## Contents

### About This Book
ix

### CHAPTER 1  Introduction
1
- Embedded SQL overview .............................................. 1
- Embedded SQL features ............................................. 2
- Transact-SQL support in Embedded SQL ...................... 2
- Getting started .......................................................... 3
  - Using the examples .................................................. 3
  - Backward compatibility .......................................... 4
- Creating and running an Embedded SQL program ............. 4
- How the precompiler processes your applications .......... 5
- Multiple Embedded SQL source files ......................... 6
- Precompiler-generated files ....................................... 6
- Group element referencing ....................................... 6

### CHAPTER 2  General Information
9
- Five tasks of an Embedded SQL program .................... 9
- Simplified Embedded SQL program ............................ 10
- General rules for Embedded SQL ................................ 11
- Statement placement ............................................... 12
- Comments ............................................................. 12
- Identifiers .............................................................. 13
- Quotation marks ...................................................... 13
- Reserved words ..................................................... 13
- Variable naming conventions .................................... 13
- Scoping rules ......................................................... 14
- Statement batches ................................................... 14
- Embedded SQL constructs ........................................ 14

### CHAPTER 3  Communicating with Adaptive Server Enterprise
17
- Scoping rules: SQLCA, SQLCODE, and SQLSTATE ........ 18
- Declaring SQLCA .................................................... 18
- Multiple SQLCAs .................................................... 18
Contents

SQLCA variables................................................................. 18
Accessing SQLCA variables ............................................... 19
SQLCODE within SQLCA ................................................... 20
Declaring SQLCODE as a standalone area......................... 20
Using SQLSTATE ............................................................. 21
Obtaining SQLSTATE codes and error messages................. 22

CHAPTER 4 Using Variables.................................................. 23
Declaring variables............................................................ 23
Declaring a character array ................................................. 24
Using host variables.......................................................... 25
Host input variables......................................................... 25
Host result variables....................................................... 26
Host status variables....................................................... 26
Host output variables..................................................... 27
Using indicator variables................................................... 27
Indicator variables and server restrictions......................... 27
Using host variables with indicator variables.................. 27
Host variable conventions............................................... 30
Using arrays..................................................................... 31
Multiple arrays............................................................... 31
Scoping rules................................................................... 31
Datatypes........................................................................ 33
Elementary data items..................................................... 34
Group data items............................................................. 34
Special data items.......................................................... 35
Comparing COBOL and Adaptive Server Enterprise datatypes 35
Converting datatypes....................................................... 36

CHAPTER 5 Connecting to Adaptive Server Enterprise .......... 39
Connecting to a server.................................................... 39
user .............................................................................. 40
password ....................................................................... 40
connection_name.......................................................... 40
server .......................................................................... 40
connect example............................................................ 41
Changing the current connection.................................... 41
Establishing multiple connections................................. 41
Naming a connection....................................................... 42
Using Adaptive Server Enterprise connections............... 43
Disconnecting from a server............................................ 44
CHAPTER 6  Using Transact-SQL Statements ............................................................................... 45
  Transact-SQL statements in Embedded SQL ................................................................. 45
    exec sql syntax ........................................................................................................ 45
    Invalid statements ................................................................................................. 46
    Transact-SQL statements that differ in Embedded SQL ........................................... 46
  Selecting rows .......................................................................................................... 46
    Selecting one row .................................................................................................... 47
    Selecting multiple rows through arrays ................................................................ 47
    Using stored procedures ....................................................................................... 61
  Grouping statements ................................................................................................. 64
    Grouping statements by batches ............................................................................ 64
    Grouping statements by transactions .................................................................... 65
  Including files and directories .................................................................................. 67

CHAPTER 7  Using Dynamic SQL .......................................................................................... 69
  When to use dynamic SQL ........................................................................................ 69
  Dynamic SQL protocol ............................................................................................. 70
  Method 1: Using execute immediate ......................................................................... 71
    Method 1 examples .................................................................................................. 72
  Method 2: Using prepare and execute ....................................................................... 73
    prepare ................................................................................................................... 73
    execute ................................................................................................................... 74
    Method 2 example .................................................................................................. 75
  Method 3: Using prepare and fetch with a cursor ...................................................... 76
    prepare ................................................................................................................... 76
    declare .................................................................................................................. 76
    open ...................................................................................................................... 77
    fetch and close ..................................................................................................... 78
    Method 3 example .................................................................................................. 78
  Method 4: Using prepare and fetch with system descriptors ..................................... 80
    Method 4 dynamic descriptors .............................................................................. 80
    Dynamic descriptor statements ............................................................................. 81
    Method 4 example .................................................................................................. 82
    About SQLDAs ....................................................................................................... 85
    Using SYBSETSQDLDA ......................................................................................... 87
    Method 4 example using SQLDAs ....................................................................... 90

CHAPTER 8  Handling Errors .................................................................................................. 95
  Testing for errors ....................................................................................................... 96
    Using SQLCODE .................................................................................................... 96
  Testing for warning conditions ................................................................................. 96
  Trapping errors with the whenever statement ......................................................... 97
    whenever testing conditions .................................................................................. 98
## Contents

whenever actions ................................................................. 99  
Using get diagnostics ......................................................... 100  
Writing routines to handle warnings and errors .................. 100  
Precompiler-detected errors ............................................... 101

### CHAPTER 9  
**Embedded SQL Statements: Reference Pages** ................. 103

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>allocate descriptor</td>
<td>105</td>
</tr>
<tr>
<td>begin declare section</td>
<td>106</td>
</tr>
<tr>
<td>begin transaction</td>
<td>107</td>
</tr>
<tr>
<td>close</td>
<td>109</td>
</tr>
<tr>
<td>commit</td>
<td>110</td>
</tr>
<tr>
<td>connect</td>
<td>113</td>
</tr>
<tr>
<td>deallocate cursor</td>
<td>115</td>
</tr>
<tr>
<td>deallocate descriptor</td>
<td>116</td>
</tr>
<tr>
<td>deallocate prepare</td>
<td>118</td>
</tr>
<tr>
<td>declare cursor (dynamic)</td>
<td>119</td>
</tr>
<tr>
<td>declare cursor (static)</td>
<td>121</td>
</tr>
<tr>
<td>declare cursor (stored procedure)</td>
<td>123</td>
</tr>
<tr>
<td>declare scrollable cursor</td>
<td>125</td>
</tr>
<tr>
<td>delete (positioned cursor)</td>
<td>126</td>
</tr>
<tr>
<td>delete (searched)</td>
<td>128</td>
</tr>
<tr>
<td>describe input (SQL descriptor)</td>
<td>130</td>
</tr>
<tr>
<td>describe input (SQLDA)</td>
<td>132</td>
</tr>
<tr>
<td>describe output (SQL descriptor)</td>
<td>133</td>
</tr>
<tr>
<td>describe output (SQLDA)</td>
<td>135</td>
</tr>
<tr>
<td>disconnect</td>
<td>137</td>
</tr>
<tr>
<td>exec</td>
<td>139</td>
</tr>
<tr>
<td>exec sql</td>
<td>142</td>
</tr>
<tr>
<td>execute</td>
<td>143</td>
</tr>
<tr>
<td>execute immediate</td>
<td>145</td>
</tr>
<tr>
<td>exit</td>
<td>146</td>
</tr>
<tr>
<td>fetch</td>
<td>147</td>
</tr>
<tr>
<td>scroll fetch</td>
<td>149</td>
</tr>
<tr>
<td>get descriptor</td>
<td>151</td>
</tr>
<tr>
<td>get diagnostics</td>
<td>153</td>
</tr>
<tr>
<td>include “filename”</td>
<td>154</td>
</tr>
<tr>
<td>include sqlca</td>
<td>156</td>
</tr>
<tr>
<td>include sqlda</td>
<td>157</td>
</tr>
<tr>
<td>initialize_application</td>
<td>157</td>
</tr>
<tr>
<td>open (dynamic cursor)</td>
<td>159</td>
</tr>
<tr>
<td>open (static cursor)</td>
<td>160</td>
</tr>
<tr>
<td>open scrollable cursor</td>
<td>162</td>
</tr>
<tr>
<td>prepare</td>
<td>162</td>
</tr>
<tr>
<td>rollback</td>
<td>164</td>
</tr>
</tbody>
</table>
Contents

select ................................................................................................................. 165
set connection ............................................................................................... 167
set descriptor ................................................................................................. 168
update ............................................................................................................. 170
whenever .......................................................................................................... 171

CHAPTER 10 Open Client/Server Configuration File ..................................... 177
  Purpose of the Open Client/Server configuration file ...................... 177
  Accessing the configuration functionality ......................................... 177
  Default settings ......................................................................................... 178
  Syntax for the Open Client/Server configuration file .................... 179
    Syntax ........................................................................................................ 179
  Sample programs .................................................................................... 181
    Embedded SQL/COBOL sample programs ......................................... 182
    Embedded SQL program version for use with the -x option.... 182
    Same Embedded SQL program with the -e option .................... 184

APPENDIX A Precompiler Warning and Error Messages .................. 187
  Understanding the codes in the tables ................................................. 187

Glossary ......................................................................................................... 203

Index ............................................................................................................. 211
About This Book

This book explains how to use Embedded SQL™ and the Embedded SQL precompiler with COBOL applications. Embedded SQL is a superset of Transact-SQL® that lets you place Transact-SQL statements in application programs written in languages such as COBOL and C.

The information in this book is platform-independent. For platform-specific instructions on using Embedded SQL, see the Open Client™ and Open Server™ Programmers Supplement.

Audience

This book is intended for application developers and others interested in Embedded SQL concepts and uses. To use this guide, you should:

- Be familiar with the information in the Adaptive Server® Enterprise Reference Manual
- Have COBOL programming experience

How to use this book

This book contains these chapters:

- Chapter 1, “Introduction,” presents a brief overview of Embedded SQL and describes its advantages and capabilities.
- Chapter 2, “General Information,” describes the tasks of an Embedded SQL program and provides general rules for programming with Embedded SQL.
- Chapter 3, “Communicating with Adaptive Server Enterprise,” describes how to establish and use a communication area with SQLCA, SQLCODE, and SQLSTATE. This chapter also describes the system variables used in the communication area.
- Chapter 4, “Using Variables,” explains how to declare and use host and indicator variables in Embedded SQL. This chapter also describes arrays and explains datatype conversions.
- Chapter 5, “Connecting to Adaptive Server Enterprise,” explains how to use Embedded SQL to connect an application program to Adaptive Server® Enterprise and data servers in general.
Chapter 6, “Using Transact-SQL Statements,” describes how to use Transact-SQL in an Embedded SQL application program. This chapter describes how to select rows using arrays and batches, and how to group Transact-SQL statements.

Chapter 7, “Using Dynamic SQL,” describes how to create Embedded SQL statements that your application’s users can enter interactively at runtime.

Chapter 8, “Handling Errors,” describes return codes and the Embedded SQL precompiler’s facilities for detecting and handling errors.

Chapter 9, “Embedded SQL Statements: Reference Pages,” provides a reference page for each Embedded SQL statement.

Chapter 10, “Open Client/Server Configuration File,” describes the use of an external configuration file with Embedded SQL.

Appendix A, “Precompiler Warning and Error Messages,” lists precompiler and runtime messages.

The Glossary defines many of the terms used in this documentation.

Related documents

You can see these books for more information:

- The Open Server and SDK New Features for Windows, Linux, and UNIX, which describes new features available for Open Server and the Software Developer’s Kit. This document is revised to include new features as they become available.

- The Open Server Release Bulletin for your platform contains important last-minute information about Open Server.

- The Software Developer’s Kit Release Bulletin for your platform contains important last-minute information about Open Client™ and SDK.

- The jConnect™ for JDBC™ Release Bulletin contains important last-minute information about jConnect.

- The Open Client and Open Server Configuration Guide for your platform contains information about configuring your system to run Open Client and Open Server.

- The Open Client Client-Library/C Programmers Guide contains information on how to design and implement Client-Library applications.


• The *Open Client and Open Server Common Libraries Reference Manual* contains reference information for CS-Library, which is a collection of utility routines that are useful in both Client-Library and Server-Library applications.


• The *Open Client and Open Server Programmers Supplement* for your platform contains platform-specific information for programmers using Open Client and Open Server. This document includes information about:
  • Compiling and linking an application
  • The sample programs that are included with Open Client and Open Server
  • Routines that have platform-specific behaviors

• The *Installation and Release Bulletin Sybase® SDK DB-Library Kerberos Authentication Option* contains information about installing and enabling the MIT Kerberos security mechanism to be used on DB-Library. DB-Library only supports network authentication and mutual authentication in the Kerberos security mechanism.

• The *Open Client and Open Server International Developers Guide* provides information about creating internationalized and localized applications.

• The *Open Client Embedded SQL™/C Programmers Guide* explains how to use Embedded SQL and the Embedded SQL precompiler with C applications.

• The *jConnect for JDBC Programmers Reference* describes the jConnect for JDBC product and explains how to access data stored in relational database management systems.

• The *Adaptive Server Enterprise ADO.NET Data Provider Users Guide* provides information on how to access data in Adaptive Server using any language supported by .NET, such as C#, Visual Basic .NET, C++ with managed extension, and J#.
• The *Adaptive Server Enterprise ODBC Driver by Sybase Users Guide* for Microsoft Windows and UNIX, provides information on how to access data from Adaptive Server on Microsoft Windows and UNIX platforms, using the Open Database Connectivity (ODBC) Driver.

• The *Adaptive Server Enterprise OLE DB Provider by Sybase Users Guide for Microsoft Windows* provides information on how to access data from Adaptive Server on Microsoft Windows platforms, using the Adaptive Server OLE DB Provider.

• The *Adaptive Server Enterprise Database Driver for Perl Programmers Guide* provides information for Perl developers to connect to an Adaptive Server database and query or change information using a Perl script.

• The *Adaptive Server Enterprise extension module for PHP Programmers Guide* provides information for PHP developers to execute queries against an Adaptive Server database.

• The *Adaptive Server Enterprise extension module for Python Programmers Guide* provides information about Sybase-specific Python interface that can be used to execute queries against an Adaptive Server database.

---

**Other sources of information**

Use the Sybase® Product Documentation Web site to learn more about your product:

• The Sybase Product Documentation Web site is accessible using a standard Web browser. In addition to product documentation, you will find links to EBFs/Maintenance, Technical Documents, Case Management, Solved Cases, newsgroups, and the Sybase Developer Network.

To access the Sybase Product Documentation Web site, go to Product Documentation at http://www.sybase.com/support/manuals/.

**Sybase certifications on the Web**

Technical documentation at the Sybase Web site is updated frequently.

❖ **Finding the latest information on product certifications**


2. Click Partner Certification Report.

3. In the Partner Certification Report filter select a product, platform, and timeframe and then click Go.

4. Click a Partner Certification Report title to display the report.
❖ Finding the latest information on component certifications

2. Either select the product family and product under Search by Base Product; or select the platform and product under Search by Platform.
3. Select Search to display the availability and certification report for the selection.

❖ Creating a personalized view of the Sybase Web site (including support pages)

Set up a MySybase profile. MySybase is a free service that allows you to create a personalized view of Sybase Web pages.

2. Click MySybase and create a MySybase profile.

❖ Finding the latest information on EBFs and software maintenance

2. Select EBFs/Maintenance. If prompted, enter your MySybase user name and password.
3. Select a product.
4. Specify a time frame and click Go. A list of EBF/Maintenance releases is displayed.
   Padlock icons indicate that you do not have download authorization for certain EBF/Maintenance releases because you are not registered as a Technical Support Contact. If you have not registered, but have valid information provided by your Sybase representative or through your support contract, click Edit Roles to add the “Technical Support Contact” role to your MySybase profile.
5. Click the Info icon to display the EBF/Maintenance report, or click the product description to download the software.
### conventions

Table 1: Syntax conventions

<table>
<thead>
<tr>
<th>Key</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>command</td>
<td>Command names, command option names, utility names, utility flags, and other keywords are in <strong>sans-serif</strong> font.</td>
</tr>
<tr>
<td>variable</td>
<td>Variables, or words that stand for values that you fill in, are in <strong>italics</strong>.</td>
</tr>
<tr>
<td>{ }</td>
<td>Curly braces indicate that you choose at least one of the enclosed options. Do not include the braces in the command.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Brackets mean choosing one or more of the enclosed items is optional. Do not include the braces in the command.</td>
</tr>
<tr>
<td>( )</td>
<td>Parentheses are to be typed as part of the command.</td>
</tr>
<tr>
<td></td>
<td>The vertical bar means you can select only one of the options shown.</td>
</tr>
<tr>
<td></td>
<td>The comma means you can choose as many of the options shown as you like, separating your choices with commas to be typed as part of the command.</td>
</tr>
</tbody>
</table>

### accessibility features

This document is available in an HTML version that is specialized for accessibility. You can navigate the HTML with an adaptive technology such as a screen reader, or view it with a screen enlarger.

Open Client and Open Server documentation has been tested for compliance with U.S. government Section 508 Accessibility requirements. Documents that comply with Section 508 generally also meet non-U.S. accessibility guidelines, such as the World Wide Web Consortium (W3C) guidelines for Web sites.

**Note** You might need to configure your accessibility tool for optimal use. Some screen readers pronounce text based on its case; for example, they pronounce **ALL UPPERCASE TEXT** as initials, and **MixedCase Text** as words. You might find it helpful to configure your tool to announce syntax conventions. Consult the documentation for your tool.

For information about how Sybase supports accessibility, see Sybase Accessibility at http://www.sybase.com/accessibility. The Sybase Accessibility site includes links to information on Section 508 and W3C standards.

**If you need help**

Each Sybase installation that has purchased a support contract has one or more designated people who are authorized to contact Sybase Technical Support. If you cannot resolve a problem using the documentation or online help, please have the designated person contact Sybase Technical Support or the Sybase subsidiary in your area.
CHAPTER 1

Introduction

This chapter includes the following topics to introduce Embedded SQL and the Embedded SQL precompiler.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Embedded SQL overview</td>
<td>1</td>
</tr>
<tr>
<td>Embedded SQL features</td>
<td>2</td>
</tr>
<tr>
<td>Transact-SQL support in Embedded SQL</td>
<td>2</td>
</tr>
<tr>
<td>Getting started</td>
<td>3</td>
</tr>
<tr>
<td>Creating and running an Embedded SQL program</td>
<td>4</td>
</tr>
<tr>
<td>How the precompiler processes your applications</td>
<td>5</td>
</tr>
</tbody>
</table>

Embedded SQL overview

Embedded SQL is a superset of Transact-SQL that lets you place Transact-SQL statements in application programs written in languages such as COBOL and C.

Embedded SQL is a product that enables you to create programs that access and update Adaptive Server Enterprise data. Embedded SQL programmers write SQL statements directly into an application program written in a conventional programming language such as C or COBOL. A preprocessing program—the Embedded SQL precompiler—processes the completed application program, resulting in a program that the host language compiler can compile. The program is linked with Open Client Client-Library before it is executed.

Embedded SQL is one of the two programming methods Sybase provides for accessing Adaptive Server Enterprise. The other programming method is the call-level interface. With the call-level interface, you place Client-Library calls directly into an application program and then link with Client-Library.
Embedded SQL features

You can place Embedded SQL statements anywhere in a host program and mix them with host language statements. All Embedded SQL statements must begin with the keywords `exec sql` and end with `end-exec`.

You can use host variables in Embedded SQL statements to store data retrieved from Adaptive Server Enterprise and as parameters in Embedded SQL statements; for example, in the `where` clause of a `select` statement. In Dynamic SQL, host variables can also contain text for Embedded SQL statements.

Embedded SQL features

Embedded SQL provides several advantages over a call-level interface:

- Embedded SQL is easy to use because it is simply Transact-SQL with some added features that facilitate using it in an application.
- It is an ANSI/ISO-standard programming language.
- It requires less coding to achieve the same results as a call-level approach.
- Embedded SQL is essentially identical across different host languages. Programming conventions and syntax change very little. Therefore, to write applications in different languages, you need not learn new syntax.
- The precompiler can optimize execution time by generating stored procedures for the Embedded SQL statements.

Transact-SQL support in Embedded SQL

With the exception of `print`, `raiserror`, `readtext`, and `writetext`, all Transact-SQL statements, functions, and control-of-flow language are valid in Embedded SQL. You can develop an interactive prototype of your Embedded SQL application in Transact-SQL to facilitate debugging your application, then easily incorporate it into your application.

Most Adaptive Server Enterprise datatypes have an equivalent in Embedded SQL. Also, you can use host language datatypes in Embedded SQL. Many datatype conversions occur automatically when a host language datatype does not exactly match an Adaptive Server Enterprise datatype.
You can place host language variables in Embedded SQL statements wherever literal quotes are valid in Transact-SQL. Enclose the literal with either single (’) or double (“) quotation marks. For information on delimiting literals that contain quotation marks, see the Adaptive Server Enterprise Reference Manual.

Embedded SQL has several features that Transact-SQL does not have:

- **Automatic datatype conversion** occurs between host language types and Adaptive Server Enterprise types.
- **Dynamic SQL** lets you define SQL statements at runtime.
- **SQLCA, SQLCODE, and SQLSTATE** lets you communicate between Adaptive Server Enterprise and the application program. The three entities contain error, warning, and informational message codes that Adaptive Server Enterprise generates.
- **Return code testing routines** detect error conditions during execution.

## Getting started

Before attempting to run the precompiler, make sure that Client-Library version 12.5 or later is installed, since the precompiler uses it as the runtime library. Also, make sure Adaptive Server Enterprise version 12.5 or later is installed. If products are missing, contact your **System Administrator**.

Invoke the precompiler by issuing the appropriate command at the operating system prompt. See the *Open Client and Open Server Programmers Supplement* for details.

The precompiler command can include several flags that let you determine options for the precompiler, including the input file, login user name and password, invoking HA failover, and precompiler modes. The *Open Client and Open Server Programmers Supplement* contains operating system-specific information on precompiling, compiling, and linking your Embedded SQL application.

## Using the examples

The examples in this guide use the pubs2 database. To run the examples, specify the pubs2 database with the Transact-SQL use statement.
This product is shipped with several online examples. For information on running these examples, see the *Open Client and Open Server Programmers Supplement*.

**Backward compatibility**

The precompiler is compatible with precompilers that are ANSI SQL-89-compliant. However, you may have applications created with earlier Embedded SQL versions that are not ANSI-compliant. This precompiler uses most of the same Embedded SQL statements used in previous precompiler versions, but it processes them differently.

To migrate applications created for earlier precompiler versions:

1. Remove the following SQL statements and keywords from the application, because System 11 and later does not support them:
   - `release connection_name`
   - `recompile`
   - `noparse`
   - `noproc`
   - `pcoptions sp_syntax`
   - `cancel`

   *release* causes a precompiler error; the precompiler ignores the other keywords. The `cancel` statement causes a runtime error.

2. Use the precompiler to precompile the application again.

**Creating and running an Embedded SQL program**

Follow these steps to create and run your Embedded SQL application program:

1. Write the application program and include the Embedded SQL statements and variable declarations.
2. Save the application in a `.pco` file.
3 Precompile the application. If there are no severe errors, the precompiler generates a file containing your application program. The file has the same name as the original source file, with a different extension, depending on the requirements of your COBOL compiler. For details, see the Open Client and Open Server Programmers Supplement.

4 Compile the new source code as you would compile a standard COBOL program.

5 Link the compiled code, if necessary, with the required libraries.

6 If you specified the precompiler option to generate stored procedures, load them into Adaptive Server Enterprise by executing the generated script with isql.

7 Run the application program as you would any standard COBOL program.

How the precompiler processes your applications

The Embedded SQL precompiler translates Embedded SQL statements into COBOL data declarations and call statements. After precompiling, you can compile the resulting source program as you would any conventional COBOL program.

The precompiler processes your application in two passes. In the first pass, the precompiler parses the Embedded SQL statements and variable declarations, checking the syntax and displaying messages for any errors it detects. If the precompiler detects no severe errors, it proceeds with the second pass, wherein it does the following:

- Adds declarations for the precompiler variables, which begin with “SQL--”. To prevent confusion, do not begin your variable names with “SQL”.
- Converts the text of the original Embedded SQL statements to comments.
- Generates stored procedures and calls to stored procedures if you set this option in the precompile command line.
- Converts Embedded SQL statements to calls to runtime routines.
How the precompiler processes your applications

- Generates up to three files: a target file, an optional listing file, and an optional isql script file.

**Note** For detailed descriptions of precompiler command line options, see the Open Client and Open Server Programmers Supplement.

**Multiple Embedded SQL source files**

If the Embedded SQL application consists of more than one source file, the following statements apply:

- Connection names are unique and global to the entire application.
- Cursor names are unique for a given connection.
- Prepared statement names are global to the connection.
- Dynamic descriptors are global to the application.

**Precompiler-generated files**

The target file is similar to the original input file, except that all SQL statements are converted to runtime calls.

The listing file contains the input file and its source statements, plus any informational, warning, or error messages.

The isql script file contains the precompiler-generated stored procedures. The stored procedures are written in Transact-SQL.

**Group element referencing**

The Embedded SQL COBOL precompiler supports the COBOL language structure syntax for host variables in exec sql statements. For example, for a structure A containing structure B, which in turn contains a fundamental structure data item C, A.B.C is equivalent to C OF B OF A.

White spaces are allowed between the elements and the period (.). It is illegal to mix the two syntaxes, such as C OF A.B. Following is an example of group element referencing:
EXEC SQL BEGIN DECLARE SECTION END-EXEC.

  01 AU-ID PIC X(15).
  01 GROUP1.
  05 GROUP2.
      10 LNAME PIC X(40).
      10 FNAME PIC X(40).
      10 PHONE PIC X(15).

EXEC SQL END DECLARE SECTION END-EXEC.
...
EXEC SQL USE pubs2 END-EXEC.

MOVE "724-80-9391" TO AU-ID.
EXEC SQL SELECT INTO :GROUP1. GROUP2.LNAME, :GROUP2.FNAME, :PHONE
au_lname, au_fname, phone
FROM authors
WHERE au_id = :AU-ID END-EXEC.
DISPLAY "LAST NAME = ", LNAME.
DISPLAY "FIRST NAME = ", FNAME.
DISPLAY "PHONE #    = ", PHONE.

* This SELECT does the same thing. You can use
:GROUP1.GROUP2
* which refers to the entire structure, but partially
* qualified
* names such as :LNAME OF GROUP1 do not work.
EXEC SQL SELECT INTO :GROUP1. GROUP2
au_lname, au_fname, phone
FROM authors
WHERE au_id = :AU-ID END-EXEC.

DISPLAY "----------------------------------------".
DISPLAY "GROUP LISTING FROM ENTIRE STRUCTURES".
DISPLAY "----------------------------------------".
  DISPLAY "LAST NAME = ", LNAME.
  DISPLAY "FIRST NAME = ", FNAME.
  DISPLAY "PHONE #    = ", PHONE.

...
How the precompiler processes your applications
CHAPTER 2

General Information

This chapter provides general information about Embedded SQL.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Five tasks of an Embedded SQL program</td>
<td>9</td>
</tr>
<tr>
<td>General rules for Embedded SQL</td>
<td>11</td>
</tr>
<tr>
<td>Embedded SQL constructs</td>
<td>14</td>
</tr>
</tbody>
</table>

Five tasks of an Embedded SQL program

In addition to containing the host language code, an Embedded SQL program performs five tasks. Each Embedded SQL program must perform all these tasks, to successfully precompile, compile, and execute. Subsequent chapters discuss these five tasks.

1 Establish SQL communication using SQLCA, SQLCODE, or SQLSTATE.
   Set up the SQL communication area (SQLCA, SQLCODE, or SQLSTATE) to provide a communication path between the application program and Adaptive Server Enterprise. These structures contain error, warning and information message codes that Adaptive Server Enterprise and Client-Library generate. See Chapter 3, “Communicating with Adaptive Server Enterprise.”

2 Declare Variables.
   Identify host variables used in Embedded SQL statements to the precompiler. See Chapter 4, “Using Variables.”

3 Connect to Adaptive Server Enterprise.
   Connect the application to Adaptive Server Enterprise. See Chapter 5, “Connecting to Adaptive Server Enterprise.”

4 Send Transact-SQL statements to Adaptive Server Enterprise.
Send Transact-SQL statements to Adaptive Server Enterprise to define and manipulate data. See Chapter 6, “Using Transact-SQL Statements.”

5 Handle errors and return codes.

Handle and report errors returned by Client-Library and Adaptive Server Enterprise using SQLCA, SQLCODE, or SQLSTATE. See Chapter 8, “Handling Errors.”

**Simplified Embedded SQL program**

Following is a simplified Embedded SQL program. At this point, you need not understand everything shown in the program. Its purpose is to demonstrate the parts of an Embedded SQL program. The details are explained in subsequent chapters.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EXAMPLE.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.

* Communicating with Adaptive Server Enterprise - Chapter 3
  exec sql include sqlca end-exec.

* Declaring variables - Chapter 4
exec sql begin declare section end-exec
  01 MY-ID PIC X(30).
  01 MYPASS PIC X(30).
  01 MYSERVER PIC X(30).
exec sql end declare section end-exec.

PROCEDURE DIVISION.
MAIN-SECTION.
PARA-1.

* Initializing error-handling routines - Chapter 8
exec sql whenever sqlerror perform ERR-PARA
  through ERR-PARA-END end-exec.

* Connecting to Adaptive Server Enterprise - Chapter 5
  DISPLAY "PLEASE ENTER USER-ID".
  ACCEPT MY-ID.
```
DISPLAY "PLEASE ENTER PASSWORD".
ACCEPT MYPASS.
DISPLAY "SERVER TO USE?".
ACCEPT MYSERVER.
exe sql connect :MY-ID identified by :MYPASS
using :MYSERVER end-exec.

*Issuing Transact-SQL statements  -  Chapter 6
exec sql update alltypes set account = account * 2 end-
exec.
exec sql commit work end-exec.

*Closing connection to the server  -  Chapter 5
exec sql disconnect default end-exec.
STOP RUN.

Error-handling routine  -  Chapter 8

ERR-PARA.
DISPLAY " ERROR CODE " SQLCODE
" ERROR MESSAGE: " SQLERRMC.
ERR-PARA-END.
END PROGRAM.

**General rules for Embedded SQL**

The following rules apply to Embedded SQL statements:

- Embedded SQL statements begin with these keywords:
  exe sql

- Embedded SQL requires continuation characters in column 7 and tokens from column 8 to column 72. Place exe sql at the beginning of the statement.

- The exe sql begin declare section statement must be aligned at the correct column for data declarations for the generated declaration section to be properly aligned, and to avoid compiler warnings.

- Embedded SQL keywords are not case sensitive. exe sql, EXEC SQL, Exec Sql, or any other of case mix is equally valid. This manual consistently shows Embedded SQL keywords in lowercase. For example:
  exe sql commit work end-exec.
• All Embedded SQL statements end with the keyword `end-exec`. Place a period after `end-exec` when your program’s syntax or logic requires it. For example, the following code requires a period after `end-exec` because a COBOL paragraph must end with a period:

```cobol
PARA-1.
  IF SQLCODE = 0
    exec sql commit work end-exec.
  PARA-2.
```

In the next example, there is no period after the first `end-exec` because COBOL does not allow periods between `if` and `else`.

```cobol
IF SQLCODE NOT = 0
  exec sql rollback transaction disconnect end-exec
ELSE
  exec sql commit work end-exec.
```

• Embedded SQL statements can extend across several lines. `end-exec` must be at the end of the statement’s last line or on a new line following the last line of code.

**Statement placement**

In general, an application program can have Embedded SQL statements wherever COBOL statements are valid. However, Embedded SQL statements cannot be made until the WORKING-STORAGE SECTION of a program’s DATA DIVISION has been defined. Thus, the FILE SECTION, for example, cannot contain Embedded SQL statements.

**Comments**

Comments placed within Embedded SQL and COBOL statements must follow one of three conventions.

The Transact-SQL convention is:

```sql
/* comments */
```

The COBOL convention is:

```cobol
* (in column 7)
```

The ANSI convention is:
-- comments

Comments placed outside SQL statements must conform to COBOL programming conventions.

**Identifiers**

Identifiers are used as procedure names or data names within your application. You cannot split identifiers across lines.

**Quotation marks**

Enclose literal character strings in Embedded SQL statements within single or double quotation marks. If a character string begins with a double quotation mark, end it with a double quotation mark. If a character string begins with a single quotation mark, end it with a single quotation mark.

**Reserved words**

Do not use COBOL, Transact-SQL, or Embedded SQL reserved words except as intended by the respective languages.

You can write Embedded SQL keywords in uppercase, lowercase, or mixed case. This guide shows Embedded SQL keywords in lowercase.

**Variable naming conventions**

Embedded SQL variables must conform to COBOL naming conventions. Do not place variable names within quotation marks. Applicable quotations marks are inserted automatically when the variable names are replaced with actual values. While parsing your application, the precompiler adds declarations for variables. These declarations begin “SQL-”. So, to avoid confusion, do not begin variable names with “SQL”.

Embedded SQL/COBOL Programmers Guide 13
Scoping rules

Embedded SQL and precompiler-generated statements adhere to host language scoping rules. The `whenever` statement and cursor names are exceptions.

Statement batches

As in Transact-SQL, you can batch several SQL statements in a single `exec sql` statement. Batches are useful and more efficient when an application executes a fixed set of Transact-SQL statements each time it runs.

For example, some applications create temporary tables and indexes when they start up. You could send these statements in a single batch. See the *Adaptive Server Enterprise Reference Manual* for rules about statement batches.

The following restrictions apply to statement batches:

- Statements in a batch cannot return results to the program. That is, a batch cannot contain `select` statements.
- All statements in a batch must be valid Transact-SQL statements. You cannot place Embedded SQL statements such as `declare cursor` and `prepare` in a statement batch.
- The same rules that apply to Transact-SQL batches apply to Embedded SQL batches. For example, you cannot put a `use database` statement in an Embedded SQL batch.

Embedded SQL constructs

Table 2-1 displays valid constructs in Embedded SQL statements:
### Table 2-1: Embedded SQL constructs

<table>
<thead>
<tr>
<th>Embedded SQL construct</th>
<th>Drop construct</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin declare section</td>
<td>drop trigger</td>
</tr>
<tr>
<td>begin tran</td>
<td>drop view</td>
</tr>
<tr>
<td>begin work</td>
<td>dump database</td>
</tr>
<tr>
<td>checkpoint</td>
<td>dump tran</td>
</tr>
<tr>
<td>close <code>cursor_name</code></td>
<td>end declare section</td>
</tr>
<tr>
<td>commit tran</td>
<td>exec <code>procedure_name</code></td>
</tr>
<tr>
<td>commit work</td>
<td>execute name</td>
</tr>
<tr>
<td>connect</td>
<td>execute immediate</td>
</tr>
<tr>
<td>create database</td>
<td>fetch <code>cursor_name</code></td>
</tr>
<tr>
<td>create default</td>
<td>grant</td>
</tr>
<tr>
<td>create table</td>
<td>include sqlca or file</td>
</tr>
<tr>
<td>create index</td>
<td>insert</td>
</tr>
<tr>
<td>create unique index</td>
<td>open <code>cursor_name</code></td>
</tr>
<tr>
<td>create clustered index</td>
<td>prepare <code>statement_name</code></td>
</tr>
<tr>
<td>create nonclustered index</td>
<td>revoke</td>
</tr>
<tr>
<td>create unique clustered index</td>
<td>rollback tran</td>
</tr>
<tr>
<td>create unique nonclustered index</td>
<td>rollback work</td>
</tr>
<tr>
<td>create proc</td>
<td>select</td>
</tr>
<tr>
<td>create rule</td>
<td>set</td>
</tr>
<tr>
<td>create trigger</td>
<td>truncate</td>
</tr>
<tr>
<td>create view</td>
<td>update</td>
</tr>
<tr>
<td>declare cursor</td>
<td>use</td>
</tr>
<tr>
<td>delete</td>
<td>whenever condition action</td>
</tr>
<tr>
<td>disconnect</td>
<td></td>
</tr>
<tr>
<td>drop table</td>
<td></td>
</tr>
<tr>
<td>drop default</td>
<td></td>
</tr>
<tr>
<td>drop index</td>
<td></td>
</tr>
<tr>
<td>drop proc</td>
<td></td>
</tr>
<tr>
<td>drop rule</td>
<td></td>
</tr>
</tbody>
</table>
Embedded SQL constructs
This chapter explains how to enable an application program to receive status information from Adaptive Server Enterprise. The topics covered include:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scoping rules: SQLCA, SQLCODE, and SQLSTATE</td>
<td>18</td>
</tr>
<tr>
<td>Declaring SQLCA</td>
<td>18</td>
</tr>
<tr>
<td>Declaring SQLCODE as a standalone area</td>
<td>20</td>
</tr>
<tr>
<td>Using SQLSTATE</td>
<td>21</td>
</tr>
</tbody>
</table>

To create a communication path and declare system variables to be used in communications from Adaptive Server Enterprise to the application, you must create one of the following entities:

- A SQL Communication Area (SQLCA), which includes SQLCODE
- A standalone SQLCODE long integer
- A SQLSTATE character array

SQLCODE, SQLCA, and SQLSTATE are system variables used in communication from Adaptive Server Enterprise to the application.

After Adaptive Server Enterprise executes each Embedded SQL statement, it stores return codes in SQLCA, SQLCODE, or SQLSTATE. An application program can access the variables to determine whether the statement succeeded or failed.

**Note** The precompiler automatically sets SQLCA, SQLCODE, and SQLSTATE variables, which are critical for runtime access to the database. You need not initialize or modify them.

For details on detecting and handling errors, multiple error messages, and other return codes, see Chapter 8, “Handling Errors.”
Scoping rules: SQLCA, SQLCODE, and SQLSTATE

You can declare SQLCA anywhere in the application program where a COBOL variable can be declared. The scope of the structure follows COBOL scoping rules.

If you declare SQLCA, SQLCODE, or SQLSTATE within your file, each variable must be in scope for all executable Embedded SQL statements in the file. The precompiler generates code to set each of these status variables for each Embedded SQL statement. So, if the variables are not in scope, the generated code will not compile.

Declaring SQLCA

**Warning!** Although SQLSTATE is preferred over SQLCODE and SQLCA, this version of the precompiler supports only SQLCODE. A future version will fully support both SQLCA and SQLSTATE.

Declare SQLCA in your application program’s WORKING-STORAGE SECTION. The syntax for declaring SQLCA is:

```sql
exec sql include sqlca [is external] [is global]
end-exec.
```

Multiple SQLCAs

Because a single file can contain multiple COBOL programs, you may have multiple SQLCAs. However, each SQLCA must be in a separate WORKING-STORAGE SECTION.

SQLCA variables

When the precompiler encounters the `include sqlca` statement, it inserts the SQLCA structure declaration into the application program. SQLCA is a data structure containing 26 precompiler-determined *system variables*, each of which can be accessed independently.
SQLCA variables pass information to your application program about the status of the most recently executed Embedded SQL statement.

Table 3-1 describes the SQLCA variables that hold status information, return codes, error codes, and error messages generated by Adaptive Server Enterprise:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Datatype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLCAID</td>
<td>PIC X(8)</td>
<td>Text string that contains “SQLCA”.</td>
</tr>
<tr>
<td>SQLCABC</td>
<td>PIC S9(9) COMP</td>
<td>Length of SQLCA.</td>
</tr>
<tr>
<td>SQLCODE</td>
<td>PIC S9(9) COMP</td>
<td>Contains the return code of the most recently executed SQL statement. See the SQLCODE values in Table 3-2 on page 21 for return code definitions.</td>
</tr>
<tr>
<td>SQLWARN0 to SQLWARN7</td>
<td>PIC X(1)</td>
<td>Warning flags. Each flag indicates whether a warning has been issued: a “W” for warning, or a blank space for no warning. Chapter 8 describes the SQLWARN flags.</td>
</tr>
<tr>
<td>SQLERRMC</td>
<td>PIC X(256)</td>
<td>Error message.</td>
</tr>
<tr>
<td>SQLERRML</td>
<td>PIC S9(9) COMP</td>
<td>Error message length.</td>
</tr>
<tr>
<td>SQLERRP</td>
<td>PIC X(8)</td>
<td>Procedure that detected error/warning.</td>
</tr>
<tr>
<td>SQLERRD</td>
<td>PIC S9(9) COMP OCCURS 6 TIMES</td>
<td>Details of error/warning. SQLERRD(3) is number of rows affected.</td>
</tr>
</tbody>
</table>

### Accessing SQLCA variables

The SQLCA variables listed in the previous section provide additional information about errors and return codes to help in debugging as well as in the normal processing of your application.

**Warning!** Do not define both a SQLCODE and a SQLCA as SQLCODE, as SQLCODE is a field within the SQLCA structure.
**Declaring SQLCODE as a standalone area**

**SQLCODE within SQLCA**

The application should test SQLCODE after each statement executes, because Adaptive Server Enterprise updates it after each execution. As a rule, use the whenever statement, described in Chapter 8, “Handling Errors,” to perform the SQLCODE test.

Following are examples of using SQLCODE:

\[
\begin{align*}
\text{IF } \text{SQLCODE} = 100 \\
&\text{PERFORM END-DATA-PARA.}
\end{align*}
\]

or

\[
\begin{align*}
\text{DISPLAY "SQL status code is" SQLCODE.}
\end{align*}
\]

**Declaring SQLCODE as a standalone area**

**Note** Although SQLSTATE is preferred over SQLCODE and SQLCA, this version of the precompiler supports only SQLCODE. A future version will fully support both SQLCA and SQLSTATE.

As an alternative to creating a SQLCA, use SQLCODE independently. It contains the return code of the most recently executed SQL statement. The benefit of declaring SQLCODE as a standalone area is that it executes code faster. If you have no need to review the other information that SQLCA holds and are interested only in return codes, consider using SQLCODE.

Despite SQLCODE’s faster execution speed, SQLSTATE is preferred over SQLCODE because SQLCODE is a deprecated feature that is compatible with earlier versions of Embedded SQL.

**Warning!** Do not declare SQLCODE within a `declare` section.

Following is an example of declaring SQLCODE as a standalone area:

```
01 SQLCODE S9(9) COMP.
   exec sql open cursor pub_id   end-exec.
PARAGRAPH-1:
   exec sql fetch pub_id into :PUB_NAME end-exec.
IF SQLCODE = 0 GOTO PARAGRAPH-1.
```
CHAPTER 3  Communicating with Adaptive Server Enterprise

For details on debugging any errors SQLCODE indicates, see Chapter 8, “Handling Errors.”

Table 3-2 displays SQLCODE values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Statement executed successfully.</td>
</tr>
<tr>
<td>-n</td>
<td>Error occurred. See Server-Library or Client-Library error messages. -n represents the number associated with the error or exception.</td>
</tr>
<tr>
<td>+100</td>
<td>No data exists, no rows left after fetch, or no rows met search condition for update, delete, or insert.</td>
</tr>
</tbody>
</table>

### Using SQLSTATE

**Warning!** Although SQLSTATE is preferred over SQLCODE and SQLCA, this version of the precompiler supports only SQLCODE. A future version will fully support both SQLCA and SQLSTATE.

SQLSTATE is a status parameter. Its codes indicate the status of the most recently attempted statement—either the statement completed successfully or an error occurred during the execution of the statement.

The following example illustrates a declaration of SQLSTATE:

```cobol
WORKING-STORAGE SECTION.
01 SQLSTATE PIC X(5).
...
exec sql whenever sqlerror perform ERR-PARA
end-exec
...

ERR-PARA.
  IF sqlstate = "ZD000" or
    sqlstate = "ZE000" or
    sqlstate = "ZF000" or
    sqlstate = "ZG000" or
    sqlstate = "ZH000"
    DISPLAY "Unexpected results were ignored"
  ELSE
    IF sqlstate = "08001" or sqlstate = "08000"
```
Using SQLSTATE

DISPLAY "Connection failed-quitting"
STOP RUN
ELSE
   DISPLAY "A non-results, non-connect — error occurred"
END_IF
END_IF

Table 3-3 lists SQLSTATE values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00XXX</td>
<td>Successful execution</td>
</tr>
<tr>
<td>01XXX</td>
<td>Warning</td>
</tr>
<tr>
<td>02XXX</td>
<td>No data exists; no rows affected</td>
</tr>
<tr>
<td>Any other value</td>
<td>Error</td>
</tr>
</tbody>
</table>

Obtaining SQLSTATE codes and error messages

SQLSTATE can contain a list of one or more error and/or warning messages. The messages can be informational, warning, severe, or fatal messages. Open Client Client-Library and Open Server Server Library generate the majority of SQLSTATE messages. See the appropriate documentation for a complete list of SQLSTATE codes and error messages.

See Appendix A, “Precompiler Warning and Error Messages,” for the table of SQLSTATE messages that the precompiler can generate.
CHAPTER 4

Using Variables

This chapter details the following two types of variables that pass data between your application and Adaptive Server Enterprise:

- Host variables, which are COBOL variables you use in Embedded SQL statements to hold data that is retrieved from and sent to Adaptive Server Enterprise
- Indicator variables, which you associate with host variables to indicate null data and data truncation

Declaring variables

As discussed in Chapter 3, “Communicating with Adaptive Server Enterprise,” the precompiler automatically sets the system variables when you include SQLCA, SQLCODE, or SQLSTATE in the application program. However, you must explicitly declare host and indicator variables in a declare section before using them in Embedded SQL statements.

**Warning!** The precompiler generates some variables, all of which begin with “SQL--”. Do not begin your variables with “SQL,” or you may receive an error message or unreliable data.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaring variables</td>
<td>23</td>
</tr>
<tr>
<td>Using host variables</td>
<td>25</td>
</tr>
<tr>
<td>Using indicator variables</td>
<td>27</td>
</tr>
<tr>
<td>Using arrays</td>
<td>31</td>
</tr>
<tr>
<td>Scoping rules</td>
<td>31</td>
</tr>
<tr>
<td>Datatypes</td>
<td>33</td>
</tr>
</tbody>
</table>
Declaring variables

You cannot use COPY statements in a declare section. The syntax for a declare section is:

```
exec sql begin declare section end-exec
  declarations ...
exec sql end declare section end-exec.
```

Host variable declarations must conform to the COBOL rules for data declarations. You need not declare all variables in one declare section, since you can have an unlimited number of declare sections in a program.

**Note** Version 11.1 and later does not support updates to the PIC clause.

When declaring variables, you must also specify the picture and usage clauses. For valid picture and usage clauses, see the section “Comparing COBOL and Adaptive Server Enterprise datatypes” on page 35.

The following example shows a sample declare section:

```
exec sql begin declare section end-exec
  01  E-NAME   PIC X(30).
  01  E-TYPE   PIC X(3).
  01  TINY-INT PIC S9(2) COMP.
  01  SHORT-INT PIC S9(4) COMP.
  01  MONEY-DATA CS-MONEY.
exec sql end declare section end-exec.
```

Declaring a character array

The precompiler supports *complex definitions*, which are structures and arrays. You can nest structures, but you cannot have an array of structures.

The precompiler recognizes single-dimensional arrays of all datatypes. The precompiler also recognizes double-dimensional arrays of characters, as demonstrated in the following example:

```
  01 NUMSALES PIC S9(9) OCCURS 25 TIMES.
exec sql begin declare section end-exec.
  01 DAYS-OF-THE-WEEK PIC X(31) OCCURS 7 TIMES.
exec sql end declare section end-exec.
```

For details on arrays, see “Using arrays” on page 31.
Using host variables

Host variables let you transfer values between Adaptive Server Enterprise and the application program.

Declare the host variable within the application program’s Embedded SQL declare section. Only then can you use the variable in SQL statements.

When you use the variable within an Embedded SQL statement, prefix the host variable with a colon. When you use the variable elsewhere in the program, do not use a colon. When you use several host variables successively in an Embedded SQL statement, separate them with commas or follow the grammar rules of the SQL statement.

The following example demonstrates correct host variable usage. \texttt{PAR-1}, \texttt{PAR-2}, and \texttt{PAR-3} are declared as host variables and are then used as parameters to the \texttt{myproc} procedure:

```sql
exec sql begin declare section end-exec
01 PAR-1 PIC X(10).
01 PAR-2 PIC X(10).
01 PAR-3 PIC X(10).
exec sql end declare section end-exec
exec sql exec myproc :PAR-1, :PAR-2, :PAR-3 end-exec.
```

There are four ways to use host variables:

- Input variables for SQL statements and procedures
- Result variables
- Status variables from calls to SQL procedures
- Output variables for SQL statements and procedures

Regardless of their function, declare all host variables as described in “Declaring variables” on page 23. Following are instructions for using host variables.

Host input variables

These variables pass information to Adaptive Server Enterprise. The application program assigns values to them. They hold data used in executable statements such as stored procedures, \texttt{select} statements with \texttt{where} clauses, \texttt{insert} statements with \texttt{values} clauses, and \texttt{update} statements with \texttt{set} clauses.

The following example uses the \texttt{TITLE-ID1}, \texttt{TITLE-ID2}, and \texttt{PUB-ID} variables as input variables:
Using host variables

exec sql begin declare section end-exec
   01 TITLE-ID1       PIC X(6).
   01 TITLE-ID2       PIC X(6).
   01 PUB-ID          PIC X(4).
exec sql end declare section end-exec

exec sql delete from titles
   where title_id = :TITLE-ID1 end-exec.
exec sql update titles set pub_id = :PUB-ID
   where title_id = :TITLE-ID2 end-exec.

Host result variables

These variables receive the results of `select` and `fetch` statements.

The following example uses the `TITLE-ID` variable as a result variable:

exec sql begin declare section end-exec
   01 TITLE-ID       PIC X(6).
exec sql end declare section end-exec

exec sql select title_id into :TITLE-ID from titles
   where pub_id = "0736"
   and type = "business" end-exec.

Host status variables

These variables receive the return status values of stored procedures. Status
variables indicate whether the stored procedure completed successfully or the
reasons it failed. You must use a variable that can be converted from the
Adaptive Server Enterprise type to `smallint`.

The following example uses the `RET-CODE` variable as a status variable:

exec sql begin declare section end-exec
   01 RET-CODE       PIC S9(4) COMP.
exec sql end declare section end-exec.
   ...
exec sql exec :RET-CODE = update_pubs end-exec.
if RET-CODE NOT = 0
   exec sql rollback transaction end-exec.
Host output variables

These variables pass data from stored procedures to the application program. Use host output variables when stored procedures return the value of parameters declared as **out**. See “Using stored procedures” on page 61.

The following example uses the \texttt{PAR1} and \texttt{PAR2} variables as output variables:

\begin{verbatim}
exec sql exec a_proc :PAR1 out, :PAR2 out end-exec.
\end{verbatim}

Using indicator variables

You can associate indicator variables with host variables to indicate when a database value is null. Use a space and, optionally, the \texttt{indicator} keyword to separate each indicator variable from the host variable with which it is associated. Each indicator variable must immediately follow its host variable.

Without indicator variables, Embedded SQL cannot indicate null values.

Indicator variables and server restrictions

Embedded SQL is a generic interface that can run on a variety of servers, including Adaptive Server Enterprise.

Because it is generic, Embedded SQL does not enforce or reflect any particular server’s restrictions. For example, Embedded SQL allows text and image stored procedure parameters, but Adaptive Server Enterprise does not.

When writing an Embedded SQL application, keep the application’s ultimate target server in mind. If you are unsure about what is legal on a server and what is not, consult your server documentation.

Using host variables with indicator variables

Declare host and indicator variables in a \texttt{declare} section before using them anywhere in an application program containing Embedded SQL statements.

You must declare indicator variables as one of the following in a \texttt{declare} section:
Using indicator variables

PIC S9(4) COMP
DISPLAY SIGN LEADING (and, optionally, SEPARATE)
DISPLAY SIGN TRAILING (and, optionally, SEPARATE)
COMP-3
COMP-4
COMP-5
BINARY

Prefix indicator variables with a colon when using them in an Embedded SQL statement. The syntax for associating an indicator variable with a host variable is:

:host_variable [[indicator] :indicator_variable]

The association between an indicator and host variable lasts only for the duration of one exec sql statement.

Adaptive Server Enterprise sets the indicator variable only when you assign a value to the host variable. Therefore, you can declare an indicator variable once and reuse it with different host variables in different statements.

You can use indicator variables with output, result, and input variables. When used with output and result variables, Embedded SQL sets the variable to indicate the null status of the associated host variable. When used with input variables, you set the value of the indicator variable to show the null status of the input variable before submitting it to Adaptive Server Enterprise.

Note You can use indicator variables with output, result, and input variables.

Using indicator variables with host output and result variables

When you associate an indicator variable with an output or result variable, Client-Library automatically sets it to one of the following values in Table 4-1:
Table 4-1: Indicator variable values used with output or result variable

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>The corresponding database column in Adaptive Server Enterprise contains a null value.</td>
</tr>
<tr>
<td>0</td>
<td>A non-null value was assigned to the host variable.</td>
</tr>
<tr>
<td>&gt;0</td>
<td>An overflow occurred while data was being converted for the host variable. The host variable contains truncated data. The positive number represents the length, in bytes, of the value before it was truncated.</td>
</tr>
</tbody>
</table>

The following example demonstrates associating the INDIC-V indicator variable with the PUB-NAME result variable:

```sql
exec sql begin declare section end-exec
  01 INDIC-V PIC S9(4) COMP.
  01 PUB-ID  PIC X(4).
  01 PUB-NAME PIC X(20).
exec sql end declare section end-exec

exec sql select pub_name into :PUB-NAME :INDIC-V
  from publishers where pub_id = :PUB-ID
end-exec.

if INDIC-V = -1
  display "No Publisher name"
else
  display "Publisher Name is: " PUB-NAME.
```

Using indicator variables with host input variables

When you associate an indicator variable with an input variable, you must explicitly set the indicator variable, using the values in Table 4-2 as a guide.

Table 4-2: Indicator variable values used with input variable

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>Treat the corresponding input as a null value.</td>
</tr>
<tr>
<td>0</td>
<td>Assign the value of the host variable to the column.</td>
</tr>
</tbody>
</table>

You must supply host language code to test for a null input value and set the indicator variable to -1. This informs Client-Library of a null value. When you set the indicator variable to -1, null is used regardless of the host variable’s actual value.
Using indicator variables

The following example demonstrates associating an indicator variable with an input variable. The database royalty column will be set to a null value because R-INDIC is set to -1. Changing the value of R-INDIC changes the value of royalty.

```sql
exec sql begin declare section end-exec
  01 R-INDIC PIC S9(4) COMP.
  01 R-VAR PIC X(10).
exec sql end declare section end-exec.
  MOVE -1 TO R-INDIC.
  exec sql update titles
    set royalty = :R-VAR :R-INDIC
    where pub_id="0736" end-exec.
```

Host variable conventions

A host variable name must conform to COBOL naming conventions.

You can use a host variable in an Embedded SQL statement only if a Transact-SQL literal can be used in a Transact-SQL statement at the same location.

A host variable must conform to the valid precompiler datatypes. The datatype of a host variable must be compatible with the datatype of the database column values that are returned. See Table 4-3 on page 37 and Table 4-4 on page 38 for details.

Do not use host language reserved words and Embedded SQL keywords as variable names.

A host variable cannot represent Embedded SQL keywords or database objects, except as specified in dynamic SQL. See Chapter 4, “Using Variables.”

When a host variable represents a character string in a SQL statement, do not place it within quotes.

The following example is invalid because the precompiler inserts quotes around values when necessary. You should not type the quotes.

```sql
exec sql select pub_id from publishers
  where pub_id like "*:PUB-ID"
end-exec
```

The following example is valid:

```sql
exec sql select pub_id from publishers
```
Using arrays

An array is a group of related pieces of data associated with one variable. You can use arrays as output variables for the into clause of select and fetch statements. For example:

```
01 author-array.
10 author-name    PIC X(30)   occurs 100 times.
exec sql
  select au_lname
  from authors
  into :au_array
end-exec.
```

**Note** You can fetch a single item anywhere into an array. However, you can fetch multiple rows only into the beginning of an array.

For details on using arrays with select and fetch, see “Selecting multiple rows through arrays” on page 47 in Chapter 6.

Multiple arrays

When you use multiple arrays within a single SQL statement, they must be the same size. Otherwise, you will receive an error message.

Scoping rules

The precompiler supports nested COBOL programs and COBOL’s rules for variable scoping. Host variables can use the is global and is externa] clauses. Following is a nested example:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. outer.
ENVIRONMENT DIVISION.
```
Scoping rules

CONFIGURATION SECTION.
SOURCE-COMPUTER.  xyz.
OBJECT-COMPUTER.  xyz.
DATA DIVISION.
WORKING-STOREAGE SECTION.
  exec sql begin declare section end-exec.
    01 global-var is global pic x(10).
    01 not-global-var pic x(10).
    01 shared-var is external pic x(10).
  exec sql end declare section end-exec.
procedure division.
p0.
...
IDENTIFICATION DIVISION.
PROGRAM-ID.  inner.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.  xyz.
OBJECT-COMPUTER.  xyz.
DATA DIVISION.
WORKING-STOREAGE SECTION.
procedure division.
p0.
...
* This is legal because global-var was
* declared using is global
  exec sql
    select au_lname into :global-var
      where au_id = "998-72-3567"
  end-exec.
* This is not legal because not-global-var was
* not declared using is global
  exec sql
    select au_lname into :not-global-var
      where au_id = "998-72-3567"
  end-exec.
...
end program inner.
end program outer.
IDENTIFICATION DIVISION.
PROGRAM-ID.  nonest.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.  xyz.
OBJECT-COMPUTER.  xyz.
DATA DIVISION.
CHAPTER 4    Using Variables

WORKING-STORAGE SECTION.
  exec sql begin declare section end-exec.
    01 local-var pic x(10).
    01 shared-var is external pic x(10).
  exec sql end declare section end-exec.
procedure division.
p0.
  . . .
  * This is legal.
  exec sql
    select au_lname into :local-var
    where au_id = "998-72-3567"
  end-exec.
  * So is this.
  exec sql
    select au_lname into :shared-var
    where au_id = "998-72-3567"
  end-exec.
  . . .
end program nonest.

Datatypes

The COBOL veneer layer is a library used by the precompiled application along with Open Client Client-Library. The COBOL code generated by the precompiler calls functions in the veneer layer, each of which calls a specific Client-Library function. The veneer layer performs conversions and other operations that make it possible for COBOL to communicate with Client-Library. The veneer layer also provides conversions that translate between COBOL host variables and Adaptive Server Enterprise datatypes.

There are two types of ESQL/COBOL veneer layers: static and shared dynamic. The following table lists the shared dynamic veneer layer libraries that are released on all 32-bit and 64-bit platforms:

<table>
<thead>
<tr>
<th>Platform</th>
<th>Library name</th>
<th>Reentrant version</th>
</tr>
</thead>
<tbody>
<tr>
<td>HP-UX PA-RISC 32-bit</td>
<td>libsybcobct.sl</td>
<td>libsybcobct_r.sl</td>
</tr>
<tr>
<td>HP-UX PA-RISC 64-bit</td>
<td>libsybcobct64.sl</td>
<td>libsybcobct_r64.sl</td>
</tr>
<tr>
<td>All other 32-bit platforms that support ESQL/COBOL</td>
<td>libsybcobct.so</td>
<td>libsybcobct_r.so</td>
</tr>
</tbody>
</table>
The existing static version of the ESQL/COBOL veneer layer library is called \textit{libsybcobct.a}.

There are two types of data items: elementary and group data items. The following subsections describe these types of data items.

**Elementary data items**

An \textit{elementary data item} is a complete item that cannot be broken into separate parts. You can use elementary data items as host variables.

Following is an example of an elementary data item:

\begin{verbatim}
  01 MYSTR  PIC X(26).
\end{verbatim}

You can use \textit{MYSTR} as a host variable (:MYSTR) because it is an elementary data item.

**Group data items**

When multiple elementary data items combine to form a group of related items they become a \textit{group data item}. You can use group data items as host variables. Declare group data items in \texttt{declare} sections.

Following is an example of a group item:

\begin{verbatim}
  01 AUTH-REC.
  10 AUTH-NAME           PIC X(25).
  10 STATE               PIC X(25).
  10 TOTAL-SALES PIC S9(9) COMP SYNC
\end{verbatim}

Following is an example of selecting into a group item whose data items are host variables:

\begin{verbatim}
  exec sql select au_lname, salary, tot_sales
  from table into :AUTH-REC end-exec
\end{verbatim}

The preceding example has the same effect as the following code:

\begin{verbatim}
  exec sql select au_lname, salary, tot_sales
  from table into :AUTH-NAME, :SALARY, :TOTAL-SALES
\end{verbatim}
Another equivalent example is:

```sql
exec sql select au_lname, salary, tot_sales
from table into :AUTH-NAME OF AUTH-REC,
:TOTAL-SALES OF AUTH-REC
```

Embedded SQL/COBOL also supports C language structure syntax for host variables in `exec sql` statements. For example, the preceding example could be rewritten as follows:

```sql
exec sql select au_lname, salary, tot_sales
from table into :AUTH-REC.AUTH-NAME,
:AUTH-REC.TOTAL-SALES
```

Use SYNC with COMP, COMP-4, COMP-5, and BINARY data items declared within group data items.

### Special data items

Special Sybase datatypes, such as CS_MONEY, CS-TEXT, and CS-IMAGE are declared as shown in the following example:

```cobol
01 MYTEXT PIC x(100) USAGE IS CS-TEXT.
```

### Comparing COBOL and Adaptive Server Enterprise datatypes

Host variable datatypes must be compatible with the datatypes of the corresponding database columns. So, before writing your application program, check the datatypes of the database columns.

The following rules apply to datatypes:

- When you use any of the host variables in the “To: COBOL Datatype” column as input or output, the appropriate conversions occur automatically.
- Indicator variables must be of usage COMP, COMP-3, COMP-4, COMP-5, BINARY, or a variant of DISPLAY. They must have a picture string of S9(4) or equivalent.
- You can use any value with PIC S9(1-9) COMP. If decimal truncation occurs, no truncation message results. Instead, a SQLCA or SQLSTATE error message results, which specifically indicates digital truncation.
Datatypes

For example, if you select the value “1234” into a PIC S9(4), no truncation message occurs because the value fits in the given bytes. However, if you select “1234567” into PIC S9(3), a truncation message results because the value does not fit in the given bytes.

Converting datatypes

The precompiler automatically compares the datatypes of host variables with the datatypes of table columns in Adaptive Server Enterprise. If the Adaptive Server Enterprise datatype and the host language datatype are compatible but not identical, the COBOL veneer layer converts one type to the other. Datatypes are compatible if the precompiler can convert the data from one type to the other. If the datatypes are incompatible, a conversion error occurs at runtime and SQLCODE or SQLSTATE is set to a negative number.

Be careful when converting a longer datatype into a shorter one, such as a long integer into PIC S9(4) COMP, because there is always a possibility of truncating data. If a truncation occurs, SQLWARN1 is set.

Note Do not fetch Adaptive Server Enterprise data into COBOL numeric fields that contain editing characters such as commas and decimal characters. Instead, fetch the data into an unedited field such as COMP or DISPLAY SIGN LEADING SEPARATE and then move the data into an edited field.

Converting datatypes for result variables

Table 4-3 shows which data conversions are valid for result variables. A bullet indicates that conversion is possible, but be aware that certain types of errors can result if you are not careful when choosing host variable datatypes.
### Table 4-3: Datatype conversions for result variables

<table>
<thead>
<tr>
<th>From: Adaptive Server Enterprise datatype</th>
<th>To: COBOL datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>•</td>
</tr>
<tr>
<td>varchar</td>
<td>•</td>
</tr>
<tr>
<td>bit</td>
<td>•</td>
</tr>
<tr>
<td>tinyint</td>
<td>•</td>
</tr>
<tr>
<td>smallint</td>
<td>•</td>
</tr>
<tr>
<td>int</td>
<td>•</td>
</tr>
<tr>
<td>bigint</td>
<td>•</td>
</tr>
<tr>
<td>ubigint</td>
<td>•</td>
</tr>
<tr>
<td>uint</td>
<td>•</td>
</tr>
<tr>
<td>usmallint</td>
<td>•</td>
</tr>
<tr>
<td>float</td>
<td>•</td>
</tr>
<tr>
<td>money</td>
<td>•</td>
</tr>
<tr>
<td>money4</td>
<td>•</td>
</tr>
<tr>
<td>numeric</td>
<td>•</td>
</tr>
<tr>
<td>real</td>
<td>•</td>
</tr>
<tr>
<td>date</td>
<td>•</td>
</tr>
<tr>
<td>time</td>
<td>•</td>
</tr>
<tr>
<td>datetime</td>
<td>•</td>
</tr>
<tr>
<td>datetime4</td>
<td>•</td>
</tr>
</tbody>
</table>

Key: DSL = Display Sign Leading
DSLs = Display Sign Leading Separate
DST = Display Sign Trailing
DSTS = Display Sign Trailing Separate

### Converting datatypes for input variables

Table 4-4 shows which data conversions are valid for input variables. A bullet indicates that conversion is possible. Errors, including truncation, can result if you choose nonconvertible host variable datatypes.
### Table 4-4: Datatype conversions for input variables

<table>
<thead>
<tr>
<th>From: COBOL datatype</th>
<th>varchar</th>
<th>money</th>
<th>date, time, int, smallint, bigint, ubigint, uint, usmallint, bit, float, char, numeric, real, float</th>
</tr>
</thead>
<tbody>
<tr>
<td>S9(1—9) COMP, COMP-4, COMP-5, BINARY</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>CS-DATE, CS-TIME, CS-DATETIME, CS-DATETIME4</td>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PIC X(n)</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>S9(m)V9(n) DSLS, DSL, DSTS, DST, COMP-3</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>CS-MONEY, CS-MONEY4</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
</tbody>
</table>

Key: DSL = Display Sign Leading
DSLS = Display Sign Leading Separate
DST = Display Sign Trailing
DSTS = Display Sign Trailing Separate
Connecting to Adaptive Server Enterprise

This chapter explains how to connect an Embedded SQL program to Adaptive Server Enterprise and describes how to specify servers, user names, and passwords. Topics include:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connecting to a server</td>
<td>39</td>
</tr>
<tr>
<td>Changing the current connection</td>
<td>41</td>
</tr>
<tr>
<td>Establishing multiple connections</td>
<td>41</td>
</tr>
<tr>
<td>Disconnecting from a server</td>
<td>44</td>
</tr>
</tbody>
</table>

Connecting to a server

Use the `connect` statement to establish a connection between an application program and Adaptive Server Enterprise. If an application uses both C and COBOL languages, the first `connect` statement must be issued from a COBOL program.

The syntax for the `connect` statement is:

```sql
exec sql connect :user [identified by :password]
     [at :connection_name] [using :server]
     [label_name label_name label_value label_value...]
end-exec
```

Each of the following sections describes one of the `connect` statement’s arguments. Only the `user` argument is required for the `connect` statement. The other arguments are optional.
Connecting to a server

**user**

`user` is a host variable or quoted string that represents a Adaptive Server Enterprise user name. The user name must be valid for the server specified.

**password**

`password` is a host variable or quoted string that represents the password associated with the specified user name. This argument is necessary only if a password is required to access Adaptive Server Enterprise. If the password argument is null, the user does not need to supply a password.

**connection_name**

`connection_name` uniquely identifies the Adaptive Server Enterprise connection. It can be a double-quoted or an unquoted literal. You can create an unlimited number of connections in an application program, one of which can be unnamed. `connection_name` has a maximum size of 255 characters.

When you use `connection_name` in a `connect` statement, all subsequent Embedded SQL statements that specify the same connection automatically use the server indicated in the `connect` statement. If the `connect` statement specifies no server, the `default` server is used. See the Open Client and Open Server Programmers Supplement for details on how the default server is determined.

**Note** To change the current server connection, use the `set connection` statement described in “Changing the current connection” on page 41.

An Embedded SQL statement should reference only a `connection_name` specified in a `connect` statement. At least one `connect` statement is required for each server that the application program uses.

**server**

`server` is a host variable or quoted string that represents a server name. `server` must be a character string that uniquely and completely identifies a server.
connect example

The following example uses the UNIX format to connect to the server SYBASE.

```sql
exec sql begin declare section end-exec
01 USER PIC X(16) VALUE "myname"
01 PASSWD PIC X(16) VALUE "abcdefg".
01 SERV-NAME PIC X(16).
01 MY-SERVER PIC X(512).
exec sql end declare section end-exec.
MOVE "SYBASE" TO SERV-NAME.
exec sql connect :USER identified by :PASSWD using :SERV-NAME end-exec.
```

Changing the current connection

Use the `set connection` statement to change the current connection. The statement’s syntax is:

```sql
exec sql set connection {connection_name | default}
```
where “default” is the unnamed connection, if any.

The following example changes the current connection:

```sql
exec sql connect "ME" at connect1 using "SERVER1" end-exec
exec sql connect "ME" at connect2 using "SERVER2" end-exec
exec sql set connection connect1 end-exec
exec-sql select user_id() into :MYID end-exec
```

Establishing multiple connections

Some Embedded SQL applications require or benefit from having more than one active Adaptive Server Enterprise connection. For example:

- An application that requires multiple Adaptive Server Enterprise login names can have a connection for each login account name.
Establishing multiple connections

- By connecting to more than one server, an application can simultaneously access data stored on different servers.

A single application can have multiple connections to a single server or multiple connections to different servers. Use the `connect` statement's `at` clause to name additional connections for an application.

If you open one connection and then another new named or unnamed connection, the new connection is the current connection.

**Note** If you are creating stored procedures with the precompiler for appropriate SQL statements with the precompiler, then for each Embedded SQL file, the precompiler generates a single file for all stored procedures on all servers. You can load this file into the appropriate server(s). Although the server(s) will report warnings and errors about being unable to read the procedures intended for other servers, ignore them. The stored procedures appropriate for each server will load properly on that server. Be sure to load the stored procedures on all applicable servers or your queries fail.

Naming a connection

Table 5-1 shows how a connection is named:

<table>
<thead>
<tr>
<th>If this clause is used</th>
<th>Without this clause</th>
<th>The connection name is</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>at connection_name</code></td>
<td><code>connection_name</code></td>
<td></td>
</tr>
<tr>
<td><code>using server_name at</code></td>
<td><code>server_name</code></td>
<td></td>
</tr>
<tr>
<td>None</td>
<td>The actual name of the connection &quot;DEFAULT&quot;</td>
<td></td>
</tr>
</tbody>
</table>

Invalid statements with the `at` clause

The following statements are invalid with the `at` clause:

- `connect`
- `begin declare section`
- `end declare section`
- `include file`
Using Adaptive Server Enterprise connections

Specify a connection name for any Embedded SQL statement that you want to execute on a connection other than the default unnamed connection. If your application program uses only one connection, you can leave the connection unnamed and omit the at clause.

The syntax for using multiple connections is:

\[
\text{exec sql [at connection_name] sql_statement} \\
\text{end-exec}
\]

where \textit{sql_statement} is a Transact-SQL statement.

The following example shows how two connections can be established to different servers and used in consecutive statements:

\[
\text{exec sql begin declare section end-exec} \\
01 \text{USER} \text{ PIC X(16) VALUE "myname".} \\
01 \text{PASSWD } \text{ PIC X(16) VALUE "mypass".} \\
01 \text{AU-NAME } \text{ PIC X(20).} \\
01 \text{A-VALUE } \text{ PIC S9(9) COMP.} \\
01 \text{A-TEST } \text{ PIC S9(9) COMP.} \\
01 \text{SERVER-1 } \text{ PIC X(16).} \\
01 \text{SERVER-2 } \text{ PIC X(16).} \\
\text{exec sql end declare section end-exec.} \\
\]

\[
\text{. . .} \\
\text{MOVE "sybase1" TO SERVER-1.} \\
\text{MOVE "sybase2" TO SERVER-2.} \\
\text{exec sql connect :USER identified by :PASSWD using :SERVER-1 end-exec.} \\
\text{exec sql connect :USER identified by :PASSWD at connection-2 using :SERVER-2 end-exec.} \\
\]

* This statement uses the current connection
* (connection-2)

\[
\text{exec sql select royalty into :A-VALUE from pubs} \\
\text{where author = :AU-NAME end-exec.} \\
\]

* This statement uses connection "SERVER-1"
Disconnecting from a server

IF A-VALUE = A-TEST
exec sql at SERVER-1 update titles
   set column = :A-VALUE * 2
   where author = :AU-NAME end-exec.

Disconnecting from a server

The connections your application program establishes remain open until you explicitly close them or until your program terminates. Use the disconnect statement to close a connection between the application program and Adaptive Server Enterprise.

The statement’s syntax is as follows:

exec sql disconnect {connection_name | current | default | all} end-exec

where:

• current specifies the current connection.
• default specifies the unnamed default connection.
• all specifies all connections currently open.

The disconnect statement:

1 Rolls back the transaction, ignoring any established savepoints.
2 Closes the connection.
3 Drops all temporary objects, such as tables.
4 Closes all open cursors.
5 Releases locks established for the current transactions.
6 Terminates access to the server’s databases.

disconnect does not implicitly commit current transactions.

Warning! Before the program exits, make sure you perform an exec sql disconnect or exec sql disconnect all statement for each open connection. In some configurations, Adaptive Server Enterprise may not be notified when a client exits without disconnecting. If this happens, resources held by the application will not be released.
CHAPTER 6  Using Transact-SQL Statements

This chapter explains how to use Transact-SQL statements with Embedded SQL and host variables. It also explains how to use stored procedures, which are collections of SQL statements stored in Adaptive Server Enterprise. Since stored procedures are compiled and saved in the database, they execute quickly without being recompiled each time you invoke them.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transact-SQL statements in Embedded SQL</td>
<td>45</td>
</tr>
<tr>
<td>Selecting rows</td>
<td>46</td>
</tr>
<tr>
<td>Grouping statements</td>
<td>64</td>
</tr>
<tr>
<td>Including files and directories</td>
<td>67</td>
</tr>
</tbody>
</table>

Transact-SQL statements in Embedded SQL

exec sql syntax

Embedded SQL statements must begin with the keywords exec sql and end with the keyword end-exec. The syntax for Embedded SQL statements is:

```
exec sql [at connection_name] sql_statement end-exec
```

where:

- `connection_name` specifies the connection for the statement. See Chapter 5, “Connecting to Adaptive Server Enterprise,” for a description of connections. The at keyword is valid for Transact-SQL statements and the disconnect statement.
- `sql_statement` is one or more Transact-SQL statements.
Invalid statements

Except for the following Transact-SQL statements, all Transact-SQL statements are valid in Embedded SQL:

- print
- raiserror
- readtext
- writetext

Transact-SQL statements that differ in Embedded SQL

While most Transact-SQL statements retain their functionality and syntax when used in Embedded SQL, the select, update, and delete statements (the Data Manipulation Language, or DML, statements) can be slightly different in Embedded SQL:

- The following four items are specific to the into clause of the select statement.
  - The into clause can assign one row of data to scalar host variables. This clause is valid only for select statements that return just one row of data. If you select multiple rows, a negative SQLCODE results, and only the first row is returned.
  - If the variables in an into clause are arrays, you can select multiple rows. If you select more rows than the array holds, an exception of SQLCODE <0 is raised, and the extra rows are lost.
  - select cannot return multiple rows of data in host variables, except through a cursor or by selecting into an array.
  - The update and delete statements can use the search condition where current of cursor_name.

Selecting rows

There can be a maximum of 1024 columns in a select statement. For the complete listing of the select statement’s syntax, see the Adaptive Server Enterprise Reference Manual.
Selecting one row

When you use the `select` statement without a cursor or array, it can return only one row of data. Embedded SQL requires a cursor or an array to return more than one row of data.

In Embedded SQL, a `select` statement must have an `into` clause. The clause specifies a list of host variables to be assigned values.

**Note** The current Embedded SQL precompiler version does not support `into` clauses that specify tables.

The syntax of the Embedded SQL `select` statement is:

```sql
exec sql [at connect_name]
    select [all | distinct] select_list into 
        :host_variable[[indicator]:indicator_variable] 
        [, :host_variable 
            [[indicator]:indicator_variable]...] 
    end-exec
```

For additional information on `select` statement clauses, see the *Adaptive Server Enterprise Reference Manual*.

The following `select` statement example accesses the `authors` table in the `pubs2` database and assigns the value of `au_id` to the host variable `ID`:

```sql
exec sql select au_id into :ID from authors 
    where au_lname = "Stringer" 
end-exec
```

Selecting multiple rows through arrays

You can return multiple rows with arrays. The two array actions involve selecting and fetching into arrays.

**select into arrays**

Use the `select into array` method when you know the maximum number of rows that will be returned. If a `select into` statement attempts to return more rows than the array can hold, the statement returns the maximum number of rows that the smallest array can hold.

Following is an example of selecting into an array:
**Selecting rows**

```sql
exec sql begin declare section end-exec
01 TITLEID-ARRAY PIC X(6) OCCURS 100 TIMES.
exec sql end declare section end-exec
...
exec sql select title_id into :titleid-array
from titles end-exec.
```

**Indicator arrays**

To use indicators with array fetches, declare an array of indicators of the same length as the host variable array, and use the syntax for associating the indicator with the host variable.

**Example**

```sql
exec sql begin declare section end-exec
01 ITEM-NUMBERS S9(9) OCCURS 100 TIMES.
01 I-ITEM-NUMBERS S9(4) OCCURS 100 TIMES.
exec sql end declare section end-exec
...
exec sql select it_n from item.info
into :item-numbers :i-item-numbers end-exec.
...
```

**Arrays and structures as indicator variables**

For tables with a large number of columns, you can use arrays and structures as a set of host variables that is referenced in a SQL statement. For this feature to work correctly, you must declare the indicator array or indicator structure elements with a PIC S9(4) clause and a COMP-5 clause. As with ESQL/C, use of structures and arrays as indicator variables removes the time consuming process of coding singleton indicator variables in ESQL/COBOL for every nullable column of every Embedded SQL statement in the application.

**Examples**

**Example 1** This is an example of declaring indicator arrays and executing a query on the indicator arrays:

```cobol
* Declare variables
....
01 HOST-STRUCTURE-M1.
   03 M-TITLE PIC X(64).
   03 M-NOTES PIC X(200).
   03 M-PUBNAME PIC X(40).
   03 M-PUBCITY PIC X(20).
   03 M-PUBSTATE PIC X(2).

01 INDICATOR-TABLE.
   03 I-NOTES-ARR PIC S9(4) COMP-5 OCCURS 5 TIMES.
```
* Execute query

EXEC SQL
SELECT substring(title, 1, 64), notes, pub_name, city, state
    INTO :HOST-STRUCTURE-M1:I-NOTES-ARR
FROM titles, publishers
WHERE titles.pub_id = publishers.pub_id
AND title_id = :USER-TITLEID
END-EXEC.

Example 2 This is an example declaring indicator structures and executing a query on the indicator structures:

* Declare variables

01 HOST-STRUCTURE-M1.
   03 M-TITLE PIC X(64).
   03 M-NOTES PIC X(200).
   03 M-PUBNAME PIC X(40).
   03 M-PUBCITY PIC X(20).
   03 M-PUBSTATE PIC X(2).

01 INDICATOR-STRUCTURE-I1.
   03 I-TITLE PIC S9(4) COMP-5.
   03 I-NOTES PIC S9(4) COMP-5.
   03 I-PUBNAME PIC S9(4) COMP-5.
   03 I-PUBCITY PIC S9(4) COMP-5.
   03 I-PUBSTATE PIC S9(4) COMP-5.

* Execute query

EXEC SQL
SELECT substring(title, 1, 64), notes, pub_name, city, state
    INTO :HOST-STRUCTURE-M1:INDICATOR-STRUCTURE-I1
FROM titles, publishers
WHERE titles.pub_id = publishers.pub_id
AND title_id = :USER-TITLEID
END-EXEC.

Usage When using structs and arrays as indicator variables:
Selecting rows

- The number of elements in the indicator array or struct must be exactly the same as the number of elements in the host variable structure. A mismatch causes cobpre or cobpre64 to stop processing, and code is not generated.
- The columns in the SELECT list must match by sequence, and datatype, the chosen structure name in the INTO list. A mismatch causes ct_bind() runtime errors and stops processing.

Error messages

Table 6-1 describes the Embedded SQL internal error messages created to handle host variable versus indicator variable mismatch errors for this feature.

Table 6-1: New internal error messages

<table>
<thead>
<tr>
<th>Message ID</th>
<th>Message text</th>
<th>Severity</th>
<th>Fix</th>
</tr>
</thead>
<tbody>
<tr>
<td>M_INVTYPE_V</td>
<td>Incorrect type of indicator variable found in the structure.</td>
<td>Fatal</td>
<td>Make sure that the same indicator variable is used in the hostvar and indicator declarations.</td>
</tr>
<tr>
<td>M_INVTYPE_VI</td>
<td>Mismatch between number of structure elements in the indicator structure and hostvar structure.</td>
<td>Fatal</td>
<td>Declare the same number of elements in the indicator structure and hostvar structure.</td>
</tr>
<tr>
<td>M_INVTYPE_VII</td>
<td>Mismatch between number of elements in the indicator array and hostvar structure.</td>
<td>Fatal</td>
<td>Declare the same number of elements in the indicator array and hostvar structure.</td>
</tr>
</tbody>
</table>

Limitation

You cannot mix singleton host variables or singleton indicator variables with hostvar structures, and indicator arrays or structures.

fetch into: batch arrays

fetch returns the specified number of rows from the currently active set. Each fetch returns the subsequent batch of rows. For example, if the currently active set has 150 rows and you select and fetch 60 rows, the first fetch returns the first 60 rows. The next fetch returns the following 60 rows. The third fetch returns the last 30 rows.

Note To find the total number of rows fetched, see the SQLERRD variable in the SQLCA, as described in “SQLCA variables” on page 18.

Following is an example of selecting into an array:

```sql
exec sql begin declare section end-exec
    TITLEID-ARRAY PIC X(6) occurs 100 times.
exec sql end declare section end-exec
... 
exec sql
```
select title_id into :titleid_array
    from titles
end-exec
IF (SQLERRD OF SQLCA LESS THAN 50)
    DISPLAY "No of title_ids is less than 50";
ENDIF.

Cursors and arrays

Use the fetch into array method when you do not know the number of rows to be returned into the array. Declare and open a cursor, then use fetch to retrieve groups of rows. If a fetch into attempts to return more rows than the array can hold, the statement returns the maximum number of rows that the smallest array can hold and SQLCODE displays a negative value, indicating that an error or exception occurred.

Using cursors

A cursor is a data selector that passes multiple rows of data to the host program, one row at a time. The cursor indicates the first row, also called the current row, of data and passes it to the host program. With the next fetch statement, the cursor advances to the next row, which has now become the current row. This continues until all requested rows are passed to the host program.

Use a cursor when a select statement returns more than one row of data. Client-Library tracks the rows Adaptive Server Enterprise returns and buffers them for the application. To retrieve data with a cursor, use the fetch statement.

The cursor mechanism is composed of these statements:

- declare
- open
- fetch
- update and delete where current of
- close

Cursor scoping rules

The scope of a cursor declaration is the file in which it is declared. The open statement(s) for a cursor must reside in the same file in which the cursor is declared. Once a cursor is open, its scope is the connection on which it was opened.
Selecting rows

The same cursor name can be opened for multiple connections. Cursor fetch, update, delete, and close operations can occur in files other than the one in which the cursor was declared, as long as they are executed on the same connection on which the cursor was opened.

Cursor names must be unique within a program. If, at runtime, an application attempts to declare two identically named cursors, the application fails with the following error message:

There is already another cursor with the name ‘XXX’.

Declaring cursors

The declare cursor statement is a declaration, not an executable statement. Therefore, it may appear anywhere in a file; SQLCODE, SQLSTATE, and SQLCA are not set after this statement.

Declare a cursor for each select statement that returns multiple rows of data. You must declare the cursor before using it, and you cannot declare it within a declare section.

The basic syntax for declaring a cursor is:

exec sql declare cursor_name cursor
   for select_statement end-exec.

where:

- cursor_name identifies the cursor. The name must be unique and have a maximum of 255 characters. The name must begin with a letter of the alphabet or with the symbols # or _.
- select_statement is a select statement that can return multiple rows of data. The syntax for select is the same as described in the Adaptive Server Enterprise Reference Manual, except that you cannot use into or compute clauses.

Example: Declaring a cursor

The following example demonstrates declaring cursors:

exec sql declare C1 cursor for
   select type, price from titles
   where type like :WK-TYPE end-exec
In this example, C1 is declared as a cursor for the rows that will be returned for the *type* and *price* columns. The precompiler generates no code for the declare cursor statement. It simply stores the select statement associated with the cursor.

When the cursor opens, the select statement or procedure in the declare cursor statement executes. When the data is fetched, the results are copied to the host variables.

**Note** Each cursor’s open and declare statements must be in the same file. Host variables used within the declare statement must have the same scope as the one in which the open statement is defined. However, once the cursor is open, you can perform fetch and update/delete where current of on the cursor in any file.

---

**Declaring scrollable cursors**

The syntax for declaring a scrollable cursor is:

```
exec sql declare cursor_name [cursor sensitivity] [cursor scrollability] cursor 
for select_statement ;
```

where:

- *cursor_name* identifies the cursor. The name must be unique and have a maximum of 255 characters. The name must begin with a letter of the alphabet or with the symbols “#” or “_”.

- *cursor sensitivity* specifies the sensitivity of the cursor. The options are:
  - semi_sensitive. If semi_sensitive is specified in the declare statement, scrollability is implied. The cursor is semi_sensitive, scrollable, and read-only.
  - insensitive. If insensitive is specified in the declare statement, the cursor is insensitive. Scrollability is determined by specifying SCROLL in the declare part. If SCROLL is omitted or NOSCROLL is specified, the cursor is insensitive only and non-scrollable. It is also read-only.

If cursor sensitivity is not specified, the cursor is non-scrollable and read-only.

- *cursor scrollability* specifies the scrollability of the cursor. The options are:
Selecting rows

- **scroll.** If scroll is specified in the declare statement and sensitivity is not specified, the cursor is insensitive and scrollable. It is also read-only.
- **no scroll.** If the SCROLL option is omitted or NOSCROLL is specified, the cursor is non-scrollable and read-only. See the previous cursor sensitivity description for cursor behavior.

If cursor scrollability is not specified, the cursor is non-scrollable and read-only.

- **select_statement** is a select statement that can return multiple rows of data. The syntax for select is the same as described in the *Adaptive Server Enterprise Reference Manual*, except that you cannot use into or compute clauses.

Declaring cursors that release locks at cursor close

The syntax for declaring cursors that release locks when the cursor closes is:

```
exec sql declare  cursor_name  [cursor sensitivity]  
[cursor scrollability]  [release_locks_on_close]
cursor for  select_statement  
 [for {read only | update { of column_name_list}}]  ;
```

where:

- **cursor_name** identifies the cursor. The name must be unique and have a maximum of 255 characters. The name must begin with a letter of the alphabet or with the symbols “#” or “_”.
- **cursor sensitivity** – See “Declaring scrollable cursors” on page 53.
- **cursor scrollability** – See “Declaring scrollable cursors” on page 53.
- **select_statement** is a select statement that can return multiple rows of data. The syntax for select is the same as described in the *Adaptive Server Enterprise Reference Manual*, except that you cannot use into or compute clauses.
- **column_name_list** identifies the list of columns to be affected.

You cannot use release_locks_on_close with an update clause except in this form:

```
exec sql declare cursor c1 release_locks_on_close  
cursor for select * from T for update of col_a
```
In this case, release_locks_on_close is ignored.

---

**Note**

cobpre cannot generate these ct_cursor() options:

- `CS_CUR_RELLOCKS_ONCLOSE | CS_READ_ONLY`
- `CS_CUR_RELLOCKS_ONCLOSE | CS_FOR_UPDATE`

ESQL/COBOL sample code is available in `example7.pco`.

---

### Opening cursors

To retrieve the contents of selected rows, you must first open the cursor. The `open` statement executes the `select` statement associated with the cursor in the `declare` statement.

The `open` statement’s syntax for opening a cursor is:

```sql
exec sql open cursor_name [ROW_COUNT = size] end-exec.
```

**Note**

`ROW_COUNT` should be specified with cursors when arrays are used as host variables and multi-row retrieval is required.

After you declare a cursor, you can open it wherever you can issue a `select` statement. When the `open` statement executes, Embedded SQL substitutes the values of any host variables referenced in the `declare cursor` statement’s `where` clause.

The number of cursors you may have open depends on the resource demands of the current session. Adaptive Server Enterprise does not limit the number of open cursors. However, you cannot open a currently open cursor. Doing so results in an error message.

While an application executes, you can open a cursor as many times as necessary, but you must close it before reopening it. You need not retrieve all the rows from a cursor result set before retrieving rows from another cursor result set.

---

### Fetching data using cursors

Use a `fetch` statement to retrieve data through a cursor and assign it to host variables. The syntax for the `fetch` statement is:

```sql
exec sql [at connect_name] fetch cursor_name
```
Selecting rows

```
into : host_variable
[[ indicator]: indicator_variable ]
[, : host_variable
[[ indicator]: indicator_variable ]...];
```

where there is one `host_variable` for each column in the result rows.

Prefix each `host_variable` with a colon and separate it from the next `host_variable` with a comma. The `host_variable` listed in the `fetch` statement must correspond to Adaptive Server Enterprise values that the `select` statement retrieves. Thus, the number of variables must match the number of returned values, they must be in the same order, and they must have compatible datatypes.

An `indicator_variable` is a 2-byte signed integer declared in a previous `declare` section. If a value retrieved from Adaptive Server Enterprise is null, the runtime system sets the corresponding `indicator_variable` to -1. Otherwise, the indicator is set to 0.

The data that the `fetch` statement retrieves depends on the cursor position. The cursor points to the `current row`. The `fetch` statement always returns the current row. The first `fetch` retrieves the first row and copies the values into the `host_variable` indicated. Each `fetch` advances the cursor to the next result row.

Normally, you should place the `fetch` statement within a loop so all values returned by the `select` statement can be assigned to `host_variable`. Following is a loop that is commonly used:

```
EXEC SQL
whenever sqlextend perform err-para thru err-para-end
end-exec.
EXEC SQL
whenever not found go to read-end
end-exec.

* 0 is never equal to 1, so the perform will run
* until the whenever NOT FOUND clause causes
* a jump to READ-END

    PERFORM READ-PARA UNTIL 0 = 1.

READ-END.
... .

READ-PARA.
    EXEC SQL fetch cursor_name into host-variable-list
end-exec.
```
This loop continues until all rows are returned or an error occurs. In either case, SQLCODE or SQLSTATE, which the whenever statement checks after each fetch, indicates the reason for exiting the loop. The error-handling routines ensure that an action is performed when either condition arises, as described in Chapter 8, “Handling Errors”.

**Fetching data using scrollable cursors**

Use a fetch statement to retrieve data through a cursor and assign it to host variables. The syntax for the fetch statement is:

```sql
exec sql [at connect_name] fetch [fetch orientation] cursor_name
    into : host_variable
    [[ indicator]: indicator_variable ]
    [, : host_variable
    [[ indicator]: indicator_variable ]...];
```

where one `host_variable` exists for each column in the result rows.

Prefix each host variable with a colon, and separate it from the next host variable with a comma. The host variables listed in the fetch statement must correspond to Adaptive Server Enterprise values that the select statement retrieves. Thus, the number of variables must match the number of returned values, they must be in the same order, and they must have compatible datatypes.

The `fetch orientation` specifies the fetch direction of the row to be fetched, if a cursor is scrollable. The options are: NEXT, PRIOR, FIRST, LAST, ABSOLUTE `fetch_offset` and RELATIVE `fetch_offset`. If fetch orientation is not specified, next is default. If fetch orientation is specified, the cursor must be scrollable.

The data that the fetch statement retrieves depends on the cursor position. The fetch statement typically retrieves single or multiple rows from the cursor result set, depending on the ROW_COUNT specification at cursor open time. If a cursor is not scrollable, fetch retrieves the next row in the result set. If a cursor is scrollable, commands in the fetch statement specify the row position to be fetched.
Selecting rows

Example for declaring a scrollable cursor and fetching rows

To declare a scrollable cursor and fetch rows at random, specify the scroll sensitivity and scrollability in the declare cursor, then specify the fetch orientation at fetch time. The following example demonstrates declaring an insensitive scrollable cursor and fetching rows at random:

```sql
exec sql declare c1 insensitive scroll cursor for
    select title_id, royalty, ytd_sales from authors
    where royalty < 25;
exec sql open c1;
```

In this example, scroll and insensitive are specified in the declare cursor. A fetch orientation can be specified at fetch time to indicate which row is required from the result set.

Once a cursor has been declared as scrollable and opened, a FETCH orientation can be specified at fetch time to indicate which row is wanted from the result set.

The following fetch example fetches the specified columns of the first row from the result set:

```sql
exec sql fetch first from c1 into :title,:roy,:sale;
```

The following fetch example fetches the specified columns of the previous row from the result set:

```sql
exec sql fetch prior from c1 into :title,:roy,:sale;
```

The following fetch example fetches the specified columns of row twenty from the result set:

```sql
exec sql fetch absolute 20 from c1 into :title,:roy,:sale;
```

Use sqlcode or sqlstate to determine if fetch statements return valid rows. For scrollable cursors, it is possible to fetch 0 rows if the cursor is positioned outside of result set boundaries, for example, before the first row or after the last row. In these circumstances, fetching 0 rows is not an error.

Using cursors to update and delete rows

To update or delete the current row of a cursor, specify where current of cursor_name as the search condition in an update or delete statement.

To update rows through a cursor, the result columns to be used in the updates must be updatable. They cannot be the result of SQL expressions such as max(colname). In other words, there must be a valid correspondence between the result column and the database column to be updated.

The following example demonstrates how to use a cursor to update rows:
CHAPTER 6  Using Transact-SQL Statements

exec sql declare c1 cursor for
    select title_id, royalty, ytd_sales
    from titles
    where royalty < 12
end-exec

exec sql open C1 end-exec

PERFORM READ-PARA UNTIL SQLCODE = 100.
exec sql close C1 end-exec.
STOP RUN.

READ-PARA.
    exec sql fetch C1 into :TITLE-ID, :ROYALTY, :SALES end-exec.
    IF SALES > 10000
        exec sql update titles
            set royalty = :roy + 2
            where current of C1 end-exec.

The Embedded SQL syntax of the update and delete statements is the same as in Transact-SQL, with the addition of the where current of cursor_name search condition.

For details on determining table update protocol and locking, see the Adaptive Server Enterprise Transact-SQL Users Guide.

Closing cursors

Use the close statement to close an open cursor. The syntax for the close statement is:

    exec sql [at connection] close cursor_name end-exec

To reuse a closed cursor, issue another open statement. When you reopen a cursor, it points to the first row. Do not issue a close statement for a cursor that is not open or an error will result.

Cursor example

The following example shows how to nest two cursors. Cursor C2 depends upon the value fetched into TITLE-ID from cursor C1.

The program gets the value of TITLE-ID at open time, not at declare time.

    ...
    exec sql declare C1 cursor for
        select title_id, title, royalty from titles
Selecting rows

end-exec

exec sql declare C2 cursor for
    select au_lname, au_fname, from authors
    where au_id in
        (select au_id from titleauthor
            where title_id = :TITLE-ID)
end-exec

exec sql open C1 end-exec.

PERFORM READ-TITLE UNTIL SQLCODE = 100.

READ-END.
...

READ-TITLE.
    exec sql fetch C1 into
        :TITLE-ID, :TITLE, :ROYALTY end-exec.
    IF SQLCODE NOT = 100
        MOVE ROYALTY TO DISP-ROY
        DISPLAY "Title ID: " TITLE-ID
        " , Royalty: " DISP-ROY
        IF ROYALTY > 10
            exec sql open C2 end-exec
            PERFORM READ-AUTH UNTIL SQLCODE = 100
            exec sql close C2 end-exec.

READ-AUTH.
    exec sql fetch C2 into :AU-LNAME, :AU-FNAME
end-exec
    IF SQLCODE NOT = 100
        DISPLAY " AUTHOR: " AU-LNAME " "
        AU-FNAME.

See the online sample programs for more examples using cursors. For details on accessing the online examples, see the Open Client and Open Server Programmers Supplement.
Using stored procedures

There are two types of stored procedures: user-defined and precompiler-generated. Both types run faster than standalone statements because Adaptive Server Enterprise preoptimizes the queries. You create user-defined stored procedures, and the precompiler generates stored procedures.

User-defined stored procedures

With Embedded SQL, you can execute stored procedures with select statements that return data rows. Stored procedures can return results to your program through output parameters and through a return status variable.

Stored procedure parameters can be either input or both input and output. For details on stored procedures, see the Adaptive Server Enterprise Transact-SQL Users Guide.

Syntax

Valid stored procedure names consist of uppercase and lowercase letters and the characters $, _, and #.

Do not include the use statement in a stored procedure.

To execute a stored procedure, use the following syntax:

```sql
exec sql [at connection_name]
exec [[:status_variable = status_value] procedure_name
[[[[[@parameter_name =]parameter_value [out{put}]],...]]
[into :hostvar_1 [[:indicator_1]]
[,] hostvar_n [indicator_n, ...]]]
[with recompile]
end-exec
```

where:

- `status_variable` can return either an Adaptive Server Enterprise return status value or a return code, which either indicates that the stored procedure completed successfully or gives the reasons for the failure. Negative status values are reserved for Adaptive Server Enterprise use. See the Adaptive Server Enterprise Transact-SQL Users Guide for a list of return status values for stored procedures.

- `status_value` is the value of the stored procedure return status variable `status_variable`.

- `procedure_name` is the name of the stored procedure to execute.
Selecting rows

- `parameter_name` is the name of a variable in the stored procedure. You can pass parameters either by position or by name, using the `@parameter_name` format. If one parameter is named, all of them must be named. See the Transact SQL Users Guide.

- `parameter_value` is a literal constant or host variable whose value is passed to the stored procedure. If it is a host variable, you can associate an indicator with it. Note that this variable has no keyword associated with it.

- `output` indicates that the stored procedure returns a parameter value. The matching parameter in the stored procedure must also have been created using the output keyword.

- `into:hostvar_1` causes row data returned from the stored procedure to be stored in the specified host variables (`hostvar_1` through `hostvar_n`). Each host variable can have an indicator variable.

- `indicator_n` is a two-byte host variable declared in a previous declare section. If the value for the associated `hostvar_n` is null, the indicator variable is set to -1 when the row data is retrieved. If truncation occurs, the indicator variable is set to the actual length of the result column. Otherwise, the indicator variable is 0.

- `with recompile` causes Adaptive Server Enterprise to create a new query plan for this stored procedure each time the procedure executes.

**Note** In Embedded SQL, the `exec` keyword is required to execute a stored procedure. You cannot substitute `execute` for `exec`.

**Stored procedure example**

The following example shows a call to a stored procedure where `RET-CODE` is a status variable, `a_proc` is the stored procedure, `PAR-1` is an input parameter, and `PAR-2` is an output parameter:

```sql
exec sql begin declare section end-exec
  01  PAR-1              PIC S9(9) COMP.
  01  PAR-2              PIC S9(9) COMP.
  01  RET-CODE           PIC S9(4) COMP.
exec sql end declare section end-exec

exec sql exec :RET-CODE=a_proc :PAR-1, :PAR-2 out end-exec.
```

The next example demonstrates the use of a stored procedure that retrieves data rows. The name of the stored procedure is “get_publishers”:
exec sql begin declare section end-exec.
  01 PUB-ID            PIC X(4).
  01 NAME              PIC X(45).
  01 CITY              PIC X(25).
  01 STATE             PIC X(2).
  01 RET-CODE          PIC S9(9).
  exec sql end declare section end-exec.

exec sql exec :RET-CODE = get_publishers :PUB-ID
  into :NAME :CITY :STATE END-EXEC.

See Chapter 10, “Open Client/Server Configuration File” for a more detailed example of the exec statement.

Conventions

The datatypes of the stored procedure parameters must be compatible with the COBOL host variables. Client-Library only converts certain combinations. See Chapter 4, “Using Variables” for a table of compatible datatypes.

Precompiler-generated stored procedures

You can set an optional command line switch so that the precompiler automatically generates stored procedures that can optimize the execution of Transact-SQL statements in your program.

For the list of precompiler command line option switches, see the Open Client and Open Server Programmers Supplement.

Follow these steps to activate precompiler-generated stored procedures:

1 Set the appropriate command line switch so that the precompiler automatically generates stored procedures for the Transact-SQL statements to be optimized.
   The precompiler generates an sql file containing statements that generate the stored procedures.

2 Use interactive SQL (the isql program) to execute the file.
   This loads the stored procedures on Adaptive Server Enterprise. The precompiler also creates the stored procedure calls in its output file.

By default, precompiler-generated stored procedures have the same name as the source program, minus any file extensions. The stored procedures are numbered sequentially and the file name and number are separated by a semicolon (;).
For example, the stored procedures for a source program named test1.pco, would be named test1;1 through test1;n, where n is the number of the source program’s last stored procedure.

Optionally, you can set a command line flag that lets you alter the stored procedures’ names. By using this flag, you can test a modified application without deleting a stored procedure already in production. After successfully testing the application, you can precompile it without the flag to install the stored procedure.

**Note** When you issue the declare cursor statement, only the select clause is saved as a stored procedure. If an application has syntax errors, the precompiler generates neither the target file nor stored procedures.

---

**Grouping statements**

Statements can be grouped for execution by batch or by transactions.

**Grouping statements by batches**

A batch is a group of statements you submit as one unit for execution. The precompiler executes all Transact-SQL statements within the exec sql and end-exec keywords in batch mode.

Although the precompiler saves stored procedures, it does not save batches for re-execution. The batch is effective only for the current execution.

The precompiler supports only batch mode statements that return no result sets.

```
exec sql insert into TABLE1 values (:val1)
insert into TABLE2 values (:val2)
insert into TABLE3 values (:val3)
end-exec.
```

The three insert statements are processed as a group, which is more efficient than being processed individually. Use the get diagnostics method of error handling with batches. For details, see “Using get diagnostics” on page 100.

These statements are legal within a batch because none of them returns results. See the *Adaptive Server Enterprise Transact-SQL Users Guide*.  

---

Note When you issue the declare cursor statement, only the select clause is saved as a stored procedure. If an application has syntax errors, the precompiler generates neither the target file nor stored procedures.
Grouping statements by transactions

A transaction is a single unit of work, whether the unit consists of one or 100 statements. The statements in the transaction execute as a group, so either all or none of them execute.

The precompiler supports two transaction modes: default ANSI/ISO and optional Transact-SQL. In the Transact-SQL transaction mode, each statement is implicitly committed unless it is preceded by a begin transaction statement.

The Transact-SQL mode uses relatively few system resources, while the default ANSI/ISO transaction mode can dramatically affect system response time. For details on choosing the appropriate mode for your application, see the Adaptive Server Enterprise Transact-SQL Users Guide.

You can use a precompiler option to determine the transaction mode of the connections your application opens. See the Open Client and Open Server Programmers Supplement for details.

Transact-SQL transaction mode

In this optional Transaction mode, the Embedded SQL syntax is the same as that used in Transact-SQL. The begin transaction statement explicitly initiates transactions.

The syntax of the Embedded SQL transaction statements is:

```
exec sql [at connect_name]
  begin transaction [transaction_name] end-exec

exec sql [at connect_name]
  save transaction [savepoint_name] end-exec
exec sql [at connect_name] commit transaction [transaction_name] end-exec
exec sql [at connect_name] rollback transaction [savepoint_name | transaction_name] end-exec
```

**Note** disconnect rolls back all open transactions. For details on this statement, see Chapter 5, “Connecting to Adaptive Server Enterprise.”

When you issue a begin transaction on a connection, you must also issue a save, commit, or roll back transaction on the same connection. Otherwise, an error is generated.
Grouping statements

Default ANSI/ISO transaction mode

ANSI/ISO SQL does not provide a save transaction or begin transaction statement. Instead, transactions begin implicitly when the application program executes one of the following statements:

- delete
- insert
- select
- update
- open
- exec

The transaction ends explicitly when you issue either a commit work or rollback work statement. You must use the ANSI/ISO forms of the commit and rollback statements.

The syntax is:

```
exec sql commit [work] end-exec
exec sql rollback [work] end-exec
```

Extended transactions

An extended transaction is a unit of work that has multiple Embedded SQL statements. In the Transact-SQL transaction mode, you surround an extended transaction statement with the begin transaction and commit transaction statements.

In the default ANSI mode, you are constantly within an extended transaction. When you issue a commit work statement, the current extended transaction ends and another begins. For details, see the *Adaptive Server Enterprise Transact-SQL Users Guide*.

**Note** Unless the database option allow ddl in tran is set, do not use the following Transact-SQL statements in an extended, ANSI-mode transaction: alter database, create database, create index, create table, create view, disk init, grant, load database, load transaction, revoke, truncate table, and update statistics.
Including files and directories

The include statement is essentially the same as the COBOL COPY command, except that file search and copy occur at precompile time. At precompile time, include searches for the file in the directory or directories specified in the precompile statement. See the Open Client and Open Server Programmers Supplement for details about using the precompile statement and the COBOL compiler in your environment.

You can use the Embedded SQL include statement to add any source code file to your application, such as common data definitions, just as you use the COBOL COPY command. Hence, the following example is valid:

```sql
exec sql include "myfile" end-exec.
```

The precompiler changes include statements into COBOL COPY commands, surrounding the file name with quotation marks.

You can also set a precompiler command option to specify an include file directory. At precompile time, the precompiler searches the path specified in the COBOL compile command. When you specify a directory using this option, the precompiler adds the directory to the file name and encloses the entire path name in quotation marks. The file’s path is then hard-coded into the target program. See the Open Client and Open Server Programmers Supplement for details.
Including files and directories
CHAPTER 7

Using Dynamic SQL

This chapter explains dynamic SQL, an advanced methodology that lets your Embedded SQL application users enter SQL statements while the application is running. While static SQL will suffice for most of your needs, dynamic SQL provides the flexibility to build diverse SQL statements at runtime.

Dynamic SQL is a set of Embedded SQL statements that permit users of online applications to access the database interactively at runtime.

Use dynamic SQL when one or more of the following conditions is not known until runtime:

- SQL statement the user will execute
- Column, index, and table references
- Number of host variables or their datatypes

Dynamic SQL is part of ANSI and the ISO SQL2 standard. It is useful for running interactive applications.

When to use dynamic SQL

If the application accepts only a small set of SQL statements, you can embed them within the program. However, if the application accepts many types of SQL statements, you can benefit from constructing SQL statements, and then binding and executing them dynamically.
The following type of situation would benefit from using dynamic SQL: The application program searches a bookseller’s database of books for sale. A potential buyer can apply many criteria, including price, subject matter, type of binding, number of pages, publication date, language, and so on.

A customer might say, “I want a nonfiction book about business that costs between $10 and $20.” This request is readily expressed as a Transact-SQL statement:

```sql
select * from titles where
type = "business"
and price between $10 and $20
```

It is not possible to anticipate the combinations of criteria that all buyers will apply to their book searches. Therefore, without using dynamic SQL, an Embedded SQL program can not easily generate a list of prospective books with a single query.

With dynamic SQL, the bookseller can enter a query with a different where clause search condition for each buyer. The seller can vary requests based on the publication date, book category, and other data, and can vary the columns to be displayed. For example:

```sql
select * from titles
where type = ?
and price between ? and ?
```

The question marks (“?”) are dynamic parameter markers that represent places where the user can enter search values.

**Note** The precompiler does not generate stored procedures for dynamic SQL statements because the statements are not complete until runtime. At runtime, Adaptive Server Enterprise stores them as temporary stored procedures in the tempdb database. The tempdb database must contain the user name "guest", which in turn must have create procedure permission. Otherwise, attempting to execute one of these temporary stored procedures generates the error message "Server user id user_id is not a valid user in database database_name", where user_id is the user’s user ID, and database_name is the name of the user’s database.
The dynamic SQL prepare statement sends the actual SQL statement, which can be any Data Definition Language (DDL) or Data Manipulation Language (DML) statements or any Transact-SQL statement, except create procedure, to the server.

The dynamic SQL facility performs these actions:

1. Translates the input data into a SQL statement.
2. Verifies that the SQL statement can execute dynamically.
3. Prepares the SQL statement for execution, sending it to Adaptive Server Enterprise, which compiles and saves it as a temporary stored procedure (for methods 2, 3, and 4).
4. Binds all input parameters or descriptor (for methods 2, 3, and 4).
5. Executes the statement.
   For a varying-list select, it uses a descriptor to reference the data items and rows returned (for method 2 or 4).
6. Binds the output parameters or descriptor (for method 2, 3, or 4).
7. Obtains results (for method 2, 3, or 4).
8. Drops the statement (for methods 2, 3, and 4) by reactivating the stored procedure in Adaptive Server Enterprise.
9. Handles all error and warning conditions from Adaptive Server Enterprise and Client-Library.

**Method 1: Using execute immediate**

Use execute immediate to send a complete Transact-SQL statement, stored in a host variable or literal string, to Adaptive Server Enterprise. The statement cannot return any results—you cannot use this method to execute a select statement.

The dynamically entered statement executes as many times as the user invokes it during a session. With this method:

1. The Embedded SQL program passes the text to Adaptive Server Enterprise.
2. Adaptive Server Enterprise verifies that the statement can execute dynamically and does not return rows.
Method 1: Using execute immediate

3 Adaptive Server Enterprise compiles and executes the statement.

With execute immediate, you can let the user enter all or part of a Transact-SQL statement.

The syntax for execute immediate is:

\[
\text{exec sql} \ [\text{at \ connection\_name}] \ \text{execute immediate} \\
\{:\text{host\_variable} \ | \ "\text{string}" \}\end-exec
\]

where:

- *host_variable* is a character-string variable defined in a declare section. Before calling execute immediate, the host variable should contain a complete and syntactically correct Transact-SQL statement.

- *string* is a literal Transact-SQL statement string that can be used in place of *host_variable*.

Embedded SQL sends the statement in *host_variable* or string to Adaptive Server Enterprise without any processing or checking. If the statement attempts to return results or fails, an error occurs. You can test the value of SQLCODE after executing the statement or use the whenever statement to set up an error handler. See Chapter 8, “Handling Errors” for information about handling errors in Embedded SQL programs.

Method 1 examples

The following two examples demonstrate using method 1, execute immediate. The first example prompts the user to enter a statement and then executes it:

\[
\begin{align*}
\text{exec sql} & \ \text{begin declare section end-exec} \\
01 \ & \text{CMD-1} \ \text{PIC X(50).} \\
01 \ & \text{SRC-COND} \ \text{PIC X(50).} \\
01 \ & \text{SQLSTR1} \ \text{PIC X(200).} \\
\text{exec sql} & \ \text{end declare section end-exec} \\
\text{DISPLAY} & \ "\text{ENTER statement}". \\
\text{ACCEPT} & \ \text{SQLSTR1}. \\
\text{exec sql} & \ \text{execute immediate :SQLSTR1 end-exec.}
\end{align*}
\]

The next example prompts the user to enter a search condition to specify rows in the titles table to update. Then, it concatenates the search condition to an update statement and sends the complete statement to Adaptive Server Enterprise.

\[
\begin{align*}
\text{MOVE} & \ "\text{UPDATE titles SET price = price*1.10 WHERE }"
\end{align*}
\]
TO CMD-1.
DISPLAY "ENTER SEARCH CONDITION:".
ACCEPT SRC-COND.
STRING CMD-1 delimited by size SRC-COND DELIMITED BY SIZE INTO SQLSTR1.
exec sql execute immediate :SQLSTR1 end-exec.

Method 2: Using prepare and execute

Use method 2, prepare and execute, when one of the following cases is true:

- You are certain that no data will be retrieved and you want the statement to execute more than once.
- A select statement is to return a single row. With this method, you cannot associate a cursor with the select statement.

This process is also called a single-row select. If a user needs to retrieve multiple rows, use method 3 or 4.

This method uses prepare and execute to substitute data from COBOL variables into a Transact-SQL statement before sending the statement to Adaptive Server Enterprise. The Transact-SQL statement is stored in a character buffer with dynamic parameter markers to show where to substitute values from COBOL variables.

Because this statement is prepared, Adaptive Server Enterprise compiles and saves it as a temporary stored procedure. Then, the statement executes repeatedly, as needed, during the session.

The prepare statement associates the buffer with a statement name and prepares the statement for execution. The execute statement substitutes values from a list of COBOL variables into the buffer and sends the completed statement to Adaptive Server Enterprise. You can execute any Transact-SQL statement this way.

prepare

The syntax for the prepare statement is:

    exec sql [at connection_name] prepare
    statement_name from {::host_variable | "string"}
end-exec

where:

- **statement_name** is a name up to 255 characters long that identifies the statement. It is not a COBOL variable or a literal string. It is a symbolic name that the precompiler uses to associate an `execute` statement with a `prepare` statement.
- **host_variable** is a dynamic parameter marker.
  Precede the dynamic parameter marker with a colon in standard Embedded SQL statements.
- **string** is a literal string that can be used in place of `host_variable`.

**execute**

The syntax for the `execute` statement is:

```
exec sql [at connection_name] execute statement_name
  [into {host_var_list | sql descriptor
    descriptor_name | descriptor sqlda_name }]
  [using {host_var_list | sql descriptor
    descriptor_name | descriptor sqlda_name}]
end-exec
```

where:

- **statement_name** is the name assigned in the `prepare` statement. `into` is used for a single-row `select`.
- `into` is used for a single-row `select`.
- `using` specifies the COBOL variables or descriptors that are substituted for dynamic parameter markers in variables in the `host_var_list`. The variables, which you must define in a `declare` section, are substituted in the order listed. You need only this clause when the statement contains variables using dynamic parameter markers.
- **descriptor_name** represents the area of memory that holds a description of the dynamic SQL statement’s dynamic parameter markers.
- **host_var_list** is a list of host variables to substitute into the parameter markers ("?" in the query).
- **sqlda_name** is the name of the SQLDA.
Method 2 example

The following example demonstrates using prepare and execute in method 2.
In this example, the user is prompted to enter a where clause that determines
which rows in the titles table to update. For example, entering “1.1” increases
the price by 10 percent.

```cobol
01 CUST-TYPE   PIC X.
    88 BIG-CUSTOMER VALUE "B".
    88 OTHER-CUSTOMER VALUE "O".

exec sql begin declare section end-exec
01 MULTIPLIER   PIC S9(2) COMP.
01 CMD-1        PIC X(50).
01 SRC-COND     PIC X(50).
01 SQLSTR1      PIC X(200).
exec sql end declare section end-exec

MOVE "UPDATE titles SET
    " price = price + (price * ? / 100)
WHERE "
TO CMD-1.
DISPLAY "ENTER SEARCH CONDITION:".
ACCEPT SRC-COND.
STRING CMD-1 SRC-COND DELIMITED BY SIZE
    INTO SQLSTR1.

exec sql prepare statement1 from :SQLSTR1
end-exec.

IF BIG-CUSTOMER
    MOVE 10 TO MULTIPLIER
ELSE
    MOVE 25 TO MULTIPLIER.

exec sql execute statement1 using :MULTIPLIER
end-exec.
```
Method 3: Using prepare and fetch with a cursor

Method 3 uses the `prepare` statement with cursor statements to return results from a `select` statement. Use this method for fixed-list `select` statements that may return multiple rows. That is, use it when the application has determined in advance the number and type of `select` column list attributes to be returned. You must anticipate and define host variables to accommodate the results.

When you use method 3, include the `declare`, `open`, `fetch`, and `close cursor` statements to execute the statement. This method is required because the statement returns more than one row. There is an association between the prepared statement identifier and the specified cursor name. You can also include `update` and `delete where current of` cursor statements.

As with method 2, a Transact-SQL `select` statement is first stored in a character host variable or string. It can contain dynamic parameter markers to show where to substitute values from input variables. The statement is given a name to identify it in the `prepare`, `declare`, and `open` statements.

Method 3 requires five steps:

1. `prepare`
2. `declare`
3. `open`
4. `fetch` (and, optionally, `update` and `delete`)
5. `close`

These steps are described below.

**prepare**

The `prepare` statement is the same as that used with method 2. For details, see “prepare” on page 73.

**declare**

The `declare` statement is similar to the standard `declare` statement for cursors. In dynamic SQL, however, you declare the cursor for a prepared `statement_name` instead of for a `select` statement, and any input host variables are referenced in the `open` statement instead of in the `declare` statement.
A dynamic declare statement is an executable statement rather than a declaration. As such, it must be positioned in the code where executable statements are legal, and the application should check status codes (SQLCODE, SQLCA, or SQLSTATE) after executing the declaration.

The dynamic SQL syntax for the declare statement is:

```
exec sql [at connection_name] declare cursor_name
cursor for statement_name end-exec
```

where:

- `at connection_name` specifies the Adaptive Server Enterprise connection the cursor will use.
- `cursor_name` identifies the cursor, used with the open, fetch, and close statements.
- `statement_name` is the name specified in the prepare statement, and represents the select statement to be executed.

**open**

The open statement substitutes any input variables in the statement buffer, and sends the result to Adaptive Server Enterprise for execution. The syntax for the open statement is:

```
exec sql [at connection_name] open cursor_name
[using {host_variable_list | sql descriptor descriptor_name | descriptor sqlda_name}]
end-exec
```

where:

- `cursor_name` is the name given to the cursor in the declare statement.
- `host_variable_list` consists of the names of the host variables that contain the value for a dynamic parameter marker.
- `descriptor_name` is the name of the descriptor that contains the value for the dynamic parameter markers.
- `sqlda_name` is the name of the SQLDA.
Method 3: Using prepare and fetch with a cursor

fetch and close

After a cursor opens, the result sets are returned to the application. Then, the data is fetched and loaded into the application program host variables. Optionally, you can update or delete the data. The fetch and close statements are the same as in static Embedded SQL.

The syntax for the fetch statement is:

```
exec sql [at connection_name] fetch cursor_name
   into :host_variable
   [,host_variable
   [[:indicator]:indicator_variable]]
end-exec
```

where:

- `cursor_name` is the name given to the cursor in the declare statement.
- There is one COBOL `host_variable` for each column in the result rows. The variables must have been defined in a declare section, and their datatypes must be compatible with the results returned by the cursor.

The syntax for the close statement is:

```
exec sql [at connection_name] close cursor_name
end-exec
```

where `cursor_name` is the name assigned to the cursor in the declare statement.

Method 3 example

The following example uses prepare and fetch, and prompts the user for an order by clause in a select statement:

```
exec sql begin declare section end-exec
  01 AGE                PIC S9(2) COMP.
  01 R-AGE              PIC S9(2).
  01 ROYALTY            PIC S9(9) COMP.
  01 TITLE              PIC X(25).
  01 MANAGER            PIC X(25).
  01 SQLSTR2            PIC X(100).
  01 I-TITLE            PIC S9(4) COMP.
  01 I-AGE              PIC S9(4) COMP.
  exec sql end declare section end-exec
  01 DSP-AGE            PIC 9(2).
```
01 DSP-ROYALTY PIC ZZZ,ZZZ,ZZZ.

PROCEDURE DIVISION.

MOVE 60 TO R-AGE.
MOVE "select age, royalty, title, manager from -        " inprogr where age !=?" TO SQLSTR2
MOVE 0 TO I-AGE.
exec sql prepare statement2 from :SQLSTR2 end-exec.
exec sql declare C1 cursor for statement2 end-exec
exec sql whenever not found goto NOT-FOUND end-exec
exec sql open C1 using :R-AGE indicator :I-AGE end-exec.

RET-LOOP.
MOVE 0 TO I-TITLE.
exec sql fetch C1 into
      :AGE, :ROYALTY,
      :TITLE indicator :I-TITLE,
      :MANAGER end-exec.
MOVE AGE TO DSP-AGE.
MOVE ROYALTY TO DSP-ROYALTY.
IF I-TITLE = -1
  MOVE "Null" TO TITLE.
DISPLAY "Age = " DSP-AGE
      " Royalty = " DSP-ROYALTY
      " Title = " TITLE
      " Manager = " MANAGER.
DISPLAY ".
GO TO RET-LOOP.

NOT-FOUND.
exec sql close C1 end-exec.
Method 4: Using prepare and fetch with system descriptors

This method permits varying-list select statements. That is, when you write the application, you need not know the formats and number of items the select statement will return.

Use this method when you cannot define the host variables in advance because you do not know how many variables are needed or of what type they should be.

Method 4 dynamic descriptors

A dynamic descriptor is a data structure that holds a description of the variables used in a dynamic SQL statement. There are two kinds of dynamic descriptors—SQL descriptors and SQLDA structures. Both are described later in this chapter.

When a cursor opens, it can have an input descriptor associated with it. The input descriptor contains the values to be substituted for the dynamic SQL statement’s parameter markers.

Before the cursor is opened, the user fills in the input descriptor with the appropriate information, including the number of parameters, and, for each parameter, its type, length, precision, scale, indicator, and data.

Associated with the fetch statement is an output descriptor, which holds the resultant data. Adaptive Server Enterprise fills in the data item’s attributes, including its type and the actual data being returned. If you are using an SQL descriptor, use the get descriptor statement to copy the data into host variables.

Dynamic SQL method 4 performs the following:

1. Prepares the statement for execution.
2. Associates a cursor with the statement.
3. Defines and binds the input parameters or descriptor and:
   - If using an input descriptor, allocates it
   - If using an input host variable, associates it with the statement or cursor
4. Opens the cursor with the appropriate input parameter(s) or descriptor.
5 Allocates the output descriptor if different from the input descriptor and binds the output descriptor to the statement.

6 Retrieves the data by using fetch cursor and the output descriptor.

7 Copies data from the dynamic descriptor into host program variables. If you are using an SQLDA, this step does not apply; the data is copied in step 6.

8 Closes the cursor.

9 Deallocates the dynamic descriptors.

10 Drops the statement (ultimately, the stored procedure).

**Dynamic descriptor statements**

There are statements that associate the descriptor with a SQL statement and with a cursor associated with the SQL statement. The following list briefly describes dynamic SQL statements for method 4:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>allocate descriptor</td>
<td>Notifies Client-Library to allocate a SQL descriptor.</td>
</tr>
<tr>
<td>describe input</td>
<td>Obtains information about the dynamic parameter marker in the prepare statement.</td>
</tr>
<tr>
<td>set descriptor</td>
<td>Inserts or updates data in the system descriptor.</td>
</tr>
<tr>
<td>get descriptor</td>
<td>Moves row or parameter information stored in a descriptor into host variables, thereby allowing the application program to use the information.</td>
</tr>
<tr>
<td>execute</td>
<td>Executes a prepared statement.</td>
</tr>
<tr>
<td>open cursor</td>
<td>Associates a descriptor with a cursor and opens the cursor.</td>
</tr>
<tr>
<td>describe output</td>
<td>Obtains information about the select list columns in the prepared dynamic SQL statement.</td>
</tr>
<tr>
<td>fetch cursor</td>
<td>Retrieves a row of data for a dynamically declared cursor.</td>
</tr>
<tr>
<td>deallocate descriptor</td>
<td>Deallocates a dynamic descriptor.</td>
</tr>
</tbody>
</table>

For complete descriptions of these statements, see Chapter 9, “Embedded SQL Statements: Reference Pages.”
Method 4: Using prepare and fetch with system descriptors

About SQL descriptors

A SQL descriptor is an area of memory that stores a description of the variables used in a prepared dynamic SQL statement. A SQL descriptor can contain the following information about data attributes.

- **precision** – integer.
- **scale** – integer.
- **nullable** – 1 (*cs_true*) if the column can contain nulls; 0 (*cs_false*) if it cannot. Valid only with `get descriptor` statement.
- **indicator** – value for the indicator parameter associated with the dynamic parameter marker.
- **name** – name of the dynamic parameter marker. Valid only with `get descriptor` statement.
- **data** – value for the dynamic parameter marker specified by the item number. If the value of `indicator` is -1, the value of `data` is undefined.
- **count** – number of dynamic parameter markers described in the descriptor.
- **type** – datatype of the dynamic parameter marker or host variable.
- **returned_length** – actual length of the data in an output column.

See the descriptions of the `set descriptor` and `get descriptor` commands in Chapter 9, “Embedded SQL Statements: Reference Pages.”

Method 4 example

The following example uses `prepare` and `fetch` with dynamic parameter markers and SQL descriptors.

```sql
exec sql begin declare section end-exec.
  01 COLTYPE IS GLOBAL PIC S9(9) COMP.
  01 INDEX-COLCNT IS GLOBAL PIC S9(9) COMP.
  01 INT-BUFF IS GLOBAL PIC S9(9) COMP.
  01 CHAR-BUFF IS GLOBAL PIC X(255).
  01 MISC-BUFF IS GLOBAL PIC X(255).
  01 TYPE IS GLOBAL PIC X(255).
  01 TITLE IS GLOBAL PIC X(255).
  01 COLNAME IS GLOBAL PIC X(255).
  01 SALES IS GLOBAL PIC S9(9) COMP.
  01 DESCNT IS GLOBAL PIC S9(9) COMP.
  01 OCCUR IS GLOBAL PIC S9(9) COMP.
```
CHAPTER 7 Using Dynamic SQL

01 CNT    IS GLOBAL PIC S9(9) COMP.
01 CONDCNT IS GLOBAL PIC S9(9) COMP.
01 DIAG-CNT IS GLOBAL PIC S9(9) COMP.
01 NUM-MSG IS GLOBAL PIC S9(9) COMP.
01 USER-ID IS GLOBAL PIC X(30).
01 PASS    IS GLOBAL PIC X(30).
01 SERVER-NAME IS GLOBAL PIC X(30).
01 STR1    IS GLOBAL PIC X(1024).
01 STR2    IS GLOBAL PIC X(1024).
01 STR3    IS GLOBAL PIC X(1024).
01 STR4    IS GLOBAL PIC X(1024).

exec sql end declare section end-exec.
...
PROCEDURE DIVISION.
P0.
DISPLAY "Dynamic sql Method 4".
DISPLAY "Enter in a Select statement to retrieve any kind "
DISPLAY "of information from the pubs database:"
accept str4.
DISPLAY "Enter in the larger of the columns to be "
DISPLAY "retrieved or the number "
DISPLAY "of ? in the SQL statement:"
ACCEPT occur.
exec sql prepare S4 from :str4 end-exec
exec sql declare c2 cursor for s4 end-exec
exec sql describe input s4 using sql descriptor dinout end-exec
    call "filldesc".
exec sql open c2 using sql descriptor dinout end-exec
PERFORM UNTIL SQLCODE = 100 OR SQLCODE < 0
exec sql fetch c2 into sql descriptor dinout end-exec
PERFORM "prtdesc".
END-PERFORM.
exec sql close c2 end-exec
exec sql deallocate descriptor dinout end-exec
exec sql deallocate prepare s4 end-exec
DISPLAY "Dynamic SQL Method 4 completed".
goback.
END PROGRAM dyn-m4.
IDENTIFICATION DIVISION.
PROGRAM-ID. prtdesc is common.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. xyz.
Method 4: Using prepare and fetch with system descriptors

OBJECT-COMPUTER. xyz.
DATA DIVISION.
WORKING-STORAGE SECTION.
PROCEDURE DIVISION.
P0.
exec sql get descriptor dinout :descnt = count
end-exec
DISPLAY "Column name Column data".
DISPLAY "----------- --------------------------"
DISPLAY "---------------".
PERFORM VARYING CNT FROM 1 BY 1 UNTIL cnt > descnt
* get each column attribute
  exec sql get descriptor dinout
    VALUE :index-colcnt :coltype = TYPE end-exec
  IF coltype = 1
    * character type
    exec sql get descriptor dinout VALUE :index-colcnt
    :colname = NAME, :char-buff = data end-exec
    DISPLAY colname char-buff.
  ELSE IF coltype = 4
    * integer type
    exec sql get descriptor dinout
      VALUE :index-colcnt :colname = NAME, :int-buff = DATA
    end-exec
    DISPLAY colname int-buff.
  ELSE
    * other types
    exec sql get descriptor dinout
      VALUE :index-colcnt
      :colname = NAME, :misc-buff = DATA end-exec
    DISPLAY colname misc-buff
    end-perform.
  goback.
END PROGRAM prtdesc.
...
PROCEDURE DIVISION.
P0.
exec sql get descriptor dinout :descnt = count
end-exec
  PERFORM varying cnt from 1 by 1 UNTIL cnt > descnt
  DISPLAY "Enter in the data type of the " cnt " ?".
  accept &coltype.
  IF coltype = 1
    * character type
About SQLDAs

SQLDA is a host-language structure that, like an SQL descriptor, describes the variables used in a dynamic SQL prepared statement. Unlike SQL descriptors, SQLDAs are public data structures whose fields you can access. Statements using SQLDAs may execute faster than equivalent statements using SQL descriptors.

The SQLDA structure is not part of the SQL standard. Different implementations of Embedded SQL define the SQLDA structure differently. Embedded SQL version 11.1 and later supports the SQLDA defined by Sybase; it does not support SQLDA datatypes defined by other vendors.

Embedded SQL does not limit the number of SQLDA structures that can be created by a program.

Table 7-1 describes the fields of the SQLDA structure.
Method 4: Using prepare and fetch with system descriptors

<table>
<thead>
<tr>
<th>Field</th>
<th>Datatype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SD-SQLN</td>
<td>PIC S9(9) COMP</td>
<td>The size of the sd_column array.</td>
</tr>
<tr>
<td>SD-SQLD</td>
<td>PIC S9(9) COMP</td>
<td>The number of columns in the query being described, or 0 if the statement being described is not a query. For fetch, open, and execute statements, this field indicates the number of host variables described by occurrences of sd_column or the number of dynamic parameter markers for the describe input statement.</td>
</tr>
<tr>
<td>SD-DATAFMT OF SD-COLUMN</td>
<td>Data format structure</td>
<td>The Client-Library CS_DATAFMT structure associated with this column. Refer to descriptions of ct_bind, ct_param and ct_describe in the Open Client Client-Library/C Reference Manual.</td>
</tr>
<tr>
<td>SD-SQLDATA OF SD-COLUMN</td>
<td>PIC S9(9) COMP or PIC S9(18) COMP</td>
<td>For fetch, open, and execute statements, stores the address of the statement’s host variable. This field is not used for describe or prepare statements.</td>
</tr>
<tr>
<td>SD-SQLIND OF SD-COLUMN</td>
<td>PIC S9(4) COMP</td>
<td>For fetch, open, and execute statements, this field acts as an indicator variable for the column being described. If the column’s value is null, this field is set to -1. This field is not used for describe or prepare statements. Set this field using SYBSETSQLDA (see “Using SYBSETSQLDA” on page 87).</td>
</tr>
<tr>
<td>SD-SQLLEN OF SD-COLUMN</td>
<td>PIC S9(9) COMP</td>
<td>The actual size of the Client Library CS_DATAFMT structure associated with this column.</td>
</tr>
</tbody>
</table>
Using SYBSETSQLDA

Since definitions of SQLDA fields do not correspond clearly to COBOL declarations, the SYBSETSQLDA function is provided so that you can use familiar COBOL terms. SYBSETSQLDA allows you to set the fields of a Sybase-style SQLDA. It sets the ITEM-NUMBER SQLDA-SQLDATA field of the given SQLDA to point to a given buffer, and sets datafmt fields appropriately.

Syntax

01 SQLDA-NAME.
   < rest of sqlda declaration >
01 ITEM-NUMBER PIC S9(9) COMP.
01 DATA-BUFFER < picture >.
01 PICTURE-TYPE PIC S9(9) COMP.
01 M PIC S9(9) COMP.
01 N PIC S9(9) COMP.
01 USAGE-TYPE PIC S9(9) COMP.
01 SIGN-TYPE PIC S9(9) COMP.
CALL "SYBSETSQLDA" USING SQLDA-NAME ITEM-NUMBER
   DATA-BUFFER PICTURE-TYPE M N USAGE-TYPE SIGN-TYPE

where:
• SQLDA-NAME is the SQLDA to set the information in.
• ITEM-NUMBER is the item to set the information for.
• DATA-BUFFER is the host variable with data.
• PICTURE-TYPE is the kind of picture clause the data has. See Table 7-2 for possible values.
• M is the value of “m” in the picture clause, as described in the table, or 0 if no picture.

<table>
<thead>
<tr>
<th>Field</th>
<th>Datatype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SD-SQLMORE OF SD-COLUMN</td>
<td>PIC S9(9)</td>
<td>Reserved.</td>
</tr>
<tr>
<td>or</td>
<td>COMP</td>
<td></td>
</tr>
<tr>
<td>or</td>
<td>PIC S9(18)</td>
<td></td>
</tr>
<tr>
<td>or</td>
<td>COMP</td>
<td></td>
</tr>
</tbody>
</table>
Method 4: Using prepare and fetch with system descriptors

- N is the value of “n” in the picture clause as described above, or 0 if no picture.
- SIGN-TYPE is the sign clause used to define the data. See Table 7-2 for possible values.
- USAGE-TYPE is the usage clause used to define the data. See Table 7-2 for possible values.
### Table 7-2: Values for SYBSETSQLDA

<table>
<thead>
<tr>
<th>Argument</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>USAGE-TYPE</td>
<td>SYB-BINARY-USAGE</td>
<td>USAGE IS BINARY</td>
</tr>
<tr>
<td>USAGE-TYPE</td>
<td>SYB-COMP-USAGE</td>
<td>USAGE IS COMP</td>
</tr>
<tr>
<td>USAGE-TYPE</td>
<td>SYB-COMP1-USAGE</td>
<td>USAGE IS COMP-1</td>
</tr>
<tr>
<td>USAGE-TYPE</td>
<td>SYB-COMP2-USAGE</td>
<td>USAGE IS COMP-2</td>
</tr>
<tr>
<td>USAGE-TYPE</td>
<td>SYB-COMP3-USAGE</td>
<td>USAGE IS COMP-3</td>
</tr>
<tr>
<td>USAGE-TYPE</td>
<td>SYB-COMP4-USAGE</td>
<td>USAGE IS COMP-4</td>
</tr>
<tr>
<td>USAGE-TYPE</td>
<td>SYB-COMP5-USAGE</td>
<td>USAGE IS COMP-5</td>
</tr>
<tr>
<td>USAGE-TYPE</td>
<td>SYB-COMP6-USAGE</td>
<td>USAGE IS COMP-6</td>
</tr>
<tr>
<td>USAGE-TYPE</td>
<td>SYB-COMPX-USAGE</td>
<td>USAGE IS COMP-X</td>
</tr>
<tr>
<td>USAGE-TYPE</td>
<td>SYB-DISPLAY-USAGE</td>
<td>USAGE IS DISPLAY</td>
</tr>
<tr>
<td>USAGE-TYPE</td>
<td>SYB-POINTER-USAGE</td>
<td>USAGE IS POINTER</td>
</tr>
<tr>
<td>USAGE-TYPE</td>
<td>SYB-INDEX-USAGE</td>
<td>USAGE IS INDEX</td>
</tr>
<tr>
<td>USAGE-TYPE</td>
<td>SYB-MONEY-USAGE</td>
<td>USAGE IS CS-MONEY</td>
</tr>
<tr>
<td>USAGE-TYPE</td>
<td>SYB-MONEY4-USAGE</td>
<td>USAGE IS CS-MONEY4</td>
</tr>
<tr>
<td>USAGE-TYPE</td>
<td>SYB-DATE-USAGE</td>
<td>USAGE IS CS-DATE</td>
</tr>
<tr>
<td>USAGE-TYPE</td>
<td>SYB-TIME-USAGE</td>
<td>USAGE IS CS-TIME</td>
</tr>
<tr>
<td>USAGE-TYPE</td>
<td>SYB-DATETIME-USAGE</td>
<td>USAGE IS CS-DATETIME</td>
</tr>
<tr>
<td>USAGE-TYPE</td>
<td>SYB-DATETIME4-USAGE</td>
<td>USAGE IS CS-DATETIME4</td>
</tr>
<tr>
<td>USAGE-TYPE</td>
<td>SYB-NO-USAGE</td>
<td>No usage clause</td>
</tr>
</tbody>
</table>
Method 4: Using prepare and fetch with system descriptors

<table>
<thead>
<tr>
<th>Argument</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>PICTURE-TYPE</td>
<td>SYB-NO-PIC</td>
<td>No picture clause</td>
</tr>
<tr>
<td>PICTURE-TYPE</td>
<td>SYB-SNINES-PIC</td>
<td>PIC S9(m)</td>
</tr>
<tr>
<td>PICTURE-TYPE</td>
<td>SYB-NINES-PIC</td>
<td>PIC 9(m)</td>
</tr>
<tr>
<td>PICTURE-TYPE</td>
<td>SYB-SVNINES-PIC</td>
<td>PIC S9(m)V9(n) or SV9(n)</td>
</tr>
<tr>
<td>PICTURE-TYPE</td>
<td>SYB-VNINES-PIC</td>
<td>PIC 9(m)V9(n) or V9(n)</td>
</tr>
<tr>
<td>PICTURE-TYPE</td>
<td>SYB-X-PIC</td>
<td>PIC X(m)</td>
</tr>
<tr>
<td>SIGN-TYPE</td>
<td>SYB-NO-SIGN</td>
<td>No sign clause (not an unsigned PIC clause)</td>
</tr>
<tr>
<td>SIGN-TYPE</td>
<td>SYB-LEADING-SEPARATE-SIGN</td>
<td>SIGN LEADING SEPARATE</td>
</tr>
<tr>
<td>SIGN-TYPE</td>
<td>SYB-TRAILING-SEPARATE-SIGN</td>
<td>SIGN TRAILING SEPARATE</td>
</tr>
<tr>
<td>SIGN-TYPE</td>
<td>SYB-LEADING-SIGN</td>
<td>SIGN LEADING</td>
</tr>
<tr>
<td>SIGN-TYPE</td>
<td>SYB-TRAILING-SIGN</td>
<td>SIGN TRAILING</td>
</tr>
</tbody>
</table>

Returns

No return value.

Method 4 example using SQLDAs

Following is an example that uses `prepare` and `fetch` with dynamic parameter markers and SQL descriptors.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. unittest.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. xyz.
OBJECT-COMPUTER. xyz.
DATA DIVISION.
WORKING-STORAGE SECTION.
exec sql begin declare section end-exec
  01 uid pic x(10).
  01 pass pic x(10).
```
exec sql end declare section end-exec
01 input-descriptor.
  09 SD-SQLN PIC S9(4) COMP.
  09 SD-SQLD PIC S9(4) COMP.
  09 SD-COLUMN OCCURS 3 TIMES.
  19 SD-DATAFMT.
    29 SQL--NM PIC X(132).
    29 SQL--NMLEN PIC S9(9) COMP.
    29 SQL--DATATYPE PIC S9(9) COMP.
    29 SQL--FORMAT PIC S9(9) COMP.
    29 SQL--MAXLENGTH PIC S9(9) COMP.
    29 SQL--SCALE PIC S9(9) COMP.
    29 SQL--PRECISION PIC S9(9) COMP.
    29 SQL--STATUS PIC S9(9) COMP.
    29 SQL--COUNT PIC S9(9) COMP.
    29 SQL--USERTYPE PIC S9(9) COMP.
    29 SQL--LOCALE PIC S9(9) COMP.
  19 SD-SQLDATA PIC S9(9) COMP.
  19 SD-SQLIND PIC S9(4) COMP.
  19 SD-SQLLEN PIC S9(9) COMP.
  19 SD-SQLMORE PIC S9(9) COMP.
01 output-descriptor.
  09 SD-SQLN PIC S9(4) COMP.
  09 SD-SQLD PIC S9(4) COMP.
  09 SD-COLUMN OCCURS 3 TIMES.
  19 SD-DATAFMT.
    29 SQL--NM PIC X(132).
    29 SQL--NMLEN PIC S9(9) COMP.
    29 SQL--DATATYPE PIC S9(9) COMP.
    29 SQL--FORMAT PIC S9(9) COMP.
    29 SQL--MAXLENGTH PIC S9(9) COMP.
    29 SQL--SCALE PIC S9(9) COMP.
    29 SQL--PRECISION PIC S9(9) COMP.
    29 SQL--STATUS PIC S9(9) COMP.
    29 SQL--COUNT PIC S9(9) COMP.
    29 SQL--USERTYPE PIC S9(9) COMP.
    29 SQL--LOCALE PIC S9(9) COMP.
  19 SD-SQLDATA PIC S9(9) COMP.
  19 SD-SQLIND PIC S9(4) COMP.
  19 SD-SQLLEN PIC S9(9) COMP.
  19 SD-SQLMORE PIC S9(9) COMP.
01 conversion-tester pic s9(4) comp-3.
01 charvar pic x(20).
01 temp-int-1 pic s9(9) comp.
01 temp-int-2 pic s9(9) comp.
01 temp-int-3 pic s9(9) comp.
Method 4: Using prepare and fetch with system descriptors

PROCEDURE DIVISION.
  P0.
    MOVE "sa" TO uid.
    move "to pass.
    exec sql connect :uid identified by :pass end-exec.
    * setup
    exec sql whenever sqlwarning perform err-paraend-exec.
    exec sql drop table example end-exec.
    exec sql create table example (fruit char(30), number int) end-exec.
    exec sql insert example values ('tangerine', 1) end-exec.
    exec sql insert example values ('pomegranate', 2) end-exec.
    exec sql insert example values ('banana', 3) end-exec.
    * test functionality using execute
    exec sql prepare statement from
      "select fruit from example where number = ?" end-exec.
    exec sql describe input statement using descriptor
      input-descriptor end-exec.
    if sd-sqld of input-descriptor not equal 1
      or sql--datatype of sd-datafmt of sd-column of
        input-descriptor (1) not equal cs-int-type
      display "failed on first describe input"
      move cs-fail to p-retcode
    end-if.
    move 1 to temp-int-1.
    move 4 to temp-int-2.
    move 0 to temp-int-3.
    call "SYBSETSQLDA" using retcode input-descriptor
      temp-int-1 conversion-tester syb-snines-pic
      temp-int-2 temp-int-3 syb-comp3-usage syb-no-sign .
    move 2 to conversion-tester.
    exec sql describe output statement using descriptor
      output-descriptor end-exec.
    if sd-sqld of output-descriptor not equal
      or sql--datatype of sd-datafmt of sd-column of
        output-descriptor (1) not equal cs-char-type
      display "failed on first describe output"
      move cs-fail to p-retcode
    end-if.
    move 1 to temp-int-1.
    move 20 to temp-int-2.
    move 0 to temp-int-3.
    call "SYBSETSQLDA" using retcode output-descriptor
temp-int-1 charvar syb-x-pic temp-int-2
  temp-int-3 syb-no-usage syb-no-sign.
exec sql execute statement into descriptor
  output-descriptor using descriptor
  input-descriptor end-exec.
display "Expected pomegranate, got "charvar.
exec sql deallocate prepare statement end-exec.
exec sql prepare statement from
  "select number from example where fruit = ?" end-exec.
exec sql declare c cursor for statement end-exec.
exec sql describe input statement using descriptor
  input-descriptor end-exec.
move 1 to temp-int-1.
move 20 to temp-int-2.
move 0 to temp-int-3.
call "SYBSETSQLDA" using retcode input-descriptor
  temp-int-1 charvar syb-x-pic temp-int-2
  temp-int-3 syb-no-usage syb-no-sign.
move "banana" to charvar.
exec sql open c using descriptor input-descriptor end-exec.
exec sql describe output statement using descriptor
  output-descriptor end-exec.
move 1 to temp-int-1.
move 20 to temp-int-2.
move 0 to temp-int-3.
call "SYBSETSQLDA" using retcode output-descriptor
  temp-int-1 charvar syb-x-pic temp-int-2 temp-int-3
  syb-no-usage syb-no-sign.
exec sql fetch c into descriptor output-descriptor
  end-exec.
display "Expected 3, got "charvar.
exec sql commit work end-exec.
end program unittest.
Method 4: Using prepare and fetch with system descriptors
CHAPTER 8

Handling Errors

This chapter discusses how to detect and correct errors that can occur during the execution of Embedded SQL programs. It covers the whenever and get diagnostics statements, which you can use to process warnings and errors, and the SQLCA variables that pertain to warnings and errors.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testing for errors</td>
<td>96</td>
</tr>
<tr>
<td>Testing for warning conditions</td>
<td>96</td>
</tr>
<tr>
<td>Trapping errors with the whenever statement</td>
<td>97</td>
</tr>
<tr>
<td>Using get diagnostics</td>
<td>100</td>
</tr>
<tr>
<td>Writing routines to handle warnings and errors</td>
<td>100</td>
</tr>
<tr>
<td>Precompiler-detected errors</td>
<td>101</td>
</tr>
</tbody>
</table>

While an Embedded SQL application is running, some events may occur that interfere with the application’s operation. Following are examples:

- Adaptive Server Enterprise becomes inaccessible.
- The user enters an incorrect password.
- The user does not have access to a database object.
- A database object is deleted.
- A column’s datatype changes.
- A query returns an unexpected null value.
- A dynamic SQL statement contains a syntax error.

You can anticipate these events by writing warning and error handling code to recover gracefully when one of these situations occurs.
Testing for errors

Embedded SQL places a return code in the *SQLCODE* variable to indicate the success or failure of each SQL statement sent to Adaptive Server Enterprise. You can either test the value of *SQLCODE* after each Embedded SQL statement or use the *whenever* statement to instruct the precompiler to write the test code for you. The *whenever* statement is described later in this chapter.

Using SQLCODE

Table 8-1 lists the values *SQLCODE* can contain:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No warnings or errors occurred.</td>
</tr>
<tr>
<td>&lt;0</td>
<td>Error occurred and the SQLCA variables contain useful information for diagnosing the error.</td>
</tr>
<tr>
<td>100</td>
<td>No rows returned from last statement although the statement executed successfully. This condition is useful for driving a loop that fetches rows from a cursor. When SQLCODE becomes 100, the loop and all rows that have been fetched end. This technique is illustrated in Chapter 6, “Using Transact-SQL Statements.”</td>
</tr>
</tbody>
</table>

Testing for warning conditions

Even when SQLCODE indicates that a statement has executed successfully, a warning condition may still have occurred. The 8-character array SQLCA.SQLWARN indicates such warning conditions. Each SQLWARN array element (or “flag”) stores either the space character (blank) or the character “W”. In each flag, “W” indicates that a warning condition has occurred; the kind of warning condition differs for each flag.

Table 8-2 describes what the space character or “W” means in each flag:
CHAPTER 8 Handling Errors

Table 8-2: SQLWARN flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLWARN1</td>
<td>If blank, no warning condition of any kind occurred, and all other SQLWARN flags are blank. If SQLWARN1 is set to “W,” one or more warning conditions occurred, and at least one other flag is set to “W.”</td>
</tr>
<tr>
<td>SQLWARN2</td>
<td>If set to “W,” the character string variable that you designated in a fetch statement was too short to store the statement’s result data, so the result data was truncated. You designated no indicator variable to receive the original length of the data that was truncated.</td>
</tr>
<tr>
<td>SQLWARN3</td>
<td>If set to “W,” the input sent to Adaptive Server Enterprise contained a null value in an illegal context, such as in an expression or as an input value to a table that prohibits null values.</td>
</tr>
<tr>
<td>SQLWARN4</td>
<td>The number of columns in a select statement’s result set exceeds the number of host variables in the statement’s into clause.</td>
</tr>
<tr>
<td>SQLWARN5</td>
<td>Reserved.</td>
</tr>
<tr>
<td>SQLWARN6</td>
<td>Adaptive Server generated a conversion error while attempting to execute this statement.</td>
</tr>
<tr>
<td>SQLWARN7</td>
<td>Reserved.</td>
</tr>
<tr>
<td>SQLWARN8</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>

Test for a warning after you determine that a SQL statement executed successfully. Use the whenever statement, as described in the next section, to instruct the precompiler to write the test code for you.

Trapping errors with the whenever statement

Use the Embedded SQL whenever statement to trap errors and warning conditions. It specifies actions to be taken depending on the outcome of each Embedded SQL statement sent to Adaptive Server Enterprise.

The whenever statement is not executable. Instead, it directs the precompiler to generate COBOL code that tests for specified conditions after each executable Embedded SQL statement in the program.

The syntax of the whenever statement is:

```
exec sql whenever {sqlwarning | sqlerror |
```
Trapping errors with the whenever statement

not found ]
(continue | goto label | 
program call [using param . . .]) | 
perform paragraph_1 [through paragraph_2] | 
stop};

whenever testing conditions

Each whenever statement can test for one of the following three conditions:

- sqlwarning
- sqlerror
- not found

The precompiler generates warning messages if you do not write a whenever statement for each condition. If you write your own code to check for errors and warnings, suppress the precompiler warnings by writing a whenever...continue clause for each condition. This instructs the precompiler to ignore errors and warnings.

If you precompile with the verbose option, the precompiler generates a ct_debug() function call as part of each connect statement. This causes Client-Library to display informational, warning, and error messages to your screen as your application runs. The whenever statement does not disable these messages. See the Open Client and Open Server Programmers Supplement.

After an Embedded SQL statement executes, the values of SQLCODE and SQLWARN1 determine if one of the conditions exists. Table 8-3 shows the criteria whenever uses to detect the conditions:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqlwarning</td>
<td>SQLCODE = 0 and SQLWARN1 = W</td>
</tr>
<tr>
<td>sqlerror</td>
<td>SQLCODE &lt; 0</td>
</tr>
<tr>
<td>not found</td>
<td>SQLCODE = 100</td>
</tr>
</tbody>
</table>

To change the action of a whenever statement, write a new whenever statement for the same condition. whenever applies to all Embedded SQL statements that follow it, up to the next whenever statement for the same condition.

The whenever statement ignores the application program’s logic. For example, if you place whenever at the end of a loop, it does not affect the preceding statements in subsequent passes through the loop.
whenever actions

The whenever statement specifies one of the following five actions:

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>continue</td>
<td>Causes no special action when a SQL statement returns the specified condition. Normal processing continues.</td>
</tr>
<tr>
<td>goto</td>
<td>Causes a branch to an error-handling procedure within your application program. You can enter goto as either “goto” or “go to”, followed by a valid paragraph name. The precompiler does not detect an error if the paragraph name is not defined in the program, but the COBOL compiler does.</td>
</tr>
<tr>
<td>call</td>
<td>Calls another COBOL program and, optionally, passes variables.</td>
</tr>
<tr>
<td>perform</td>
<td>Names at least one paragraph to execute when a SQL statement results in the specified condition. You can use the COBOL perform statement formats 1, 2, 3, and 4 in the perform clause. If you use a paragraph name, the paragraph must be in the section where the whenever condition applies.</td>
</tr>
<tr>
<td>stop</td>
<td>Terminates the program when a SQL statement triggers the specified condition.</td>
</tr>
</tbody>
</table>

```
exec SQL whenever sqlerror perform ERR-PARA thru ERR-PARA-END
end-exec

exec SQL select au_lname from authors
    into :AU-LNAME
    where au_id = :AU-ID
end-exec

exec SQL update authors set au_lname = :AU-LNAME
    where au_id = :AU-ID
end-exec
```
Using get diagnostics

The get diagnostics statement retrieves error, warning, and informational messages from Client-Library. It is similar to—but more powerful than—the whenever statement because you can expand it to retrieve more details of the detected errors.

If, within a whenever statement, you specify the application to go to or call another application or paragraph, specify get diagnostics in the procedure code, as follows:

```
err-handler.
    exec sql get diagnostics :num-msgs = number
    end-exec.
    perform varying condcnt from 0 by 1
    until condcnt greater or equal num-msgs
    exec sql get diagnostics exception :condcnt
        :sqlca = sqlca_info end-exec
    display "sqlcode is " sqlcode
    display "message text is " sqlerrmc
    end-perform.
```

Writing routines to handle warnings and errors

A good strategy for handling errors and warnings in an Embedded SQL application is to write custom procedures to handle them, then install the procedures with the whenever...perform statement.

The following example shows sample warning and error handling routines. For simplicity, both routines omit certain conditions that should normally be included: warn_para omits the code for SQLWARN1, and err_para omits the code that handles Client-Library errors and operating system errors:

```
* Declare the sqlca.  *
    exec sql include sqlca end-exec
    exec sql whenever sqlerror call "ERR-PARA"
        end-exec
    exec sql whenever sqlwarning call
        "WARN-PARA" end-exec
    exec sql whenever not found continue end-exec

WARN-PARA.
    * Displays error codes and numbers from the sqlca
```
and exits with an ERREXIT status.

    DISPLAY "Warning code is " SQLCODE.
    DISPLAY "Warning message is " SQLERRMC.
    
    IF SQLWARN2 EQUAL "W"
      DISPLAY "Data has been truncated.".
    IF SQLWARN3 EQUAL "W"
      DISPLAY "A null value was eliminated from 
            the argument set of a function.".
    IF SQLWARN4 EQUAL "W"
      DISPLAY "An into clause had too many or too 
            few host variables.".
    IF SQLWARN5 EQUAL "W"
      DISPLAY "A dynamic update or delete was 
            lacking a where clause.".
    IF SQLWARN6 EQUAL "W"
      DISPLAY "A server conversion or truncation 
            error occurred.".
    
WARN-PARA-END.
EXIT.

ERR-PARA.
* Print the error code, the error message, and the * line number of the command that caused the * error.

    DISPLAY "Error code is " SQLCODE.
    DISPLAY "Error message is " SQLERRMC.
    STOP RUN.

Precompiler-detected errors

The Embedded SQL precompiler detects Embedded SQL errors at precompile time. The precompiler detects syntax errors such as missing semicolons and undeclared host variables in SQL statements. These are severe errors, so appropriate error messages are generated.

You can also have the precompiler check Transact-SQL syntax errors. Adaptive Server Enterprise parses Transact-SQL statements at precompile time if the appropriate precompiler command options are set. See the precompiler reference page in the Open Client and Open Server Programmers Supplement.
Precompiler-detected errors

The precompiler substitutes host variables in Embedded SQL statements with dynamic parameter markers ("?"). Occasionally, substituting host variables with parameter markers causes syntax errors (for example, when rules or triggers do not allow the parameters).

The precompiler does not detect the error in the following example, in which a table is created and data is selected from it. The error is that the host variables’ datatypes do not match the columns retrieved. The precompiler does not detect the error because the table does not yet exist when the precompiler parses the statements:

```sql
exec sql begin declare section end-exec
  01 VAR1           PIC S9(9) COMP.
  02 VAR2           PIC X(20).
exec sql end declare section end-exec

exec sql create table T1
  (col1 int, col2 varchar(20)) end-exec
...

exec sql select * from T1 into
  :VAR2, :VAR1 end-exec.
```

Note that the error will be detected and reported at runtime.
Embedded SQL Statements: Reference Pages

This chapter consists of a reference page for each Embedded SQL statement that either does not exist in Transact-SQL or works differently from the way it works in Transact-SQL. Refer to the *Adaptive Server Enterprise Transact-SQL Users Guide* for descriptions of all other Transact-SQL statements that are valid in Embedded SQL.

<table>
<thead>
<tr>
<th>Command Statements</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>allocate descriptor</td>
<td>105</td>
</tr>
<tr>
<td>begin declare section</td>
<td>106</td>
</tr>
<tr>
<td>begin transaction</td>
<td>107</td>
</tr>
<tr>
<td>close</td>
<td>109</td>
</tr>
<tr>
<td>commit</td>
<td>110</td>
</tr>
<tr>
<td>connect</td>
<td>113</td>
</tr>
<tr>
<td>deallocate cursor</td>
<td>115</td>
</tr>
<tr>
<td>deallocate descriptor</td>
<td>116</td>
</tr>
<tr>
<td>deallocate prepare</td>
<td>118</td>
</tr>
<tr>
<td>declare cursor (dynamic)</td>
<td>119</td>
</tr>
<tr>
<td>declare cursor (static)</td>
<td>121</td>
</tr>
<tr>
<td>declare cursor (stored procedure)</td>
<td>123</td>
</tr>
<tr>
<td>declare scrollable cursor</td>
<td>125</td>
</tr>
<tr>
<td>delete (positioned cursor)</td>
<td>126</td>
</tr>
<tr>
<td>delete (searched)</td>
<td>128</td>
</tr>
<tr>
<td>describe input (SQL descriptor)</td>
<td>130</td>
</tr>
<tr>
<td>describe input (SQLDA)</td>
<td>132</td>
</tr>
<tr>
<td>describe output (SQL descriptor)</td>
<td>133</td>
</tr>
<tr>
<td>describe output (SQLDA)</td>
<td>135</td>
</tr>
<tr>
<td>disconnect</td>
<td>137</td>
</tr>
<tr>
<td>exec</td>
<td>139</td>
</tr>
<tr>
<td>exec sql</td>
<td>142</td>
</tr>
<tr>
<td>execute</td>
<td>143</td>
</tr>
<tr>
<td>execute immediate</td>
<td>145</td>
</tr>
<tr>
<td>exit</td>
<td>146</td>
</tr>
</tbody>
</table>
Except for print, raiseerror, readtext, and writetext, all Transact-SQL statements can be used in Embedded SQL, although the syntax of some statements differs, as described in this chapter.

The reference pages in this chapter are arranged alphabetically. Each statement’s reference page:

- Briefly states what the statement does
- Describes the statement’s syntax
- Explains the statement’s keywords and options
- Comments on the statement’s proper use
- Lists related statements, if any
- Demonstrates the statement’s use in a brief example
allocate descriptor

Description
Allocates a SQL descriptor.

Syntax
exec sql allocate descriptor descriptor_name
[with max [host_variable | integer_literal]]
end-exec

Parameters
descriptor_name
The name of the SQL descriptor that will contain information about the
dynamic parameter markers in a prepared statement.

with max
The maximum number of columns in the SQL descriptor.

host_variable
An integer host variable defined in a declare section.

integer_literal
A numeric value representing the size, in number of occurrences, of the SQL
descriptor.

Examples
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 COLTYPE PIC S9(9) COMP.
  01 NUMCOLS PIC S9(9) COMP.
  01 COLNUM PIC S9(9) COMP.
EXEC SQL END DECLARE SECTION END-EXEC.

... 

EXEC SQL ALLOCATE DESCRIPTOR big_desc WITH MAX 1000 END-EXEC.

EXEC SQL PREPARE dynstmt FROM "select * from huge_table" END-EXEC.

* Assume that the select returns only 1 row.
EXEC SQL EXECUTE dynstmt INTO SQL DESCRIPTOR big_desc END-EXEC.

EXEC SQL GET DESCRIPTOR big_desc :NUMCOLS = COUNT END-EXEC.

MOVE 1 TO COLNUM.
PERFORM GET-DESC-LOOP UNTIL COLNUM > NUMCOLS.

EXEC SQL DEALLOCATE DESCRIPTOR big_desc END-EXEC.
EXEC SQL DEALLOCATE PREPARE dynstmt END-EXEC.
... 

GET-DESC-LOOP.
EXEC SQL GET DESCRIPTOR big_desc VALUE
:COLNUM :COLTYPE = TYPE END-EXEC.
DISPLAY "COLUMN ",COLNUM," IS OF TYPE ", COLTYPE.
ADD 1 TO COLNUM.

Usage
- The allocate descriptor command specifies the number of item descriptor areas that Adaptive Server Enterprise allocates.
- You can allocate any number of SQL descriptors.
- When a SQL descriptor is allocated, its fields are undefined.
- If you try to allocate a SQL descriptor that is already allocated, an error occurs.
- If you do not specify a value for the with max clause, one item descriptor is assigned.
- When a SQL descriptor is allocated, the value of each of its fields is undefined.

See also deallocate descriptor, get descriptor, set descriptor

### begin declare section

**Description**
Begins a declare section, which declares host language variables used in an Embedded SQL source file.

**Syntax**
```
exec sql begin declare section end-exec
host_variable_declaration.
```

**Parameters**
`host_variable_declaration`
The declaration of one or more host language variables.

**Examples**
```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 TITLE PIC X(80).
  01 VAR1 PIC S9(9) COMP.
  01 VAR2 PIC X(100).
EXEC SQL END DECLARE SECTION END-EXEC.
```

**Usage**
- A declare section must end with the Embedded SQL statement end declare section.
- A source file can have any number of declare sections.
· declare sections can be placed anywhere that variables can be declared. The declare section that declares a variable must precede any statement that references the variable.

· Variable declarations in a declare section must conform to the rules of the host language.

· Nested structures are valid in a declare section; arrays of structures are not.

· A declare section can contain any number of Embedded SQL include statements.

· When processing Embedded SQL include statements within a declare section, the Embedded SQL precompiler treats the contents of the included file as though had been entered directly into the file being precompiled.

See also exec sql include "filename"

begin transaction

Description Marks the starting point of an unchained transaction.

Syntax exec sql [at connection_name]
begin {transaction | tran} [transaction_name] 
end-exec

Parameters transaction | tran

The keywords transaction and tran are interchangeable.

transaction_name

The name that you are assigning to this transaction. The name must conform to the rules for Transact-SQL identifiers.

Examples

* Use explicit transactions to synchronize tables on two servers.
*

EXEC SQL BEGIN DECLARE SECTION END-EXEC.

  01 TITLE-ID PIC X(6).
  01 NUM-SOLD PIC S9(9) COMP.

EXEC SQL END DECLARE SECTION END-EXEC.
EXEC SQL WHENEVER SQLERROR PERFORM ABORT-TRAN END-EXEC.

EXEC SQL CONNECT :UID IDENTIFIED BY :PASS
   AT connect1 END-EXEC.
EXEC SQL CONNECT :UID IDENTIFIED BY :PASS
   AT connect2 END-EXEC.

PERFORM TRY-UPDATE.

EXEC SQL AT connect1 BEGIN TRANSACTION END-EXEC.
EXEC SQL AT connect2 BEGIN TRANSACTION END-EXEC.

EXEC SQL AT connect1 SELECT sum(qty) INTO :NUM-SOLD
   FROM salesdetail
   WHERE title_id = :TITLE-ID END-EXEC.
EXEC SQL AT connect2 UPDATE current_sales
   SET num_sold = :NUM-SOLD
   WHERE title_id = :TITLE-ID END-EXEC.

EXEC SQL AT connect2 COMMIT TRANSACTION END-EXEC.
EXEC SQL AT connect1 COMMIT TRANSACTION END-EXEC.

IF SQLCODE <> 0
   DISPLAY "OOPS! Should have used 2-phase commit".

EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
   DISPLAY "Error code is " SQLCODE.
   DISPLAY "Error message is " SQLERRMC.
EXEC SQL AT connect2 ROLLBACK TRANSACTION END-EXEC.
EXEC SQL AT connect1 ROLLBACK TRANSACTION END-EXEC.
PERFORM TRY-UPDATE.

**Usage**

- This reference page describes aspects of the Transact-SQL begin transaction statement that differ when used with Embedded SQL. See the *Adaptive Server Enterprise Reference Manual*.
- The begin transaction statement is valid only in unchained transaction mode. In chained transaction mode, you cannot explicitly mark the starting point of a transaction.
• When nesting transactions, assign a transaction name only to the outermost begin transaction statement and its corresponding commit transaction or rollback transaction statement.

• Unless you set the database option ddl in tran, Adaptive Server Enterprise does not allow the following statements inside an unchained transaction: create database, create table, create index, create view, drop statements, select into table_name, grant, revoke, alter database, alter table, truncate table, update statistics, load database, load transaction, and disk init.

• A transaction includes only statements that execute on the connection that is current when the transaction begins.

• Remote procedures execute independently of any transaction in which they are included.

See also commit transaction, commit work, rollback transaction, rollback work

close

Description Closes an open cursor.

Syntax exec sql [at connection_name] close cursor_name end-exec

Parameters cursor_name The name of the cursor to be closed; that is, the name that you assigned when declaring the cursor.

Examples

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 LNAME   PIC X(40).
  01 FNAME   PIC X(20).
  01 PHONE   PIC X(12).
EXEC SQL END DECLARE SECTION END-EXEC.

...

EXEC SQL DECLARE authorlist CURSOR FOR
  SELECT au_lname, au_fname, phone
  FROM authors END-EXEC.

EXEC SQL OPEN authorlist END-EXEC.
PERFORM FETCH-LOOP UNTIL SQLCODE = 100.
commit

EXEC SQL CLOSE authorlist END-EXEC,
...

FETCH-LOOP.
EXEC SQL FETCH authorlist INTO
:LNAME, :FNAME, :PHONE END-EXEC.
DISPLAY LNAME, FNAME, PHONE.

Usage
• The close statement closes an open cursor. Unfetched rows are canceled.
• Reopening a closed cursor executes the associated query again, positioning the cursor pointer before the first row of the result set.
• A cursor must be closed before it is reopened.
• Attempting to close a cursor that is not open causes a runtime error.
• The commit transaction, rollback transaction, commit work, and rollback work statements close a cursor automatically unless you set a precompiler option to disable the feature.
• Closing and then reopening a cursor lets your program see any changes in the tables from which the cursor retrieves rows.

See also declare cursor, fetch, open, prepare

commit

Description
Ends a transaction, preserving changes made to the database during the transaction.

Syntax
exec sql [at connection_name] commit [transaction | tran | work] [transaction_name] end-exec

Parameters
transaction | tran | work

The keywords transaction, trans, and work are interchangeable in the rollback statement, except that only work is ANSI-compliant.

transaction_name
A name assigned to the transaction.

Examples

Example 1
* Using unchained transaction mode to synchronize tables on two servers.

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 TITLE-ID PIC X(7).
01 NUM-SOLD PIC S9(9).
EXEC SQL END DECLARE SECTION END-EXEC.
...
EXEC SQL CONNECT :UID IDENTIFIED BY :PASS
AT connect1 END-EXEC.
EXEC SQL CONNECT :UID IDENTIFIED BY :PASS
AT connect2 END-EXEC.
...
PERFORM TRY-UPDATE.
TRY-UPDATE.
EXEC SQL AT connect1 BEGIN TRANSACTION END-EXEC.
EXEC SQL AT connect2 BEGIN TRANSACTION END-EXEC.
EXEC SQL AT connect1 SELECT sum(qty) INTO :NUM-SOLD
FROM salesdetail
WHERE title_id = :TITLE-ID END-EXEC.
EXEC SQL AT connect2 UPDATE current_sales
SET num_sold = :NUM-SOLD
WHERE title_id = :TITLE-ID END-EXEC.
EXEC SQL AT connect2 COMMIT TRANSACTION END-EXEC.
EXEC SQL AT connect1 COMMIT TRANSACTION END-EXEC.
IF SQLCODE <> 0
   DISPLAY "Oops! Should have used 2-phase commit".
```

**Example 2**

* Using chained transaction mode to synchronize tables on two servers.

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 TITLE-ID PIC X(7).
01 NUM-SOLD PIC S9(9) COMP.
EXEC SQL END DECLARE SECTION END-EXEC.
```
EXEC SQL WHENEVER SQLERROR PERFORM ABORT-TRAN END-EXEC.

PERFORM TRY-UPDATE.

TRY-UPDATE.
EXEC SQL AT connect1 SELECT sum(qty) INTO :NUM-SOLD
    FROM salesdetail
    WHERE title_id = :TITLE-ID END-EXEC.

EXEC SQL AT connect2 UPDATE current_sales
    SET num_sold = :NUM-SOLD
    WHERE title_id = :TITLE-ID END-EXEC.

EXEC SQL AT connect2 COMMIT WORK END-EXEC.
EXEC SQL AT connect1 COMMIT WORK END-EXEC.

IF SQLCODE <> 0
    DISPLAY "OOPS! Should have used 2-phase commit".

ABORT-TRAN.
    DISPLAY "ERROR! ABORTING TRAN".
    DISPLAY "Error code is " SQLCODE.
    DISPLAY "Error message is " SQLERRMC.
EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
EXEC SQL AT connect2 ROLLBACK WORK END-EXEC.
EXEC SQL AT connect1 ROLLBACK WORK END-EXEC.
PERFORM TRY-UPDATE.

Usage

- This reference page mainly describes aspects of the Transact-SQL commit statement that differ when used with Embedded SQL. See the Adaptive Server Enterprise Reference Manual.

- Transaction names must conform to the Transact-SQL rules for identifiers. Transaction names are a Transact-SQL extension: they cannot be used with the ANSI-compliant keyword work.

- When nesting transactions, assign a transaction name only to the outermost begin transaction statement and its corresponding commit transaction or rollback transaction statement.

See also

begin transaction, commit work, rollback transaction, rollback work
connect

Description
Creates a connection to Adaptive Server Enterprise.

Syntax
exec sql connect user_name
[identified by password] [at connection_name]
[using server_name] [labelname label_name labelvalue label_value ...] end-exec

Parameters

user_name
The user name to be used when logging into Adaptive Server Enterprise.

password
The password to use to log in to Adaptive Server Enterprise.

connection_name
A name that you choose to uniquely identify the Adaptive Server Enterprise connection.

server_name
The server name of the Adaptive Server Enterprise to which you are connecting.

Examples

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01         UID             PIC X(32).
  01         PASS            PIC X(32).
  01         SERVER          PIC X(100).
EXEC SQL END DECLARE SECTION END-EXEC.

DISPLAY "UID NAME?".
ACCEPT UID.
DISPLAY "PASSWORD ?".
ACCEPT PASS.
DISPLAY "SERVER TO CONNECT TO ?".
ACCEPT SERVER.

EXEC SQL CONNECT :UID IDENTIFIED BY :PASS
  USING :SERVER END-EXEC.

Usage

- In every Embedded SQL program, the connect statement must be executed before any other executable SQL statement except allocate descriptor.
- The label_name and label_value clauses, if used, must be the last clauses of the connect statement.
- If a program uses both C and COBOL languages, the first connect statement must be issued from a COBOL program.
If a program has multiple connections, only one can be unnamed.

If an Embedded SQL statement does not have an at connection_name clause to direct it to a specific named connection, the statement is executed on the current connection.

To specify a null password, omit the identified by clause or use an empty string.

If the connect statement does not specify an Adaptive Server Enterprise, the server named by the DSQUERY environment variable or logical name is used. If DSQUERY is not defined, the default server is SYBASE.

Client-Library looks up the server name in the interfaces file located in the directory specified by the SYBASE environment variable or logical name.

The Adaptive Server Enterprise connection ends when the Embedded SQL program exits or issues a disconnect statement.

Opening a new connection, named or unnamed, results in the new connection becoming the current connection.

A program that requires multiple Adaptive Server Enterprise login names can have a connection for each login account.

By connecting to more than one server, a program can simultaneously access data stored on different servers.

A single program can have multiple connections to a single server or multiple connections to different servers.

Table 9-1 shows how a connection is named:

<table>
<thead>
<tr>
<th>If this clause is used</th>
<th>But without</th>
<th>Then, the ConnectionName is</th>
</tr>
</thead>
<tbody>
<tr>
<td>at connection_name</td>
<td></td>
<td>connection_name</td>
</tr>
<tr>
<td>using server_name</td>
<td>at</td>
<td>server_name</td>
</tr>
<tr>
<td>None</td>
<td></td>
<td>DEFAULT</td>
</tr>
</tbody>
</table>

See also at connection_name, exec sql, disconnect, set connection
deallocate cursor

Description
Deallocates a cursor for a static SQL statement or for a dynamic SQL statement.

Syntax
exec sql [at connection_name] deallocate cursor cursor_name end-exec

Parameters
cursor_name
The name of the cursor to be deallocated. The cursor_name must be a character string enclosed in double quotation marks or in no quotation marks—for example "my_cursor" or my_cursor. It cannot be a host variable.

Examples
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 TITLE-ID PIC X(7).
  01 BOOK-NAME PIC X(80).
  01 TTYPE PIC X(12).
  01 TITLE-INDIC S9(9).
  01 TYPE-INDIC S9(9).
EXEC SQL END DECLARE SECTION END-EXEC.

...

EXEC SQL DECLARE titlelist CURSOR FOR
   SELECT type, title_id, title FROM titles
   order by type END-EXEC.

EXEC SQL OPEN titlelist END-EXEC.

PERFORM FETCH-PARA UNTIL SQLCODE = 100.

EXEC SQL CLOSE titlelist END-EXEC.
EXEC SQL DEALLOCATE CURSOR titlelist END-EXEC.
...

FETCH-PARA.
   EXEC SQL FETCH titlelist INTO :
      TTYPE   :TYPE-INDIC,
      TITLE-ID,
      BOOK-NAME :TITLE-INDIC END-EXEC.

   IF TYPE-INDIC <> -1
      DISPLAY "TYPE  : ", TTYPE
   ELSE
      DISPLAY "TYPE  : UNDECIDED"
   END-EXEC.
**deallocate descriptor**

Deallocates a SQL descriptor.

**Usage**
- Deallocating a cursor releases all resources allocated to the cursor. In particular, `deallocate cursor` drops the Client-Library command handle and CS_COMMAND structure associated with the cursor.
- A static cursor can be deallocated at any time after it is opened. A dynamic cursor can be deallocated at any time after it is declared.
- If `cursor_name` is open, `deallocate cursor` closes it and then deallocates it.
- You cannot reference a deallocated cursor, nor can you reopen it. If you try, an error occurs.
- You can declare a new cursor having the same name as that of a deallocated cursor. Opening a cursor with the same name as a deallocated cursor is not the same as reopening the deallocated cursor. Other than the name, the new cursor shares nothing with the deallocated cursor.
- Declaring a new cursor with the same name as that of a deallocated cursor can cause the precompiler to generate a warning message.
- The `deallocate cursor` statement is a Sybase extension; it is not defined in the SQL standard.

**Note** If you are using persistent binding in your Embedded SQL program, use the `deallocate cursor` statement carefully. Needlessly deallocating cursors can negate the advantage of persistent binding.

**See also**
- close cursor, declare cursor, open (static cursor)
Syntax

```
exec sql deallocate descriptor descriptor_name
end-exec
```

**Parameters**

`descriptor_name`

The name of the SQL descriptor that contains information about the dynamic parameter markers or return values in a prepared statement.

**Examples**

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 NUMCOLS PIC S9(9) COMP.
  01 COLNUM PIC S9(9) COMP.
  01 COLTYPE PIC S9(9) COMP.
EXEC SQL END DECLARE SECTION END-EXEC.
...

EXEC SQL ALLOCATE DESCRIPTOR big_desc WITH MAX 100 END-EXEC.
EXEC SQL PREPARE dynstmt FROM "select * from huge_table" END-EXEC.

* Assume that only one row of data is returned.
  EXEC SQL EXECUTE dynstmt INTO SQL DESCRIPTOR big_desc END-EXEC.
EXEC SQL GET DESCRIPTOR big_desc :NUMCOLS = COUNT END-EXEC.

MOVE 1 TO COLNUM.
PERFORM GET-DESC-LOOP UNTIL COLNUM > NUMCOLS.

EXEC SQL DEALLOCATE DESCRIPTOR big_desc END-EXEC.
...

GET-DESC-LOOP.
EXEC SQL GET DESCRIPTOR big_desc VALUE 
  :COLNUM :COLTYPE = TYPE END-EXEC.
DISPLAY "COLUMN TYPE = ",COLTYPE.
ADD 1 TO COLNUM.
```

**Usage**

- If you attempt to deallocate a SQL descriptor that has not been allocated, an error occurs.

**See also**

allocate descriptor
deallocate prepare

Description
Deallocates a dynamic SQL statement that was prepared in a prepare statement.

Syntax
\[
\text{exec sql [at connection_name]}
\text{deallocate prepare statement_name end-exec}
\]

Parameters
\textit{statement_name}
The identifier assigned to the dynamic SQL statement when the statement was prepared.

Examples
\[
\text{EXEC SQL BEGIN DECLARE SECTION END-EXEC.}
\text{01 CMDBUF PIC X(120).}
\text{01 STATE PIC X(3).}
\text{EXEC SQL END DECLARE SECTION END-EXEC.}
\]

* The 'select into table' statement returns no results to the program, so it does not need a cursor.

\[
\text{MOVE "select * into tmp from authors where state = \?"}
\text{TO CMDBUF.}
\text{DISPLAY "STATE \? ".}
\text{ACCEPT STATE.}
\text{EXEC SQL PREPARE dynstmt FROM :CMDBUF END-EXEC.}
\text{EXEC SQL EXECUTE dynstmt USING :STATE END-EXEC.}
\]

\[
\text{EXEC SQL DEALLOCATE PREPARE dynstmt END-EXEC.}
\text{EXEC SQL COMMIT WORK END-EXEC.}
\]

Usage
\begin{itemize}
\item A statement must be prepared before it is deallocated. Attempting to deallocate a statement that has not been prepared results in an error.
\item \textit{statement_name} must uniquely identify a statement buffer and must conform to the SQL identifier rules for naming variables. \textit{statement_name} can be either a literal or a character array host variable.
\end{itemize}
The deallocate prepare statement closes and deallocates any dynamic cursors declared for statement_name.

**Warning!** If you are using persistent binds in your Embedded SQL program, use the deallocate prepare statement carefully. Needlessly deallocating prepared statements can negate the advantage of persistent binds.

**See also**
declare cursor (dynamic), execute, execute immediate, prepare

declare cursor (dynamic)

**Description**
Declares a cursor for processing multiple rows returned by a prepared dynamic select statement.

**Syntax**
exec sql [at connection_name]
declare cursor_name
cursor for prepped_statement_name end-exec

**Parameters**

- **cursor_name**
  The cursor’s name, used to reference the cursor in open, fetch, and close statements. A cursor’s name must be unique on each connection and must have no more than 255 characters.

- **prepped_statement_name**
  The name (specified in a previous prepare statement) that represents the select statement to be executed.

**Examples**

```sql
EXEC SQL BEGIN DECLARE SECTION END-EXEC

01 QUERY       PIC X(100).
01 DATAVAL     PIC X(100).
01 COUNTER     PIC S9(9) COMP.
01 NUMCOLS     PIC S9(9) COMP.
01 COLNAME     PIC X(32).
01 COLTYPE     PIC S9(9) COMP.
01 COLLEN      PIC S9(9) COMP.

EXEC SQL END DECLARE SECTION END-EXEC.

EXEC SQL WHENEVER SQLERROR PERFORM ERR-Para END-EXEC.
EXEC SQL WHENEVER SQLWARNING PERFORM WARN-Para END-EXEC
```
EXEC SQL WHENEVER NOT FOUND STOP END-EXEC.

EXEC SQL USE pubs2 END-EXEC.
MOVE "SELECT * FROM publishers" TO QUERY.
EXEC SQL ALLOCATE_DESCRIPTOR dout WITH MAX 100 END-EXEC.
EXEC SQL PREPARE dynstmt FROM :QUERY END-EXEC.
EXEC SQL DECLARE dyncur CURSOR FOR dynstmt END-EXEC.
EXEC SQL OPEN dyncur END-EXEC.
PERFORM FETCH-LOOP UNTIL SQLCODE = 100.

* Clean-up all open cursors, descriptors and dynamic statements.

EXEC SQL CLOSE dyncur END-EXEC.
EXEC SQL DEALLOCATE CURSOR dyncur END-EXEC.
EXEC SQL DEALLOCATE PREPARE dynstmt END-EXEC.
EXEC SQL DEALLOCATE DESCRIPTOR dout END-EXEC.
EXEC SQL COMMIT WORK END-EXEC.
STOP RUN.

FETCH-LOOP.
EXEC SQL FETCH dyncur INTO SQL_DESCRIPTOR dout END-EXEC
EXEC SQL GET_DESCRIPTOR dout :NUMCOLS = COUNT END-EXEC
DISPLAY "COLS = ", NUMCOLS
MOVE 1 TO COUNTER
PERFORM GET-DESC-PARA UNTIL COUNTER > NUMCOLS.
END-FETCH-LOOP.

GET-DESC-PARA.
EXEC SQL GET_DESCRIPTOR dout VALUE :COUNTER
:COLNAME = NAME,
:COLTYPE = TYPE,
:COLLEN = LENGTH
END-EXEC
DISPLAY "NAME :", COLNAME
DISPLAY "TYPE :", COLTYPE
DISPLAY "LENGTH :", COLLEN

EXEC SQL GET_DESCRIPTOR dout VALUE :COUNTER
:DATAVAL = DATA END-EXEC
DISPLAY "DATA :", DATAVAL
DISPLAY " ", DATAVAL
ADD 1 TO COUNTER.
END-GET-DESC-PARA.

Usage

- The prepped_statement_name must not have a compute clause.
• The cursor_name must be declared on the connection where prepped_statement_name was prepared.

• The dynamic declare cursor statement is an executable statement, whereas the static declare cursor statement is simply a declaration. The dynamic declare statement must be located where the host language allows executable statements and the program should check return codes (SQLCODE, SQLCA, or SQLSTATE).

• The for update and read only clauses for a dynamic cursor are not part of the declare cursor statement but rather should be included in the prepared statement’s select query.

See also close, connect, fetch, open, prepare

declare cursor (static)

Description
Declares a cursor for processing multiple rows returned by a select statement.

Syntax
exec sql declare cursor_name
cursor for select_statement
[for update [of col_name_1 [, col_name_n]...]]
for read only] end-exec

Parameters

cursor_name
The cursor’s name, used to reference the cursor in open, fetch, and close statements. A cursor’s name must be unique on each connection and must have no more than 255 characters.

select_statement
The Transact-SQL select statement to be executed when the cursor is opened. See the description of the select statement in the Adaptive Server Enterprise Reference Manual.

for update
Specifies that the cursor’s result list can be updated. (To update the result list, you use the update statement.

of col_name_n
The name of a column to be updated.

for read only
Specifies that the cursor’s result list cannot be updated.
**declare cursor (static)**

**Examples**

```sql
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 TITLE-ID      PIC X(6).
01 BOOK-NAME         PIC X(25).
01 TYPE           PIC X(15).
EXEC SQL END DECLARE SECTION END-EXEC.
01 ANSWER           PIC X(1).

DISPLAY "TYPE OF BOOKS TO RETRIEVE ? ".
ACCEPT BOOK-TYPE.
EXEC SQL DECLARE titlelist CURSOR FOR
SELECT title_id, substring(title,1,25) FROM
    titles WHERE type = :BOOK-TYPE END-EXEC.

EXEC SQL OPEN titlelist END-EXEC.
PERFORM FETCH-PARA UNTIL SQLCODE = 100.
EXEC SQL CLOSE titlelist END-EXEC.
EXEC SQL DEALLOCATE CURSOR titlelist END-EXEC.
EXEC SQL COMMIT WORK END-EXEC.

FETCH-PARA.
EXEC SQL FETCH titlelist INTO
    :TITLE-ID, :BOOK-NAME END-EXEC.
DISPLAY "TITLE ID : ",TITLE-ID
DISPLAY "TITLE : ",BOOK-NAME
IF SQLCODE = 100
    DISPLAY "NO RECORDS TO FETCH. END OF PROGRAM RUN."
ELSE
    DISPLAY "UPDATE/DELETE THIS RECORD (U/D)? ".
    ACCEPT ANSWER.
    IF ANSWER = "U"
        DISPLAY "ENTER NEW TITLE :"
        ACCEPT BOOK-NAME
        EXEC SQL UPDATE titles SET title = :BOOK-NAME
            WHERE CURRENT OF titlelist END-EXEC
    ELSE
        IF ANSWER = "D"
            EXEC SQL DELETE titles
                WHERE CURRENT OF titlelist END-EXEC
        END-IF
```
Usage

- The Embedded SQL precompiler generates no code for the declare cursor statement.
- The select_statement does not execute until your program opens the cursor by using the open cursor statement.
- The syntax of the select_statement is identical to that shown in the Adaptive Server Enterprise Reference Manual, except that you cannot use the compute clause in Embedded SQL.
- The select_statement can contain host variables. The values of the host variables are substituted when your program opens the cursor.
- If you omit either the for update or read only clause, Adaptive Server Enterprise determines whether the cursor is updatable.

See also

close, connect, deallocate cursor, declare cursor (stored procedure),
declare cursor (dynamic), fetch, open, update

declare cursor (stored procedure)

Description
Declares a cursor for a stored procedure.

Syntax

cursor for execute procedure_name
(@param_name = :host_var)...
end-exec

Parameters

cursor_name
- The cursor's name, used to reference the cursor in open, fetch, and close statements. A cursor's name must be unique on each connection and must have no more than 255 characters.

procedure_name
- The name of the stored procedure to be executed.

param_name
- The name of a parameter in the stored procedure.

host_var
- The name of a host variable to be passed as a parameter value.
declare cursor (stored procedure)

Examples

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 TITLE-ID    PIC X(6).
  01 BOOK-NAME   PIC X(65).
  01 BOOK-TYPE   PIC X(15).
EXEC SQL END DECLARE SECTION END-EXEC.
  01 ANSWER     PIC X(1).
  ....

* Create the stored procedure.

EXEC SQL create procedure p_titles (@p_type varchar(30))
  as
    select title_id, substring(title,1,64)
    from titles
    where type = @p_type
END-EXEC.

* To execute stored procedures, you must disable chained mode.
EXEC SQL SET CHAINED OFF END-EXEC.

DISPLAY "TYPE OF BOOKS TO RETRIEVE ? ".
ACCEPT BOOK-TYPE.
EXEC SQL DECLARE titlelist CURSOR FOR
  execute p_titles :BOOK-TYPE END-EXEC.
EXEC SQL OPEN titlelist END-EXEC.
PERFORM FETCH-LOOP UNTIL SQLCODE = 100.
EXEC SQL CLOSE titlelist END-EXEC.
EXEC SQL COMMIT WORK END-EXEC.

FETCH-LOOP.
EXEC SQL FETCH titlelist INTO
  :TITLE-ID, :BOOK-NAME END-EXEC
DISPLAY "TITLE ID : ", TITLE-ID
DISPLAY "TITLE    : ", BOOK-NAME
IF SQLCODE = 100
    DISPLAY "NO RECORDS TO FETCH. END OF PROGRAM RUN."
ELSE
    DISPLAY "UPDATE/DELETE THIS RECORD ? 
    ACCEPT ANSWER
    IF ANSWER = "U"
        DISPLAY "ENTER NEW TITLE :"
ACCEPT BOOK-NAME
EXEC SQL UPDATE titles SET title = :BOOK-NAME
WHERE CURRENT OF titlelist END-EXEC.
ELSE
IF ANSWER = "D"
EXEC SQL DELETE titles WHERE CURRENT OF
titlelist END-EXEC
END-IF
END-IF.

Usage

- procedure_name must consist of only one select statement.
- It is not possible to retrieve output parameter values from a stored procedure executed using a cursor.
- It is not possible to retrieve the return status value of a stored procedure executed using a cursor.

See also close, deallocate cursor, declare cursor (static), declare cursor (dynamic), fetch, open, update

**declare scrollable cursor**

**Description**
Declares a scrollable cursor.

**Syntax**
EXEC SQL DECLARE <curs_name>
[ <cursor sensitivity> ]
[ <cursor scrollability> ] CURSOR
FOR <cursor specification>

<cursor sensitivity>: =
SEMI_SENSITIVE
| INSENSITIVE

<cursor scrollability>: =
SCROLL
| NO SCROLL

<cursor specification>: =
<select statement> [ <updatability clause> ]

<updatability clause>: =
FOR (READ ONLY | UPDATE [ OF <column name list> ])
END-EXEC

**Parameters**
cursor sensitivity
Declares a cursor semi-sensitive or insensitive.
cursor scrollability

Declares a cursor scrollable or non-scrollable.

Note A scrollable cursor does not use fetch loops but rather single fetch calls. Only non-scrollable and forward-only cursors use fetch loops.

Examples

EXEC SQL DECLARE c1 INSENSITIVE SCROLL CURSOR FOR
    select title_id, royalty
    from authors
    where royalty < 25 END-EXEC.
EXEC SQL OPEN c1 END-EXEC.

Usage

- If cursor sensitivity is specified as INSENSITIVE, SCROLL is not implied.
- If cursor sensitivity is not specified as INSENSITIVE or SEMI_SENSITIVE, and SCROLL is also not specified in the declare cursor, the cursor is scrollable and read-only with the specified sensitivity.
- If cursor sensitivity is not specified, the cursor is declared as non-sensitive, non-scrollable and read-only.
- If cursor scrollability is specified as SCROLL, the cursor is INSENSITIVE.
- If cursor scrollability is not specified, the default is NO SCROLL, and the cursor is declared as non-scrollable and read-only.

See also scroll fetch, open

delete (positioned cursor)

Description

Removes, from a table, the row indicated by the current cursor position for an open cursor.

Syntax

exec sql [at connection_name] delete
[from] table_name
where current of cursor_name end-exec

Parameters

table_name

The name of the table from which the row will be deleted.
where current of cursor_name
Causes Adaptive Server Enterprise to delete the row of the table indicated by the current cursor position for the cursor cursor_name.

Examples

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01    PUB-NAME    PIC X(40).
  01    PUB-ID      PIC X(4).
  01    PUB-CTY     PIC X(15).
  01    PUB-ST      PIC X(2).
  01    ANSWER      PIC X(1).
EXEC SQL END DECLARE SECTION END-EXEC.

EXEC SQL DECLARE delcursor CURSOR FOR
  SELECT * FROM publishers END-EXEC.
EXEC SQL OPEN delcursor END-EXEC.
PERFORM FETCH-LOOP UNTIL SQLCODE = 100.
EXEC SQL CLOSE delcursor END-EXEC.
EXEC SQL DEALLOCATE CURSOR delcursor END-EXEC.
EXEC SQL COMMIT WORK END-EXEC.

FETCH-LOOP.
  EXEC SQL FETCH delcursor INTO
    :PUB-ID, :PUB-NAME,
    :PUB-CTY, PUB-ST END-EXEC.
  DISPLAY "PUB ID    ": PUB-ID
  DISPLAY "PUB NAME  ": PUB-NAME
  DISPLAY "PUB CITY  ": PUB-CTY
  DISPLAY "PUB STATE ": PUB-ST
  IF SQLCODE = 100
    DISPLAY "NO MORE RECORDS TO FETCH. END OF PROGRAM RUN."
  ELSE
    DISPLAY "DELETE THIS RECORD ?(Y/N) ": ANSWER
    ACCEPT ANSWER
    IF ANSWER = "Y"
      EXEC SQL DELETE publishers WHERE CURRENT OF
delscursor END-EXEC
  END-IF.

Usage
  • This reference page mainly describes aspects of the Transact-SQL delete statement that differ when used with Embedded SQL. See the Adaptive Server Enterprise Reference Manual.
delete (searched)

- This form of the delete statement must execute on the connection where the cursor cursor_name was opened. If the delete statement includes the at connection_name clause, the clause must match the at connection_name clause of the open cursor statement that opened cursor_name.
- The delete statement fails if the cursor was declared for read only, or if the select statement included an order by clause.

See also close, declare cursor, fetch, open, update

delete (searched)

Description
Removes rows specified by search conditions.

Syntax
exec sql [at connection_name] delete table_name_1
   [from table_name_n [,...]]
   [where search_conditions] end-exec

Parameters
- table_name_1
  The name of the table from which this delete statement deletes rows.
- from table_name_n
  The name of a table to be joined with table_name_1 to determine which rows of table_name_1 will be deleted. The delete statement does not delete rows from table_name_n.
- where search_conditions
  Specifies which rows will be deleted. If you omit the where clause, the delete statement deletes all rows of table_name_1.

Examples
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 AU-FNAME PIC X(30).
01 AU-LNAME PIC X(30).
01 AU-ID PIC X(11).
01 TITLE-ID PIC X(6).
EXEC SQL END DECLARE SECTION END-EXEC.

EXEC SQL WHENEVER SQLERROR PERFORM ROLLBACK-PARA.

EXEC SQL USE pubs2 END-EXEC.
DISPLAY "AUTHOR FIRST NAME ? "
ACCEPT AU-FNAME.
DISPLAY "AUTHOR LAST NAME ? "
ACCEPT AU-LNAME.

EXEC SQL SELECT au_id FROM authors INTO :AU-ID
    WHERE au_fname = :AU-FNAME
    AND au_lname = :AU-LNAME END-EXEC.

EXEC SQL BEGIN TRANSACTION END-EXEC.

* Delete matching records from the 'au_pix' table.
EXEC SQL DELETE au_pix WHERE au_id = :AU-ID END-EXEC.

* Delete matching records from the 'blurbs' table.
EXEC SQL DELETE blurbs WHERE au_id = :AU-ID END-EXEC.

* Delete matching records from the titleauthor table. Since
  we can't have titles associated with this author in other
  related tables, we delete those records too.
EXEC SQL DECLARE selcursor CURSOR FOR
    SELECT title_id FROM titleauthor
    WHERE au_id = :AU-ID END-EXEC.
EXEC SQL OPEN selcursor END-EXEC.
PERFORM FETCH-DEL-LOOP UNTIL SQLCODE = 100.
EXEC SQL CLOSE selcursor END-EXEC.
EXEC SQL DEALLOCATE CURSOR selcursor END-EXEC.

* Delete matching records from the 'authors' table.
EXEC SQL DELETE authors WHERE au_id = :AU-ID END-EXEC.

* Commit all the transactions to the database.
EXEC SQL COMMIT TRANSACTION END-EXEC.

...
* Rollback the transaction in case of errors.
ROLLBACK-PARA.
DISPLAY "ERROR! ROLLING BACK TRANSACTION!"
  DISPLAY "Error code is " SQLCODE.
  DISPLAY "Error message is " SQLERRMC.
EXEC SQL ROLLBACK TRANSACTION END-EXEC.

Usage

- This reference page describes mainly aspects of the Transact-SQL delete statement that differ when used with Embedded SQL. See the *Adaptive Server Enterprise Reference Manual*.
- If you need to remove rows specified by the current position of a cursor pointer, use the delete (positioned cursor) statement.

See also close, declare cursor, fetch, open, update

**describe input (SQL descriptor)**

**Description**
Obtains information about dynamic parameter markers in a prepared dynamic SQL statement and stores that information in a SQL descriptor.

For a list of possible SQL descriptor datatype codes, see Table 9-5 on page 175.

**Syntax**

```sql
exec sql describe input statement_name
using sql descriptor descriptor_name end-exec
```

**Parameters**

- `statement_name` The name of the prepared statement about which you want information. `statement_name` must identify a prepared statement.

- `sql descriptor` Identifies `descriptor_name` as a SQL descriptor.

- `descriptor_name` The name of the SQL descriptor that is to store information about the dynamic parameter markers in the prepared statement.

**Examples**

```sql
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 QUERY PIC X(100).
01 NIN PIC S9(9) COMP.
```
01 COUNTER PIC S9(9) COMP.
01 COLTYPE PIC S9(9) COMP.
01 COLLEN PIC S9(9) COMP.
EXEC SQL END DECLARE SECTION END-EXEC.

... EXEC SQL ALLOCATE DESCRIPTOR din WITH MAX 256 END-EXEC.

DISPLAY "ENTER QUERY :"
ACCEPT QUERY.

EXEC SQL PREPARE dynstmt FROM :QUERY END-EXEC.
EXEC SQL DESCRIBE INPUT dynstmt USING
   SQL DESCRIPTOR din END-EXEC.

EXEC SQL GET DESCRIPTOR din :NIN = COUNT END-EXEC.
MOVE 1 TO COUNTER.
PERFORM GET-DESC-LOOP UNTIL COUNTER > NIN.
EXEC SQL DEALLOCATE PREPARE dynstmt END-EXEC.
EXEC SQL DEALLOCATE DESCRIPTOR din END-EXEC.

... GET-DESC-LOOP.
   EXEC SQL GET DESCRIPTOR din VALUE
      :COUNTER :COLTYPE = TYPE END-EXEC
   EXEC SQL GET DESCRIPTOR din VALUE
      :COUNTER :COLLEN = LENGTH END-EXEC
   DISPLAY "TYPE OF INPUT = ", COLTYPE
   DISPLAY "INPUT LENGTH = ", COLLEN
   ADD 1 TO COUNTER.
END-GET-DESC-LOOP.

Usage

- Information about the statement is written into the descriptor provided in the using clause. Use the get descriptor statement after executing the describe input statement to extract information from the descriptor into host variables.

- The descriptor must be allocated before the describe input statement can be executed.

See also

allocate descriptor, deallocate descriptor, describe output, get descriptor, prepare, set descriptor
**describe input (SQLDA)**

**Description**
Obtains information about dynamic parameter markers in a prepared dynamic SQL statement and stores that information in a SQLDA structure.

**Syntax**

```sql
exec sql describe input statement_name
using descriptor descriptor_name end-exec
```

**Parameters**

- `statement_name`
  - The name of the prepared statement about which you want information. `statement_name` must identify a prepared statement.

- `descriptor`
  - Identifies `descriptor_name` as a SQLDA structure.

- `descriptor_name`
  - The name of the SQLDA structure that is to store information about the dynamic parameter markers in the prepared statement.

**Examples**

```sql
... EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 QUERY PIC X(100).
  EXEC SQL END DECLARE SECTION END-EXEC.
  01 din.
   05 SD-SQLN PIC S9(4) COMP.
   05 SD-SQLD PIC S9(4) COMP.
   05 SD-COLUMN OCCURS 3 TIMES.
    10 SD-DATAPMT.
     15 SQL--NM PIC X(132).
     15 SQL--NMLEN PIC S9(9) COMP.
     15 SQL--DATATYPE PIC S9(9) COMP.
     15 SQL--FORMAT PIC S9(9) COMP.
     15 SQL--MAXLENGTH PIC S9(9) COMP.
     15 SQL--SCALE PIC S9(9) COMP.
     15 SQL--PRECISION PIC S9(9) COMP.
     15 SQL--STTUS PIC S9(9) COMP.
     15 SQL--COUNT PIC S9(9) COMP.
     15 SQL--USERTYPE PIC S9(9) COMP.
     15 SQL--LOCALE PIC S9(9) COMP.
    10 SD-SQLDATA PIC S9(9) COMP.
    10 SD-SQLIND PIC S9(9) COMP.
    10 SD-SQLLEN PIC S9(9) COMP.
    10 SD-SQLMORE PIC S9(9) COMP.
  01 TMP PIC Z(8).```

---

Open Client
DISPLAY "ENTER QUERY :"
ACCEPT QUERY.

EXEC SQL ALLOCATE DESCRIPTOR din WITH MAX 256 END-EXEC.
EXEC SQL PREPARE dynstmt FROM :QUERY END-EXEC.
EXEC SQL DECLAR selcursor CURSOR FOR dynstmt END-EXEC.
EXEC SQL DESCRIBE INPUT dynstmt USING DESCRIPTOR din END-EXEC.

* SD-SQLD contains the number of columns in the query being described
  MOVE SD-SQLD TO TMP.
  DISPLAY "Number of input parameters = ", SD-SQLD.

Usage
Information about the statement is written into the descriptor specified in the
using clause. After the get descriptor statement is executed, you can read the
information out of the SQLDA structure.

See also allocate descriptor, deallocate descriptor, describe output, get descriptor, prepare,
set descriptor

describe output (SQL descriptor)

Description
Obtains row format information about the result set of a prepared dynamic
SQL statement.

For a list of possible SQL descriptor datatype codes, see Table 9-5 on
page 175.

Syntax
exec sql describe [output] statement_name
using sql descriptor descriptor_name end-exec

Parameters
output
An optional keyword that has no effect on the describe output statement but
provides conformance to the SQL standard.

statement_name
The name (specified in a prepare statement) that represents the select
statement to be executed.

sql descriptor
Identifies descriptor_name as a SQL descriptor.
*describe output (SQL descriptor)*

The name of a SQL descriptor that is to store the information returned by the `describe output` statement.

**Examples**

```sql
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 QUERY PIC X(100).
  01 NOUT PIC S9(9) COMP.
  01 DATAVAL PIC X(100).
  01 COUNTER PIC S9(9) COMP.
  01 NUMCOLS PIC S9(9) COMP.
  01 COLNAME PIC X(32).
  01 COLTYPE PIC S9(9) COMP.
  01 COLLEN PIC S9(9) COMP.
EXEC SQL END DECLARE SECTION END-EXEC.

... DISPLAY "ENTER QUERY :"
ACCEPT QUERY.

EXEC SQL ALLOCATE DESCRIPTOR desc_out WITH MAX 256 END-EXEC.
EXEC SQL PREPARE dynstmt FROM :QUERY END-EXEC.
EXEC SQL DECLARE selcursor CURSOR FOR dynstmt END-EXEC.
EXEC SQL OPEN selcursor USING SQL DESCRIPTOR desc_out END-EXEC.
EXEC SQL DESCRIBE OUTPUT dynstmt USING SQL DESCRIPTOR desc_out END-EXEC.

PERFORM FETCH-LOOP UNTIL SQLCODE = 100.
EXEC SQL CLOSE selcursor END-EXEC.
EXEC SQL DEALLOCATE CURSOR selcursor END-EXEC.
EXEC SQL DEALLOCATE PREPARE dynstmt END-EXEC.
EXEC SQL DEALLOCATE DESCRIPTOR desc_out END-EXEC.

... FETCH-LOOP.
  EXEC SQL FETCH selcursor INTO SQL DESCRIPTOR desc_out END-EXEC
  EXEC SQL GET DESCRIPTOR desc_out :NOUT = COUNT END-EXEC
  DISPLAY "COLUMNS RETRIEVED = ", NOUT
  MOVE 1 TO COUNTER
  PERFORM GET-DESC-PARA UNTIL COUNTER > NOUT.
END-FETCH-LOOP.

GET-DESC-PARA.
  EXEC SQL GET DESCRIPTOR desc_out VALUE :COUNTER
    :COLNAME = NAME,
    :COLTYPE = TYPE,
    :COLLEN = LENGTH
  END-EXEC
  DISPLAY "NAME : ", COLNAME
```
DISPLAY "TYPE      ":", COLTYPE
DISPLAY "LENGTH   ":", COLLEN

EXEC SQL GET DESCRIPTOR desc_out VALUE :COUNTER
    :DATAVAL = DATA END-EXEC
DISPLAY "DATA     ":", DATAVAL
DISPLAY " 
    ADD 1 TO COUNTER.
END-GET-DESC-PARA.

Usage
- The information obtained is the type, name, length (or precision and scale, if a number), nullable status, and number of items in the result set.
- The information is about the result columns from the select column list.
- Execute this statement before the prepared statement executes. If you perform a describe output statement after you execute and before you perform a get descriptor, the results will be discarded.

See also allocate descriptor, describe input, execute, get descriptor, prepare

describe output (SQLDA)

Description
Obtains row format information about the result set of a prepared dynamic SQL statement and stores that information in a SQLDA structure.

Syntax
exec sql describe [output] statement_name
    using descriptor sqlda_name end-exec

Parameters
- output
  An optional keyword that has no effect on the describe output statement but provides conformance to the SQL standard.

- statement_name
  The name (specified in a prepare statement) that represents the select statement to be executed.

- descriptor
  Identifies descriptor_name as a SQLDA structure.

- sqlda_name
  The name of a SQLDA structure that is to store the information returned by the describe output statement:
describe output (SQLDA)

Examples

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 QUERY      PIC X(100).
  01 CHARVAR    PIC X(100).
EXEC SQL END DECLARE SECTION END-EXEC.

  01 dout.
    05 SD-SQLN   PIC S9(4) COMP.
    05 SD-SQLD   PIC S9(4) COMP.
    05 SD-COLUMN OCCURS 3 TIMES.
    10 SD-DATAFMT.
      15 SQL--NM    PIC X(132).
      15 SQL--NMLEN PIC S9(9) COMP.
      15 SQL--DATATYPE PIC S9(9) COMP.
      15 SQL--FORMAT PIC S9(9) COMP.
      15 SQL--MAXLENGTH PIC S9(9) COMP.
      15 SQL--SCALE PIC S9(9) COMP.
      15 SQL--PRECISION PIC S9(9) COMP.
      15 SQL--STTUS PIC S9(9) COMP.
      15 SQL--COUNT PIC S9(9) COMP.
      15 SQL--USERTYPE PIC S9(9) COMP.
      15 SQL--LOCALE PIC S9(9) COMP.
    10 SD-SQLDATA PIC S9(9) COMP.
    10 SD-SQLIND PIC S9(9) COMP.
    10 SD-SQLLEN PIC S9(9) COMP.
    10 SD-SQLMORE PIC S9(9) COMP.
  01 TMP       PIC Z(8)9.
  01 COLNUM    PIC S9(9) COMP.
  01 TMP1      PIC S9(9) COMP.
  01 TMP2      PIC S9(9) COMP.
  01 RETCODE   PIC S9(9) COMP.
...

DISPLAY "ENTER QUERY :"
ACCEPT QUERY.

EXEC SQL ALLOCATE DESCRIPTOR dout WITH MAX 256 END-EXEC.
EXEC SQL PREPARE dynstmt FROM :QUERY END-EXEC.
EXEC SQL DECLARE selcursor CURSOR FOR dynstmt END-EXEC.
EXEC SQL OPEN selcursor END-EXEC.
EXEC SQL DESCRIBE OUTPUT dynstmt
    USING DESCRIPTOR dout END-EXEC.

MOVE 1 TO COLNUM.
MOVE 25 TO TMP1.
MOVE 0 TO TMP2.

CALL "SYBSETSQLDA" USING RETCODE dout COLNUM
  CHARVAR SYB-X-PIC TMP1 TMP2 SYB-NO-USAGE
  SYB-NO-SIGN.

EXEC SQL FETCH selcursor INTO DESCRIPTOR dout END-EXEC.
DISPLAY "CHARVAR = ", CHARVAR.

EXEC SQL CLOSE selcursor END-EXEC.
EXEC SQL DEALLOCATE CURSOR selcursor END-EXEC.
EXEC SQL DEALLOCATE PREPARE dynstmt END-EXEC.
EXEC SQL DEALLOCATE DESCRIPTOR dout END-EXEC.

Usage
  • The information obtained is data held in the SQLDA fields, such as the
type, name, length (or precision and scale, if a number), nullable status,
and number of items in the result set.
  • The information is about the result columns from the select column list.

See also describe input, execute, prepare

disconnect

Description Closes one or more connections to a Adaptive Server Enterprise.

Syntax

exec sql disconnect
{connection_name | current | DEFAULT | all} end-exec

Parameters

connection_name
  The name of a connection to be closed.

current
  Specifies that the current connection is to be closed.

DEFAULT
  Specifies that the default connection is to be closed. This keyword must be
  in uppercase letters if you specify the default connection_name using a
  character string variable, for example:

  exec sql disconnect :hv;

all
  Specifies that all active connections be closed.
**disconnect**

**Examples**

```sql
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 SERV-NAME PIC X(25).
  01 USER-NAME PIC X(25).
  01 PASSWORD PIC X(25).
  01 CONN-NAME PIC X(25).
EXEC SQL END DECLARE SECTION END-EXEC.
...
MOVE "sa" TO USER-NAME.
MOVE "" TO PASSWORD.

* Make a default connection.
  EXEC SQL CONNECT :USER-NAME IDENTIFIED BY :PASSWORD END-EXEC.
  EXEC SQL SELECT @@servername into :srvname END-EXEC.
  DISPLAY "NOW CONNECTED TO SERVER ", srvname.

* Accept a server name from the user and make a new connection.
  DISPLAY "SERVER NAME? ".
  ACCEPT SERV-NAME.
  EXEC SQL CONNECT :USER-NAME IDENTIFIED BY :PASSWORD
    At conn2 USING :SERV-NAME END-EXEC.
  EXEC SQL SELECT @@servername into :srvname END-EXEC
  DISPLAY "NOW CONNECTED TO SERVER ", srvname.

* Make a third connection.
  EXEC SQL CONNECT :USER-NAME IDENTIFIED BY :PASSWORD
    At conn3 USING :SERV-NAME END-EXEC.
  EXEC SQL SELECT @@servername into :srvname END-EXEC
  DISPLAY "NOW CONNECTED TO SERVER ", srvname.

* Now set the current connection to DEFAULT.
  EXEC SQL SET CONNECTION DEFAULT END-EXEC.

* Now disconnect the first connection which is the default.
  DISPLAY "DISCONNECTING DEFAULT!".
  EXEC SQL DISCONNECT DEFAULT END-EXEC.

* Now set the current connection to connection2.
  EXEC SQL SET CONNECTION conn2 END-EXEC.

* Now disconnect the third connection.
  DISPLAY "DISCONNECTING THIRD!".
  EXEC SQL DISCONNECT conn3 END-EXEC.
```
* Disconnect remaining connections - case 'conn2' will be closed.
  DISPLAY "DISCONNECTING ALL!".
  EXEC SQL DISCONNECT ALL END-EXEC.

Usage

- By itself, the disconnect keyword is not a valid statement. Instead, it must be followed by connection_name, current, DEFAULT, or all.
- Closing a connection releases all memory and resources associated with that connection.
- disconnect does not commit current transactions; it rolls them back. If an unchained transaction is active on the connection, disconnect rolls it back, ignoring any savepoints.
- Closing a connection closes open cursors, drops temporary Adaptive Server Enterprise objects, releases any locks the connection has in the Adaptive Server Enterprise, and closes the network connection to the Adaptive Server Enterprise.

See also

commit work, commit transaction, connect, rollback transaction, rollback work

exec

Description

Runs a system procedure or a user-defined stored procedure.

Syntax

exec sql [at connection_name]
exec [:status_var = status_value] procedure_name
[([@parameter_name =]param_value [out|put]],...]
[into :hostvar_1 [[:indicator_1]]
, hostvar_n [indicator_n,...]]
[with recompile] end-exec

Note

Do not confuse the exec statement with the Embedded SQL execute statement; they are not related. The Embedded SQL exec statement is, however, the equivalent of the Transact-SQL execute statement.

Parameters

status_var

A host variable to receive the return status of the stored procedure.

status_value

The value of the stored procedure return status variable status_var.

procedure_name

The name of the stored procedure to be executed.
**parameter_name**

The name(s) of the stored procedure’s parameter(s).

**param_value**

A host variable or literal value.

**output**

Indicates that the stored procedure returns a parameter value. The matching parameter in the stored procedure must also have been created using the output keyword.

**into :hostvar_1**

Causes row data returned from the stored procedure to be stored in the specified host variables (hostvar_1 through hostvar_n). Each host variable can have an indicator variable.

**with recompile**

Causes Adaptive Server Enterprise to create a new query plan for this stored procedure each time the procedure executes.

**Examples**

**Example 1**

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 TITLE-ID PIC X(6).
01 TOTAL-DISC PIC S9(9).
01 RET-STATUS PIC S9(9).
EXEC SQL END DECLARE SECTION END-EXEC.
...
EXEC SQL CREATE PROC get_sum_discounts(@title_id tid,
   @discount int output) as
begin
   select @discount = sum (qty*discount)
       from salesdetail
       where title_id = @title_id
end
END-EXEC.

EXEC SQL SET CHAINED ON END-EXEC.
DISPLAY "TITLE ID ? ".
ACCEPT TITLE-ID.

EXEC SQL EXEC :RET-STATUS = get_sum_discounts
   :TITLE-ID, :TOTAL-DISC OUT END-EXEC.

DISPLAY "TOTAL DISCOUNTS FOR TITLE ID ", TITLE-ID," = ",TOTAL-DISC.
...

---

Open Client
Example 2

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
    01 PUB-ID PIC X(4).
    01 NAME PIC X(25).
    01 CITY PIC X(25).
    01 STATE PIC X(2).   
    01 RET-STATUS PIC S9(9).
EXEC SQL END DECLARE SECTION END-EXEC.

EXEC SQL CREATE PROC get_publishers(@pubid char(4)) as
    select pub_name, city, state from publishers where pub_id = @pubid
END-EXEC.

DISPLAY " DETAIL RECORD FOR PUBLISHER ? ".
ACCEPT PUB-ID.

EXEC SQL EXEC :RET-STATUS = get_publishers :PUB-ID
    INTO :NAME, :CITY, :STATE END-EXEC.

IF RET-STATUS = 0
    DISPLAY " PUBLISHER NAME : ", NAME
    DISPLAY " CITY : ", CITY
    DISPLAY " STATE : ", STATE

Usage

• Only one select statement can return rows to the client application.

• If the stored procedure contains select statements that can return row data, you must use one of two methods to store the data. You can either use the into clause of the exec statement or declare a cursor for the procedure. If you use the into clause, the stored procedure must not return more than one row of data, unless the host variables that you specify are arrays.

• The value param_value can be a host variable or literal value. If you use the output keyword, param_value must be a host variable.

• You can specify the output keyword for parameter_name only if that keyword was also used for the corresponding parameter of the create procedure statement that created procedure_name.

• The Embedded SQL exec statement works much like the Transact-SQL execute statement.

See also          declare cursor (stored procedure), select
**exec sql**

Description
Marks the beginning of a SQL statement embedded in a host language program.

Syntax
exec sql [at connection_name] sql_statement end-exec

Parameters
at
Causes the SQL statement sql_statement to execute at the Adaptive Server connection connection_name.

connection_name
The connection name that identifies the Adaptive Server connection where sql_statement is to execute. The connection_name must be defined as a previous connect statement.

sql_statement
A Transact-SQL statement or other Embedded SQL statement.

Examples

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 SITE1 PIC X(25).
  01 SALES1 PIC S9(9) COMP.
EXEC SQL END DECLARE SECTION END-EXEC.

EXEC SQL CONNECT "user" identified by "password"
  AT server1 USING "server1" END-EXEC.
EXEC SQL CONNECT "user" identified by "password"
  AT server2 USING "server2" END-EXEC.

EXEC SQL AT server1 USE pubs2 END-EXEC.
EXEC SQL AT server2 USE pubmast END-EXEC.

EXEC SQL AT server1 SELECT count(*) FROM sales
  INTO :sales1 END-EXEC.

MOVE "server1" TO SITE1.

EXEC SQL SET CONNECTION server2 END-EXEC.
EXEC SQL INSERT numsales VALUES (:SITE1, :SALES1) END-EXEC.
EXEC SQL COMMIT WORK END-EXEC.
```

Usage
- SQL statements embedded in a host language must begin with "exec sql".
- The keywords exec sql can appear anywhere that a host language statement can begin.
• The statement sql_statement can occupy one or more program lines; however, it must conform to host language rules for line breaks and continuation lines.

• The at clause affects only the statement sql_statement. The clause does not affect subsequent SQL statements, and does not reset the current connection.

• The at clause is not valid when sql_statement is one of the following SQL statements:

<table>
<thead>
<tr>
<th>Table 9-2: Statements that cannot use the at clause of exec sql</th>
</tr>
</thead>
<tbody>
<tr>
<td>allocate descriptor</td>
</tr>
<tr>
<td>deallocate descriptor</td>
</tr>
<tr>
<td>exit</td>
</tr>
<tr>
<td>include sqica</td>
</tr>
<tr>
<td>whenever</td>
</tr>
</tbody>
</table>

• connection_name must be defined in a previous connect statement.

• Each Embedded SQL statement must end with a terminator. In COBOL, the terminator is the keyword end-exec.

See also  
begin declare section, connect, disconnect, set connection

**execute**

**Description**  
Executes a dynamic SQL statement from a prepared statement.

See execute immediate on page 145.

**Syntax**  
exec sql [at connection_name] execute statement_name  
[into {host_var_list | descriptor descriptor_name | sql descriptor descriptor_name}]  
[using {host_var_list | descriptor descriptor_name | sql descriptor descriptor_name}] end-exec

**Note**  
Do not confuse the Embedded SQL execute statement with the Embedded SQL exec statement or the Transact-SQL execute statement.
Parameters

**statement_name**
A unique identifier for the statement, defined in a previous `prepare` statement.

**descriptor_name**
Specifies the area of memory, or the SQLDA structure, that describes the statement’s dynamic parameter markers or `select` column list.

**into**
An `into` clause is required when the statement executes a `select` statement, which must be a single-row select. The target of the `into` clause can be a SQL descriptor, a SQLDA structure, or a list of one or more Embedded SQL host variables.

Each host variable in the `host_var_list` must first be defined in a declare section. An `indicator variable` can be associated with a host variable to show when a null data value is retrieved.

**descriptor**
Identifies `descriptor_name` as a SQLDA structure.

**sql descriptor**
Identifies `descriptor_name` as a SQL descriptor.

**using**
The host variables that are substituted for dynamic parameter markers in `host_var_list`. The host variables, which you must define in a declare section, are substituted in the order listed. Use this clause only when `statement_name` contains dynamic parameter markers. The dynamic descriptor can also contain the values for the dynamic parameter markers.

Examples

```sql
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 DEMO-BUF   PIC X(100).
  01 TITLE-ID   PIC X(6).
  01 ORDER-NO   PIC X(20).
  01 QTY        PIC S9(9).
EXEC SQL END DECLARE SECTION END-EXEC.
...
MOVE "INSERT salesdetail(ord_num, title_id, qty) VALUES( :?, :?, :?)"
  -                 TO DEMO-BUF.
EXEC SQL PREPARE ins_stmt FROM :DEMO-BUF END-EXEC.

DISPLAY "RECORDING BOOK SALES".
DISPLAY "ORDER # ? ".
ACCEPT ORDER-NO.
DISPLAY "TITLE ID? ".
```
ACCEPT TITLE-ID.
DISPLAY "QTY SOLD? ".
ACCEPT QTY.

EXEC SQL EXECUTE ins_stmt USING :ORDER-NO, :TITLE-ID, :QTY END-EXEC.

Usage

- `execute` is the second step in method 2 of dynamic SQL. The first step is the `prepare` statement.
- `prepare` and `execute` are valid with any SQL statement except a multirow select statement. For multirow select statements, use either dynamic cursor.
- The statement in `statement_name` can contain dynamic parameter markers ("?"). They mark the positions where host variable values are to be substituted before the statement executes.
- The `execute` keyword distinguishes this statement from `exec`. See the `exec` on page 139 reference page for information on `exec`.

See also
- `declare section`, `get descriptor`, `prepare`, `set descriptor`

**execute immediate**

**Description**

Executes a dynamic SQL statement stored in a character-string host variable or quoted string.

**Syntax**

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 HOST-VAR PIC X(100).
EXEC SQL END DECLARE SECTION END-EXEC.
```

**Parameters**

- `host_variable`
  A character-string host variable defined in a `declare` section. Before calling `execute immediate`, the host variable should contain a complete and syntactically correct Transact-SQL statement.
- `string`
  A quoted literal Transact-SQL statement string that can be used in place of `host_variable`.

**Examples**

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 HOST-VAR PIC X(100).
EXEC SQL END DECLARE SECTION END-EXEC.

... DISPLAY "ENTER A NON-SELECT SQL STATEMENT: ".
ACCEPT HOST-VAR.
```
EXEC SQL EXECUTE IMMEDIATE :HOST-VAR END-EXEC.

Usage

• Using the execute immediate statement is dynamic SQL method 1. See Chapter 7, “Using Dynamic SQL,” for information about the four dynamic SQL methods.

• Except for messages, the statement in host_variable cannot return results to the your program. Thus, the statement cannot be, for example, a select statement.

• The Embedded SQL precompiler does not check the syntax of the statement stored in host_variable before sending it to Adaptive Server Enterprise. If the statement’s syntax is incorrect, Adaptive Server Enterprise returns an error code and message to your program.

• Use prepare and execute (dynamic SQL method 2) to substitute values from host variables into a dynamic SQL statement.

• Use prepare, open, and fetch (dynamic SQL method 3) to execute select statements with dynamic SQL statements that return results.

See also

execute, prepare

exit

Description

Closes Client-Library and deallocates all Embedded SQL resources allocated to your program.

Syntax

exec sql exit end-exec

Examples

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01   HOST-VAR   PIC X(100).
EXEC SQL END DECLARE SECTION END-EXEC.
...
EXEC SQL SELECT getdate() INTO :HOST-VAR END-EXEC.

DISPLAY "THE CURRENT DATE AND TIME IS: ", HOST-VAR.

* Note that the exit statement must be the last embedded SQL statement in the program.

EXEC SQL EXIT END-EXEC.
Usage

- The exit statement closes all connections that your program opened. Also, exit deallocates all Embedded SQL resources and Client-Library resources allocated to your program.
- Although the exit statement is valid on all platforms, it is required only on some. See the Open Client and Open Server Programmers Supplement.
- The exit statement is a Sybase extension; it is not defined in the SQL standard.

See also disconnect

fetch

Description
Copies data values from the current cursor row into host variables or a dynamic descriptor.

Syntax
```
exec sql [at connection_name] fetch [rebind | norebind] cursor_name
into {:host_variable [[indicator]:indicator_variable]}
[:host_variable
 [[indicator]:indicator_variable]]... |
descriptor descriptor_name |
sql descriptor descriptor_name} end-exec
```

Parameters
- rebind | norebind
  Specifies whether host variables require rebinding for this fetch statement. The rebind clause overrides precompiler options that control rebinding.
- cursor_name
  The name of the cursor. The name is defined in a preceding declare cursor statement.
- host_variable
  A host language variable defined in a declare section.
- indicator_variable
  A 2-byte host variable declared in a previous declare section. If the value for the associated variable is null, fetch sets the indicator variable to -1. If truncation occurs, fetch sets the indicator variable to the actual length of the result column. Otherwise, it sets the indicator variable to 0.
**fetch**

**descriptor**
Identifies *descriptor_name* as a SQLDA structure.

**sql descriptor**
Identifies *descriptor_name* as a SQL descriptor.

*descriptor_name*
The name of the dynamic descriptor that is to hold a result set.

**Examples**

```sql
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01    TITLE-ID   PIC X(6).
  01    BOOK-NAME PIC X(80).
  01    BOOK-TYPE PIC X(12).
  01    I-TITLE   PIC S9(9).
  01    I-TYPE    PIC S9(9).
EXEC SQL END DECLARE SECTION END-EXEC.

EXEC SQL DECLARE title_list CURSOR FOR
    SELECT type, title_id, title FROM titles
    ORDER BY type END-EXEC.

EXEC SQL OPEN title_list END-EXEC.
PERFORM FETCH-LOOP UNTIL SQLCODE = 100.
EXEC SQL CLOSE title_list END-EXEC.

FETCH-LOOP.
  EXEC SQL FETCH title_list INTO
      :BOOK-TYPE :I-TYPE,
      :TITLE-ID,
      :BOOK-NAME :I-TITLE END-EXEC
  * Check the indicator value - if not null display the value, else
    * display UNDECIDED.
      IF I-TYPE <> -1
        DISPLAY "TYPE : ", BOOK-TYPE
      ELSE
        DISPLAY "TYPE : UNDECIDED"
      END-IF

      DISPLAY "TITLE ID : ", TITLE-ID
      IF I-TITLE <> -1
        DISPLAY "TITLE : ", BOOK-NAME
      ELSE
        DISPLAY "TITLE : UNDECIDED"
      END-IF.
END-FETCH-LOOP.
```
CHAPTER 9  Embedded SQL Statements: Reference Pages

Usage

- The fetch statement can be used both with static cursors and with cursors in dynamic SQL.
- The open statement must execute before the fetch statement executes.
- The first fetch on an open cursor returns the first row or group of rows from the cursor’s result table. Each subsequent fetch returns the next row or group of rows.
- You can fetch multiple rows into an array.
- The “current row” is the row most recently fetched. To update or delete it, use the where current of cursor_name clause with the update or delete statement. These statements are not valid until after a row has been fetched.
- After all rows have been fetched from the cursor, calling fetch sets SQLCODE to 100. If the select statement furnishes no results on execution, SQLCODE is set to 100 on the first fetch.
- There must be one, and only one, host_variable for each column of the result set.
- When neither the rebind nor the norebind option is specified, the binding behavior is determined by the precompiler option -b. See the Open Client and Open Server Programmers Supplement for details on precompiler options.
- An indicator_variable must be provided for a host_variable that can receive a null value. A runtime error occurs when a null value is fetched for a host variable that has no indicator variable.
- When possible, Client-Library converts the datatype of a result column to the datatype of the corresponding host variable. If Client-Library cannot convert a datatype, it issues an error message. If conversion is not possible, an error occurs.

See also
allocate descriptor, close, declare, delete (positioned cursor), open, prepare, update

scroll fetch

Description
Fetches single or multiple rows from the cursor result set, depending on the ROW_COUNT specification at CURSOR OPEN time.
If a cursor is specified as scrollable, the fetch orientation in the FETCH statement specifies the fetch direction.

If the cursor is not specified as scrollable, FETCH retrieves the next row in the result set.

Syntax

```sql
EXEC SQL FETCH [ <fetch orientation> ]
      [ FROM ] <cursor name>
      [ [ INTO <fetch target list> ] ]
      [ SQL DESCRIPTOR <> ]

<fetch orientation> ::= NEXT | PRIOR | FIRST | LAST | ABSOLUTE <fetch offset> | RELATIVE <fetch offset>

<fetch offset> ::= <signed_numeric_literal>
<fetch target list> ::= <target specification>
                      [ { <comma> <target specification> } ]
END-EXEC
```

Parameters

- **fetch orientation**
  Specified as NEXT, PRIOR, FIRST, LAST, ABSOLUTE, or RELATIVE.

- **fetch offset**
  Specified as an exact, signed numeric value with a scale of zero.

Examples

To fetch a row when a cursor is declared and open:

```sql
EXEC SQL FETCH LAST FROM c1 INTO :title,:roy END-EXEC.
```

To fetch a previous row:

```sql
EXEC SQL FETCH PRIOR FROM c1 INTO :title,:roy END-EXEC.
```

To fetch row 20:

```sql
EXEC SQL FETCH ABSOLUTE 20 FROM c1 INTO :title, :roy END-EXEC.
```

Usage

If fetch orientation is not specified, NEXT is the default.

**Note** If you specify fetch orientation as any type except NEXT on a non-scrollable cursor, you receive the following message:
The fetch type can only be used with scrollable cursors. If fetch orientation positions the cursor beyond the last row or before the first row, sqlca.sqlcode is set to 100, indicating that no rows are found. If an error handler is installed, it may provide additional information.

See also declare, open

get descriptor

Description
Retrieves attribute information about dynamic parameter markers and select column list attributes and data from a SQL descriptor.

For a list of SQL descriptor datatype codes, see Table 9-5 on page 175.

Syntax
exec sql get descriptor descriptor_name
{ :host_variable = count | value item_number :host_variable = item_name
[, :host_variable = item_name]...} end-exec

Parameters

descriptor_name
The name of the SQL descriptor that contains information about the dynamic parameter markers or return columns in a prepared statement.

host_variable
A variable defined in a declare section.

count
The number of dynamic parameters retrieved.

item_number
A number specifying the nth dynamic parameter marker or select column, for which get descriptor is to retrieve information.

item_name
The name of an attribute to be retrieved. See Table 9-3 for details.
**Table 9-3: Valid item_name values**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>data</strong></td>
<td>Value for the dynamic parameter marker or target associated with the specified SQL descriptor. If indicator is negative, this field is undefined.</td>
</tr>
<tr>
<td><strong>indicator</strong></td>
<td>Value for the indicator parameter associated with the dynamic parameter marker or target.</td>
</tr>
<tr>
<td><strong>length</strong></td>
<td>The length, in characters, of the dynamic parameter marker or target for the specified SQL descriptor.</td>
</tr>
<tr>
<td><strong>name</strong></td>
<td>The name of the specified SQL descriptor containing information about the dynamic parameter markers.</td>
</tr>
<tr>
<td><strong>nullable</strong></td>
<td>Equals 0 if the dynamic parameter marker can accept a null value; otherwise, equals 1.</td>
</tr>
<tr>
<td><strong>precision</strong></td>
<td>An integer specifying the total number of digits of precision for the CS_NUMERIC variable.</td>
</tr>
<tr>
<td><strong>returned_length</strong></td>
<td>The length of character types of the values from the select column list.</td>
</tr>
<tr>
<td><strong>scale</strong></td>
<td>An integer specifying the total number of digits after the decimal point for the CS_NUMERIC variable.</td>
</tr>
<tr>
<td><strong>type</strong></td>
<td>The datatype of this column (item number) in the row. For values, see Table 9-5 on page 175.</td>
</tr>
</tbody>
</table>

**Examples**

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 QUERY       PIC X(100).
  01 CHARBUF     PIC X(100).
  01 NUMCOLS     PIC S9(9) COMP.
  01 COLNUM      PIC S9(9) COMP.
  01 COLTYPE     PIC S9(9) COMP.
  01 INTBUF      PIC S9(9).
EXEC SQL END DECLARE SECTION END-EXEC.
...  
DISPLAY "ENTER A SELECT STATEMENT :"  
ACCEPT QUERY.
EXEC SQL ALLOCATE DESCRIPTOR big_desc WITH MAX 256 END-EXEC.
EXEC SQL PREPARE dynstmt FROM :QUERY END-EXEC.
EXEC SQL EXECUTE dynstmt INTO SQL DESCRIPTOR big_desc END-EXEC.
EXEC SQL GET DESCRIPTOR big_desc :NUMCOLS = COUNT END-EXEC.

MOVE 1 TO COLNUM.
```
PERFORM GET-DESC-LOOP UNTIL COLNUM > NUMCOLS.
EXEC SQL DEALLOCATE PREPARE dynstmt END-EXEC.
EXEC SQL DEALLOCATE DESCRIPTOR big_desc END-EXEC.

GET-DESC-LOOP.
EXEC SQL GET DESCRIPTOR big_desc
  VALUE :COLNUM
  :COLTYPE = TYPE END-EXEC
* Check the type data returned and store in appropriate host variables.
  IF COLTYPE = 4
    DISPLAY "INTEGER DATA!"
    EXEC SQL GET DESCRIPTOR big_desc
    VALUE :COLNUM :INTBUF = DATA END-EXEC
  ELSE
    IF COLTYPE = 1
      DISPLAY "CHARACTER DATA!"
      EXEC SQL GET DESCRIPTOR big_desc
      VALUE :COLNUM :CHARBUF = DATA END-EXEC
  ELSE
    * Handle other data types accordingly or store them all as characters.
    ...
    ADD 1 TO COLUMN.
    END-GET-DESC-LOOP.

Usage

- The get descriptor statement returns information about the number or attributes of dynamic parameters specified or the select list columns in a prepared statement.
- This statement should be executed after a describe input, describe output, execute, or fetch (dynamic) statement has been issued.
- It is not possible to retrieve data, indicator, or returned_length until the data associated with the descriptor is retrieved from the server by an execute statement or fetch statement.

See also
describe input, describe output, fetch, set descriptor

get diagnostics

Description
Retrieves error, warning, and informational messages from Client-Library.

Syntax
get diagnostics
  (:hv = statement_info [, :hv = statement_info]...)
  exception :condition_number
include "filename"

```sql
.hv = condition_info [ , .hv = condition_info]...
end-exec
```

**Parameters**

**statement_info**

The keyword number is currently the only supported statement_info type. It returns the total number of exceptions in the diagnostics queue.

**condition_info**

Any one of the keywords sqlca_info, sqlcode_number, and returned_sqlstate.

**Examples**

```sql
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 NUM-MSGS PIC S9(9) COMP.
  01 CONDCNT PIC S9(9) COMP.
EXEC SQL END DECLARE SECTION END-EXEC.
...
EXEC SQL GET DIAGNOSTICS :NUM-MSGS = NUMBER END-EXEC.
MOVE 1 TO CONDCNT.
PERFORM GET-DIAG-PARA UNTIL CONDCNT > NUM-MSGS.
...
GET-DIAG-PARA.
  EXEC SQL GET DIAGNOSTICS EXCEPTION
     :CONDCNT :SQLCA = SQLCA_INFO END-EXEC
  DISPLAY "DIAG. SQLCODE = ",SQLCODE
  DISPLAY "DIAG. MESSAGE = ",SQLERRMC
  ADD 1 TO CONDCNT.
END-GET-DIAG-PARA.
```

**Usage**

- Many Embedded SQL statements are capable of causing multiple warnings or errors. Typically, only the first error is reported using SQLCODE, SQLCA, or SQLSTATE. Use get diagnostics to process all the errors.
- You can use get diagnostics, which is the target of the call, perform, or go to clause of a whenever statement, in the code.
- You can use get diagnostics after a statement for which you want to retrieve informational messages.

**See also**

whenever

**include “filename”**

**Description**

Includes an external file in an Embedded SQL source file.
Syntax
exec sql include "filename" end-exec

Parameters

“filename”
The name of the file to be included in the Embedded SQL source file containing this statement.

Note The maximum supported length for the COPY statement is 70 characters, including the file and pathname.

Examples

Example 1: using COPY
COPY "generic".
...
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01     SRV-NAME   PIC X(80).
  01     UID        PIC X(32).
  01     PASS       PIC X(32).
EXEC SQL END DECLARE SECTION END-EXEC.
...
MOVE USER-NAME TO UID.
MOVE PASSWORD  TO PASS.
EXEC SQL CONNECT :UID IDENTIFIED BY :PASS END-EXEC.
EXEC SQL SELECT @@servername INTO :SRV-NAME END-EXEC.
DISPLAY "CONNECTED TO SERVER ",SRV-NAME.

Copy-file code:

  01 USER-NAME   PIC X(33) VALUE IS "sa".
  01 PASSWORD    PIC X(33) VALUE IS "syb123".

Example 2: using INCLUDE
EXEC SQL INCLUDE "/generic" END-EXEC.

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01     SRV-NAME   PIC X(80).
EXEC SQL END DECLARE SECTION END-EXEC.
EXEC SQL CONNECT :USER-NAME IDENTIFIED BY :PASSWORD END-EXEC.
EXEC SQL SELECT @@servername INTO :SRV-NAME END-EXEC.
DISPLAY "CONNECTED TO SERVER ",SRV-NAME.

Copy-file code:

01 USER-NAME PIC X(33) VALUE IS "sa".
01 PASSWORD PIC X(33) VALUE IS "syb123".

Usage

- The Embedded SQL precompiler processes the included file as though it were part of the Embedded SQL source file, recognizing all declare sections and SQL statements. The Embedded SQL precompiler writes the resulting host language source code into the generated file.
- Use the include path precompiler command line option to specify the directories to be searched for any included files. Refer to the Open Client and Open Server Programmers Supplement.
- Included files can be nested up to a maximum depth of 32 files.
- The include "filename" statement can be used anywhere.

See also declare section

### include sqlca

**Description**

Defines the SQL Communications Area (SQLCA) in an Embedded SQL program.

**Syntax**

```sql
exec sql include sqlca end-exec
```

**Examples**

```sql
EXEC SQL INCLUDE SQLCA END-EXEC.
...
EXEC SQL UPDATE test SET col1 = col1 + 100 END-EXEC.
IF SQLCODE = 0
   DISPLAY "UPDATED ",SQLERRD(3), " ROWS."
ELSE
   IF SQLCODE = 100
      DISPLAY "NO ROWS WERE AFFECTED."
   ELSE
      DISPLAY "AN ERROR OCCURED - ",SQLERRMC.
   END-IF
END-IF.
EXEC SQL COMMIT WORK END-EXEC.
```
Usage

The include sqlda statement can be used anywhere that host language declarations are allowed.

See also

begin declare section

include sqlda

Description

Defines the SQLDA structure in an Embedded SQL program.

Syntax

exec sql include sqlda;

Usage

The include sqlda statement can be used anywhere that host language declarations are allowed.

initialize_application

Description

Generates a call to set the application name on the global CS_CONTEXT handle. If precompiled with the -x option, it will also set the cs_config(CS_SET, CS_EXTERNAL_CONFIG, CS_TRUE) property.

Syntax

exec sql initialize_application

Examples

EXEC SQL INCLUDE SQLCA END-EXEC.

EXEC SQL BEGIN DECLARE SECTION END-EXEC.

 01 SPID       PIC S9(9) COMP.
 01 PROG-NAME  PIC X(33).
 01 UID        PIC X(33).
 01 PASS       PIC X(33).

EXEC SQL END DECLARE SECTION END-EXEC.

PROCEDURE DIVISION.

PO.

* The INITIALIZE_APPLICATION MUST be the FIRST embedded SQL statement * in the program.

EXEC SQL INITIALIZE_APPLICATION APPLICATION_NAME

  = "TEST" END-EXEC.
initialize_application

* The body of the main procedure division goes here including all ESQL statements.

...EXEC SQL CONNECT :UID IDENTIFIED BY :PASS END-EXEC.
EXEC SQL SELECT @@spid INTO :SPID END-EXEC.
EXEC SQL SELECT program_name INTO :PROG-NAME FROM master..sysprocesses
WHERE spid = :SPID END-EXEC.
DISPLAY "THIS APPLICATION'S NAME IN SYSPROCESSES IS ", PROG-NAME.

...EXEC SQL EXIT END-EXEC.

Usage

- **application_name** is either a string literal or a character variable containing the name of the application.

- If `initialize_application` is the first Embedded SQL statement executed by an application, `-x` causes `ct_init` to use external configuration options to initialize the Client-Library part of the CS_CONTEXT structure.

- If `initialize_application` is not the first Embedded SQL statement, `ct_init` does not pick up external configuration options.

- Regardless of whether or not `initialize_application` is the first Embedded SQL statement, `-x` causes `exec sql connect` statements to use external configuration data. If `-e` is also specified, Sybase uses the server name as a key to the configuration data. If `-e` is not specified, then the application name (or DEFAULT) is used as the key to the configuration data.

- If you specify `-x` and the application name, the following applies:
  - `ct_init` uses the application name to determine which section of the external configuration file to use for initialization.
  - The application name is passed to Adaptive Server Enterprise as part of the `connect` statement. The application name is entered in the `sysprocesses.program_name` table.
  - If `-e` is specified without `-x`, then `ct_init` uses external configuration data when initializing, but every connection will use the server name as a key to the external configuration data. See the *Open Client and Open Server Programmers Supplement* for information on command-line options.

See also exit

158 Open Client
open (dynamic cursor)

Description
Opens a previously declared dynamic cursor.

Syntax
exec sql [at connection_name] open cursor_name
[row_count = size] [using (host_var_list |
descriptor descriptor_name |
sql descriptor descriptor_name)] end-exec

Parameters

- cursor_name
  Names a cursor that has been declared using the declare cursor statement.

- size
  The number of rows moved in a network roundtrip, not the number fetched into the host variable. The size argument can be either a literal or a declared host variable.

- host_var_list
  Names the host variables that contain the values for dynamic parameter markers.

- descriptor
  Identifies descriptor_name as a SQLDA structure.

- sql descriptor
  Identifies descriptor_name as a SQL descriptor.

- descriptor_name
  Names the dynamic descriptor that contains information about the dynamic parameter markers in a prepared statement.

Examples

```cobol
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01   DYNABUF   PIC X(200).
  01   TITLE-ID  PIC X(6).
  01   LNAME     PIC X(15).
  01   FNAME     PIC X(15).
  01   PHONE     PIC X(15).
EXEC SQL END DECLARE SECTION END-EXEC.
...
MOVE "SELECT a.au_lname, a.au_fname, a.phone
    FROM authors a, titleauthor t
    WHERE a.au_id = t.au_id
    AND   t.title_id = ? " TO DYNABUF.
EXEC SQL PREPARE dynastmt FROM :DYNABUF END-EXEC.
EXEC SQL DECLARE who_wrote CURSOR FOR dynastmt END-EXEC.
```
open (static cursor)

DISPLAY "LIST AUTHORS FOR WHAT TITLE ? "
ACCEPT TITLE-ID.

EXEC SQL OPEN who_wrote USING :TITLE-ID END-EXEC.
PERFORM FETCH-LOOP UNTIL SQLCODE = 100.
EXEC SQL CLOSE who_wrote END-EXEC.
EXEC SQL DEALLOCATE CURSOR who_wrote END-EXEC.
EXEC SQL DEALLOCATE dynastmt END-EXEC.

... FETCH-LOOP.
EXEC SQL FETCH who_wrote INTO
    :LNAME, :FNAME, :PHONE END-EXEC
DISPLAY "LAST NAME : ", LNAME
DISPLAY "FIRST NAME : ", FNAME
DISPLAY "PHONE : ", PHONE.
END-FETCH-LOOP.

Usage
- open executes the statement specified in the corresponding declare cursor statement. You can then use the fetch statement to retrieve the results of the prepared statement.
- You can have any number of open cursors.
- The using clause substitutes host-variable or dynamic-descriptor contents for the dynamic parameter markers ("?") in the select statement.

See also close, declare, fetch, prepare

open (static cursor)

Description
Opens a previously declared static cursor. This statement can be used to open any static cursor, including one for a stored procedure.

Syntax
```sql
exec sql [at connection_name] open cursor_name
[row_count = size] end-exec
```

Parameters
- `cursor_name`
The name of the cursor to be opened.
- `row_count`
The number of rows moved in a network roundtrip, not the number fetched into the host variable.
**size**

The number of rows that are moved at the same time from Adaptive Server Enterprise to the client. The client buffers the rows until they are fetched by the application. This parameter allows you to tune network efficiency.

**Examples**

```cobol
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 TITLE-ID PIC X(6).
  01 BOOK-NAME PIC X(25).
  01 BOOK-TYPE PIC X(15).
EXEC SQL END DECLARE SECTION END-EXEC.
  01 ANSWER PIC X(1).
...
DISPLAY "TYPE OF BOOKS TO RETRIEVE ? ".
ACCEPT BOOK-TYPE.
EXEC SQL DECLARE titlelist CURSOR FOR
  SELECT title_id, substring(title,1,25) FROM
  titles WHERE type = :BOOK-TYPE END-EXEC.
EXEC SQL OPEN titlelist END-EXEC.
PERFORM FETCH-LOOP UNTIL SQLCODE = 100.
EXEC SQL CLOSE titlelist END-EXEC.
EXEC SQL DEALLOCATE CURSOR titlelist END-EXEC.
EXEC SQL COMMIT WORK END-EXEC.

FETCH-LOOP.
  EXEC SQL FETCH titlelist INTO :TITLE-ID, :BOOK-NAME END-EXEC.
  DISPLAY "TITLE ID : ", TITLE-ID
  DISPLAY "TITLE : ", BOOK-NAME
  DISPLAY "UPDATE/DELETE THIS RECORD ? ".
  ACCEPT ANSWER
  IF ANSWER = "U"
      DISPLAY "ENTER NEW TITLE :
      ACCEPT BOOK-NAME
      EXEC SQL UPDATE titles SET title = :TITLE
      WHERE CURRENT OF titlelist END-EXEC
  ELSE
      IF ANSWER = "D"
          EXEC SQL DELETE titles WHERE CURRENT OF
          titlelist END-EXEC
      END-IF
  END-IF.
END-FETCH-LOOP.
```
open scrollable cursor

Usage

- open executes the select statement given by the declare cursor statement and prepares results for the fetch statement.
- You can have an unlimited number of open cursors.
- A static cursor must be opened only in the file where the cursor is declared. The cursor can be closed in any file.
- The values of host variables embedded in the declare cursor statement are taken at open time.
- When specifying cursor_name, you can use the name of a deallocated static cursor. If you do, the precompiler declares and opens a new cursor having the same name as that of the deallocated cursor. Thus, the precompiler does not reopen the deallocated cursor but instead creates a new one. The results sets for the two cursors can differ.

open scrollable cursor

Description
Opens a previously declared static cursor.

Syntax
EXEC SQL OPEN <cursor_name> [ ROW_COUNT = size ] END-EXEC

Parameters

- size
  Specified as the pre-fetch count. The value is the same as the host array size.
- ROW_COUNT
  Specified only when host arrays are used as host variables.

Usage
The size value is the same as the host array size.

See also
scroll fetch, declare

prepare

Description
Declares a name for a dynamic SQL statement buffer.

Syntax
exec sql [at connection_name] prepare statement_name from
   (:host_variable | "string") end-exec
Parameters

statement_name
An identifier used to reference the statement. The statement_name must uniquely identify the statement buffer and must conform to the SQL identifier rules for naming variables. It can also be a host_variable string containing a valid SQL identifier. statement_name must not be longer than 255 characters.

host_variable
A character-string host variable that contains an executable SQL statement. Place dynamic parameter markers ("?" ) anywhere in the select statement where a host variable value will be substituted.

string
A literal string that can be used in place of host_variable.

Examples

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 DEMO-BUFFER PIC X(120).
01 STATE PIC X(3).
EXEC SQL END DECLARE SECTION END-EXEC.
...
* The 'select into table' statement returns no results
* to the program, so it does not need a cursor.

MOVE "select * into #work from authors where state = ?" TO -
- DEMO-BUFFER.

DISPLAY "STATE ? ".
ACCEPT STATE.

EXEC SQL PREPARE dynstmt FROM :DEMO-BUFFER END-EXEC.
EXEC SQL EXECUTE dynstmt USING :STATE END-EXEC.

EXEC SQL DEALLOCATE PREPARE dynstmt END-EXEC.

Usage

• In the current implementation, Sybase creates a temporary stored procedure for a dynamic SQL statement stored in a character string literal or host variable.

• prepare sends the contents of host_variable to the Adaptive Server Enterprise to convert into a temporary stored procedure. This temporary stored procedure remains in tempdb on Adaptive Server Enterprise until the statement is deallocated or the connection is disconnected.
rollback

- The scope of statement_name is global to your program but local to the connection connection_name. The statement persists until the program either deallocates it or closes the connection.
- prepare is valid with Dynamic SQL methods 2, 3, and 4.
- With method 2, (prepare and execute), an execute statement substitutes values from host variables, if any, into the prepared statement and sends the completed statement to Adaptive Server Enterprise. If there are no host variables to substitute and no results, you can use execute immediate, instead.
- With method 3, prepare and fetch, a declare cursor statement associates the saved select statement with a cursor. An open statement substitutes values from host variables, if any, into the select statement and sends the result to Adaptive Server Enterprise for execution.
- With methods 2, 3, and 4, prepare and fetch with parameter descriptors, the dynamic parameter descriptors, represented by question marks ("?"), indicate where host variables will be substituted.
- A prepared statement must be executed on the same connection on which it was prepared. If the prepared statement is used to declare a cursor, all operations on that cursor use the same connection as the prepared statement.
- The statement in host_variable can contain dynamic parameter markers that indicate where to substitute values of host variables into the statement.

See also
declare cursor, execute, execute immediate, deallocate prepare

rollback

Description
Rolls a transaction back to a savepoint inside the transaction or to the beginning of the transaction.

Syntax
exec sql [at connection_name]
rollback [transaction | tran | work]
[transaction_name | savepoint_name] end-exec

Parameters
transaction | trans | work
The keywords transaction, trans, and work are interchangeable in the rollback statement, but only work is ANSI-compliant.
transaction_name
The name of the transaction being rolled back.

savepoint_name
The name assigned to the savepoint in a save transaction statement. If you omit savepoint_name, Adaptive Server rolls back the entire transaction.

Examples

EXEC SQL CONNECT "user" IDENTIFIED BY "password"
   AT connect1 USING "srvname" END-EXEC.

EXEC SQL AT connect1 UPDATE test SET col1 = 'x' END-EXEC.
IF SQLCODE = 0
DISPLAY "ROWS UPDATED = ",SQLERRD(3)
ELSE
   DISPLAY "AN ERROR OCCURED = ",SQLERRMC
ESQL SQL AT connect1 ROLLBACK TRANSACTION END-EXEC
END-IF.

Usage
• This reference page mainly describes aspects of the Transact-SQL rollback statement that differ when used with Embedded SQL. See the Adaptive Server Enterprise Reference Manual.
• Transaction names and savepoint names must conform to the Transact-SQL rules for identifiers.
• Transaction names and savepoints are Transact-SQL extensions; they are not ANSI-compliant. Do not use a transaction name or savepoint name with the ANSI-compliant keyword work.

See also
begin transaction, commit

select

Description
Retrieves rows from database objects.

Syntax
exec sql [at connect_name]
    select select_list
    into destination
    from table_name... end-exec

Parameters
select_list
Same as select_list in the Transact-SQL select statement, except that select_list cannot perform variable assignments in Embedded SQL.
select

destination
A table or a series of one or more Embedded SQL host variables. Each host variable must first be defined in a previous declare section. Indicator variables can be associated with the host variables.

Examples

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01   LNAME    PIC X(25).
01   FNAME    PIC X(25).
01   PHONE    PIC X(15).
01   AU-ID    PIC X(12).
EXEC SQL END DECLARE SECTION END-EXEC.

... 

DISPLAY "AUTHOR ID ? ".
ACCEPT AU-ID.

EXEC SQL SELECT au_lname, au_fname, phone
    INTO :LNAME, :FNAME, :PHONE
    FROM authors
    WHERE au_id = :AU-ID END-EXEC.

IF SQLCODE = 100
DISPLAY "COULD NOT LOCATE AUTHOR ",AU-ID
ELSE
    DISPLAY "DETAIL RECORD FOR AUTHOR: ", AU-ID
    DISPLAY "NAME ":", LNAME, " ", FNAME
    DISPLAY "PHONE ":", PHONE
END-IF.

Usage

- This reference page mainly describes aspects of the Transact-SQL select statement that differ when the statement is used in Embedded SQL. See the Adaptive Server Enterprise Reference Manual.

- The compute clause of the Transact-SQL select statement cannot be used in Embedded SQL programs.

- Host variables in a select statement are input variables only, except in the statement's into clause. Host variables in the into clause are output variables.

- Previously declared input host variables can be used anywhere in a select statement that a literal value or Transact-SQL variable is allowed. Indicator variables can be associated with input host variables to specify null values.
If a select statement returns more than one row, each host variable in the statement’s into clause must be an array with enough space for all the rows. Otherwise, you must use a cursor to bring the rows back one at a time.

See also declare cursor

**set connection**

*Description*
Causes the specified existing connection to become the current connection.

*Syntax*

```
set connection {connection_name | DEFAULT} end-exec
```

*Parameters*

- `connection_name`
  The name of an existing connection that you want to become the current connection.
- `default`
  Specifies that the unnamed default connection is to become the current connection.

*Examples*

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 MYID PIC X(33).
EXEC SQL END DECLARE SECTION END-EXEC.

EXEC SQL CONNECT "user1" AT connect1 USING "SERVER1" END-EXEC.
EXEC SQL CONNECT "user2" AT connect2 USING "SERVER2" END-EXEC.

* The next statement executes on connect2, because that was the last connection made.

EXEC SQL SELECT user_name() INTO :MYID END-EXEC.
DISPLAY "The user connected to SERVER2 is: ",MYID.

* Explicitly set the connection to now use to connect1.

EXEC SQL SET CONNECTION connect1 END-EXEC.

* The following statement will execute on connect1.

EXEC SQL SELECT user_name() INTO :MYID END-EXEC.
DISPLAY "The user connected to SERVER1 is: ",MYID.
```
set descriptor

Usage

- The set connection statement specifies the current connection for all subsequent SQL statements, except those preceded by the `exec sql` clause at.
- A set connection statement remains in effect until you choose a different current connection by using the set connection statement again.

See also at connection_name, connect

set descriptor

Description

 Inserts or updates data in a SQL descriptor.

For a list of possible SQL descriptor datatypes, see Table 9-5 on page 175.

Syntax

```
exec sql set descriptor descriptor_name
count = host_variable] |
{value item_number (item_name =
:host_variable)][,...] end-exec
```

Parameters

- `descriptor_name`
  The name of the SQL descriptor that contains information about the dynamic parameter markers in a prepared statement.

- `count`
  The number of dynamic parameter specifications to be described.

- `host_variable`
  A host variable defined in a declare section.

- `item_number`
  Represents the nth occurrence of either a dynamic parameter marker or a select column.

- `item_name`
  Represents the attribute information of either a dynamic parameter marker or a select list column. Table 9-4 lists the values for `item_name`. 
Table 9-4: Values for item_name

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>Value for the dynamic parameter marker or target associated with the specified SQL descriptor. If indicator is negative, this field is undefined.</td>
</tr>
<tr>
<td>length</td>
<td>The length, in characters, of the dynamic parameter marker of target for the specified SQL descriptor.</td>
</tr>
<tr>
<td>precision</td>
<td>An integer specifying the total number of digits of precision for the CS_NUMERIC variable.</td>
</tr>
<tr>
<td>scale</td>
<td>An integer specifying the total number of digits after the decimal point for the CS_NUMERIC variable.</td>
</tr>
<tr>
<td>type</td>
<td>The datatype of this column (item number) in the row. For values, see Table 9-5 on page 175.</td>
</tr>
</tbody>
</table>

Examples

```cobol
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 TITLE-ID PIC X(6).
01 SALES1 PIC S9(9).
01 SALES2 PIC S9(9).
01 ROYALTY PIC S9(9) COMP.
EXEC SQL END DECLARE SECTION END-EXEC.
...
EXEC SQL ALLOCATE DESCRIPTOR roy_desc WITH MAX 3 END-EXEC.
EXEC SQL PREPARE getroylty FROM "SELECT royalty FROM roysched
WHERE title_id = ? and lorange <= ? AND hirange > ?"
END-EXEC.
MOVE "BU1032" TO TITLE-ID.
MOVE 1000 TO SALES1.
MOVE 10 TO SALES2.
EXEC SQL SET DESCRIPTOR roy_desc VALUE 1 DATA = :TITLE-ID END-EXEC.
EXEC SQL SET DESCRIPTOR roy_desc VALUE 2 DATA = :SALES1 END-EXEC.
EXEC SQL SET DESCRIPTOR roy_desc VALUE 3 DATA = :SALES2 END-EXEC.
EXEC SQL EXECUTE getroylty INTO :ROYALTY USING SQL DESCRIPTOR roy_desc END-EXEC.
DISPLAY "ROYALTY = ", ROYALTY.
```

Usage

An Embedded SQL program passes attribute and value information to Client-Library, which holds the data in the specified SQL descriptor until the program issues it a request to execute a statement.
**update**

Description
Modifies data in rows of a table.

Syntax
```
exec sql [at connection_name] update table_name
set [table_name]
column_name1 = {expression1
| NULL | (select_statement)}
[, column_name2 =
{expression2} | NULL
| (select_statement)]...
[from table_name
[, table_name]...
[where {search_conditions | current of cursor_name}]
end-exec
```

Parameters
- `table_name` The name of a table or view, specified in any format that is valid for the update statement in Transact-SQL.

Examples
```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01  STORE-NAME   PIC X(40).
  01  DISC-TYPE   PIC X(40).
  01  LOWQTY      PIC S9(9) COMP.
  01  HIGHQTY     PIC S9(9) COMP.
  01  DISCOUNT    PIC S9(9) COMP.
EXEC SQL END DECLARE SECTION END-EXEC.
...
EXEC SQL DECLARE upd_cursor CURSOR FOR
SELECT s.stor_name, d.discounttype, d.lowqty,
      d.highqty, d.discount
FROM stores s, discounts d
WHERE s.stor_id = d.stor_id END-EXEC.
EXEC SQL OPEN upd_cursor END-EXEC.
PERFORM FETCH-LOOP UNTIL SQLCODE = 100.
EXEC SQL CLOSE upd_cursor END-EXEC.
EXEC SQL DEALLOCATE CURSOR upd_cursor END-EXEC.
EXEC SQL COMMIT WORK END-EXEC.
...
```
FETCH-LOOP.
EXEC SQL FETCH upd_cursor INTO :STORE-NAME, :DISC-TYPE,:LOWQTY ,
:HIGHQTY,:DISCOUNT END-EXEC.
IF SQLCODE = 100
    DISPLAY "NO MORE RECORDS TO FETCH. END OF PROGRAM RUN."
ELSE
    DISPLAY "NEW DISCOUNT : ">
    ACCEPT DISCOUNT
    EXEC SQL UPDATE discounts
        SET discount = :DISCOUNT
        WHERE CURRENT OF upd_cursor END-EXEC
    END-IF.
END-FETCH-LOOP.

Usage

- This reference page mainly describes aspects of the Transact-SQL update statement that differ when the statement is used in Embedded SQL. See the Adaptive Server Enterprise Reference Manual.
- Host variables can appear anywhere in an expression or in any where clause.
- You can use the where clause to update selected rows in a table. Omit the where clause to update all rows in the table. Use where current of cursor_name to update the current row of an open cursor.
- When where current of cursor_name is specified, the statement must be executed on the connection specified in the open cursor statement. If the at connection_name clause is used, it must match the open cursor statement.

See also close, delete cursor, fetch, open, prepare

whenever

Description

Specifies an action to occur whenever an executable SQL statement causes a specified condition.

Syntax

exec sql whenever {sqlerror | not found | sqlwarning}
{continue | go to label | goto label | stop | call routine_name [args]} end-exec

Parameters

sqlerror

Specifies an action to take when an error is detected, such as a syntax error returned to the Embedded SQL program from Adaptive Server.
whenever

  not found
  Specifies an action to take when a fetch or select into statement retrieves no
data or when a searched update or delete statement affects no rows.

sqlwarning
  Specifies an action to take when a warning is received; for example, when a character string is truncated.

continue
  Take no action when the condition occurs.

go to | goto
  Transfer control to the program statement at the specified label.

label
  A host language statement label, such as a C label.

stop
  Terminate the Embedded SQL program when the condition occurs.

call
  Transfer control to a callable routine in the program, such as a user-defined function or subroutine.

routine_name
  A host language routine that can be called. The routine must be able to be called from the source file that contains the whenever statement. You may need to declare the routine as external to compile the Embedded SQL program.

args
  One or more arguments to be passed to the callable routine, using the parameter-passing conventions of the host language. The arguments can be any list of host variables, literals, or expressions that the host language allows. A space character should separate each argument from the next.

Examples

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01    LNAME    PIC X(15).
  01    FNAME    PIC X(15).
  01    PHONE    PIC X(15).
EXEC SQL END DECLARE SECTION END-EXEC.

EXEC SQL WHENEVER SQLERROR PERFORM ERR-PARA END-EXEC.
EXEC SQL WHENEVER SQLWARNING PERFORM WARN-PARA END-EXEC.
* If there are no more records to process from the fetch, stop the * program.
EXEC SQL WHENEVER NOT FOUND STOP END-EXEC.

EXEC SQL DECLARE au_list CURSOR FOR
    SELECT au_lname, au_fname, phone
    FROM authors
    ORDER BY au_lname END-EXEC.

EXEC SQL OPEN au_list END-EXEC.

PERFORM FETCH-LOOP UNTIL SQLCODE = 100 END-EXEC.
EXEC SQL CLOSE au_list END-EXEC.

FETCH-LOOP.
    EXEC SQL FETCH au_list INTO :
        LNAME, :FNAME, :PHONE END-EXEC
    DISPLAY "LAST NAME : ", LNAME
    DISPLAY "FIRST NAME : ", FNAME
    DISPLAY "PHONE : ", PHONE
END-FETCH-LOOP.

WARN-PARA.
    DISPLAY "Warning code is " SQLCODE.
    DISPLAY "Warning message is " SQLERRMC.

    WARN-PARA-END.
    EXIT.

ERR-PARA.
    * print the error code, the error message and the line number of
    * the command that caused the error.
    *
    DISPLAY "Error code is " SQLCODE.
    DISPLAY "Error message is " SQLERRMC.

EXIT.

Usage
- The whenever statement causes the Embedded SQL precompiler to
generate code following each executable SQL statement. The generated
code includes the test for the condition and the host language statement or
statements that carry out the specified action.
whenever

- The Embedded SQL precompiler generates code for the SQL statements that follow the `whenever` statement in the source file, including SQL statements in subroutines that are defined in the same source file.

- Use `whenever...continue` to cancel a previous `whenever` statement. The `continue` action causes the Embedded SQL precompiler to ignore the condition. To prevent infinite loops, use `whenever...continue` in an error handler before executing any Embedded SQL statements.

- When you use `whenever...go to label`, `label` must represent a valid location to resume execution. In C, for example, `label` must be declared in any routine that has executable SQL statements within the scope of the `whenever` statement. C does not allow a `goto` statement to jump to a label declared in another function.

- If you have a `whenever` statement in your program but you have not declared SQLCA or SQLSTATE status variables, the Embedded SQL precompiler assumes that you are using the SQLCODE variable. Be sure that SQLCODE is declared. Otherwise, the generated code will not compile.

**SQL descriptor codes**

Table 9-5 pertains to the SQL descriptor used for dynamic SQL statements. Sybase’s use of dynamic SQL values conforms to the ANSI/ISO 185-92 SQL-92 standards. See the appropriate ANSI/ISO documentation.
Table 9-5: SQL descriptor datatype codes

<table>
<thead>
<tr>
<th>ANSI SQL datatype</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit</td>
<td>14</td>
</tr>
<tr>
<td>character</td>
<td>1</td>
</tr>
<tr>
<td>character varying</td>
<td>12</td>
</tr>
<tr>
<td>date, time</td>
<td>9</td>
</tr>
<tr>
<td>decimal</td>
<td>3</td>
</tr>
<tr>
<td>double precision</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>6</td>
</tr>
<tr>
<td>integer</td>
<td>4</td>
</tr>
<tr>
<td>numeric</td>
<td>2</td>
</tr>
<tr>
<td>real</td>
<td>7</td>
</tr>
<tr>
<td>smallint</td>
<td>5</td>
</tr>
</tbody>
</table>

Sybase-defined datatype Client-Library code

<table>
<thead>
<tr>
<th>Smalldatetime</th>
<th>-9</th>
</tr>
</thead>
<tbody>
<tr>
<td>money</td>
<td>-10</td>
</tr>
<tr>
<td>smallmoney</td>
<td>-11</td>
</tr>
<tr>
<td>text</td>
<td>-3</td>
</tr>
<tr>
<td>image</td>
<td>-4</td>
</tr>
<tr>
<td>tinyint</td>
<td>-8</td>
</tr>
<tr>
<td>binary</td>
<td>-5</td>
</tr>
<tr>
<td>varbinary</td>
<td>-6</td>
</tr>
<tr>
<td>long binary</td>
<td>-7</td>
</tr>
<tr>
<td>longchar</td>
<td>-2</td>
</tr>
</tbody>
</table>

Table 9-6: SQL descriptor identifier values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>The datatype of this column (item number) in the row. For values, see Table 9-5 on page 175.</td>
</tr>
<tr>
<td>length</td>
<td>The length, in characters, of the dynamic parameter marker of target for the specified SQL descriptor.</td>
</tr>
<tr>
<td>returned_length</td>
<td>The length of char types of the values from the select column list.</td>
</tr>
<tr>
<td>precision</td>
<td>An integer specifying the total number of digits of precision for the CS_NUMERIC variable.</td>
</tr>
<tr>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td>scale</td>
<td>An integer specifying the total number of digits after the decimal point for the CS_NUMERIC variable.</td>
</tr>
<tr>
<td>nullable</td>
<td>Equals 0 if the dynamic parameter marker can accept a null value; otherwise, equals 1.</td>
</tr>
<tr>
<td>indicator</td>
<td>Value for the indicator parameter associated with the dynamic parameter marker or target.</td>
</tr>
<tr>
<td>data</td>
<td>Value for the dynamic parameter marker or target associated with the specified SQL descriptor. If indicator is negative, this field is undefined.</td>
</tr>
<tr>
<td>name</td>
<td>The name of the specified SQL descriptor containing information about the dynamic parameter markers.</td>
</tr>
</tbody>
</table>
Open Client/Server Configuration File

Open Client/Server applications can easily be configured using the Open Client/Server configuration file. By default, the file is named `ocs.cfg` and is located in the `$SYBASE/$SYBASE_OCS/config` directory for UNIX and `%SYBASE%\%SYBASE_OCS%\ini` directory for Microsoft Windows.

### Purpose of the Open Client/Server configuration file

The Open Client/Server configuration file provides a single location where all Open Client/Server application connections can be configured. Using the configuration file simplifies the tasks of establishing configuration standards and managing configuration changes.

### Accessing the configuration functionality

This feature is available through two new command-line options of the `initialize_application` statement:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose of the Open Client/Server configuration file</td>
<td>177</td>
</tr>
<tr>
<td>Accessing the configuration functionality</td>
<td>177</td>
</tr>
<tr>
<td>Default settings</td>
<td>178</td>
</tr>
<tr>
<td>Syntax for the Open Client/Server configuration file</td>
<td>179</td>
</tr>
<tr>
<td>Sample programs</td>
<td>181</td>
</tr>
</tbody>
</table>
Default settings

- `-x` – this option allows for external configuration. The application needs to initialize an application with a name. The Open Client/Server configuration file will have a section with this application name. Under this section, place all properties that need to be set for this application. The `-x` option is useful only when used with `initialize_application`. If initializing is not done, and the `-x` option is used, the default section of the configuration file will be accessed.

- `-e` – this option allows us to configure by SERVER NAME. No call to `initialize_application` is required. The server name will be used as a key to look up in the configuration file for properties to be set the section defined by the server name. This allows users to associate connection names with specific connection properties.

**Note** If `initialize_application` is not the first Embedded SQL statement to be executed, external configuration properties will not be set. If it is the first Embedded SQL statement to be executed, then the external configuration options will be used for initialization.

---

**Default settings**

The following is the Open Client/Server configuration file with default settings. You can customize the file as needed.

```
[DEFAULT]

;This is the default section loaded by applications that use the external configuration feature, but which do not specify their own application name. Initially this section is empty. Defaults from all properties will be the same as earlier versions of Open Client libraries.

[ANSI_ESQL]

;This section defines configuration which an ANSI conforming Embedded SQL application should use to get ANSI-defined behavior from Adaptive Server Enterprises and Open Client libraries. This set
```
of ;configuration ;properties matches the set which earlier ;versions of Embedded SQL (version 10.0.x) automatically set for ;applications during execution of a CONNECT statement.

CS_CAP_RESPONSE=CS_RES_NOSTRIPBLANKS
CS_EXTRA_INF=CS_TRUE
CS_ANSI_BINDS=CS_TRUE
CS_OPT_ANSINULL=CS_TRUE
CS_OPT_ANSIPERM=CS_TRUE
CS_OPT_STR_RTRUNC=CS_TRUE
CS_OPT_ARITHABORT=CS_FALSE
CS_OPT_TRUNCIGNORE=CS_TRUE
CS_OPT_ISOLATION=CS_OPT_LEVEL3
CS_OPTCHAINXACTS=CS_TRUE
CS_OPT_TIMEOUTXACT=CS_TRUE
CS_OPT_QUOTED_IDENT=CS_TRUE
;End of default sections

Syntax for the Open Client/Server configuration file

The syntax for the Open Client/Server configuration file matches the existing syntax for Sybase localization and configuration files supported by CS-Library with minor variations.

Syntax

• ; – Signifies a comment line.
• [section_name] – Section names are wrapped in square brackets. The Open Client/Server configuration file comes with sections named DEFAULT and ANSI_ESQL. The application name will be used as the section name for an application that has been compiled with the -x option. For an application that has been compiled with the -e option, the server name will be used for the section name. Any name can be used as a section name for those sections that contain settings that will be used in multiple sections. The following example shows a section arbitrarily named GENERIC, and how that section is included in other sections:
Syntax for the Open Client/Server configuration file

[GENERIC]
CS_OPT_ANSINULL=CS_TRUE

[APP_PAYROLL]
include=GENERIC
CS_CAP_RESPONSE=CS_RES_NOSTRIPBLANKS

[APP_HR]
include=GENERIC
CS_OPT_QUOTED_IDENT=CS_TRUE

- entry_name=entry_value
  - Entry values can be anything: integers, strings, and so on. If an entry value line ends with \n, the entry value continues to the next line.
  - White spaces are trimmed from the beginning and end of entry values.
  - If white spaces are required at the beginning or end of an entry value, wrap them in double quotes.
  - An entry that begins with a double quote must end with a double quote. Two double quote characters in a row within a quoted string represent a single double quote in the value string. If a newline is encountered within double quotes, it is considered to be literally part of the value.
  - Entry names and section names can consist of alphabetic characters (both uppercase and lowercase), the digits 0 - 9, and any of the following punctuation characters: ! " # $ % & ' ( ) * + , - . / : ; < > ? @ \ ^ _ ` { | } ~.
    Square brackets ([ ]), space, and equal sign (=) are not supported. The first letter MUST be alphabetic.
  - Entry and section names are case sensitive.
  - Include=earlier_section
    If a section contains the entry include, then the entire contents of that previously defined section are considered to be replicated within this section. In other words, the properties defined in the previous section are inherited by this section.
Note that the included section must have been defined before being included in another section. This allows the configuration file parsing to happen in a single pass and eliminates the need to detect recursive included directives.

If an included section in turn includes another section, the order of entry values is defined by a “depthfirst” search of the included sections.

Sections cannot include a reference to themselves. In other words, recursion is not possible because you must include a previously defined section—you cannot include the section being defined.

All direct entry values defined in a given section supersede any values that may have been included from another section. In the following example, CS_OPT_ANSINULL will be set to false in the APP.PAYROLL application. Note that the position of the include statement does not affect this rule.

```
[GENERIC]
  CS_OPT_ANSINULL=CS_TRUE

[APP_PAYROLL]
  CS_OPT_ANSINULL=CS_FALSE
  include=GENERIC
```

Sample programs

Consider the following scenario: An Embedded SQL program defines a cursor to retrieve rows from the titles table in the pubs2 database. The WHERE clause uses non-ANSI standard NULL checking. To clarify, IS NULL and IS NOT NULL are ANSI standards which is the default used by Embedded SQL programs. However, an Embedded SQL program wishing to use = NULL or != NULL will need to turn OFF ANSINULL behavior and use Transact-SQL syntax instead. If you wanted to make comparisons with NULLs in Transact-SQL syntax in Embedded SQL prior to version 11.1, you would need to make the following call:

```sql
EXEC SQL set ansinull off END-EXEC.
```
In the following example, no change is made to the Embedded SQL code, but the desired behavior is attained by setting appropriate properties in the Open Client/Server configuration file.

There are two versions of the same program listed below. One is to be used with the -e option and the other with the -x option.

**Embedded SQL/COBOL sample programs**

Perform the following before you use the sample programs:

- On IBM, set the SYBPLATFORM environment variable to “rs6000” for the Embedded SQL/COBOL makefile, provided to build sample programs.
- On Sun Solaris, set the SYBPLATFORM environment variable to “sun_svr4” for the Embedded SQL/COBOL makefile, provided to build sample programs.
- On HP, set the SYBPLATFORM environment variable to “hpux” for the Embedded SQL/COBOL makefile, provided to build sample programs.
- On HP Itanium, set the SYBPLATFORM environment variable to “hpia” for the Embedded SQL/COBOL makefile, provided to build sample programs.
- On Linux, set the SYBPLATFORM environment variable to “linux” for the Embedded SQL/COBOL makefile, provided to build sample programs.

**Embedded SQL program version for use with the -x option**

* ocs_ex.pco

* Description :
* This program declares a cursor which retrieves rows from
* the 'titles' table based on condition checking for NULLS
* in the NON-ANSI style (CS_OPT_ANSINULL = CS_FALSE).
* The program will be compiled using the -x option which will
* use an external configuration file (ocs.cfg) based on the
* name of the application. The name of the application is
* defined at the time of INITIALIZING the application.
*
EXEC SQL INCLUDE SQLCA END-EXEC.

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
...
01 TITLE-ID PIC X(6).
01 PRICE PIC X(30).
EXEC SQL END DECLARE SECTION END-EXEC.
...

EXEC SQL INITIALIZE_APPLICATION APPLICATION_NAME = "TEST1" END-EXEC.

EXEC SQL CONNECT :UID IDENTIFIED BY :PASS END-EXEC.
EXEC SQL USE pubs2 END-EXEC.

* Declare and open the cursor for select
EXEC SQL DECLARE title_list CURSOR FOR
SELECT title_id, price FROM titles
WHERE price != NULL END-EXEC.

EXEC SQL OPEN title_list END-EXEC.

* Fetch the data into host variables.
PERFORM FETCH-LOOP UNTIL SQLCODE = 100.
...

EXEC SQL CLOSE title_list END-EXEC.
EXEC SQL DEALLOCATE CURSOR title_list END-EXEC.

STOP RUN.

FETCH-LOOP.

EXEC SQL FETCH title_list INTO
:TITLE-ID,
:PRICE END-EXEC.
**Sample programs**

...  

END-IF.

---

**Note** Set the precompiler option in the makefile: `cobpre -x`.

---

The following is a sample configuration file for the preceding program:

```
[DEFAULT]

[Test1]
; This is name of the application set by INITIALIZE_APPLICATION. ; Therefore this is the section that will be referred to a runtime.

CS_OPT_ANSINULL=CS_FALSE

; The above option will enable comparisons of nulls in the NON-ANSI style.
```

---

**Same Embedded SQL program with the -e option**

* Program name: ocs_test.cp
* Description : This program declares a cursor that retrieves rows from the 'titles' table based on condition checking for NULLS in the NON-ANSI style.
* The program will be compiled using the -e option, which will use the server name that the application connects to, as the corresponding section to look up in the configuration file.

```
EXEC SQL INCLUDE SQLCA END-EXEC.

EXEC SQL BEGIN DECLARE SECTION END-EXEC.

...  

01 TITLE-ID PIC X(6).
01 PRICE PIC X(30).

EXEC SQL END DECLARE SECTION END-EXEC.

...  

EXEC SQL CONNECT :UID IDENTIFIED BY :PASS END-EXEC.
EXEC SQL USE pubs2 END-EXEC.
* Declare and open the cursor for select
  EXEC SQL DECLARE title_list CURSOR FOR
  SELECT title_id, price FROM titles
  WHERE price != NULL END-EXEC.

  EXEC SQL OPEN title_list END-EXEC.

* Fetch the data into host variables.
  PERFORM FETCH-LOOP UNTIL SQLCODE = 100.

  ...

  EXEC SQL CLOSE title_list END-EXEC.
  EXEC SQL DEALLOCATE CURSOR title_list END-EXEC.

  STOP RUN.

FETCH-LOOP.

  EXEC SQL FETCH title_list INTO
  :TITLE-ID,
  :PRICE END-EXEC.

  ...

  END-IF.

---

**Note** Precompiler option to set in the makefile: **cobpre -e**.

The following is a sample configuration file for the preceding program:

```
[DEFAULT]

[SYBASE]
;This is name of the server that the application connect to. Therefore
;this is the section that will be referred to a runtime.
;
CS_OPT_ANSIUNULL-CS_FALSE
;The above option will enable comparisons of nulls in the NON-ANSI
;style.
```

The above configuration files have been vastly simplified. A typical Open
Client/Server configuration file would be in the following format:
Sample programs

[DEFAULT]
;
[ANSI_ESQL]
CS_CAP_RESPONSE=CS_RES_NOSTRIPOBLANKS
CS_EXTRA_INF=CS_TRUE
CS_ANSI_BINDS=CS_TRUE
CS_OPT_ANSINULL=CS_TRUE
CS_OPT_ANSIPERM=CS_TRUE
CS_OPT_STR_RTRUNC=CS_TRUE
CS_OPT_ARITHABORT=CS_FALSE
CS_OPT_TRUNCIGNORE=CS_TRUE
CS_OPT_ISOLATION=CS_OPT_LEVEL3
CS_OPT_CHAINXACTS=CS_TRUE
CS_OPT_CLOSEONXACT=CS_TRUE
CS_OPT_QUOTED_IDENT=CS_TRUE
;
;The following is a sample section showing how to alter standard
;configuration:
;
[RELEVANT_SECTION_NAME]
;
;Use most of the ANSI properties defined above,
;
include=ANSI_ESQL

;but override some default properties

CS_OPT_ANSINULL=CS_TRUE ; enable non-ansi style null comparisons
CS_OPT_CHAINXACTS=CS_FALSE ; run in autocommit mode
The Embedded SQL precompiler generates the informational, warning, and error messages shown in this appendix’s tables.

Understanding the codes in the tables

Use this key for decoding the “Severity” column in Tables A-1 through A-9:

- Information – no error or warning was detected, and the precompiler succeeded. The message is purely informational.
- Warning – a noncritical error was detected, but the program precompiled.
- Severe – an error occurred, and no code was generated. The precompilation failed.
- Fatal – a severe error occurred from which the precompiler cannot recover. No further attempt will be made to process your files. Precompiler exits.

Table A-1: Command line option messages

<table>
<thead>
<tr>
<th>Message ID</th>
<th>Message text</th>
<th>Severity</th>
<th>Fix</th>
</tr>
</thead>
<tbody>
<tr>
<td>M_COMPAT_INFO</td>
<td>Compatibility mode specified.</td>
<td>Information</td>
<td>No fix required.</td>
</tr>
<tr>
<td>M_DUPOPT</td>
<td>Duplicate command line option specified.</td>
<td>Severe</td>
<td>Do not duplicate the options specified on the command line remove the offending duplicate option.</td>
</tr>
</tbody>
</table>
### Understanding the codes in the tables

<table>
<thead>
<tr>
<th>Message ID</th>
<th>Message text</th>
<th>Severity</th>
<th>Fix</th>
</tr>
</thead>
<tbody>
<tr>
<td>M_EXCFG_OVERRIDE</td>
<td>The switch value will have no effect because the external switch value has been specified.</td>
<td>Warning</td>
<td>When you use an external configuration file, you may override configuration options set on the command line. Choose one means of setting options.</td>
</tr>
<tr>
<td>M_INVALID_COMPAT</td>
<td>Unrecognized compatibility mode specified.</td>
<td>Information</td>
<td>No fix required.</td>
</tr>
<tr>
<td>M_INVALID_FILE_FMT</td>
<td>Invalid character in file value at line value.</td>
<td>Severe</td>
<td>Check to be sure that characters in the input file are valid and that you have correctly set the character set you want to use.</td>
</tr>
<tr>
<td>M_INVALID_FIPLEVEL</td>
<td>Invalid FIPS level specified.</td>
<td>Severe</td>
<td>Valid values are SQL-92E and SQL-89.</td>
</tr>
<tr>
<td>M_INVALID_SYNLEVEL</td>
<td>Invalid syntax checking level specified.</td>
<td>Severe</td>
<td>Valid values are NONE, SYNTAX, SEMANTIC.</td>
</tr>
<tr>
<td>M_INVLD_HLANG</td>
<td>Host Language specified is invalid.</td>
<td>Severe</td>
<td>Valid options are COB_MF1, COB_MF2, COB_RM1, COB_RM2, COB_LPI, COB_VAXVMS.</td>
</tr>
<tr>
<td>M_INVLD_OCLIB_VER</td>
<td>The Open Client Client-Library version is invalid.</td>
<td>Severe</td>
<td>The correct version string is ‘CS_VERSION_110’ or later.</td>
</tr>
<tr>
<td>M_INVOPT</td>
<td>Option is invalid.</td>
<td>Severe</td>
<td>Invalid option specified. Substitute the correct value.</td>
</tr>
<tr>
<td>M_LABEL_SYNTAX</td>
<td>Security label is improperly specified; the proper format is ‘labelname=labelvalue’.</td>
<td>Severe</td>
<td>Use the allowed syntax.</td>
</tr>
<tr>
<td>M_MSGINIT_FAIL</td>
<td>Error initializing localized error messages.</td>
<td>Warning</td>
<td>Verify that the Sybase installation is complete and that there is a valid entry for the LANG variable in the locales.dat file.</td>
</tr>
<tr>
<td>M_MULTI_IN_USE_DEF_OUT</td>
<td>When precompiling multiple input files, you cannot specify output (Listing, SQL, or Language) file names.</td>
<td>Severe</td>
<td>Remove all -G, -L, and -O flags from the command line or precompile the files one at a time.</td>
</tr>
</tbody>
</table>
### Message ID | Message text | Severity | Fix
--- | --- | --- | ---
M_NO_INPUT_FILE | Error: No input file is specified to be precompiled. | Severe | Specify an input file for precompilation.  
*Note* This error may occur if you precede the input file name with a flag (such as -G, for generate stored procedures), which takes an optional argument. To fix, put another flag in front of the input file name. For example, replace `cpre -G file.pc` with `cpre -G -C compilername`.  

M_NO_PERSISTENT_COBOL | The option -p for persistent input host variables is not available. | Information | No fix required.  

M_OPEN_INCLUDE | Unable to open the specified include file `file`. | Severe | The specified file is either not in the path or is missing the required read permission. Specify the path with the -I flag and verify the read permission.  

M_OPEN_INPUT | Unable to open the specified input file `file`. | Severe | Check the validity of the path and file name specified. If the file name extension is not provided, the precompiler searches for the default extension.  

M_OPEN_ISQL | Unable to open the specified ISQL file `file`. | Severe | Check the validity of the isql file name (the file in which the stored procedures are written). Verify that you have the write permission in the directory where the file is being created.  

M_OPEN_LIST | Unable to open the specified listing file `file`. | Severe | Check the validity of the listing file name. Verify that you have write permission in the directory where the file is being created.
## Understanding the codes in the tables

<table>
<thead>
<tr>
<th>Message ID</th>
<th>Message text</th>
<th>Severity</th>
<th>Fix</th>
</tr>
</thead>
<tbody>
<tr>
<td>M_OPEN_TARGT</td>
<td>Unable to open the specified target file <code>file</code>.</td>
<td>Severe</td>
<td>Check the validity of the output file name. Verify that you have write permission in the directory where the file is being created.</td>
</tr>
<tr>
<td>M_OPT_MUST_BE_PROVIDED</td>
<td>Option <code>value</code> must be provided.</td>
<td>Severe</td>
<td>Provide a value for option.</td>
</tr>
<tr>
<td>M_OPT_REINIT</td>
<td>Warning: <code>value</code> switch initialized multiple times.</td>
<td>Warning</td>
<td>The specified switch has been initialized multiple times. The second and subsequent values are ignored.</td>
</tr>
<tr>
<td>M_PATH_OFIL</td>
<td>Error: Max allowed paths for &quot;INCLUDE&quot; files is 64 (OVERFLOWED).</td>
<td>Severe</td>
<td>The maximum allowed paths on the command line have been exceeded. Reduce the number of directories from which the INCLUDE files are fetched.</td>
</tr>
<tr>
<td>M_STATIC_HV_CNAME</td>
<td>Static cursor names cannot be host-variables: <code>line</code>.</td>
<td>Severe</td>
<td>Replace the host variable with a SQL identifier.</td>
</tr>
<tr>
<td>M_UNBALANCED_DQ</td>
<td>Unbalanced quotes in delimited identifier.</td>
<td>Severe</td>
<td>Balance the quote.</td>
</tr>
<tr>
<td>M_VMS_NO_PERSISTENT_COBOL</td>
<td>The persistent option is not available.</td>
<td>Information</td>
<td></td>
</tr>
</tbody>
</table>

### Table A-2: First pass parser messages

<table>
<thead>
<tr>
<th>Message ID</th>
<th>Message text</th>
<th>Severity</th>
<th>Fix</th>
</tr>
</thead>
<tbody>
<tr>
<td>M_64BIT_INT</td>
<td>Warning: 64 bit integer host variables are not supported. Line <code>value</code>.</td>
<td>Warning</td>
<td>Use some other host variable type (float, numeric, or 32-bit integer). If necessary, copy the value between the host variable and the 64-bit program variable.</td>
</tr>
<tr>
<td>M_BLOCK_ERROR</td>
<td>Non-matching block terminator in <code>value</code> at line: <code>value</code>.</td>
<td>Severe</td>
<td>Correct your program syntax.</td>
</tr>
<tr>
<td>M_COB_INC_SQLDA</td>
<td>Error: the INCLUDE SQLDA statement is not valid in ESQL/COBOL.</td>
<td>Severe</td>
<td>Remove the invalid statement. See “About SQLDAs” on page 85 and “Using SYBSETSQLDA” on page 87.</td>
</tr>
<tr>
<td>Message ID</td>
<td>Message text</td>
<td>Severity</td>
<td>Fix</td>
</tr>
<tr>
<td>--------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>----------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>M_CONST_FETCH</td>
<td>Error: Attempted fetch into CONST storage class variable value.</td>
<td>Severe</td>
<td>You cannot fetch into a constant type. To fetch the value, remove the constant qualifier in its declaration.</td>
</tr>
<tr>
<td>M_DUP_HV</td>
<td>Duplicate host variable in file at line line.</td>
<td>Severe</td>
<td>Another host variable with the same name is already declared in the same block. Verify that each variable within a given block has a unique name.</td>
</tr>
<tr>
<td>M_DUP_STRUNION</td>
<td>Duplicate structure/union in file at line.</td>
<td>Severe</td>
<td>Another structure with the same name is already declared in the same block. Verify that each variable within a given block has a unique name.</td>
</tr>
<tr>
<td>M_IDENT_OR_STRINGVAR</td>
<td>Error: item must be a SQL-identifier or a string-type variable.</td>
<td>Severe</td>
<td>Verify that the connection, cursor, or statement name is of type string or SQL identifier.</td>
</tr>
<tr>
<td>M_IDENT_TOO_LONG</td>
<td>Error: Identifier value is too long (value bytes). Maximum size allowed is value bytes.</td>
<td>Severe</td>
<td>Verify that the identifier length is within the allowed limit. For SDK 15.0 and later, the maximum length of an identifier is 255 characters. For earlier versions, the maximum length is 132 characters.</td>
</tr>
<tr>
<td>M_ILL_LITERAL_USAGE</td>
<td>Error: Use of literal parameters to an RPC with an OUTPUT qualifier is not legal.</td>
<td>Severe</td>
<td>Do not use a literal as an OUTPUT parameter to a stored procedure.</td>
</tr>
<tr>
<td>M_ILL_PARAM_MODE</td>
<td>Error: Mixing calling modes in an rpc call in file at line.</td>
<td>Severe</td>
<td>Call the stored procedure with arguments passed by name or by position. Mixing these modes in the same call is illegal.</td>
</tr>
<tr>
<td>M_INDICVAR</td>
<td>Error: item must be an indicator-type variable.</td>
<td>Severe</td>
<td>Use a short integer.</td>
</tr>
<tr>
<td>M_INTVAR</td>
<td>Error: item must be an integer-type variable.</td>
<td>Severe</td>
<td>Use an integer.</td>
</tr>
<tr>
<td>M_INVLD_HV_BT</td>
<td>Cobol host variable: value of type: value is not supported.</td>
<td>Severe</td>
<td>Check the datatypes of the host variables. An unsupported type was detected.</td>
</tr>
</tbody>
</table>
### Understanding the codes in the tables

<table>
<thead>
<tr>
<th>Message ID</th>
<th>Message text</th>
<th>Severity</th>
<th>Fix</th>
</tr>
</thead>
<tbody>
<tr>
<td>M_MISMATCHED_QUOTES</td>
<td>Error: mismatched quotes on hex literal value.</td>
<td>Severe</td>
<td>Make quotes match.</td>
</tr>
<tr>
<td>M_MULTIDIM_ARRAY</td>
<td>Error: at line. Multiple-dimensional array variables are not supported.</td>
<td>Severe</td>
<td>Multiple-dimensional arrays are not supported. Break up an $m \times n$ array into $m$ arrays of $n$ elements each.</td>
</tr>
<tr>
<td>M_MULTI_RESULTS</td>
<td>Error: Embedded Query at line <em>line</em> returns multiple result sets.</td>
<td>Severe</td>
<td>Break the query into multiple queries, each returning one result set. Alternatively, rewrite the queries to fill a temporary table with all the values, then select from the temporary table, thus giving a single result set.</td>
</tr>
<tr>
<td>M_NODCL_NONANSI</td>
<td>Warning: Neither SQLCODE nor SQLCA declared in non-ANSI mode.</td>
<td>Warning</td>
<td>In non-ANSI mode, declare either SQLCA, SQLCODE, or both. Verify that the scope is applicable for all Embedded SQL statements within the program.</td>
</tr>
<tr>
<td>M_NOLITERAL</td>
<td>Error: Item may not be an unquoted name.</td>
<td>Severe</td>
<td>Use a quoted name or host variable.</td>
</tr>
<tr>
<td>M_NOSQUOTE</td>
<td>Error: Item may not be a single quoted string. Use double quotes.</td>
<td>Severe</td>
<td>Use double quotes.</td>
</tr>
<tr>
<td>M_NOT_AT_ABLE</td>
<td>An “at” clause is used with a statement type which does not allow it.</td>
<td>Severe</td>
<td>Remove the at clause from the specified statement.</td>
</tr>
<tr>
<td>M_NUMBER_OR_INDICVAR</td>
<td>Error: Item must be an integer or an indicator-type variable.</td>
<td>Severe</td>
<td>Use a literal integer or a short integer or CS_SMALLINT.</td>
</tr>
<tr>
<td>M_NUMBER_OR_INTVAR</td>
<td>Error: Item must be an integer constant or an integer type variable.</td>
<td>Severe</td>
<td>Unused. May be used to raise an error if some field in the dynamic SQL statements (such as MAX, Value $n$) is not an integer type or an integer constant.</td>
</tr>
<tr>
<td>M_PARAM_RESULTS</td>
<td>Error: Embedded Query at line <em>line</em> returns unexpected parameter result sets.</td>
<td>Severe</td>
<td>Arises only during optional server syntax checking. Determine why the query is returning parameters, and rewrite it.</td>
</tr>
<tr>
<td>Message ID</td>
<td>Message text</td>
<td>Severity</td>
<td>Fix</td>
</tr>
<tr>
<td>--------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>-----------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>M_PTR_IN_DEC_SEC</td>
<td>Warning: Pointers are not yet supported in Declare section.</td>
<td>Warning</td>
<td>None.</td>
</tr>
<tr>
<td>M_QSTRING_OR_STRINGVAR</td>
<td>Error: Item must be a quoted string or a type string variable.</td>
<td>Severe</td>
<td>Verify that server name, user name, and password are either double-quoted strings or of type string.</td>
</tr>
<tr>
<td>M_SCALAR_CHAR</td>
<td>Error: Non-array character variable value is being used illegally as a host variable at line line.</td>
<td>Severe</td>
<td>Use a character array.</td>
</tr>
<tr>
<td>M_SQLCA.IGNORED</td>
<td>Warning: Both SQLCODE and SQLCA declared: SQLCA ignored.</td>
<td>Warning</td>
<td>Remove one of the two declarations.</td>
</tr>
<tr>
<td>M_SQLCA_WARN</td>
<td>Warning: An INCLUDE SQLCA seen while in ANSI mode: SQLCA ignored.</td>
<td>Warning</td>
<td>None.</td>
</tr>
<tr>
<td>M_SQLCODE_UNDECL</td>
<td>Warning: SQLCODE not declared while in ANSI mode.</td>
<td>Warning</td>
<td>Declare SQLCODE.</td>
</tr>
<tr>
<td>M_STATE_CODE</td>
<td>Warning: Both SQLSTATE and SQLCODE declared: SQLCODE ignored.</td>
<td>Warning</td>
<td>Remove one of the two declarations.</td>
</tr>
<tr>
<td>M_STATE_SQLCA</td>
<td>Warning: Both SQLSTATE and SQLCODE declared: SQLCA ignored.</td>
<td>Warning</td>
<td>Remove one of the two declarations.</td>
</tr>
<tr>
<td>M_STATUS_RESULTS</td>
<td>Error: Embedded Query at line line returns unexpected status result sets.</td>
<td>Severe</td>
<td>Arises only during optional server syntax checking. Determine why the query is returning status results and rewrite it.</td>
</tr>
<tr>
<td>M_STICKY_AUTOVAR</td>
<td>Warning: Automatic variable value used with sticky binds at line line. This may cause incorrect results or errors at runtime.</td>
<td>Warning</td>
<td>Be certain that your program logic will not allow errors in this case. Alternatively, use a static or global variable.</td>
</tr>
<tr>
<td>M_STICKY_REGVAR</td>
<td>Error: Register variable value cannot be used with sticky binds at line line.</td>
<td>Severe</td>
<td>Remove the register qualifier.</td>
</tr>
</tbody>
</table>
**Understanding the codes in the tables**

<table>
<thead>
<tr>
<th>Message ID</th>
<th>Message text</th>
<th>Severity</th>
<th>Fix</th>
</tr>
</thead>
<tbody>
<tr>
<td>M_STRUCT_NOTFOUND</td>
<td>Structure/union definition not found in scope in file at line.</td>
<td>Severe</td>
<td>Verify that the definition of the structure or union is within the scope of the specified line.</td>
</tr>
<tr>
<td>M_SYNTAX_PARSE</td>
<td>Syntax error in file file at line.</td>
<td>Severe</td>
<td>Check the indicated line number for a syntax error in the Embedded SQL grammar.</td>
</tr>
<tr>
<td>M_UNBALANCED_DQ</td>
<td>Unbalanced quotes in delimited identifier.</td>
<td>Severe</td>
<td>Balance the quotes.</td>
</tr>
<tr>
<td>M_UNDEF_ELM</td>
<td>Error value: illegal structure/union element.</td>
<td>Severe</td>
<td>The specified element of the structure is not included in the structure definition. Correct the definition.</td>
</tr>
<tr>
<td>M_UNDEF_HV</td>
<td>Host variable value undefined.</td>
<td>Severe</td>
<td>Define the host variable in the proper place.</td>
</tr>
<tr>
<td>M_UNDEF_IV</td>
<td>Indicator variable value undefined.</td>
<td>Severe</td>
<td>Define the indicator variable in the proper place.</td>
</tr>
<tr>
<td>M_UNDEF_STR</td>
<td>Error structure value undefined.</td>
<td>Severe</td>
<td>Undefined structure on the specified line. Define the structure in the proper scope.</td>
</tr>
<tr>
<td>M_UNSUP</td>
<td>The value, feature is not supported in this version.</td>
<td>Fatal</td>
<td>This feature is not supported.</td>
</tr>
</tbody>
</table>

**Table A-3: Second pass parser messages**

<table>
<thead>
<tr>
<th>Message ID</th>
<th>Message text</th>
<th>Severity</th>
<th>Fix</th>
</tr>
</thead>
<tbody>
<tr>
<td>M_CURSOR_RD</td>
<td>The cursor value is redefined at line in file.</td>
<td>Warning</td>
<td>A cursor with same name has already been declared. Use a different name.</td>
</tr>
<tr>
<td>M_HOSTVAR_MULTIBIND</td>
<td>Warning: host variable used as a bind variable value more than once per statement.</td>
<td>Warning</td>
<td>Do not use a host variable multiple times in a single fetch statement. You cannot fetch multiple results into one location. Client-Library causes the last value fetched to be put in the variable.</td>
</tr>
<tr>
<td>M_INVTYPE_IV</td>
<td>Indicator variable is an incorrect type.</td>
<td>Severe</td>
<td>The indicator variable should be of type CS_SMALLINT or of type INDICATOR.</td>
</tr>
</tbody>
</table>
## APPENDIX A Precompiler Warning and Error Messages

<table>
<thead>
<tr>
<th>Message ID</th>
<th>Message text</th>
<th>Severity</th>
<th>Fix</th>
</tr>
</thead>
<tbody>
<tr>
<td>M_INVTYPE_V</td>
<td>Incorrect type of indicator variable found in structure value.</td>
<td>Fatal</td>
<td>All indicator variables in a structure must be of type CS_SMALLINT or INDICATOR.</td>
</tr>
<tr>
<td>M_INVTYPE_VI</td>
<td>Mismatch between number of structure elements in the indicator structure value and hostvar structure value.</td>
<td>Fatal</td>
<td>The number of elements in an indicator structure must be the same as the number of elements in the hostvar structure.</td>
</tr>
<tr>
<td>M_INVTYPE_VII</td>
<td>Mismatch between number of elements in the indicator array value and hostvar structure value.</td>
<td>Fatal</td>
<td>The number of elements in an indicator array must be the same as the number of elements in the hostvar structure.</td>
</tr>
<tr>
<td>M_PARSE_INTERNAL</td>
<td>Internal parser error at line line. Please contact a Sybase representative.</td>
<td>Fatal</td>
<td>Immediately report this internal consistency parser error to Sybase Technical Support.</td>
</tr>
<tr>
<td>M_SQLCANF</td>
<td>‘INCLUDE SQLCA’ statement not found.</td>
<td>Warning</td>
<td>Add the statement.</td>
</tr>
<tr>
<td>M_TAB_IN_LIT</td>
<td>Warning: TAB character in quoted string converted to space. (This warning will only appear once.)</td>
<td>Warning</td>
<td>If this is a problem, manually expand quoted &lt;tabs&gt; to spaces in your queries.</td>
</tr>
<tr>
<td>M_WHEN_ERROR</td>
<td>Unable to find the SQL statement ‘WHENEVER SQLERROR’.</td>
<td>Warning</td>
<td>Add WHENEVER SQLERROR statement, or use the command line option to suppress warning and INTO messages (see the Open Client and Open Server Programmers Supplement).</td>
</tr>
<tr>
<td>M_WHEN_NF</td>
<td>Unable to find the SQL statement &quot;WHENEVER NOT FOUND&quot;.</td>
<td>Warning</td>
<td>Enter a WHENEVER NOT FOUND statement, or use the command line option to suppress warning and INTO messages (see the Open Client and Open Server Programmers Supplement).</td>
</tr>
</tbody>
</table>
Understanding the codes in the tables

<table>
<thead>
<tr>
<th>Message ID</th>
<th>Message text</th>
<th>Severity</th>
<th>Fix</th>
</tr>
</thead>
<tbody>
<tr>
<td>M_WHEN_WARN</td>
<td>Unable to find the SQL statement &quot;WHENEVER NOT FOUND&quot;.</td>
<td>Warning</td>
<td>Enter a WHENEVER WARNING statement, or use the command line option to suppress warning and INTO messages (see the Open Client and Open Server Programmers Supplement).</td>
</tr>
<tr>
<td>M_INCLUDE_PATHLEN</td>
<td>An included or copied file path was too long. Leaving the path off the generated file name: value.</td>
<td>Warning</td>
<td>Use links or move the file to a shorter path.</td>
</tr>
<tr>
<td>M_WRITE_ISQL</td>
<td>Unable to write to the isql file. Return code: value.</td>
<td>Fatal</td>
<td>Verify your permission to create and write to the isql file and in the directory. Also, verify that the file system is not full.</td>
</tr>
<tr>
<td>M_WRITE_TARGT</td>
<td>Unable to write to the target file. Return code: value.</td>
<td>Fatal</td>
<td>Unable to write to the target file. Verify your permission to create and write to a file in the directory where the precompiler is generating the target file. Also, verify that the file system is not full.</td>
</tr>
</tbody>
</table>

**Table A-4: Code generation messages**

<table>
<thead>
<tr>
<th>Message ID</th>
<th>Message text</th>
<th>Severity</th>
<th>Fix</th>
</tr>
</thead>
<tbody>
<tr>
<td>M_INCLUDE_PATHLEN</td>
<td>An included or copied file path was too long. Leaving the path off the generated file name: value.</td>
<td>Warning</td>
<td>Use links or move the file to a shorter path.</td>
</tr>
<tr>
<td>M_WRITE_ISQL</td>
<td>Unable to write to the isql file. Return code: value.</td>
<td>Fatal</td>
<td>Verify your permission to create and write to the isql file and in the directory. Also, verify that the file system is not full.</td>
</tr>
<tr>
<td>M_WRITE_TARGT</td>
<td>Unable to write to the target file. Return code: value.</td>
<td>Fatal</td>
<td>Unable to write to the target file. Verify your permission to create and write to a file in the directory where the precompiler is generating the target file. Also, verify that the file system is not full.</td>
</tr>
</tbody>
</table>

**Table A-5: FIPS flag messages**

<table>
<thead>
<tr>
<th>Message ID</th>
<th>Message text</th>
<th>Severity</th>
<th>ANSI extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>M_FIPS_ARRAY</td>
<td>FIPS-flagger Warning: ANSI extension ARRAY type at line.</td>
<td>Information</td>
<td>Arrays. As for all FIPS messages, do not use this feature if you need to be ANSI-compliant.</td>
</tr>
<tr>
<td>M_FIPS_DATAINIT</td>
<td>FIPS-flagger Warning: ANSI extension Data Initialization at line.</td>
<td>Information</td>
<td>Data initialization.</td>
</tr>
<tr>
<td>M_FIPS_GPITEM</td>
<td>FIPS-Flagger Warning: ANSI extension group item syntax. (line line).</td>
<td>Information</td>
<td></td>
</tr>
</tbody>
</table>
### Table A-6: Internal error messages

<table>
<thead>
<tr>
<th>Message ID</th>
<th>Message text</th>
<th>Severity</th>
<th>Fix</th>
</tr>
</thead>
<tbody>
<tr>
<td>M_FIPS_SQLDA</td>
<td>FIPS-flagger Warning: ANSI extension sqlda. (line line)</td>
<td>Information</td>
<td>The SQLDA structure.</td>
</tr>
<tr>
<td>M_FIPS_STMT</td>
<td>FIPS-flagger Warning: ANSI extension statement (line line)</td>
<td>Information</td>
<td>The statement at this line is an extension.</td>
</tr>
<tr>
<td>M_FIPS_SYBTYPE</td>
<td>FIPS-flagger Warning: ANSI extension Sybase SQL-Type line.</td>
<td>Information</td>
<td>Sybase-specific datatypes.</td>
</tr>
</tbody>
</table>

### Table A-7: Platform and language messages

<table>
<thead>
<tr>
<th>Message ID</th>
<th>Message text</th>
<th>Severity</th>
<th>Fix</th>
</tr>
</thead>
<tbody>
<tr>
<td>M_ALC_MEMORY</td>
<td>Unable to allocate a block of memory.</td>
<td>Fatal</td>
<td>Check system resources.</td>
</tr>
<tr>
<td>M_FILE_STACK_OVFL</td>
<td>File stack overflow: Max allowed nesting is value.</td>
<td>Fatal</td>
<td>The file stack overflowed while trying to process the nested INCLUDE statement. Do not exceed the nested depth maximum of 32.</td>
</tr>
<tr>
<td>M_INTERNAL_ERROR</td>
<td>Fatal Internal Error at file line line: Argument inconsistency error. Please contact Sybase representative.</td>
<td>Fatal</td>
<td>This is an internal error. Contact your Sybase representative.</td>
</tr>
</tbody>
</table>

### Table A-8: Sybase and Client-Library messages

<table>
<thead>
<tr>
<th>Message ID</th>
<th>Message text</th>
<th>Severity</th>
<th>Fix</th>
</tr>
</thead>
<tbody>
<tr>
<td>M_COLMCNT</td>
<td>The bind count of the bind variable count and the column count of result set are incompatible.</td>
<td>Warning</td>
<td>The number of returned columns is different from the number of results columns returned with the bind variable types and number.</td>
</tr>
</tbody>
</table>
### Understanding the codes in the tables

<table>
<thead>
<tr>
<th>Message ID</th>
<th>Message text</th>
<th>Severity</th>
<th>Fix</th>
</tr>
</thead>
<tbody>
<tr>
<td>M_COLVARLM</td>
<td>The host variable name length value is less than the column length of value.</td>
<td>Warning</td>
<td>The host variable may not be able to hold the fetched column. Check the column length and adjust the length of the host variable accordingly.</td>
</tr>
<tr>
<td>M_COLVARPS</td>
<td>The host variable name precision and scale: value are different from the column's precision value and scale: value.</td>
<td>Warning</td>
<td>The precision and scale of the host variable is different from that of the column being fetched or inserted into. Make the scale and precision compatible.</td>
</tr>
<tr>
<td>M_COLVARTM</td>
<td>Open Client unable to convert type value to type value for host variable name.</td>
<td>Warning</td>
<td>Illegal type. Use cs_convert, as Open Client cannot convert by default.</td>
</tr>
<tr>
<td>M_CTMSG</td>
<td>Client Library message: value.</td>
<td>Information</td>
<td>None. If needed, contact Sybase Technical Support for assistance.</td>
</tr>
<tr>
<td>M_OCAPI</td>
<td>Error during execution of the Open Client API value. Error: value.</td>
<td>Warning</td>
<td>Depending on the context in which this warning occurs, you may be required to take corrective action before proceeding.</td>
</tr>
<tr>
<td>M_OPERSYS</td>
<td>Operating system error: value occurred during execution of the Open Client API.</td>
<td>Warning</td>
<td>An operating system error occurred. See the systems administrator.</td>
</tr>
<tr>
<td>M_PRECLINE</td>
<td>Warning(s) during check of query on line value.</td>
<td>Information</td>
<td>Examine the query for problems.</td>
</tr>
<tr>
<td>M_SYBSERV</td>
<td>Sybase Server error. Server: value. Message: name.</td>
<td>Warning</td>
<td>Check the syntax of the statement sent to the Server which caused this error. Verify that all resources are available in Adaptive Server Enterprise to process the SQL statement.</td>
</tr>
</tbody>
</table>
### Table A-9: Runtime messages

<table>
<thead>
<tr>
<th>SQLSTATE Code</th>
<th>Message text</th>
<th>Severity</th>
<th>Fix</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZZ000</td>
<td>Unrecoverable error occurred.</td>
<td>Fatal</td>
<td>Immediately report this error to Sybase Technical Support.</td>
</tr>
<tr>
<td>ZA000</td>
<td>Internal error occurred.</td>
<td>Fatal</td>
<td>Immediately report this error to Sybase Technical Support.</td>
</tr>
<tr>
<td>ZD000</td>
<td>Unexpected CS_COMPUTE_RESULT received.</td>
<td>Severe</td>
<td>Embedded SQL cannot retrieve compute results. Rewrite the query so it does not return them.</td>
</tr>
<tr>
<td>ZE000</td>
<td>Unexpected CS_CURSOR_RESULT received.</td>
<td>Severe</td>
<td>Verify that the value returned by the CS_LIBRARY routine is valid. Consult your CS-Library documentation for details.</td>
</tr>
<tr>
<td>ZF000</td>
<td>Unexpected CS_PARAM_RESULT received.</td>
<td>Severe</td>
<td>Verify that the value returned by the CS_LIBRARY routine is valid. Consult your CS-Library documentation for details.</td>
</tr>
<tr>
<td>ZG000</td>
<td>Unexpected CS_ROW_RESULT received.</td>
<td>Severe</td>
<td>Verify that the value returned by the CS_LIBRARY routine is valid. Consult your CS-Library documentation for details.</td>
</tr>
<tr>
<td>ZB000</td>
<td>No message(s) returned for SQLCA, SQLCODE, or SQLSTATE.</td>
<td>Information</td>
<td>Informational message. No action is required.</td>
</tr>
<tr>
<td>ZC000</td>
<td>Connection has not been defined yet.</td>
<td>Severe</td>
<td>Enter a valid connect statement.</td>
</tr>
</tbody>
</table>
### Understanding the codes in the tables

<table>
<thead>
<tr>
<th>SQLSTATE Code</th>
<th>Message text</th>
<th>Severity</th>
<th>Fix</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZH000</td>
<td>Unexpected CS_STATUS_RESULT received.</td>
<td>Severe</td>
<td>Verify that the value returned by the CS_LIBRARY routine is valid. Consult your CS-Library documentation for details.</td>
</tr>
<tr>
<td>ZI000</td>
<td>Unexpected CS_DESCRIBE_RESULT received.</td>
<td>Severe</td>
<td>Verify that the value returned by the CS_LIBRARY routine is valid. Consult your CS-Library documentation for details.</td>
</tr>
<tr>
<td>22005</td>
<td>Data exception—error in assignment of item descriptor type.</td>
<td>Severe</td>
<td>Enter a valid descriptor type.</td>
</tr>
<tr>
<td>ZJ000</td>
<td>Memory allocation failure.</td>
<td>Severe</td>
<td>There is an insufficient amount of memory to allocate to this operation.</td>
</tr>
<tr>
<td>ZK000</td>
<td>Adaptive Server Enterprise must be version 10 or later.</td>
<td>Severe</td>
<td>Verify that your installation has an installed, functioning copy of Adaptive Server Enterprise 10.0 or later. If you do not have Adaptive Server Enterprise 10.0 or later, have your installation’s designated person contact Sybase Technical Support.</td>
</tr>
<tr>
<td>ZM000</td>
<td>Error initializing Client Library.</td>
<td>Severe</td>
<td>Check your $SYBASE set-up.</td>
</tr>
<tr>
<td>ZN000</td>
<td>Error taking a mutex.</td>
<td>Severe</td>
<td>Unused.</td>
</tr>
</tbody>
</table>
APPENDIX A  Precompiler Warning and Error Messages

<table>
<thead>
<tr>
<th>SQLSTATE Code</th>
<th>Message text</th>
<th>Severity</th>
<th>Fix</th>
</tr>
</thead>
<tbody>
<tr>
<td>08002</td>
<td>Connection name in use.</td>
<td>Severe</td>
<td>Check your program logic: Are you re-opening an open connection? Or use a new name for the second connection.</td>
</tr>
<tr>
<td></td>
<td>Note</td>
<td></td>
<td>You cannot have two DEFAULT connections.</td>
</tr>
<tr>
<td>ZO000</td>
<td>HA FAILOVER has occurred.</td>
<td>Information</td>
<td>No action required.</td>
</tr>
</tbody>
</table>
Understanding the codes in the tables
Glossary

Adaptive Server Enterprise  A server in Sybase's client/server architecture. Adaptive Server Enterprise manages multiple databases and multiple users, keeps track of the actual location of data on disks, maintains mapping of logical data description to physical data storage, and maintains data and procedure caches in memory.

array  A structure composed of multiple identical variables that can be individually addressed.

array binding  The process of binding a result column to an array variable. At fetch time, multiple rows’ worth of the column are copied into the variable.

batch  A group of commands or statements:

A Client-Library command batch is one or more Client-Library commands terminated by an application’s call to ct_send. For example, an application can batch together commands to declare, set rows for, and open a cursor.

A Transact-SQL statement batch is one or more Transact-SQL statements submitted to Adaptive Server Enterprise by means of a single Client-Library command or Embedded SQL statement.

browse mode  A method that DB-Library and Client-Library applications can use to browse through database rows, updating their values one row at a time. Cursors provide similar functionality and are generally more portable and flexible.

bulk copy  A utility for copying data in and out of databases. Also called bcp.

callback event  In Open Client and Open Server, an occurrence that triggers a callback routine.

callback routine  A routine that Open Client or Open Server calls in response to a triggering event, known as a callback event.

capabilities  Determine the types of client requests and server responses permitted for a client/server connection.
**character set**
A set of specific (usually standardized) characters with an encoding scheme that uniquely defines each character. ASCII and ISO 8859-1 (Latin 1) are two common character sets.

**character set conversion**
Changing the encoding scheme of a set of characters on the way into or out of a server. Conversion is used when a server and a client communicating with it use different character sets. For example, if Adaptive Server Enterprise uses ISO 8859-1 and a client uses Code Page 850, character set conversion must be turned on so that both server and client interpret the data passing back and forth in the same way.

**client**
In client/server systems, the part of the system that sends requests to servers and processes the results of those requests.

**Client-Library**
Part of Open Client, a collection of routines for use in writing client applications. Client-Library is a library designed to accommodate cursors and other advanced features in the Sybase product line.

**code set**
See character set.

**collating sequence**
See sort order.

**command**
In Client-Library, a server request initiated by an application’s call to ct_command, ct_dynamic, or ct_cursor and terminated by the application’s call to ct_send.

**command structure**
A hidden Client-Library structure (CS_COMMAND) that Client-Library applications use to send commands and process results.

**connection structure**
A hidden Client-Library structure (CS_CONNECTION) that defines a client/server connection within a context.

**context structure**
A CS-Library hidden structure (CS_CONTEXT) that defines an application “context,” or operating environment, within a Client-Library or Open Server application. The CS-Library routines cs_ctx_alloc and cs_ctx_drop allocate and drop a context structure, respectively.

**conversion**
See character set conversion.

**CS-Library**
Included with both the Open Client and Open Server products, a collection of utility routines that are useful to both Client-Library and Server-Library applications.

**current row**
With respect to cursors, the row to which a cursor points. A fetch against a cursor retrieves the current row.

**cursor**
A symbolic name that is associated with a SQL statement.
In Embedded SQL, a cursor is a data selector that passes multiple rows of data to the host program, one row at a time.

database
A set of related data tables and other database objects that are organized to serve a specific purpose.

datatype
A defining attribute that describes the values and operations that are legal for a variable.

DB-Library
Part of Open Client, a collection of routines for use in writing client applications.

deadlock
A situation that arises when two users, each having a lock on one piece of data, attempt to acquire a lock on the other’s piece of data. Adaptive Server Enterprise detects deadlocks and resolves them by killing one user’s process.

default
Describes the value, option, or behavior that Open Client/Server products use when none is explicitly specified.

default database
The database that a user gets by default when he or she logs in to a database server.

default language
1. The language that Open Client/Server products use when an application does no explicit localization. The default language is determined by the “default” entry in the locales file.
2. The language that Adaptive Server Enterprise uses for messages and prompts when a user has not explicitly chosen a language.

Dynamic SQL
Allows an Embedded SQL or Client-Library application to execute SQL statements containing variables whose values are determined at runtime.

error message
A message that an Open Client/Server product issues when it detects an error condition.

event
An occurrence that prompts an Open Server application to take certain actions. Client commands and certain commands within Open Server application code can trigger events. When an event occurs, Open Server calls either the appropriate event-handling routine in the application code or the appropriate default event handler.

event handler
In Open Server, a routine that processes an event. An Open Server application can use the default handlers Open Server provides or can install custom event handlers.
### Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exposed structure</td>
<td>A structure whose internals are exposed to Open Client/Server programmers. Open Client/Server programmers can declare, manipulate, and de-allocate exposed structures directly. The CS_DATAFMT structure is an example of an exposed structure.</td>
</tr>
<tr>
<td>extended transaction</td>
<td>In Embedded SQL, a transaction composed of multiple Embedded SQL statements.</td>
</tr>
<tr>
<td>FIPS</td>
<td>Federal Information Processing Standards. If FIPS flagging is enabled, Adaptive Server Enterprise or the Embedded SQL precompiler issue warnings when a non-standard extension to a SQL statement is encountered.</td>
</tr>
<tr>
<td>gateway</td>
<td>A gateway is an application that acts as an intermediary for clients and servers that cannot communicate directly. Acting as both client and server, a gateway application passes requests from a client to a server and returns results from the server to the client.</td>
</tr>
<tr>
<td>hidden structure</td>
<td>A hidden structure is a structure whose internals are hidden from Open Client/Server programmers. Open Client/Server programmers must use Open Client/Server routines to allocate, manipulate, and de-allocate hidden structures. The CS_CONTEXT structure is an example of a hidden structure.</td>
</tr>
<tr>
<td>host language</td>
<td>The programming language in which an application is written.</td>
</tr>
<tr>
<td>host program</td>
<td>In Embedded SQL, the host program is the application program that contains the Embedded SQL code.</td>
</tr>
<tr>
<td>host variable</td>
<td>In Embedded SQL, a variable that enables data transfer between Adaptive Server Enterprise and the application program. See also indicator variable, input variable, output variable, result variable, and status variable.</td>
</tr>
<tr>
<td>indicator variable</td>
<td>A variable whose value indicates special conditions about another variable’s value or about fetched data. When used with an Embedded SQL host variable, an indicator variable indicates when a database value is null.</td>
</tr>
<tr>
<td>input variable</td>
<td>A variable that is used to pass information to a routine, a stored procedure, or Adaptive Server Enterprise.</td>
</tr>
<tr>
<td>interfaces file</td>
<td>A file that maps server names to transport addresses. When a client application calls ct_connect or dbopen to connect to a server, Client-Library or DB-Library searches the interfaces file for the server’s address. Note that not all platforms use the interfaces file. On these platforms, an alternate mechanism directs clients to server addresses.</td>
</tr>
</tbody>
</table>
### Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>isql script file</strong></td>
<td>In Embedded SQL, one of the three files the precompiler can generate. An isql script file contains precompiler-generated stored procedures, which are written in Transact-SQL.</td>
</tr>
<tr>
<td><strong>key</strong></td>
<td>A subset of row data that uniquely identifies a row. Key data uniquely describes the current row in an open cursor.</td>
</tr>
<tr>
<td><strong>keyword</strong></td>
<td>A word or phrase that is reserved for exclusive use in Transact-SQL or Embedded SQL. Also called a reserved word.</td>
</tr>
<tr>
<td><strong>listing file</strong></td>
<td>In Embedded SQL, one of the three files the precompiler can generate. A listing file contains the input file’s source statements and informational, warning, and error messages.</td>
</tr>
<tr>
<td><strong>locales file</strong></td>
<td>A file that maps locale names to language/character set pairs. Open Client/Server products search the locales file when loading localization information.</td>
</tr>
<tr>
<td><strong>locale name</strong></td>
<td>A character string that represents a language/character set pair. Locale names are listed in the locales file. Sybase predefines some locale names, but a system administrator can define additional locale names and add them to the locales file.</td>
</tr>
<tr>
<td><strong>locale structure</strong></td>
<td>A CS-Library hidden structure (CS_LOCALE) that defines custom localization values for a Client-Library or Open Server application. An application can use a CS_LOCALE to define the language, character set, datepart ordering, and sort order it will use. The CS-Library routines cs_loc_alloc and cs_loc_drop allocate and drop a locale structure.</td>
</tr>
<tr>
<td><strong>localization</strong></td>
<td>The process of setting up an application to run in a particular national language environment. An application that is localized typically generates messages in a local language and character set and uses local date, time, and datetime formats.</td>
</tr>
<tr>
<td><strong>login name</strong></td>
<td>The name a user uses to log in to a server. An Adaptive Server Enterprise login name is valid if Adaptive Server Enterprise has an entry for that user in the system table syslogins.</td>
</tr>
<tr>
<td><strong>message number</strong></td>
<td>A number that uniquely identifies an error message.</td>
</tr>
<tr>
<td><strong>message queue</strong></td>
<td>In Open Server, a linked list of message pointers through which threads communicate. Threads can write messages into and read messages from the queue.</td>
</tr>
<tr>
<td><strong>multi-byte character set</strong></td>
<td>A character set that includes characters encoded using more than 1 byte. EUC JIS and Shift-JIS are examples of multibyte character sets.</td>
</tr>
</tbody>
</table>
**Glossary**

**mutex**
A mutual exclusion semaphore. This is a logical object that an Open Server application uses to ensure exclusive access to a shared object.

**null**
Having no explicitly assigned value. NULL is not equivalent to zero or to blank. A value of NULL is not considered to be greater than, less than, or equivalent to any other value, including another value of NULL.

**Open Server**
A Sybase product that provides tools and interfaces for creating custom servers.

**Open Server application**
A custom server constructed with Open Server.

**output variable**
In Embedded SQL, a variable that passes data from a stored procedure to an application program.

**parameter**
1. A variable that is used to pass data to and retrieve data from a routine.
2. An argument to a stored procedure.

**passthrough mode**
When in passthrough mode, a gateway relays Tabular Data Stream™ (TDS) packets between a client and a remote data source without unpacking the packets’ contents.

**property**
A named value stored in a structure. Context, connection, thread, and command structures have properties. A structure’s properties determine how it behaves.

**query**
1. A data retrieval request; usually a `select` statement.
2. Any SQL statement that manipulates data.

**registered procedure**
In Open Server, a collection of C statements stored under a name. Open Server-supplied registered procedures are called *system registered procedures*.

**remote procedure call**
1. One of two ways in which a client application can execute an Adaptive Server Enterprise stored procedure. (The other is with a Transact-SQL `execute` statement.) A Client-Library application initiates a remote procedure call command by calling `ct_command`. A DB-Library application initiates a remote procedure call command by calling `dbrpcinit`.

2. A type of request a client can make of an Open Server application. In response, Open Server either executes the corresponding registered procedure or calls the Open Server application’s RPC event handler.

3. A stored procedure executed on a different server from the server to which the user is connected.
result variable

In Embedded SQL, a variable which receives the results of a `select` or `fetch` statement.

server

In client/server systems, the part of the system that processes client requests and returns results to clients.

Server-Library

A collection of routines for use in writing Open Server applications.

sort order

Used to determine the order in which character data is sorted. Also called *collating sequence*.

SQLCA

1. In an Embedded SQL application, SQLCA is a structure that provides a communication path between Adaptive Server Enterprise and the application program. After executing each SQL statement, Adaptive Server Enterprise stores return codes in SQLCA.

2. In a Client-Library application, SQLCA is a structure that the application can use to retrieve Client-Library and server error and informational messages.

SQLCODE

1. In an Embedded SQL application, SQLCODE is a structure that provides a communication path between Adaptive Server Enterprise and the application program. After executing each SQL statement, Adaptive Server Enterprise stores return codes in SQLCODE. A SQLCODE can exist independently or as a variable within a SQLCA structure.

2. In a Client-Library application, SQLCODE is a structure that the application can use to retrieve Client-Library and server error and informational message codes.

statement

In Transact-SQL or Embedded SQL, an instruction that begins with a keyword. The keyword names the basic operation or command to be performed.

status variable

In Embedded SQL, a variable that receives the return status value of a stored procedure, thereby indicating the procedure’s success of failure.

stored procedure

In Adaptive Server Enterprise, a collection of SQL statements and optional control-of-flow statements stored under a name. Adaptive Server Enterprise-supplied stored procedures are called *system procedures*.

System Administrator

The user in charge of Adaptive Server Enterprise system administration, including creating user accounts, assigning permissions, and creating new databases. On Adaptive Server Enterprise, the System Administrator’s login name is “sa”.

system descriptor

In Embedded SQL, a system descriptor is an area of memory that holds a description of variables used in Dynamic SQL statements.
<table>
<thead>
<tr>
<th>Glossary</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>system procedures</strong></td>
<td>Stored procedures that Adaptive Server Enterprise supplies for use in system administration. These procedures are provided as shortcuts for retrieving information from system tables, or as mechanisms for accomplishing database administration and other tasks that involve updating system tables.</td>
</tr>
<tr>
<td><strong>system registered procedures</strong></td>
<td>Internal registered procedures that Open Server supplies for registered procedure notification and status monitoring.</td>
</tr>
<tr>
<td><strong>target file</strong></td>
<td>In Embedded SQL, one of three files the precompiler can generate. A target file is similar to the original input file, except that all SQL statements are converted to Client-Library function calls.</td>
</tr>
<tr>
<td><strong>TDS</strong></td>
<td>(Tabular Data Stream) An application-level protocol that Sybase clients and servers use to communicate. It describes commands and results.</td>
</tr>
<tr>
<td><strong>thread</strong></td>
<td>A path of execution through Open Server application and library code and the path’s associated stack space, state information, and event handlers.</td>
</tr>
<tr>
<td><strong>Transact-SQL</strong></td>
<td>An enhanced version of the database language SQL. Applications can use Transact-SQL to communicate with Adaptive Server Enterprise.</td>
</tr>
<tr>
<td><strong>transaction</strong></td>
<td>One or more server commands that are treated as a single unit for the purposes of backup and recovery. Commands within a transaction are committed as a group; that is, either all of them are committed or all of them are rolled back.</td>
</tr>
<tr>
<td><strong>transaction mode</strong></td>
<td>The manner in which Adaptive Server Enterprise manages transactions. Adaptive Server Enterprise supports two transaction modes: Transact-SQL mode (also called “unchained transactions”) and ANSI mode (also called “chained transactions”).</td>
</tr>
<tr>
<td><strong>user name</strong></td>
<td>See login name.</td>
</tr>
</tbody>
</table>
Index

Symbols
#  61
$  61
?  61
   and dynamic parameter markers  70
_  61

A
Adaptive Server
   connecting to  39
   multiple connections  41
allocate descriptor  104
allow ddl in tran  109
arrays  47
   batch  50
   double-dimensional  24
   Indicator  47
   multiple  31
   select into  47
   using  31
at connect_name
   named connection  113
at connection_name  42, 43
exec sql statement  142

B
batches
   get diagnostics  64
   restrictions  14
   statements  14
begin transaction  65, 66
binding  62, 69

C
close  109
close and cursors  59
close cursor  59
COBOL veneer layer  33
   and conversions  36
colons
   and host variables  25
   and indicator variables  28
command line options, precompiler  6
comments
   in Embedded SQL  12
commit  44
commit transaction  66, 110
commit work  66
compatibility  46
   backward  4
complex definition  24
compute clause
   disallowed  166
connect  39
   multiple connections  41
   using both COBOL and C  39
connections
   closing  114, 137
   default  113
   multiple  41
   named  113
   naming  42
conversion, datatype  3
converting datatypes  36
current row  51, 56
cursors  51, 57, 59, 119, 121, 123, 159, 160
   and scoping  51
   closing  59, 109
   declaring  52
   deleting current row  58
Index

deleting rows 127
dynamic 118, 162
dynamic 162
example 59
opening 55
position 56, 57
retrieving data 55
updating current row 58
updating rows 170

D
data declarations 24
Data Definition Language (DDL) 71
data definitions 67
data items
elementary and group 34
Data Manipulation Language (DML) 46, 71


databases
accessing 39
connecting to a Server 39
pubs2 3


selecting rows 165
datatype conversions 3
input variables 37
result variables 36
datatypes 33
COBOL and Adaptive Server 33, 35
converting 36


DDL (Data Definition Language) 71
ddl in tran 109
deallocate descriptor 116
deallocate prepare 118
declarations
data 24
declare cursor 52, 53, 64, 119, 121, 123
dynamic 119
static 121
stored procedure 123


declare scrollable cursor 125
declare sections 23
multiple 24
default server
connecting to 40
default transaction mode 65
delete 58


positioned cursor 126
searched 128
with cursors 58
describe input 130
describe output 133
descriptor area 71
directories
and searches 67
disconnect 44, 137
DML (Data Manipulation Language) 46, 71
documentation
online 60
double-dimensional array 24
DSQUERY environment variable 114
dynamic binding 69
dynamic parameter markers 70, 73, 145
dynamic SQL 69, 118, 145, 147, 162
method 1 71, 72
method 2 72, 75
method 3 75, 79
method 4 79, 85
prepare and execute 145, 164
prepare and fetch 164

E
elementary data items 34
Embedded SQL ix, 1, 2
constructs 14
definition 1
Embedded SQL statements
syntax-checking 101
environment variables 114
SYBASE 114
error
failure to detect example 102
testing 3
error handler
writing 100
error-handling
warning-handling routines 100
errors
SQLSTATE 22
testing for 96
trapping 97
ESQL/COBOL veneer layers 33
host variables
  using 25
examples 3
exec 139
exec sql 142
executable
  building 5
execute 143
execute immediate 71, 72, 145
  example 72
extended transaction 66
external configuration file 177

F
features and enhancements 2
fetch 55, 57, 147
  and host variables 26
  within a loop 56
fetch into 31
files
  directory 67
  isql 63
  listing 98
  multiple 6
  precompiler-generated 6

G
get descriptor 149
get diagnostics 64, 100, 153
  batches 64
group data items 34
group element referencing 6

H
handlers
  error and warning 100
host input variables 25
host output variables 27
host result variables 26
host status variables 26
host variables 2, 28, 30
  and datatypes 37
  assigning data to 55
  character string 30
  declaring 23, 24
  in fetch 56, 57
  naming 30
  using 25
  with indicator variables, using 27
host variables with indicator variables
  using 28

I
identifiers
  in Embedded SQL 13
  include 67, 156, 157
    filename 154
  include file directory 67
  include sqlca 156, 157
  indicator arrays 47
  indicator variables
    and colons 28
    declaring 23, 24
    using 27
    with host input variables 29, 30
    with host output and result variables 28
input variables
  converting datatypes for 37
  host 25
interactive SQL 63
interfaces file 114
  into 46, 62
invalid statements
  print 46
  raiserror 46
  readtext 46
  writetext 46
is global 31
isql file 6, 63
Index

K
keywords
  in Embedded SQL 13

L
label
  variable 39
labels 174
listing file 6
logical names 114

M
markers
  dynamic parameter 145, 160, 164
multiple arrays 31
multiple connections 41
multiple source files 6
multiple SQLCAs 18

N
named connections 113
nesting
  stored procedure 63
null
  input value 29
null password
  specifying 114

O
online sample programs 60
open 55, 159
dynamic cursor 158
scrollable cursor 162
static cursor 160
output 62
output file 63

P
parse 5, 13, 102
password 40
  null specifying 114
placement
  Embedded SQL statements 12
precompiler
  command line options 6
detected errors 101
diagnostics 101
functionality 5
prepare 73, 162
prepare and execute 73, 75, 145
ever example 75
prepare and fetch
ever example 78
procedure_name 61
product family x
program
  creating 4
pubs2 database 3

Q
question mark
  and dynamic parameter markers 70
quotation marks
  in Embedded SQL 3, 13

R
related documents x
reserved words
  in Embedded SQL 13
result variables
  converting datatypes for 36
  host 26
return code 17, 20
SQLCODE 20
testing 3
return values
  SQLCODE 96
rollback
  and Adaptive Server triggers 67
in a trigger 65
work 66
rollback transaction 164
routines
  error- and warning-handling 100
rows
  current 56
deleting 126
rules 102

S
sample programs
  online 60
scoping 13, 18, 31
  and cursors 51
cursor 51
rules 13
SQLCA, SQLCODE, and SQLSTATE 17
scroll fetch 149
scrollable cursors
  declaring 53
retrieving data 57
select 14, 31, 64, 165
  and host variables 26
cursors 119, 121, 123, 147
  returning multiple rows 51, 57
  returning single rows 47
syntax 47
set connection 41, 167
set descriptor 168
source files 67
  multiple 6
SQLCA
  and include 67
  declaring 18
  declaring multiple 18
SQLCA variables 18
  accessing 19
  Adaptive Server-related 19
  setting 17
SQLCODE 96, 97
  and multiple row selects 46
  and whenever 57
  as a standalone 20
fetch 149
return values 96
setting variables 17
values 21
within SQLCA 20
SQLSTATE
  codes and error messages 22
  setting variables 17
  using 21
SQLWARN 96
statement batches 14
statement labels
  whenever 174
statements
  dynamic SQL 81
  Embedded SQL 11, 12
status information 17
status variables
  host 26
  status_variable 61
stored procedures 2, 45, 61, 64
  and parameters 61
  and return status variables 61
  executing 61
  types of 61
SYBASE environment variable 114
syntax checking
  Embedded SQL statements 101
system variables 17, 23

T
  tables
    deleting rows 126
target file 6
transaction mode
  ANSI 66
default 65
Transact-SQL 65
transactions 65, 110
  ANSI 65
  extended 66
  ISO 65
  restricted statements 66
  rolling back 164
Index

Transact-SQL
  invalid keywords in Embedded SQL  2, 46
  keywords in Embedded SQL  13
  support  2
  using with Embedded SQL  45
Transact-SQL statements  126, 139, 165, 170
triggers  65, 102
truncation  36

canceling  174
scope  173
statement  97, 98
testing  97
whenever action
  call  99
  continue  99
goto  99
  perform  99, 100
whenever statement
  20
WORKING-STORAGE SECTION  18

update  58, 170
  protocol  59
    with cursors  58
user  40

variables
  declaring  23
  examples in declare section  24
  host  3, 23, 30
  host result  26
  host status  26
  indicator  23
  input  23
  input host  25
  picture, usage clauses  24
precompiler  13
system  17, 23
veneer layer  33
  and conversions  36
veneer layers
  static and shared dynamic  33

warning handler
  writing  100
warnings
  error-handling routines  100
  testing for  96, 97
whenever