nvarchar</td>
<td>27</td>
</tr>
<tr>
<td>longsysnam</td>
<td>27</td>
</tr>
<tr>
<td>char</td>
<td>28</td>
</tr>
<tr>
<td>nchar</td>
<td>28</td>
</tr>
<tr>
<td>timestamp</td>
<td>29</td>
</tr>
<tr>
<td>varbinary</td>
<td>29</td>
</tr>
<tr>
<td>binary</td>
<td>30</td>
</tr>
<tr>
<td>text</td>
<td>31</td>
</tr>
<tr>
<td>image</td>
<td>32</td>
</tr>
<tr>
<td>date</td>
<td>33</td>
</tr>
<tr>
<td>time</td>
<td>34</td>
</tr>
<tr>
<td>daten</td>
<td>35</td>
</tr>
<tr>
<td>timen</td>
<td>36</td>
</tr>
</tbody>
</table>
Mixed-mode arithmetic and datatype hierarchy

<table>
<thead>
<tr>
<th>datatype</th>
<th>hierarchy</th>
</tr>
</thead>
<tbody>
<tr>
<td>bigdatetim</td>
<td>37</td>
</tr>
<tr>
<td>bigtime</td>
<td>38</td>
</tr>
<tr>
<td>bigdatetim</td>
<td>39</td>
</tr>
<tr>
<td>bigtimen</td>
<td>40</td>
</tr>
<tr>
<td>extended t</td>
<td>99</td>
</tr>
</tbody>
</table>

**Note** `uinteger_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, Adaptive Server issues a conversion error.

See the *Reference Manual: Building Blocks* for more information about the datatype hierarchy.

**Working with money datatypes**

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:
CHAPTER 6  Using and Creating Datatypes

```sql
select convert (money, moneycol * percentcol)
    from debits, interest
drop table mytable
```

Determining precision and scale

For the numeric and decimal types, each combination of precision and scale is a distinct Adaptive Server 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 \( s2 \), Adaptive Server determines the precision and scale of the results as shown in Table 6-5:

### Table 6-5: Precision and scale after arithmetic operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Precision</th>
<th>Scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n1 + n2 )</td>
<td>( \max(s1, s2) + \max(p1 - s1, p2 - s2) + 1 )</td>
<td>( \max(s1, s2) )</td>
</tr>
<tr>
<td>( n1 - n2 )</td>
<td>( \max(s1, s2) + \max(p1 - s1, p2 - s2) + 1 )</td>
<td>( \max(s1, s2) )</td>
</tr>
<tr>
<td>( n1 \times n2 )</td>
<td>( s1 + s2 + (p1 - s1) + (p2 - s2) + 1 )</td>
<td>( s1 + s2 )</td>
</tr>
<tr>
<td>( n1 / n2 )</td>
<td>( \max(s1 + p2 + 1, 6) + p1 - s1 + s2 )</td>
<td>( \max(s1 + p2 + 1, 6) )</td>
</tr>
</tbody>
</table>

Creating 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. See Chapter 14, “Defining Defaults and Rules for Data.”
Use `sp_addtype` to create user datatypes. It takes as parameters the name of the datatype being created, the Adaptive Server-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 Adaptive Server-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`.

Specifying length, precision, and scale

In some cases, you must specify additional parameters:

- The `char`, `nchar`, `varchar`, `nvarchar`, `binary`, and `varbinary` datatypes expect a length in parentheses. If you do not supply one, Adaptive Server assumes the default length of 1 character.
- The `float` datatype expects a precision in parentheses. If you do not supply one, Adaptive Server 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, Adaptive Server 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.

Specifying 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”, “nonnull”, “NONULL”, “not null”, or “NOT NULL”. `bit` and identity types do not allow null values.
If you omit the null type, Adaptive Server 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.

**Associating 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.

See Chapter 14, “Defining Defaults and Rules for Data.”

**Creating 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:

```sql
sp_addtype typename, "numeric (precision, 0)", "identity"
```

The following example creates a user-defined type, `IdentType`, with the IDENTITY property:

```sql
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:

```sql
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)
```
Getting information about datatypes

```sql
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.

Creating 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.

Dropping 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.

Getting 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:

```sql
sp_help money
```

<table>
<thead>
<tr>
<th>Type_name</th>
<th>Storage_type</th>
<th>Length</th>
<th>Prec</th>
<th>Scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>money</td>
<td>money</td>
<td>8</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nulls</th>
<th>Default_name</th>
<th>Rule_name</th>
<th>Identity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>
(return status = 0)

```
sp_help tid
  Type_name | Storage_type | Length | Prec | Scale |
          |             | ------ | ----- | ------
  tid     | varchar     | 6      | NULL | NULL   |
          |             |        | NULL | NULL   |
```

Nulls | Default_name | Rule_name | Identity
      |             |          | ------
  0 | NULL | NULL | 0

(return status = 0)
Getting information about datatypes
CHAPTER 7

Adding, Changing, Transferring, and Deleting Data

After you create a database, tables, and indexes, you can put data into the tables and work with it—adding, changing, and deleting data 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.
Referential integrity

The data modification commands also work on views, with some restrictions. See Chapter 12, “Views: Limiting Access to Data.” Database owners and the owners of database objects can use the grant and revoke commands to specify the users who are allowed 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 Chapter 17, “Managing User Permissions,” in the System Administration Guide: Volume 1.

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.

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.

Keeping data modifications consistent throughout all tables in a database is called referential integrity. One way to manage it 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). See Chapter 20, “Triggers: Enforcing Referential Integrity”; and Chapter 8, “Creating Databases and Tables.”

To delete data from referential integrity tables, change the referenced tables first and then the referencing table.
Transactions

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 Adaptive Server 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.

There is a more complete discussion of transactions in Chapter 23, “Transactions: Maintaining Data Consistency and Recovery.”

Using the sample databases

To follow the examples in this chapter, Sybase 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 with this chapter. 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
```
Datatype entry rules

Several of the Adaptive Server-supplied datatypes have special rules for entering and searching for data, which are discussed here. For datatypes, see Chapter 8, “Creating Databases and Tables.”

char, nchar, unichar, univarchar, varchar, nvarchar, unitext, and text

All character, text, date, and time 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, Adaptive Server 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), Adaptive Server treats it as text data, which has restrictive rules regarding implicit and explicit conversion to other datatypes. See Chapter 1, “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_truncation 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 described in Chapter 2, “Queries: Selecting Data from a Table,” to search for character, text, and datetime data.


date and time

Adaptive Server allows datatypes `datetime`, `smalldatetime`, `date`, `time`, `bigdatetime`, and `bigtime`.

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 1997 10:23PM.” The `convert` command provides options to display seconds and milliseconds and to display the date with other date-part ordering. See Chapter 16, “Using Transact-SQL Functions in Queries,” for more information about displaying date values.

Adaptive Server 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, Adaptive Server treats the entry as an identifier.

Adaptive Server 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 case Adaptive Server 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. Adaptive Server 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. For `bigtime`, the earliest time is 12:00:00.000000AM; the latest is 11:59:59.999999PM.
Datatype entry rules

Entering times

The order of time components is significant. First, enter the hours; 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]
- 4 am
- 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]
```

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 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.
Entering 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”.

Date formats

Adaptive Server recognizes three basic date formats, as described below. Each format must be enclosed in quotes and can be preceded or followed by a time specification, as described under “Entering times” on page 230.

- The month is entered in alphabetic format.
- Valid formats for specifying the date alphabetically are:
  - `15 [19]97 apr[1l]
    1997 APR[IL] [15]
- 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.
Datatype entry rules

- If the day is missing, Adaptive Server 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:
  \(<\text{num}> <\text{sep}> <\text{num}> <\text{sep}> <\text{num}> [ <\text{time spec}> ]\)
  or:
  \([ <\text{time spec}> ] <\text{num}> <\text{sep}> <\text{num}> <\text{sep}> <\text{num}>\)
- 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)
  \(1997/15/[0]4\) (ymd)
  \([0]4/[19]97/15\) (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.

Searching for 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, Adaptive Server 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 Adaptive Server 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.
Datatype entry rules

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.

**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), Adaptive Server treats the literal as `varbinary` data. If the length of the literal is greater than 16KB, Adaptive Server 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”.


**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.

**float, real, and double precision**

Enter the 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, Adaptive Server multiplies the mantissa by 10 raised to the given exponent. Table 7-1 shows examples of float, real, and double precision data:

<table>
<thead>
<tr>
<th>Data entered</th>
<th>Mantissa</th>
<th>Exponent</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>10E2</td>
<td>10</td>
<td>2</td>
<td>10 * 10^2</td>
</tr>
<tr>
<td>15.3e1</td>
<td>15.3</td>
<td>1</td>
<td>15.3 * 10^1</td>
</tr>
<tr>
<td>-2.e5</td>
<td>-2</td>
<td>5</td>
<td>-2 * 10^5</td>
</tr>
<tr>
<td>2.2e-1</td>
<td>2.2</td>
<td>-1</td>
<td>2.2 * 10^-1</td>
</tr>
<tr>
<td>+56E+2</td>
<td>56</td>
<td>2</td>
<td>56 * 10^2</td>
</tr>
</tbody>
</table>

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, Adaptive Server flags the entry as an error.

**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*, which you specify using this syntax:

```
datatype [(precision [, scale ])]
```
Adaptive Server 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, Adaptive Server flags the entry as an error. Here are some examples of valid dec and numeric data:

<table>
<thead>
<tr>
<th>Data entered</th>
<th>Datatype</th>
<th>Precision</th>
<th>Scale</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.345</td>
<td>numeric(5,3)</td>
<td>5</td>
<td>3</td>
<td>12.345</td>
</tr>
<tr>
<td>-1234.567</td>
<td>dec(8,4)</td>
<td>8</td>
<td>4</td>
<td>-1234.567</td>
</tr>
</tbody>
</table>

The following entries result in errors because they exceed the column’s precision or scale:

<table>
<thead>
<tr>
<th>Data entered</th>
<th>Datatype</th>
<th>Precision</th>
<th>Scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234.567</td>
<td>numeric(3,3)</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1234.567</td>
<td>decimal(6)</td>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>

**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, as described in the preceding section.
**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. Adaptive Server updates the `timestamp` value after each insert or update. See “Inserting data into specific columns” on page 238.

---

**Adding new data**

Use the `insert` command to add rows to the database in two ways:

- 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:

  ```sql
  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 50 tables, including the table into which you are inserting). A simplified version of the syntax for the `insert` command using a `select` statement is:

  ```sql
  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. See “Inserting data into specific columns” on page 238 and “Changing text, unitext, and image data” on page 257.
Adding new rows with values

This insert statement adds a new row to the publishers table, giving a value for every column in the row:

```sql
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.

Inserting 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:

```sql
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:

```sql
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"

Adaptive Server enters null values in the city and state columns because no value was given for these columns in the insert statement, and the publishers table allows null values in these columns.

Restricting 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. See Chapter 14, “Defining Defaults and Rules for Data.”
Adding new data

Using the NULL character string

Only columns for which NULL was specified in the create table statement and into which you have explicitly entered NULL (no quotes), or into which no data has been entered, 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, Adaptive Server 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.

Inserting 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.

**Adding rows without values in all columns**

When you specify values for only some of the columns in a row, one of these things happen to the columns with no values:

- If a default value exists for the column or user-defined datatype of the column, it is entered. See Chapter 14, “Defining Defaults and Rules for Data,” or `insert` in the Reference Manual: Commands.

- 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, Adaptive Server rejects the row and displays an error message.

Table 7-4 shows what you would see under these circumstances:

<table>
<thead>
<tr>
<th>Default exists for column or datatype</th>
<th>Column defined NOT NULL</th>
<th>Column defined to allow NULL</th>
<th>Column is IDENTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>The default</td>
<td>The default</td>
<td>Next sequential value</td>
</tr>
<tr>
<td>No</td>
<td>Error message</td>
<td>NULL</td>
<td>Next sequential value</td>
</tr>
</tbody>
</table>

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_helpdef`.

**Changing a column’s value to NULL**

To set a column value to NULL, use the `update` statement:

```sql
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:

```sql
update titles
```
Adding new data

```sql
set advance = null
where title_id = "TC3218"
```

**Adaptive-Server-generated values for IDENTITY columns**

When you insert a row into a table with an IDENTITY column, Adaptive Server 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`:

```sql
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:

```sql
create table idtext (a int, b numeric identity, c char(1))
-------------------
(1 row affected)
insert idtext values(98,"z")
-------------------
(1 row affected)
insert idtext values (99, "v")
-------------------
(1 row affected)
select * from idtext
-------------------
98   1   z
99   2   v
(2 rows affected)
```

The following statement shows the row that was added to `sales_daily`. Adaptive Server automatically generates the next sequential value, 2, for `row_id`:

```sql
select * from sales_daily
where stor_id = "7896"
sale_id  stor_id
```
Explicitly inserting 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:

```sql
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** Adaptive Server 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.
Retrieving 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:

```sql
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
```

Reserving a block of IDENTITY column values

The identity grab size configuration parameter allows each Adaptive Server 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 Adaptive Server engine must hold an internal synchronization structure when inserting implicit identity values. For example, to set the number of reserved values to 20:

```sql
sp_configure "identity grab size", 20
```
When a user performs an insert into a table containing an IDENTITY column, Adaptive Server 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, Adaptive Server 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:

```sql
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:

```sql
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")
```

Adaptive Server 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`. Adaptive Server 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 Adaptive Server, 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.

### Reaching the IDENTITY column’s maximum value

The maximum value that you can insert into an IDENTITY column is \(10^{\text{Precision} - 1}\). If you do not specify a precision for the IDENTITY column, Adaptive Server 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.
Adding new data

Modify the IDENTITY column’s maximum value

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.

Create a new table with a larger precision

If the table contains IDENTITY columns that are used for referential integrity, retain the current numbers for the IDENTITY column values.

1 Use `create table` to create a new table that is identical to the old one except with a larger precision value for the IDENTITY column.
2 Use `insert into` to copy the data from the old table into the new one.

Renumber the table’s 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.

To sequentially renumber IDENTITY column values and remove the gaps:

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 Adaptive Server, 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 Adaptive Server assign the IDENTITY column values when loading data from the host file, and `-c` instructs `bcp` to use character mode.

4 In Adaptive Server, drop the old table, and use `sp_rename` to change the new table name to the old table name.
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*.

**Adding new rows with `select`**

To pull values into 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, Adaptive Server 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 Adaptive Server 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, Adaptive Server 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.

### Using 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.

### Inserting data into some columns

You can use the select statement to add data to some, but not all, columns in a row. Simply specify the columns to which you want to add data in the insert clause.

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.

Inserting 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

<table>
<thead>
<tr>
<th>pub_id</th>
<th>pub_name</th>
<th>city</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>0736</td>
<td>New Age Books</td>
<td>Boston</td>
<td>MA</td>
</tr>
<tr>
<td>0877</td>
<td>Binnet &amp; Hardley</td>
<td>Washington</td>
<td>DC</td>
</tr>
<tr>
<td>1389</td>
<td>Algodata Infosystems</td>
<td>Berkeley</td>
<td>CA</td>
</tr>
<tr>
<td>9999</td>
<td>test</td>
<td>Boston</td>
<td>MA</td>
</tr>
</tbody>
</table>

(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.
Creating 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.

Adaptive Server treats nonmaterialized columns the same way it treats null columns: if a column is not physically present in the row, Adaptive Server 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.

Adding nonmaterialized columns

Use alter table . . . not materialized to create nonmaterialized columns:

```
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
```

Adaptive Server 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.

Adaptive Server 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.

**Tables that already have nonmaterialized columns**

Adaptive Server 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.

**Nonmaterialized column storage**

Once nonmaterialized columns are instantiated, Adaptive Server stores them in the same way as any other row in the table.

When Adaptive Server 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, Adaptive Server instantiates any columns that are nonmaterialized.
Creating nonmaterialized, non-null columns

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, Adaptive Server 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, Adaptive Server 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).

Altering 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.

Limitations

Adaptive Server version 15.7 does not support `alter table ... modify` for 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 Adaptive Servers using data rows with nonmaterialized, non-NULL columns to versions of Adaptive Servers earlier than version 15.7. You must first run `reorg rebuild` on tables containing these columns to convert them to regular, non-nullable columns.
Changing 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.

Adaptive Server 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 discussion of `writetext` in “Changing text, unitext, and image data” on page 257.

See the *Reference Manual: Commands*. 
Changing existing data

Using the set clause with update

The set clause specifies columns and the changed values. The where clause determines which rows are to be updated. If you do not have a 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.

Assigning 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"

<table>
<thead>
<tr>
<th>title_id</th>
<th>total_sales</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>19.99</td>
</tr>
<tr>
<td></td>
<td>4096</td>
</tr>
</tbody>
</table>

(1 row affected)

See the Reference Manual: Commands. For more information on declared variables, see “Local variables” on page 511.

Using 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:

```sql
update authors
set au_lname = "MacBadden"
where au_lname = "McBadden"
and au_fname = "Heather"
```

Using the from clause with update

Use the from clause to pull data from one or more tables into the table you are updating.

For example, earlier in this chapter, an example was given for inserting some 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. When one of these authors, Dirk Stringer, writes a book, *The Psychology of Computer Cooking*, a title identification number is assigned to his book in the titles table. You can modify his row in the titleauthor table by adding a title identification number for him:

```sql
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"
```
Changing existing data

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.

Performing updates with joins

The following example 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"
```

Updating 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
```
Changing 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:

```sql
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. Adaptive Server 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:

```sql
/* 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
```
Truncating trailing zeros

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."

For more information on batch files and the control-of-flow language used in
this example, see Chapter 15, “Using Batches and Control-of-Flow
Language.”

Truncating trailing zeros

The disable varbinary truncation configuration parameter enables or disables
trailing zeros from varbinary and binary null data (see Chapter 5, “Setting

By default, disable varbinary truncation is off for the server.

When Adaptive Server 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)

Adaptive Server 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, Adaptive Server does not truncate the zeros:

```sql
select * from test1
coll
-----------
0x12345600
```

Adaptive Server considers data with or without trailing zeros as equal for comparisons (that is, 0x1234 is the same as 0x123400).

Because Adaptive Server 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, Adaptive Server 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:

```sql
sp_configure "disable varbinary truncation", 1
select * into test2 from test1
```

Then reinsert the same data again:

```sql
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 set when you ran `select into`:

```sql
select * from test2
coll
-----------
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.
Truncating trailing zeros

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
```
CHAPTER 7 Adding, Changing, Transferring, and Deleting Data

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 Adaptive Server 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
  ```
Transferring data incrementally

Transferring data incrementally

The transfer command allows you to transfer data incrementally, and, if required, to a different product. In versions of Adaptive Server earlier than 15.5, you could transfer only whole tables from one Adaptive Server to another.

**Note** Adaptive Server 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 Adaptive Server 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: IQ (Sybase IQ), ASE (Adaptive Server Enterprise), bulk-copy (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 Adaptive Server is running (the file can be an NFS file that Adaptive Server 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.
Marking 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.

Transferring tables from a destination file

Use the `transfer table` command to load data into Adaptive Server from a table contained in an external file. See the Reference Manual: Commands.

Note Adaptive Server version 15.5 must use its internal format to import tables.

The table you are loading 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 Adaptive Server to find and remove the old row the new row replaces.
Transferring data incrementally

The table you are loading 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, Adaptive Server 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 /sybase/data into Adaptive Server:

```sql
transfer table titles from '/sybase/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. Adaptive Server version 15.5 includes 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)

### Converting Adaptive Server datatypes to IQ

Table 7-5 shows how Adaptive Server datatypes are represented in Sybase IQ. When transferring the datatypes, Adaptive Server makes any necessary transformations to convert its data to the indicated IQ format.
### Table 7-5: Adaptive Server to IQ datatype conversion

<table>
<thead>
<tr>
<th>Adaptive Server</th>
<th>Size (in bytes)</th>
<th>IQ</th>
<th>Size (in bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bigint</td>
<td>8</td>
<td>bigint</td>
<td>8</td>
</tr>
<tr>
<td>unsigned bigint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>unsigned int</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>smallint</td>
<td>2</td>
<td>smallint</td>
<td>2</td>
</tr>
<tr>
<td>unsigned smallint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tinyint</td>
<td>1</td>
<td>tinyint</td>
<td>1</td>
</tr>
<tr>
<td>numeric(P,S)</td>
<td>2 – 17</td>
<td>numeric(P,S)</td>
<td>2 – 26</td>
</tr>
<tr>
<td>decimal(P,S)</td>
<td></td>
<td>decimal(P,S)</td>
<td></td>
</tr>
<tr>
<td>double precision</td>
<td>8</td>
<td>double</td>
<td>8</td>
</tr>
<tr>
<td>real</td>
<td>4</td>
<td>real</td>
<td>4</td>
</tr>
<tr>
<td>float(P)</td>
<td>4, 8</td>
<td>float(P)</td>
<td>4, 8</td>
</tr>
<tr>
<td>money</td>
<td>8</td>
<td>money</td>
<td>IQ stores this as numeric(19,4)</td>
</tr>
<tr>
<td>smallmoney</td>
<td>4</td>
<td>smallmoney</td>
<td>IQ stores this as numeric(10,4)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bigdatetime</td>
<td>8</td>
<td>datetime</td>
<td>8</td>
</tr>
<tr>
<td>datetime</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>smalldatetime</td>
<td>4</td>
<td>smalldatetime</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>date</td>
<td>4</td>
<td>date</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>time</td>
<td>4</td>
<td>time</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bigtime</td>
<td>8</td>
<td>time</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>char(N)</td>
<td>1 – 16296</td>
<td>char(N)</td>
<td>1 – 16296</td>
</tr>
<tr>
<td>char(N) (null)</td>
<td>1 – 16296</td>
<td>char(N) (null)</td>
<td>1 – 16296</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>varchar(N) (null)</td>
<td>1 – 16296</td>
<td>varchar(N) (null)</td>
<td>1 – 16296</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>unichar(N)</td>
<td>1 – 8148</td>
<td>binary(N*2)</td>
<td>1 – 16296</td>
</tr>
<tr>
<td>univarchar(N) (null)</td>
<td>1 – 8148</td>
<td>varbinary(N*2) (null)</td>
<td>1 – 16296</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary(N)</td>
<td>1 – 16296</td>
<td>binary(N)</td>
<td>1 – 16296</td>
</tr>
<tr>
<td>varbinary(N)</td>
<td>1 – 16296</td>
<td>varbinary(N)</td>
<td>1 – 16296</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary(N) (null)</td>
<td>1 – 16296</td>
<td>varbinary(N) (null)</td>
<td>1 – 16296</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bit</td>
<td>1</td>
<td>bit</td>
<td>1</td>
</tr>
</tbody>
</table>
Transferring data incrementally

Consider the following when converting Adaptive Server datatypes to datatypes for IQ:

- Define precision and scale the same on IQ and Adaptive Server.
- The storage size for `float` is 4 or 8 bytes, depending on precision. If you do not supply a value for precision, Adaptive Server stores `float` as `double precision`, but IQ stores it as `real`. Adaptive Server 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 Adaptive Server for a column with datatypes `char`, `unichar`, or `binary` depends on your installation’s page size. The maximum size specified in Table 7-5 is the largest possible column in a 16K page.
- Adaptive Server typically requires two bytes per character to store Unicode characters. Because IQ does not include the Unicode datatype, Adaptive Server transmits `unichar(N)` to IQ as `binary(N X 2)`. However, Adaptive Server does not transform Unicode characters: Adaptive Server pads Unicode strings with NULL(0x00) to transfer them to IQ.
- IQ does not have a native Unicode datatype. Adaptive Server transfers Unicode strings to IQ as binary data, that is, two bytes long for each Unicode character. For example, `unichar(40)` in Adaptive Server converts to `binary(80)` in IQ. IQ cannot display the Unicode data as strings after the transfer.

### Storing transfer information

Transfer information is stored in:

- `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

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):

```
timestamp     8
varbinary(8) null 8
```

### Table 7-5: Adaptive Server vs. IQ Datatypes

<table>
<thead>
<tr>
<th>Adaptive Server</th>
<th>IQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>timestamp</td>
<td>8</td>
</tr>
<tr>
<td>varbinary(8)</td>
<td>8</td>
</tr>
</tbody>
</table>

Adaptive Server Enterprise
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.


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 Adaptive Server, but Adaptive Server 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 Adaptive Server creates the table automatically).

sp_help reports incremental transfer as a table attribute.

The columns in spt_TableTransfer are:

<table>
<thead>
<tr>
<th>Column</th>
<th>Datatype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>end_code</td>
<td>unsigned smallint</td>
<td>The transfer’s ending status. 0 – success. error code – failure.</td>
</tr>
<tr>
<td>id</td>
<td>int</td>
<td>The object ID of the transferred table.</td>
</tr>
<tr>
<td>ts_floor</td>
<td>bigint</td>
<td>The beginning transaction timestamp.</td>
</tr>
<tr>
<td>ts_ceiling</td>
<td>bigint</td>
<td>The transaction timestamp after which rows are uncommitted, and therefore not transferred.</td>
</tr>
</tbody>
</table>
Transferring data incrementally

<table>
<thead>
<tr>
<th>Column</th>
<th>Datatype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>time_begin</td>
<td>datetime not null</td>
<td>• The date and time the transfer began, or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The time Adaptive Server 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.</td>
</tr>
<tr>
<td>time_end</td>
<td>datetime not null</td>
<td>• The date and time the transfer ended, or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 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</td>
</tr>
<tr>
<td>row_count</td>
<td>bigint not null</td>
<td>The number of rows transferred.</td>
</tr>
<tr>
<td>byte_count</td>
<td>bigint not null</td>
<td>The number of bytes written.</td>
</tr>
<tr>
<td>sequence_id</td>
<td>int not null</td>
<td>A number that tracks this transfer, unique to each transfer of a table.</td>
</tr>
<tr>
<td>col_order</td>
<td>tinyint not null</td>
<td>A number representing the column order of the output:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 1 – id</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 2 – offset</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 3 – name</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 4 – name_utf8</td>
</tr>
<tr>
<td>output_to</td>
<td>tinyint not null</td>
<td>A number representing the output format:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 1 – ase</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 2 – bcp</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 3 – csv</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 4 – iq</td>
</tr>
<tr>
<td>tracking_id</td>
<td>int null</td>
<td>An optional customer-supplied tracking ID. If you do not use with tracking_id = nnn, this column is null.</td>
</tr>
<tr>
<td>pathname</td>
<td>varchar (512) null</td>
<td>The output file name.</td>
</tr>
<tr>
<td>row_sep</td>
<td>varchar (64) null</td>
<td>The row separator string used for csv.</td>
</tr>
<tr>
<td>col_sep</td>
<td>varchar(64) null</td>
<td>The column separator used for csv.</td>
</tr>
</tbody>
</table>

monTableTransfer monTableTransfer table provides:
• Historical transfer information for tables for which Adaptive Server currently holds transfer information in memory. This is true for any table accessed since the most recent Adaptive Server restart, unless you have not configured Adaptive Server memory large enough to hold all current table information that:
  • Information about transfers currently in progress and for completed transfers for tables that Adaptive Server 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 Adaptive Server.
    • 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.

Exceptions and errors

You cannot use the data transfer utility to transfer data:
• Using the for bcp format over pipes
• Using pipes on the Windows platform

Adaptive Server generates an error message if you attempt to:
• 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
Transferring data incrementally

- 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 Adaptive Server has write permission in the directory.
- Adaptive Server cannot obtain sufficient memory for saved data for each table in a transfer. If this happens, increase the amount of memory available to Adaptive Server.

Sample session for incremental data transfer

This tutorial describes 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, showing that transfer appends the data to the file, and doesn’t overwrite it.

Note Although this example transfers data out of, and back into, the same table, 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:
   
   ```sql
   sp_setup_table_transfer
   ```

2. Configure max transfer history. The default is 10, which means that Adaptive Server 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:

   ```sql
   sp_configure 'max transfer history', 5
   ```

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Default</th>
<th>Memory Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Config Value</td>
<td>Run Value</td>
<td>Instance Name</td>
</tr>
<tr>
<td>Type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>----------------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Adaptive Server Enterprise
3 Create the transfer_example table, which has the transfer attribute enabled and uses datarow locking:

```sql
create table transfer_example (f1 int, f2 varchar(30), f3 bigdatetime, primary key (f1))
with transfer table on
lock datarows
```

4 Populate the transfer_example table with sample data:

```sql
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:

<table>
<thead>
<tr>
<th>f1</th>
<th>f2</th>
<th>f3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AAA</td>
<td>Jul 17 2009 4:40:14.465789PM</td>
</tr>
<tr>
<td>2</td>
<td>BBBBBB</td>
<td>Jul 17 2009 4:40:14.480031PM</td>
</tr>
<tr>
<td>3</td>
<td>CCCCCCCC</td>
<td>Jul 17 2009 4:40:14.511749PM</td>
</tr>
<tr>
<td>4</td>
<td>DDDDDDDDDDDDDD</td>
<td>Jul 17 2009 4:40:14.536653PM</td>
</tr>
<tr>
<td>5</td>
<td>EEEEEEEEEEEEEE</td>
<td>Jul 17 2009 4:40:14.559480PM</td>
</tr>
<tr>
<td>6</td>
<td>FFFFFFFFFFFFFFFFFP</td>
<td>Jul 17 2009 4:40:14.583400PM</td>
</tr>
<tr>
<td>7</td>
<td>GGGGGGGGGGGGGGGG</td>
<td>Jul 17 2009 4:40:14.607196PM</td>
</tr>
<tr>
<td>8</td>
<td>HHHHHHHHHHHHHHHHHHH</td>
<td>Jul 17 2009 4:40:14.632152PM</td>
</tr>
<tr>
<td>9</td>
<td>IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII</td>
<td>Jul 17 2009 4:40:14.655184PM</td>
</tr>
<tr>
<td>10</td>
<td>JJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJ</td>
<td>Jul 17 2009 4:40:14.678938PM</td>
</tr>
</tbody>
</table>
5 Transfer the transfer_example data to an external file using the for ase format:

```sql
transfer table transfer_example
to 'transfer_example-data.ase'
for ase
```

(10 rows affected)

The data transfer creates this history record in spt_TableTransfer:

```sql
select id, sequence_id, end_code, ts_floor, ts_ceiling, row_count
from spt_TableTransfer
where id = object_id('transfer_example')
```

<table>
<thead>
<tr>
<th>id</th>
<th>sequence_id</th>
<th>end_code</th>
<th>ts_floor</th>
<th>ts_ceiling</th>
<th>row_count</th>
</tr>
</thead>
<tbody>
<tr>
<td>592002109</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>5309</td>
<td>10</td>
</tr>
</tbody>
</table>

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):

```sql
alter table transfer_example
set transfer table off
```

After the alter table command runs, spt_TableTransfer is empty:

```sql
select id, sequence_id, end_code, ts_floor, ts_ceiling, row_count
from spt_TableTransfer
where id = object_id('transfer_example')
```

<table>
<thead>
<tr>
<th>id</th>
<th>sequence_id</th>
<th>end_code</th>
<th>ts_floor</th>
<th>ts_ceiling</th>
<th>row_count</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(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:

```sql
update transfer_example
set f2 = 'no data',
f3 = 'Jan 1, 1900 12:00:00.000001AM'
```
**CHAPTER 7 Adding, Changing, Transferring, and Deleting Data**

(10 rows affected)

After the update, `transfer_example` contains this data.

```sql
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 into the table. Because it has a unique primary index, the incoming rows replace the existing data and do not create duplicate key errors:

```sql
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.

```sql
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   DDDDDDDDDDDDD Jul 17 2009 4:40:14.536653PM
5   EEEEEEEEEEEEEEE Jul 17 2009 4:40:14.559480PM
```
Transferring data incrementally

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)
```

Replacing data with new rows

If, when using incremental transfer, 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 whose keys have not changed.

1 transfer_example uses the f1 column as the primary-key column. Adaptive Server 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
------ -------------------------------
3    ------- ------------------------------ ----------------------------
3 Transfer transfer_example out again: all 13 rows are transferred. Adaptive Server 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: Adaptive Server 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
```
Deleting 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.

Using the from clause with delete  

The optional from immediately after the delete keyword is included for compatibility with other versions of SQL. The from clause in the second position of a delete statement is a Transact-SQL that 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.
CHAPTER 7   Adding, Changing, Transferring, and Deleting Data

This is the correct statement:

```
drop 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.

Deleting 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:

```
drop 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. See “Renumber the table’s IDENTITY columns with bcp” on page 246.

Deleting 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. See “General rules for creating referential integrity constraints” on page 314.

**truncatetable syntax**

The syntax of `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
```

For information about the `PARTITION` clause, see Chapter 10, “Partitioning Tables and Indexes.”

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. See Chapter 20, “Triggers: Enforcing Referential Integrity.”
CHAPTER 8

Creating 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.

Adaptive Server database objects include:

- Tables
- Rules
- Defaults

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Databases and tables</td>
<td>279</td>
</tr>
<tr>
<td>Using and creating databases</td>
<td>282</td>
</tr>
<tr>
<td>Altering the sizes of databases</td>
<td>287</td>
</tr>
<tr>
<td>Dropping databases</td>
<td>288</td>
</tr>
<tr>
<td>Creating tables</td>
<td>288</td>
</tr>
<tr>
<td>Managing identity gaps in tables</td>
<td>301</td>
</tr>
<tr>
<td>Defining integrity constraints for tables</td>
<td>307</td>
</tr>
<tr>
<td>Designing and creating a table</td>
<td>318</td>
</tr>
<tr>
<td>Creating new tables from query results: select into</td>
<td>321</td>
</tr>
<tr>
<td>Altering existing tables</td>
<td>327</td>
</tr>
<tr>
<td>Dropping tables</td>
<td>345</td>
</tr>
<tr>
<td>Computed columns</td>
<td>347</td>
</tr>
<tr>
<td>Assigning permissions to users</td>
<td>356</td>
</tr>
<tr>
<td>Getting information about databases and tables</td>
<td>356</td>
</tr>
</tbody>
</table>
Databases and tables

- 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 and are discussed in this chapter. Indexes describe how data is organized in tables. They are not considered database objects by Adaptive Server and are not listed in **sysobjects**. Indexes are discussed in Chapter 13, “Creating Indexes on Tables.”

**Enforcing 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.

Consistency of data values in the database is another example of data integrity, which is described in Chapter 23, “Transactions: Maintaining Data Consistency and Recovery.”

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. These integrity constraints are described later in this chapter.

Permissions within databases

Whether or not you can create and drop databases and database objects depends on your permissions or privileges. 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 Adaptive Server 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 Adaptive Server, that is, anyone who is listed in master..syslogins, has access to pubs2 and pubs3, 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. Adaptive Server 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, Adaptive Server admits you as yourself, but may give you different permissions from those of “guest.”

**Note** All the examples in this chapter assume you are being treated as “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.


### Using and creating databases

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

When you install Adaptive Server, it contains these **system databases**:

- **master** – controls the user databases and the operation of Adaptive Server 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 Adaptive Server 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 Adaptive Server 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. Adaptive Server manages each database by means of system tables. The data dictionary tables in the master database and in other databases are considered system tables.

Choosing a database: use

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 Adaptive Server, so to use another database, issue the use command. You or a system administrator can change the database to which you initially connect by using sp_modifylogin. Only a system administrator can change the default database for another user.

Creating a user database with create 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 Adaptive Server, and must follow the rules for identifiers described under “Identifiers” on page 11. Adaptive Server 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.

Adaptive Server 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.


This chapter describes all the create database options except with override. For information about with override, see Chapter 6, “Creating and Managing User Databases,” the System Administration Guide: Volume 2.
The on clause

The optional on clause allows you to specify where to store the database and how much space, in megabytes, to allocate for it. 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

```
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.

The log on clause

Unless you are creating very small, noncritical databases, always use the log on database_device extension to create database. This 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.
Using and creating databases

- 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 pubsd = 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 Chapter 8, “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.

**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 Chapter 12, “Backing Up and Restoring User Databases,” in the System Administration Guide: Volume 2.

**quiesce database command**

You can put a database in sleep mode by using:

```
quiesce database
```

Altering the sizes of databases

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.

For information about `with override`, see Chapter 6, “Creating and Managing User Databases,” in the `System Administration Guide: Volume 2`. 
Dropping databases

Use the `drop database` command to remove a database. `drop database` deletes the database and all of its contents from Adaptive Server, 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.

Creating 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. 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.

Maximum number of columns per table

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. See the *Reference Manual: Commands*. 
Example

Use the newpubs database you created in the previous section to try these examples. Otherwise, these changes will affect another database, like pubs2 or pubs3.

The simplest form of create table is:

```sql
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:

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

If you have set quoted_identifier on, both the table name and the column names can be delimited identifiers. Otherwise, they must follow the rules for identifiers described under “Identifiers” on page 11. 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. Datatypes are discussed in Chapter 6, “Using and Creating Datatypes.”

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.
Choosing table names

The create table command builds the new table in the currently open database. 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..”. For more information, see “Using temporary tables” on page 297.

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.

Creating 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:

```sql
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.

create table syntax

The create table statement:

- Defines each column in the table.
- 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:

```sql
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.

The following sections describe components of table definition: system-supplied datatypes, user-defined datatypes, null types, and IDENTITY columns.

**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.

### Using IDENTITY columns

An IDENTITY column contains a value for each row, generated automatically by Adaptive Server, 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.
Creating tables

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:

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

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

```sql
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, Adaptive Server 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. Adaptive Server provides several methods of controlling identity gaps described in “Managing identity gaps in tables” on page 301.

Creating 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:

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

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

```sql
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.

**Referencing 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:

```sql
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:

```sql
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:

```sql
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.

See “Retrieving IDENTITY column values with @@identity” on page 244. For information on transactions, see Chapter 23, “Transactions: Maintaining Data Consistency and Recovery.”

Referring 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
```

Automatically creating “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, Adaptive Server 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
```
Allowing null values in a column

If you omit null or not null in the create table statement, Adaptive Server 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, Adaptive Server displays an error message. See “Unknown” values: NULL” on page 66.

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.

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
```

See “Defining integrity constraints for tables” on page 307 for more information on constraints. Rules are covered in detail in Chapter 14, “Defining Defaults and Rules for Data.”
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`. The default constraint is described later in this chapter; `create default` is described in Chapter 14, “Defining Defaults and Rules for Data.”

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.

Table 8-1 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.

<table>
<thead>
<tr>
<th>Column definition</th>
<th>User entry</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null and default defined</td>
<td>Enters no value</td>
<td>Default used</td>
</tr>
<tr>
<td></td>
<td>Enters NULL value</td>
<td>NULL used</td>
</tr>
<tr>
<td>Null defined, no default defined</td>
<td>Enters no value</td>
<td>NULL used</td>
</tr>
<tr>
<td></td>
<td>Enters NULL value</td>
<td>NULL used</td>
</tr>
<tr>
<td>Not null, default defined</td>
<td>Enters no value</td>
<td>Default used</td>
</tr>
<tr>
<td></td>
<td>Enters NULL value</td>
<td>NULL used</td>
</tr>
<tr>
<td>Not null, no default defined</td>
<td>Enters no value</td>
<td>Error</td>
</tr>
<tr>
<td></td>
<td>Enters NULL value</td>
<td>Error</td>
</tr>
</tbody>
</table>

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, Adaptive Server converts it to the corresponding variable-length datatype. Adaptive Server does not inform you of the type change.

Table 8-2 lists the fixed-length datatypes and the variable-length datatypes to which Adaptive Server converts them. Certain variable-length datatypes, such as `moneyn`, are reserved; you cannot use them to create columns, variables, or parameters.
Table 8-2: Conversion of fixed-length to variable-length datatypes

<table>
<thead>
<tr>
<th>Original fixed-length datatype</th>
<th>Converted to</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>varchar</td>
</tr>
<tr>
<td>nchar</td>
<td>nvarchar</td>
</tr>
<tr>
<td>unichar</td>
<td>univarchar</td>
</tr>
<tr>
<td>binary</td>
<td>varbinary</td>
</tr>
<tr>
<td>datetime</td>
<td>datetimn</td>
</tr>
<tr>
<td>float</td>
<td>floatn</td>
</tr>
<tr>
<td>bigint, int, smallint, tinyint,</td>
<td>intn</td>
</tr>
<tr>
<td>decimal</td>
<td>decimaln</td>
</tr>
<tr>
<td>numeric</td>
<td>numericn</td>
</tr>
<tr>
<td>money and smallmoney</td>
<td>moneyn</td>
</tr>
</tbody>
</table>

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.

**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 `Writeltext`.

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 `Writeltext`. See the *Reference Manual: Commands*.

**Using temporary tables**

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

There are two kinds of temporary tables:

* Tables that can be shared among Adaptive Server sessions
Creating tables

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 Adaptive Server sessions:

```sql
CREATE TABLE tempdb..authors
(
    au_id char(11)
)

DROP TABLE tempdb..authors
```

Adaptive Server does not change the names of temporary tables created this way. The table exists until the current session ends or until its owner drops it using `DROP TABLE`.

- Tables that are accessible only by the current Adaptive Server session or procedure

Create a nonshareable temporary table by specifying a pound sign (#) before the table name in the `CREATE TABLE` statement. For example:

```sql
CREATE TABLE #authors
(
    au_id char(11)
)
```

The table exists until the current session or procedure ends, or until its owner drops it using `DROP TABLE`.

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:

```sql
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`. 

298

Adaptive Server Enterprise
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 Adaptive Server so `model` is copied to tempdb.

Ensuring that the temporary table name is unique

To ensure that a temporary table name is unique for the current session, Adaptive Server:

- Truncates the table name to 238 bytes, including the pound sign (#), if necessary
- Appends a 17-digit numeric suffix that is unique for an Adaptive Server session

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

```sql
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)
```

Manipulating temporary tables in stored procedures

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

Temporary tables with names beginning with “#”

Temporary tables with names beginning with “#” that are created within stored procedures disappear when the procedure exits. A single procedure can:

- Create a temporary table
- Insert data into the table
Creating tables

- Run queries on the table
- Call other procedures that reference the table

Since the temporary table must exist in order to create procedures that reference it, here are the steps to follow:

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 tables with names beginning with `tempdb..`

You can 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 steps above to create these tables.

**Warning!** 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.

General rules for temporary tables

Temporary tables with names that begin with `#` are subject to the following restrictions:

- You cannot create views on these tables.
- You cannot associate triggers with these tables.
- From within a stored procedure, you cannot:
  a. Create a temporary table
  b. Drop it
  c. 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 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 Adaptive Server was restarted.

- You do not have to set the select into/bulkcopy option on to select into a temporary table.

Managing identity gaps in tables

The IDENTITY column contains a unique ID number, generated by Adaptive Server, 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, Adaptive Server 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.

**Parameters for controlling identity gaps**

Adaptive Server provides parameters that allow you to control gaps in identity numbers as described in Table 8-3.

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Scope</th>
<th>Used with</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>identity_gap</td>
<td>Table-specific</td>
<td>create table or select into</td>
<td>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.</td>
</tr>
<tr>
<td>identity burning set factor</td>
<td>Server-wide</td>
<td>sp_configure</td>
<td>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 (10⁷) 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 10⁷ (or 1,500,000): sp_configure &quot;identity burning set factor&quot;, 1500000</td>
</tr>
<tr>
<td>identity grab size</td>
<td>Server-wide</td>
<td>sp_configure</td>
<td>Reserves a block of contiguous ID numbers for each process. Works with identity numbers for each process. Works with identity burning set factor and identity_gap.</td>
</tr>
</tbody>
</table>
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 Adaptive Server 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 Adaptive Server 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, Adaptive Server 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 Adaptive Server 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.

Reducing the identity number gap

Creating the `books` table (from the example above) 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, Adaptive Server takes the next 1000 numbers, starting with 1001, and stores the number 2000 as the highest number.

If Adaptive Server fails after row number 1002, it uses the numbers 1000 through 1002: numbers 1003 through 2000 are lost. When you restart Adaptive Server, 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.
Managing identity gaps in tables

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.

**Setting 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:

```sql
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:

```sql
select IdNum into newtable
with identity_gap = 20
from mytable
```

**Changing the table-specific identity gap**

To change the identity gap for a specific table, use `sp_chgattribute`:

```sql
sp_chgattribute "table_name", "identity_gap", set_number
```
For example:

```sql
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:

```sql
sp_chgattribute "mytable", "identity_gap", 0
```

See the *Reference Manual: Procedures*.

### Displaying 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.

```sql
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.

```sql
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
```
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, Adaptive Server 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, Adaptive Server discards the value 56 and uses a value of 57 for the next insertion.

**When table inserts reach 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. See “Reaching the IDENTITY column’s maximum value” on page 245.

**Defining integrity constraints for tables**

Transact-SQL provides two methods of maintaining data integrity in a database:

- Defining rules, defaults, indexes, and triggers
- Defining create table integrity constraints

Choosing one method over the other 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). See “Allowing null values in a column” on page 295 for information about the null and not null keywords.

For information about finding any constraints defined for a table, see “Using `sp_helpconstraint` to find a table’s constraint information” on page 360.

**Warning!** Do not define or alter the definitions of constraints for system tables.

### Specifying table-level or 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. Adaptive Server 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`:

```sql
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. The check constraint is described in “Specifying check constraints” on page 315.

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.

Creating error messages for constraints

You can create error messages with sp_addmessage, and bind them to constraints with sp_bindmsg.

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:
Defining integrity constraints for tables

```
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`.

After creating a check constraint

When you create a check constraint, **source text**, which describes the check constraint, is stored in the text column of the `syscomments` system table.

**Warning!** Do not remove this information from `syscomments`; doing so can cause problems for future upgrades of Adaptive Server.

If you have security concerns, encrypt the text in `syscomments` by using `sp_hidetext`, described in the *Reference Manual: Procedures*. See “Compiled objects” on page 4.

Specifying default column values

Before you define any column-level integrity constraints, you can specify a default value for the column. The **default clause** assigns a default value to a column as part of the `create table` statement. When a user does not enter a value for the column, Adaptive Server inserts the default value.

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 Adaptive Server insert the user name as the default. The datatype of the column must be either `char(30)` or `varchar(30)` to use this default.

- **null** – specifies that Adaptive Server insert the null value 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:

```sql
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.

### Specifying 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, Adaptive Server 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.
Defining integrity constraints for tables

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:

```sql
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. See Chapter 13, “Creating Indexes on Tables.”

Specifying 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 following two tables from the pubs3 database illustrate how declarative referential integrity works. The first table, stores, is a “referenced” table:

```sql
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:
CHAPTER 8 Creating Databases and Tables

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.

Table-level or column-level referential integrity constraints

You can define referential integrity constraints at the column level or the table level. The referential integrity constraints in the preceding examples were defined at the column level, using the references keyword in the create table statement.

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. See “Specifying table-level or column-level constraints” on page 308.
Defining integrity constraints for tables

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.

**Maximum number of references allowed for a table**

The maximum number of references allowed for a table is 192. See “Using `sp_helpconstraint` to find a table’s constraint information” on page 360.

**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))
```

**General rules for creating referential integrity constraints**

When you define referential integrity constraints in a table:

- Make sure you have references permission on the referenced table. See Chapter 17, “Managing User Permissions,” in the *System Administration Guide: Volume 1.*
Creating Databases and Tables

CHAPTER 8

Specifying 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

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:

```sql
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. See Chapter 20, “Triggers: Enforcing Referential Integrity.”

Specifying 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

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:

```sql
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. See Chapter 20, “Triggers: Enforcing Referential Integrity.”
Defining integrity constraints for tables

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:

```sql
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:

```sql
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:

```sql
col_name != null
```

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 Adaptive Server 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:

```sql
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 Adaptive Server 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.
Designing and creating a table

This section shows a create table statement you can use to create a practice table of your own. If you do not have create table permission, see a system administrator or the owner of the database in which you are working.

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.

First, plan the table's design:

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. See Chapter 13, "Creating Indexes on Tables."

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. See Chapter 14, "Defining Defaults and Rules for Data."
3. Bind any defaults and rules using sp_binddefault and sp_bindrule. Any existing defaults or rules on a user-defined datatype already used in a create table statement, are automatically used. See Chapter 17, "Using Stored Procedures."


Making a design sketch

The table called `friends_etc` is used in this chapter and subsequent chapters to illustrate how to create indexes, defaults, rules, triggers, and so forth. It can hold names, addresses, telephone numbers, and personal information about your friends. 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.

Table 8-4 shows the proposed structure of the `friends_etc` table and the indexes, defaults, and rules that go with each column.

<table>
<thead>
<tr>
<th>Column</th>
<th>Datatype</th>
<th>Null?</th>
<th>Index</th>
<th>Default</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pname</code></td>
<td>nm</td>
<td>NOT NULL</td>
<td><code>nmind(composite)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>sname</code></td>
<td>nm</td>
<td>NOT NULL</td>
<td><code>nmind(composite)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>address</code></td>
<td>varchar(30)</td>
<td>NULL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>city</code></td>
<td>varchar(30)</td>
<td>NOT NULL</td>
<td></td>
<td><code>citydflt</code></td>
<td></td>
</tr>
<tr>
<td><code>state</code></td>
<td>char(2)</td>
<td>NOT NULL</td>
<td></td>
<td><code>statedflt</code></td>
<td></td>
</tr>
<tr>
<td><code>zip</code></td>
<td>char(5)</td>
<td>NULL</td>
<td><code>zipind</code></td>
<td><code>zipdflt</code></td>
<td><code>ziprule</code></td>
</tr>
<tr>
<td><code>phone</code></td>
<td>p#</td>
<td>NULL</td>
<td></td>
<td><code>phonerule</code></td>
<td></td>
</tr>
<tr>
<td><code>age</code></td>
<td>tinyint</td>
<td>NULL</td>
<td></td>
<td><code>agerule</code></td>
<td></td>
</tr>
<tr>
<td><code>bday</code></td>
<td>datetime</td>
<td>NOT NULL</td>
<td></td>
<td><code>bdflt</code></td>
<td></td>
</tr>
<tr>
<td><code>gender</code></td>
<td>bit</td>
<td>NOT NULL</td>
<td></td>
<td><code>gndrdflt</code></td>
<td></td>
</tr>
<tr>
<td><code>debt</code></td>
<td>money</td>
<td>NOT NULL</td>
<td></td>
<td><code>gndrdflt</code></td>
<td></td>
</tr>
<tr>
<td><code>notes</code></td>
<td>varchar(255)</td>
<td>NULL</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Creating the user-defined datatypes

The first two columns are for the personal (first) name and surname. They are defined as the nm datatype. Before you create the table, create the datatype. The same is true of the p# datatype for the phone column:

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

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.

Choosing the 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. This 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. The relationship between defaults and rules is discussed in Chapter 14, “Defining Defaults and Rules for Data.”

Defining the table

Write the create table statement:

```sql
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,
```

Adaptive Server Enterprise
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.

Creating 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.

<table>
<thead>
<tr>
<th>name</th>
<th>db_size</th>
<th>owner</th>
<th>dbid</th>
<th>created</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>pubs2</td>
<td>20.0 MB</td>
<td>sa</td>
<td>4</td>
<td>Apr 25, 2005</td>
<td>select into/bulkcopy/pllsort, trunc log on chkpt, mixed log and data</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>device_fragments</th>
<th>size</th>
<th>usage</th>
<th>created</th>
<th>free kbytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>master</td>
<td>10.0MB</td>
<td>data and log</td>
<td>Apr 13 2005</td>
<td>1792</td>
</tr>
<tr>
<td>pubs_2_dev</td>
<td>10.0MB</td>
<td>data and log</td>
<td>Apr 13 2005</td>
<td>9888</td>
</tr>
<tr>
<td>device</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Creating new tables from query results: select into

---

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>master</td>
<td>default</td>
</tr>
<tr>
<td>master</td>
<td>logsegment</td>
</tr>
<tr>
<td>master</td>
<td>system</td>
</tr>
<tr>
<td>pubs_2_dev</td>
<td>default</td>
</tr>
<tr>
<td>pubs_2_dev</td>
<td>logsegment</td>
</tr>
<tr>
<td>pubs_2_dev</td>
<td>system</td>
</tr>
<tr>
<td>pubs_2_dev</td>
<td>seg1</td>
</tr>
<tr>
<td>pubs_2_dev</td>
<td>seg2</td>
</tr>
</tbody>
</table>

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. See Chapter 12, "Views: Limiting Access to Data."

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.

```sql
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:

```sql
select *
from newtable
```

<table>
<thead>
<tr>
<th>pub_id</th>
<th>pub_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0736</td>
<td>New Age Books</td>
</tr>
<tr>
<td>0877</td>
<td>Binnet &amp; Hardley</td>
</tr>
<tr>
<td>1389</td>
<td>Algodata Infosystems</td>
</tr>
</tbody>
</table>

`newtable` 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:

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

(0 rows affected)

```sql
select *
from newtable2
```

<table>
<thead>
<tr>
<th>pub_id</th>
<th>pub_name</th>
<th>city</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>-------</td>
<td>--------------</td>
<td>------</td>
<td>-------</td>
</tr>
</tbody>
</table>

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:

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

(6 rows affected)

```sql
select * from #whatspent
```

<table>
<thead>
<tr>
<th>type</th>
<th>Total_amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNDECIDED</td>
<td>NULL</td>
</tr>
<tr>
<td>business</td>
<td>25,125.00</td>
</tr>
<tr>
<td>mod_cook</td>
<td>15,000.00</td>
</tr>
<tr>
<td>popular_comp</td>
<td>15,000.00</td>
</tr>
<tr>
<td>psychology</td>
<td>21,275.00</td>
</tr>
</tbody>
</table>
Creating new tables from query results: select into

```
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:

```sql
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.

**Checking for errors**

`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 a `select into` statement fails after creating a new table, Adaptive Server does not automatically drop the table or deallocate its first data page. 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.

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

This section describes special rules for using the `select into` command with tables containing IDENTITY columns.

Selecting 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`:

```
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.

Selecting 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"
```
Creating new tables from query results: select into

Adding 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.

Defining a column for which the value must be computed

IDENTITY column values are generated by Adaptive Server. 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.
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.


Altering 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. See Chapter 10, “Partitioning Tables and Indexes,” and the Reference Manual: Commands.

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:

```sql
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. See “Data copying” on page 338.

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** Adaptive Server 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, Adaptive Server 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.

**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.
Using `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 Adaptive Server.


Adding columns

The following adds 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
```

Adding 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, Table 8-5 lists the default column IDs of the `salesdetail` table:

<table>
<thead>
<tr>
<th>Column name</th>
<th>Col ID</th>
<th>stor_id</th>
<th>ord_num</th>
<th>title_id</th>
<th>qty</th>
<th>discount</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>stor_id</td>
<td>ord_num</td>
<td>title_id</td>
<td>qty</td>
<td>discount</td>
<td></td>
</tr>
<tr>
<td>Column name</td>
<td>stor_id</td>
<td>ord_num</td>
<td>title_id</td>
<td>qty</td>
<td>discount</td>
<td></td>
</tr>
<tr>
<td>Col ID</td>
<td>stor_id</td>
<td>ord_num</td>
<td>title_id</td>
<td>qty</td>
<td>discount</td>
<td></td>
</tr>
<tr>
<td>stor_id</td>
<td>ord_num</td>
<td>title_id</td>
<td>qty</td>
<td>discount</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stor_id</td>
<td>ord_num</td>
<td>title_id</td>
<td>qty</td>
<td>discount</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stor_id</td>
<td>ord_num</td>
<td>title_id</td>
<td>qty</td>
<td>discount</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stor_id</td>
<td>ord_num</td>
<td>title_id</td>
<td>qty</td>
<td>discount</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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
```
**Altering existing tables**

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.

**Adding 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.

Adaptive Server 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:”

```sql
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).

**Adding 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:

```sql
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, Adaptive Server 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.

For information about dropping a constraint, see “Dropping constraints” on page 332.

### Dropping 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:

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

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

### Dropping columns renames the column ID

`alter table` renames 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:

<table>
<thead>
<tr>
<th>Column name</th>
<th>au_id</th>
<th>title_id</th>
<th>au_ord</th>
<th>royaltyper</th>
</tr>
</thead>
<tbody>
<tr>
<td>Column ID</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

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

```sql
alter table titleauthor drop au_ord
```

`titleauthor` now has these column names and column IDs:
### Altering existing tables

#### Table 8-7: Column IDs after dropping au_ord

<table>
<thead>
<tr>
<th>Column name</th>
<th>au_id</th>
<th>title_id</th>
<th>royaltyper</th>
</tr>
</thead>
<tbody>
<tr>
<td>Column ID</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

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.

### Dropping constraints

Use `alter table` to drop a constraint; for example:

```sql
alter table titles
drop constraint advance_chk
```

See “Using `sp_helpconstraint` to find a table’s constraint information” on page 360.

### Modifying 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:

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

**Warning!** 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.
Converting 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 Chapter 1, “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, Adaptive Server raises an error message and the operation is aborted.

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, Adaptive Server issues a message similar to this:

Warning: ALTER TABLE operation did not affect column ‘au_lname’.
Msg 13905, Level 16, State 1:
Server ‘SYBASE1’, Line 1:
Warning: no columns to drop, add or modify. ALTER TABLE ‘authors’ was aborted.

Modifying tables may prevent successful bulk copy of previous dump

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.

Decreasing 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:

```sql
select title from titles

<table>
<thead>
<tr>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>Bu</td>
</tr>
<tr>
<td>Co</td>
</tr>
<tr>
<td>Co</td>
</tr>
<tr>
<td>Em</td>
</tr>
</tbody>
</table>
```
Alter existing tables

Adaptive Server 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.

Modifying 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, Adaptive Server does not support modifying a column from smalldatetime to char datatype. See the Reference Manual: Commands.

Modifying 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:

```sql
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.

Modifying 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)), Adaptive Server 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 arithignore numeric_truncation is on or off.

If arithignore arith_overflow is on and alter table causes a numeric overflow,
Adaptive Server issues a warning. However, if arithabort arithignore
arith_overflow is off, Adaptive Server does not issue a warning if alter table
causes a numeric overflow. By default, arithignore arith_overflow is off when
you install Adaptive Server.

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.

Modifying text, unitext, and image columns

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, Adaptive Server truncates the data to fit the column length.

Adding, dropping, and modifying IDENTITY columns

This section describes adding, dropping, and modifying IDENTITY columns using alter table. For a general discussion of IDENTITY columns, see “Using IDENTITY columns” on page 291.

Adding 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, Adaptive Server 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.

Dropping IDENTITY columns

You can drop IDENTITY columns just like any other column, for example:

```sql
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. See “Referring to IDENTITY columns with `syb_identity`” on page 294.

- 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:

  ```sql
  set identity_insert table_name off
  ```

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

  ```sql
  set identity_insert table_name on
  ```

Modifying 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 was used up because of a server shutdown.
Altering existing tables

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.

Data copying

Adaptive Server 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, Adaptive Server 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.

Adaptive Server 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, Adaptive Server 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). Adaptive Server performs data copying in case one row has a NOT NULL value for this column.
CHAPTER 8  Creating Databases and Tables

- 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 Adaptive Server 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.

### Changing `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.

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. Adaptive Server 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`.
Modifying locking schemes and table schema

If `alter table` performs a data copy, you can also include a command to change the locking scheme of a table. 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:

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

However, you cannot use `alter table` to change table schema and the locking scheme of a table that has a clustered index. If a table has a clustered index you can:

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.

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.

Altering columns with user-defined datatypes

You can use `alter table` to add, drop, or modify columns that use user-defined datatypes.

Adding a column with user-defined datatypes

Use the same syntax to add a column with a user-defined datatype as with a system-defined datatype. For example, to add a column to the authors table of pubs2 using the `usertype` datatype:

```sql
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.

**Dropping 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.

**Modifying a column with user-defined datatypes**

The syntax to modify a column to include user-defined datatypes is the same as modifying a column to include system-defined datatypes. For example, to modify the `au_lname` of the `authors` table to use the user-defined `newtype` datatype:

```sql
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).

**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). You must fix the errors or warnings that refer to schema objects that depend on the affected column before you reissue the command. 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 Adaptive Server actually performs the work.

alter table detects and reports certain errors when actually running the command (for example, if you are dropping a column, the presence of a referential constraint). 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.

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.

Scripts generated by if exists()...alter table

Scripts that include constructs like the following may produce errors if the table described in the script does not include the specified column:

```sql
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.

Adaptive Server 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, Adaptive Server does not raise these errors.

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

```sql
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")
```
Alter existing tables

Because the execute immediate statement is run only if the if exists() function succeeds, Adaptive Server 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.

Renaming 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"
Renaming 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 Adaptive Server. See the Utility Guide.

Dropping 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 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.

Computed columns

Computed columns, computed column indexes, and function-based indexes provide easier 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.
Computed columns

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. See “Deterministic property” on page 351.
- You can create a clustered index on a computed column, but not a clustered function-based index.

For more information on function-based indexes, see “Indexing with function-based indexes” on page 448.

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.

Using computed columns

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:

```sql
create table rental
    (cust_id int, start_date as getdate() materialized, prod_id int)
create index ind_start_date on rental (start_date)
```
Computed columns

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:

```sql
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:

```sql
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:

```sql
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:

```sql
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:

```sql
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()`:

```sql
alter table parts_table add name_in_myorder compute
    Xform_to_myorder(name) materialized
```

To return the result in the customized format:

```sql
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):

```sql
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.

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.

**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.
**Computed columns**

- `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.

```sql
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
)
```

**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:

```sql
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)
```

Adaptive Server evaluates the computed columns and uses the results to build or update indexes when you create or update an index.
Deterministic property

All expressions and functions are deterministic or nondeterministic:

- Deterministic expressions and functions always return the same result, if they are evaluated with the same set of input values. This expression is deterministic:
  \[ c_1 \times c_2 \]

- 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.

How does the deterministic property affect computed columns?

There are two types of computed columns in Adaptive Server:

- Virtual computed columns
- Materialized computed columns

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.
Computed columns

- 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.

Examples

The following examples illustrate both the usefulness and the dangers of using nondeterministic computed columns and index keys. They are intended only for illustrative purposes.

Example 1

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:

```sql
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.

How the deterministic property affects virtual computed columns

Adaptive Server 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:

```sql
select Cust_ID, Property_ID from Renting
    where Formatted_Name = 'RICHARD HUANG'
```

Adaptive Server 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:

```sql
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.

How the deterministic property affects materialized computed columns

Adaptive Server 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, Adaptive Server 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 Example 1, 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.

Example 2

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 Adaptive Server 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 Adaptive Server to reevaluate Property_Name and update the value stored in the data row. Each time Adaptive Server 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.

How does the deterministic property affect 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 (for more information on function-based indexes, see “Indexing with function-based indexes” on page 448).

- Adaptive Server 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.
Assigning permissions to users

- Adaptive Server 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.

Assigning permissions to users

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 the Reference Manual: Commands and Chapter 17, “Managing User Permissions,” in the System Administration Guide: Volume 1.

Getting information about databases and tables

Adaptive Server includes several procedures and functions you can use to get information about databases, tables, and other database objects. This section describes some of them. See the Reference Manual: Procedures and also the Reference Manual: Building Blocks for information about the procedures and functions described here and about others that provide additional information.

Getting help on databases

sp_helpdb can report information about a specified database or about all Adaptive Server databases.

    sp_helpdb [dbname]

This example displays a report on pubs2 on a server using a page size of 8K.

<table>
<thead>
<tr>
<th>name</th>
<th>db_size</th>
<th>owner</th>
<th>dbid</th>
<th>created</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>pubs2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CHAPTER 8  Creating Databases and Tables

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
Getting help on database objects

Adaptive Server provides system procedures, catalog stored procedures, and built-in functions that return helpful information about database objects, such as tables, columns, and constraints.

Using sp_help 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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Owner</th>
<th>Object_type</th>
<th>Create_date</th>
</tr>
</thead>
<tbody>
<tr>
<td>publishers</td>
<td>dbo</td>
<td>user table</td>
<td>Nov 9 2004 9:57AM</td>
</tr>
</tbody>
</table>

(1 row affected)

<table>
<thead>
<tr>
<th>Column_name</th>
<th>Type</th>
<th>Length</th>
<th>Prec</th>
<th>Scale</th>
<th>Nulls</th>
<th>Default_name</th>
<th>Rule_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>pub_id</td>
<td>char</td>
<td>4</td>
<td>NULL</td>
<td>NULL</td>
<td>0</td>
<td>NULL</td>
<td>pub_idrule</td>
</tr>
<tr>
<td>pub_name</td>
<td>varchar</td>
<td>40</td>
<td>NULL</td>
<td>NULL</td>
<td>1</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>city</td>
<td>varchar</td>
<td>20</td>
<td>NULL</td>
<td>NULL</td>
<td>1</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>state</td>
<td>char</td>
<td>2</td>
<td>NULL</td>
<td>NULL</td>
<td>1</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Access_Rule_name: Computed_Column_object: Identity

<table>
<thead>
<tr>
<th>Access_Rule_name</th>
<th>Computed_Column_object</th>
<th>Identity</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>0</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>0</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>0</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>0</td>
</tr>
</tbody>
</table>

Object has the following indexes

<table>
<thead>
<tr>
<th>index_name</th>
<th>index_keys</th>
<th>index_description</th>
<th>index_max_rows_per_page</th>
</tr>
</thead>
<tbody>
<tr>
<td>pubind</td>
<td>pub_id</td>
<td>clustered, unique</td>
<td>0</td>
</tr>
</tbody>
</table>

(1 row affected)

<table>
<thead>
<tr>
<th>index_fill_factor</th>
<th>index_reservepagegap</th>
<th>index_created</th>
<th>index_local</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Nov 9 2004 9:58AM</td>
<td>Global Index</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>index_ptn_name</th>
<th>index_ptn_segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>pubind_416001482</td>
<td>default</td>
</tr>
</tbody>
</table>
(1 row affected)

<table>
<thead>
<tr>
<th>keytype</th>
<th>object</th>
<th>related_object</th>
<th>related_keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>primary</td>
<td>publishers</td>
<td>-- none --</td>
<td>pub_id, *, *, *, *, *</td>
</tr>
<tr>
<td>foreign</td>
<td>titles</td>
<td>publishers</td>
<td>pub_id, *, *, *, *, *</td>
</tr>
</tbody>
</table>

(1 row affected)

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>partition_type</th>
<th>partitions</th>
<th>partition_keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>publishers</td>
<td>base table</td>
<td>roundrobin</td>
<td>1</td>
<td>NULL</td>
</tr>
</tbody>
</table>

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(min/avg)
-----------|-----------|-----------|-----------------
1 | 1 | 1 | 1.000000 | 1.000

No defined keys for this object.

name | type | partition_type | partitions | partition_keys
-----|------|----------------|------------|----------------|
mytable | base table | roundrobin | 1 | NULL |

partition_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)
-----------|-----------|-----------|-----------------
1 | 1 | 1 | 1.000000

Lock scheme Allpages

The attribute ‘exp_row_size’ is not applicable to tables with
Getting information about databases and tables

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 0
(1 row affected)

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.

Using sp_helpconstraint to find a table’s constraint information

sp_helpconstraint reports information about the declarative referential integrity
constraints 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
```

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>Num_referential_constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>80003316</td>
<td>titles</td>
<td>4</td>
</tr>
<tr>
<td>16003088</td>
<td>authors</td>
<td>3</td>
</tr>
<tr>
<td>176003658</td>
<td>stores</td>
<td>3</td>
</tr>
<tr>
<td>256003943</td>
<td>salesdetail</td>
<td>3</td>
</tr>
<tr>
<td>208003772</td>
<td>sales</td>
<td>2</td>
</tr>
<tr>
<td>336004228</td>
<td>titleauthor</td>
<td>2</td>
</tr>
<tr>
<td>896006223</td>
<td>store_employees</td>
<td>2</td>
</tr>
<tr>
<td>48003202</td>
<td>publishers</td>
<td>1</td>
</tr>
<tr>
<td>128003487</td>
<td>roysched</td>
<td>1</td>
</tr>
<tr>
<td>400004456</td>
<td>discounts</td>
<td>1</td>
</tr>
<tr>
<td>448004627</td>
<td>au_pix</td>
<td>1</td>
</tr>
<tr>
<td>496004798</td>
<td>blurbs</td>
<td>1</td>
</tr>
</tbody>
</table>

(11 rows affected)

This report shows that the titles table has the largest number of referential constraints in the pubs3 database.
Finding out how much space a table uses

To find out how much space a table uses, enter:

```
sp_spaceused [objname]
```

`sp_spaceused` computes and displays the number of rows and data pages used by a table or a clustered or nonclustered index.

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.

Listing 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
-------------------
-------------------
-------------------
-------------------
-------------------
-------------------
-------------------
```

Adaptive Server Enterprise
Finding an object name and ID

To identify the ID and name of an object, use `object_id()` and `object_name()`:

```sql
select object_id("titles")
----------
208003772
```

Object names and IDs are stored in the `sysobjects` system table.
Getting information about databases and 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.

Do not confuse SQL-derived tables with 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:

```sql
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:

```plaintext
city
--------
Boston
Washington
Berkeley
```
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:

```sql
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:

```sql
select * from vw_colorado_titles
```

Drop the view when it is no longer needed:

```sql
drop view vw_colorado_titles
```

If the query results are only needed once, you might instead use a SQL-derived table:

```sql
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.
SQL-derived tables and optimization

Queries expressed as a single SQL statement make more efficient use of the optimizer than queries 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:

```sql
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
```

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:

```sql
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) ]
```
**SQL-derived table syntax**

```
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 
```


**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.
Correlated SQL-derived tables

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`:

```sql
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:

```sql
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:

```sql
select * from publishers
where pub_id in (select pub_id from
(select pub_id from titles)
dt_publishers
where pub_id = publishers.pub_id)
```

Using SQL-derived tables

You can use SQL-derived tables to form part of a larger integrated query that uses assorted SQL clauses and operators.
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.

```sql
select postalcode
from (select postalcode
      from (select postalcode
            from authors) dt_1) dt_2
```

The degree of nesting is limited to 25.

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:

```sql
select pub_name from publishers
where "business" in
  (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. See Chapter 5, “Subqueries: Using Queries Within Other Queries.”

Unions

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:

```sql
select * from
  (select stor_id, ord_num from sales
   union
   select stor_id, ord_num from sales_east)
dt_sales_info
```
Here, the union of two select operations defines the SQL-derived table `dt_sales_info`.

**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:

```sql
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)
```

**Renaming 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, as in the following example:

```sql
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.

**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:
You can specify column names for the target list of a derived-table expression using a derived column list:

```sql
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:

```sql
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.

**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.

```sql
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.

**Joins with SQL-derived tables**

The following example illustrates a join between a SQL-derived table and an existing table. The join is specified by the where clause. The two tables joined are dt_c, which is a SQL-derived table, and publishers, which is an existing table in the pubs2 database.

```sql
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.

```sql
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. Sybase supports both left and right outer joins. The following example illustrates a left outer join between two SQL-derived tables.
Using 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
```

Creating a table from a SQL-derived table

Data obtained from a SQL-derived table can be inserted into a new table:

```
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`.

Using views with SQL-derived tables

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 INTO view_colo_publishers
VALUES ('1799', 'Gigantico Publications', 'Denver',
```

Adaptive Server Enterprise
'CO')

You can also update existing data through a view that uses a SQL-derived table:

```sql
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:

```sql
drop view view_colo_publishers
```

### 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:

```sql
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`. 
Using SQL-derived tables
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.

Adaptive Server 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 Chapter 5, “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, Adaptive Server 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.

---

**Upgrading from Adaptive Server 12.5.x and earlier**

When you upgrade from version 12.5.x and earlier, Adaptive Server 15.0 and later changes all database tables into unpartitioned tables. Indexes do not change; they remain global and unpartitioned.

To repartition database tables or partition indexes, follow the instructions in this chapter.
CHAPTER 10  Partitioning Tables and Indexes

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, Sybase 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. In versions earlier than Adaptive Server 15.0, all indexes were global. With Adaptive Server 15.0 and later, 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.

Adaptive Server supports local and global indexes.
Partition IDs

- 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. Sybase 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.

partitioning types

Adaptive Server supports four data-partitioning types:

- Range partitioning
• Hash partitioning
• List partitioning
• Round-robin partitioning

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.

Hash partitioning

In hash partitioning, Adaptive Server uses a hash function to specify the partition assignment for each row. You select the partitioning key columns, but Adaptive Server 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.

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.

Round-robin partitioning

In round-robin partitioning, Adaptive Server 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
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, Adaptive Server 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, Adaptive Server 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.

Adaptive Server 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, Adaptive Server 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.

### Example

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 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)
```
Partitioning types

(p1 values <= (5000, 14),
p2 values <= (10000, 10),
p3 values <= (100000, 25))

Adaptive Server 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).

Adaptive Server evaluates tables with more than two partitioning key columns in a similar manner.

Partition pruning

Semantics-based partitioning allow Adaptive Server 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, Adaptive Server can quickly ascertain whether rows in a particular partition can satisfy the query. This behavior is called partition pruning, or partition elimination, and it can save considerable time and resources during execution.

- For range and list partitioning – Adaptive Server can apply partition pruning on equality (=) and range (>, >=, <, and <=) predicates on partition-key columns on a single table.
- For hash partitioning – Adaptive Server can apply partitioning pruning only on equality predicates on a single table.
- For range, list, and hash partitioning – Adaptive Server 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 (see “Composite partitioning keys” on page 383). Adaptive Server 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 Adaptive Server can return query results more efficiently because it needs to scan only p1 and p2.

In these examples, Adaptive Server does not use partition pruning:

```sql
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.

---

**Indexes and partitions**

Indexes help Adaptive Server locate data. They 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.

**Global indexes**

Global indexes span data in one or more partitions, which are not equipartitioned with the base table. Because Sybase supports only unpartitioned global indexes, a global index spans all partitions.
Indexes created on partitioned tables in versions of Adaptive Server earlier than 15.0 are all global indexes. Global indexes are supported for compatibility with earlier versions of Adaptive Server, and because they are particularly useful in OLTP environments.

Adaptive Server 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**

The example in Figure 10-1 shows the default nonclustered index configuration, which is supported in Adaptive Server 12.5 and later.

To create this index on the unpartitioned `publishers` table, enter:

```
create nonclustered index publish5_idx on publishers(pub_id)
```

*Figure 10-1: Global nonclustered index on an unpartitioned table*

```
Nonclustered index on the pub_id column

Index pages

Data pages
```

**Global clustered index on unpartitioned table**

Figure 10-2 shows the default clustered index configuration. The table and index are unpartitioned.

To create this table on an unpartitioned `publishers` table, enter:
Global clustered index on round-robin partitioned table

Adaptive Server supports unpartitioned, clustered global indexes for round-robin partitioned tables only.

Note If a round-robin table has more than 255 data partitions, you cannot create a global index on that table. You must create a local index.

An unpartitioned index allows a full table scan on all partitions. Because the leaf index page is also the data page on allpages-locked tables, this index is most useful when all data partitions reside on the same segment. You must create the index on the data-partitioning key.

For this example (Figure 10-3), we create three round-robin partitions on the publishers table from pubs2. Drop any indexes before creating the partitions.

    alter table publishers partition 3

To create a clustered index on the round-robin partitioned publishers table, enter:

    create clustered index publish1_idx
    on publishers(pub_id)
Indexes and partitions

Figure 10-3: Global clustered index on a round-robin partitioned table

Global nonclustered index on partitioned table

You can create nonclustered, unpartitioned global indexes for all partitioning table strategies.

The index and the data partitions can reside on the same or different segments. You can create the index on any indexable column in the table.

The example in Figure 10-4 is indexed on the `pub_name` column; the table is partitioned on the `pub_id` column.

For this example, we use `alter table` to repartition `publishers` with three range partitions on the `pub_id` column.

```sql
alter table publishers partition by range(pub_id)
(a values <= ('100'),
b values <= ('200'),
c values <= ('300'))
```

To create a global nonclustered index on the `pub_name` column, enter:

```sql
create nonclustered index publish2_idx
on publishers(pub_name)
```
In Figure 10-5, the example is indexed on the `pub_id` column; the table is also partitioned on the `pub_id` column.

To create a global nonclustered index on the `pub_id` column, enter:

```
create nonclustered index publish3_idx
  on publishers (pub_id)
```

Figure 10-5: Global nonclustered index on a partitioned table
Indexes and partitions

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.

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.

Figure 10-6 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
```

Figure 10-6: Local clustered index – unique

Clustered index on the pub_id column
The example in Figure 10-7 is indexed on the pub_name column. It cannot enforce uniqueness. See “Guaranteeing a unique index” on page 392.

To create this example on the range-partitioned publishers table, enter:

```sql
create clustered index publish7_idx
  on publishers(pub_name)
  local index p1, p2, p3
```

**Figure 10-7: Local clustered index— not unique**

Clustered index on the name column

![Clustered index diagram]

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 “Global nonclustered index on partitioned table” on page 388, create a partitioned, nonclustered index on the pub_id and city columns:

```sql
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:

```sql
create nonclustered index publish9_idx (B)
  on publishers(city)
  local index p1, p2, p3
```

Figure 10-8 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. See “Guaranteeing a unique index” on page 392.
Guaranteeing 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. See Chapter 13, “Creating Indexes on Tables,” for more information about creating unique indexes.

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. See example A in Figure 10-8.

• 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.

Creating and managing partitions

To enable semantic partitioning at your licensed site, enter:

```
sp_configure 'enable semantic partitioning', 1
```

Round-robin partitioning is always available, and unaffected by the value of `enable semantic partitioning`.

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`

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.
Creating and managing partitions

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.

A typical order of partitioning tasks is:

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:

   ```sql
   use master
   disk init
   name = "pubs_dev1",
   physname = "SYB_DEV01/pubs_dev",
   size = "50M"
   ```


2. Use `alter database` to assign the new device to the database containing the table or index to partition. For example:

   ```sql
   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`.

   ```sql
   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:

   ```sql
   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:

   ```sql
   use master
   sp_dboption pubs2,"select into", true
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`:

```sql
use pubs2
alter table salesdetail partition by range (qty)
(smsales values <= (1000) on seg1,
 medsales values <= (5000) on seg2,
 lgsales values <= (10000) on seg3)
```

Re-create indexes on the partitioned table. For example, on the `salesdetail` table:

```sql
use pubs2
create nonclustered index titleidind
    on salesdetail (title_id)
create nonclustered index salesdetailind
    on salesdetail (stor_id)
```

### Creating data partitions

This section describes how to use `create table` to create range-, hash-, list-, and round-robin–partitioned tables. See the *Reference Manual: Commands*.

### Creating a range-partitioned table

This example creates a range-partitioned table called `fictionsales`; it has four partitions, one for each quarter of the year. For best performance, each partition resides on a separate segment:

```sql
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:

```sql
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)
```

**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, Adaptive Server 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`:

  ```sql
  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 Adaptive Server converts the bound values to `integer` values. The partition bounds are not compatible with the partition-key datatype.

  ```sql
  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. Adaptive Server 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"))

Creating a hash-partitioned table

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. Adaptive Server chooses the hash function and attempts to distribute the rows equally among the partitions.

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.
Creating 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:

```sql
create table my_publishers
(publish_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.

Creating 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:

```sql
create table currentpublishers
(publish_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.

Creating partitioned indexes

This section describes how to create partitioned indexes using create index. See the Reference Manual: Commands.
Indexes can be created in serial or parallel mode, however, you can create global indexes on round-robin-partitioned tables only in parallel mode. See Chapter 7, “Controlling Optimization,” in the Performance and Tuning Series: Query Processing and Abstract Plans.

Creating global indexes

You can create global, clustered indexes only for round-robin–partitioned tables. Adaptive Server supports global, nonclustered, unpartitioned indexes for all types of partitioned tables.

You can create clustered and nonclustered global indexes on partitioned tables using syntax supported in Adaptive Server version 12.5.x and earlier.

When you create an index on a partitioned table, Adaptive Server automatically creates a global index, if you:

- Create a nonclustered index on any partitioned table, and do not include the local index keywords. For example, on the hash-partitioned table mysalesdetail, described in “Creating a hash-partitioned table” on page 397, enter:

```
create nonclustered index ord_idx on mysalesdetail (au_id)
```

- Create a clustered index on a round-robin–partitioned table, and do not include the local index keywords. For example, on currentpublishers table, described in “Creating a round-robin–partitioned table” on page 398, enter:

```
create clustered index pub_idx on currentpublishers
```

Creating local indexes

Adaptive Server 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, Adaptive Server 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—see “Creating a hash-partitioned table” on page 397. 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.
Creating clustered indexes on partitioned tables

You can create a clustered index on a partitioned table if:

- The `select into/bulkcopy/pllsort` database option is set to true, and
- There are as many worker threads available as the number of partitions.

**Note** Before creating a global index on a round-robin–partitioned table, make sure the server is configured to run in parallel.

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.

Creating 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:

```sql
select * into sales_report partition by range (qty)
(smallorder values <= (500) on seg1,
bigorder values <= (5000) on seg2)
```
from salesdetail

Alter data partitions

You can use the alter table command to:

• Change an unpartitioned table to a multipartitioned table
• 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

See the Reference Manual: Commands.

❖ Repartitioning a table

The general procedure for repartitioning a table is:

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.

Changing an unpartitioned table to a partitioned table

This example changes an unpartitioned titles table to a table with three range partitions:

```sql
alter table titles partition by range (total_sales)
(smallsales values <= (500) on seg1,
mediumsales values <= (5000) on seg2,
bigsales values <= (25000) on seg3)
```
Adding 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.

Changing the partitioning type

The titles table was range-partitioned in “Adding partitions to a partitioned table” on page 402. 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)
```

**Note** You must drop all indexes before changing the partitioning type.

Changing the partitioning key

The titles table was repartitioned by range on the `total_sales` column in “Changing the partitioning type” on page 402. 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"),
q4 values <= "12/31/2006")

Note You must drop all indexes before changing the partitioning key.

### Unpartitioning 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. See “Using partitions to load table data” on page 408.

### Using the `partition` parameter

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. Adaptive Server 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, Adaptive Server builds empty local indexes on the new partitions. If, when you created the table, you declared a segment, Adaptive Server 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.
Altering partition-key columns

Modify partition-key columns with care. These rules apply:

- 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

See “Restrictions on partition keys and bound values for range-partitioned tables” on page 396 for more information and examples of unsupported conversions.

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: \( p_1 \text{ values} <= 5 \), \( p_2 \text{ values} <= 10 \). A row with a partition key of 5.5 is converted to 5, and the row migrates from \( p_2 \) to \( p_1 \).

- 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.

CHAPTER 10  Partitioning Tables and Indexes

- 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.

Configuring partitions

You can configure partitions to improve performance. The configuration parameters for partitions are:

- number of open partitions – specifies the number of partitions that Adaptive Server 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.


Updating, deleting, and inserting 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. Adaptive Server determines which partitions are to be accessed through a combination of internal logic and the table’s partitioning strategy.

Adaptive Server 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. See “Updating values in partition-key columns” on page 406.

**Updating values in partition-key columns**

For semantically partitioned tables, updating the value in a partition-key column can cause the data row to move from one partition to another.

Adaptive Server 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 Chapter 1, “Controlling Physical Data Placement,” in the *Performance and Tuning Series: Physical Database Tuning* and Chapter 5, “Indexes,” in the *Performance and Tuning Series: Locking and Concurrency Control*.

### Displaying 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*.

### Using functions

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.
Truncating 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.

Using partitions to load table data

You can use partitioning to expedite the loading of large amounts of table data, even when the table will eventually be used as an unpartitioned table.

Use the round-robin-partitioning method, and place all partitions on the same segment.

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

When the partitions are created, Adaptive Server 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.

Updating partition statistics

The Adaptive Server 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 Adaptive Server 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 in “Changing an unpartitioned table to a partitioned table” on page 401, enter:

    update statistics titles partition smallvalues

See the Reference Manual: Commands.
### VIRTUALLY HASHED TABLES

#### Structure of a virtually hashed table

You can perform hash-based index scans using nonclustered indexes or clustered indexes on data-only-locked tables. 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.

Structure of a virtually hashed table

A virtually hashed table contains two regions—the “hashed” region and the “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.

Figure 11-1: 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.
Creating 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:

```sql
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.

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.

Adaptive Server 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, Adaptive Server 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.
Creating a virtually hashed table

However, because the value 35 is greater than or equal to 10 (the hash factor for the next key column, id), Adaptive Server 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:

```sql
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.

Figure 11-2: Page layout for data

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 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.
Limitations for virtually hashed tables

These are the limitations for virtually hashed tables:

- `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 Adaptive Server 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 Adaptive Server 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. Adaptive Server 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.

Commands that support virtually hashed tables

The changes to the create table command are in “Creating a virtually hashed table” on page 413.

• `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.

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`
Monitor counter support

The `am_srch_hashindex` monitor counter counts the number of times Adaptive Server performs a search using a virtually hashed clustered index.

System procedure support

These system procedures support virtually hashed tables:

- `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:

```
<table>
<thead>
<tr>
<th>attribute_class</th>
<th>attribute</th>
<th>int_value</th>
<th>comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>char_value</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------</td>
<td>-----------</td>
<td>----------</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------</td>
<td>-----------</td>
<td>----------</td>
</tr>
<tr>
<td>misc table info</td>
<td>hash_key_factors</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>id:10.0, id2:1.0, max_hash_key=1000.0</td>
<td>NULL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
CHAPTER 12

Views: Limiting Access to Data

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. 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.

### How views work

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:

```sql
create view authors_ut
as select * from authors
where state = "UT"
```

To display the authors_ut view, enter:

```sql
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. The exceptions are explained later in this chapter.

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.

**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 useful when users of many different interests and skill levels share the same database.

**Security**

Through a view, users can query and modify only the data they can see. The rest of the database is invisible 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 Chapter 17, “Managing User Permissions,” in the System Administration Guide: Volume 1.

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.
How views work

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.

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.

Examples

The first 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:

```sql
create view hiprice
as select *
from titles
where price > $15
    and advance > $5000
```

When Adaptive Server 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`, Adaptive Server 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:

```sql
update hiprice
set price = price * 2
```

Adaptive Server finds the view definition in the system tables and converts the update command into the statement:

```sql
update titles
set price = price * 2
where price > $15
    and advance > $5000
```

In other words, Adaptive Server 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.
Creating views

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.

Creating views

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. Otherwise, the view name must follow the rules for identifiers given in “Identifiers” on page 11.

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.

create view syntax

As illustrated in the `create view` example in “Examples” on page 422, you need not specify any column names in the `create` clause of a view definition statement. Adaptive Server 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:

```sql
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.

Using the select statement 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.

There are a few restrictions on the select statements in a view definition:

- You cannot include order by or compute clauses.
- You cannot include the into keyword.
- You cannot 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 and “Compiled objects” on page 4.

View definition with projection

To create a view with all the rows of the titles table, but with only a subset of its columns, enter:

create view titles_view
as select title, type, price, pubdate
from titles
Creating views

No column names are included in the create view clause. The view titles_view inherits the column names given in the select list.

View definition with a computed column

Here is a view definition statement that creates a view with a computed column generated from the columns price, royalty, and total_sales:

```sql
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.

View definition with an aggregate or built-in function

A view definition that includes an aggregate or built-in function must include column names in the create clause. For example:

```sql
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:

```sql
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. See “Organizing query results into groups: the group by clause” on page 82.
View definition with a join

You can create a view 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:

```sql
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
```

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.

Views derived from other views

You can define a view in terms of another view, as in this example:

```sql
create view hiprice_computer
    as select title, price
           from hiprice
      where type = "popular_comp"
```
**Creating views**

**distinct views**

You can ensure that the rows contained in a view are unique, as in this example:

```sql
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.

Adaptive Server 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:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

When you enter this query:

```sql
select a, b from myview
```

the results look like this:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

(3 rows affected)

The first and third rows appear to be duplicates. However, the underlying view’s rows are still unique.

**Views that include IDENTITY columns**

You can define a view that includes an IDENTITY column by listing the column name, or the `sys_identity` keyword, in the view’s `select` statement. For example:

```sql
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, Adaptive Server 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, Adaptive Server 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.
Validating a view’s selection criteria

Normally, Adaptive Server 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.

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.

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, Adaptive Server 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_ca, Adaptive Server 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"
```

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”:

```sql
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`:

```sql
update stores_cal30
set payterms = "Net 60"
where stor_id = "7067"
```

## Retrieving data through views

When you retrieve data through a view, Adaptive Server 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, Adaptive Server 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:

```sql
create view hiprice
as select * 
from titles
where price > $15 
and advance > $5000
select title, type 
from hiprice
where type = "popular_comp"
```

Internally, Adaptive Server combines the query of `hiprice` with its definition, converting the query to:

```sql
select title, type
```
Retrieving data through views

```sql
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. See “Views derived from other views” on page 427.

**Note** You can use `select` on text and image columns in views. However, you cannot use `readtext` and `writetext` in views.

View resolution

When you define a view, Adaptive Server 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, Adaptive Server 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. These are discussed under “Modifying data through views” on page 435.

If any of these checks fails, Adaptive Server issues an error message.
Redefining views

You can redefine a view without redefining other views that depend on it, unless the redefinition makes it impossible for Adaptive Server 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 recreating 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.

Renaming views

You can rename a view using:

```
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 discussed in “Identifiers” on page 11.
- 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.

Altering or dropping underlying objects

You can change the name of a view’s underlying objects. For example, if a view references a table entitled 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, Adaptive Server 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.

Modifying data through views

Although Adaptive Server 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, the following rules apply to various data modification operations:

- 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. Adaptive Server 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.
Modifying data through 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, Adaptive Server checks to make sure that none of the above restrictions is violated and that no data integrity rules are violated.

Restrictions on updating views

Restrictions on updated views apply to these areas:

- Computed columns in a view definition
- `group by` or `compute` in a view definition
- Null values in underlying objects
- Views created using `with check option`
- Multitable views
- 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.

```sql
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:

```sql
select * from accounts
```
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:

```sql
create view categories (category, average_price) as
select type, avg(price)
from titles
group by type

select * from categories
```

<table>
<thead>
<tr>
<th>category</th>
<th>average_price</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNDECIDED</td>
<td>NULL</td>
</tr>
<tr>
<td>business</td>
<td>13.73</td>
</tr>
<tr>
<td>mod_cook</td>
<td>11.49</td>
</tr>
<tr>
<td>popular_comp</td>
<td>21.48</td>
</tr>
<tr>
<td>psychology</td>
<td>13.50</td>
</tr>
<tr>
<td>trad_cook</td>
<td>15.96</td>
</tr>
</tbody>
</table>

(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 determines what types of modifications you can make through views that join columns from multiple tables. Adaptive Server 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:

```sql
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:

```sql
update multitable_view
    set type = "user_friendly"
    where type = "popular_comp"
```

However, this statement fails because it affects columns from both titles and publishers:

```sql
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.
Dropping views

To delete a view from the database, use:

```
drop view [owner.]view_name [, [owner.]view_name]...
```

As indicated, you can drop more than one view at a time. 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, Adaptive Server 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.

Using views as security mechanisms

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. 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.

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:

```sql
revoke all on titles to public
grant all on bookview to public
grant all on titles to sales
```

Getting information about views

System procedures, catalog stored procedures, and Adaptive Server built-in functions provide information from the system tables about views. See the Reference Manual: Procedures.

Using sp_help and sp_helptext to display view information

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
------------------------------------------------------------------
------------------------------------------------------------------
------------------------------------------------------------------
------------------------
--Adaptive Server 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, Adaptive Server displays a message advising you that the text is hidden. See the Reference Manual: Procedures.
Using \texttt{sp\_depends} to list dependent objects

\texttt{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.

\begin{verbatim}
sp\_depends titles

<table>
<thead>
<tr>
<th>object</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbo.history_proc</td>
<td>stored procedure</td>
</tr>
<tr>
<td>dbo.title_proc</td>
<td>stored procedure</td>
</tr>
<tr>
<td>dbo.titleid_proc</td>
<td>stored procedure</td>
</tr>
<tr>
<td>dbo.deltitle</td>
<td>trigger</td>
</tr>
<tr>
<td>dbo.totalsales_trig</td>
<td>trigger</td>
</tr>
<tr>
<td>dbo.accounts</td>
<td>view</td>
</tr>
<tr>
<td>dbo.bookview</td>
<td>view</td>
</tr>
<tr>
<td>dbo.categories</td>
<td>view</td>
</tr>
<tr>
<td>dbo.hiprice</td>
<td>view</td>
</tr>
<tr>
<td>dbo.multitable_view</td>
<td>view</td>
</tr>
<tr>
<td>dbo.titleview</td>
<td>view</td>
</tr>
</tbody>
</table>

(return status = 0)
\end{verbatim}

Listing all views in a database

\texttt{sp\_tables} lists all views in a database when used in this format:

\begin{verbatim}
sp\_tables @table\_type = "'VIEW'"
\end{verbatim}

Finding an object name and ID

The system functions \texttt{object\_id} and \texttt{object\_name} identify the ID and name of a view. For example:

\begin{verbatim}
select object\_id("titleview")

----------
480004741
\end{verbatim}

Object names and IDs are stored in the \texttt{sysobjects} table.
CHAPTER 13

Creating 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 to users accessing data from that table; Adaptive Server automatically determines when to use the indexes created for tables.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>How indexes work</td>
<td>443</td>
</tr>
<tr>
<td>Creating indexes</td>
<td>446</td>
</tr>
<tr>
<td>Indexes on computed columns</td>
<td>452</td>
</tr>
<tr>
<td>Function-based indexes</td>
<td>452</td>
</tr>
<tr>
<td>Using clustered or nonclustered indexes</td>
<td>453</td>
</tr>
<tr>
<td>Specifying index options</td>
<td>455</td>
</tr>
<tr>
<td>Dropping indexes</td>
<td>457</td>
</tr>
<tr>
<td>Determining what indexes exist on a table</td>
<td>458</td>
</tr>
<tr>
<td>Updating statistics about indexes</td>
<td>460</td>
</tr>
</tbody>
</table>

For information on how you can design indexes to improve performance, see the Performance and Tuning Series: Locking and Concurrency Control. For information about creating and managing partitioned indexes, see Chapter 10, “Partitioning Tables and Indexes.”

How indexes work

Indexes speed data retrieval by pointing to the location of a table column’s data on disk. For example, suppose you need to run frequent queries using the identification numbers of stores in the stores table. To prevent Adaptive Server from having to search through each row in the table—which can be time-consuming if the stores table contains millions of rows—you could create the following index, entitled stor_id_ind:

```sql
create index stor_id_ind 
  on stores (stor_id)
```
How indexes work

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; Adaptive Server determines whether to use the indexes for each query submitted for that table. As the data in a table changes over time, Adaptive Server may change the table’s indexes to reflect those changes. Again, these changes are transparent to users.

Adaptive Server 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. Adaptive Server 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 Adaptive Server 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.

Local and global indexes are described in Chapter 10, “Partitioning Tables and Indexes.” The remaining types of indexes are described in more detail later in this chapter.
Comparing the two ways to create indexes

You can create indexes on tables either by using the create index statement (described in this chapter) 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. For information about the unique and primary key constraints, see Chapter 8, “Creating Databases and Tables.”

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 generally be indexed, so that Adaptive Server can take advantage of the indexed order.
Creating indexes

- 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.

Creating indexes

Before executing `create index`, turn on `select into`:

```sql
sp_dboption,'select into', true
```

The simplest form of `create index` is:

```sql
create index index_name
    on table_name (column_name)
```

To create an index on the `au_id` column of the `authors` table, execute:

```sql
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.

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.

**create index syntax**

The following sections explain the various options to the `create index` command. For information about creating and managing index partitions, including use of the `index_partition_clause`, see Chapter 10, “Partitioning Tables and Indexes.”

**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.

**Indexing more than one column: composite indexes**

To create a composite index on the combined values in all the specified columns, specify two or more column names.

Use composite indexes when two or more columns are best searched as a unit, for example, the first and last names of authors in the `authors` table. List the columns to be included in the composite index in sort-priority order, inside the parentheses after the table name, like this:

```
create index auth_name_ind
on authors(au_fname, au_lname)
```

The columns in a composite index do not have to be in the same order as the columns in the `create table` statement; for example, you can reverse the order of `au_lname` and `au_fname`.  

create index auth_name_ind
on authors(au_lname, au_fname)
You can specify up to 31 columns in a single composite index. All the columns in a composite index must be in the same table. See Chapter 4, "Table and Index Size," in the Performance and Tuning Series: Physical Database Tuning.

Indexing with 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.

For more information on computed columns, see “Computed columns” on page 345.

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
```

Using the *unique* option

A unique index permits no two rows to 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.

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 Adaptive Server 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 try to change data on which there is a unique index, the results depend on whether you have used the *ignore_dup_key* option. See “Using the ignore_dup_key option” on page 455.

You can use the *unique* keyword on composite indexes.

Including 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.
Creating indexes

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:

```sql
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.

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:

```sql
create index nonclustered cust_order_date
    on Orders
    (customer_ID asc,
     date desc)
```

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. See the Reference Manual: Commands.Table 13-1 summarizes information about the space-management properties for indexes.
### Table 13-1: Summary of space-management properties for indexes

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Use</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>fillfactor</td>
<td>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.</td>
<td>Applies only to a clustered index on a data-only-locked table.</td>
<td>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.</td>
</tr>
<tr>
<td>max_rows_per_page</td>
<td>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.</td>
<td>Applies only to allpages-locked tables. The maximum value that you can set this property to is 256.</td>
<td>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.</td>
</tr>
<tr>
<td>reservepagegap</td>
<td>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.</td>
<td>Applies to pages in all locking schemes. If reservepagegap is not specified, no pages are left empty when extents are allocated.</td>
<td></td>
</tr>
</tbody>
</table>
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)
```

Adaptive Server evaluates the computed columns and uses the results to build or update indexes when you create or update an index.

Function-based indexes

With the function-based index feature, you can create indexes directly on functions and expressions. Like computed column indexes, this feature is useful for user-defined ordering and DDS applications.

This example creates an index on a generalized index key, using three columns in the table:

```
CREATE INDEX generalized_index on parts_table
    (general_key(part_no,listPrice,
               part_no>version)
```

In some situations, when you cannot create an index on individual columns, you can create a generalized index key by invoking a user-defined function that returns a composite value on multiple columns.
Using clustered or nonclustered indexes

With a clustered index, Adaptive Server 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. 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 definition of primary and foreign keys, see Chapter 20, “Triggers: Enforcing Referential Integrity.”

With a nonclustered index, the physical order of the rows is not the same as their indexed order. The leaf level of a nonclustered index 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, Adaptive Server 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:

```sql
drop index titles.titleidind
```

Then, create the clustered index:

```sql
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 in Chapter 8, “Creating Databases and Tables,” by postal code, create a nonclustered index on the postalcode column:

```sql
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:

```sql
create clustered index nmind
on friends_etc(pname, sname)
```

### Creating 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.
Specifying 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. Table 13-2 shows which option to use, based on the type of index.

Table 13-2: Index options

<table>
<thead>
<tr>
<th>Index type</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clustered</td>
<td>ignore_dup_row</td>
</tr>
<tr>
<td>Unique clustered</td>
<td>ignore_dup_key</td>
</tr>
<tr>
<td>Nonclustered</td>
<td>None</td>
</tr>
<tr>
<td>Unique nonclustered</td>
<td>None</td>
</tr>
</tbody>
</table>

Using the ignore_dup_key option

If you try to insert a duplicate value into a column that has a unique index, the command is canceled. You can avoid this situation by including the 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, Adaptive Server 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
```
Specifying index options

Using the *ignore_dup_row* and *allow_dup_row* options

generate and allow_row are options for creating a nonunique, clustered index. These options are not relevant when creating a nonunique, nonclustered index. Since an Adaptive Server 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, Adaptive Server 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* option applies only to tables with nonunique indexes; you cannot use this keyword if a unique index exists on any column in the table.

Table 13-3 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.

<table>
<thead>
<tr>
<th>Option</th>
<th>Has duplicates</th>
<th>Enter duplicates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neither option set</td>
<td>create index command fails.</td>
<td>Command fails.</td>
</tr>
<tr>
<td>allow_dup_row set</td>
<td>Command completes.</td>
<td>Command completes.</td>
</tr>
<tr>
<td>ignore_dup_row set</td>
<td>Index created but duplicate rows deleted; error message.</td>
<td>Duplicates not inserted/updated; error message; transaction completes.</td>
</tr>
</tbody>
</table>

Table 13-3: Duplicate row options in indexes
Using the \textit{sorted\_data} option

The \textit{sorted\_data} option of \texttt{create index} speeds index creation when the data in the table is already in sorted order, for example, when you have used \texttt{bcp} to copy data that has already been sorted 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 \textit{sorted\_data}, but data is not in sorted order, an error message appears and the command is aborted.

\textit{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 \texttt{create index} options require a sort even if \textit{sorted\_data} is specified. See the \textit{Reference Manual: Commands}.

Using the \textit{on segment\_name} option

The \textit{on segment\_name} clause specifies a database segment name on which the index is to be created. A nonclustered index can be created on a different segment than the data pages. For example:

\begin{verbatim}
create index titleind
on titles(title)
on seg1
\end{verbatim}

If you use \textit{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.

Dropping indexes

The \texttt{drop index} command removes an index from the database.

When you use this command, Adaptive Server removes the specified indexes from the database, reclaiming their storage space.
Determining what indexes exist on a table

Only the owner of an index can drop it. The \texttt{drop index} command cannot be transferred to other users. The \texttt{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 \texttt{phone_ind} in the \texttt{friends_etc} table:

\begin{verbatim}
  drop index friends_etc.phone_ind
\end{verbatim}

Use \texttt{sp_post_xupload} to check and rebuild indexes after a cross-platform load database where the endian types are different.

---

\textbf{Determining what indexes exist on a table}

Use \texttt{sp_helpindex} to see the indexes that exist on a table. Here is a report on the \texttt{friends_etc} table:

\begin{verbatim}
sp_helpindex friends_etc

<table>
<thead>
<tr>
<th>index_name</th>
<th>index_keys</th>
<th>index_description</th>
<th>index_max_rows_per_page</th>
</tr>
</thead>
<tbody>
<tr>
<td>nmind</td>
<td>pname,sname</td>
<td>clustered</td>
<td>0</td>
</tr>
<tr>
<td>postalcodeind</td>
<td>postalcode</td>
<td>nonclustered</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>index_fillfactor</th>
<th>index_reservepagegap</th>
<th>index_created</th>
<th>index_local</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>May 24 2005 1:49PM</td>
<td>Global Index</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>May 24 2005 1:49PM</td>
<td>Global Index</td>
</tr>
</tbody>
</table>

(2 rows affected)

<table>
<thead>
<tr>
<th>index_ptn_name</th>
<th>index_ptn_seg</th>
</tr>
</thead>
<tbody>
<tr>
<td>nmind_1152004104</td>
<td>default</td>
</tr>
<tr>
<td>postalcodeind_1152004104</td>
<td>default</td>
</tr>
</tbody>
</table>

(2 rows affected)

\texttt{sp_help} runs \texttt{sp_helpindex} at the end of its report.

\texttt{sp_statistics} returns a list of indexes on a table. For example:

\begin{verbatim}
  sp_statistics friends_etc
\end{verbatim}
### Creating Indexes on Tables

<table>
<thead>
<tr>
<th>table_qualifier</th>
<th>table_owner</th>
<th>table_name</th>
<th>index_qualifier</th>
<th>index_name</th>
<th>non_unique</th>
<th>type</th>
<th>seq_in_index</th>
<th>column_name</th>
<th>collation</th>
<th>cardinality</th>
<th>pages</th>
<th>status</th>
<th>status2</th>
</tr>
</thead>
<tbody>
<tr>
<td>pubs2</td>
<td>dbo</td>
<td>friends_etc</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(4 rows affected)
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:

```sql
sp_spaceused friends_etc, 1
```

<table>
<thead>
<tr>
<th>index_name</th>
<th>size</th>
<th>reserved</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>nmind</td>
<td>2 KB</td>
<td>32 KB</td>
<td>28 KB</td>
</tr>
<tr>
<td>postalcodeind</td>
<td>2 KB</td>
<td>16 KB</td>
<td>14 KB</td>
</tr>
</tbody>
</table>

(name rowtotal reserved data index_size unused)

| friends_etc     | 48 KB | 2 KB | 4 KB | 42 KB |

(return status = 0)

## Updating statistics about indexes

The `update statistics` command helps Adaptive Server 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. Use `update statistics` when a large amount of data in an indexed column has been added, changed, or deleted.

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:

```sql
update statistics authors
```

To update the statistics only for the index on the `au_id` column, enter:

```sql
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. Adaptive Server 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.
A default is a value that Adaptive Server 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 with a given user-defined datatype. You can use defaults and rules to help maintain the integrity of data across the database.

### How defaults and rules work

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. As discussed in Chapter 8, “Creating Databases and Tables,” 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 Adaptive Server.

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.
Creating defaults

Every time a user enters a value, Adaptive Server 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. See “Sharing inline defaults” on page 476.

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. For more information about integrity constraints, see Chapter 8, “Creating Databases and Tables.”

Creating defaults

You can create or drop defaults at any time, before or after data has been entered in a table. In general, to create defaults:

1. Define the default, using `create default`.
2. Bind the default to the appropriate table column or user-defined datatype using `sp_bindefault`.
3. 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. This issue is discussed in more detail under “Binding defaults” on page 466.

- 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.

**create default syntax**

The of create default syntax is:

```sql
create default [owner.]default_name
   as constant_expression
```

Default names must follow the rules for identifiers. You can create a default only in the current database.

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:

```sql
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.
Creating defaults

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.

Binding defaults

After you have created a default, use `sp_bindefault` to bind the 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 Adaptive Server 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, Adaptive Server ignores such defaults. When you insert a row into a table without specifying a value for the IDENTITY column, Adaptive Server 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]
```
Creating defaults

_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 Adaptive Server. Instead, use _sp_hidetext_ to encrypt the text in **syscomments**. See the **Reference Manual: Procedures**, and “Compiled objects” on page 4.

Unbinding 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:

```sql
execute sp_unbindefault "friends_etc.city"
```

To unbind a default from the user-defined datatype _citytype_, use:

```sql
sp_unbindefault citytype
```

The complete syntax of _sp_unbindefault_ is:

```sql
sp_unbindefault _objname_ [, _futureonly_] 
```

If the _objname_ parameter you give is not of the form _table.column_, Adaptive Server 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.
How defaults affect NULL values

If you specify NOT NULL when you create a column and do not create a default for it, Adaptive Server produces an error message whenever anyone inserts a row and fails to make an entry in that column.

When you drop a default for a NULL column, Adaptive Server 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.

Table 14-1 illustrates the relationship between the existence of a default and the definition of a column as NULL or NOT NULL.

<table>
<thead>
<tr>
<th>Column definition</th>
<th>User entry</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null and default defined</td>
<td>No value</td>
<td>Default used</td>
</tr>
<tr>
<td></td>
<td>NULL value</td>
<td>NULL used</td>
</tr>
<tr>
<td>Null defined, no default defined</td>
<td>No value</td>
<td>NULL used</td>
</tr>
<tr>
<td></td>
<td>NULL value</td>
<td>NULL used</td>
</tr>
<tr>
<td>Not null, default defined</td>
<td>No value</td>
<td>Default used</td>
</tr>
<tr>
<td></td>
<td>NULL value</td>
<td>Error</td>
</tr>
<tr>
<td>Not null, no default defined</td>
<td>No value</td>
<td>Error</td>
</tr>
<tr>
<td></td>
<td>NULL value</td>
<td>Error</td>
</tr>
</tbody>
</table>

Dropping 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. (See “Unbinding defaults” on page 468.) If you try to drop a default that is still bound, Adaptive Server displays an error message and the drop default command fails.

Here is how to remove citydft. First, you unbind it:

```
sp_unbindefault citydft
```

Then you can drop citydft:

```
drop default citydflt
```

The complete syntax of drop default is:
Creating rules

A rule lets you specify what users can or cannot enter into a particular column or any column with a user-defined datatype. In general, to create a rule you:

1. Create the rule using `create rule`.
2. Bind the rule to a column or user-defined datatype using `sp_bindrule`.
3. 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.

create rule syntax

The syntax of `create rule` 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:

```sql
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:

```sql
create rule agerule
as @age between 1 and 120 and @age != 17
```

### Binding 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`:

```sql
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.
Creating rules

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, Adaptive Server 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 Adaptive Server. Instead, encrypt the text in **syscomments** by using **sp_hidetext**. See the Reference Manual: Procedures and “Compiled objects” on page 4.

These restrictions apply:

- You cannot bind a rule to a text, unitext, image, or timestamp datatype column.
- You cannot use rules on system tables.

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. This is how **pub_idrule** was bound to **publishers.pub_id**:

```
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.

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#"
```
Precedence of rules

Rules bound to columns always take precedence over rules 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.

Table 14-2 indicates the precedence when binding rules to columns and user datatypes where rules already exist:

<table>
<thead>
<tr>
<th>New rule bound to</th>
<th>Old rule bound to</th>
<th>Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>User datatype</td>
<td>Replaces old rule</td>
<td>No change</td>
</tr>
<tr>
<td>Column</td>
<td>Replaces old rule</td>
<td>Replaces old rule</td>
</tr>
</tbody>
</table>

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"
```

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:
Creating rules

@val in (1,2,3)
The column definition overrides the rule, even a rule that specifies:
@val is not null

Unbinding 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,” Adaptive Server 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.
Dropping 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, Adaptive Server 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.

Getting 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`, Adaptive Server displays a message advising you that the text is hidden. See the `Reference Manual: Procedures`.
Sharing inline defaults

If the system security officer has reset the allow select on syscomments.text column parameter with sp_configure (as required to run Adaptive Server 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_helpdef. See Chapter 12, “Introduction to Security,” in the System Administration Guide: Volume 1.

Sharing inline defaults

When you create a new inline default, Adaptive Server looks for an existing shareable inline default having the same value in the database belonging to the same user. If one exists, Adaptive Server binds this object to the column instead of creating a new default. However, if Adaptive Server does not find an existing shareable inline default, it creates a new default.

Adaptive Server 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 Chapter 5, “Setting Configuration Parameters,” in the System Administration Guide, Volume 1.

Creating an inline shared default

Use create table, alter table...add, and alter table...replace to create shareable inline defaults.

Adaptive Server automatically creates and uses sharable inline defaults when the default clause is used in create table or alter table commands. See the Reference Manual: Commands.

For example, if you create this table:

```sql
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:

```sql
create table my_authors2
(auth_id char(6),
```
CHAPTER 14  Defining Defaults and Rules for Data

```sql
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        definition                  created
  ------------------- -------------------
my_titles_moddate_1791386948 DEFAULT '12/12/2012' Dec 6 2010 10:55AM
```

```
sp_helpconstraint my_authors2
  name        definition                  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.

### Unbinding a shared inline default

Similar to regular inline defaults, you cannot explicitly unbind a sharable inline default from a column. Adaptive Server automatically unbinds or drops it from the column during `drop table` or `alter table` commands. When you drop or alter a table, Adaptive Server checks whether any inline defaults are shared with other columns. If they are, Adaptive Server unbinds the columns from the inline default object without dropping the shared default.

If an inline default object is no longer used by any column, Adaptive Server drops it.
Limitations

These 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 Adaptive 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_unbinddefault` and `drop default` commands using the internal default names.
CHAPTER 15  

Using 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. Adaptive Server 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.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>479</td>
</tr>
<tr>
<td>Rules associated with batches</td>
<td>480</td>
</tr>
<tr>
<td>Using control-of-flow language</td>
<td>485</td>
</tr>
<tr>
<td>Local variables</td>
<td>511</td>
</tr>
<tr>
<td>Global variables</td>
<td>516</td>
</tr>
</tbody>
</table>

Introduction

Adaptive Server 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:

```sql
select count(*) from titles
select count(*) from authors
go
```
Rules associated with batches

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, discussed in Chapter 5, “Subqueries: Using Queries Within Other Queries,” 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.

Rules associated with batches

There are rules to govern which Transact-SQL statements can be combined into a single batch. These batch rules are as follows:

- Before referencing objects in a database, issue a use statement for that database. For example:
  ```sql
  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_binddefault 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.

Adaptive Server compiles a batch before executing it. During compilation, Adaptive Server makes no permission checks on objects, such as tables and views, that are referenced by the batch. Permission checks occur when Adaptive Server executes the batch. An exception to this is when Adaptive Server accesses a database other than the current one. In this case, Adaptive Server displays an error message at compilation time without executing any statements in the batch.

Assume that your batch contains these statements:

```sql
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), Adaptive Server returns an error message for that statement and returns results for all the others.

### Examples of using batches

The examples in this section illustrate batches using the format of 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:

```sql
select count(*) from titles
```
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:

```sql
create table test
  (column1 char(10), column2 int)
insert test
  values ('hello', 598)
select * from test
```

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.

```sql
use master
```

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
select 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.

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
create table hello
(column1 char(10), column2 int)
insert hello
values ("hello", 598)
select * from hello

Here are the results of submitting this file to the isql utility:

-------------
  18
 -------------
(1 row affected)
Using control-of-flow language

Use control-of-flow language with interactive statements, in batches, and in stored procedures. Table 15-1 lists the control-of-flow and related keywords and their functions.

See isql in the Utility Guide for environment-specific information about running batches stored in files.
**Table 15-1: Control-of-flow and related keywords**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>if</td>
<td>Defines conditional execution.</td>
</tr>
<tr>
<td>...else</td>
<td>Defines alternate execution when the if condition is false.</td>
</tr>
<tr>
<td>case</td>
<td>Defines conditional expressions using when...then statements instead of if...else.</td>
</tr>
<tr>
<td>begin</td>
<td>Beginning of a statement block.</td>
</tr>
<tr>
<td>…end</td>
<td>End of a statement block.</td>
</tr>
<tr>
<td>while</td>
<td>Repeat performance of statements while condition is true.</td>
</tr>
<tr>
<td>break</td>
<td>Exit from the end of the next outermost while loop.</td>
</tr>
<tr>
<td>…continue</td>
<td>Restart while loop.</td>
</tr>
<tr>
<td>declare</td>
<td>Declare local variables.</td>
</tr>
<tr>
<td>goto label</td>
<td>Go to label:, a position in a statement block.</td>
</tr>
<tr>
<td>return</td>
<td>Exit unconditionally.</td>
</tr>
<tr>
<td>waitfor</td>
<td>Set delay for command execution.</td>
</tr>
<tr>
<td>print</td>
<td>Print a user-defined message or local variable on user’s screen.</td>
</tr>
<tr>
<td>raiserror</td>
<td>Print a user-defined message or local variable on user’s screen and set a system flag in the global variable @@@error.</td>
</tr>
</tbody>
</table>

/* comment */ or --comment  Insert a comment anywhere in a Transact-SQL statement.

**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:

```sql
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:

```sql
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. See Chapter 5, “Subqueries: Using Queries Within Other Queries.”

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.

```sql
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)

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>id</th>
</tr>
</thead>
<tbody>
<tr>
<td>authors</td>
<td>U</td>
<td>16003088</td>
</tr>
<tr>
<td>publishers</td>
<td>U</td>
<td>48003202</td>
</tr>
<tr>
<td>roysched</td>
<td>U</td>
<td>80003316</td>
</tr>
<tr>
<td>sales</td>
<td>U</td>
<td>112003430</td>
</tr>
<tr>
<td>salesdetail</td>
<td>U</td>
<td>144003544</td>
</tr>
<tr>
<td>titleauthor</td>
<td>U</td>
<td>176003658</td>
</tr>
<tr>
<td>titles</td>
<td>U</td>
<td>208003772</td>
</tr>
<tr>
<td>stores</td>
<td>U</td>
<td>240003886</td>
</tr>
<tr>
<td>discounts</td>
<td>U</td>
<td>272004000</td>
</tr>
<tr>
<td>au_pix</td>
<td>U</td>
<td>304004114</td>
</tr>
<tr>
<td>blurbs</td>
<td>U</td>
<td>336004228</td>
</tr>
<tr>
<td>friends_etc</td>
<td>U</td>
<td>704005539</td>
</tr>
<tr>
<td>test</td>
<td>U</td>
<td>912006280</td>
</tr>
<tr>
<td>hello</td>
<td>U</td>
<td>1056006793</td>
</tr>
</tbody>
</table>
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.

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
Using case expression for alternative representation

Use case expression to represent data in a manner that is more meaningful to the user. 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:

```sql
select title, "Contract Status" =
      case
          when contract = 1 then "Contract"
          when contract = 0 then "No Contract"
      end
from titles
```

<table>
<thead>
<tr>
<th>title</th>
<th>Contract Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Busy Executive’s Database Guide</td>
<td>Contract</td>
</tr>
<tr>
<td>Cooking with Computers: Surreptitio</td>
<td>Contract</td>
</tr>
<tr>
<td>You Can Combat Computer Stress!</td>
<td>Contract</td>
</tr>
<tr>
<td>. . .</td>
<td></td>
</tr>
<tr>
<td>The Psychology of Computer Cooking</td>
<td>No Contract</td>
</tr>
<tr>
<td>. . .</td>
<td></td>
</tr>
<tr>
<td>Fifty Years in Buckingham Palace</td>
<td>Contract</td>
</tr>
<tr>
<td>Sushi, Anyone?</td>
<td>Contract</td>
</tr>
</tbody>
</table>

(18 rows affected)

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):

```sql
select title_id, total_sales, advance,
      total_sales/advance from titles
```

<table>
<thead>
<tr>
<th>title_id</th>
<th>total_sales</th>
<th>advance</th>
<th>total_sales/advance</th>
</tr>
</thead>
<tbody>
<tr>
<td>BU1032</td>
<td>4095</td>
<td>5,000.00</td>
<td>0.82</td>
</tr>
<tr>
<td>BU1111</td>
<td>3876</td>
<td>5,000.00</td>
<td>0.78</td>
</tr>
<tr>
<td>BU2075</td>
<td>18722</td>
<td>10,125.00</td>
<td>1.85</td>
</tr>
<tr>
<td>BU7832</td>
<td>4095</td>
<td>5,000.00</td>
<td>0.82</td>
</tr>
</tbody>
</table>
Using control-of-flow language

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:

```sql
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
```

<table>
<thead>
<tr>
<th>title_id</th>
<th>total_sales</th>
<th>advance</th>
<th>Cost Per Book</th>
</tr>
</thead>
<tbody>
<tr>
<td>BU1032</td>
<td>4095</td>
<td>5,000.00</td>
<td>0.82</td>
</tr>
<tr>
<td>BU1111</td>
<td>3876</td>
<td>5,000.00</td>
<td>0.78</td>
</tr>
<tr>
<td>BU2075</td>
<td>18722</td>
<td>10,125.00</td>
<td>1.85</td>
</tr>
<tr>
<td>BU7832</td>
<td>4095</td>
<td>5,000.00</td>
<td>0.82</td>
</tr>
<tr>
<td>MC2222</td>
<td>2032</td>
<td>0.00</td>
<td>No Books Sold</td>
</tr>
<tr>
<td>MC3021</td>
<td>22246</td>
<td>15,000.00</td>
<td>1.48</td>
</tr>
<tr>
<td>MC3026</td>
<td>NULL</td>
<td>NULL</td>
<td>No Books Sold</td>
</tr>
<tr>
<td>TC3218</td>
<td>375</td>
<td>7,000.00</td>
<td>0.05</td>
</tr>
<tr>
<td>TC4203</td>
<td>15096</td>
<td>4,000.00</td>
<td>3.77</td>
</tr>
<tr>
<td>TC7777</td>
<td>4095</td>
<td>8,000.00</td>
<td>0.51</td>
</tr>
</tbody>
</table>

(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 Adaptive Server evaluates constant expressions before executing the `case` logic. This sometimes causes division by zero. As a work around:

- Use `nullif()`. For example:
  ```sql
(x/nullif(@foo.0)0
  ```

- Include column values so that they cancel each other, forcing Adaptive Server to evaluate the expression for each row. For example:
  ```sql
(x/(@foo + (col1 - col1))
  ```
Using 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:

```sql
    case expression
      when value1 then result1
      when value2 then result2
      when value3 then result3
      ...
    end
```

are equivalent to this form of case expression:

```sql
    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:

```sql
    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:

```sql
    select
      CASE convert(int, (RAND() * 3))
        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”.

### Case expression results

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. A case expression has a series of alternative result expressions (R1, R2, ..., Rn in the example below) which are specified by the then and else clauses. For example:

```sql
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 ith column, also determine the datatype of a case expression. The datatype of case is determined in the same manner as by the following query:

```sql
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.

### 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 queries with structures similar to:

```sql
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:

```sql
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:

```sql
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*. 
Using control-of-flow language

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:

```sql
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
```

Determining the result set
Use case expression to test for conditions that determine the result set.

The syntax is:

```sql
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:
```sql
select stor_id, title_id, qty, "Book Sales Category" =
    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 Category
---------- -------- ---- ------------------
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:

```sql
select title, royaltyer, "Royalty Category" =
    case
        when (select avg(royaltyer) from titleauthor tta
            where t.title_id = tta.title_id) > 60
            then "High Royalty"
        when (select avg(royaltyer) 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
```

<table>
<thead>
<tr>
<th>title</th>
<th>royaltyer</th>
<th>royalty Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>But Is It User Friendly?</td>
<td>100</td>
<td>High Royalty</td>
</tr>
<tr>
<td>Computer Phobic and Non-Phobic Ind</td>
<td>25</td>
<td>Medium Royalty</td>
</tr>
</tbody>
</table>
```
Using control-of-flow language

<table>
<thead>
<tr>
<th>Title</th>
<th>Price</th>
<th>Royalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer Phobic and Non-Phobic Ind</td>
<td>75</td>
<td>Medium</td>
</tr>
<tr>
<td>Cooking with Computers: Surreptiti</td>
<td>40</td>
<td>Medium</td>
</tr>
<tr>
<td>Cooking with Computers: Surreptiti</td>
<td>60</td>
<td>Medium</td>
</tr>
<tr>
<td>Emotional Security: A New Algorithm</td>
<td>100</td>
<td>High</td>
</tr>
<tr>
<td>Sushi, Anyone?</td>
<td>40</td>
<td>Low</td>
</tr>
<tr>
<td>The Busy Executive’s Database Guide</td>
<td>40</td>
<td>Medium</td>
</tr>
<tr>
<td>The Busy Executive’s Database Guide</td>
<td>60</td>
<td>Medium</td>
</tr>
<tr>
<td>The Gourmet Microwave</td>
<td>75</td>
<td>Medium</td>
</tr>
<tr>
<td>You Can Combat Computer Stress!</td>
<td>100</td>
<td>High</td>
</tr>
</tbody>
</table>

(25 rows affected)

case and value comparisons

This form of case is used for value comparisons. It allows only an equality check between two values; no other comparisons are allowed.

The syntax is:

```sql
case valueT
    when value1 then result1
    when value2 then result2
    . . .
    when valueN then resultN
    else resultX
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:

```sql
case
    when valueT = value1 then result1
    when valueT = value2 then result2
    . . .
    when valueT = valueN then resultN
    else resultX
end
```

This is the same format used for case and search conditions (see “Determining the result set” on page 494).
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:

```sql
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:

```sql
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
```

**coalesce**

coaesce examines a series of values (value1, value2, ..., valuen) and returns the first non-null value. The syntax of coalesce is:

```sql
coalesce(value1, value2, ..., valuen)
```
Using control-of-flow language

Where value1, value2, ..., valuен are expressions. If value1 is non-null, the value of coalesce is value1; if value1 is null, value2 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, Adaptive Server translates it internally to:

```
case
  when value1 is not NULL then value1
  when value2 is not NULL then value2
  . . .
  when valuен-1 is not NULL then valuен-1
  else valuен
end
```

valuен-1 refers to the next to last value, before the final value, valuен.

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
```

<table>
<thead>
<tr>
<th>stor_id</th>
<th>discount</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>10.500000</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>6.700000</td>
<td>100</td>
</tr>
<tr>
<td>NULL</td>
<td>10.000000</td>
<td>1001</td>
</tr>
<tr>
<td>8042</td>
<td>5.000000</td>
<td>NULL</td>
</tr>
</tbody>
</table>

(4 rows affected)

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, Adaptive Server 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
```

<table>
<thead>
<tr>
<th>title</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Busy Executive’s Database Guide</td>
<td>business</td>
</tr>
<tr>
<td>Cooking with Computers: Surreptiti</td>
<td>business</td>
</tr>
<tr>
<td>You Can Combat Computer Stress!</td>
<td>business</td>
</tr>
<tr>
<td>. . .</td>
<td>NULL</td>
</tr>
<tr>
<td>The Psychology of Computer Cooking</td>
<td>trad_cook</td>
</tr>
<tr>
<td>Fifty Years in Buckingham Palace K</td>
<td>trad_cook</td>
</tr>
<tr>
<td>Sushi, Anyone?</td>
<td>trad_cook</td>
</tr>
</tbody>
</table>

(18 rows affected)

The Psychology of Computing is stored in the table as “UNDECIDED,” but the query returns it as type NULL.

**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
```
Using control-of-flow language

```
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.

**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
```

<table>
<thead>
<tr>
<th>title_id</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC1035</td>
<td>22.95</td>
</tr>
<tr>
<td>PS1372</td>
<td>21.59</td>
</tr>
<tr>
<td>TC3218</td>
<td>20.95</td>
</tr>
</tbody>
</table>

(0 rows affected)

<table>
<thead>
<tr>
<th>title_id</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC1035</td>
<td>22.95</td>
</tr>
<tr>
<td>PS1372</td>
<td>21.59</td>
</tr>
<tr>
<td>TC3218</td>
<td>20.95</td>
</tr>
</tbody>
</table>

(3 rows affected)

(18 rows affected)

(0 rows affected)

<table>
<thead>
<tr>
<th>title_id</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.

**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.

See “Local variables” on page 511.
**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.

**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,
```
Using control-of-flow language

```sql
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.

You can find examples using `return` values in Chapter 17, “Using Stored Procedures.”

**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:

```sql
print (format_string | @local_variable | @@global_variable) [,arg_list]
```

For example:

```sql
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:

```sql
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: \texttt{\%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. “\texttt{\%1!}” is the first argument in the original version, “\texttt{\%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:

\texttt{\%1! is not allowed in \%2!}.

The German version of this message is:

\texttt{\%1! ist in \%2! nicht zulässig}.

The Japanese version of the message is:

\textit{Figure 15-1: Japanese message}

\texttt{不在中に \%1! \%2! は許されません。}

Figure 15-1 shows a Japanese phrase, containing the characters “\%1!” 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 \texttt{arg_list} can be a series of either variables or constants. An argument can be any datatype except \texttt{text} or \texttt{image}; it is converted to the \texttt{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 \texttt{format_string} plus all arguments after substitution is 512 bytes.
Using control-of-flow language

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 Adaptive Server. 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 (" "), Adaptive Server 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, Adaptive Server 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.

Creating 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:

```sql
/*
** 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=Case when @remotename is null then "null"
when @remotename is null then @remotename
print @msg, @arg1, @remoteserver
return (1)
end
return(0)
go
```

You can also bind user-defined messages to constraints, as described in "Creating error messages for constraints" on page 309.

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`. 
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 Adaptive Server to wait until the specified period of time has passed. `time 'time'` instructs Adaptive Server 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 Adaptive Server to wait until 4:23 p.m.:

```
waitfor time "16:23"
```

This command instructs Adaptive Server to wait 1 hour and 30 minutes:

```
waitfor delay "01:30"
```

For a review of the formats for time values, see “Entering times” on page 230.

`errorexit` instructs Adaptive Server 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 Adaptive Server 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:

```sql
waitfor delay "0:00:10"
```

After you give the `waitfor` command, you cannot use your connection to Adaptive Server until the time or event that you specified occurs.

### 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:

```sql
/* text of comment */
```

and the “double-hyphen” style:

```sql
-- text of comment
```

### 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:

```sql
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:

```sql
/* 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
```
Using control-of-flow language

```sql
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"
```

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.

Adaptive Server 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:

```sql
-- 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"
```
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.

Declaring local variables

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
```

Local variables and select statements

When you declare a variable, it has the value NULL. Assign values to local variables with a select statement.

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:

```sql
/* 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:

```sql
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.”

```sql
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`. 

Adaptive Server Enterprise
If you are using variables in an update statement, see “Using the set clause with update” on page 254.

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.

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
```

Local variables and while loops and if…else blocks
The following example uses 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
```
Local variables

/* 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

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.

Table 15-2 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.
Table 15-2: Comparing null values

<table>
<thead>
<tr>
<th>Type of comparison</th>
<th>Using the = operator</th>
<th>Using the &lt;, &gt;, &lt;=, !=, !&lt;, !&gt;, or &lt;&gt; operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparing column_value to column_value</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td>Comparing column_value to expression</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>Comparing expression to column_value</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>Comparing expression to expression</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

For example, this test:

```sql
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:

```sql
declare @m money
select title_id, advance
from titles
where advance = @m
title_id advance
-------- ----------------
MC3026   NULL
PC9999   NULL
```

(2 rows affected)
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.


Transactions and global variables

Some global variables provide information to use in transactions.

Checking 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. See “Checking the status from the last fetch” on page 518.

Checking 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.

**Checking 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.

**Checking 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 Table 15-3:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Transaction in progress: an explicit or implicit transaction is in effect; the previous statement executed successfully.</td>
</tr>
<tr>
<td>1</td>
<td>Transaction succeeded: the transaction completed and committed its changes.</td>
</tr>
<tr>
<td>2</td>
<td>Statement aborted: the previous statement was aborted; no effect on the transaction.</td>
</tr>
<tr>
<td>3</td>
<td>Transaction aborted: the transaction aborted and rolled back any changes.</td>
</tr>
</tbody>
</table>

`@@transtate` changes only due to execution errors. Syntax and compile errors do not affect the value of `@@transtate`. See “Checking the state of transactions” on page 717 for examples showing `@@transtate` in transactions.

**Checking 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.
Global variables

Checking 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:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The fetch statement completed successfully.</td>
</tr>
<tr>
<td>1</td>
<td>The fetch statement resulted in an error.</td>
</tr>
<tr>
<td>2</td>
<td>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.</td>
</tr>
</tbody>
</table>

@@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:

```sql
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:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Fetch operation successful.</td>
</tr>
<tr>
<td>-1</td>
<td>Fetch operation unsuccessful.</td>
</tr>
<tr>
<td>-2</td>
<td>Reserved for future use.</td>
</tr>
</tbody>
</table>
Using Transact-SQL Functions in Queries

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setting up a query</td>
<td>519</td>
</tr>
<tr>
<td>Built-in functions</td>
<td>520</td>
</tr>
<tr>
<td>User-created functions</td>
<td>523</td>
</tr>
</tbody>
</table>

Adaptive Server functions are Transact-SQL routines that return information from the database or the system tables. Adaptive Server provides a set of built-in functions. In addition, you can use the `create function` command to create Transact-SQL and SQLJ functions.


Setting up a query

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
```

---

Transact-SQL Users Guide 519
Built-in functions

There are several different types of built-in functions. The rules governing how you use these function types can differ. This section describes the different function types and provides the basics of how you can use them. See the Reference Manual: Building Blocks and, for XML functions, XML Services.

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
```

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 (see “Concatenating expressions” on page 521), and nest them (see “Nesting string functions” on page 522).

Most string functions can be used only on char, nchar, unichar, varchar, univarchar, and datatypes that implicitly convert to char, nchar, 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. Adaptive Server automatically converts the argument to the desired type.
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:

```sql
select "The due date is " + convert(varchar(30),
    pubdate)
from titles
where title_id = "BU1032"

---------------------------------------
The due date is Jun 12 2006 12:00AM
```

Adaptive Server evaluates the empty string (" " or ' ') as a single space. This statement:

```sql
select "abc" + "" + "def"
```

produces:

```
abc def
```

### 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`

### Nesting 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:

```sql
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:

```sql
select substring(pub_id + title_id, 1, 6)  
from titles  
where price > $20
```

--------------

1389PC  
0877PS  
0877TC

**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, Adaptive Server does not report an error if a displayed string is truncated. For example:

```sql
select replicate("a", 16383) + replicate("B", 4000)
```

This shows that the total length would be 20383, but the result string is restricted to 16K.

**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`, `@@texptr`, 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. Sybase 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.

---

### Using `readtext` on unitext columns

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*.

---

### 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. They cannot be used in a where clause.

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

- 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.

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.

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

Vector and scalar aggregates

Aggregate functions can be applied to all the rows in a table, in which case they produce a single value, 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:

```sql
select type, avg(price), avg(avg(price))
from titles
group by type
```
The group by clause applies to the vector aggregate—in this case, \( \text{avg}(\text{price}) \). The scalar aggregate, \( \text{avg}(\text{avg}(\text{price})) \), 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 (\text{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, \text{avg}(\text{price}), \text{avg}(\text{advance})
from titles
group by 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, \text{title_id}, \text{avg}(\text{price}), \text{avg}(\text{advance})
from titles
group by type
order by type
```

```
type title_id
---------- ---------- ---------- ----------
UNDECIDED NULL 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 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. Example 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
```

```
<table>
<thead>
<tr>
<th>type</th>
<th>title_id</th>
<th>price</th>
<th>advance</th>
</tr>
</thead>
<tbody>
<tr>
<td>psychology</td>
<td>PS1372</td>
<td>21.59</td>
<td>7,000.00</td>
</tr>
<tr>
<td>psychology</td>
<td>PS2091</td>
<td>10.95</td>
<td>2,275.00</td>
</tr>
<tr>
<td>psychology</td>
<td>PS2106</td>
<td>7.00</td>
<td>6,000.00</td>
</tr>
<tr>
<td>psychology</td>
<td>PS3333</td>
<td>19.99</td>
<td>2,000.00</td>
</tr>
<tr>
<td>psychology</td>
<td>PS7777</td>
<td>7.99</td>
<td>4,000.00</td>
</tr>
<tr>
<td>sum</td>
<td>sum</td>
<td>67.52</td>
<td>21,275.00</td>
</tr>
</tbody>
</table>
```

Note the difference in output between Example 3 and the examples without compute (Example 1 and Example 2).
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:

```sql
select type, sum(price), sum(advance)
from titles
where type like "%cook"
group by type

<table>
<thead>
<tr>
<th>type</th>
<th>sum(price)</th>
<th>sum(advance)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mod_cook</td>
<td>22.98</td>
<td>15,000.00</td>
</tr>
<tr>
<td>trad_cook</td>
<td>47.89</td>
<td>19,000.00</td>
</tr>
</tbody>
</table>

(2 rows affected)
```

```sql
select type, price, advance
from titles
where type like "%cook"
order by type
```

(2 rows affected)

compute sum(price), sum(advance) by type

<table>
<thead>
<tr>
<th>type</th>
<th>price</th>
<th>advance</th>
</tr>
</thead>
<tbody>
<tr>
<td>mod_cook</td>
<td>2.99</td>
<td>15,000.00</td>
</tr>
<tr>
<td>mod_cook</td>
<td>19.99</td>
<td>0.00</td>
</tr>
<tr>
<td>sum</td>
<td>sum</td>
<td></td>
</tr>
<tr>
<td></td>
<td>22.98</td>
<td>15,000.00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>price</th>
<th>advance</th>
</tr>
</thead>
<tbody>
<tr>
<td>trad_cook</td>
<td>11.95</td>
<td>4,000.00</td>
</tr>
<tr>
<td>trad_cook</td>
<td>14.99</td>
<td>8,000.00</td>
</tr>
<tr>
<td>trad_cook</td>
<td>20.95</td>
<td>7,000.00</td>
</tr>
<tr>
<td>sum</td>
<td>sum</td>
<td></td>
</tr>
<tr>
<td></td>
<td>47.89</td>
<td>19,000.00</td>
</tr>
</tbody>
</table>

(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:

```sql
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)
```

```sql
compute sum(c), max(b), min(a)
```
### Statistical aggregate functions

Statistical aggregate functions permit statistical analysis of numeric data. See also “Aggregate functions” on page 524.

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:
  
  \[ \text{statistical\_agg\_function\_name} ([\text{all} | \text{distinct}] \text{ expression}) \]

- 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.

Formulas for computing standard deviations

See the stddev_samp, stddev_pop var_samp, and var_pop reference pages in the Reference Manual: Blocks to see the formulas that Adaptive Server 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.

Mathematical functions

Mathematical functions return values 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. Adaptive Server converts the argument to the desired type. See “Conversion error handling” on page 544.

Adaptive Server provides error traps to handle domain or range errors. Use the arithabort and arithignore functions to specify how domain errors are handled.

Table 16-1 displays examples using the floor, ceiling, and round mathematical functions.
Table 16-1: Examples of mathematical functions

<table>
<thead>
<tr>
<th>Statement</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>select floor(123)</td>
<td>123</td>
</tr>
<tr>
<td>select floor(123.45)</td>
<td>123.000000</td>
</tr>
<tr>
<td>select floor(1.2345E2)</td>
<td>123.000000</td>
</tr>
<tr>
<td>select floor(-123.45)</td>
<td>-124.000000</td>
</tr>
<tr>
<td>select floor(-1.2345E2)</td>
<td>-124.000000</td>
</tr>
<tr>
<td>select floor($123.45)</td>
<td>123.00</td>
</tr>
<tr>
<td>select ceiling(123.45)</td>
<td>124.000000</td>
</tr>
<tr>
<td>select ceiling(-123.45)</td>
<td>-123.000000</td>
</tr>
<tr>
<td>select ceiling(1.2345E2)</td>
<td>124.000000</td>
</tr>
<tr>
<td>select ceiling(-1.2345E2)</td>
<td>-123.000000</td>
</tr>
<tr>
<td>select ceiling($123.45)</td>
<td>124.00</td>
</tr>
<tr>
<td>select round(123.4545, 2)</td>
<td>123.4500</td>
</tr>
<tr>
<td>select round(123.45, -2)</td>
<td>100.00</td>
</tr>
<tr>
<td>select round(1.2345E2, 2)</td>
<td>123.450000</td>
</tr>
<tr>
<td>select round(1.2345E2, -2)</td>
<td>100.000000</td>
</tr>
</tbody>
</table>

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. Adaptive Server 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 Adaptive Server. Table 16-2 lists each date part, its abbreviation (if there is one), and possible integer values for that part.
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 \( \text{date2} - \text{date1} \) 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
```

<table>
<thead>
<tr>
<th>pubdate</th>
<th>newdate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jun 12 2006</td>
<td>1632</td>
</tr>
<tr>
<td>Jun 9 2005</td>
<td>2000</td>
</tr>
<tr>
<td>Jun 30 2005</td>
<td>1979</td>
</tr>
<tr>
<td>Jun 22 2004</td>
<td>2352</td>
</tr>
<tr>
<td>Jun 9 2006</td>
<td>1635</td>
</tr>
<tr>
<td>Jun 15 2004</td>
<td>2356</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

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
```
Datatype conversion functions

Datatype conversions change an expression from one datatype to another, and reformat the display format for date and time information.

Adaptive Server 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, Adaptive Server 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. See “Using the convert function for explicit conversions” on page 536.

Some datatype conversions are not allowed, either implicitly or explicitly. For example, you cannot convert smallint or binary data to datetime. Table 16-3 and Table 16-4 show whether individual datatype conversions are performed implicitly, explicitly, or are not supported in the tables.

- E – explicit datatype conversion required.
- I – conversion can be explicit or implicit.
- U – datatype conversion is not allowed.
### Table 16-3: Explicit, implicit, and unsupported datatype conversions

<table>
<thead>
<tr>
<th>From</th>
<th>binary</th>
<th>varbinary</th>
<th>bit</th>
<th>[n]char</th>
<th>[n]varchar</th>
<th>datetime</th>
<th>smalldatetime</th>
<th>bigdatetime</th>
<th>bigint</th>
<th>tinyint</th>
<th>unsigned smallint</th>
<th>int</th>
<th>unsigned int</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary</td>
<td>–</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I U</td>
<td>I I I I I</td>
<td>I I I I</td>
<td>I I</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>varbinary</td>
<td>I</td>
<td>–</td>
<td>I</td>
<td>I</td>
<td>I U</td>
<td>I I I I I</td>
<td>I I I I</td>
<td>I I I I</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>bit</td>
<td>I</td>
<td>I</td>
<td>–</td>
<td>I</td>
<td>I U</td>
<td>U U U U I</td>
<td>I I I I</td>
<td>I I I I</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>[n]char</td>
<td>I</td>
<td>E</td>
<td>–</td>
<td>I</td>
<td>I I I I</td>
<td>E E E E E</td>
<td>E E E E</td>
<td>E E E E</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>[n]varchar</td>
<td>I</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>I I I I</td>
<td>E E E E E</td>
<td>E E E E</td>
<td>E E E E</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>datetime</td>
<td>I</td>
<td>I</td>
<td>U</td>
<td>I</td>
<td>I I I I</td>
<td>I I I I I</td>
<td>U U U U U U U</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>smalldatetime</td>
<td>I</td>
<td>I</td>
<td>U</td>
<td>I</td>
<td>I I I I</td>
<td>I I I I I</td>
<td>I I I I U U U U</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>bigdatetime</td>
<td>I</td>
<td>U</td>
<td>I</td>
<td>I</td>
<td>I I I I</td>
<td>I I I I I</td>
<td>I I I I U U U U</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>bigint</td>
<td>I</td>
<td>U</td>
<td>I</td>
<td>I</td>
<td>I I I I</td>
<td>I I I I I</td>
<td>I I I I U U U U</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>tinyint</td>
<td>I</td>
<td>I</td>
<td>E</td>
<td>E</td>
<td>E U</td>
<td>U U U U U</td>
<td>I I I I</td>
<td>I I I I</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>smallint</td>
<td>I</td>
<td>I</td>
<td>E</td>
<td>E</td>
<td>E U</td>
<td>U U U U U</td>
<td>I I I I</td>
<td>I I I I</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>unsigned smallint</td>
<td>I</td>
<td>I</td>
<td>E</td>
<td>E</td>
<td>E U</td>
<td>U U U U U</td>
<td>I I I I</td>
<td>I I I I</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>int</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>E</td>
<td>E U</td>
<td>U U U U U</td>
<td>I I I I</td>
<td>I I I I</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>unsigned int</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>E</td>
<td>E U</td>
<td>U U U U U</td>
<td>I I I I</td>
<td>I I I I</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>bigint</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>E</td>
<td>E U</td>
<td>U U U U U</td>
<td>I I I I</td>
<td>I I I I</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>unsigned bigint</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>E</td>
<td>E U</td>
<td>U U U U U</td>
<td>I I I I</td>
<td>I I I I</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>decimal</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>E</td>
<td>E U</td>
<td>U U U U U</td>
<td>I I I I</td>
<td>I I I I</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>numeric</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>E</td>
<td>E U</td>
<td>U U U U U</td>
<td>I I I I</td>
<td>I I I I</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>float</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>E</td>
<td>E U</td>
<td>U U U U U</td>
<td>I I I I</td>
<td>I I I I</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>real</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>E</td>
<td>E U</td>
<td>U U U U U</td>
<td>I I I I</td>
<td>I I I I</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>money</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I U</td>
<td>U U U U U</td>
<td>I I I I</td>
<td>I I I I</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>smallmoney</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I U</td>
<td>U U U U U</td>
<td>I I I I</td>
<td>I I I I</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>text</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>E</td>
<td>E U</td>
<td>U U U U U</td>
<td>U U U U U U U</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>untext</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E U</td>
<td>U U U U U</td>
<td>U U U U U U U</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>image</td>
<td>E</td>
<td>E</td>
<td>U</td>
<td>U</td>
<td>U U</td>
<td>U U U U U</td>
<td>U U U U U U U</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>unichar</td>
<td>I</td>
<td>I</td>
<td>E</td>
<td>I</td>
<td>I I I I I</td>
<td>I I I I I</td>
<td>I E E E E</td>
<td>E E E E</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>univarchar</td>
<td>I</td>
<td>I</td>
<td>E</td>
<td>I</td>
<td>I I I I I</td>
<td>I I I I I</td>
<td>I E E E E</td>
<td>E E E E</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>date</td>
<td>I</td>
<td>I</td>
<td>U</td>
<td>I</td>
<td>I U</td>
<td>I U U U U</td>
<td>U U U U U U U</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>time</td>
<td>I</td>
<td>I</td>
<td>U</td>
<td>I</td>
<td>I U</td>
<td>I U U U U</td>
<td>U U U U U U U</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>
### Table 16-4: Explicit, implicit, and unsupported datatype conversions

<table>
<thead>
<tr>
<th>From</th>
<th>bigint</th>
<th>unsigned bigint</th>
<th>decimal</th>
<th>numeric</th>
<th>float</th>
<th>real</th>
<th>money</th>
<th>smallmoney</th>
<th>text</th>
<th>untext</th>
<th>image</th>
<th>unichar</th>
<th>univarchar</th>
<th>date</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary</td>
<td>1 I I I I I I I I I I U I I I I I I I I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>varbinary</td>
<td>1 I I I I I I I I I I U I I I I I I I I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bit</td>
<td>I I I I I I I I I I U U U E E E U U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[n]char</td>
<td>E E E E E E E E E I I I I I I I I I I I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[n]varchar</td>
<td>E E E E E E E E E E I I I I I I I I I I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>datetime</td>
<td>U U U U U U U U U U U U U U U I I I I I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>smalldatetime</td>
<td>U U U U U U U U U U U U U U U I I I I I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bigdatetime</td>
<td>U U U U U U U U U U U U U U U I I I I I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bigtime</td>
<td>U U U U U U U U U U U U U U U I I I I I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tinyint</td>
<td>I I I I I I I I I I U U E E E U U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>smallint</td>
<td>I I I I I I I I I I U U U E E U U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>unsigned smallint</td>
<td>I I I I I I I I U U U E E U U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>I I I I I I I I I I U U U E E U U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>unsigned int</td>
<td>I I I I I I I I I I U U U E E U U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bigint</td>
<td>– I I I I I I I I I I U U U E E U U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>unsigned bigint</td>
<td>I – I I I I I I I I U U U E E U U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>decimal</td>
<td>I I I I I I I I I I U U U E E U U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>numeric</td>
<td>I I I – I I I I I I U U U E E U U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>float</td>
<td>I I I I – I I I I I I U U U E E U U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>real</td>
<td>I I I I I – I I U U U E E U U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>money</td>
<td>I I I I I – I U U U E E U U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>smallmoney</td>
<td>I I I I I I I I U U U E E U U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>text</td>
<td>U U U U U U U U – U E E U U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>untext</td>
<td>U U U U U U U U U U U I – I U U U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>image</td>
<td>U U U U U U U U U U U U U I – E E E U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>unichar</td>
<td>E E E E E E E E E E I I I I – I I I I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>univarchar</td>
<td>E E E E E E E E E E I I I I – I I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>date</td>
<td>U U U U U U U U U U U U U I I I I – I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>time</td>
<td>U U U U U U U U U U U U U U I I I I –</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Using the convert function 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, Adaptive Server uses the default length of 30 for character and binary data. If you do not specify a precision or scale, Adaptive Server uses the defaults of 18 and 0, respectively.

**Datatype conversion guidelines and constraints**

**Converting 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, Adaptive Server 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
Built-in functions

Implicit conversions between unichar/univarchar and datetime/smalldatetime are supported.

Converting 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 untext 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.

Converting 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.
Converting 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.

Rounding during conversion to and from money types

The money and smallmoney types store four digits to the right of the decimal point, but round 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.

Converting 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.
Built-in functions

When datetime values are converted to smalldatetime, they are rounded to the nearest minute.

See “Changing the date format” on page 542.

Converting 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, Adaptive Server does not implicitly convert numerics that have a fractional part but return a scale error message. For example, Adaptive Server returns error 241 for numerics that have a fractional part and error 257 if other datatypes are passed.

Use the arithabortion and arithignore options to determine how Adaptive Server handles errors resulting from numeric conversions.

Conversions 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 inttobin 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.
Converting between binary and numeric or decimal types

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. When you convert a binary or varbinary type to numeric or decimal, be sure to specify the “00” or “01” values after the “0x” digit; otherwise, the conversion fails.

For example, to convert the following binary data to numeric use:

```sql
select convert(numeric (38, 18), 0x000000000000000006b14bd1e6eea0000000000000000000000000000000)
```

```
123.456000
```

To convert the same numeric data back to binary use:

```sql
select convert(binary, convert(numeric(38, 18), 123.456))
```

```
0x000000000000000006b14bd1e6eea0000000000000000000000000000000
```

Converting 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.

Converting other types to bit

Exact and approximate numeric types can be converted to the bit type implicitly. 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.

Converting 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.
Built-in functions

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:

```sql
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.

Converting 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.

Converting NULL value

You can use the convert function to change NULL to NOT NULL and NOT NULL to NULL.

Changing 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)
Table 16-5 shows the possible values for \textit{style}, and the variety of date formats you can use. When you use \textit{style} with \textit{smalldatetime}, the styles that include seconds or milliseconds show zeros in those positions.

\textbf{Key to 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.

\textit{Table 16-5: Converting date formats with the style parameter.}

<table>
<thead>
<tr>
<th>Without century (yy)</th>
<th>With century (yyyy)</th>
<th>Standard</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>0 or 100</td>
<td>Default</td>
<td>\textit{mon dd yyyy hh:mm AM (or PM)}</td>
</tr>
<tr>
<td>1</td>
<td>101</td>
<td>USA</td>
<td>mm/dd/yy</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>SQL standard</td>
<td>yy.mm.dd</td>
</tr>
<tr>
<td>3</td>
<td>103</td>
<td>English/French</td>
<td>dd/mm/yy</td>
</tr>
<tr>
<td>4</td>
<td>104</td>
<td>German</td>
<td>dd.mm.yy</td>
</tr>
<tr>
<td>5</td>
<td>105</td>
<td></td>
<td>dd-mm-yy</td>
</tr>
<tr>
<td>6</td>
<td>106</td>
<td></td>
<td>dd mon yy</td>
</tr>
<tr>
<td>7</td>
<td>107</td>
<td></td>
<td>mon dd, yy</td>
</tr>
<tr>
<td>8</td>
<td>108</td>
<td></td>
<td>HH:mm:ss</td>
</tr>
<tr>
<td>-</td>
<td>9 or 109</td>
<td>Default + milliseconds</td>
<td>\textit{mon dd yyyy hh:mm:ss AM (or PM)}</td>
</tr>
<tr>
<td>10</td>
<td>110</td>
<td>USA</td>
<td>mm-dd-yyyy</td>
</tr>
<tr>
<td>11</td>
<td>111</td>
<td>Japan</td>
<td>yy/mm/dd</td>
</tr>
<tr>
<td>12</td>
<td>112</td>
<td>ISO</td>
<td>yymmdd</td>
</tr>
<tr>
<td>13</td>
<td>113</td>
<td></td>
<td>yy/dd/mm</td>
</tr>
<tr>
<td>14</td>
<td>114</td>
<td></td>
<td>mm/yy/dd</td>
</tr>
<tr>
<td>15</td>
<td>115</td>
<td></td>
<td>dd/yy/mm</td>
</tr>
<tr>
<td>-</td>
<td>16 or 116</td>
<td></td>
<td>\textit{mon dd yyyy HH:mm:ss}</td>
</tr>
<tr>
<td>17</td>
<td>117</td>
<td></td>
<td>hh:mmAM</td>
</tr>
<tr>
<td>18</td>
<td>118</td>
<td></td>
<td>HH:mm</td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td>hh:mm:ss:zzzAM</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td>HH:mm:ss:zzz</td>
</tr>
<tr>
<td>21</td>
<td></td>
<td></td>
<td>yy/mm/dd HH:mm:ss</td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
<td>yy/mm/dd hh:mm AM (or PM)</td>
</tr>
<tr>
<td>23</td>
<td></td>
<td></td>
<td>yyyy-mm-ddTHH:mm:ss</td>
</tr>
</tbody>
</table>

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, \textit{dd/mm/yy}: 

```sql
SELECT DATEPART(yyyy, GETDATE()) AS Year, DATEPART(month, GETDATE()) AS Month, DATEPART(day, GETDATE()) AS Day, CONVERT(varchar, DATEPART(time, GETDATE()) + 'T' + DATEPART(millisecond, GETDATE()) / 1000, 103) AS Time
FROM master..spt_values;
```
Built-in functions

```
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) in Table 16-5 to specify the display format. The default value is 100 (*mon dd yyyy hh:mm 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:mm 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
```

Conversion error handling

**Arithmetic overflow and divide-by-zero errors**

Divide-by-zero errors occur when Adaptive Server 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 Adaptive Server 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 Adaptive Server does not execute statements that follow the error-generating statement in the batch. If you set arithabort arith_overflow off, Adaptive Server 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 Adaptive Server 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, Adaptive Server 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, Adaptive Server truncates the query results and continues processing.

**Note** For entrylevel ANSI SQL compliance, set:
- arithabort arith_overflow off
- arithabort numeric_truncation on
- arithignore off
User-created functions

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 happens rarely.

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.

XML functions

The XML functions let you manage XML in the Adaptive Server database. The XML functions are described in the XML Services book.

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 Adaptive Server Database and the Reference Manual: Commands.
User-created functions
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. Adaptive Server also provides system procedures for performing administrative tasks updating the system tables.

You can also create and use extended stored procedures to call procedural language functions from Adaptive Server. See Chapter 18, “Using Extended Stored Procedures.”

### How stored procedures work

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 Adaptive 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 Adaptive 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 Adaptive 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 Adaptive Server 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.

Adaptive Server 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 Sybase-provided system procedures.

Examples

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. See “Restrictions associated with stored procedures” on page 580. 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 Adaptive Server 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 Adaptive Server, include the server name. The syntax for a remote procedure call is:

```
execute server_name.[database_name].[owner].procedure_name
```

See “Executing procedures remotely” on page 570.

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 Adaptive Server. Use `sp_hidetext` to encrypt the text in `syscomments`. See the *Reference Manual: Procedures* and “Compiled objects” on page 4.

Use `sp_helptext` to display the source text of a procedure:

```sql
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, Adaptive Server 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:

```sql
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
```
Permissions

Permissions 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. See Chapter 17, "Managing User Permissions," in the System Administration Guide: Volume 1.
Creating and executing stored procedures

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 Adaptive Server 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 Chapter 5, “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.

Performance

As a database changes, you can optimize 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


Creating and executing stored procedures

You can create a procedure in the current database.

In versions of Adaptive Server 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.
CHAPTER 17 Using Stored Procedures


Permission to issue create procedure defaults to the database owner, who can transfer it to other users.

Using deferred name resolution

When this feature is active (using the set option deferred_name_resolution, or made globally active by using the configuration parameter deferred name resolution), objects inside procedures are resolved at execution time, instead of at creation time. This option allows you 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:

```sql
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.

```sql
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.
Creating and executing stored procedures

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—see “Identifiers” on page 11. 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.

```sql
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`:

```sql
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:

```sql
create proc showind @table varchar(30) as
select table_name = sysobjects.name,
       index_name, index_id
from sysindexes
where object_id = sysobjects.id
and sysindexes.object_id = sysobjects.id
and sysindexes.object_name = @table
```
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>
<thead>
<tr>
<th>table_name</th>
<th>index_name</th>
<th>index_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>titles</td>
<td>titleidind</td>
<td>0</td>
</tr>
<tr>
<td>titles</td>
<td>titleind</td>
<td>2</td>
</tr>
</tbody>
</table>

(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:
Creating and executing stored procedures

```sql
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

<table>
<thead>
<tr>
<th>title</th>
<th>total_sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>Silicon Valley Gastronomic Treats</td>
<td>2032 Books sold</td>
</tr>
</tbody>
</table>

(1 row affected)

Default parameters

You can assign a default value for the 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.

```sql
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, Adaptive Server 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>
<thead>
<tr>
<th>table_name</th>
<th>index_name</th>
<th>index_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>authors</td>
<td>auuidind</td>
<td>1</td>
</tr>
<tr>
<td>authors</td>
<td>auunmind</td>
<td>2</td>
</tr>
</tbody>
</table>
```

(2 rows affected, return status = 0)

If the user does not supply a value, Adaptive Server uses the default, `titles`.

```
showind2

<table>
<thead>
<tr>
<th>table_name</th>
<th>index_name</th>
<th>index_id</th>
</tr>
</thead>
</table>
```
Creating and executing stored procedures

---
| 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, Adaptive Server displays an error message listing the parameters expected by the procedure.

Using default parameters in stored procedures

If you create a stored procedure that uses defaults for parameters, and a user issues the stored procedure, but misspells the parameter name, Adaptive Server executes the stored procedure using the default value and does not issue an error message. For example, if you create the following procedure:

```sql
create procedure test @x int = 1
as select @x
```

It returns the following:

```sql
exec test @x = 2  
go
```

However, if you pass this stored procedure an incorrect parameter, it returns an incorrect result set, but does not issue a error message:

```sql
exec test @z = 4  
go
```

NULL as the default parameter

In the `create procedure` statement, you can declare null as the default value for individual parameters:

```sql
create procedure procedure_name
    @param datatype [ = null ]
[ , @param datatype [ = null ]]...
```

If the user does not supply a parameter, Adaptive Server 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:

```sql
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, Adaptive Server 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`.

**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:

```sql
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
```

**Using more than one parameter**

Here is a variant of `au_info` that uses defaults with wildcard characters for both parameters:

```sql
create proc au_info2
    @lastname varchar(30) = "D%",
```
Creating and executing stored procedures

```sql
@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_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_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, Adaptive Server 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, Adaptive Server attempts to execute the procedure and then the command. For example, if you issue:

sp_help checkpoint

Adaptive Server returns the output from sp_help and runs the checkpoint command. Using delimited identifiers for procedure parameters can produce unintended results.

Using large object text, untext, and image datatypes in stored procedures

With Adaptive Server version 15.7 and later, you can declare a large object (LOB) text, image, or untext datatype for a local variable, and pass that variable as an input parameter to a stored procedure. Prior to version 15.7, a named parameter for text, image, or untext data in a stored procedure could not exceed the current page size: 2, 4, 8, or 16KB.

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.
Creating and executing stored procedures

```
declare @a text
select @a = a2 from t1 where a1 = 3
exec my_procedure @a
```

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

### Procedure groups

The optional semicolon and integer number after the name of the procedure in the `create procedure` and `execute` statements allow you 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 Adaptive Server 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 Chapter 5, “Setting Configuration Parameters,” in the *System Administration Guide: Volume 1*. 
Using *with recompile* in *create procedure*

In the `create procedure` statement, the optional clause `with recompile` comes immediately before the SQL statements. It instructs Adaptive Server 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`, Adaptive Server 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 Adaptive Server to create an execution plan that is different from the one it created when the procedure was first executed. In such situations, Adaptive Server needs a new execution plan.

Use `with recompile` in a `create procedure` statement when you think you need a new plan. See the *Reference Manual: Commands*.

Using *with recompile*

The documentation for Adaptive Server version 12.5 states that if the data to be operated on for various executions of a stored procedure is not uniform, then the stored procedure should be created using `with recompile`, so Adaptive Server recompiles the stored procedure for each execution rather than using the plan from a previous execution. Using this option is much more important in Adaptive Server version 15.0, and becomes crucial with the introduction of deferred compilation in version 15.0.2 and later.

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. The same thing can happen in Adaptive Server 12.5, but since the procedure is optimized using magic numbers, the plans are probably the same, so dramatic performance swings are much less likely.
Troubleshooting

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.

Using WITH RECOMPILE IN EXECUTE

In the execute statement, the optional clause WITH RECOMPILE comes after any parameters. It instructs Adaptive Server 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.

Nesting procedures within 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:

```sql
CREATE PROCEDURE test1 @proc_name VARCHAR(30)
    AS EXEC @proc_name
```
Using temporary tables in stored procedures

You can create and use temporary tables in a stored procedure, but the temporary table exists only for the duration of the stored procedure that creates it. When the procedure completes, Adaptive Server automatically drops the temporary table. 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:
   ```sql
   create table #tempstores
   (stor_id char(4), amount money)
   ```

   **Note** Using `set deferred_name_resolution` makes this step unnecessary; see “Using deferred name resolution” on page 555.

2. Create the procedures that access the temporary table (but not the one that creates it).
   ```sql
   create procedure inv_amounts as
   select stor_id, "Total Due" = sum(amount)
   from #tempstores
   group by stor_id
   ```

3. Drop the temporary table:
   ```sql
   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:
   ```sql
   create procedure inv_proc as
   create table #tempstores
   (stor_id char(4), amount money)
Creating and executing stored procedures

When you run the inv_proc procedure, it creates the table, which only exists during the procedure’s execution. Try inserting values into the #tempstores table or running the inv_amounts procedure:

```sql
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
```

You cannot, because the #tempstores table no longer exists.

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.

Setting 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.

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.
Arguments for stored procedures

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. Sybase recommends that you first test any stored procedures you write that include large numbers of arguments before you implement them in a production environment.

Length of 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 Adaptive Server 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 Adaptive Server, restricted by the lower maximum, may now return larger string values because of the larger maximum page sizes.

Because of the larger value, Adaptive Server 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")
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>255</td>
<td>255</td>
</tr>
</tbody>
</table>

Executing stored procedures

You can execute stored procedures after a time delay, or remotely.
Creating and executing stored procedures

Executing 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:

```sql
begin
    waitfor delay "0:30:00"
    exec testproc
end
```

After issuing the `waitfor` command, you cannot use that connection to Adaptive Server until the specified time or event occurs.

Executing procedures remotely

You can execute procedures on a remote Adaptive Server from your local Adaptive Server. Once both servers are properly configured, you can execute any procedure on the remote Adaptive 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`:

```sql
exec gateway.remotedb.dbo.remoteproc
```

The following examples execute the procedure `namelist` in the `pubs2` database on the GATEWAY server:

```sql
execute gateway.pubs2..namelist
gateway.pubs2.dbo.namelist
exec gateway...namelist
```

The last example works only if `pubs2` is your default database.

See Chapter 15, “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 Adaptive Server appear on your local terminal.
Use the return status from procedures to capture and transmit information messages about the execution status of your procedures. See “Return status” on page 572.

**Warning!** If Component Integration Services is not enabled, Adaptive Server 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, Adaptive Server 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*.

### Deferred compilation in stored procedures

Adaptive Server optimizes stored procedures when they are first executed, as long as values passed for variables are available.

With deferred compilation, Adaptive Server 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.

Adaptive Server 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.

Versions of Adaptive Server earlier than 15.0.2 compiled all statements in a stored procedure before the statements were executed. This meant that the actual values of local variables or knowledge of temporary tables created within the stored procedure were unavailable during optimization. The compiled stored procedure, including the plan, was placed in the procedure cache.

Beginning with Adaptive Server 15.0.2, deferred compilation is used for stored procedures that reference local variables or temporary tables are not compiled until they are ready to be executed.
Returning information from stored procedures

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.

Returning information from stored procedures

Stored procedures can return the following types of information:

- 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 Adaptive Server system procedures include another procedure that verifies certain parameters as valid identifiers.

Remote procedure calls, which are stored procedures that run on a remote Adaptive Server, 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.

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.

Reserved return status values

Adaptive Server reserves 0, to indicate a successful return, and negative values from -1 through -99, to indicate the reasons for failure. Numbers 0 and -1 through -14 are currently used in versions 12 and later, as shown in Table 17-1:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Procedure executed without error</td>
</tr>
<tr>
<td>-1</td>
<td>Missing object</td>
</tr>
<tr>
<td>-2</td>
<td>Datatype error</td>
</tr>
<tr>
<td>-3</td>
<td>Process was chosen as deadlock victim</td>
</tr>
<tr>
<td>-4</td>
<td>Permission error</td>
</tr>
<tr>
<td>-5</td>
<td>Syntax error</td>
</tr>
<tr>
<td>-6</td>
<td>Miscellaneous user error</td>
</tr>
<tr>
<td>-7</td>
<td>Resource error, such as out of space</td>
</tr>
<tr>
<td>-8</td>
<td>Nonfatal internal problem</td>
</tr>
<tr>
<td>-9</td>
<td>System limit was reached</td>
</tr>
<tr>
<td>-10</td>
<td>Fatal internal inconsistency</td>
</tr>
<tr>
<td>-11</td>
<td>Fatal internal inconsistency</td>
</tr>
<tr>
<td>-12</td>
<td>Table or index is corrupt</td>
</tr>
<tr>
<td>-13</td>
<td>Database is corrupt</td>
</tr>
<tr>
<td>-14</td>
<td>Hardware error</td>
</tr>
</tbody>
</table>

Values -15 through -99 are reserved for future use by Adaptive Server.
If more than one error occurs during execution, the status with the highest absolute value is returned.
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:

```sql
create proc checkcontract @titleid tid
as
if (select contract from titles where
    title_id = @titleid) = 1
    return 1
else
    return 2
```

For example:

```sql
checkcontract MC2222
(return status = 1)
```

The following stored procedure calls checkcontract, and uses conditional clauses to check the return status:

```sql
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:

```sql
get_au_stat MC2222
Contract is valid
```

Checking 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:

```sql
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:

```sql
test_proc
You have SA role.
```

## 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):

```sql
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.

```sql
declare @result int
```
Returning information from stored procedures

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. Adaptive Server 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 \( \text{@guess} \) 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 \( \text{@store} \) to hold the value of the variable during the execution of the stored procedure, and by using the “new” returned value of \( \text{@guess} \) in conditional clauses:

\[
\begin{align*}
\text{declare } \text{@guess int} \\
\text{declare } \text{@store int} \\
\text{select } \text{@guess = 32} \\
\text{select } \text{@store = @guess} \\
\text{execute math tutor 5, 6,} \\
\text{@result = @guess output} \\
\text{select Your\_answer = @store,} \\
\text{Right\_answer = @guess} \\
\text{if @guess = @store} \\
\quad \text{print "Bingo!"} \\
\text{else} \\
\quad \text{print "Wrong, wrong, wrong!"} \\
\end{align*}
\]

(1 row affected)
(1 row affected)
(return status = 0)

\[
\begin{align*}
\text{@result} & \quad \text{-----------} \\
\text{30} \quad \text{-----------} \\
\text{Your\_answer} & \quad \text{Right\_answer} \\
\text{-----------} & \quad \text{-----------} \\
\text{32} & \quad \text{30} \\
\end{align*}
\]

Wrong, wrong, wrong!

This stored procedure checks to determine whether new book sales would cause an author’s royalty percentage to change (the \( \text{@pc} \) parameter is defined as an output parameter):

\[
\begin{align*}
\text{create proc roy\_check @title tid, @newsales int,} \\
\text{ @pc int output} \\
\text{as} \\
\text{declare @newtotal int} \\
\text{select @newtotal = (select titles.total\_sales +} \\
\text{ @newsales from titles where title\_id = @title)} \\
\text{select @pc = royalty from roysched} \\
\text{where @newtotal } \geq \text{ roysched.lorange and}
\end{align*}
\]
Returning information from stored procedures

@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:

```sql
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`.

### Passing values in parameters

To pass values in parameters, use this format:

```sql
@parameter = @variable
```

You cannot pass constants; there must be a variable name to “receive” the return value. The parameters can be of any Adaptive Server datatype except text, unitext, or image.

**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`.

### 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. The output keyword can be abbreviated to `out`.

```sql
exec myproc @a = @myvara out, @b = @myvarb out
```
Restrictions associated with stored procedures

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.

Restrictions associated with stored procedures

These are additional restrictions to 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.
- Adaptive Server 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.

Qualifying 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, Adaptive Server 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.
Renaming stored procedures

- 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:

```sql
select object_id ('tempdb..mytab')
```

To obtain the correct object ID when you do not have sa_role privileges, use the `exec` command:

```sql
exec("select object_id('tempdb..mytab')")
```

Renaming stored procedures

Use `sp_rename` to rename stored procedures:

```sql
sp_rename objname, newname
```

For example, to rename `showall` to `countall`:

```sql
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.

Renaming objects referenced by procedures

If you rename any of the objects it 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 Adaptive Server 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.
Using 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.


Dropping stored procedures

Use `drop procedure` to remove stored procedures:

```
drop procedure [owner.]procedure_name [, [owner.]procedure_name] ...
```

If a stored procedure that was dropped is called by another stored procedure, Adaptive Server 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.

System procedures

System procedures are:

- Shortcuts for retrieving information from the system tables
- 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 Chapter 17, “Managing User Permissions,” in the System Administration Guide: Volume 1.

The names of system procedures begin with “sp_”. They are created by the installmaster script in the sybsystemprocs database during Adaptive Server installation. The name of the system procedure usually indicates its purpose. For example sp_addalias adds an alias.

**Executing 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.

**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.
Types of system procedures

System procedures can be grouped by function, such as auditing, security administration, data definition, and so on. The following sections list system procedures sorted by type. See the Reference Manual: Procedures for detailed descriptions of all system procedures, listed alphabetically.

Other Sybase-supplied procedures

Sybase provides catalog stored procedures, system extended stored procedures (system ESPs), and dbcc procedures.

Catalog stored procedures

Catalog stored procedures are system procedures that retrieve information from the system tables in tabular form.

System extended stored procedures

Extended stored procedures (ESPs) call procedural language functions from Adaptive Server. 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_”.

See Chapter 18, “Using Extended Stored Procedures.”

dbcc procedures

The 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.

Getting information about stored procedures

Several system procedures provide information from the system tables about stored procedures.
Getting information about stored procedures

Getting 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
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Owner</th>
<th>Object_type</th>
<th>Create_date</th>
</tr>
</thead>
<tbody>
<tr>
<td>byroyalty</td>
<td>dbo</td>
<td>stored procedure</td>
<td>Jul 27 2005 4:30PM</td>
</tr>
</tbody>
</table>

(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.

Viewing 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`, Adaptive Server displays a message advising you that the text is hidden. See the *Reference Manual: Procedures*.

**Identifying 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`:

```sql
sp_depends byroyalty
```

Things the object references in the current database.

<table>
<thead>
<tr>
<th>object</th>
<th>type</th>
<th>updated</th>
<th>selected</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dbo.titleauthor</code></td>
<td>user</td>
<td>table</td>
<td>no</td>
</tr>
</tbody>
</table>

(return status = 0)

The following statement uses `sp_depends` to list all the objects that reference the table `titleauthor`:

```sql
sp_depends titleauthor
```

Things inside the current database that reference the object.

<table>
<thead>
<tr>
<th>object</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dbo.byroyalty</code></td>
<td>stored procedure</td>
</tr>
<tr>
<td><code>dbo.titleview</code></td>
<td>view</td>
</tr>
</tbody>
</table>

(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)

You must drop and re-create the procedure if any of its referenced objects have been renamed.
Getting information about stored procedures

Using `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:

```sql
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:

```sql
set deferred_name_resolution on
-----------------
create procedure p as
select id from sysobjects
where id = 1
sp_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)
```

```sql
sp_depends p
-----------
The object references in the current database.
object updated
selected
```

Adaptive Server Enterprise
Identifying permissions with **sp_helprotect**

*sp_helprotect* reports permissions on a stored procedure (or any other database object). For example:

```
sp_helprotect byroyalty

<table>
<thead>
<tr>
<th>grantor</th>
<th>grantee</th>
<th>type</th>
<th>action</th>
<th>object</th>
<th>column</th>
<th>grantable</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbo</td>
<td>public</td>
<td>Grant</td>
<td>Execute</td>
<td>byroyalty</td>
<td>All</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

(return status = 0)
```
Getting information about stored procedures
Using Extended Stored Procedures

Extended stored procedures (ESPs) provide a mechanism for calling external procedural language functions from within Adaptive Server. 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.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>591</td>
</tr>
<tr>
<td>Creating functions for ESPs</td>
<td>597</td>
</tr>
<tr>
<td>Registering ESPs</td>
<td>606</td>
</tr>
<tr>
<td>Removing ESPs</td>
<td>608</td>
</tr>
<tr>
<td>Executing ESPs</td>
<td>608</td>
</tr>
<tr>
<td>System ESPs</td>
<td>609</td>
</tr>
<tr>
<td>Getting information about ESPs</td>
<td>610</td>
</tr>
<tr>
<td>ESP exceptions and messages</td>
<td>611</td>
</tr>
</tbody>
</table>

Overview

Extended stored procedures allow you to load and dynamically execute external procedural language functions from within Adaptive Server. Each ESP is associated with a corresponding function, which is executed when the ESP is invoked from Adaptive Server.

An ESP allows Adaptive Server to perform a task outside Adaptive Server in response to an event occurring within Adaptive Server. 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

Adaptive Server supplies some system ESPs. For example, one system ESP, xp_cmdshell, executes an operating system command from within Adaptive Server. You can also write your own ESPs using a subset of the Open Server application programming interface (API).

XP Server

Extended stored procedures are implemented by an Open Server application called XP Server, which runs on the same machine as Adaptive Server. Adaptive Server and XP Server communicate through remote procedure calls (RPCs). Running ESPs in a separate process prevents Adaptive Server being affected by failures that may result from faulty ESP code. The advantage of using ESPs over RPCs is that the ESP runs in Adaptive Server 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 Adaptive Server. 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 Adaptive Server license.

XP Server must be running for Adaptive Server to execute an ESP. Adaptive Server starts XP Server the first time an ESP is invoked and shuts down XP Server when Adaptive Server exits.

On Windows, if the start mail session configuration parameter is set to 1, XP Server automatically starts when Adaptive Server starts.
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.

Sybase 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.

Normally, there is no reason for a user to start XP Server manually, since Adaptive Server 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`.

**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:
Overview

- XP Server exits
- The sp_freedll system procedure is invoked
- The esp unload dll configuration parameter is set using sp_configure

Open Server API

Adaptive Server uses the Open Server API, which allows users to run the system ESPs provided with Adaptive Server. Users can also use the Open Server API to implement their own ESPs.

Table 18-1 lists the Open Server routines required for ESP development. For complete documentation of these routines, see the Open Server Server-Library/C Reference Manual.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>srv_bind</td>
<td>Describes and binds a program variable to a parameter</td>
</tr>
<tr>
<td>srv_descfmt</td>
<td>Describes a parameter</td>
</tr>
<tr>
<td>srv_numparams</td>
<td>Returns the number of parameters in the ESP client request</td>
</tr>
<tr>
<td>srv_senddone</td>
<td>Sends results completion message</td>
</tr>
<tr>
<td>srv_sendinfo</td>
<td>Sends messages</td>
</tr>
<tr>
<td>srv_sendstatus</td>
<td>Sends status value</td>
</tr>
<tr>
<td>srv_xferdata</td>
<td>Sends and receives parameters or data</td>
</tr>
<tr>
<td>srv_yield</td>
<td>Suspends execution of the current thread and allows another thread to execute</td>
</tr>
</tbody>
</table>

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.
CHAPTER 18  Using Extended Stored Procedures

Table 18-2: Naming conventions for DLL extensions

<table>
<thead>
<tr>
<th>Platform</th>
<th>DLL extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>HP 9000/800 HP-UX</td>
<td>.sl</td>
</tr>
<tr>
<td>Sun Solaris</td>
<td>.so</td>
</tr>
<tr>
<td>Windows</td>
<td>.dll</td>
</tr>
</tbody>
</table>

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 Adaptive Server:

```sql
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.

```sql
create procedure getonemsg @msg int
    as external name "msgs.dll"
```

When Adaptive Server 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 Adaptive Server by itself or is the first statement in a batch. For example, you can execute `getmsgs` in any of these ways:

```sql
getmsgs
execute getmsgs
exec getmsgs
```

You can execute `getonemsg` in any of these ways:

```sql
getonemsg 20
getonemsg @msg=20
execute getonemsg 20
execute getonemsg @msg=20
exec getonemsg 20
exec getonemsg @msg=20
```
ESPacks and permissions

You can grant and revoke permissions on an ESP as you would on a regular stored procedure.

In addition to normal Adaptive Server 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.

ESPacks and performance

Since both Adaptive Server 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 ESP packs 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-Language/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 ESP packs.
Creating functions for ESPs

There are no restrictions on the contents of a function that implements an ESP. The only requirement is that it be written in a procedural programming language capable of:

- Calling a C-language function
- Manipulating C-language datatypes
- 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 Adaptive Server, either to the one from which it was originally invoked or to another one.

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 $SYBASE/$SYBASE_OCS/include. To locate these files in your source files, include the following in the source code:

- ospublic.h
- oserrror.h

The Open Server library is in $SYBASE/$SYBASE_OCS/lib. The source for the sample program shown in “ESP function example” on page 599 is in $SYBASE/$SYBASE_ASE/sample/esp.

Open Server data structures

These data structures may be useful 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.

### 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**

### Outline of a simple ESP function

An ESP function should have the following basic structure with regard to its interaction with the Open Server API:

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.

**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. This example includes 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 $SYBASE/$SYBASE_ASE/sample/esp.

```c
/*
** 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 Adaptive Server 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
*/
```
Creating functions for ESPs

```c
/*
void xp_message
{
    SRV_PROC *srvproc, /* Pointer to Open Server thread
    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_deacfmt(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;

Building the DLL

You can use any compiler that can produce the required DLL on your server platform.

For general information about compiling and linking a function that uses the Open Server API, see the Open Client/Server Supplement.

Search order for DLLs

Windows searches for DLLs in this order:

1 The directory from which the application was invoked
Creating functions for ESPs

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 $SYBASE/DLL on Windows or
$SYBASE/lib on other platforms.

Absolute path names for the DLL are not supported.

Sample makefile (UNIX)

The following makefile, make.unix, was 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 $SYBASE/$SYBASE_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 $SYBASE/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
BINAR Y = $(PROGRAM)
EXAMPLEDLL = $(PROGRAM)

# Include path where ospublic.h etc reside. You may have them in
# the standard places like /usr/lib etc.
INCLUDEPATH = $(SYBASE)/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)
LDLFLAGS = $(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

Sample definitions file

The following file, xp_echo.def, must be in the same directory as xp_echo.mak. It lists every function to be used as an ESP function in the EXPORTS section.

LIBRARY examples

CODE PRELOAD MOVEABLE DISCARDABLE
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`.

### Using `create procedure`

The `create procedure` 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. It must be the same as the name of its supporting function.

The parameter declarations are the same as for stored procedures, as described in Chapter 17, “Using Stored Procedures.”

Adaptive Server 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 described in the example in “ESP function example” on page 599, assuming that the function is compiled in a DLL named `examples.dll`. The ESP is registered in the `pubs2` database.

```sql
use pubs2
create procedure xp_echo @in varchar(255)
    as external name "examples.dll"
```

See the Reference Manual: Commands.

**Using `sp_addextendedproc`**

`sp_addextendedproc` is compatible with the syntax used by Microsoft SQL Server. You can use it as an alternative to `create procedure`. The syntax is:

```sql
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.

`spx_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 Adaptive Server or for default values for parameters.

The following statements register in the master database an ESP supported by the `xp_echo` routine described in the example in “ESP function example” on page 599, assuming that the function is compiled in a DLL named `examples.dll`:

```sql
use master
sp_addextendedproc "xp_echo", "examples.dll"
```

Removing 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.

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. To change the name of an ESP:

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.

Executing ESPs

Execute an ESP using the same `execute` command that you use to execute a regular stored procedure. See “Creating and executing stored procedures” on page 554.
You can also execute an ESP remotely. See “Executing procedures remotely” on page 570.

Because the execution of any ESP involves a remote procedure call between Adaptive Server 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, Adaptive Server 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, Adaptive Server 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 “ESP function example” on page 599 and the Open Server Server-Library/C Reference Manual for more information about passing values from an ESP function. From the Adaptive Server side, returned values from an ESP are handled the same as for a regular stored procedure.

**System ESPs**

In addition to xp_cmdshell, there are several system ESPs that support Windows features, such as the integration of Adaptive Server 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 Adaptive Server 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_deletemail
- xp_enumgroups
- xp_findnextmsg
- xp_logevent
- xp_readmail
- xp_sendmail
- xp_startmail
- xp_stopmail


### Getting 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
```

<table>
<thead>
<tr>
<th>ESP Name</th>
<th>DLL</th>
</tr>
</thead>
<tbody>
<tr>
<td>getmsgs</td>
<td>msgs.dll</td>
</tr>
</tbody>
</table>

Since the system ESPs are in the sybsystemprocs database, you must be using the sybsystemprocs database to display their names and DLLs:
```sql
use sybsystemprocs
sp_helpextendedproc

<table>
<thead>
<tr>
<th>ESP Name</th>
<th>DLL</th>
</tr>
</thead>
<tbody>
<tr>
<td>xp_freedll</td>
<td>sybyesp</td>
</tr>
<tr>
<td>xp_cmdshell</td>
<td>sybyesp</td>
</tr>
</tbody>
</table>
```

If the source text of an ESP was encrypted using `sp_hidetext`, Adaptive Server displays a message advising you that the text is hidden. See the Reference Manual: Procedures.

### ESP exceptions and messages

Adaptive Server 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. See the sample `xp_message` function in the “ESP function example” on page 599.
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.

Adaptive Server version 15.0 introduces scrollable, read-only cursors, which changes cursor functionality significantly from earlier releases. The 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.

Sybase 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.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selecting rows with cursors</td>
<td>614</td>
</tr>
<tr>
<td>Sensitivity and scrollability</td>
<td>614</td>
</tr>
<tr>
<td>Types of cursors</td>
<td>615</td>
</tr>
<tr>
<td>Cursor scope</td>
<td>616</td>
</tr>
<tr>
<td>Cursor scans and the cursor result set</td>
<td>617</td>
</tr>
<tr>
<td>Making cursors updatable</td>
<td>618</td>
</tr>
<tr>
<td>How Adaptive Server processes cursors</td>
<td>621</td>
</tr>
<tr>
<td>Monitoring cursor statements</td>
<td>624</td>
</tr>
<tr>
<td>Using declare cursor</td>
<td>625</td>
</tr>
<tr>
<td>Opening cursors</td>
<td>627</td>
</tr>
<tr>
<td>Fetching data rows using cursors</td>
<td>628</td>
</tr>
<tr>
<td>Updating and deleting rows using cursors</td>
<td>634</td>
</tr>
<tr>
<td>Closing and deallocating cursors</td>
<td>636</td>
</tr>
<tr>
<td>Scrollable and forward-only cursor examples</td>
<td>637</td>
</tr>
<tr>
<td>Using cursors in stored procedures</td>
<td>643</td>
</tr>
<tr>
<td>Cursors and locking</td>
<td>645</td>
</tr>
<tr>
<td>Enhanced transaction support for updatable cursors</td>
<td>647</td>
</tr>
<tr>
<td>Getting information about cursors</td>
<td>648</td>
</tr>
<tr>
<td>Using browse mode instead of cursors</td>
<td>650</td>
</tr>
</tbody>
</table>
Selecting rows with cursors

Cursors are associated with a select statements, and 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.

Sensitivity and scrollability

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

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.

**Types of cursors**

There are four types of cursors:

- **Client cursors** – declared through Open Client calls (or Embedded SQL). Open Client keeps track of the rows returned from Adaptive Server 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.
Cursor scope

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.

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
- Within a trigger

Within a user session, the cursor exists only until a user ends the session. After the user logs off, Adaptive Server 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. Adaptive Server 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:

```sql
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
```
Cursor scans and the cursor result set

The method Adaptive Server 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, Adaptive Server 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, Adaptive Server 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. Adaptive Server 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. See “Cursors and locking” on page 645.

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.

Adaptive Server 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:

```sql
declare storinfo_crsr cursor
for select stor_id, stor_name, payterms
from stores
where state = "CA"
```
Making cursors updatable

The table stores, specified with the above cursor, does not have any indexes. Adaptive Server 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.

Making 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, Adaptive Server attempts to determine whether a cursor can be updated before designating it as read-only.

For information on making cursors updateable by using configuration parameter select for update in Adaptive Server 15.7 and later, see “Using select for update” on page 49.

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 Adaptive Server correctly performs positioned updates. Make sure the table being updated has a unique index. If it does not, Adaptive Server 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. Adaptive Server 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. See “Types of cursors” on page 615.

**Determining 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. Adaptive Server attempts to use unique indexes for updatable cursors when scanning the base table. For cursors, Adaptive Server considers an index containing an IDENTITY column to be unique, even if it is not so declared.

Adaptive Server 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, Adaptive Server uses the unique index on the pub_id column of publishers (even though pub_id is not included in the definition of newpubs_crsr):

```sql
declare newpubs_crsr cursor
for select pub_name, city, state
from publishers
for update
```

If you do not specify the for update clause, Adaptive Server 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, Adaptive Server must use a unique index defined for one or more of the columns to scan the base table. If no unique index exists, Adaptive Server returns an error message.
Making cursors updatable

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 `column_name_list`. Adaptive Server uses it during the cursor scan. If the table has more than one unique index, you can include the index column in the `column_name_list`, so that Adaptive Server 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 `column_name_list`:

```sql
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 `column_name_list` as follows:

```sql
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, Adaptive Server needs at least one unique index key, not on the list, to perform a cursor scan.

Allowing Adaptive Server 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 Adaptive Server 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 Chapter 8, “Optimization for Cursors,” in the *Performance and Tuning Series: Query Processing and Abstract Plans*. 
How Adaptive Server processes cursors

When accessing data using cursors, Adaptive Server divides the process into several operations:

- **Declaring the cursor**  
  When you declare a cursor, Adaptive Server creates a cursor structure. The server does not compile the cursor from the cursor declaration, however, until the cursor is open.

  See “Using declare cursor” on page 625.

  The following cursor declaration of a default, non-scrollable cursor, `business_crsr`, finds the titles and identification numbers of all business books in the titles table.

  ```sql
  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 Adaptive Server 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.

  ```sql
  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.

- **Opening a cursor**  
  When you open a cursor that has been declared outside of a stored procedure, Adaptive Server 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.
How Adaptive Server processes cursors

When you declare a cursor within a stored procedure, Adaptive Server compiles the cursor the first time the stored procedure is called. Adaptive Server 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, Adaptive Server 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.

- **Fetching from a cursor** 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`. See “Getting multiple rows with each fetch” on page 632.

  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:

  ```sql
  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
```

- **Processing a row**  
  Adaptive Server 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.

- **Closing a cursor**  
  Adaptive Server 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, Adaptive Server 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.

- **Deallocation a cursor**  Adaptive Server 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 Adaptive Server version 15.0 and later, The keyword `cursor` is optional for this command.

  You must declare the cursor again before using it.

---

**Monitoring cursor statements**

Adaptive Server 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_cursors`), and then uses the `show_cached_text` function to show the SQL text for `new_cursors`:

```sql
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


Using 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. See Reference Manual: Commands. 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.

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.

cursor_scrolability

You can use either scroll or no scroll to specify cursor_scrolability. 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.

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.

declare cursor examples

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
```
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>
```

## Opening cursors

After you declare a cursor, you must open it to fetch, update, or delete 
rows. Opening a cursor lets Adaptive Server 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.

Fetching 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, Adaptive Server 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, Adaptive Server positions the cursor at the first index key.

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, Adaptive Server 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 Adaptive Server 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
----------- ------------------- ---------------
```

Fetching data rows using cursors

After you fetch all the rows, the cursor points to the last row of the result set. If you fetch again, Adaptive Server returns a warning through the @@sqlstatus or @@fetch_status global variables (described under “Checking cursor status” on page 630), 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.

Using the into clause

The into clause specifies that Adaptive Server 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>
```

Checking cursor status

Adaptive Server 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 Adaptive Server version 15.0 and later.

Table 19-1 lists @@sqlstatus values and their meanings:
Table 19-1: @@sqlstatus values

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Successful completion of the <code>fetch</code> statement.</td>
</tr>
<tr>
<td>1</td>
<td>The <code>fetch</code> statement resulted in an error.</td>
</tr>
<tr>
<td>2</td>
<td>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 <code>fetch</code> statement for that cursor.</td>
</tr>
</tbody>
</table>

Table 19-2 lists @@fetch_status values and meanings:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>fetch</code> operation successful.</td>
</tr>
<tr>
<td>-1</td>
<td><code>fetch</code> operation unsuccessful.</td>
</tr>
<tr>
<td>-2</td>
<td>Value reserved for future use.</td>
</tr>
</tbody>
</table>

The following example determines the @@sqlstatus for the currently open authors_csr cursor:

```sql
select @@sqlstatus
--------
0
```

(1 row affected)

The following example determines the @@fetch_status for the currently open authors_csr cursor:

```sql
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.
Fetching data rows using cursors

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>Indicates one of the following:</td>
</tr>
<tr>
<td></td>
<td>• 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.</td>
</tr>
<tr>
<td></td>
<td>• 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.</td>
</tr>
<tr>
<td>0</td>
<td>No cursors have been opened, no rows are qualified from the last opened cursor, or the last opened cursor is closed or deallocated.</td>
</tr>
<tr>
<td>n</td>
<td>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.</td>
</tr>
</tbody>
</table>

Getting 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
<table>
<thead>
<tr>
<th>au_id</th>
<th>au_lname</th>
<th>au_fname</th>
</tr>
</thead>
<tbody>
<tr>
<td>648-92-1872</td>
<td>Blotchet-Halls</td>
<td>Reginald</td>
</tr>
<tr>
<td>712-45-1867</td>
<td>del Castillo</td>
<td>Innes</td>
</tr>
<tr>
<td>722-51-5424</td>
<td>DeFrance</td>
<td>Michel</td>
</tr>
</tbody>
</table>
```
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 Adaptive Server, which improves performance.

Checking 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
```
Updating and deleting rows using cursors

If the cursor is updatable, use the update or delete statement to update or delete rows. Adaptive Server 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. See “Making cursors updatable” on page 618.

Updating 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.

Adaptive Server 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.

Deleting 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, Adaptive Server 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, Adaptive Server 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):

```sql
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.

---

**Closing and deallocating cursors**

When you are finished with the result set of a cursor, you can close it using:

```sql
close cursor_name
```

Closing a cursor does not change its definition. If you reopen a cursor, Adaptive Server creates a new cursor result set using the same query as before. For example:

```sql
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:
**Scrollable and forward-only cursor examples**

**Forward-only (default) cursors**

The cursor example in this discussion uses this query:

```
select author = au_fname + " " + au_lname, au_id
from authors
```

The results of the query are:

<table>
<thead>
<tr>
<th>author</th>
<th>au_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>Johnson White</td>
<td>172-32-1176</td>
</tr>
<tr>
<td>Marjorie Green</td>
<td>213-46-8915</td>
</tr>
<tr>
<td>Cheryl Carson</td>
<td>238-95-7766</td>
</tr>
<tr>
<td>Michael O’Leary</td>
<td>267-41-2394</td>
</tr>
<tr>
<td>Dick Straight</td>
<td>274-80-9391</td>
</tr>
<tr>
<td>Meander Smith</td>
<td>341-22-1782</td>
</tr>
<tr>
<td>Abraham Bennet</td>
<td>409-56-7008</td>
</tr>
<tr>
<td>Ann Dull</td>
<td>427-17-2319</td>
</tr>
<tr>
<td>Burt Gringlesby</td>
<td>472-27-2349</td>
</tr>
<tr>
<td>Chastity Locksley</td>
<td>486-29-1786</td>
</tr>
<tr>
<td>Morningstar Greene</td>
<td>527-72-3246</td>
</tr>
<tr>
<td>Reginald Blotchet Halls</td>
<td>648-92-1872</td>
</tr>
<tr>
<td>Akiko Yokomoto</td>
<td>672-71-3249</td>
</tr>
<tr>
<td>Innes del Castillo</td>
<td>712-45-1867</td>
</tr>
<tr>
<td>Michel DeFrance</td>
<td>722-51-5454</td>
</tr>
<tr>
<td>Dirk Stringer</td>
<td>724-08-9931</td>
</tr>
<tr>
<td>Stearns MacFeather</td>
<td>724-80-9391</td>
</tr>
</tbody>
</table>
Scrollable and forward-only cursor examples

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

<table>
<thead>
<tr>
<th>author</th>
<th>au_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>Johnson White</td>
<td>172-32-1176</td>
</tr>
</tbody>
</table>

(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

<table>
<thead>
<tr>
<th>author</th>
<th>au_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>Marjorie Green</td>
<td>213-46-8915</td>
</tr>
<tr>
<td>Cheryl Carson</td>
<td>238-95-7766</td>
</tr>
<tr>
<td>Michael O’Leary</td>
<td>267-41-2394</td>
</tr>
<tr>
<td>Dick Straight</td>
<td>274-80-9391</td>
</tr>
<tr>
<td>Meander Smith</td>
<td>341-22-1782</td>
</tr>
</tbody>
</table>

(5 rows affected)
Each subsequent fetch returns the next five rows:

```
fetch newauthors_crsr
```

<table>
<thead>
<tr>
<th>author</th>
<th>au_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abraham Bennet</td>
<td>409-56-7008</td>
</tr>
<tr>
<td>Ann Dull</td>
<td>427-17-2319</td>
</tr>
<tr>
<td>Burt Gringlesby</td>
<td>472-27-2349</td>
</tr>
<tr>
<td>Chastity Locksley</td>
<td>486-29-1786</td>
</tr>
<tr>
<td>Morningstar Greene</td>
<td>527-72-3246</td>
</tr>
</tbody>
</table>

(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, Adaptive Server 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.

---

**Example table for scrollable cursors**

The examples in this section are executed by a scrollable cursor. To generate the data in Table 19-4, execute:

```
select emp_id, fname, lname
from emp_tab
where emp_id > 2002000
```
Scrollable and forward-only cursor examples

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.

<table>
<thead>
<tr>
<th>Row position</th>
<th>emp_id</th>
<th>fname</th>
<th>lname</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2002010</td>
<td>Mari</td>
<td>Cazalis</td>
</tr>
<tr>
<td>2</td>
<td>2002020</td>
<td>Sam</td>
<td>Clarac</td>
</tr>
<tr>
<td>3</td>
<td>2002030</td>
<td>Bill</td>
<td>Darby</td>
</tr>
<tr>
<td>4</td>
<td>2002040</td>
<td>Sam</td>
<td>Burke</td>
</tr>
<tr>
<td>5</td>
<td>2002050</td>
<td>Mary</td>
<td>Armand</td>
</tr>
<tr>
<td>6</td>
<td>2002060</td>
<td>Mickey</td>
<td>Phelan</td>
</tr>
<tr>
<td>7</td>
<td>2002070</td>
<td>Sam</td>
<td>Fife</td>
</tr>
<tr>
<td>8</td>
<td>2002080</td>
<td>Wanda</td>
<td>Wolfe</td>
</tr>
<tr>
<td>9</td>
<td>2002090</td>
<td>Nina</td>
<td>Howe</td>
</tr>
<tr>
<td>10</td>
<td>2002100</td>
<td>Sam</td>
<td>West</td>
</tr>
</tbody>
</table>

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_tab
where emp_id > 2002000
```

```
open CI
```

The scrolling worktable is now populated with the data in Table 19-4. 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.
CHAPTER 19  Cursors: Accessing Data

fetch absolute 2 CI

The cursor reads the second row from the cursor result set, and returns Row 2, “2002020, Sam, Clarac.” Because the cursor is insensitive, the updated value is invisible to the cursor, and the value of the returned row—“Sam”—is the same as the value of Row 2 in Table 19-4.

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 semisensitive 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
```

The initial rows of the result set contain the data shown in Table 19-4. 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:
Scrollable and forward-only cursor examples

```
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.

---

### Using 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. See “Cursor scope” on page 616.

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.

```sql
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
```
Using cursors in stored procedures

```sql
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)

See Chapter 17, “Using Stored Procedures.”

**Cursors and locking**

**Cursor-locking in server version 15.7 and later**

Adaptive Server versions 15.7 and later support `select for update` to exclusively lock rows in datarows-locked tables for subsequent updates within the same transaction, and for updatable cursors.

For more information on `select for update`, see “Using `select for update`” on page 49 and the Reference Manual: Commands.

If you run `select for update` within a cursor context, the cursor open and fetch statements must be within the context of a transaction. Otherwise, Adaptive Server reverts to pre-15.7 functionality.

**Cursor-locking in server versions earlier than 15.7**

Adaptive Server uses the same cursor-locking mechanisms in versions earlier than 15.7 and when configuration parameter `select for update` is not set in server versions 15.7 and later.

Cursor-locking methods are similar to other locking methods for Adaptive Server. 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. To reduce deadlocks and improve concurrency, Adaptive Server often precedes an exclusive lock with an update lock, which indicates that the client intends to change data on the page.

For updatable cursors, Adaptive Server 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** Adaptive Server 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.

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*.

**Cursor-locking options**

These are the effects of specifying the `holdlock` or `shared` options (of the `select` statement) when you define an updatable cursor:

- 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.

---

**Enhanced transaction support for updatable cursors**

**Transaction support in server versions earlier than 15.7**

In versions earlier than 15.7, Adaptive Server closes a cursor declared with a for update clause and opened within a transaction context, if:

- you explicitly commit the transaction before closing the cursor
- `set close on endtran` is set

See “Using select for update in cursors and DML” on page 49.

**Transaction support in server version 15.7 and later**

In version 15.7 and later, and when `select for update` is set, Adaptive Server supports fetch operations on open cursors after the transaction has been committed.

When you open a cursor, Adaptive Server uses different locking mechanisms based on the transaction mode:

- **Chained mode** – Adaptive Server 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. Adaptive Server continues to use exclusive locks for fetched rows in the new transaction.
- **Unchained mode** – Adaptive Server uses exclusive locks only if you execute an explicit `begin tran` statement before opening the cursor. Otherwise, Adaptive Server 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, Adaptive Server releases all exclusive locks. For subsequent fetch commands, Adaptive Server acquires locks based on the transaction mode:

- Chained mode – Adaptive Server acquires exclusive row locks on fetched rows. Adaptive Server may also acquire update row locks in certain non-optimized conditions.
- Unchained mode – Adaptive Server acquires update row locks on fetched rows. If the fetch commands are preceded by begin tran, Adaptive Server acquires exclusive row locks.

If update row locks are acquired, Adaptive Server 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.

### Getting 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)
Using browse mode instead of cursors

You can also check the status of a cursor using the @@sqlstatus, @@fetch_status, @@cursor_rows, and @@rowcount global variables. See “Checking cursor status” on page 630 and “Checking the number of rows fetched” on page 633.


Using browse mode instead of 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, Sybase discourages browse mode use in more recent Client-Library applications (versions 10.0.x and later). Additionally, browse mode is Sybase-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.

Browsing 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.
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.

Timestamping a new table for browsing

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:

```sql
create table newtable(col1 int, timestamp, col3 char(?))
```

When you insert or update a row, Adaptive Server timestamps it by automatically assigning a unique varbinary value to the timestamp column.

Timestamping an existing table

To prepare an existing table for browsing, use alter table to add a column named timestamp. For example:

```sql
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:

```sql
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.

```sql
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`. 
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.

### How triggers work

Triggers are automatic. They work 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:

```sql
create trigger del_pub
    on publishers
for delete
as
```

---

**CHAPTER 20**

**Triggers: Enforcing Referential Integrity**

---

### How triggers work

Triggers are automatic. They work 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:

```sql
create trigger del_pub
    on publishers
for delete
as
```
How triggers work

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 Adaptive Server 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 Adaptive Server 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 royshed.

• 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.

Using 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.

Chapter 23, “Transactions: Maintaining Data Consistency and Recovery.”

If your application requires one of the above tasks, use a trigger. Otherwise, use a referential integrity constraint to enforce data integrity. Adaptive Server checks referential integrity constraints before it checks triggers so that a data modification statement that violates the constraint does not also fire the trigger.

For more information about referential integrity constraints, see Chapter 8, “Creating Databases and Tables.”

Creating 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 Except for the trigger named deltitle, the triggers discussed in this chapter are not included in the pubs2 database shipped with Adaptive Server. To work with the examples shown in this chapter, create each trigger example by typing in the create trigger statement. 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.
Creating triggers

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.

SQL statements that are not allowed in triggers

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
Using 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 royshed. The titles table is the master table in relation to titleauthor, salesdetail, and royshed.

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.
Using triggers to maintain referential integrity

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.

Testing data modifications against the trigger test tables

Adaptive Server 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:

  ```sql
  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:

![Trigger Test Tables Diagram](image)

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.
Using triggers to maintain referential integrity

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.

```sql
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."
end
```

@@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_ids 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:

```sql
insert salesdetail
values ("7066", "234517", "TC9999", 70, 45)
```

To test for the second condition, enter:

```sql
insert salesdetail
values ("7896", "234518", "TC3218", 75, 80)
```
Delete trigger examples

When you delete a primary-key row, 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 leave 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_ids removed from titles and is now stored in the deleted table. If the trigger finds any such rows, it removes them.

```sql
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.
Using triggers to maintain referential integrity

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:

```sql
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.

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:

```sql
sp_addmessage 35003, "restrict_dtrig - delete failed: row exists in titleauthor for this title_id."
```

The trigger is:

```sql
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:

```sql
```
CHAPTER 20 Triggers: Enforcing Referential Integrity

```
delete titles
where title_id = "PS2091"
```

### Update trigger examples

The following 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
begin
  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
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

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. 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.
Sybase 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.

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.”

```sql
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
```

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.

```sql
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

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):

```sql
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.

**Updating a foreign key**

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. 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.

```sql
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
```
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.

Insert trigger example using multiple rows

The following insert trigger 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.

```sql
create trigger intrig
on salesdetail
for insert as
/* check value of @@rowcount */
```
Delete trigger example using multiple rows

The next example is a delete trigger that updates the `total_sales` column in the `titles` table every time one or more `salesdetail` rows are deleted.

```sql
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`. 
Multirow considerations

Update trigger example using multiple rows

The following update trigger 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.

```sql
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
end
```
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. See “Using correlated subqueries” on page 193. 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:

```sql
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.

```sql
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", "PSSyyyy", "PSSyyyy", 50, 40)
```

When you insert data from junesales into salesdetail, the statement looks like this:

```sql
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:

```sql
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"
Rolling back triggers

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.

Rolling 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]
```

The syntax for rollback transaction is described in Chapter 23, “Transactions: Maintaining Data Consistency and Recovery.”

`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."
```

For more information about `raiserror`, see Chapter 15, “Using Batches and Control-of-Flow Language.”
The following example of an insert trigger performs a similar task to the trigger forinsertrig1 described in “Insert trigger example” on page 660. 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.

```sql
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, Adaptive Server 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, Adaptive Server 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:

```sql
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:

```sql
begin tran
update stores set payterms = "Net 30"
where stor_id = "8042"
insert salesdetail values ("7777", "JR123", "PS9999", 75, 40)
commit tran
```

See Chapter 23, “Transactions: Maintaining Data Consistency and Recovery,” for information on user-defined transactions.
Global login triggers

Adaptive Server 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 Adaptive Server to roll back the transaction and abort the current statement batch.

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 sp_modifylogin or sp_addlogin to set a user-specific login trigger.

Note You can activate sp_logintrigger by setting trace flag -T4073.


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

Sybase suggests that you use nested triggers in an order-dependent sequence. Use separate triggers to cascade data modifications, as in the earlier example of delcascadetrig, described under “Cascading delete example” on page 661.

**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.

**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:

```sql
select * from new_budget
unit parent_unit budget
-------------- -------------- --------
```
Nesting triggers

You can create a trigger that recursively updates `new_budget` whenever its `budget` column is changed, as follows:

```sql
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, Adaptive Server 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:

```sql
select * from new_budget
```

<table>
<thead>
<tr>
<th>unit</th>
<th>parent_unit</th>
<th>budget</th>
</tr>
</thead>
<tbody>
<tr>
<td>one_department</td>
<td>one_division</td>
<td>13</td>
</tr>
<tr>
<td>one_division</td>
<td>company_wide</td>
<td>103</td>
</tr>
<tr>
<td>company_wide</td>
<td>NULL</td>
<td>1003</td>
</tr>
</tbody>
</table>

(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.

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.

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.

Adaptive Server 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, Adaptive Server 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.
Rules associated with triggers

Trigger restrictions

Adaptive Server imposes these 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, Adaptive Server 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.

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.
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.

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.

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.

Trigger tips

Consider these tips when creating 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.

### Disabling triggers

The insert, update, and delete commands normally fire any trigger they encounter, which increases the time needed to perform the operation. 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.

`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:

```sql
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:

```sql
alter table pubs2
    disable del_pub
```

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:

```sql
alter table pubs2
    enable del_pub
```
Dropping triggers

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.

Dropping 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:

```sql
drop trigger [owner.]trigger_name [,...]
```

When you drop a table, Adaptive Server drops any triggers associated with it. drop trigger permission defaults to the trigger table owner cannot be transferred.

Getting 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:

```sql
select *
from sysobjects
where type = "TR"
```

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.
**sp_help**

Get a report on a trigger using `sp_help`. For example, you can get information on `deltitle` using:

```
sp_help deltitle
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Owner</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>deltitle</td>
<td>dbo</td>
<td>trigger</td>
</tr>
</tbody>
</table>

Data_location_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

Data_location_on_segment  When_created
not applicable  Aug 30 1998 11:40PM
Trigger enabled
(return status = 0)

**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
Getting information about triggers

```sql
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`, Adaptive Server 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 Adaptive Server 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 Chapter 12, “Introduction to Security,” in the System Administration Guide: Volume 1.

### `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
```

<table>
<thead>
<tr>
<th>object</th>
<th>type</th>
<th>updated</th>
<th>selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbo.salesdetail</td>
<td>user table</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>dbo.titles</td>
<td>user table</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

(return status = 0)

This statement lists all the objects that reference the `salesdetail` table:

```
sp_depends salesdetail
```

<table>
<thead>
<tr>
<th>object</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbo.deltitle</td>
<td>trigger</td>
</tr>
<tr>
<td>dbo.history_proc</td>
<td>stored procedure</td>
</tr>
<tr>
<td>dbo.insert_salesdetail_proc</td>
<td>stored procedure</td>
</tr>
<tr>
<td>dbo.totalsales_trig</td>
<td>trigger</td>
</tr>
</tbody>
</table>

(return status = 0)
In-Row Off-Row LOB

Overview

Adaptive Server 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), Adaptive Server 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 Adaptive Server 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, Adaptive Server 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 includes serialized Java classes, which are stored in-row only if they are shorter than a fixed maximum length.
Compressing in-row LOB columns

If you set up a table for row- or page-level compression, any in-row LOB columns (or their metadata) are not compressed. If, however, 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.

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, Adaptive Server 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.

Migrating off-row LOB data to use 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. See “In-row LOB columns and bulk copy.”

You can migrate existing data using any of these methods, reducing space usage for the text pages and moving to in-row LOB storage.
In-row LOB columns and bulk copy

The Adaptive Server 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, Adaptive Server 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, Sybase recommends you use the `bcp -Y` option to force Adaptive Server 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.

Examples of methods for migrating existing data

The following examples show the different syntax for each method used to migrate off-row LOB data to in-row storage, as well as compare the results. Since the results are all very similar, which 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.

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 (
Migrating off-row LOB data to use in-row storage

```sql
(9564 rows affected)
```

```
1> exec sp_spaceusage archive, 'table', mymsgs
```

```
1> use pubs2
1>
```

```
Migrating 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'.

<table>
<thead>
<tr>
<th>OwnerName</th>
<th>TableName</th>
<th>IndId</th>
<th>NumRows</th>
<th>UsedPages</th>
<th>RsvdPages</th>
<th>ExtentUtil</th>
<th>ExpRsvdPages</th>
<th>PctBloatUsedPages</th>
<th>PctBloatRsvdPages</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbo</td>
<td>mymsgs_test_upd</td>
<td>0</td>
<td>9564.0</td>
<td>318.0</td>
<td>320.0</td>
<td>99.37</td>
<td>272.0</td>
<td>22.31</td>
<td>17.65</td>
</tr>
<tr>
<td>dbo</td>
<td>mymsgs_test_upd</td>
<td>255</td>
<td>NULL</td>
<td>19132.0</td>
<td>19136.0</td>
<td>99.97</td>
<td>19136.0</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

(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
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'.

<table>
<thead>
<tr>
<th>OwnerName</th>
<th>TableName</th>
<th>IndId</th>
<th>NumRows</th>
<th>UsedPages</th>
<th>RsvdPages</th>
<th>ExtentUtil</th>
<th>ExpRsvdPages</th>
<th>PctBloatUsedPages</th>
<th>PctBloatRsvdPages</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbo</td>
<td>mymsgs_test_upd</td>
<td>0</td>
<td>9564.0</td>
<td>1246.0</td>
<td>1258.0</td>
<td>99.04</td>
<td>272.0</td>
<td>379.23</td>
<td>17.65</td>
</tr>
<tr>
<td>dbo</td>
<td>mymsgs_test_upd</td>
<td>255</td>
<td>NULL</td>
<td>1246.0</td>
<td>1258.0</td>
<td>99.04</td>
<td>272.0</td>
<td>379.23</td>
<td>17.65</td>
</tr>
</tbody>
</table>

(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)
Migrating off-row LOB data to use in-row storage

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.

**Using 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:

```sql
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 'mymgs_test_reorg'.
(9564 rows affected)
REORG REBUILD of table 'mymgs_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 ExpRsvdPages
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
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)
```
Migrating 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):

```sql
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'.

<table>
<thead>
<tr>
<th>OwnerName</th>
<th>TableName</th>
<th>IndId</th>
<th>NumRows</th>
<th>UsedPages</th>
<th>RsvdPages</th>
<th>ExtentUtil</th>
<th>ExpRsvdPages</th>
<th>PctBloatUsedPages</th>
<th>PctBloatRsvdPages</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbo</td>
<td>mymsgs_test_alttab</td>
<td>0</td>
<td>9564.0</td>
<td>1252.0</td>
<td>1258.0</td>
<td>99.52</td>
<td>1728.0</td>
<td>-27.46</td>
<td>-27.20</td>
</tr>
<tr>
<td>dbo</td>
<td>mymsgs_test_alttab</td>
<td>255</td>
<td>NULL</td>
<td>6.0</td>
<td>32.0</td>
<td>18.75</td>
<td>16.0</td>
<td>0.00</td>
<td>100.00</td>
</tr>
</tbody>
</table>
```

(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:

```sql
1> exec sp_spaceusage report, 'table', 'mymsgs%', 'OwnerName, TableName, IndId,'
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.

Identifying in-row LOB length selection

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

<table>
<thead>
<tr>
<th>minlen</th>
<th>datarowsize</th>
<th>maxlen</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>105.000000</td>
<td>201</td>
</tr>
</tbody>
</table>

2 Compute the minimum, 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

<table>
<thead>
<tr>
<th>minlen</th>
<th>avglen</th>
<th>maxlen</th>
</tr>
</thead>
<tbody>
<tr>
<td>536</td>
<td>7608</td>
<td>16080</td>
</tr>
</tbody>
</table>

Out of a total of about 190,000 rows in the DYNPSOURCE table, approximately 65000 of them had the off-row LOB column DYNPSOURCE 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.
You cannot downgrade an Adaptive Server while it has any tables 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 Adaptive Server, copy all data out (`bcp out`), then copy it back in (`bcp in`) on the affected tables.
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. Adaptive Server 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.

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)
```
CHAPTER 22  Using instead of Triggers

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.

**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.

**Nesting and recursion**

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.

### 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.

### 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 Adaptive Server supplies the correct values.

### Example

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 Adaptive Server-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:

```sql
--Create a base table with a not null column that has
--a default
CREATE TABLE td1 (col1 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 col1 != 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.

**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 Adaptive Server 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.
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.

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:

```sql
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:

```sql
DELETE myview FROM myview, mytab
WHERE myview.col1 = mytab.col1
```

But the equivalent statement, using a subquery, is allowed:

```sql
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:
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 Adaptive Server 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 Adaptive Server until the buffered rows are all read. By default, Adaptive Server 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 Adaptive Server 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:

```sql
-- 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 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
Adaptive Server and in the application: Adaptive Server 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 Adaptive Server. Because
Adaptive Server 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
Adaptive Server to find the row.
Getting information about triggers

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 Adaptive Server supports.

Getting information about 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.
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.

### How transactions work

Adaptive Server 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:

  ```sql
  begin {transaction | tran} [transaction_name]
  ```
**How transactions work**

A *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:
  
  ```sql
  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:
  
  ```sql
  commit [transaction | tran | work]
  [transaction_name]
  ```

- **rollback** – rolls a transaction back to a savepoint or to the beginning of a transaction:
  
  ```sql
  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:

```sql
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 */
rollback transaction percentchanged */

commit transaction

Transactions allow Adaptive Server 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, Adaptive Server 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. See “Selecting the transaction mode and isolation level” on page 721.

Transactions and consistency

In a multiuser environment, Adaptive Server 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.

Adaptive Server 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.
The Adaptive Server recovery time is measured in minutes and seconds. You can specify the maximum acceptable recovery time.

The SQL commands related to recovery and backup are discussed in “Backup and recovery of transactions” on page 741.

**Note** Grouping large numbers of Transact-SQL commands into one long-running transaction may affect recovery time. If Adaptive Server fails before the transaction commits, recovery takes longer, because Adaptive Server 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 Adaptive Server DTM features, transactions that update data in multiple servers can also benefit from transactional consistency. See *Using Adaptive Server Distributed Transaction Features*.

**Using transactions**

The `begin transaction` and `commit transaction` commands tell Adaptive Server 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.
Allowing 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`.

**Warning!** 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`
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
System procedures that are not allowed in transactions

You cannot use these 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

Beginning and committing 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 Adaptive Server if the transaction is not currently active.

Rolling back and saving 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. See Table 23-2 on page 734 for the effects of rollback during processing.
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 statements  Rolls back group B
  Group C
  commit tran  Commits groups A and C
```

Until you issue a commit transaction, Adaptive Server considers all subsequent statements to be part of the transaction, unless it encounters another begin transaction statement. At that point, Adaptive Server considers all subsequent statements to be part of the new, nested transaction. See “Nested transactions” on page 719.

rollback transaction or save transaction does not affect Adaptive Server 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.

**Warning!** 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.

Checking the state of transactions

The global variable `@@transtate` keeps track of the current state of a transaction. Adaptive Server determines what state to return by keeping track of any transaction changes after a statement executes.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Transaction in progress. A transaction is in effect; the previous statement executed successfully.</td>
</tr>
<tr>
<td>1</td>
<td>Transaction succeeded. The transaction completed and committed its changes.</td>
</tr>
</tbody>
</table>
Using transactions

Adaptive Server 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 occurred in database 'pubs2', table 'publishers', rule 'pub_idrule', column 'pub_id'.
select @@transtate
----------
2
(1 row affected)
rollback transaction
```
Adaptive Server changes @@transtate only in response to an action taken by a transaction. Syntax and compile errors do not affect the value of @@transtate.

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. Adaptive Server 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 Adaptive Server until the final commit transaction:

```sql
begin tran
select @@trancount
/* @@trancount = 1 */
begin tran
    select @@trancount
    /* @@trancount = 2 */
begin tran
    select @@trancount
    /* @@trancount = 3 */
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.

Example of a transaction

This example shows how you might specify a transaction:

```sql
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_change
/* After updating the royaltyper entries for */
/* the two authors, the user inserts the */
/* savepoint "percent_change," 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_change
/* The transaction rolls back to the savepoint */
```
Selecting the transaction mode and isolation level

Adaptive Server provides these options to 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.

Choosing a transaction mode

Adaptive Server supports these transaction modes:

- 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.

You can set either mode using the chained option of the set command. However, 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:

```
set chained
```

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.

### 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.

Finding the status of the current transaction mode

The global variable @@tranchained to determine Adaptive Server’s current transaction mode. select @@tranchained returns 0 for unchained mode or 1 for chained mode.

Choosing 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.

- 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 Adaptive Server.

- 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. Adaptive Server supports level 2 for data-only-locked tables. It is not supported for allpages-locked tables.
Selecting the transaction mode and isolation level

- Level 3 – ensures that data read by one transaction is valid until the end of that transaction, preventing phantom rows. Adaptive Server 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:

```sql
set transaction isolation level 0
```

Default isolation levels for Adaptive Server and ANSI SQL

By default, the Adaptive Server 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 Adaptive Server to apply a holdlock to all select operations in a transaction. For example:

```sql
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 Adaptive Server 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 Adaptive Server default isolation level:

```sql
set transaction isolation level 1
```
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 Adaptive Server to choose a nonunique index or a table scan, as follows:

```sql
select * from table_name (index table_name)
```

Activity on the underlying table may abort the scan before completion.

Repeatabile 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. 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.
Selecting the transaction mode and isolation level

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. See “Changing the isolation level for a query” on page 727.

Finding 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
```

---------
Changing 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:

<table>
<thead>
<tr>
<th>at isolation option</th>
<th>Isolation level</th>
</tr>
</thead>
<tbody>
<tr>
<td>read uncommitted</td>
<td>0</td>
</tr>
<tr>
<td>read committed</td>
<td>1</td>
</tr>
<tr>
<td>serializable</td>
<td>3</td>
</tr>
</tbody>
</table>

For example, the following two statements query the same table at isolation levels 0 and 3, respectively:

```sql
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. Adaptive Server 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.
Selecting the transaction mode and isolation level

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, Adaptive Server 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.

Isolation level precedences

Precedence rules as they apply to the 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 *
CHAPTER 23 Transactions: Maintaining Data Consistency and Recovery

Transact-SQL Users Guide

from titles holdlock

Adaptive Server issues a warning before executing the above query.

Cursors and isolation levels

Adaptive Server provides three isolation levels for cursors:

- Level 0 – Adaptive Server 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 – Adaptive Server 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 Adaptive Server.

- Level 3 – Adaptive Server 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
```
Selecting the transaction mode and isolation level

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), Adaptive Server 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.

Adaptive Server 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.

Adaptive Server 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:

```sql
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 */
```

Stored procedures and isolation levels

The Sybase 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.
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, Adaptive Server sets the trigger’s isolation level to 1 before executing its first statement.

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. If your application uses cursors, you must also set the close on endtran option. These options are described in “Using cursors in transactions” on page 739.

Using the lock table command to improve performance

The lock table command allows you to explicitly request that before a table is accessed, a table lock be put on it 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 (see “wait/nowait option of the lock table command” on page 743).

These considerations apply to the use of lock table:

- You can issue lock table only within a transaction.
Using transactions in stored procedures and triggers

- 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.

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. Adaptive Server 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. Adaptive Server 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. Adaptive Server 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, which are described in “Transaction modes and stored procedures” on page 736. Triggers are not affected by the current transaction mode, since they are called as part of a data modification statement.

## Errors and transaction rollbacks

Data integrity errors can affect the state of implicit or explicit transactions:

- **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`. Adaptive Server always rolls back any uncommitted transactions at the end of a session.

- **Errors in data modification commands that affect data integrity (see Table 23-3 on page 735):**
  
  - Arithmetic overflow and divide-by-zero errors (effects on transactions can be changed with the `set arithabort arith_overflow` command)
  - Permissions violations
  - Rules violations
Using transactions in stored procedures and triggers

- Duplicate key violations

Table 23-2 summarizes how rollback affects Adaptive Server processing in several different contexts.

**Table 23-2: How rollback affects processing**

<table>
<thead>
<tr>
<th>Context</th>
<th>Effects of rollback</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction only</td>
<td>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.</td>
</tr>
<tr>
<td>Stored procedure only</td>
<td>None.</td>
</tr>
<tr>
<td>Stored procedure in a transaction</td>
<td>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.</td>
</tr>
<tr>
<td>Trigger only</td>
<td>Trigger completes, but trigger effects are rolled back. Any remaining commands in the batch are not executed. Processing resumes at the next batch.</td>
</tr>
<tr>
<td>Trigger in a transaction</td>
<td>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.</td>
</tr>
<tr>
<td>Nested trigger</td>
<td>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.</td>
</tr>
<tr>
<td>Nested trigger in a transaction</td>
<td>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.</td>
</tr>
</tbody>
</table>

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.

Table 23-3 summarizes how a rollback caused by a duplicate key error or a rules violation affects Adaptive Server processing in several different contexts.

Table 23-3: Rollbacks caused by duplicate key errors/rules violations

<table>
<thead>
<tr>
<th>Context</th>
<th>Effects of data modification errors during transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction only</td>
<td>Current command is aborted. Previous commands are not rolled back, and subsequent commands are executed.</td>
</tr>
<tr>
<td>Transaction within a stored procedure</td>
<td>Same as above.</td>
</tr>
<tr>
<td>Stored procedure in a transaction</td>
<td>Same as above.</td>
</tr>
<tr>
<td>Trigger only</td>
<td>Trigger completes, but trigger effects are rolled back. Any remaining commands in the batch are not executed. Processing resumes at the next batch.</td>
</tr>
<tr>
<td>Trigger in a transaction</td>
<td>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.</td>
</tr>
<tr>
<td>Nested trigger</td>
<td>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.</td>
</tr>
<tr>
<td>Nested trigger in a transaction</td>
<td>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.</td>
</tr>
<tr>
<td>Trigger with rollback followed by an error in the transaction</td>
<td>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.</td>
</tr>
</tbody>
</table>
Using transactions in stored procedures and triggers

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:

```sql
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 Adaptive Server 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 Sybase system procedures (except for `sp_procmode`) that can be invoked by sessions using any transaction mode. If no transaction is active when you execute a system procedure, Adaptive Server turns off chained mode for the duration of the procedure. Before returning, it resets the mode to its original setting.

Adaptive Server 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.

**Warning!** When using transaction modes, be aware of the effects each setting can have on your applications.
Running system procedures in chained mode

Adaptive Server 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_procmode` to change the transaction mode to `anymode`:
  - `sp_addengine`
  - `sp_dropengine`
  - `sp_showplan`
  - `sp_sjobcontrol`
  - `sp_sjobcmd`
  - `sp_sjobcreate`


- `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, Adaptive Server 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` – Adaptive Server 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.

Setting 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

<table>
<thead>
<tr>
<th>procedure name</th>
<th>transaction mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>byroyalty</td>
<td>Any Mode</td>
</tr>
<tr>
<td>discount_proc</td>
<td>Unchained</td>
</tr>
<tr>
<td>history_proc</td>
<td>Unchained</td>
</tr>
<tr>
<td>insert_sales_proc</td>
<td>Unchained</td>
</tr>
<tr>
<td>insert_salesdetail_proc</td>
<td>Unchained</td>
</tr>
<tr>
<td>storeid_proc</td>
<td>Unchained</td>
</tr>
<tr>
<td>storename_proc</td>
<td>Unchained</td>
</tr>
<tr>
<td>title_proc</td>
<td>Unchained</td>
</tr>
<tr>
<td>titleid_proc</td>
<td>Unchained</td>
</tr>
</tbody>
</table>

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.
Using cursors in transactions

By default, Adaptive Server does not change a cursor’s state (open or closed) when a transaction ends through a commit or rollback. The SQL standards, however, 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, Adaptive Server provides the close on endtran option of the set command. In addition, if you set chained mode to on, Adaptive Server 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 occurring 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, Adaptive Server 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.
Issues to consider when using transactions

- 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.

Issues to consider when using transactions

Consider these issues 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 Adaptive Server fails during a long transaction, recovery time increases, since Adaptive Server must first undo the entire transaction.

- You can have as many databases in a user transaction as there are in your Adaptive Server installation. For example, if your Adaptive Server 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 Adaptive Server 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.
CHAPTER 23 Transactions: Maintaining Data Consistency and Recovery

- 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.

- Adaptive Server 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, Adaptive Server remembers the “next” log page (in the backward direction) containing these log records. During the post-commit phase, Adaptive Server 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 sylogs 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.

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 sylogs. This table is called the transaction log.

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 Adaptive Server 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 Adaptive Server from writing any changes and increases recovery time.

The Adaptive Server 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, Adaptive Server handles all aspects of the recovery process. Adaptive Server 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.

Setting a time limit on waiting for locks

Adaptive Server allows you to specify a lock wait period that determines how long a command waits to acquire 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.

`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.
Setting a time limit on waiting for locks

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.
CHAPTER 24  Locking Commands and Options

Setting a 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.

Setting a 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:
```

Transact-SQL Users Guide 745
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"
```

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.

Readpast locking for queue processing

Readpast locking is an option that is available for the select and readtext commands and the data modification commands update, delete, and writetext. It 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.

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.

### 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.

### Effects of isolation levels select queries with readpast

Readpast locking is designed to be used at transaction isolation level 1 or 2.

### Session-level transaction isolation levels and readpast

For data-only-locked tables, the effects of readpast on a table in a select command are shown in Table 24-1.

**Table 24-1: Session-level isolation level and the use of readpast**

<table>
<thead>
<tr>
<th>Session Isolation Level</th>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>0, read uncommitted (dirty reads)</td>
<td>readpast is ignored, and rows containing uncommitted transactions are returned to the user. A warning message prints.</td>
</tr>
<tr>
<td>1, read committed</td>
<td>Rows or pages with incompatible locks are skipped; no locks are held on the rows or pages read.</td>
</tr>
<tr>
<td>2, repeatable read</td>
<td>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.</td>
</tr>
</tbody>
</table>
Readpast locking for queue processing

<table>
<thead>
<tr>
<th>Session Isolation Level</th>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>3, serializable</td>
<td>readpast is ignored, and the command executes at level 3. The command blocks on any rows or pages with incompatible locks.</td>
</tr>
</tbody>
</table>

**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 `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.

**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.
**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.

**Readpast-locking examples**

The following examples illustrate readpast locking.

To skip all rows that have exclusive locks on them:

```sql
select * from titles readpast
```

To update only rows that are not locked by another session:

```sql
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:

```sql
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:

```sql
delete stores from stores readpast, authors
where stores.city = authors.city
```
APPENDIX A

The pubs2 Database

This appendix describes the sample database pubs2, which 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.

A diagram of the pubs2 database appears in Figure A-1 on page 761.

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, Sybase 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.

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.

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:

```sql
create table publishers
    (pub_id char(4) not null,
```
Tables in the pubs2 database

```
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]"
```

**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

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:

```sql
create default typedflt as "UNDECIDED"
sp_bindefault typedflt, "titles.type"
```

The pubdate column has this default:

```sql
create default datedflt as getdate()
sp_bindefault datedflt, "titles.pubdate"
```

titles uses this trigger:

```sql
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:

```sql
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
```

**titleauthor table**

The titleauthor table shows the author ID, title ID, and royalty of titles by percentage.

titleauthor is defined as:

```sql
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:

```sql
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
```

**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
```
Tables in the pubs2 database

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][0-9][0-9]\" or`  
  - `@title_id like "[MT]C[0-9][0-9][0-9][0-9][0-9]\" or`  
  - `@title_id like "P(SC)_[0-9][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 inserted`  
      - `where titles.title_id = inserted.title_id)`  
    - `where title_id in (select title_id from inserted)`
The sales table contains store IDs, order numbers, and dates of sales.

`sales` is defined as:

```sql
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`:

```sql
sp_primarykey sales, stor_id, ord_num
```

Its `stor_id` column is a foreign key to `stores`:

```sql
sp_foreignkey sales, stores, stor_id
```

The stores table contains the names, addresses, ID numbers, and payment terms for stores.

`stores` is defined as:

```sql
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`:

```sql
sp_primarykey stores, stor_id
```
**royshed table**

The roysched table contains royalties, defined as a percentage of price.

royshed is defined as:

```sql
create table roysched
    title_id tid not null,
    lorange int null,
    hirange int null,
    royalty int null)
```

Its primary key is `title_id`:

```sql
sp_primarykey roysched, title_id
```

Its `title_id` column is a foreign key to `titles`:

```sql
sp_foreignkey roysched, titles, title_id
```

Its nonclustered index for the `title_id` column is defined as:

```sql
create nonclustered index titleidind
    on roysched (title_id)
```

**discounts table**

The discounts table contains discounts in stores.

discounts is defined as:

```sql
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`:

```sql
sp_primarykey discounts, discounttype, stor_id
```

Its `stor_id` is a foreign key to `stores`:

```sql
sp_foreignkey discounts, stores, stor_id
```

**blurbs table**

The blurbs table contains sample blurbs for books.
blurbs is defined as:

```sql
create table blurbs
(au_id int not null,
copy text null)
```

Its primary key is au_id:

```sql
sp_primarykey blurbs, au_id
```

Its au_id column is a foreign key to authors:

```sql
sp_foreignkey blurbs, authors, au_id
```

**au_pix table**

The author_pix table contains photographs of authors in the pubs2 database.

au_pix is defined as:

```sql
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:

```sql
sp_primarykey au_pix, au_id
```

Its au_id column is a foreign key to authors:

```sql
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 Sybase stores image data. Sybase 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.
Diagram of the pubs2 database

Figure A-1 shows the tables in the pubs2 database and some of the relationships among them.
Figure A-1: Diagram of the pubs2 database
Diagram of the pubs2 database
The *pubs3* Database

This appendix describes the sample database *pubs3*, which 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.

A diagram of the *pubs3* database appears in Figure B-1 on page 771.

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, Sybase 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.

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.

**publishers table**

The *publishers* table contains the publisher ID, and name, city, and state. *publishers* is defined as:

```sql
create table publishers
    (pub_id char(4) not null,
```
Tables in the pubs3 database

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]"

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

titles table

The titles table contains the name, title ID, type, and other information about titles.

titles is defined as:

```sql
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,
um_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:

```sql
create nonclustered index titleind
on titles (title)
```

Its title_idrule is defined as:

```sql
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:

```sql
create default typedflt as "UNDECIDED"
sp_bindefault typedflt, "titles.type"
```

The pubdate column uses this default:

```sql
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

**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

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
Tables in the pubs3 database

`@title_id like "[A-Z][A-Z]yyyy"`

`salesdetail` uses this trigger:

```sql
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)
```

**sales table**

The `sales` table contains the store ID, order numbers, and dates of sales.

`sales` is defined as:

```sql
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))
```
stores table

The stores table contains the store ID, store name, and other information about stores.

stores is defined as:

```sql
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))
```

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:

```sql
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))
```
Tables in the pubs3 database

**roysched table**

The roysched table contains title ID, royalty percentage, and other information about title royalties.

roysched is defined as:

```sql
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:

```sql
create nonclustered index titleidind
    on roysched (title_id)
```

**discounts table**

The discounts table contains the discount type, store ID, quantity, and percentage discount.

discounts is defined as:

```sql
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)
```

**blurbs table**

The blurbs table contains the author ID and blurb for books in the pubs3 database.

blurbs is defined as:

```sql
create table blurbs
    (au_id id not null
    references authors(au_id),
    copy text null)
```
Diagram of the *pubs3* database

Figure B-1 shows the tables in the *pubs3* database and some of the relationships among them.

*Figure B-1: Diagram of the pubs3 database*
Diagram of the pubs3 database
Symbols

& (ampersand)
  “and” bitwise operator 20

* (asterisk)
  multiplication operator 19
  pairs surrounding comments 509
  select and 38

*= (asterisk equals) outer join operator 125, 154

*/ (asterisk slash), comment keyword 509

@ (at sign)
  local variable name 511
  procedure parameters and rule arguments 471

@@ (at signs), global variable name 516

\ (backslash)
  character string continuation with 24

^ (caret)
  “exclusive or” bitwise operator 20

, (comma)
  in default print format for money values 202

$ (dollar sign)
  in identifiers 13
  in money datatypes 234

-- (double hyphen) comments 7, 510

=* (equals asterisk) outer join operator 125, 154

= (equals sign)
  comparison operator 22

> (greater than)
  comparison operator 22
  range specification 57

>= (greater than or equal to) comparison operator 22

< (less than)
  comparison operator 22
  range queries and 57

<= (less than or equal to) comparison operator 22

- (minus sign)
  arithmetic operator 19
  for negative monetary values 234

!= (not equal to) comparison operator 22

<> (not equal to) comparison operator 22

!> (not greater than) comparison operator 22

!< (not less than) comparison operator 22

() (parentheses)
  in arithmetic statements 45
  in expressions 18
  with union operators 114

% (percent sign)
  arithmetic operator (modulo) 19

| (pipe)
  “or” bitwise operator 20

+ (plus)
  arithmetic operator 19
  null values and 71
  string concatenation operator 21, 521

# (pound sign), temporary table identifier prefix 290, 297, 299

£ (pound sterling sign)
  in identifiers 13
  in money datatypes 234

“” (quotation marks)
  comparison operators and 22
  enclosing column headings 41–42
  enclosing empty strings 23, 240
  enclosing parameter values 559
  enclosing values 204, 228, 229
  in expressions 23
  literal specification of 23

/ (slash)
  arithmetic operator (division) 19

/* (slash asterisk), comment keyword 509

~ (tilde)
  “not” bitwise operator 20

¥ (yen sign)
  in identifiers 13
  in money datatypes 234

@@error global variable 516
Index

Numerics
“0x” 234

counted in textsize 47

A
abbreviations
date parts 532
out for output 579

abstract plan derived tables
differences from SQL derived tables 365
accounts, server.
See logins; users
adding
column data with insert 238, 247–249
foreign keys 660
IDENTITY column to a table 336
rows to a table or view 237–249
timestamp column 651
user-defined datatypes 219, 220
users to a database 282

addition operator (+) 19

administrative instructions and results 2

aggregate functions 75–80, 524–530
See also row aggregates; individual function names
compute clause and 26, 104–111
cursors and 619
datatypes and 77
difference from row aggregates 528
distinct keyword and 76, 79
group by clause and 82–96, 526
on multiple columns 110
nesting 89
null values and 80
scalar aggregates 77, 525
subqueries including 177
vector aggregates 83, 525
views and 435
where clause, not permitted 77

aggregate functions and cursors 525
aggregate functions, in order by clause 101
aliases
table correlation names 54

all keyword
comparison operators and 179, 189

group by 92
searching with 180
select 51–53
subqueries including 24, 181, 189
union 114

all tables marked for incremental data transfer except
system tables and worktables 263
allow nested triggers configuration parameter 674–677
allow_dup_row option, create index 456
alter database command 287
See also create database command
alter table
adding a column 327–331
adding columns with user-defined datatypes 340
adding columns, effect on column IDs 329–330
adding constraints 330–331
adding IDENTITY columns 336
adding not-null columns 330
and CIS 329
and dumping the transaction log 328
bulk copying existing dumps into modified columns 333
changing exp_row_size during data copy 339
command 327
converting datatypes 333
data copying 338, 339
decreasing column length truncates data 333
dropping columns 331–332
dropping columns with user-defined datatypes 341
dropping columns, effect on column IDs 331–332
dropping constraints 332
dropping IDENTITY columns 337–338
error messages 341–343
errors generated by alter table modify 342
exclusive table lock 328
execute immediate statement in 343
modifying columns 332–336
modifying columns with precision 335
modifying columns with scale 335
modifying columns with user-defined datatypes 341
modifying datetime columns 334
modifying locking scheme 340
modifying NULL default values 334, 335
modifying text and image columns 335
objects that perform a select * 328
of table with a clustered index 340
on remote tables 329
restrictions for dropping IDENTITY columns 337–338
when it performs a data copy 339
with arithabort numeric_truncation turned on or
off 335
alter table command
adding timestamp column 651
altering data partitions 401
altering.
See changing and (&) bitwise operator 20
and keyword
in expressions 25
in joins 127
in search conditions 72, 73
ansinull option, set 9
any keyword
in expressions 24
searching with 179
subqueries using 182–184, 189
applications, data decision support (DDS) 349
approximate numeric datatypes 202
arguments
text functions 523
arithabort option, set
arith_overflow and 8, 545
mathematical functions and numeric_truncation 545
arithignore option, set
arith_overflow and 9, 545
arithmetic errors 8
arithmetic expressions 18, 41
not allowed with distinct 79
arithmetic operations 76
mixed mode 216–219
arithmetic operators 42
in expressions 19, 41, 349
precedence of 72
ascending order, asc keyword 100
asterisk (*)
multiplication operator 19
pairs surrounding comments 509
select and 38
in subqueries with exists 189
at sign (@)
local variable name 511
procedure parameters and rule arguments and 471
@@error global variable
select into and 324
@@identity global variable 244, 293, 516
@@isolation global variable 726
@@nextlevel global variable
nested procedures and 566
nested triggers and 674
@@rowcount global variable
 cursors and 633
triggers and 659
@@sqlstatus global variable 518
@@textsize global variable 46
@@trancount global variable 517, 719
@@transstate global variable 517, 717, 719
au_pix table, pubs2 database
author blurbs table
 pubs2 database 758
 pubs3 database 770
authors table
 pubs2 database 752
 pubs3 database 764
auto identity database option 294
 identity in nonunique indexes and 449
automatic operations
chained transaction mode 721
datatype conversion 215, 534
hidden IDENTITY columns 294
triggers 653
avg aggregate function 76
See also aggregate functions
as row aggregate 107

B
backing up.
See recovery
backslash (\) for character string continuation 24
base date 233
Index

base tables.
   See tables
bases
    choosing 283
batch processing 479
   control-of-flow language 26, 479, 480, 485–510
   errors in 481, 484
   go command 484
   local variables and 502
   rules for 480–483
   submitting as files 484, 485
bcp (bulk copy utility)
   IDENTITY columns and 247
begin transaction command 715
begin...end commands 499
between keyword 56, 57
   check constraint using 315
bigint datatype 200
biginttohex function 541
   binary datatype 209
   See also datatypes
like and 60
   binary datatypes 209, 210
   “0x” prefix 234
   concatenating 520
   operations on 520–523
binary expressions
   concatenating 21, 521, 522
   binary representation of data for bitwise operations 19
   binding
      defaults 466–468
      rules 471, 473
   bit datatype 210
   See also datatypes
   bitwise operators 19–21
blanks
   character datatypes and 205
   in comparisons 22, 55, 65
   empty string evaluated as 23
   like and 65
blurbs table
   pubs2 database 758
   pubs3 database 770
Boolean (logical) expressions 18
   select statements in 487
branching 503
browse mode 650–652
   cursor declarations and 651
timestamp datatype and 211
built-in functions 524
   conversion 534
date 532, 532–533
image 523
   mathematical 531
security 546
string 520–523
text 523
type conversion 534
views and 426
by row aggregate subgroup 528
bytes
   composite index limit 448
datatype storage 198, 289
delimited identifier limit 14
   hexadecimal digits and 234
   identifier limit 11
output string limit 505
print messages limit 504
   quoted column limit 42
retrieved by readtext 47
subquery limit 165
temporary table name limit 12
C
   calculating dates 533
Cartesian product 126
cascading changes (triggers) 654, 661
case expressions 488–499
   when... then keyword 494
coalesce 497
data representation 489
determining datatype 492
division by zero avoidance 489
   search conditions in 494
stored procedure example 558
   value comparisons and 496
case-insensitive format, presenting queries in 349
case-sensitivity 13
   in comparison expressions 22
chained mode, running system procedures in 737
chained option, set 722
chained transaction mode 7, 721
changing
  See also updating
database size 287
default database 32
index names 344
object names 344–345	
tables 327–345
view definitions 433
changing data.
  See data modification
char datatype 204–205
  See also character data; datatypesente rule 228
  in expressions 23
like and 60
character data 203
  See also individual character datatype names
avoiding “NULL” in 240
  entry rules 228
  operations on 520–523
  trailing blanks in 205
character datatypes 204
character expressions 18
character sets 10
  conversion errors 13
  iso_1 13
character strings
  continuation with backslash (\) 24
  empty 23
  matching 60
  select list using 42
  specifying quotes within 23
  truncation 8
characters
  0x 540
  special 10
  wildcard 61–66, 561
check constraints 308, 315
checkpoint command 742
CIS RPC mechanism 593
classes 2
client
cursors 615
close command 636
close on endtran option, set 739
closing cursors 623
clustered constraint
  create index 453
  create table 312
clustered index
  not with function-based index 448
  on computed column, creating 350, 452
  with computed columns 448
clustered indexes 453–454
  See also indexes
  integrity constraints 311
coaesce keyword, case 497
column length
  decreasing with alter table 333
column name
  in parentheses 528
column names 15
  qualifying in subqueries 166
column pairs.
  See joins; keys
column-level constraints 308
columns
  See also database objects; select command
  adding data with insert 238, 247–249
  adding with alter table 327–331
  defaults for 310, 466–468
  dropping columns with user-defined datatypes 341
  dropping with alter table 331–332
  gaps in IDENTITY values 306–307
group by and 88
IDENTITY 291–294
indexing more than one 447
initializing text 257
joins and 122, 125
length definition 297
modifying columns with user-defined datatypes 341
modifying with alter table 332–336
null values and check constraints 316
null values and default 296
numeric, and row aggregates 528
order in insert statements 238, 247
order in select statements 40
qualifying names in subqueries 166
rules 471
rules conflict with definitions of  297, 473  
system-generated  291  
variable-length  296  
comma (,)  
default print format for money values  202  
command  
  count_big  76  
  group_big  76  
  options to create table for virtually-hashed tables  417  
  select into command  446  
sp_dboption stored procedure, using to create index  446  
command terminator  32  
commands  2  
See also individual command names  
not allowed in user-defined transactions  714  
comments  
  ANSI style  7  
  double-hyphen style  510  
in SQL statements  509  
commit command  715  
common keys  
  See also foreign keys; joins; primary keys  
comparing values  
in expressions  22  
for joins  125  
null  68, 514  
timestamp  652  
comparison operators  55–56  
  See also relational expressions  
  correlated subqueries and  195–196  
in expressions  21  
modified, in subqueries  179, 180  
null values and  67, 514  
symbols  22  
unmodified, in subqueries  176–178  
compatibility, data  
create default and  465  
complex datatypes, composing and decomposing  345, 348  
Component Integration Services  
alter table command on remote tables  329  
and update statistics command  460  
aromatic IDENTIFY columns  294  
connecting to a server  32  
described  30  
image datatype  210  
joins  119  
text datatype  208  
transactions and  712  
triggers  678  
composing, decomposing complex datatypes  345  
composite  
  indexes  447  
  partitioning key columns  383  
computations.  
See computed columns  
compute clause  104–112  
different aggregates in same  110  
grand totals  111–112  
multiple columns and  108, 110  
not allowed in cursors  625  
row aggregates and  107–110  
subgroups and  108  
subtotals for grouped summaries  104  
union and  117  
using more than one  109  
compute clause and row aggregates  527  
computed column  448  
defined by expression  349  
deterministic  346  
deterministic property  351  
differences from function-based index  347  
index  345  
indexability  346  
materialized or nonmaterialized  346  
two types  351  
computed columns  42, 345, 436  
indexable and deterministic  347  
insert  248  
with null values  43  
update  254  
views and  426, 436  
XML documents, mapping  348  
computing dates  533  
concatenation  521  
binary data  520  
using + operator  21, 521  
using + operator, null values and  71  
connections  
transactions and  741  
consistency  
transactions and  711
Index

constants
  in expressions 41
constraints 281, 307
  adding with \texttt{alter table} 330–331
check 308, 315
  column-level 308
  \texttt{default} 308
  dropping with \texttt{alter table} 332
primary key 308, 311
  referential integrity 308, 312
  table-level 308
unique 308, 311
  with NULL values 295
  with rules 473
continuation lines, character string 24
control-break report 104
control-of-flow language 26, 28
conventions
  in identifiers 11
  naming 10, 18
  Transact-SQL 10, 18
conversion
  between character sets 13–14
  datatypes 297
    implicit 23, 215, 534
    lower to higher datatypes 23
  string concatenation 21
\texttt{convert} function 215, 537
  concatenation and 21, 522
converting hexadecimal numbers 540
copying
  data with \texttt{insert...select} 248
  rows 249
  tables with \texttt{bcp} 246
  tables with \texttt{select into} 321
correlated subqueries 193–196
  comparison operators in 195
  correlation names and 194
  \texttt{exists} and 189
  \texttt{having} clause in 196
correlation names
  in joins (ANSI syntax) 138–140
  self-joins 130
SQL derived tables and 368
subqueries using 167, 194
table names 54, 130

\texttt{count} aggregate function 76
  See also aggregate functions
  on columns with null values 80, 90
  as row aggregate 107
\texttt{count(*)} aggregate function 75–76, 78, 107
  See also aggregate functions
  on columns with null values 90
  including null values 80
\texttt{count_big} command 76
\texttt{create database} command 283–287
  batches using 481
\texttt{create default} command 465–466
  batches and 480
  \texttt{create procedure} with 580
\texttt{create index} command 446–457
  batches using 481
  \texttt{ignore_dup_key} 449
\texttt{create procedure} command 553–554, 606
  See also stored procedures; extended stored
  procedures (ESPs)
  batches using 480
  null values and 560
  \texttt{output} keyword 575–580
  rules for 580
  with recompile option 565
\texttt{create rule} command 470
  batches using 480
\texttt{create procedure} with 580
\texttt{create schema} command 314
\texttt{create table} command 289–291
  batches using 481
  composite indexes and 447
  constraints and 307
  in different database 290
  example 291, 318
  null types and 295
  null values and 240
  in stored procedures 581
  user-defined datatypes and 220
\texttt{create trigger} command 655–657
  batches using 480
  \texttt{create procedure} with 580
  displaying text of 683
\texttt{create view} command 424
  batches using 481
  \texttt{create procedure} with 580
Index

union prohibited in 117
creating databases 283–286
 datatypes 219, 223
defaults 465–466, 469
 indexes 446–454
 rules 470
 stored procedures 571–580
tables 318–327
temporary tables 290, 297–299
triggers 655–657
 virtually-hashed tables 412, 413
creating clustered index 350, 452
creating index on function or expression 452
CS_DATAFMT structure 598
CS_SERVERMSG structure 598
currency symbols 234
current database changing 283
cursor result set 614, 617
cursor rows option, set 632
cursors buffering client rows 633
close 636
deallocating 636
 declaring 620
deleting rows 635
 execute 615
 fetching 628–633
 fetching multiple rows 632
for browse and 651
Halloween problem 620
language 615
 locking and 645
 name conflicts 616
 nonunique indexes 617
number of rows fetched 633
opening 627
position 614
read-only 618
scans 617
scope 616, 616
server 615
status 630
 stored procedures and 643
subqueries and 165
transactions and 739–740
unique indexes 617
updatable 618
updating rows 634
variables 630
cursors and aggregate functions 525
cursors, scrollable 613
custom datatypes.
 See user-defined datatypes
 customizing data presentation 348
D
data copying 338, 339
 when alter table performs a data copy 339
data decision support (DDS) applications 349
data dependency.
 See dependencies, database object
data dictionary.
 See system tables
data integrity 27, 280, 470
 See also data modification; referential integrity
 methods 307
Data Manipulation Language (DML), materializing
 ordered data with 349
data modification 2, 420
 permissions 226
 text and image with writetext 257–258
 update 253, 256
 views and 435
Data partitions 379
data partitions
 adding 402
 altering 401
 creating 395, 398
 partitioning key 402
data presentation, customizing 348
data transfer, incremental, overview 262
data transforming, using sortkey() 349
database devices 285
database integrity.
 See data integrity; referential integrity
database object owners
 names in stored procedures 581
database objects 279
See also individual object names
dropping 481
renaming 344–345
restrictions for alter table 328
stored procedures and 580, 581, 582
Database Owners
adding users 282
transferring ownership 283
databases 282–288
See also database objects
adding users 282
creating user 283–286
default 30
dropping 288
help on 356
joins and design 121
name 284
number of server 284
optional 282
ownership 283
size 285, 287
system 282
use command 283
user 282
datatype
bigint 200
complex, XML 350, 452
entry format 229
image 350, 452
indexable 350, 452
Java class 350, 452
text 350, 452
unsigned bigint 200
unsigned int 200
unsigned smallint 200
datatype conversions 534
automatic 215
binary and numeric data 541
bit information 541
case expressions 492
character information 537, 538
column definitions and 297
date and time information 539
domain errors 546
functions 541
hexadecimal-like information 540
image 541
money information 539
numeric information 538, 540
overflow errors 544
rounding during 539
scale errors 545
datatype precedence.
See precedence
datatype, definition of 197
datatypes 197–216
aggregate functions and 77
altering columns with user-defined 340–341
approximate numeric 202
character 203
converting with alter table 333
create table and 289, 295, 447
creating 219, 223
datetime values comparison 22
defaults and 221, 466–468
dropping columns with user-defined 341
date parts 231, 532
entry rules 201, 228–237
entry format 229
integer 200
joins and 125
length 220
local variables and 511
mixed, arithmetic operations on 19
modifying columns with user-defined 341
money 202
rules and 221, 471–473
summary of 198–200
temporal tables and 298
unichar 205
union 112
views and 424
datatypes, custom.
See user-defined datatypes
date datatype 229
date formats 532, 533
date parts 231, 532
abbreviation names and values 532
datediff function 533
dateformat option, set 231–233
datename function 233
Index

datepart function 233
dates
  See also time values
  acceptable range of 229
  calculating 533
  comparing 22, 55
  display formats 204, 532
  entry formats 203, 229–234
  functions for 532, 532–533
  like and 233
  searching for 233
datetime datatype 203, 229, 532
  See also dates; timestamp datatype
  comparison of 22
  concatenating 522
  like and 60
datetimedelta datatype
  operations on 532
day date part 533
dayofyear date part abbreviation and values 533
dbcc (database consistency checker)
  stored procedures and 581
  dbcc checkstorage 417
  dbcc checktable for virtually-hashed tables 417
  DB-Library programs
  transactions and 735
  dd
  See day date part
  DDS (data decision support) applications 349
deallocate cursor command 636
deallocating cursors 624, 636
debugging aids 28
decimal datatype 235
declare command 511
declare cursor command 625
  declaring
  cursors 620, 621
  local variables 511–515
  parameters 556
default database 30
default database devices 285
default keyword 310
  create database 285
default parameters in stored procedures 560
  default settings
  databases 32
date display format 204
  language 30, 231, 233
  parameters for stored procedures 558–561
  print format for money values 202
default values
  datatype length 204, 209
  datatype precision 201
  datatype scale 201
  datatypes 220
defaults 28, 464–465
  See also database objects
  binding 466–468
  column 241
  creating 465–466, 469
datatypes and 221, 466–468
  dropping 469
  insert statements and 238
  naming 465
  null values and 320, 469
  sharing inline 476–478
  unbinding 468
defining local variables 511–515
definition, of deterministic property 351
delayed execution (waitfor) 508–509
delete command 276–277, 437
  See also dropping
  cursors and 635
  multitable views and 435
  subqueries and 173
  triggers and 656, 658–659, 661–662
  views and 435
deleted table 658–659
deleting
  cursor rows 635
  cursors 623
  rows 276–277
  views 440
delimited identifiers 7, 14
dependencies
  database objects 587
  display 442
dependent
tables 661
  views 433
derived column list
  and SQL derived tables 368
in a derived table expression 368
derived table
part of view 368
referred in cursors 368
derived table expression
aggregate functions and 373
constant expressions and 371
correlated attributes and 375
correlated SQL derived tables and 369
creating a table from 374
defining a SQL derived table 365
derived column lists 368
differences from create view command 368
joins and 373
nesting 370
renaming columns 371
See also SQL derived tables 365
subqueries and 192, 370
syntax 367
union operator and 370
views and 374
derived table expression See also SQL derived tables
derived tables
SQL derived tables 365
descending order (desc keyword) 100
designing tables 318
detail tables 657
deterministic
function-based index 448
deterministic principle
function-based index 448
deterministic property 345
computed column 346
definition 351
function-based index 346
in computed columns 351
in function-based indexes 351
deterministic property in function-based indexes 355
deterministic property, examples 352
deterministic property, used by function-based index 448
devices 285
See also sysdevices table
diagram
pubs2 database 761
pubs3 database 771
difference (with exists and not exists) 191–192
dirty reads 723
See also isolation levels
discouts table
pubs2 database 758
pubs3 database 770
disk crashes.
See recovery
distinct keyword
aggregate functions and 76, 79
cursors and 618
expression subqueries using 178
order by and 103
row aggregates and 107
select 51–53
select, null values and 53
distribution pages 277
division operator (/) 19
DLL (dynamic link library).
See dynamic link libraries
DLLs 606
DML, Data Manipulation Language 349
dollar sign ($) in identifiers 13
in money datatypes 234
double precision datatype 202
double precision datatype entry format 235
drop command
in batches 481
drop database command 288
drop default command 469
drop index command 457
drop procedure command 564, 583, 608
drop rule command 475
drop table command 345
drop trigger command 682
drop view command 440
dropping
See also delete command; individual drop commands
databases 288
defaults 469
indexes 457
objects 481
primary keys 661–662
procedures 583
rows from a table 276–277
rules 475
system tables 345
tables 345, 434
triggers 682
views 434, 440
dump, database 742
duplicate key errors, user transaction 735
duplicate rows
indexes and 456
removing with union 114
dw.
See weekday date part
dy
See dayofyear date part
dynamic dumps 742
dynamic Link Libraries
for extended stored procedures 603, 606
sample definition file 605
search order 603
UNIX makefile 604
dynamic link libraries
building 603–606

E
e or E exponent notation
approximate numeric datatypes 235
money datatypes 234
eligible tables for IDT 263, 269
else keyword.
See if...else conditions
embedding join operations 120
empty string (""") or ("")
not evaluated as null 240
as a single space 23
empty string (""") or ("") 205
encryption data 5
end keyword 494, 499
enhancements to SQL 25–29
equal to.
See comparison operators
equijoins 124, 127
@@error global variable
select into and 324
error handling 28
error messages
12205 745
12207 744
constraints and 309
lock table command 744
numbering of 506
set lock wait 745
severity levels of 506
user-defined transactions and 740
errorexit keyword, waitfor 508
errors
arithmetic overflow 544
in batches 481–484
convert function 537–546
divide-by-zero 544
domain 546
duplicate key 735
return status values 572–580
scale 545
triggers and 675
in user-defined transactions 735
escape characters 64
esp execution priority configuration parameter 596
esp unload dll configuration parameter 594, 596
ESPs.
See extended stored procedures
European characters in object identifiers 13
deterministic property 352
examples, virtually-hashed tables 414
execute command 17
for ESPs 608
output keyword 575–580
with recompile option 566
execute cursors 615
execution
extended stored procedures 595
exceptions and errors in IDT 269
exists keyword 191–192, 487
search conditions 188
explicit null value 240
explicit transactions 722
explicit values for IDENTITY columns 243

784

Adaptive Server Enterprise
expression
contains functions, arithmetic operators, case expressions, global variables 349
defining computed column 349
used by function-based index 448
expression subqueries 178
expressions 76
concatenating 521
converting datatypes 534
definition of 18
including null values 71
replacing with subqueries 174
types of 18
extended stored procedures 27, 591–611
creating 606, 607
creating functions for 597, 606
DLL support for 593
example 594
exceptions 611
executing 608
freeing memory of 596
function example 599
messages from 611
naming 594
Open Server support for 594
performance impact of 596
permissions on 596
priority of 596
removing 608
renaming 608
extensions, Transact-SQL 10, 29–30
fixed-length columns
binary datatypes for 209
character datatypes for 204
compared to variable-length 205
null values in 296
float datatype 202
See also datatypes
entry format 235
floating-point data
See also float datatype; real datatype
for browse option, select 117
not allowed in cursors 625
for load option
alter database 287
create database 286
for read only option, declare cursor 618, 626
for update option, declare cursor 618, 626
foreign key constraint 313
foreign keys 657–658
inserting 660
sp_help report on 360
updating 667
format strings
print 504
from keyword 53
delete 276
joins 123
SQL derived tables 367
update 255
front-end applications, browse mode and 650
full name 31
function
sortkey 349
sybesp_dll_version 593
function-based index 345
always materialized 448
contain expressions 448
creating and using 448
deterministic property 346, 351, 448
deterministic property of 355
different from computed column 347
global variables 448
indexability 346
materialized 346
must be deterministic 448
not clustered 448

F
FALSE, return value of 188
fetch command 628
fetching cursors 622
fields, data.
See columns
files
batch 484–485
finding
object dependencies 345
FIPS flagger 6
fipsflagger option, set 6

Transact-SQL Users Guide
Index

used for 448

functions
aggregate 524
biginttohex 541
conversion 534
date 532, 532–533
getdate 347
image 523
in expressions 349
mathematical 531
security 546
sortkey 349
sp_version 580
string 520–523
text 523
views and 426

futureonly option
defaults 467, 468
rules 472, 474

G

gaps in IDENTITY column values 301–307
generalized index keys 452
getdate function 347
global indexes 385
creating 399
definition of 380
global variable
@@error 516
global variables 516–518, 581
See also individual variable names
global variables, in expressions 349
go command terminator 32, 479
goto keyword 503
grand totals
compute 111–112
order by 100
greater than.
See comparison operators
group by clause 82–89, 618
aggregate functions and 82–96, 105
all and 92–93
correlated subqueries compared to 195–196
having clause and 94–96
multiple columns in 88
nesting 89
null values and 89
order by and 102
subqueries using 178–179
triggers using 668–670
union and 117
where clause and 91–92
without aggregate functions 82, 87
group by clause and aggregate functions 526
group_big command 76
grouping
See also user-defined transactions
procedures 734
procedures of the same name 564
groups
database users 31
discretionary access control and 31
guest users 281, 282

H

Halloween problem 620
hash
key use in virtual hashing 411
table, virtually-hashed 411
virtually-hashed table, structure 412
hash partitioning 381
hash_factor, determining values 413
having clause 94–99
different from where clause 94
group by and 94
logical operators and 95
subqueries using 178–179, 196
union and 117
without aggregates 95
without group by 99
headings, column 40–42
help reports
See also individual system procedures
columns 475
database devices 285
database object 356–360
databases 356
datatypes 356–360
Index

defaults 475
dependencies 587
indexes 458
rules 475
system procedures 585–589
text, object 586
triggers 683
hexadecimal numbers
“0x” prefix for 47, 234
converting 541
hexadecimal numbers, converting 540
hextobigint function 541
hextoint function 215
hh.
See hour date part hierarchy
See also precedence
datatype 216–218
operators 18
holdlock keyword 625, 711
cursors and 646
hour date part 533
hyphens as comments 510

identifiers 7, 10
delimited 7, 14
quoted 14
user-defined and other 11
IDENTITY columns 291–294
@@identity global variable 516
adding 336
and referential integrity 293
Component Integration Services 294
creating tables with 291
datatype of 292
deleting with syb_identity keyword 277
dropping 337–338
explicit values for 243
gaps in values 246, 306–307
gaps, eliminating 246
inserting values into 243, 439
nonunique indexes 449
renumbering 246
reserving block of 244
restrictions for dropping 337–338
selecting 294, 325
syb_identity in views 428
syb_identity keyword 294
unique values for 243
updating with syb_identity keyword 256
user-defined datatypes and 292
using with select into 325–327
views and 428–429, 439
@@identity global variable 244, 293, 516
identity grab size configuration parameter 244
identity in nonunique index database option 449, 617
identity keyword 292
IDENTITY property for user-defined datatypes 221
identity_insert option, set 243
IDT
eligible tables for 263, 269
exceptions and errors 269
marking tables for, when table created or later using
alter table 263, 269
new tables spt_TableTransfer 266
new tables spt_TableTransfer, monTableTransfer 266
overview 262
transfer failure, reasons for 269
if update clause, create trigger 678–679
if...else conditions 486–488, 501
case expressions compared to 488
ignore_dup_key option, create index 449, 455
ignore_dup_row option, create index 456
image
complex datatype 348
image datatype 210
See also datatypes
‘0x’ prefix for 234
changing with writetext 257
Component Integration Services 210
initializing with null values 297
inserting 237
null values in 297
prohibited actions on 101, 102
selecting 46–50
selecting in views 432
subqueries using 164
triggers and 678
### Index

- **union** not allowed on 117
- updating 253
- **writext** to 436
- image functions 523
- implicit conversion (of datatypes) 23, 215, 534
- implicit transactions 721
- **in** keyword 58–59
  - check constraint using 315
  - in expressions 24
  - subqueries using 183, 185–187
- incremental data transfer
  - marking tables for 263
- incremental data transfer (IDT)
  - enabled when database license purchased and registered 262
  - marking tables for 263
- incremental data transfer (IDT), overview 262
- incremental data transfer, overview 262
- index keys, generalized 452
- index on computed column, creating 350, 452
- index on function or expression, creating 452
- index partitions 379
  - creating 398
- index scans, hash-based 411
- indexability
  - computed column 346
  - function-based index 346
- indexes
  - See also clustered indexes; database objects
    - composite 447
    - creating 446–454
    - distribution pages 277
    - dropping 457
    - duplicate values and 455
    - guidelines for 445–446
    - IDENTITY columns in nonunique 449
  - integrity constraints 311
  - joins and 446
  - key values 453
  - leaf level 453, 454
    - on multiple columns 447
  - naming 15
  - nonclustered 453–454
  - options 455–457
  - on presorted data 457
  - on primary keys 446, 453
  - renaming 344
  - retrieval speed and 445, 446, 453
  - searching 446
  - space used by 362
  - unique 449, 455
  - views and 424
- individual object names
  - infected processes 508
  - information messages (Server)
    - See error messages; severity levels
  - inline defaults, sharing 476–478
  - inner joins 140–143
    - join table (ANSI syntax) 142
    - nested (ANSI syntax) 142
    - on clause (ANSI syntax) 142–143
  - inner queries
    - See subqueries
  - inner tables
    - predicate restrictions (ANSI syntax) 146–148
  - in-row off-row large objects 685–694
  - ins
    - updates using 255
  - **insert** command 237–249, 437
    - batches using 481
    - duplicate data from 455, 456
    - IDENTITY columns and 242
    - image data and 210
    - null/not null columns and 240
    - rules and 464
  - select 237
    - subqueries and 173
    - text data and 208
    - triggers and 656, 658–659, 660
    - union operator in 117
    - views and 435
  - inserted table 658–659
  - inserting rows 237–249
  - installing
    - sample image data in pubs2 759
  - installmaster script 584
  - int datatype 236
    - See also integer data; smallint datatype; tinyint datatype
  - integer data 200
    - See also individual datatype names
  - integer datatypes, converting to 540

---

**788**

*Adaptive Server Enterprise*
Index

Component Integration Services 119
correlation names (ANSI syntax) 138–140
correlation names and 130
equijoins 124, 127
from clause in 123
help for 161
indexes and 446
inner joins (ANSI syntax) 138, 140–143
joined table 138
left join 136
many tables in 134–135
natural 127
not-equal 129, 131–134
null values and 68, 160
operators for 124, 125, 129
outer 125, 135–157
outer joins (ANSI syntax) 143–154
process of 120, 126
relational model and 121
relational operators and 124
restrictions 125
right join 136
select list in 121–123
selection criteria for 128
self-joins 130–131
self-joins compared to subqueries 167
subqueries compared to 186–187
theta 125
used with views 137, 427
views and 427
where clause in 124, 126, 128

J
Japanese character sets 204
object identifiers and 13
Java classes
complex datatype 348
Java objects, in expressions 349
joined table 138
joins 4
column names in 125
column order in results 122
comparison operators 129

K
key values 453
keys, table
See also common keys; foreign keys; primary keys
views and 424
keywords 10
control-of-flow 485–510
new 9
phrases 2
Index

L
labels 503
language cursors 615
language defaults 30, 231, 233
language option, set 231, 233
languages, alternate
  effect on date parts 231, 233
leaf levels of indexes 453, 454
less than.
  See comparison operators
levels
  nested transactions 719
  @@trancount global variable 517, 719
  transaction isolation 721–726
like keyword 60–66
  check constraint using 315
  searching for dates with 233
limitations, virtually-hashed tables 416
lines (text), entering long 24
list partitioning 382
list, matching in select 58–59
listing
  database objects 362
  datatypes with types 216
literal character specification
  quotes (“ ”) 23
literal values
  null 71
load
  rebuild indexes 458
  loading table data, partitions 408
LOB locators 211
  convert locators to LOB value 214
  creating 212
  datatypes supported 212
  declare local variables 212
  explicitly creating 213
  implicitly creating 213
  in T-SQL statements 211
  scope of locators 215
local and remote servers.
  See remote servers
local indexes 390
  creating 399
  definition of 379
local variables 502–515
displaying on screen 504–505, 506
last row assignment 512
printing values 512
SQL derived tables and 368
lock table command 731, 744
  error message 12207 and 744
lock timeouts
  set lock wait command 745
lock wait option, set command 745
lock wait period configuration parameter 745
locking
  cursors and 645
  transactions and 711
locks 380
log on option
  create database 285
logging out
  of isql 32
logical expressions
  case expression and 488
  if...else 486
  syntax 24
  truth tables for 25
logical operators 71–73
having clauses 95
login process 31
logins
  account information 31
  logs.
    See transaction logs
longsysname custom datatype 211
loops
  break and 501
  continue and 501
  while 500–502
lower and higher datatypes.
  See precedence
M
Macintosh character set 13
makefile for ESP DLLs 604
marking tables for IDT 263
master database 282
guest user in 282
master tables 657
master-detail relationship 657
matching
  row (*= or =*), outer join 135
materialization
  function-based index 448
  materialized or nonmaterialized
    function-based index 346
  materializing transformed ordering data 349
max aggregate function 76
  See also aggregate functions
    as row aggregate 107
messages 504–507
  transactions and 740
mi.
  See minute date part
millisecond date part 533
min aggregate function 76
  See also aggregate functions
    as row aggregate 107
minus sign (-)
  subtraction operator 19
minute date part 533
mirrorexit keyword 508
mixed datatypes, arithmetic operations on 19, 216–219
mm.
  See month date part
model database 219, 282
  user-defined datatypes in 299
modifying
  data. See also data modification
    databases 287
    tables 327
modulo operator (%) 19
money data type 202, 218
  entry format 234
money datatype
  See also smallmoney datatype
monitor counters, changes to for virtually-hashed tables 418
monTableTransfer 266
monTableTransfer, columns of 268
month date part 533
  month values
    date part abbreviation and 533
ms.
  See millisecond date part
multibyte character sets
  converting 538
    datatypes for 204–205
multicolumn index.
  See composite indexes
multiple SQL statements.
  See batch processing
multiplication (*) operator 19, 46
multitable views 137, 427, 438
delete and 137, 427
insert and 137, 427

N
“N/A”, using “NULL” or 240
names
  alias for table 54
  of transactions 717
naming
  See also renaming
    columns 15
    conventions 10
    databases 284
    indexes 15
    labels 503
    local variables 511
    parameters in procedures 556
    rules 470
    savepoints 716
    stored procedures 16, 18
    tables 290, 344–345
    temporary tables 12, 290, 299
    transactions 709, 710, 715
    triggers 656
    views 16, 18
natural joins 127
nchar datatype 204–205
  entry rules 228
  like and 60
  operations on 520–523
  negative sign (-) in money values 234
  nested queries.
Index

See nesting; subqueries

nesting
See also joins
aggregate functions
begin transaction/commit statements
begin...end blocks
comments
group by clauses
if...else conditions
levels
sorts
stored procedures
subqueries
transactions
triggers
vector aggregates
warning on transactions
while loops
@@nestlevel global variable
nested procedures and
nested triggers and
noholdlock keyword, select
nonclustered constraint
create index
nonclustered indexes
integrity constraints
“none”, using “NULL” or
nonrepeatable reads
nonsharable temporary tables
normalization
not between keyword
not exists keyword
See also exists keyword
not in keyword
null values and
subqueries using
not keyword
See also logical operators
search conditions
not like keyword
not null keyword
See also null values
not null values
not-equal joins
compared to subqueries

null keyword
See also Null values
defaults and
in user-defined datatypes
null string in character columns
null values
aggregate functions and
case expressions
case expressions and
check constraints and
comparing
in computed columns
constraints and
create procedure and
datatypes allowing
default parameters as
defaults and
defining
distinct keyword and
group by and
insert and
inserting substitute values for
joins and
new rules and column definition
not allowed in IDENTITY columns
null defaults and
parameter defaults as
rules and
selecting
sort order of
triggers and
variables and
number (quantity of)
databases within transactions
server databases
tables allowed in a query
numeric data
concatenating
numeric data and row aggregates
numeric datatype
nvarchar datatype
See also character data; datatypes
entry rules
like and
operations on
O

oating-point data 235
objects
See database objects.
of option, declare cursor 626
on clause
in outer joins (ANSI syntax) 145–149
of an inner join (ANSI syntax) 142–143
on an inner table (ANSI syntax) 147
on keyword
alter database 287
create database 285
create index 447, 454, 457
create table 291
open command 627
opening cursors 621
operators
arithmetic 19, 42–46
bitwise 19–21
comparison 21, 55–56
join 124
logical 71–73
precedence 18, 72–73
relational 124
option
set quoted_identifier 7
or () bitwise operator 72
See also logical operators
or keyword
in expressions 25
in joins 127
order
See also indexes; precedence; sort order
of execution of operators in expressions 19
of null values 102
order by clause
compute by and 107–108
indexing and 445
select and 100–102
union and 116
ordering data, materializing 349
ordering, user-defined 348
outer joins 135–157
See also joins
ANSI syntax 144–145
converting outer joins with join-order dependency
(ANSI syntax) 152–154
how predicates are evaluated in Transact-SQL outer joins 152
nested outer joins (ANSI syntax) 149–151
on clause (ANSI syntax) 145–149
on clause in nested outer join (ANSI syntax) 150–151
operators 125, 155–157
parentheses in nested outer joins (ANSI syntax) 150
placement of predicate (ANSI syntax) 145–149
position of the on clause in a nested outer join (ANSI syntax) 150–151
restrictions 136–137
restrictions for the where clause (ANSI syntax) 148–149
roles of inner tables (ANSI syntax) 143
where clause (ANSI syntax) 145–149
outer tables
predicate restrictions (ANSI syntax) 145–146
output option 575–580

P
padding, data
null values and 297
underscores in temporary table names 299
pair of columns.
See common keys; joins
parameters, procedure 556–562
delimited identifiers not allowed 14
maximum number 581
parameters, using defaults 560
parentheses ()
See also Symbols section of this index
in arithmetic statements 45
in an expression 18
partition elimination 384
partition key columns 379, 406
composite 383
partition pruning 384
partitioning key columns 402, 404
partitions 377–410
adding 402
benefits of 378
Index

configuring 405
creating 395, 400
creating data partitions 395
creating index partitions 398
data partitions 379
enabling 393
enabling semantic 378
global index 380
global indexes 385
IDs 380
index 379
licensing 378
loading table data 408
local index 379
local indexes 390
locks 380
preparing for 393
semantic-based 377
statistics 409
strategies 377, 380
temporary partitioned tables 400
truncating 408
unique indexes 392
unpartitioning 403
upgrading from slices 378
path expressions, in expressions 349

patindex string function
See also wildcard characters
datatypes for 520
percent sign (%)
modulo operator 19
performance
indexes and 446
log placement and 286
stored procedures and 554
transactions and 712
triggers and 680
variable assignment and 511
period (.)
separator for qualifier names 16
permissions 32, 281, 284
create procedure 555
data modification 226
database object owners 278
readtext and column 297
referential integrity 314

stored procedures 553, 583
system procedures 584
triggers and 677, 682
views 425, 440
writetext and column 297
phantoms in transactions 724
placeholders
print message 504
plus (+)
  arithmetic operator 19, 46
  null values and 71
  string concatenation operator 21, 521
pointers
text, unitext, or image column 257
positioning cursors 614
pound sign (#) temporary table name prefix 290, 297, 299
pound sterling sign (£)
in identifiers 13
in money datatypes 234
precedence
  of lower and higher datatypes 23
  of operators in expressions 18
precision, datatype
  exact numeric types 236
predicate
  placement in outer join (ANSI syntax) 145–149
pre-evaluated computed columns 351
primary keys 360, 657–658
  constraints 311
  dropping 661–662
  indexing on 446, 453
  referential integrity and 657, 661
  updating 663–667
print command 504–505
proc_role system function 574
procedure calls, remote 31
procedures.
  See remote procedure calls; stored procedures;
  system procedures
processes (server tasks)
  infected 508
processexit keyword, waitfor 508
processing cursors 621
projection
  See also select command
distinct views 428
queries 3
views 425

publishers table
pubs2 database 751
pubs3 database 763

pubs2 database 2, 751–760
changing data in 227
diagram 761
guest user in 281
organization chart 761
table names 751
pubs3 database 2, 763–771
table names 763
punctuation
enclosing in quotation marks 220

Q

qq.
See quarter date part

qualifying

column names in joins 122
column names in subqueries 166
database objects 16
object names within stored procedures 581
table names 290

quarter date part 533
queries 2, 3
nesting subqueries 168
optimizing 554
projection 3
query performance, improving 349
query processing 550
query processor, changes to for virtually-hashed tables 417
quotation marks (” ”)
comparison operators and 22
for empty strings 240
enclosing column headings 41–42
enclosing parameter values 559
enclosing values 204, 228, 229
in expressions 23
literal specification of 23
quoted identifiers 14

R
raiserror command 506
range partitioning 381
range queries 56, 446
using < and > 57
read-only cursors 618, 626
readpast option 746–749
isolation levels and 747
readtext command 47
isolation levels and 727
views and 432
real datatype 202, 235
real datatype
See also datatypes; numeric data
rebuild indexes
sp_post_xolod 458
records, table.
See rows, table
recovery 741–742
backing up databases 322
log placement and 286
temporary tables and 298
time and transaction size 740
transactions and 712
reference information
datatypes 197
references constraint 313
referential integrity 226, 312–315
See also data integrity; triggers
constraints 308, 312
create schema command 314
cross-referencing tables 314
general rules 314
IDENTITY columns 293
maximum references allowed 314
tips 316
triggers for 657–658
regular columns 349
regulations
batches 480
identifiers 10
relational expressions 24
See also comparison operators
relational model, joins and 121
relational operations 3
relational operators 124
Index

remarks text.
See comments
remote procedure calls 17, 18, 31, 570–571
syntax for 551
user-defined transactions 735, 740
remote servers 17, 18, 570–571
Component Integration Services 30
execute and 551
non-Sybase 30
removing.
See dropping
renaming
See also naming
database objects 344–345
stored procedures 582
tables 344–345, 434
views 434
repartitioning 401
repeating subquery. See subqueries
repositioning cursors 624, 628
restoring
sample databases 751, 763
restrictions 3
See also select command
results
cursor result set 614
results of row aggregate operations 528
retrieving
data, See queries
null values 67
return command 503–504, 574
return parameters 575–580
return status 572–574
Risk Analytics Program (RAP) includes IDT 262
rules
stored procedures and 574
rollback command 715–716
See also transactions
in stored procedures 740
triggers and 672, 740
rollback trigger command 672
rounding
datetime values 204, 540
money values 202, 539
round-robin partitioning 382
row aggregates 104, 528
compared to aggregate functions 107
compute and 26, 107, 527
difference from aggregate functions 528
group by clause and 108
views and 435
@@rowcount global variable
cursors and 633
triggers and 659
rows, table 3
See also triggers
adding 237–249
changing 253–256
choosing 35, 54
copying 249
detail and summary results 528
dropping 276–277
duplicate 114, 456
row aggregates and 528
summary 104–106
unique 456
roysched table
pubs2 database 758
pubs3 database 770
RPCs (remote procedure calls). See remote procedure calls
rules 28, 239, 470
binding 471, 473
column definition conflict with 71
creating new 470
datatypes and 221
dropping user-defined 475
naming user-created 470
null values and 295
precedence 473
source text 472
specifying values with 471
testing 464, 470
triggers and 654
unbinding 474
views and 424
violations in user transaction 735
with temporary constraints 473
sales table
  pubs2 database  757
  pubs3 database  768
salesdetail table
  pubs2 database  755
  pubs3 database  767
save transaction command  715–716
  See also transactions
  savepoints  716
    setting using save transaction  710
scalar aggregates  77, 89
scalar aggregates and nesting vector aggregates within  525
schemas  314
scope of cursors  616
screen messages  504–507
scrollable cursors  613
search conditions  54
second date part  533
security
  stored procedures as  553, 583
  views and  420, 440
  security functions  546
seed values
  set identity_insert and  243
segments, placing objects on  291, 447, 454, 457
select * command  38–39, 123
  limitations  248–249
  new columns and limitations  566
select * syntax changes features
  select * syntax  37
select command  3, 35, 36
  See also joins; subqueries; views
    Boolean expressions in  487
    character strings in display  42
    choosing columns  36
    choosing rows  35, 54
    column headings  40–41
    column order in  40
    combining results of  112–117
    computing in  42
create view and  425
  creating tables for results  321–323
  database object names and
    displaying results of  36, 40–42
  with distinct  51–53
    eliminating duplicate rows with  51–53
for browse  650
if...else keyword and  486
image data  46–50
inserting data with  237, 238, 247–249
isolation levels and  727
matching character strings with  60–66
quotation marks in  41–42
reserved words in  42
restrictions in standard SQL  526
  See also SQL derived tables  365
SQL derived tables  367
text data  46
  in Transact-SQL compared to standard SQL  526
  variable assignment and  502–515
  variable assignment and multiple row results  512
views and  435
  wildcard characters in  61–64
select distinct query, group by and  103
select distinct query, order by and  103
select into command  321–323
  compute and  107
    IDENTITY columns and  326
    not allowed in cursors  625
SQL derived tables  367
temporary table  301
union and  116
select into command not allowed with compute  530
select into, using with create index  446
select into/bulkcopy/pllsort database option
  select into and  321
writetext and  257
select list  48, 121–123
  subqueries using  189
union statements  112, 114
selections.
  See select command
self-joins  130–131
  compared to subqueries  167
server cursors  615
servers
  executing remote procedures on  31
  permissions for remote logins  31
set command
  chained transaction mode  7
character string truncation and 8
inside a stored procedure 568
transaction isolation levels and 724
within update 254
set quoted_identifier option 7
set theory operations 191–192
setuser command 284
severity levels, error
user-defined messages 506
shareable temporary tables 297
shared keyword
cursors and 646
locking and 728
sharing inline defaults 476–478
size
database 285, 287
size of columns, by datatype 198
 slash (/)
division operator 19
slash asterisk (/*) comment keyword 509
smalldatetime datatype
entry format 229
smalldatetime datatype 203
See also datetime datatype; timestamp datatype
date functions and 532
operations on 532
smallint datatype 236
See also int datatype; tinyint datatype
smallmoney datatype 202, 218
See also money datatype
entry format 234
sort order
See also order by clause
comparison operators and 22
order by and 100
sorted_data option, create index 457
sortkey, function 349
source text 4
encrypting 5
sp_addextendedproc system procedure 607
sp_addmessage system procedure 507
sp_addmessage system procedure print command 506
sp_addtype system procedure 197, 220, 299
sp_bindefault system procedure 221, 466–468
batches using 481
sp_bindrule system procedure 221, 471–473
batches using 481
sp_chagattributet, changes to for virtually-hashed
tables 418
sp_changecolumn system procedure 283
sp_commonkey system procedure 162
sp_dboption system procedure
transactions and 713
sp_dboption, using with create index 446
sp_dephende system procedure 442, 587, 684
sp_displaylogin system procedure 31
sp_dropextendedproc system procedure 608
sp_dropsegment system procedure 287
sp_dropsegments system procedure 222
sp thiểu segment system procedure 287
sp_foreignkey system procedure 162, 360, 658
sp_freedll system procedure 594
sp_getmessage system procedure 506–507
sp_help system procedure 356–360
IDENTITY columns and 429
sp_help, changes to for virtually-hashed tables 418
sp_helpconstraint system procedure 360
sp_helpdb system procedure 356
sp_helpdevice system procedure 285
sp_helpextendedproc system procedure 610
sp_helpindex system procedure 458
sp_helpjoins system procedure 161, 657
sp_helpprotect system procedure 589
sp_helpuserdata system procedure
defaults 475
procedures 552, 586
rules 475
triggers 682, 683
sp_modifypassword system procedure 283
sp_placeobject, changes to for virtually-hashed tables 418
sp_post_xoload
rebuild indexes 458
sp_primarykey system procedure 657
sp_procmode system procedure 738
sp_recompile system procedure 554
sp_rename system procedure 344–345, 343, 582
sp_spaceused system procedure 362
sp_unbindefault system procedure 468
sp_unbindrule system procedure 474
sp_version function 580
space
database storage 287
estimating table and index size 362
freeing with \texttt{truncate table} 277
for index pages 362
spaces, character
empty strings (’ ’) or (’ ’) as 23, 240
special characters 10
speed (server)
of recovery 740
\textit{spt\_TableTransfer}, columns in 267
\textit{spt\_TableTransfer}, introduction 266
\textit{spt\_TableTransfer}, create using \texttt{sp\_transfer\_table} 267
SQL derived tables
advantages 366
aggregate functions and 373
and derived column lists 368
constant expressions and 371
correlated 369
correlated attributes and 375
correlation names and 368
creating tables from 374
defined by derived table expressions 365
differences from abstract plan derived tables 365
from keyword 367
joins and 373
local variables and 368
nesting 370
optimization and 367
renaming columns 371
See also derived table expression 365
\texttt{select} command 367
\texttt{select} into command 367
subqueries and 192, 370
subqueries with the \texttt{union} operator 371
syntax 367
temporary tables and 368
\texttt{union} operator and 370
using 369
using views with 374
SQL standards 5
aggregate functions and 526
set options for 6
transactions 731
SQL.
See Transact-SQL

\texttt{@@sqlstatus} global variable 518, 630
SRV\_PROC structure 597
ss.
See second date part
statements 2, 10
statement blocks (\texttt{begin...end}) 499
timed execution of statement blocks 508
\texttt{store\_employees} table, \texttt{pubs3} database 769
stored procedures 26, 549–552
See also system procedures; triggers
checking for roles in 574
compiling 550
control-of-flow language 485–510
creating 571–580
cursors within 643
default parameters 558–561
dependencies 587
dropping 583
grouping 564
information on 585
isolation levels 736
local variables and 511
modes 736
naming 16, 18
nesting 566
object owner names in 581
parameters 556, 562
permissions on 583
renaming 582
results display 551
return parameters 575–580
return status 572–574
rollback in 740
as security mechanisms 553, 583
source text 552
storage of 551
temporary tables and 299, 581
timed execution of 508
transactions and 732–738
with recompile 565
\texttt{stores} table
\texttt{pubs2} database 757
\texttt{pubs3} database 769
string functions 520–523
nesting 522
\texttt{string\_rtruncation} option, set 8
strings
  concatenating 21, 520
  empty 205
  matching with like 60–66
  truncating 228
stringsfunctions
  concatenating 521
structure, virtually-hashed table 412
Structured Query Language (SQL) 1
subqueries 163
  See also joins
  aggregate functions and 177, 178
  all keyword and 177, 181, 189
  any keyword and 24, 177, 182, 189
  column names in 166
  comparison operators in 182, 189
  comparison operators in correlated 195–196
  correlated or repeating 193–196
  correlation names in 167, 194
  datatypes not allowed in 164
  delete statements using 173
  exists keyword in 188–192
  in expressions 24
  expressions, replacing with 174
  group by clause in 178–179, 195–196
  having clause in 178–179, 196
  in keyword and 58, 183, 185–187, 189
  insert statements using 173
  joins compared to 186–187
  manipulating results in 165
  modified comparison operators and 179, 180, 189
  nesting 168
  not exists keyword and 190–192
  not in keyword and 187–188
  not-equal joins and 133–134
  null values and 164
  order by and 102
  repeating 193–196
  restrictions on 164
  select lists for 189
  SQL derived tables and 192, 370
  syntax 175
  types 175
  unmodified comparison operators and 176–178
  update statements using 173
  where clause and 164, 186, 188
subquery construction 619
subtraction operator (-) 19
suffix names, temporary table 299
sum aggregate function 76
  See also aggregate functions
  as row aggregate 107
  summary rows 104–106
  summary values 26, 75
  aggregate functions and 104
  triggers and 668–670
syb_identity keyword 294, 429
  IDENTITY columns and 243, 294
sybexp_dll_version function 593
sybsystemdb database 282
sybsystemprocs database 584, 610
symbols
  See also wildcard characters; Symbols section of this index
  arithmetic operator 19
  comparison operator 22
  matching character strings 61
  money 234
synonyms
  datatype 198
  keywords 9
  out for output 579
syntax
  Transact-SQL 10, 18
  virtually-hashed tables 413
  syscolumns table 210
  syscomments table 551, 682
  syssdatabases table 284
  syssdevices table 285
  SYSINDEXES.индfirst 412
  SYSINDEXES.indroot 412
  syskeys table 162, 657
  syslogs table 741
  sysmessages table
  raiserror and 506
  sysname custom datatype 211
  sysobjects table 358–360
  sysprocedures table 682
  sysprocesses table 508
System Administrator
  database ownership 283
  system datatypes.
See datatypes
system extended stored procedures 609–610
system procedures 26, 583–585
See also stored procedures; individual procedure names
changes to for virtually-hashed tables 418
delimited identifiers not allowed as parameters 14
isolation level 730
not allowed in user-defined transactions 715
reoptimizing queries with 554
running in chained mode 737
on temporary tables 301
viewing text of 586
system tables 281, 423
See also tables; individual table names
dropping 345
system procedures and 584
triggers and 678, 682–684
systypes table 216, 358, 360
sysusages table 284
sysuserrmessages table 506
sysusers table 281

T
table
virtually-hashed, structure of 412
table columns.
See columns
table rows.
See rows, table
table-level constraints 308
tables 3, 288–290
See also database objects; triggers; views allowed in a from clause 53, 123
changing 327–345
correlation names 54, 130, 167
correlation names in subqueries 194
creating new 318–327
dependent 661
designing 318
dropping 345, 434
IDENTITY column 291–292
inner 136

isnull system function and 240
joins of 119–162
names, in joins 123, 130
naming 12, 54, 290
outer 136
renaming 344–345
row copying in 249
space used by 362
temporary, names beginning tempdb 300
virtually-hashed 411
virtually-hashed, creating 412, 413
virtually-hashed, syntax 413
tables, temporary
See temporary tables
tempdb database 282
temporary tables 53
See also tables; tempdb database
create table and 290, 297–299
creating 297–299
names beginning tempdb.. 300
naming 12, 290, 299
select into and 301, 321–323
SQL derived tables and 368
stored procedures and 581
triggers and 298, 678
views not permitted on 298, 424
text
complex datatype 348
line continuation with backslash (\) 24
text datatype 208
See also datatypes
changing with writetext 257
Component Integration Services 208
converting 538
table
entry rules 228
initializing with null values 297
inserting 237
like and 60, 62
operations on 520, 523
prohibited actions on text, image 102
selecting 46
selecting in views 432
subqueries using 164
triggers and 678
union not allowed on 117
Index

- updating 253
- updating in views 436
- where clause and 55, 62
- text functions 523
- text pointer values 257
- @@textsize global variable 46
- theta joins 125
  See also joins
time datatype 229
  entry format 229
  time interval, execution 508
- time option, waitfor 508
- time values
  See also dates; datetime datatype; smalldatetime datatype
display format 532
  entry format 229–230
  functions on 532
  like and 233
  searching for 233
  storage 532
timestamp datatype 211
  See also datetime datatype; smalldatetime datatype
  browse mode and 650
  comparing timestamp values 652
  inserting data and 237
  skipping 238
timestamp tsequal function 652
timing
  @@error status check 516
tinyint datatype 236
  See also int datatype; smallint datatype
titleauthor table
  pubs2 database 754
  pubs3 database 766
titles table
  pubs2 database 753
  pubs3 database 765
totals
  See also aggregate functions
  grand (compute without by) 111
  with compute clause 104
- Trailing zeros
  mixed in data 259
  preserved 260
  truncating 258–261
- @@trancount global variable 517, 719
- transaction isolation levels
  readpast option and 747
- transaction logs 741
  on a separate device 285
  size 286
- writetext and 257
- transactions 227, 709–742
  canceling 740
  Component Integration Services 712
cursors and 739
  isolation levels 7, 721
  locking 711
  modes 7, 721
  names not used in nested 717
  naming 715
  nesting levels 719, 732
  number of databases allowed 740
  performance and 712
  recovery and 712, 742
  SQL standards compliance 731
  states 717
  stored procedures and 716
  stored procedures and triggers 732
  timed execution 508
  @@transtate global variable 717
  triggers and 672, 716
- Transact-SQL
  aggregate functions in 526
  enhancements to 25–29
  extensions 29–30
  translation
    of integer arguments into binary numbers 20
  @@transstate global variable 517
  transtate global variable 717–719
  trigger tables 656, 658
  dropping 682
  triggers 27, 653–684
  See also database objects; stored procedures
  Component Integration Services 678
  creating 655–657
  dropping 682
  help on 683
  on image columns 678
  naming 656
  nested 674–677
nested, and rollback trigger 673
null values and 678–679
object renaming and 680
performance and 680
permissions and 677, 682
recursion 675
restrictions on 656, 678
rollback in 740
rolling back 672
rules and 654
self-recursion 675
set commands in 680
storage 682
summary values and 668–670
system tables and 678, 682–684
temporary tables and 678
test tables 658–659
on text columns 678
transactions and 672, 732–738
truncate table command and 678
views and 424, 678
TRUE, return value of 188
truncate table command 277–278
referential integrity and 315
triggers and 678
truncating partitions 408
t truncating trailing zeros 258–261
truncation
  binary datatypes 234
  character string 8
  temporary table names 299
  truth tables
  logical expressions 25
tsequal system function 652

U
unbinding
  defaults 468
  rules 474
unchained transaction mode 721
unichar
  data 205
  unichar datatype 204, 206
  like and 60

Index

operations on 520–523
union operator 112–117, 619
unique constraints 308, 311
unique indexes 392, 449, 455
unique keyword 449
duplicate data from 456
univarchar datatype
  like and 60
  operations on 520–523
unknown values.
  See null values
unpartitioned table
definition of 378
unpartitioning 403
unsigned int datatype 200
unsigned smallint datatype 200
updatable cursors 618
update command 253–256, 437
cursors and 634
duplicate data from 455, 456
image data and 210
multitable views 435
null values and 241, 296, 297
subqueries using 173
text data and 208
triggers and 658–659, 663–667, 678
views and 137, 427, 435
update statistics command 460
  with Component Integration Services 460
updating
  See also changing; data modification
  in browse mode 650
cursor rows 634
cursors 623
foreign keys 667
image datatype 253
index statistics 460
primary keys 663–667
text datatype 253
using join operations 255
use command 283
  batches using 480
  create procedure with 580
user databases 282
user keyword
  create table 310
Index

user-defined
  ordering 348
  procedures 549
  transactions 712
user-defined datatypes 219, 220, 223
  altering columns 340
  defaults and 296, 466
  dropping a column with 341
  IDENTITY columns and 221, 292
  longsysname as 211
  modifying columns with 341
  rules and 219–221, 471–473
  sysname as 211
  temporary tables and 301
  timestamp as 211
user-defined function 452
user-defined sort order, instead of sortkey 349
user-defined sort order, instead of sortkey() 349
users
  adding 282

V

valid_name system function
  checking strings with 13
values
  IDENTITY columns 291
values option, insert 237–238
varbinary datatype 209
  See also binary data; datatypes
  “0x” prefix 234
like and 60
  operations on 520–523
varchar datatype 205
  See also character data; datatypes
  entry rules 228
  in expressions 23
like and 60
  operations on 520–523
variable
  in derived table syntax 368
variable-length columns
  compared to fixed-length 205
  null values in 296
variables
  comparing 514–515
  declaring 511–515
  global 516–518, 581
  in update statements 254
  last row assignment 512
  local 502–515, 581
  printing values 512
  vector aggregates 83, 525
    nesting 89
    nesting inside scalar aggregates 525
views 419, 423
  See also database objects
  advantages 420
  aggregate functions and 435
  allowed in a from clause 53, 123
  column names 424
  computed columns and 436
  creating 423
  data modification and 435
  defaults and 424
  dependent 433
distinct and 424
  dropping 434, 440
  functions and 426
  help on 441
  IDENTITY columns and 439
  indexes and 424
insert and 430–431
  joins and 119, 427
  keys and 424
  naming 16
  permissions on 425, 440
  projection of distinct 428
  projections 425
  querying 431
readtext and 432
  redefining 433
  references 442
  renaming 434
  resolution 431, 432
  restrictions 435–439
  retrieving data through 431
  rules and 424
security and 420
source text 425
  temporary tables and 298, 424
triggers and 298, 424, 678
union and 117
update and 430–431
updates not allowed 436
using SQL derived tables with 374
with check option 137, 427, 430–431, 435, 438
with joins 137, 427
writetext and 432
virtual columns 347
virtual computed columns 351
virtually-hashed tables 411
changes to monitor counters 418
changes to query processor 417
changes to system procedures 418
commands 417
creating 413
examples of 414
limitations of 416
structure of 412

W
waitfor command 508–509
week date part 533
weekday date part 533
when...then conditions
in case expressions 488, 494
where clause
aggregate functions not permitted in
compared to having 94
group by clause and 91–92
in outer join (ANSI syntax) 145–149
join and 128
joins and 124
like and 62
null values in a 68
search conditions in 54
skeleton table creation with 323
subqueries using 164, 186, 188
text data and 55, 62
update 255
where current of clause 634, 635
while keyword 500–502
wildcard characters
default parameters using 561
in a like match string 61–66
searching for 63
with check option option
views and 430–431
with log option, writetext 227
with recompile option
create procedure 565
execute 566
wk.
See week date part
work keyword (transactions) 715
write-ahead log 741
writetext command 257
views and 432, 436
with log option 227

X
XML
complex datatype 348
XP Server 27, 592, 593
xp_cmdshell context configuration parameter 596
xp_cmdshell system extended stored procedure 609
xp_cmdshell context configuration parameter and 596
xpserver utility command 593

Y
year
date part 533
yen sign (¥ )
in identifiers 13
in money datatypes 234
yy. See year date part

Z
zero x (0x) 540
zeros
using NULL or 240
Index